COMPUTER FUNDAMENTALS & C PROGRAMMING

About the Author

Sumitabha Das obtained a degree in electronics engineering from Calcutta University (1977). After serving as Deputy Director in the Ministry of Defence, he was lured by the world of UNIX into prematurely leaving his service in 1989. In 1993, his book on UNIX hit the Indian market, and in 2001, yet another one reached the US shores. Das has worked extensively on Oracle PL/SQL programming and has also taught Systems Programming (in C) and Java in his brief stint at Bengal Engineering and Science University. In 2006, at the behest of Dr. Ambedkar Institute of Technology, he accomplished the task of teaching Perl and Systems Programming (in C) to around 40 lecturers and professors located in and around Bengaluru. He eventually decided to write a book on C programming, long after everyone else had done their bit in an over-saturated market.

COMPUTER FUNDAMENTALS & C PROGRAMMING

SUMITABHA DAS

McGraw Hill Education (India) Private Limited
CHENNAI

McGraw Hill Education Offices

Chennai New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

McGraw Hill Education (India) Private Limited
Published by McGraw Hill Education (India) Private Limited
444/1, Sri Ekambara Naicker Industrial Estate, Alapakkam, Porur, Chennai 600 116

Computer Fundamentals & C Programming

1 2 3 4 5 6 7 8 9 D103074 22 21 20 19 18

Printed and bound in India.

Print Book
ISBN (13): 978-93-87886-07-0
ISBN (10): 93-87886-07-7

E-Book
ISBN (13): 978-93-87886-08-7
ISBN (10): 93-87886-08-5

Director—Science & Engineering Portfolio: *Vibha Mahajan*
Senior Portfolio Manager—Science & Engineering: *Hemant K Jha*
Associate Portfolio Manager—Science & Engineering: *Mohammad Salman Khurshid*

Senior Manager—Content Development: *Shalini Jha*
Content Developer: *Ranjana Chaube*

Production Head: *Satinder S Baveja*
Copy Editor: *Taranpreet Kaur*
Assistant Manager—Production: *Anuj K Shriwastava*

General Manager—Production: *Rajender P Ghansela*
Manager—Production: *Reji Kumar*

Typeset at APS Compugraphics, 4G, PKT 2, Mayur Vihar Phase-III, Delhi 96, and printed at

Visit us at: www.mheducation.co.in

Write to us at: info.india@mheducation.com

CIN: U22200TN1970PTC111531

Toll Free Number: 1800 103 5875

To Dr. N Subrahmanyam who knew me long before I began to know myself.

Preface

A language is not worth knowing unless it teaches you to think differently.

—Larry Wall (the creator of Perl)

Enthused by the success of my previous books on one of Dennis Ritchie's two creations and undeterred by the fierce competition, I decided to focus on Ritchie's other creation—the C language. A major course correction in the midst of my venture led me to include content that addressed the needs of beginners who lacked knowledge of computers. The consequent delay was perhaps inevitable, but the much awaited book is eventually here.

UNIX and C were developed simultaneously at Bell Labs four decades ago. Ritchie created the C language with an unusual objective—to rewrite the code for the UNIX operating system. That did happen eventually, but C was too well designed to remain stuck there. Today, C is a general-purpose language whose application areas extend from operating systems and device drivers to the "apps" on the iPhone. The question arises that in spite of the subsequent entry of numerous languages like Java, Python and Ruby, how has C been able to strongly hold on to its forte?

The answer lies in the features that make it difficult to classify C purely as a high-level language. C understands that resources can be scarce at times, so why waste them? Why use one byte when one bit will serve the purpose? Using *all* the symbols available on the computer keyboard, C provides a vast arsenal of 40 operators that can perform amazing operations on data. C programs are compact, fast but also tricky—and often cryptic. This book is all about revealing these tricks and demystifying one feature that routinely confuses people—pointers.

How This Book is Different

With ample books on C already available in the market, the obvious question must be asked: Why another book now? The plain truth is that none of the current books has really worked for me. I believe that a text book must identify potential areas of confusion and anticipate the questions a reader is likely to come up with. As done previously, I put myself in the student's shoes, posed these questions to myself and answered them as well as I could. This book is primarily meant for self-study even though some guidance at times may be helpful.

I strongly disapprove of filling up a book with an endless stream of programs and tables simply for the "sake of completeness." Instead, this book focuses on the essential concepts supported by

well-annotated and properly indented programs. *Every program in this book has been adequately explained*. Many of these programs are based on real-life situations that may often seem familiar if not interesting. Wherever possible, a program has been progressively enhanced with the exposition of a new feature of the language.

The strong feature of the book is its pedagogical aids. Apart from the standard highlights (like *Note*, *Tip* and *Caution*), there is also a *Takeaway* feature which summarizes key information. Special asides entitled *How It Works* provide additional information for serious programmers. Rather than providing a list of terms at the end of each chapter, a detailed *Glossary* has been included for quick lookups. Extensive cross-referencing has been made available by embedding parenthesized section numbers in the text. I have no doubt that you'll often follow these references either for refreshing your memory or for knowing what lies ahead.

This book was originally meant to be a text on C programming, but changed circumstances demanded the inclusion of three separate chapters on computer fundamentals. This background should prove useful, but excessive coverage on computer basics (which often is the case) can dull your interest in C. So some basics have been provided that should more than fulfill the readers' needs. However, it is possible for the reader to jump straight to C by ignoring the first three chapters.

All programs in this book have been tested on the GCC compiler that is shipped with every Linux distribution. Many of these programs have also been compiled and executed successfully on Microsoft Visual Studio. However, these programs might not run error-free on every system. Bugs in these programs are possible though, and I welcome the ones that you may hit upon.

Acknowledgments

The prime mover for this project is Vibha Mahajan, who has been associated with my activities for the last two decades. Hemant K Jha graciously withstood the punches without losing a firm grip on the project. I appreciate the content-enhancing efforts put in by Shalini Jha and Ranjana Chaube, and Salman Khurshid who provided valuable inputs related to pedagogy. Finally, Satinder Singh and Anuj Shriwastava handled all production-related issues with indisputable competence. Thank you, everyone including other team members who have not been named here.

Dennis Ritchie created C because he felt that "it was a good thing to do." When you step into the commercial world and explore other languages, the foundation provided by C would stand you in good stead for years to come. Every other language—be it C++, Java or Python—will be well within your reach. It won't be long before you realize that learning C represented a major milestone in your life. It was more than just "a good thing to do."

SUMITABHA DAS

Contents in Brief

Contents

CHAPTER 4 A Taste of C 121

CHAPTER 5 Data—Variables and Constants 148

CHAPTER 15 File Handling 484

1 Introducing Computers and Hardware

WHAT TO LEARN

- Progressive evolution of the concepts that computers are based on.
- Classification of computers based on *type* and *generation*.
- Structure of the *Central Processing Unit (CPU)* and how it runs a program.
- Role of *memory* in and outside the CPU.
- Types of *ports* and connectors supported by a computer.
- The important input and output devices that are connected to these ports.
- The technology that connects multiple computers in a network.
- The devices used by a network and the Internet.

1.1 THE COMPUTER BOOM

The computer is probably man's greatest invention. From its humble beginnings several thousand years ago, this machine has evolved through numerous path-breaking changes to successfully intrude into all walks of our lives. Computers made it possible for man to visit the moon. Today, they process business data and control machines and equipment. Humans use computers to both kill and heal people. Millions of computers acting in tandem power the Internet. Think of the automobile, radio or television; nothing has impacted our lives as much as this machine has.

Computers are powerful devices capable of computing accurately at lightning speeds. By being *programmable*, they can process millions of data records with a *single* set of instructions. If the problem changes, only the instructions need to change and not the computer. This property has for many decades served as the basis for use of computers in processing employee and financial data. Computers are an old friend of the aviation industry; ticket reservation was done on computers even when they were giant and expensive power guzzlers.

Most of the game-changing advances in computing technologies have occurred in the last fifty years. One of them—network technology—has elevated computers from standalone devices to connected machines capable of communicating with one another. The Internet is only about thirty years old and one of its services—the World Wide Web—provided a versatile platform for the development of a plethora of services like Google, Facebook, Twitter and Amazon. Online shopping is now a serious threat to the brick-and-mortar stores. Also, data is no longer restricted to text; multimedia data transmission has now become the norm.

The development of the *microprocessor*, accompanied by rapid reduction in prices, enabled the computer to make inroads into practically every domain of human and industrial activity. The personal computer moved to the user's home. Computers are interfaced with electronic devices. Medical science has delegated many of its functions to computers. Miniaturization has enabled computers to be embedded in hand-held devices like smartphones. We look forward to seeing computers using artificial intelligence for controlling robots and driverless cars.

Developments in hardware design have also been accompanied by changes in the way we program computers. Computer programming has moved from direct handling of 0s and 1s to the development of languages that have progressively moved closer to the human language. Even though we are not there yet, it won't be long before computers are programmed by natural languages using text and voice recognition. In this book, we take up one of the important languages that still remain important and relevant—the C language.

1.2 COMPUTER BASICS

What then is a computer? Simply put, a computer is a device that can automatically perform a set of *instructions*. The computer takes as input these instructions as a *single* unit, uses them to manipulate data, and outputs the results in user-specified ways. The processing is fast, accurate and consistent, and is generally achieved without significant human intervention.

A computer owes its power to the *Central Processing Unit (CPU)* whose primary task is to process these instructions. An instruction can do only simple work, like moving a constant to a register (a chunk of memory in the CPU) or adding one to it. Instructions need *data* to operate on, and this data can be made available from multiple sources—keyboard input, for instance. Data may also occur in the program itself. The output could be written to any device—a disk, monitor or printer, to name a few.

A *program* containing instructions is typically held in a *file* that is normally stored on the computer's hard disk (a form of memory). To execute this program, the contents of its file must be loaded into *main memory* (another form of memory). The data also get loaded, either with the program or when demanded by it. The CPU executes a single instruction at a time (could be different on multi-CPU machines). The instructions are executed in a linear sequence unless an instruction itself specifies a different execution sequence. For instance, an instruction can specify a set of other instructions to be performed repeatedly.

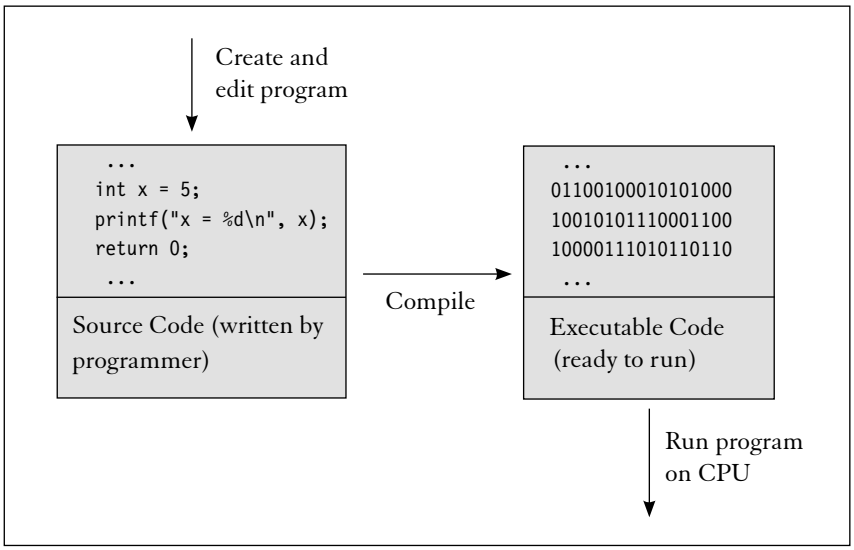


FIGURE 1.1 Program Transformation Before Execution

However, the programs we write are not the ones that are executed. Why? Computers, being binary devices, understand only *meaningful* sequences of 0s and 1s, commonly called *machine language*. Thus, all instructions and data must be input to the CPU in machine language (0s and 1s). Because humans don't feel comfortable using this language, we need to use a *programming language*, and the computer world offers a number of them (like C). These languages are mildly similar to spoken languages (like English). They support a very small set of *keywords* and symbols held together by a rigid and unforgiving grammar.

The CPU doesn't understand programming languages, so every program you write must be translated to its corresponding machine language code before execution. The written program is first saved in a file on the computer's disk. A separate program, called the *compiler*, translates or compiles this source code to its equivalent *machine code* in binary. As shown in Figure 1.1, it is this code that is eventually executed by the CPU.

The CPU needs adequate memory to run programs. It fetches program instructions and data from memory and writes temporary and final data to memory. Memory can be *volatile* (erased when power is withdrawn) or non-volatile, and the CPU uses the appropriate type depending on how long the data need to be preserved. Some memory is inside the CPU while some are outside it but inside the computer housing. For enabling programs to interact with the external world, computers also use external devices as memory (like the portable hard disk and flash memory).

Takeaway: The programs we write using a language like C have to be processed by a compiler program for generation of the corresponding machine code. This code, comprising 0s and 1s, eventually runs on the CPU.

1.3 THE BACKGROUND

It was in Babylonia around 2000 B.C. that the computer made its beginning as a simple calculator. The device, called the *abacus*, comprised a number of rings or beads that could be moved along rods enclosed by a wooden frame. This calculator could be used only for addition and subtraction, which were adequate for the age. Even though the device is still used in many countries of Eastern Europe and China, many other devices followed, each implementing a key computing concept that ultimately led to the creation of the digital computer.

1.3.1 Napier, Pascal and Leibniz: The Early Pioneers

The next major leap in computing occurred with the invention of the *logarithm* by John Napier in the early 17th century. Napier established that multiplication and division could be achieved by adding and subtracting, respectively, the logarithm of the numbers. Napier's invention soon led to the development of the *slide rule* (Fig. 1.2) which was extensively used by engineers and scientists right up to the 1960s. NASA scientists used slide rules to land men on the moon and bring them back to earth. This calculating device was eventually obsoleted by the electronic calculator even though it has not completely disappeared.

The 17th century also saw the emergence of the gear-based *digital* mechanical calculator. (The slide rule is actually an *analog* device.) Blaise Pascal was the first to create one but it could perform only addition and subtraction. The *stepped reckoner* developed by Gottfried Leibniz improved upon Pascal's creation by adding the features of multiplication and division. Both devices were based on the decimal system, but Leibniz recommended the use of the binary system, which was to be adopted centuries later.

1.3.2 The Programmable Computer

The concept of using a *program* to control a sequence of operations was first seen in the *Jacquard loom* in the early 19th century. This textile loom used a set of punched cards to weave complex patterns in the cloth. To weave a different design, one simply had to change the card set. This idea appealed to Charles Babbage who used the concept in his two conceptual creations—the *Difference engine* and *Analytical engine*. However, while the holes in the cards of the Jacquard loom determined whether a thread would pass through it, Babbage saw an opportunity to use them to encode instructions and data.

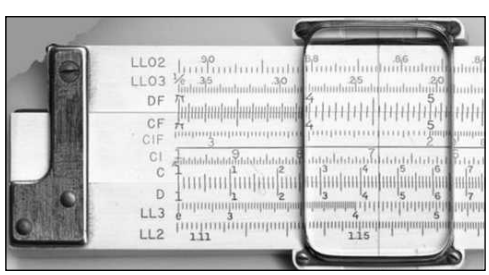


FIGURE 1.2 The Slide Rule

The analytical engine was fully automatic and designed to run on steam power. It had a processing unit and memory—two essential components of any programmable device. Apart from *loops* which allowed a set of statements to be repeated, the analytical engine featured a *conditional* statement which would permit disruption of the normal sequence of operations. Babbage also utilized the services of Ada Lovelace who designed a few *algorithms* for the engine. Even though neither engine was ever built (probably due to paucity of funds), Babbage is rightly considered to be the father of the computer while Lovelace is accepted as the first programmer.

When electric motors became available, Herman Hollerith used the technology in the late 19th century to design the *Hollerith desk*. This machine featured a card reader, writer and gear-driven counter. The machine could count and sort data and display the output on a set of dials. *Moreover, the machine could also punch output data on to the cards.* Hollerith employed the Hollerith Desk to complete the 1890 census in the USA and his Tabulating Machine Company, after consolidation, was renamed International Business Machines (IBM). In fact, IBM continued to make and sell these machines well into the 1960s before they were driven out by magnetic media.

1.3.3 The Programmable Electronic Computer

In the period leading up to World War II, most computers used electro-mechanical relays and switches to perform computation. The Z3 computer, designed by Konrad Zuse during the war, used over 2000 relays. The Z3 is considered to be the world's first programmable and automatic digital computer. It was also the first computer to be based on the binary system, and program control was achieved by punched film. During the same period, the Colossus and ENIAC computers used vacuum tubes to be the first all-electronic programmable computers. However, a program in the Colossus or ENIAC was represented by the state of its switches, so reprogramming the machine for a different job needed manual resetting of these switches.

Because of the difficulty of programming, speed was an issue even with electronic computers. This problem was addressed by Alan Turing, and later reiterated by John von Neumann, in the concept of the *stored-program computer*. In a paper prepared in 1936, Turing reasoned that computer speeds could be vastly improved if both program and data were stored in the same memory. A program fed through paper tape couldn't be fast enough because the computer could only load one instruction at a time. This concept matches the present-day practice of using program and data in the same memory. In 1948-49, the Manchester Mark 1 became the first computer to implement the stored program concept.

The 1940s also saw the dominance of the *vacuum tube* (also called *valves*) for performing computation. But these tubes generated a lot of heat, consumed enormous power and were notoriously unreliable. The 1950s saw the introduction of *transistors* which were smaller, generated less heat and were more reliable than vacuum tubes. Transistors too made way for *integrated circuits (ICs)* and eventually to the *microprocessor* which incorporated the entire CPU on a single chip. The rest, as they say, is history.

Note: Even though Leibniz had, in the 17th century, advocated the use of the binary number system for computation, it was finally adopted in the middle of the 20th century.

1.4 COMPUTER GENERATIONS

Section 1.3 presented an outline of 4000 years of history by identifying three major phases (mechanical, electro-mechanical and electronic), with the third phase having the shortest lifespan. A closer look at this phase reveals several technological developments that can be grouped into a number of *generations* (Table 1.1). The computer of each generation is faster, smaller and more powerful than its counterpart of the preceding generation. Consequently, advances in hardware have also resulted in the development of powerful and user-friendly programming languages. This section examines hardware generations and Section 3.14 discusses language generations.

1.4.1 Vacuum Tubes: The First Generation

Computers of this generation (like the Colossus and ENIAC) used thousands of vacuum tubes for computation. Memory requirements were met by magnetic drums (forerunner of today's hard disk). Because of the size of vacuum tubes, first generation computers took up a lot of space. They also consumed enormous amounts of power and generated a lot of heat. In spite of housing these computers in air-conditioned enclosures, frequent breakdowns were common. The ENIAC used 18,000 vacuum tubes, occupied 1800 sq ft of room space and consumed 180KW of power. Machines of this generation were prohibitively expensive to buy and maintain.

First-generation computers were programmed using a first-generation language—machine language. This low-level cryptic language understands only 0s and 1s, the reason why these computers were difficult to program. Program input was provided by punched cards and output was obtained on paper. Because of the mentioned drawbacks, first-generation computers were only used for scientific work and were not deployed commercially.

TABLE 1.1 Computer Generations (Hardware)

Generation	*Based on*	*Other Features*
First	Vacuum tubes	Magnetic drums for memory
Second	Transistors	Magnetic cores, disks, punched cards and printouts.
Third	Integrated circuits (ICs)	Keyboard, monitor and operating system
Fourth	Microprocessors	Networking
Fifth	ULSI and nano technology	Mainly unclear

1.4.2 Transistors: The Second Generation

Transistors replaced vacuum tubes in second-generation computers which became available in the 1950s. Compared to vacuum tubes, transistors were faster, smaller and consumed less power. Smaller magnetic *cores* also replaced the first-generation magnetic drums. Even though transistors generated less heat, second-generation computers still needed air-conditioning. These computers were thus more reliable than their immediate predecessors. The input-output mechanism, however, remained largely unchanged.

Second-generation computers were programmed using a symbolic or *assembly* language. Programming was no longer a nightmare because a programmer could specify words and symbols as instructions instead of 0s and 1s. The computers also implemented the stored program concept which allowed both program and data to reside in memory. Higher-level languages like COBOL and Fortran also began to make their appearance. These languages were soon to displace assembly language for most applications.

1.4.3 Integrated Circuits: The Third Generation

Computer speed and efficiency got a sharp boost with the development of the integrated circuit. This technology, developed by Texas Instruments, made it possible for the components of an electronic circuit—transistors, resistors, capacitors and diodes—to be integrated onto a single chip. By virtue of miniaturization, computers consequently got smaller, cheaper and energy-efficient. For these reasons, they could be seen in several medium-sized organizations.

This generation bid goodbye to punched cards and adopted a keyboard and monitor to interact with the user. Memory capacity increased substantially and the magnetic hard disk was used for secondary storage. Third-generation computers also had an *operating system*, which is a special program meant to control the resources of the computer. By virtue of a feature known as *time sharing*, the computer could run programs invoked by multiple users. The existing programming languages were supplemented by BASIC, C, C++ and Java.

1.4.4 The Microprocessor: The Fourth Generation

In this current generation which produced the microprocessor, integration of components went several steps ahead. Using *LSI (Large Scale Integration)* and *VLSI (Very Large Scale Integration)* technology, it is now possible to have the entire CPU, its associated memory and input/output control circuitry on a single chip. Intel introduced the 4004 microprocessor in 1971 and improvement in the usual parameters (like speed, heat generation, size, etc.) continues at a frenetic pace to this day. Microprocessors have invaded our homes to drive desktops, laptops, smartphones, microwave ovens and washing machines.

There have been other sweeping changes in this generation. Laptops and smartphones offer gigabytes (GB) of memory compared to a few megabytes (MB) that were available in the early days of this generation. Operating systems have moved from the rudimentary MSDOS to a mouse-based *Graphical User Interface (GUI)* like Windows. More advanced systems like Linux are now available for desktops and laptops, and a variant of it (Android) powers most of our smartphones. Fourth-generation languages (4GLs), which resemble natural languages, have also come into being.

This generation has also made rapid strides in networking technology. Sharing of information became possible by connecting computers in a network. Using *TCP/IP* technology developed by the US Defense Department, millions of computers are today connected to this vast network called the Internet. Nothing developed in this generation has been obsoleted yet; the fourth generation remains work-in-progress.

Note: Speed and miniaturization are linked. As components get smaller, electrical signals have shorter distances to travel, thus resulting in higher speed.

Takeaway: A computer needs an operating system to run. This is a special program that runs in the CPU as long as the machine is up. MSDOS, Windows and Linux are some of the well-known operating systems.

1.4.5 Artificial Intelligence: The Fifth Generation

The fifth generation represents a vision of the computers of the future. The conventional parameters of computing (speed, size, energy consumption, VLSI to ULSI, etc.) would continue to improve, but path-breaking changes in the way we use computers are also expected. *Artificial Intelligence (AI)* and use of natural languages are the main features of this generation.

Fifth-generation systems *should* be capable of producing human-like behavior. These systems are expected to interact with users in natural language and learn from experience. Speech recognition and speech output should also be possible with these systems. A few of these objectives (like voice recognition) have been partially realized today. As complexity increases manifold, the following things need to happen:

- Computer speeds need to make an exponential jump, a feat that would be possible using *quantum computers*. Google's D-Wave 2X quantum computer is 100 million times faster than today's machines. If a machine has to learn itself, quantum computing will help it solve a problem in an instant.
- Computers must be able to perform *parallel processing* so that multiple processors can concurrently handle different aspects of a problem. We have already seen the introduction of quad-core and octa-core processors in laptops and smartphones.
- *Neural networks* and *expert systems* have to be developed. These applications would be able to make decisions and advise humans by analyzing data using human-like intelligence but without using the services of an expert. Financial institutions currently use these technologies for detection of credit card fraud.

There are other possibly disruptive technologies—like *molecular computing*—that could take miniaturization to molecular levels. Since our view of this generation is prospective and not retrospective (which the previous generations were), it is difficult to fathom where this generation would end and the next one would begin. We are already in a situation where serious people have begun to question whether the power bestowed on robots has already exceeded the power of humans to control it.

1.5 COMPUTER TYPES

Apart from being classified by generations, computers can also be categorized by their size. The size of a computer is often an indirect indicator of its capabilities and the application domains where

they are employed. Because of constantly changing technology, there would be some overlap here; today's mainframe could be tomorrow's minicomputer. We present these categories in order of descending size.

1.5.1 Supercomputers

These are huge machines having the most powerful and fastest processors. A supercomputer uses multiple CPUs for parallel data processing, which means portions of the same program are handled independently by multiple processors. Speeds are measured in *flops* (floating point operations per second). The fastest supercomputer (the Tianhe-2) operates at a speed of 34 petaflops. (1 peta = 1000 tera = 1,000,000 giga). Supercomputers can handle multiple users but that's not its unique feature.

Supercomputers are too powerful to be used for transaction processing. They are employed in areas that require enormous number crunching—like weather forecasting, analysis of geological data, nuclear simulation and space exploration. They are also used to solve complex scientific problems. Supercomputers have enormous storage, use huge amounts of power and generate a lot of heat. Because of their exorbitant cost, they are mainly used by government agencies.

1.5.2 Mainframes

Before the advent of supercomputers, mainframe computers were the largest computers available. These are multi-user machines that can support hundreds or thousands of users using the feature of *time sharing* supported by systems like Linux. Users interact with this system using a terminal and keyboard, which is akin to the way we use PCs. Mainframes can concurrently run multiple programs even with a single CPU. The processor speed in a mainframe is measured in *mips* (million instructions per second).

Mainframes are generally used to handle data and applications related to the organization as a whole. Their use is no longer confined to processing the payroll. Today, mainframes are employed to handle online transactions (stock exchange transactions, for instance). The capability to handle large amounts of data makes the mainframe suitable for use in government, banks and financial institutions, and large corporations.

Note: The supercomputer is employed to run one program as fast as possible. The mainframe is meant to serve a large number of users. If heavy number crunching is not required, mainframe is a better option.

1.5.3 Minicomputers

Minicomputers or midrange computers (as they are known today) can be considered as downsized mainframes since they have the essential features of mainframes. Minicomputers can serve hundreds of users and are small enough to partially occupy a room. But they are not affordable enough to be used in the home. The minicomputer, which Digital Equipment Corporation (DEC) introduced in the 1970s, is thus rightly positioned between a mainframe and a microcomputer. It is unusual, but not impossible, for a mini to have multiple CPUs.

Minicomputers are used in smaller organizations or a department of a large one. They are also used as captive machines of turnkey equipment running specialized software. While minis were initially deployed as multi-user systems, the emergence of the *local area network* led to their decline. It is now possible to connect several microcomputers in a network where one computer doesn't need to have all the processing power.

1.5.4 Microcomputers

The microcomputer or personal computer (PC) is a late entrant to the computer family. Introduced by Apple and later endorsed by IBM, this computer is a single-user machine powered by a single-chip microprocessor. Today's PCs are very powerful machines having gigabytes of memory and a terabyte or two of disk storage. They are used both in the standalone mode (at home) and in a network (in office). A microcomputer takes the form of a desktop, notebook (laptop) or a netbook (smaller laptop). Even though a PC has a single CPU, high-end PCs support microprocessors with multiple *cores* (like the Intel Core i5 and i7). Each core can be considered—with some approximation—as a processor by itself.

PCs today are powered by three types of operating systems—Windows (7, 8 or 10), Mac OS X (Apple) and Linux. Businesses and individuals use the PC for word processing, spreadsheet handling and desktop publishing. PCs also support Internet browsing software like Firefox and Google Chrome. All PCs are multimedia ready; they can handle images, audio and video files. The PC is also an entertainment device. The range of software available is virtually unlimited; more software is available for the PC that for any other type of computer.

A variant of the microcomputer is the *workstation* which essentially is a microcomputer with a more powerful processor, a high resolution terminal and high quality graphic capabilities. They are used for engineering and scientific applications (like CAD/CAM) and for software development. Workstations are usually connected in a network for sharing of its resources, the reason why some of them are also diskless.

1.5.5 Smartphones and Embedded Computers

The relentless drive toward miniaturization has led to the emergence of two types of devices called smartphones and *embedded* computers. The smartphone is a general purpose computer that is also capable of making phone calls. The embedded computer is a small computer-like system that is part of a larger system. For the sake of completeness, we need to have a brief look at both of these devices.

The smartphone has a powerful processor, usually with multiple cores (like the quad-core Snapdragon 820). It also supports gigabytes of main memory but doesn't have a hard disk for secondary storage. This requirement is met by flash memory *(1.10.4)*. Smartphones today run a well-developed operating system (Android or iOS), and can run a wide range of applications (popularly called "apps"). Like a computer, a smartphone has a keyboard and a high-resolution display, both operated by touch or a stylus. Applications are written in a high-level language—Objective-C for the iPhone and Java for Android phones.

Embedded computers (also called *micro-controllers*) arrived before smartphones. These are very small circuits containing a CPU, non-volatile memory, and input and output handling facilities. They are embedded into many of the machines that we use—cars, washing machines, MP3 players and cameras. The processor here runs a single unmodifiable program stored in memory. Embedded computers can't match the capabilities of a smartphone, but they don't need to. Because the processor is dedicated to a specific task, its capabilities are just adequate to operate the device.

1.6 BITS, BYTES AND WORDS

We have already used the terms *megabytes*, *gigabytes*, etc. for quantifying memory, so let's clearly understand these units because we'll be using them frequently. In everyday life, we use the decimal system for computing where a digit can have ten values (or states). However, a computer understands only two states: 0 or 1. A digit that can have only two states or values is known as a binary digit, abbreviated to *bit*. All binary numbers are represented by combining these 0s and 1s. The number 13 in decimal is represented as 1101 in the binary system.

The name *byte* was coined to represent eight bits. The byte is the standard unit of measurement of computer memory, data storage and transmission speed. A DVD-ROM has a capacity of 4.7 gigabytes (4×10^9 bytes), and internet speeds of 4 megabits per sec (4×10^6) are common. The relationships between these units are expressed in the following manner:

Unit	*Equivalent to*	*Remarks*
1 kilobyte (KB)	1024 bytes	Space used by 10 lines of text.
1 megabyte (MB)	1024 kilobytes	Memory of the earliest PCs.
1 gigabyte (GB)	1024 megabytes	Storage capacity of a CD-ROM.
1 terabyte (TB)	1024 gigabytes	Capacity of today's hard disks.
1 petabyte (PB)	1024 terabytes	Space used for rendering of film Avatar.

A similar relationship exists for bits as well (or, for that matter, any quantity). So, the speed of a network expressed as 1 Mbps (megabits per second) is the same as 1000 Kbps. It is customary to use "B" to mean bytes and "b" to signify bits. Thus, 1 KB signifies 1000 bytes but 1Kb is the same as 1000 bits.

Even though computer memory is measured in bytes (rather mega- or gigabytes), the CPU handles memory data in larger units, called *words*, where a word is usually an even multiple of bytes (two bytes, four bytes etc.). Most machine instructions are one word in size, and because a CPU register must be able to hold an instruction, it too is one word wide. When we refer to a computer as a 32-bit (4 bytes) machine, we mean that the size of its word is 32 bits.

Takeaway: Memory is measured in KB (kilobytes), MB (megabytes), GB (gigabytes) and TB (terabytes). Each unit is 1024 times as big as its immediate predecessor. For instance, 1 GB = 1024 MB. For reasons of convenience, we often use 1000 as the multiplier instead of 1024.

1.7 INSIDE THE COMPUTER

We'll now make a generic examination of the way a computer works with reference to the schematic diagram shown in Figure 1.3. The brain of the computer is the *Central Processing Unit (CPU)*, represented by a single chip on a PC. The CPU carries out every instruction stored in a program, while interacting with other agencies as and when necessary. Most of the work is done by the *Arithmetic and Logic Unit (ALU)*, which is an integral part of the CPU.

The CPU needs both fast and slow memory to work with. Fast memory is represented by *primary memory*, also known as *Random Access Memory (RAM)*. It is divided into a number of contiguously numbered cells. The numbers represent the *addresses* of the cells. The CPU accesses a memory location by sending its address to the memory unit. Primary memory is used for storing instructions and data of the program currently in execution (online storage).

The computer also supports slower *secondary memory*, also called *secondary storage* or *auxiliary memory*. This is generally the hard disk but it can also be a CD-ROM or DVD-ROM. Secondary memory is used for storing data not required currently (offline storage). Data in secondary memory are stored as *files* having unique names. A program is executed by loading instructions and data from secondary memory to primary memory.

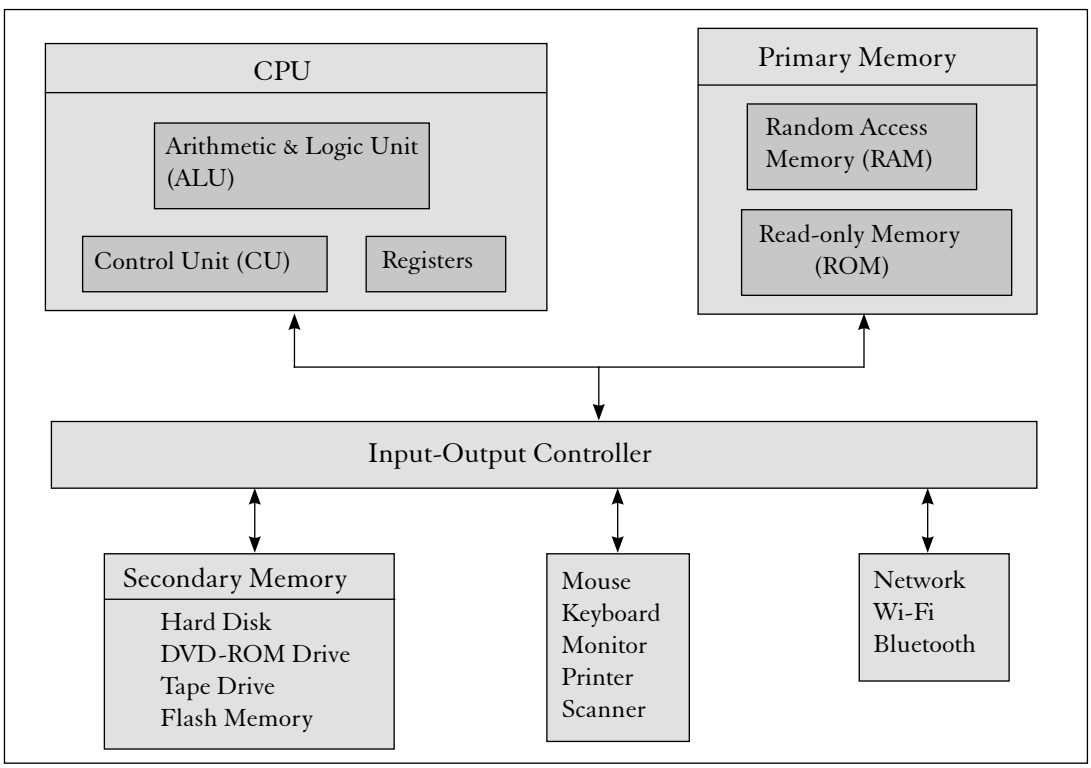


FIGURE 1.3 Architecture of a Computer with a Single Processor

Users interact with a computer system using input and output devices. Data keyed in through the keyboard (an input device) are converted to binary form by the device before they are saved in RAM. After processing, the output data (also saved in RAM) have to undergo reverse conversion (binary to a human-readable form) before they can be displayed on the monitor or printer (both output devices). Some forms of secondary storage can also act as input/output devices.

There are three types of data that move between the various components of the computer. The data of a specific type move along a distinct pathway, called a *bus*, which is reserved for that type. Program instructions and data move along the *data bus*. Memory addresses travel along the *address bus*. All control signals use the *control bus*.

We will now embark on a discussion of the various components that have been highlighted in this section. Emphasis will be laid on the forms seen on the PC because that's where you are likely to encounter these components.

1.8 THE CENTRAL PROCESSING UNIT (CPU)

The CPU has evolved from a bulky vacuum tube-based unit of the 1940s to a modern 5cm square chip that is commonly called the *microprocessor*, or simply, *processor*. It comprises the following components:

- Arithmetic and Logic Unit (ALU)
- Control Unit (CU)
- Special-purpose registers
- A clock

Acting in tandem, they control and perform all arithmetic and logical operations. This chip generates a lot of heat, the reason why it is mounted on the motherboard with both a heat sink and a fan to dissipate the heat.

The *Arithmetic and Logic Unit (ALU)* is a "super calculator". Apart from carrying out all arithmetic tasks, the ALU compares two numbers and performs *boolean* operations (like AND, OR and NOT) on them. The *Control Unit (CU)* doesn't perform any computational tasks. Using signals, the CU controls the way data is moved between the various components of the computer. Both ALU and CU use the services of a clock for sequencing and synchronizing their operations.

The CPU uses a few high-speed *registers* (a form of memory) to store the current instruction and its data. One of the registers, called the *program counter*, stores the address in primary memory of the next instruction to be executed. This counter is incremented after an instruction has been fetched.

All program instructions are executed using the *fetch-decode-execute* mechanism. An instruction is first fetched from primary memory using the address stored in the program counter. The counter is immediately incremented to reflect the address of the next instruction. The fetched instruction is next decoded to create the signals needed for the operation. The converted instruction is then executed by the CPU (ALU, specifically, if an arithmetic or logical operation is involved). The output, if any, is then written to another register from where it is copied to primary memory. The next cycle can now begin.

All of these activities need synchronization by the clock which generates pulses at regular intervals. The CPU cannot execute more than one instruction in one clock pulse even though one instruction may take several clock pulses to execute. Thus, the faster the clock, the faster is the execution. CPUs today are rated in GHz (gigahertz). The Intel Core i7 has a clock speed of 3.4 GHz, which means it emits 3.4 billion pulses in one second. However, CPU performance is rated in MIPS (million instructions per second), which will always be lower than the number of clock pulses emitted.

The preceding discussions were based on the single-CPU architecture. But today, the higher-end processors (including those meant for smartphones) have multiple *cores* in the chip. An octa-core processor has eight processors in the chip. For a program to take advantage of multiple processors, different segments of the program must be able to run independently without depending on one another. The operating system must also have the capability to distribute the work among these multiple processors.

Takeaway: The CPU fetches instructions and data from primary memory and stores them in its registers. After processing, the CPU stores the output data in registers before moving them to primary memory. All activities are synchronized by a clock.

1.9 PRIMARY MEMORY

Memory is an essential resource used by the computer and its users. The CPU needs memory for the currently running program. Users need memory to back up programs and data that are not needed currently. For meeting the differing requirements, the computer supports multiple types of memory. For convenience, we categorize memory into two types—primary and secondary. This section examines primary memory which includes the following types:

- Random Access Memory (RAM—SRAM and DRAM)
- Read Only Memory (ROM, PROM, EPROM, EEPROM)
- Cache Memory (L1, L2 and L3)
- CPU Registers

All of these memory units are housed on the motherboard of the PC. The fastest ones are included in the CPU itself, while the relatively slower RAM is outside the CPU. The CPU needs to use all of them for the program currently in execution. Except for the ROM family, all of the memory types are volatile, i.e., they lose their contents when power to the computer is withdrawn.

1.9.1 Random Access Memory (RAM)

Random Access Memory (RAM) stores the code and data of the currently running program. This function is in accordance with the stored program concept of Turing and von Neumann *(1.3.3)*. On multi-user systems, RAM contains several programs that run in time-sharing mode. It is currently implemented as a set of densely packed transistorized cells fabricated into integrated circuits (ICs). The motherboard contains slots where a memory module containing these ICs can be inserted (or removed). RAM is fast, volatile and more expensive than secondary memory.

When a program starts execution, the CPU picks up each program instruction from RAM. It also writes back the output and temporary data into RAM. Each cell has an address (a number) which is used by other parts of the computer to access this location. To retrieve data from RAM, the CPU sends the address to the memory module through the address bus and retrieves the data via the data bus. Unlike ROM, RAM is capable of being both read and written. The access is random rather than sequential, hence the term "random access."

RAM can be broadly divided into two types—static and dynamic. *Static* RAM (SRAM) uses multiple transistors for each cell to store one bit of information. It, therefore, takes up more space than *dynamic* RAM (DRAM) which uses one transistor and a leaky capacitor for each cell. Capacitors hold the charge, but because of inherent leakage, DRAM cells have to be refreshed every two or three milli-seconds. Because of these properties, SRAM is faster, bigger and more expensive than DRAM. The cache memory in the CPU is implemented as SRAM. However, DRAM can provide large capacities at moderate speed and low cost. A typical PC configuration offers around 2 GB DRAM.

Even though RAM is faster than secondary memory, it is the slowest of all the primary memory types. In some situations, the CPU needs even faster memory, and this requirement is met by the other three types.

Note: The CPU has a direct connection with RAM but not with secondary memory. The contents of a hard disk (a form of secondary storage) have to be loaded into RAM before they can be accessed by the CPU.

1.9.2 Read Only Memory (ROM)

Unlike RAM, *Read Only Memory (ROM)* signifies permanent memory that can be read but not written. The contents of a ROM are written at the time of its manufacture, though later enhancements have enabled ROMs to have limited write capabilities. Memory access is faster in ROMs compared to RAMs. ROMs are also non-volatile, i.e. their contents are retained after withdrawal of power. Their capacity is severely limited—1, 2 or 4 MB compared to 1, 2 or 4 GB for DRAM (differing by a factor of 1024).

ROMs are used to store permanent programs or information that don't change. In the PC, a ROM stores a special program called the *BIOS* (Basic Input Output System). This is a small program that initiates the computer's startup process before transferring control to the operating system. The ROM is also used to store lookup tables. The code for computing the sine or cosine (two trigonometric functions) of numbers are often stored in a ROM.

The need to change the contents of a ROM have led to the development of superior types. All of the following types of ROM are non-volatile and have the capacity to be written at least once:

- *Programmable Read Only Memory (PROM)* This type is initially left blank at the time of manufacture. It is subsequently programmed to contain a program of the customer's choice. A device called a PROM programmer *burns* data into the PROM. However, once written, the contents of the PROM can't be changed.

- *Eraseable Programmable Read Only Memory (EPROM)* EPROM is one step ahead because it can be rewritten (but only once) even though it has been previously burned. An EPROM is erased by exposing it to ultra-violet radiation. This type is largely obsolete today as it has been superseded by EEPROM.
- *Electrically Eraseable Read Only Memory (EEPROM)* Unlike EPROM, an EEPROM can be erased and rewritten multiple times. A higher than normal electric voltage is used to erase an EEPROM, which can be allowed to remain in the computer. There is a limit, however, to the number of times an EEPROM can be programmed. The flash drive (pen drive or SD card) that we use today is a type of EEPROM.

The CD-ROM and DVD-ROM are also technically ROMs, but because enhancements have changed them to CD-RW and DVD-RW (i.e., rewrite capability has been added), they appropriately belong to Section 1.10.3 where they are treated as offline storage.

Note: Strictly speaking, EEPROMs are more like RAMs rather than ROMs because of their write capability.

1.9.3 Cache Memory

The computer also supports *cache memory* to hold those portions of a program that are frequently used by the CPU. This fast but small memory is available as a buffer between the CPU and RAM. When executing a program, the CPU first looks for the instruction and data in the cache. If it finds them there (cache hit), access of the slower RAM is avoided. If the data is not found in the cache (cache miss), the operating system places it there. Generally, hits exceed misses, so cache memory helps programs execute faster. Cache is generally implemented as SRAM (static RAM).

Modern computers support multiple levels of cache. The CPU itself contains level 1 (L1) cache, which is the smallest and fastest cache of the system. Level 2 (L2) cache could be located outside the CPU but close to it. Some CPUs also support a level 3 (L3) cache, which is slower than L1 and L2 but faster than RAM. If data is not found in L1, the CPU looks for it in L2 and then in L3.

Cache memory is expensive, so they are limited in size. The four-core, 64-bit Intel i7 CPU supports separate L1 and L2 caches, but L3 is shared by the cores. The size of L1 is 32 KB, that of L2 is 256 KB, while the four cores share 8 MB of L3 cache (2 MB per core).

1.9.4 Registers

The small number of ultra-fast *registers* integrated into the CPU represent the fastest memory of the computer. Unlike the cache which enhances performance, the registers are the workplace where the CPU does all of its work. Each register has the length of the computer's word, so the register of the Intel i7 is 64 bits wide. The computer loads an instruction and its data from main memory to registers before processing them.

Registers, being very few in number, are numbered, and a program instruction specifies these numbers. Consider, for instance, the multiplication of two numbers, represented by the pseudo-instruction "OR1R2R3". The CPU would interpret this instruction in this manner: Run the multiplication operation *O* on its operands stored in registers *R1* and *R2* and place the

output in register *R3*. The contents of *R3* would eventually be moved to RAM before the next instruction is read in.

Note: Prior to this section, we used the term *primary memory* to represent RAM. This section redefines it to signify the memory that is directly connected to the CPU, irrespective of whether it is volatile or not. Contemporary usage of this term is consistent with the one pursued here.

1.10 SECONDARY MEMORY

The computer also supports *secondary memory*, also known as *secondary storage* or *auxiliary memory*. This memory type is not directly connected to the CPU but it can exist both in the machine or external to it. Secondary memory is non-volatile and thus meets the requirements of offline and long-term storage. These devices are slower than primary memory but they have larger capacities. They are also way cheaper than primary memory; 1 GB of hard disk space is much cheaper than 1 GB of RAM.

The last couple of decades have seen the emergence of multiple types of storage devices. They have different physical sizes and capacities and may or may not require power. In this section, we consider the following storage devices with their typical capacities indicated in parentheses:

- Hard disk including the portable disk (500 GB to 4 TB).
- Magnetic tape (20 TB).
- CD-ROM (700 MB—less than 1 GB).
- DVD-ROM (4.7 GB and 8.5 GB).
- Blu-ray disk (27 GB and 50 GB).
- Flash memory based on the EEPROM (1 GB to 128 GB).
- The obsoleted floppy disk (1.2 MB and 1.44 MB).

Some of these devices may be part of the computer's configuration, while others may be connected to it through ports and connectors. Even though these devices are meant for offline storage, a program in execution often needs to access them. A program may, for instance, read data from a CD-ROM and write the output to flash memory. Further, the data in these devices are organized in *files* unlike in primary memory where they are arranged contiguously.

Takeaway: Data in secondary memory is accessed by the name of the *file* associated with the data. On the other hand, data in primary memory is accessed by the *addresses* of the memory locations containing the data.

1.10.1 The Hard Disk

Also known as *hard drive* or *fixed disk* because it is fixed inside the computer, the *hard disk* represents one of the oldest but most commonly used forms of secondary storage. Hard disks have very high capacities with the cheapest cost. It is common for a laptop to have 500 GB of storage. Desktops

typically have 1 TB (terabyte) of storage. The hard disk is no longer an offline device because it houses the computer's operating system along with all programs and data.

Every disk contains a spindle that holds one or more *platters* made of non-magnetic material like glass or aluminium (Fig. 1.4). Each platter has two surfaces coated with magnetic material. Information is encoded onto these platters by changing the direction of magnetization using a pair of read-write heads available for each platter surface. Eight surfaces require eight heads. The heads are mounted on a single arm and cannot be controlled individually.

Each surface is composed of a number of concentric and serially numbered *tracks*. There are as many tracks bearing the same track number as there are surfaces. You can then visualize a *cylinder* comprising all tracks bearing the same number on each disk surface. Thus, there will be as many cylinders in the disk as there are tracks on each usable surface. Each track is further broken up into *sectors* or *blocks*. So, if each track has 32 blocks and a disk has eight surfaces, then there are 256 blocks per cylinder. A block typically comprises 512 bytes.

The disk spins constantly at speeds between 5400 and 7200 rpm (revolutions per minute). The disk heads move radially from track to track, and when the head is positioned above a particular track, all of its blocks pass through the head in a very short time. The access time is more here compared to RAM because of the delay in moving the head to the right location. The heads are not meant to touch the platter surface, and if that happens (on account of dust creeping in), then a *disk crash* results. The data could then be unrecoverable.

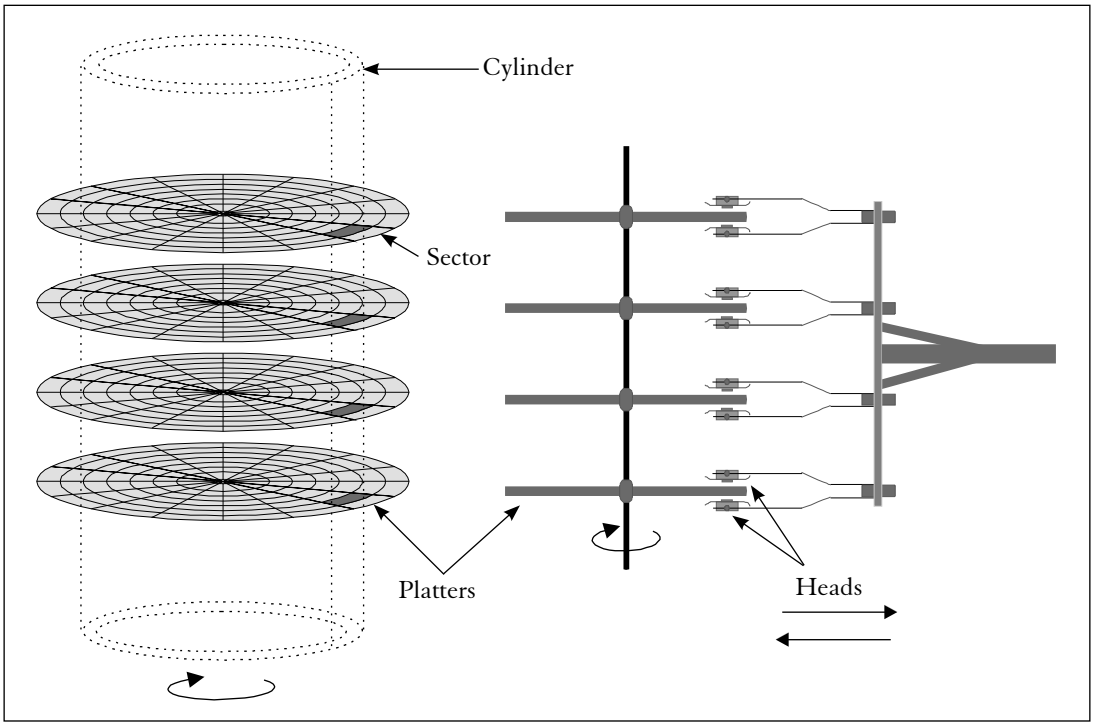


FIGURE 1.4 The Hard Disk

Note: The disk on the author's Linux computer has the following parameters:

Heads = 255,
Cylinders = 121,601,
Sectors per track = 63,
Bytes per sector = 512 (a common block size)

The capacity of this hard disk is 255 × 121601 × 63 × 512 = 1,000,202,273,280 bytes. This is approximately 1 terabyte (TB) or 1024 gigabytes (GB).

A variant of the fixed disk is the portable disk that has recently acquired popularity. This disk is available in two types. The first type, available up to 4 TB in capacity, uses the USB port *(1.11)* of the desktop or laptop to power the device and transfer data. The second type, supporting capacities exceeding 4 TB, needs an external power supply and uses the USB port only for data transfer.

Note: The hard disk can act like an input and output device at the same time. It is possible for one program to read a disk while another writes onto it.

1.10.2 Magnetic Tape

The age-old magnetic tape is still around thanks to the enhancements that have been made to this device. The basic technology has not changed though; the tape is made of a plastic film with one side coated with magnetic material. The entire mechanism comprising two spools and the tape is encapsulated in a small cassette or cartridge. Current technology supports capacities of 1 TB or more, but 200 TB tapes are expected to be launched in the near future. The device is not fully portable though because a separate tape drive is required, and most computers don't have one.

Data are read from and written to the tape using a read-write head and an erasure head. The write operation is preceded by the erasing operation. Unlike RAM, data access is sequential and not random. To locate a file, the tape has to be rewound before a sequential search can begin. This makes it rather unsuitable for restoring individual files. Tape backup is most suitable for archiving data of the hard disk that are not expected to be needed at short notice. The backup is inexpensive and convenient for restoring lost data.

1.10.3 Optical Disks: The CD-ROM, DVD-ROM and Blu-Ray Disk

Non-volatile read-only memory, which we saw in the ROM family (including PROM, EPROM and EEPROM), is also available on optical disks. These disks, comprising mainly the CD-ROM and DVD-ROM, can hold large volumes of data (700 MB to 8.5 GB) on inexpensive media. The Blu-ray disk enhances this capacity to 50 GB. A laser beam in their drives controls the read and write operations. Because most PCs and laptops have these drives (not Blu-ray though), data backups on optical disks are easily distributed.

Optical disks are made of carbonate material with a thin layer or two of reflective film. A laser beam is used to construct *pits* and *lands* by burning (writing) selected areas along its tracks. When reading data, the beam is reflected from a land (signifying a 1) but not from a pit (signifying a 0).

Unlike in the hard disk, the tracks represent a single spiral that moves outward from the center. The disk rotates at high speed which, for CDs, is specified as a multiple of the speed of music CDs (150 KB/second). A music CD is rated as 1X while data CDs are commonly rated as 52X.

A CD-ROM has a capacity of 700 MB. A DVD-ROM which has its pits and lands much more closely spaced has a capacity of 4.7 GB. Addition of a second layer increases the capacity to 8.5 GB (a dual-layer DVD). Further, with the introduction of the "R" (recordable) and "RW" (rewritable) types, it is now possible to record data onto the media:

CD-R, DVD-R—Data can be recorded only once.

CD-RW, DVD-RW—Data can be recorded multiple times.

Initially, we had separate drives for reading and writing, but today a DVD writer can read and write both media. Even though CDs and DVDs are reliable media, they are likely to make way for flash memory which already supports much higher capacities without using any moving parts. The future of the Blu-ray disk for data backup is uncertain, even though it is extensively used by the music and film industries.

Note: The optical drive uses three motors for the following functions: operating the tray, spinning the disk and guiding the laser beam.

1.10.4 Flash Memory

This class of devices, having no moving parts, is based on the EEPROM. It is available in various forms—pen drive, solid state disk (SSD) and the magnetic card (SD card) (Fig. 1.5). They are portable, need little power and are quite reliable. The capacities offered by this class of devices are increasing exponentially while their prices continue to decline substantially.

The *memory stick or pen drive* is the most common type of flash memory used on the computer. It is a small, removable piece of circuit wrapped in a plastic casing. It connects to the USB port of the computer where it is detected as yet another drive by the operating system (say, `G:` in Windows). Even though it is a read-write device, the number of writes is limited (around 10,000), which is adequate for most users. With 16 GB devices available at affordable prices, the USB pen drive has now become a serious competitor to the CD and DVD as an offline storage device.

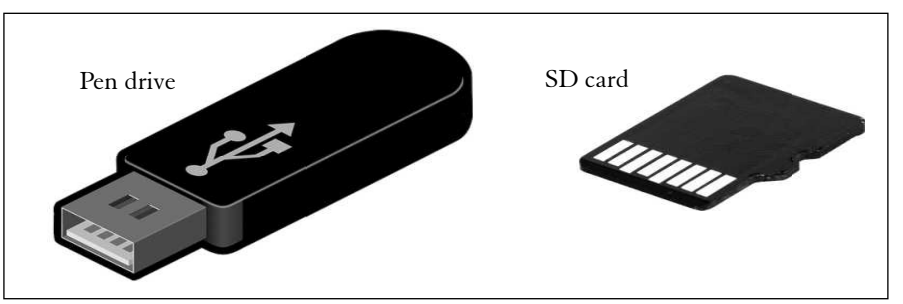


FIGURE 1.5 Flash Memory

The *solid state disk (SSD)* is a bigger device meant to replace the traditional magnetic hard disk. Many small laptops (like Chrome books) have the operating system and a small set of programs stored on this *online* device. The third device, the *magnetic card*, is used mainly in cameras, but using adapters, they can connect to the USB port as well. The most popular form of this device is the micro-SD card, which is available in SDHC and SDXC flavors. The SD card offer high capacities that can go up to 128 GB.

Note: Unlike CDs and DVDs, which can only be written using a software like Nero, flash memory is written in the same way the hard disk is written. No separate software is required.

1.10.5 Floppy Diskette

The *floppy diskette* was once the only form of portable storage that could be carried in the pocket. This storage medium is represented by a rectangular plastic case containing a thin magnetic disk. A read/write head actually makes contact with this disk while it is rotating. The floppy was available in two sizes (5.25" and 3.5"), offering capacities of 1.2 MB and 1.44 MB (yes, MB not GB), respectively.

The small capacity of a floppy diskette was enough for most uses in those days, but today a diskette can't even hold a single 3-minute song encoded in MP3. This made it unsuitable for backing up large files. Previously, operating systems (like UNIX and Windows) were offered on multiple diskettes. The medium is unreliable and prone to frequent irrecoverable damage. It was eventually replaced with the reliable CD-ROM and DVD-ROM, which offer much higher capacities. Desktop PCs no longer support a drive for a floppy diskette and Windows offers no support for it.

Note: On a lighter note, Windows has not been able to forget the floppy. The hard disk is still labeled C:, while A: and B: continue to be reserved for the floppy. It must remain this way as long as legacy file-handling programs continue to use A: and B: (unlikely though).

1.11 PORTS AND CONNECTORS

Before we examine the external devices used with a computer, let's spend some time on the major sockets that computers support. Devices like printers and scanners connect to a computer through docking points called *ports*. Current computers and laptops offer fewer types of ports today than they used to previously. All ports are connected to the motherboard but are visible from the outside. They are shaped differently, so it is impossible to use a wrong connector for a port. These are the ports that you'll encounter:

- *Universal Serial Bus (USB)* The USB port has virtually replaced the serial and parallel ports in the motherboard. Most computers offer at least four USB ports to support scanners, printers and mice. A USB port has four lines, two each for data and power. The current version, USB 3.0, can transfer a 1 GB file in 20 seconds. Also, a smaller variant, the micro-USB port, is used on portable hard disks and smartphones.

- *Serial port* Once used by the keyboard, terminals, mice and modems, serial ports are offered in 9- and 25-pin configurations. Data pass through a serial port one bit at a time.
- *Parallel port* If a printer is not using the USB port, then it is probably using the parallel port. The common implementation (introduced by IBM on the PC) uses 25 pins where data are transferred in parallel, i.e., synchronously.
- *Video Graphics Array (VGA) port* This 15-pin port allows transfer of *analog* (continuous) video data to the monitor. This port is being replaced with the *digital video interface (DVI)* which uses *digital* data that is used by flat LCD panels (which also use analog signals). DVI offers better video resolution than VGA.
- *RJ45 port* This port is used by the Ethernet network *(1.14)*. Both computer and the network device (like hub or router) have the female form of the port. Even if the computer connects wirelessly to a network (using built-in Wi-Fi or a USB adapter), the wired RJ45 remains a useful option.
- *PS/2 port* This port has replaced the serial port for connecting the keyboard and mouse. It has 6 pins but occurs as a pair in two different colors. The port and connectors for the keyboard are purple, while the mouse uses the green port. The laptop doesn't support these ports but the desktop does. USB has invaded this area also.
- *High Definition Multimedia Interface (HDMI)* This is now the industry standard for transferring audio and video data between computers and HDTVs, projectors and home theaters. Because a single cable handles both audio and video, it is driving out the RCA 3-plug system which all desktop PCs still support.

As shown in Figure 1.6, these ports occur in male (pins) or female (holes) forms on the computer. RJ45, USB and HDMI don't use pins or holes; they use flattened pins instead. Also, the counterpart of these ports on the device side may not have the same form. For instance, a USB printer uses a different port on the printer side.

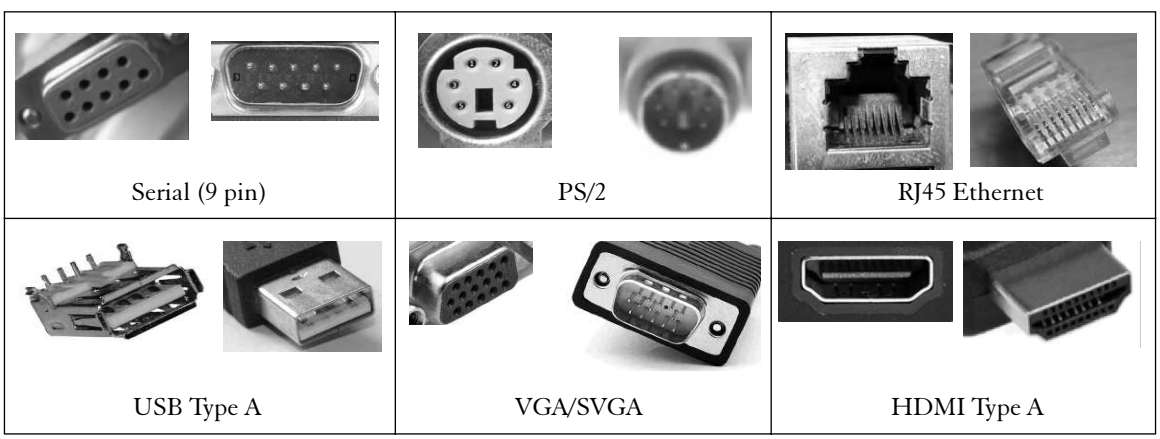


FIGURE 1.6 Common Ports

Note: The power lines of a USB port are often used to operate LED lights and small fans, and charge low-end mobile phones.

1.12 INPUT DEVICES

You need to interact with programs for providing input. You also need to interact with the operating system for performing the daily chores like editing, copying and deleting files. External devices like scanners also provide input to programs. In this section, we consider some of these input devices. Some are part of the basic computer configuration while the others have to be obtained separately.

1.12.1 The Keyboard

Every computer supports a keyboard—either a physical one or a touchscreen. The central portion of the keyboard has the *QWERTY* layout and numerals are placed above the three layers of letters. There are also a large number of symbols that were never seen in the typewriter (like ^, ~ and `). All symbols seen on the keyboard have significance in C.

Each letter, numeral or symbol is known as a *character*, which represents the smallest piece of information that you can deal with. All of these characters have unique values assigned to them, called the *ASCII value* (ASCII—American Standard Code for Information Interchange). For instance, the letter `A` has the ASCII value 65, while the bang or exclamation mark (`!`) has the value 21. When you press `A`, the binary value of 65 is transferred to RAM. You must also know the function of the following special keys:

- The *[Enter]* key terminates a line. The text that you key in remains hidden from the system until this key is pressed.
- The backspace key erases input from right to left, while *[Delete]* erases text from left to right.
- The *[Ctrl]* (called control) key, found in duplicate on the lowest row, is always used in combination with other keys. The *[Ctrl-c]* sequence (two keys) copies selected input, while *[Ctrl-v]* pastes it elsewhere.
- The 12 function keys labeled *[F1]*, *[F2]*, etc. are located in the top-most row. Many application software (like Microsoft Word) make use of these keys.

The keyboard is an external device of the desktop which connects to it using the PS/2 port. Laptops and touchscreen-based computers don't need this port because their keyboard is a builtin. If you need an external keyboard for your laptop, use a USB keyboard.

1.12.2 Pointing Devices

Graphical user interfaces (GUI) like Windows need a pointing device to control the movement of the cursor on the screen. This is commonly implemented as the mouse in the desktop and touchpad in the laptop. The earliest form of the mouse, one that is connected to a computer port, has a rotating ball at the bottom and two buttons at the top. Moving the mouse by hand partially

rotates the ball and thereby controls the motion of the cursor on the screen. The buttons are generally used in the following ways:

- Clicking on any object with the left button selects that object. Double-clicking of the same button activates a program. A text section can be selected by dragging the mouse with the left button pressed (no clicking here). A left click on a text segment is also required to commence text entry.
- Clicking on the right button activates a context-sensitive menu that can be used to change the attributes of the selected object. This menu often supports options for copying and pasting text and change the properties of the selected object.

The scroll wheel scrolls a document in two ways. You can keep the mouse stationery and move the wheel. Alternatively, click the wheel as you click a button and move the mouse a little. The document then scrolls continuously without any further motion of the mouse.

Because of its usefulness, the mouse has undergone rapid changes in design. The original mechanical mouse is now being gradually phased out in favor of the following options:

- The *optical mouse* uses an infrared laser or LED to track movement. Unlike the mechanical mouse, the optical mouse works practically on any surface.
- The *wireless mouse* uses radio frequency technology to communicate with the computer. The transmitter in the mouse communicates with a USB-connected receiver.

Where space is a constraint, you can use a keyboard having an integrated *trackball*. In this configuration, you need to move the ball which is placed in half-inserted condition on the right of the keyboard surface. Laptop users need to use the *touchpad* which is a rectangular flat surface having two buttons. Moving the tip of the finger on the surface moves the cursor. The optical and wireless mouse can be used with the laptop as well.

1.12.3 The Scanner

A *scanner* is a device that creates a digital image of a document by optically scanning it. Scanners come in various forms that are used in diverse industries. In this section, we examine the *flatbed* scanner that we all use for scanning documents not exceeding A4 size. This device connects to the USB port and is operated using a special software that is shipped with the product.

The document to be scanned is placed on a glass plate that is covered by a lid before scanning. A shining light is beamed at the document and the reflected light is captured by a photosensitive array of *charged coupled devices (CCD)*. The differing levels of brightness are converted into electronic signals. The signals are then processed to create the digital image which is saved as a file in the hard disk. The scanner can also act in photocopier mode and directly print the copied document without saving it.

Images of reasonable quality are created when the document is scanned at 300 dpi (dots per inch). Generally, pictures are converted to JPEG files and documents to PDF files. JPEG files are edited with an editing software like Photoshop before they are saved in disk or sent as a mail attachment. To keep the file size low, it may be necessary to convert a colored image to grayscale or black-and-white. PDF files, which have relatively small file sizes, are read by Adobe Acrobat Reader.

Tip: Choose the lowest resolution that will serve your purpose. Low resolution and black-and-white images have small file sizes. Size is a limiting factor when files are sent across the Internet. The cost of transmission is directly related to the size of the file.

Modern scanners have the *Optical Character Recognition (OCR)* facility by which a document (or its image) can be scanned in a special mode to *extract the text as a stream of characters*. This needs special software that can interpret the dot pattern in a picture as a character, say, `A`. The advantage in using OCR is that an image file can be converted to a text file which can be edited with text editing software. The file size also reduces substantially. If your existing scanner doesn't have the OCR function, then you need an OCR scanner.

Takeaway: If you have lost the file of a Word document but have its hard copy, then you can use OCR to "restore" the file which can then be edited with the same software.

There are other types of scanners available. Hand-held scanners are manually moved across the document. Banks use *Magnetic Ink Character Recognition (MICR)* scanners to read the codes printed on bank cheques. Shops use hand-held barcode scanners to determine the product name and price from the barcodes printed on the product label.

1.13 OUTPUT DEVICES

Do all programs generate output? The vast majority of them do. We generally use the term "output" to mean information that can be seen or heard. Printers and loudspeakers belong to this category. The scanner creates an image of a document but saves the "output" to disk. That, for us, is not output. In the following sections, we will discuss the commonly used visual output devices.

1.13.1 The Monitor

The monitor is an integral part of a computer's configuration. Programs use it to display text or graphical output, while users also need a monitor to view the input that they key in. The performance of a monitor is judged mainly by its image quality, resolution, energy consumption and its effect on the eyes. There are two competing technologies; the old CRT monitors have virtually made way for the modern LCD-based types. Monitors are connected to the desktop's VGA or DVI port. The connection is internal for the laptop.

CRT Monitor The CRT (cathode ray tube) monitor uses a rarefied tube containing three electron guns (for the three primary colors) and a screen coated with phosphorescent material. The guns emit electrons to create images on the screen by selectively lighting up the phosphors. CRT monitors typically have a resolution of 640 × 480 pixels which translates to an *aspect ratio* of 4:3 (Aspect ratio = width / height). They are large and heavy, energy-inefficient and generate a lot of heat.

LCD Monitors Most computers today are shipped with LCD monitors (or their variants, LED and TFT). An LCD screen comprises thousands of liquid crystals, which, by themselves, don't

generate light but may allow or block the passage of light through them. An image is formed by selectively applying a voltage to these crystals and using a separate light source for light to pass through them. The backlight is provided either by fluorescent light (the standard LCD) or by LEDs (hence the term *LED monitors* and *LED TVs*).

LCD monitors have a wider aspect ratio (16:9) compared to the 4:3 offered by CRT monitors. From a generic standpoint, they have greatly improved upon the drawbacks of the CRT monitor. They consume less power, generate less heat, have increased life span and take up a lot less space. It is even possible to mount an LCD monitor on the wall. However, the viewing angle is less than 180 degrees, but this attribute is undergoing constant improvement.

Note: LEDs are simply backlight providers and have no role in the formation of the image. The newer TFT (thin film transistor) is simply a variant of the basic LCD technology.

1.13.2 Impact Printers

Printers produce *hard copies* of output (in contrast to disk files which are soft copies) which include text and graphics. There are currently two basic technologies used for printing—impact and non-impact. The older impact technology uses a printhead to strike (impact) a ribbon placed between the printhead and paper. Impact printers are noisy and are being phased out, but one variant (the dot-matrix printer) is still in active use.

Dot-matrix Printer The printhead of the dot-matrix printer has either 9 or 24 pins which are fired in multiple combinations to generate letters, numerals and symbols. The ribbon is impregnated with ink, and when the pins fire against the ribbon, an impression is created on the paper behind it. The paper itself is moved forward by a drum after one line of printing has been completed. Because each pin can be controlled individually, it is possible to print graphics with this technology. However, multiple fonts can't be used.

The speed of a dot-matrix printer can be as high as 300 cps (characters per second). A printer with 24 pins offers the best print quality (low as it is) at 144 dpi (dots per inch). However, the gaps between the pins are clearly visible, so this technology doesn't produce quality output. This printer type refuses to go away simply because it can produce carbon copies on continuous stationery. Invoices and bills are often printed in multiple copies, the reason why shops generally use dot-matrix printers.

Daisy-wheel Printer The now obsolete daisy-wheel printer uses a different technology to strike the ribbon. It employs a wheel with separate characters distributed along its outer edge. The characters are thus pre-formed and not generated. For printing a character, the wheel is rotated so that the desired character directly faces the ribbon. You can't print graphics with this printer but you can change the wheel to obtain a different set of fonts.

Line Printer For heavy printing, the line printer is still the best bet. This technology uses a print chain containing the characters. The chain rotates continuously in front of the paper. Hammers strike the paper in the normal manner, but the print speeds attained with this technology are as high as 1200 lpm (lines per minute). Line printers are used to print low-quality voluminous reports and can print unattended for long stretches of time. However, a line printer is extremely noisy, the reason why it is generally kept in a separate room.

1.13.3 Non-Impact Printers

Non-impact printers address the drawbacks of impact printers. They are quiet, fast and produce documents of very high resolution. The two types that are most commonly used are *laser printers* and *inkjet printers*. The third type not discussed here is the *thermal printer* which uses heat to print text and images on heat-sensitive paper.

Laser Printer A laser printer works somewhat like a photocopier. A laser beam creates an image of the page to be printed on a light-sensitive drum. The charged areas attract black magnetic powder, called the *toner*. The image, which is created in the form of dots, is then transferred from the drum to the paper by actual contact. A separate roller heats up the paper to melt the toner which then gets fused onto the paper. Color laser printers use colored toner.

Laser printers have built-in RAM for storing documents. This RAM acts as a buffer which can free up the computer after a job has been submitted for printing. A laser printer also has a ROM to store fonts. This technology is suitable for printing text and graphics in high quality. The resolution varies from 300 dpi (dots per inch) to 1200 dpi. Speeds of 20 ppm (pages per minute) are typical, while the fastest printers operate at 200 ppm. Originally used for desktop publishing, falling prices have made the laser printer available everywhere.

Ink-jet Printer The ink-jet printer is an affordable non-impact printer whose print quality considerably outperforms a dot-matrix printer but underperforms a laser printer. In this system, a printhead sprays tiny drops of ink at high pressure as it moves along the paper. The ink, stored in a replaceable cartridge, passes through a matrix comprising a number of tiny nozzles. Like with the pins of a dot-matrix printer, characters are formed by choosing the nozzles that have to be activated. Color ink-jet printers either have separate cartridges for each color or use a multi-chambered cartridge.

Inkjet printers typically have a resolution of 300 dpi, but their speed is low—around 1 to 6 pages per minute. They are thus not suitable for high-volume printing where laser printers offer a better option. Better print quality can be obtained by using high-quality acid-free paper that can retain the quality even after years of storage.

1.13.4 Plotters

Unlike printers which print text and graphics, a *plotter* makes line drawings. It uses one or more automated pens to complete each line before taking up the next one. The commands are taken from special files called *vector graphic* files. Depending on the type of plotter, either the plotter pen moves or the paper moves or both. Plotters can handle large paper sizes, and are thus suitable for creating drawings of buildings and machines. They are also slow and expensive, the reason why wide-format printers are increasingly replacing them.

Note: Programs also take input from and write output to the hard disk. For convenience and to avoid conflict in definition, we don't consider the disk as an input/output device. We use the term storage instead.

1.14 COMPUTERS IN A NETWORK

Most organizations, large or small, no longer have standalone computers. Today, computers cooperate with one another by being connected to a network. An authorized user can now share their files with other users. An application run by multiple users can now be hosted on a single machine instead of being stored on individual machines. Many network applications today are designed in a way that enable sharing of the workload by multiple machines. As rightly perceived by Sun Microsystems many years ago, "the network is the computer."

Apart from sharing files and programs, a network allows the sharing of scarce hardware resources by all connected computers. A printer can be used by several users if the printer itself is connected to one of the computers in the network. The computer (or *node*, in network parlance) that offers these services would need to be more powerful than the computers it serves. That's how *servers* came into being as the central nodes in a network. Servers also form the backbone of the largest network of all—the Internet.

All networked nodes need a common addressing scheme that would ensure delivery of data to the right node. The scheme used by most is *Ethernet*, a hardware standard that defines the addressing, cabling and signaling requirements of the network. Ethernet is currently the dominant technology, having replaced the *token ring* and *FDDI* technologies. In this scheme, data are divided into *frames* containing the source and destination addresses. A set of network *protocols* (rules) are also used for data transmission, but they will be discussed in Chapter 2.

Note: Ethernet, token ring and FDDI are hardware technologies. For a network to function, a set of network (software) protocols also need to run. *TCP/IP* is the most dominant network protocol and it powers the Internet.

1.14.1 Network Topology

There are a number of ways or *topologies* of connecting computers (Fig. 1.7), each having its own merits and demerits. The *bus topology* uses a single cable as a bus to which all computers are connected. The failure of a single node doesn't disrupt the communication between the remaining nodes.

The *star topology* uses a central hub to which all nodes are connected. Because network traffic passes through the hub, the entire network fails if the hub fails to work. On the other hand, new nodes can be added without disrupting the service.

Nodes using the *ring topology* are connected in a closed loop without using a hub. Data moves from one node to the next one which examines each packet to determine whether it should be allowed to move further on to the next node. For uni-directional rings, failure of one node causes the network to shut down.

Nodes in a *mesh topology* are connected to one another, offering a choice of multiple routes for data to travel. When a node breaks down, the packet simply changes its route. This is the most expensive of all the topologies, and is found mainly in corporate networks.

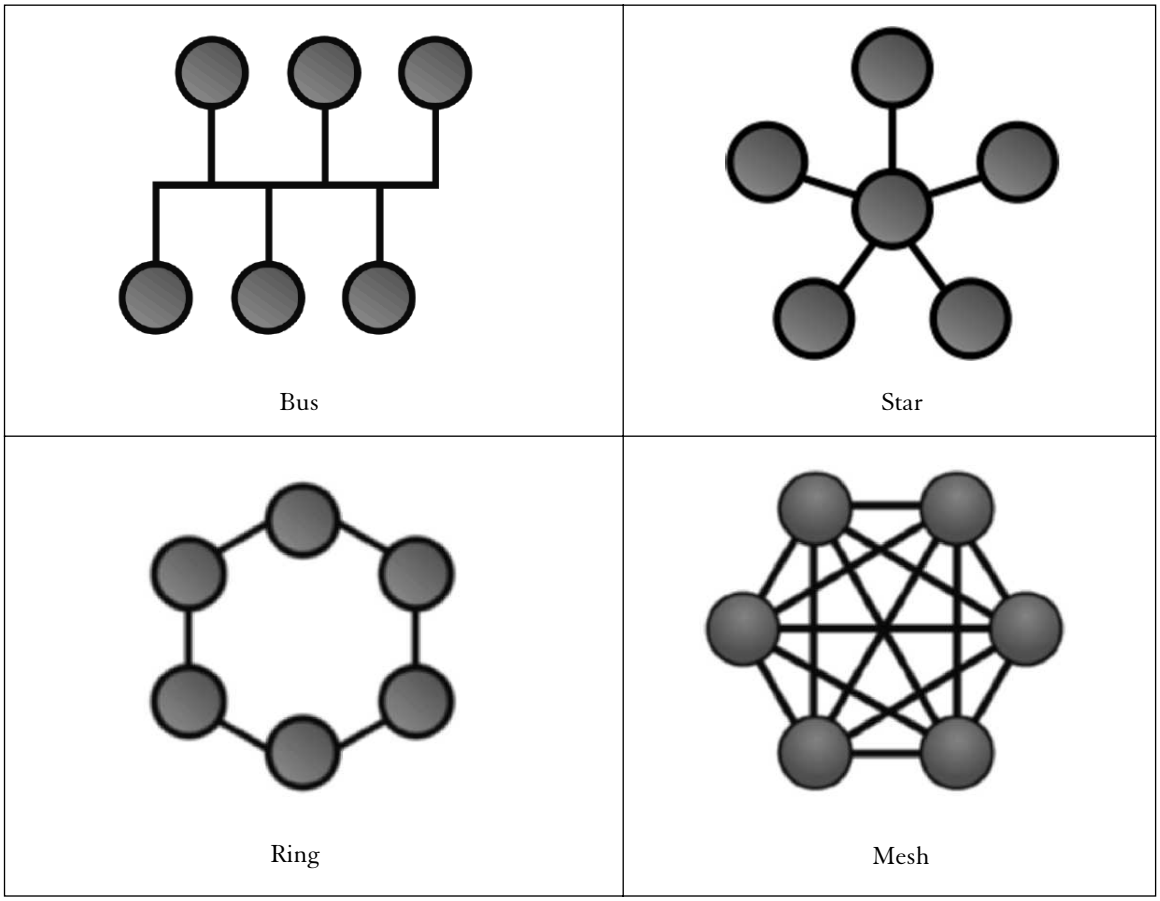


FIGURE 1.7 Network Topologies

1.14.2 Network Types

Networks are also classified based on their size. The two most common types are the *Local Area Network (LAN)* and *Wide Area Network (WAN)*. LANs are used by small organizations where the area of operation is confined to one building. They generally use Ethernet technology. The newer *Wireless LAN (WLAN)*, also known as *Wi-Fi*, has become a serious contender to the wired LAN. Standard Ethernet speeds are ruling at 100 Mbps. Wi-Fi networks using 802.11g operate at around 20 Mbps.

WANs extend the range used by LANs. It can connect a number of cities or establishments in the same city. Because of the large distances, optic fiber cables, leased telephone lines and radio transmission are used to connect WAN nodes. WANs can also be formed by interconnecting LANs using *routers (1.15.3)*. Banks, airline and hotel reservation companies use WANs for their real-time operations. The Internet is the largest WAN in the world.

Technology advances have led to the birth of other types of networks. They are briefly discussed as follows:

- *Metropolitan Area Network (MAN)* This type is sandwiched between a LAN and a WAN. A MAN is employed for interconnecting computers in the same city.
- *Campus Area Network (CAN)* This is smaller than MAN and it is used to connect an entire university campus.
- *Personal Area Network (PAN)* This is the smallest network of all and has only recently come of age. A PAN operates within a range of a few meters. It connects small devices like cellphones and laptops with infrared or Bluetooth technology.

1.14.3 The Internet and internet

The Internet (also called *Net*) is the largest wide area network on the planet. It is a collection of over 50,000 networks that run a special set of network protocols called *TCP/IP*. Born in 1983 at the behest of the US Ministry of Defense, it is today a self-governing network without an owner. The Internet is different from other networks; it grows continuously without disturbing the existing structure. The Net truly shines in its plethora of services some of which are discussed in Chapter 2.

The hardware technology used by the Internet is no different from any LAN or WAN. For this reason, it is possible to create small networks of this type that can replicate its services, albeit on a much smaller scale. This network that runs the same set of network protocols is called an *internet* (with lowercase "l" or *intranet*. Depending on the way it is configured, an internet may or may not have access to the Internet. The addresses used by internet nodes are not valid on the Internet.

1.15 NETWORK HARDWARE

Connecting computers in a network require additional devices that are not part of the computer's basic configuration. Wireless connectivity may eventually lead to the phasing out of some of these technologies, but that is not likely to happen anytime soon. In the following paragraphs, we discuss the essential devices and components of a network.

1.15.1 Network Interface Card

Network transmission begins from the *Network Interface Card (NIC)* or *network card* that is housed inside the computer. Because most networks use Ethernet technology, this card is often referred to as *Ethernet card*. The card functionality is implemented in the following ways:

- Built into the motherboards of laptops and current desktops.
- As a separate card inserted into a slot on the motherboard.
- As a wireless card inserted in the same manner; needs no cables to connect to the network.
- As a USB adapter that runs on the USB port.

Every Ethernet card has an address hard-coded into the board by the card manufacturer. This address, known as *MAC* (Machine Access Control) address, is unique to the card. No two

Ethernet cards can have the same MAC address. The card is connected to a *hub* or *switch* via a cable which has an RJ45 connector at both ends. The devices have the corresponding (female) sockets for the RJ45 connector.

Note: Built-in cards are autodetected by the operating system which loads the appropriate software at the time of installation. External cards may need vendor-supplied software to be installed. USB adapters are also autodetected by the computer.

1.15.2 Hub and Switch

Computers in a single network and using the star topology need a central device to connect to, and this device is either a *hub* or a *switch*. This device contains a number of RJ45 ports that are connected by cables to the corresponding ports available in the nodes (Fig. 1.8). A *hub* accepts network data from a computer and simply broadcasts the data to all nodes. Since the data contain the MAC address of the destination, only the node having that address will accept the data. The technology is wasteful because data have to travel to all nodes even though only one among them will accept it.

A *switch* on the other hand is intelligent because it has a table of all MAC addresses of the connected devices. The switch extracts the MAC address from the data and then looks up the table to determine the node the packet is meant for. The packet is then sent only to that node. A switch is thus efficient and is able to maintain the speed of a network.

1.15.3 Bridge and Router

As electrical signals travel along a cable, they weaken, so Ethernet sets a limit on the maximum size of the cable (100 meters). Also, because generally only one device is able to transmit at a time, too many nodes can lead to network congestion. A network supporting many nodes must, therefore, be split into a number of *segments*, with a *bridge* connecting them. A bridge maintains a table of MAC addresses of all machines in the segments connected to it. It connects two networks using the same protocol (say, Ethernet). Bridges are on their way out because their functionality has been taken over by the switch.

FIGURE 1.8 A Hub and Switch

A *router* connects two similar or dissimilar networks which may be separated by long distances. It is a part of both networks and thus has two addresses. A router looks up its *routing table* to determine the path a packet has to travel to reach its destination. If the destination node is on the same network, then the router sends the packet directly to the node. Otherwise, it sends it to another router that takes it closer to its destination. A router is smarter than a bridge because it can choose the best route for guiding the packet. This routing principle is used on the Internet.

Note: Router functionality is often available in a switch. Also, the ADSL modem that is used for broadband connectivity is both a router and a hub.

Takeaway: A router doesn't give up when an address is not found in its routing table. It knows which router can provide a *better* solution. A bridge, on the other hand, can't handle a packet at all if it doesn't belong to a directly connected network.

WHAT NEXT?

A computer with excellent hardware is totally useless if there is no software to take advantage of this infrastructure. Both hardware and software need to cooperate with each other for a computer to deliver useful output. All programs, including C programs, belong to the category of software.

WHAT DID YOU LEARN?

A *computer* accurately processes instructions that act on data at high speeds. The modern digital computer uses the *binary* system and the *stored program* concept to maintain both program and data in memory.

Computers can be categorized into *generations* which used vacuum tubes, transistors, integrated circuits and microprocessors. Hardware changes went hand-in-hand with development of programming languages.

Computers can also be classified as supercomputers, mainframes and microcomputers. Consumer devices like smartphones and cars use *embedded* computers.

The central feature of a computer is the *Central Processing Unit (CPU)* which comprises the *Arithmetic and Logic Unit (ALU)* and *Control Unit (CU)*. Users interact with a computer using devices like the keyboard and monitor.

Running programs use *primary memory* which comprises RAM, ROM (non-volatile), cache memory and registers. *Secondary memory* is used as backup and includes the hard disk, magnetic tape, optical disks and flash memory.

Devices connect to the computer using *ports* and *sockets*. The USB port has become the standard port for most devices. HDMI is the defacto standard for audio-visual data.

Input devices include the keyboard, mouse and scanner. Output devices include the monitor and printer.

Multiple computers are connected in a network for sharing programs, data and devices. A network

uses *network cards*, *hubs*, *switches* and *routers*. The standard hardware protocol used is *Ethernet* and the standard network protocol is *TCP/IP*. The latter is also used on the Internet.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

1.1 1 MB is the same as 1 Mb.
1.2 A computer program is executed by loading its file and associated data to memory.
1.3 Machine code comprises only 0s and 1s.
1.4 Integrated circuits were first used in second-generation computers.
1.5 Molecular computing is a feature of fourth-generation computers.
1.6 The microprocessor contains the CPU in a single chip.
1.7 The minicomputer became obsolete because of the emergence of the local area network.
1.8 Android is the name of a type of processor used in mobile phones.
1.9 Primary memory is slower than secondary memory.
1.10 The computer's BIOS is a small program stored in a ROM.
1.11 Registers represent the slowest memory in the computer.
1.12 RAM is faster than a hard disk because it has no moving parts.
1.13 Flash memory is based on RAM.
1.14 The USB port is used to connect printers, scanners and mice.
1.15 Non-impact printers include dot-matrix and daisy-wheel printers.
1.16 Like the star topology, the ring topology also needs a hub.
1.17 The MAC address of the network interface card is unique throughout the world.
1.18 The Wi-Fi router uses the 802.11 protocol.
1.19 There is no difference between the Internet and internet.

A2. FILL IN THE BLANKS

1.1 A computer executes a set of _____ that operate on _____.
1.2 Computers run faster if both _____ and _____ are resident in memory.
1.3 _____ memory loses its contents when power to it is withdrawn.
1.4 Vacuum tubes were used in _____ generation computers.
1.5 Complete the sequence: kilobyte, megabytes, gigabytes, _____.
1.6 COBOL and Fortran replaced _____ for most applications.
1.7 The special program that controls the resources of the computer is called the _____ _____.

1.8 As component size decreases, _____ increases.

1.9 Artificial intelligence is a feature of _____ generation computers.

1.10 A microcomputer with a high resolution display and a powerful processor is called a _____.

1.11 Data in primary memory is accessed by the _____ of the cells. Data in secondary memory is stored in _____.

1.12 _____ memory stores those parts of a program that are frequently used.

1.13 Optical disks like the CD-ROM store binary data in _____ and _____.

1.14 A scanner uses the _____ facility to extract text from an image file.

1.15 The ink-jet printer is faster than a dot-matrix printer but slower than a _____ printer.

1.16 An entire network using the _____ topology will fail if the hub fails.

1.17 The _____ runs on TCP/IP technology.

A3. MULTIPLE-CHOICE QUESTIONS

1.1 A program that translates programmer's code to machine language is called a (A) translator, (B) compiler, (C) coder, (D) none of these.

1.2 Supercomputers are mainly used for (A) serving a large number of users, (B) heavy number crunching, (C) transaction processing, (D) all of these.

1.3 A byte generally comprises (A) 2 bits, (B) 4 bits, (C) 6 bits, (D) 8 bits.

1.4 In a 32-bit computer, (A) the CPU can handle 32 bits at a time, (B) the word size is 32 bits, (C) A and B, (D) none of these.

1.5 Computation in the CPU is performed by the (A) ALU, (B) CU, (C) registers, (D) none of these.

1.6 The motherboard houses only the following memory types: (A) RAM and ROM, (B) RAM and cache, (C) RAM, ROM and cache, (D) all of them plus registers.

1.7 One of the following memory types is non-volatile: (A) RAM, (B) cache, (C) hard disk, (D) registers.

1.8 HDMI ports can handle (A) HTML documents, (B) audio data, (C) video data, (D) B and C.

1.9 JPEG files contain (A) audio, (B) graphics, (C) video, (D) none of these.

1.10 To write 15GB of data to an optical disk, you need to use a (A) CD-ROM, (B), DVD-ROM, (C) Blu-ray disk, (D) any of these.

1.11 Ink is sprayed for printing in a (A) laser printer, (B) inkjet printer, (C) plotter, (D) none of these.

1.12 Select the odd device out: (A) switch, (B) hub, (C) plotter, (D) router, (E) bridge.

A4. MIX AND MATCH

1.1 Match each computer generation with its distinctive feature:

(A) first, (B) second, (C) third, (D) fourth, (E) fifth.

(1) microprocessor, (2) integrated circuit, (3) vacuum tube, (4) speech recognition, (5) transistor.

1.2 Match each port with its compatible device:

(A) parallel, (B) DVI, (C) RJ45, (D) USB, (E) PS/2, (F) HDMI.

(1) keyboard, (2) home theater, (3) printer, (4) LCD panel, (5) Ethernet card, (6) pen drive.

1.3 Arrange the devices in descending order of access speed (shown with asterisks):

(A) DVD-ROM, (B) hard disk, (C) register, (D) ROM/RAM.

(1) ****, (2) ***, (3) **, (4) *

1.4 Associate the memory devices with their attributes or applications:

(A) ROM, (B) EPROM, (C) EEPROM, (D) hard disk, (E) tape drive, (F) DVD-ROM.

(1) SD card, (2) laser beam, (3) sequential access, (4) BIOS, (5) UV radiation, (6) sectors.

1.5 Match the devices with their typical memory capacities:

(A) RAM, (B) cache, (C) hard disk, (D) tape drive, (E) Blu-ray disk, (F) DVD-ROM.

(1) 20 TB, (2) 4 TB, (3) 50 GB, (4) 4.7 GB, (5) 4 GB, (6) 32 KB.

1.6 Associate the devices with their attributes or components:

(A) scanner, (B) mouse, (C) monitor, (D) printer, (E) network card, (F) plotter.

(1) MAC address, (2) toner, (3) optical character recognition, (4) pens, (5) scroll wheel, (6) aspect ratio.

CONCEPT-BASED QUESTIONS

1.1 Explain the significance of the *stored-program* concept proposed by Turin and von Neumann.

1.2 Name three important features of fifth-generation computers.

1.3 Mainframes are used in areas that are not handled by minicomputers. Explain.

1.4 How is a smartphone different from an embedded computer?

1.5 Explain the functions of the different bus types used by the CPU.

1.6 Explain the *fetch-decode-execute* mechanism used by the CPU.

1.7 How do static and dynamic RAM differ? Which type is used for cache memory?

1.8 Why does the CPU need cache memory and registers when it can use RAM for its work?

1.9 Explain how data is organized in a hard disk.

1.10 Name five secondary storage devices along with their typical capacities.

1.11 Explain the advantages and disadvantages of magnetic tape compared to the hard disk.

1.12 What is the key difference between LCD and LED monitors?

1.13 Name two advantages of a laser printer compared to an inkjet printer.

1.14 Explain how a switch is more intelligent than a hub.

1.15 Explain the role of the router in a network.

2 Computer Software

WHAT TO LEARN

- Significance of *system software* and *application software*.
- Role of the *Basic Input Output System (BIOS)* when starting a computer.
- Managerial role of the *operating system (OS)*.
- Why operating systems need *device driver* software.
- Features and commands of MSDOS, Windows and UNIX/Linux operating systems.
- Working knowledge of the components of Microsoft Office—Word, Excel and Powerpoint.
- TCP/IP software technology that is used on the Internet.
- Network applications *Ftp*, *Telnet*, *Secure shell* and the *Web browser*.

2.1 WHY COMPUTERS NEED SOFTWARE

A computer is probably the only machine that can't do anything immediately after it has been built. If in doubt, switch on a PC that's just been assembled. The display freezes (after initially appearing to do something) simply because the CPU (the computer's brain) has not been specifically instructed to do any work. In contrast, a car is ready to be driven after assembly. A washing machine after purchase can be used straightaway. For a computer, it's the *software* that provides the directives for the CPU to work. The most powerful CPU is totally useless without software that can make it extremely useful.

The earliest ancestors of the computer (the calculator and slide rule, for instance) needed no software. Every instruction to the device was provided by the operator (a human). The beginnings of software can be traced to the instructions that were coded into punched cards and paper tape. But because only one instruction could be loaded onto memory at a time, computers were slow even though the hardware was capable of achieving higher speeds. It was only after Turing and von Neumann introduced the concept of the *stored program* that program and data could be stored in memory. That's the way computers use software today.

In the days of the first and second generation computers, software was used mainly to solve engineering or scientific problems. Subsequently, the application became a payroll or an inventory list. As machines got more powerful, the scope of software got even wider. First, the computer itself needed software for its own use—for managing the programs that run on its CPU. That's how the *operating system* was born. Second, use of software exploded to manipulate any signal that could be represented in the form of 0s and 1s. Today, software has invaded the healthcare and entertainment space and is heading to power driverless cars.

Only computers need software, right? No, any device having a microprocessor needs software that tells it what to do. The reason why we can use a washing machine immediately after its purchase is that the software is built into the machine's PROM or flash memory. Software comes preinstalled in the smartphone's "internal storage" (i.e., secondary storage). Unlike a washing machine, however, a smartphone allows software to be added, modified or removed. The same is true for computers.

2.2 SOFTWARE BASICS

Software is a collection of code that drives a computer to perform a related group of tasks. It comprises one or more programs, supported by *libraries* and *configuration files* that programs need to access. Software may reside in a ROM, RAM, secondary storage (the hard disk) or may be loaded from a network. Most software are modifiable or removable, but software embedded in a ROM or PROM is usually permanent. The term *software* has a wider meaning today; it is considered to be incomplete without the documentation.

Programs in software use a *language*. There are a number of standard programming languages to choose from—C, C++, Java, Python, to name a few. However, the *source code* that is created by a programmer using any of these languages is different from the *machine language* that actually runs on the CPU. A special category of software (called *compilers*) convert source code to machine code and it is the latter that is distributed by software vendors. We'll find out how Linux is different.

It is common for software to be developed for multiple *platforms* (operating systems, to be precise). Software developed for Windows 8 may run without modification on Windows 10, but it certainly won't run on Linux, even if both Windows and Linux run on identical hardware. Microsoft Office (or its open source alternative) and Mozilla Firefox (the program we use to access the World Wide Web) are available separately for Windows, Linux and Mac OS.

Software vendors generally retain the rights to their product. This makes it illegal to copy it. Also, they make only the machine code available but not the source code. *It is not possible to retrieve the source code by reverse-engineering the machine code.* These restrictions have led to the development of *open source software* which are completely free to use, copy and redistribute. Moreover, the source code is also available. Linux is free, and so is the Web browser (Firefox or Chrome).

Note: All open source software are free to be used and copied, but the reverse may not be true. The Android system that powers most smartphones is open source software and can be used freely. However, most of the "apps" that users download from Google Play are not open source even though they are free to be used.

2.3 SOFTWARE TYPES

Software gets down to work the moment you switch on the computer. Until the computer is shut down (if at all), a variety of software keep the computer busy. Some software run continuously without your knowing it while others are explicitly run by users. Computer software can be broadly divided into two types:

- *System software* This is the software run by the computer to manage the hardware connected to it. Users don't have much control over its functioning, but their own programs (application software) use the services offered by system software whenever required.
- *Application software* Software of this type relates to a specific application, say, one that makes hotel bookings or creates special effects in movies. The vast majority of programmers are engaged in developing application software.

These two types of software have vastly different characteristics and require different skills; it is rare for a programmer to be proficient in both. In this chapter, we examine the commonly used software of both categories. Some of them may not be available on your computer, so make sure you have them installed before you can use them.

2.3.1 System Software

A program typically needs to access the CPU, memory, hard disk, monitor and keyboard. If it acts alone, then it must know the details of these resources. Because the same program may also run on different hardware, how does one design an application that need not know the hardware it is running on? This is possible only if another software sits between the application and the hardware. This go-between is *system software*. Because of its direct access to the hardware, system software is able to provide the services needed by programs and users.

Most system software is run by the computer on startup, first from a ROM and then from the hard disk. The others are invoked when programs need them, say, for printing a report. *In some cases, a user may also directly use a program of this type.* System software include but are not restricted to the following:

- *Basic Input Output System (BIOS)* This is a small program that checks the hardware devices and peripherals at boot time and then loads the operating system.
- *Operating system* This is the central system software that manages both the hardware and programs running on the computer. Because all programs need its services, the operating system remains in memory as long as the computer is up.
- *Device driver* Every hardware needs a special software that knows how to handle it. Programs access a device driver by making a call to the operating system.
- *Compilers and associated programs* This category of system software is special because it is invoked by users. A programmer invokes a compiler program to convert the source code written by them to machine code.

When a program run by a user (application software) needs to access hardware (say, a printer), it makes a call to system software to get the job done. For this reason, the BIOS and operating system are

loaded into memory before users can run their programs. On an idle machine, it's not unusual for 1 GB of RAM to be occupied by the operating system even though CPU activity may be very low.

System software is a specialized domain that requires knowledge of hardware. Lower-level languages have this knowledge, the reason why many routines of system software are written in assembly language (a low-level language). A C programmer must also know assembly language to be able to write system software.

We will briefly examine the first three categories of system software in this chapter. Compilers, being a programmer's tool, belong to Chapter 4 where it is discussed in depth.

Takeaway: When a computer is booted, the BIOS residing in ROM makes an initial check of the computer's hardware. It then loads the operating system from disk. Subsequently, application software use the OS to access all hardware, the reason why the OS must always reside in memory.

Note: In general, system software is written by the vendor that makes the hardware.

2.3.2 Application Software

Software not directly connected with hardware but related to a specific real-life situation is known as *application software*. The vast majority of the software that we encounter every day belongs to this type. The smartphone industry even has a smart name for it; they call it "apps." The first application software was probably the payroll but today application software represents virtually every business activity:

- *Office software* When the desktop PC came into being, office software was one of its first applications. This category now comprises three separate applications—word processing, spreadsheet and presentations. Word processing almost single-handedly drove out the typewriter from the market. We will be discussing these three applications in this chapter.
- *Database software* This software package, commonly known as *Data Base Management System (DBMS)*, allows data having a uniform structure to be stored in a database. Each line of a database comprises multiple *fields* that may be added, modified and queried using simple instructions. Microsoft Office has a DBMS component (Access), but it is Oracle that dominates this space.
- *Communications software* Computer networking led to the development of software that allowed users to communicate with one another. Email had been popular for offline communication before Skype and Whatsapp moved in as online applications. Mail handling is discussed in Section 2.14.1.
- *Entertainment software* Home entertainment on the PC has also seen the rise of software related to gaming and multimedia. The VideoLAN (VLC) software serves as a one-stop shop for playing most audio and video formats. Gaming software remains extremely popular and challenging for software developers.

- *Anti-virus software* A *virus* is a small program designed by a person with malicious intent. A virus can cause your programs to misbehave or not work at all. It can even wipe out data from the hard disk. Anti-virus software is now an essential program to have on your computer.
- *Special-purpose software* Apart from the general-purpose software mentioned previously, application software are also available for desktop publishing, computer-aided design/manufacturing (CAD/CAM) and distance learning, to name a few.

Application software are not hardware-dependent, so programmers don't need to know the details of the machine that will run the software. While hardware manufacturers are also engaged in the development of system software for their products, they are not in the application software business. This domain attracts most programmers (but not necessarily the best talent).

Note: A virus is one of the components of *malware*, the name given to all malicious software. *Spyware* and *trojan horses* are other components of malware. Spyware steals your confidential information while a trojan horse harms the computer by appearing to look innocent. Anti-virus software are meant to handle malware but they don't always succeed.

2.4 BASIC INPUT OUTPUT SYSTEM (BIOS)

We begin our study of system software by examining the BIOS that resides in every personal computer. This *Basic Input Output System* is a small program (a *firmware*, actually) stored in EEPROM or flash memory on the computer's motherboard. The BIOS performs some basic hardware-related tasks that are just adequate to load the operating system. Here's what happens when a computer is powered on:

- The BIOS loads device information and settings stored in a *CMOS* (Complementary Metal Oxide Semiconductor) chip on the motherboard. For enabling the CPU to access the basic hardware, the BIOS also loads an elementary set of software that drives these devices (the *device drivers*).
- The BIOS then performs the *Power-On-Self-Test (POST)*. This test checks the connectivity of devices like the keyboard, display, hard disk, optical drives and the ports (device list obtained from CMOS). The BIOS also checks every byte of RAM.
- After successful completion of POST, the BIOS examines a list of bootable devices and then loads the operating system. The BIOS eventually hands over control to the operating system.

The CMOS settings (including date and time) can be changed by the user with the *BIOS setup menu*. The key that invokes this menu is flashed at boot time on the screen (usually *[F2]*, *[Del]* or *[Esc]*). If you have replaced the hard disk, use this menu to make the information available to the BIOS. Using this menu, you can also change the default boot sequence from, say, optical drive to hard disk. These settings are kept alive in the otherwise volatile CMOS by an onboard battery. The BIOS checks the status of the battery as well.

POST is entered only during a *cold boot*, i.e., when the system is powered on or the reset button is pressed. The system skips POST on a *warm boot* that occurs when the *[Ctrl][Alt][Del]* sequence

is pressed. If POST encounters a problem, it responds with a well-documented combination of beeps. The system halts at that moment and does not load the operating system.

After POST, the BIOS tries to locate the drive containing the operating system. It checks CMOS for the list of bootable drives and their search sequence. Following that sequence, the BIOS tries to locate the *boot loader* (a small program) in the first sector of every drive. If found, the BIOS runs the boot loader which eventually loads the operating system. This process is called *bootstrapping*, an allusion to the lifting of one's boots by its bootstraps.

We said that the BIOS loads *device drivers*, but what are they? A device driver is a system software that is associated with a device. System and application programs access and drive the device by invoking its driver. The BIOS needs to load an initial set of these drivers because the CPU must have access to the key devices to be able to load the operating system. However, modern operating systems eventually replace the drivers loaded by the BIOS with their own set of drivers.

The BIOS has its limitations of course. It doesn't allow booting from a network or from a drive whose capacity exceeds 2.2 TB. After dominating the personal computer for over 30 years, the BIOS is making way for the *Unified Extensible Firmware Interface (UEFI)* which overcomes these limitations. UEFI is hardly a basic system; it is a pseudo-operating system that can be used to back up files or access the Internet even before the OS has been loaded. Modern PCs support UEFI; your PC may already be having one.

Tip: If the computer doesn't load the operating system even after successfully clearing POST, check whether you have left a DVD or CD in the drive. Disks not containing an OS won't have the boot loader in the first sector, so the boot sequence is halted until you remove the disk and reboot the system.

Note: Changes made to the BIOS settings are actually made in the battery-powered CMOS RAM and not to the BIOS itself, which contains only the program. The BIOS is usually resident in EEPROM or flash memory, so the usual update restrictions apply *(1.9.2)*. A BIOS in flash memory or EEPROM is easily replaced but not one that resides in a ROM, where the entire ROM chip has to be replaced.

2.5 THE OPERATING SYSTEM (OS)

We have been constantly referring to this term as if our lives depended on it. At least a computer's life depends on the operating system because no hardware or software can do without it. An *operating system (OS)* is a system software that manages the computer's hardware to provide a complete and safe environment for running programs. It acts as an interface between programs and the hardware resources that these programs access (like memory, hard disk and printer).

If a computer had to run only one program, then it wouldn't need an operating system. All the code for accessing hardware could be built into the program itself. If the printer needed replacement, the program would have to be changed as well. It would then be impossible to accommodate a second program to run on the same computer. Fortunately, a computer runs multiple programs that need all resources to be shared and managed. The following represent the key features of an operating system:

- *CPU Management* The instructions provided in the program are executed by the CPU. The OS keeps track of the instruction that was last executed. This enables it to resume a program if it had to be taken out of the CPU before it completed execution.
- *Memory Management* The OS allocates memory for the program and loads the program to the allocated memory. After the program has completed execution, the OS cleans up the memory (and registers) and makes them available for the next program.
- *Device Management* If the program needs to access the hardware, it makes a call to the OS rather than attempt to do the job itself. For instance, if the program needs to print a file, the OS directs the device driver associated with the printer to operate the device.
- *File Management* Programs read and write files and the OS has to ensure that disk blocks are allocated to the program as and when needed. It maintains a record of all blocks used by a file so it can readily provide the data to a program that reads the file. Also, the OS has to free up these blocks when a file is deleted.

Because the OS directly handles the hardware, it must present a consistent view of devices to all programs. Application programmers can thus write programs with the confidence that they will continue to run even after the hardware has been replaced or upgraded. The confidence doesn't end here; a program must also run on other similar computers but with different hardware configurations. Operating systems alone can't handle this flexibility; they need *device drivers (2.6)* to bail them out.

In addition to these basic services, an OS provides a wide range of other services for the user. It offers standalone programs for creating files and directories, handling processes, copying files across a network and performing backups. These programs don't form the core of the operating system, but may be considered as additional services that benefit both users and programmers. The schematic view of an operating system is depicted in Figure 2.1.

2.5.1 How Users Interact with an Operating System

We know that programs make a call to the operating system whenever they need to access the hardware. But who asks the OS to run a program and how? We, the users do, using an interface called the *shell*. The primary job of the shell is to wait for user input and then invoke a program associated with the input. There are basically two forms of the shell—the Character User Interface (CUI) and the Graphical User Interface (GUI).

The *Character User Interface (CUI)* presents a screen with a *prompt* string beside a blinking cursor. To talk to the OS, you must key in the name of a program at this `$` or `C>` prompt and then press *[Enter]*:

`$ a.out` *[Enter]*	*UNIX/Linux systems*
`C> AOUT` *[Enter]*	*MSDOS/Windows systems*

The string `a.out` or `AOUT` is interpreted by the shell as a *command* to execute. This command is often an executable program. The shell looks for a file of this name in specified directories of the hard disk. If it finds the file, it directs the OS to run the program.

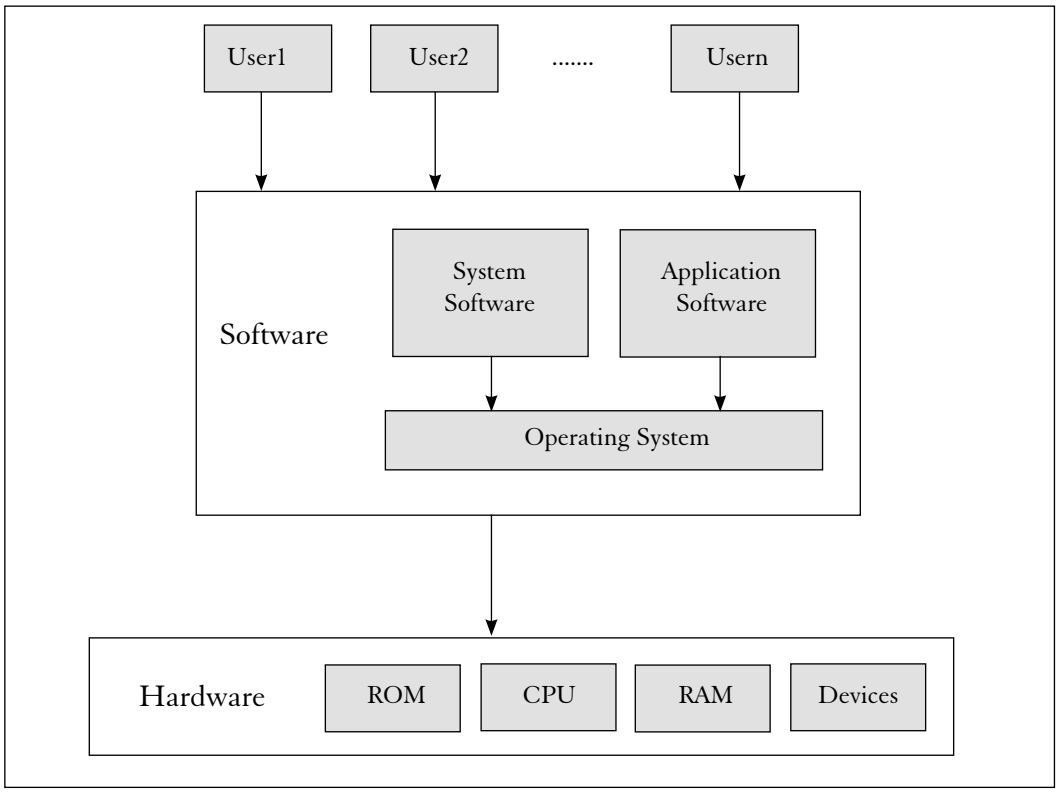


FIGURE 2.1 Role of the Operating System

The CUI is essentially a UNIX feature, but it is also available in MSDOS and Windows. In this book, the shell is variously referred to as *command line*, *command prompt* or *shell prompt*.

Operating systems also support a mouse-based *Graphical User Interface (GUI)*. In this environment, you click on an icon and the program associated with the icon is executed. The GUI was designed to keep the user ignorant of the existence of the OS beneath it. Windows and Apple are primarily GUI systems, but the GUI is an addon in UNIX/Linux systems. *Even though the CUI is less user-friendly, it is more powerful than the GUI.*

Note: The shell in Windows is the *Explorer* program. You can launch any program from Explorer by locating its file and then double-clicking on it. You can also copy, delete and remove files with your mouse. These actions would require separate commands to be invoked on a CUI system.

2.5.2 Classification of Operating Systems

As computer hardware evolved, operating systems have evolved as well to take advantage of the advancements made in processor technology. The first operating systems could only handle one program at a time, but today's computers can handle multiple programs run by multiple users. Operating systems are generally divided into the following categories:

- *Batch processing systems* This represents one of the earliest systems where programs were run in batches. Each program was allowed to run to completion before the next program was taken up. A batch processing OS is usually non-interactive, so all inputs have to be provided in the program itself.
- *Multi-programming systems* As the term implies, multiple programs scheduled for execution reside in memory but only one program can run at a time. A program is generally allowed to run its course until it completes or performs an I/O operation that doesn't need the CPU. The program may voluntarily give up the CPU or the OS may exercise this control itself.
- *Multi-tasking systems* As with multi-programming, multiple programs reside in memory, but an OS can run these programs *concurrently*. Using the concept of time-sharing, the OS scheduler allots small slices to each program to give the impression of concurrency. In this scheme, a program can still give up the CPU when doing an I/O-bound task and resume execution after completion of the task.
- *Multi-user systems* Modern systems like UNIX and Linux are multi-user systems where multiple users share the CPU and memory. Each user can also run multiple programs, so a multi-user system is also a multi-tasking system that uses the concept of time-sharing to share the CPU. Users log in to a multi-user system either using dumb terminals (obsolete today) or using a program named Telnet *(2.14.2)* or the Secure Shell *(2.14.4)* from their networked PCs.
- *Multi-processing systems* With the advent of multi-CPU machines, it is now possible to (i) *actually* run multiple programs, one on each CPU, (ii) run different parts of the *same* program on multiple processors. The programming language must at least support *multi-threading* but the OS has to take the final call on whether to use multiple processors at all.
- *Real-time systems* In all of the previous cases, even though the OS scheduler allocates to each job a fair share of the CPU, the response time is not guaranteed for any job. This limitation won't do for a program that replaces a car driver, or a process where an event must take place at the exact moment for an accident or damage to be prevented. Real-time systems provide a guaranteed response time.

It must be noted that on computers with a single CPU, only one program can run at any instant. The term *concurrency* must be understood in that context. In a multi-tasking or multi-user system, the OS provides the illusion of concurrency by allowing a program to run for a small instant of time, save its current state and then load the next program in the queue. This sequence is repeated in round-robin fashion until the first program is taken up again.

Note: The term *multi-programming* is often misunderstood and misinterpreted. Many authors use the term to include multi-tasking, multi-user and multi-processing. This term makes little sense today, so henceforth we won't use it in this book.

Caution: Don't be lured into thinking that a computer is *multi-user* simply because multiple users share its files or a connected printer. In a true multi-user system, a logged-in user *must* use the computer's CPU and RAM to run a program. This is not the case with many versions of Windows where multiple logins are permitted but only for sharing resources other than CPU and RAM.

2.5.3 The Current Operating Systems

There have been numerous operating systems in the past, one at least from each hardware vendor. They had virtually nothing in common which eventually led to a chaotic situation. Let alone programs, even data created on one machine could not be read on another without using proprietary conversion tools. Fortunately, the situation has been largely rectified by modern operating systems. These are the ones that are currently used or have recently been obsoleted:

- *UNIX/Linux systems* Originally developed at Bell Labs, UNIX is the earliest of the systems that are still in active use today. Because of its openness (the free availability of its source code) and standardization (the UNIX standard), Linux could emerge as a viable and powerful implementation of UNIX.
- *MSDOS* This OS from Microsoft started the PC revolution. It supports no GUI but only a CUI with a small set of programs that could be directly invoked by users. With the entry of mouse- and icon-based Windows, MSDOS was pushed out of frontal view on most PCs. Windows users can still access it though.
- *Windows* This is a fully GUI-based system from Microsoft that virtually replaced MSDOS. Because of the inherent advantages of using a high-resolution display, Windows supports a vast ocean of applications (like Word and Excel) that exploit this display. It is likely to remain the dominant OS on the PC for some time.
- *Mac OS* Apple created the first GUI-based operating system and their longest running series, the Mac OS, runs on the Macintosh. Today's versions (Mac OS X or MacOS) are actually a UNIX-like system with a GUI, and can be used in the same way as one would use Linux.

Programmers must have a working knowledge of the operating system they use for developing programs. Programs in every form—source code or machine code—are actually stored as *files*, and a programmer uses the tools supported by the OS to manipulate these files. Whether you use a CUI or GUI is a matter of choice, but hardcore programmers tend to prefer CUI-based systems like Linux.

Takeaway: The operating system works both for the user and the programs run by them. Users command the OS to run a program from the shell. After the OS has launched the program, it has to service all calls made by the program to access the hardware.

2.6 DEVICE DRIVERS

You have bought a new printer, connected the cables and powered it on. Your Windows system identifies it and asks whether you have a CD or DVD containing the software for the device. You insert the CD that was shipped with the printer, Windows loads the files from it and then flashes the message that installation is complete. You have just installed the device driver for the printer.

A *device driver* is a program that controls or drives a device attached to a computer. It signifies a software interface to a hardware device. The operating system or a program accesses the device by making a call to this interface (Fig. 2.2). The application programmer knows how to *make* this call

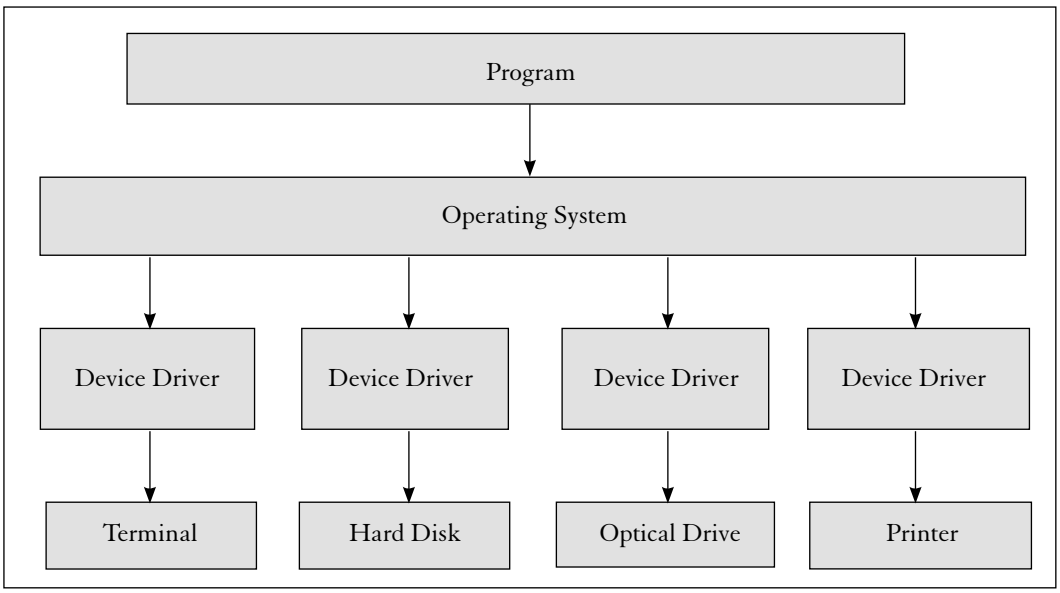


FIGURE 2.2 Role of the Device Driver

but the systems programmer knows how to *implement* the call in the driver software. Programming a device thus needs detailed knowledge of the device internals, the reason why device drivers are normally written by the device manufacturer.

Instead of having a separate program for driving a device, doesn't it make sense to include the code for it in the OS or the application? No, because if the device changes, the OS and all applications using it would need to be changed as well. The outcome would then be a programmer's nightmare. Applications must be independent of the type of hardware they access. Keeping the device driver separate from the OS and programs allows a driver to be shared between multiple programs.

That said, it is true that operating systems include built-in drivers for some commonly used devices. You don't need a separate driver for attaching flash memory. But even if the OS contains a driver for a device, you should install the (possibly) updated version that is shipped with the product. The newer version would contain improved features and fixes for bugs encountered in previous versions. If you can't locate the media containing the driver, get the driver software from the Web site of the device manufacturer.

Takeaway: Even though the operating system shields the programmer from knowledge of the hardware, it's the device driver that completely knows the hardware. The OS manages the call it makes to the driver and the response it receives from it.

Caution: A driver is meant for a specific model of the device and a specific version of the operating system. Installing a wrong driver may damage the hardware or make it behave improperly. It is also possible that the device may not work at all.

2.7 MSDOS: A CUI OPERATING SYSTEM

We begin our examination of the commonly used operating systems with MSDOS, the Microsoft Disk Operating System. MSDOS was used in the earliest personal computers and is now a part of Windows. The OS shipped with IBM's own PC was identical to MSDOS but was called PCDOS. In its initial years, MSDOS was loaded from a floppy diskette on a PC having two diskette drives but no hard disk. MSDOS is a single-user and single-tasking OS that needs only 640 KB of RAM. We must know this "outdated" OS because it gives us a first-hand feel of working with an OS.

The startup sequence loads the boot loader *(2.4)* located in the first sector of the hard disk, which then pulls up the basic OS by loading the files **IO.SYS**, **MSDOS.SYS** and **COMMAND.COM** into memory. The system eventually displays the prompt:

```
C:\>                                    Ready to accept commands
```

You are now in the MSDOS shell from where you can key in any of its commands or a program that you have written. You will see this shell prompt in all versions of Microsoft Windows, which offer a trimmed version of MSDOS (as **cmd.exe**).

Note: Most features of MSDOS are derived from UNIX.

2.7.1 Files and the File System

Every operating system needs a *file system* to hold all files and directories in a rigid structure. (A *directory* is a folder that contains multiple files.) Like UNIX, MSDOS supports a hierarchical file system as an inverted tree structure (Fig. 2.3). The top of this structure is called *root* (represented by \), which serves as the reference point for all files.

A file is uniquely identified by a *pathname* which specifies its location in the file system. For instance, the pathname **C:\PROGS\INTRO.C** locates the file **INTRO.C** in the directory **PROGS**, which in turn is under **** (root) of the `C:` drive or partition of the hard disk. The prompt `C:\>` shows this root directory.

Earlier versions of MSDOS supported an 8.3 file-naming convention which allowed a maximum length of 8 characters for the basic filename and 3 characters for the extension. Thus, **INTRO.C** and **AUTOEXEC.BAT** are valid filenames, but not **INTRODUCTION.C** or **INDEX.HTML**. However, these filenames are valid in later versions of MSDOS (Windows 95 onwards) that support 255-character filenames. MSDOS filenames are not case-sensitive, so you can't have two files named **foo** and **FOO** in the same directory.

MSDOS is always installed in the file system named `C:`. The names `A:` and `B:` are reserved for the two floppy drives that once resided on the PC. File systems created subsequently have the labels `E:`, `F:`, `G:` etc. (If the computer has an optical drive, it would be accessed as `D:`.) Thus, the pathname **C:\PROGS\INTRO.C** won't conflict with **E:\PROGS\INTRO.C**.

Note: We used uppercase for MSDOS commands and filenames simply because that was the way this OS was initially used. Versions of MSDOS built into the later versions of Windows use lowercase though.

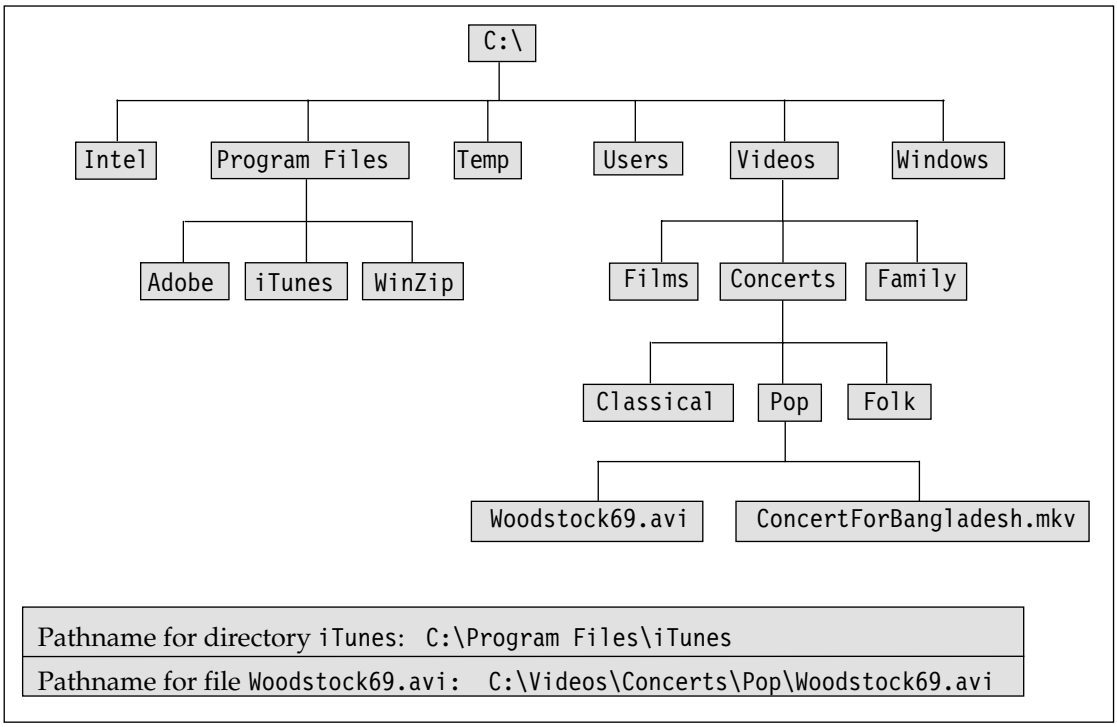


FIGURE 2.3 The MSDOS Hierarchical File System

2.7.2 Internal and External Commands

MSDOS supports a set of *internal commands* for basic operations and a larger set of *external commands* for advanced tasks (Table 2.1). Internal commands are built into the shell (the file **COMMAND.COM** or **cmd.exe** in later versions of Windows). For instance, the **DIR** command, which lists files, is an internal command of MSDOS. There's no file named **DIR.COM** or **DIR.EXE** in the file system. The commands that copy and remove files (**COPY** and **DEL**) are also internal.

The external commands are available as executable files having the `.EXE`, `.COM` or `.BAT` extension. Commands that format a disk partition (**FORMAT.COM**), back up files (**BACKUP.EXE**), or check the integrity of the disk (**CHKDSK.EXE**) are implemented as external commands. However, the location of the external commands must be known to the OS, so the `PATH` variable must be set accordingly.

The default behavior of a command can be changed by using it with *options* or *switches*. For instance, **DIR** can be used with the `/P` switch (as in **DIR /P**) for the output to pause at every page.

Takeaway: Internal commands are built into the shell. External commands exist as separate files which the shell locates using the information provided in the `PATH` variable.

Tip: Use the **/?** switch with any MSDOS command to learn about its usage. For instance, **MKDIR /?** displays the help screen for the **MKDIR** command.

TABLE 2.1 The External MSDOS Commands (Use *command* /? for details.)

Command	*Significance*
ATTRIB	Displays or changes the attributes of one or more files.
FIND	Displays line containing a specified string.
FORMAT	Reformats a device to create an empty file system.
MORE	Displays output one page at a time.
SORT	Orders a file in ASCII sequence but is case-insensitive.
TREE	Graphically displays directory structure with all subdirectories.
XCOPY	Copies an entire directory structure.

2.7.3 Working with Files and Directories

Like UNIX, MSDOS supports a powerful illusion that a user is *placed* at a specific point in the file system. You can know where you are, create and remove directories and navigate the file system without losing track of your location. Some of these commands take *arguments*, which are additional words that follow a command. The following internal commands are used to handle directories:

`MKDIR progs`	Makes a directory `progs`. (Also `MD progs`)
`RMDIR progs`	Removes directory `progs`. (Also `RD progs`)
`CHDIR`	Displays current directory. (Also `CD`)
`CHDIR progs`	Changes current directory to `progs`. (Also `CD progs`)
`DIR`	Lists contents of current directory.

Note from the description that MSDOS also offers synonyms for three commands: **`MD`**, **`RD`** and **`CD`**. **`MKDIR`** and **`RMDIR`** compulsorily need arguments but **`CHDIR`** and **`DIR`** can work both with and without arguments. Also keep in mind that **`RMDIR progs`** won't work unless the directory `progs` is empty and you are placed *above* it.

The preceding commands obviously won't work on ordinary files, which are handled by a separate set of internal commands:

`TYPE foo`	Displays the contents of `foo`.
`COPY foo bar`	Copies file `foo` to `bar`.
`DEL foo`	Deletes file `foo`. (Also `ERASE foo`)
`RENAME foo bar`	Renames `foo` to `bar`. (Also `REN foo bar`)
`MOVE foo bar`	Renames `foo` to `bar`.

There are two synonyms here as well (**`ERASE`** and **`REN`**). You need to be comfortable working with the last four commands because they can be used on multiple files.

The **`COPY`** command is special; apart from copying a file, it can be used to create a file and concatenate (combine) multiple files:

`COPY CON foo`	Creates a file `foo`. (Press *[Ctrl-z]* at end.)
`COPY foo1 + foo2 + foo3 foo4`	Concatenates three files to `foo4`.

We normally don't use **COPY CON** to create a file, but the command is quite convenient to create one by keying in a few lines of text. Don't forget to press *[Ctrl-z]* to signify end of input.

2.7.4 Using Wild-Cards with Files

MSDOS supports the use of *wild-cards*, a concept that is borrowed from UNIX. These are special symbols used to match multiple filenames. For instance, the expression `*.c` matches all files having the `.c` extension. The `?` matches a single character. A C programmer or Web site developer would find the following command lines quite useful:

`COPY *.c F:\progs`	Copies all C programs to the directory `progs` in the `F:` drive.
`REN *.html *.htm`	Renames all files with `.html` extension to `.htm`.
`MOVE *.exe backup_dir`	Moves all `.exe` files to the directory `backup_dir`.
`DEL *.*`	Removes all files in the current directory.

Can you do all of this with the Windows *File Explorer* program? No way except perhaps for the last one (deleting all files). The MSDOS shell, i.e., the command-line interface, is clearly superior here. The author has used the Linux and MSDOS command lines for compiling and executing all C programs of this book.

2.7.5 The Utilities

MSDOS also supports a number of utilities as internal commands. One of them clears the screen, but the others display important information of your computer system. Two of them (**DATE** and **TIME**) could change the date and time in the non-Windows implementations of MSDOS. Here are five utilities that are built into **COMMAND.COM**:

`CLS`	Clears the screen.
`VER`	Displays version of the operating system.
`DATE`	Displays and changes system date. (Change not permitted in Windows)
`TIME`	Displays and changes system time. (Change not permitted in Windows)
`PATH`	Displays path followed to locate executable programs.

The **PATH** command lists a set of directories that would be searched by the system for locating a command before invoking it. Another internal command, **PROMPT**, is only used with a string to change the default prompt setting. Here are two examples featuring their usage:

`PATH %PATH%; F:\C_PROGS`	Appends directory `F:\C_PROGS` to existing `PATH`.
`PROMPT $P$G`	Changes prompt to include drive name, pathname and `>`.

We examined only the internal commands here, but Table 2.1 lists some of the external ones. The ones selected for inclusion in the table are still relevant in spite of convenient alternatives available in GUI-based Windows.

Note: Even though we seldom use MSDOS today, every Windows system provides an "escape" to the MSDOS shell. The command **COMMAND** or **cmd** in the *Run* window takes you there.

2.8 WINDOWS: A GUI OPERATING SYSTEM

Windows is the most popular operating system in the world today. It shot into prominence with Version 3.0 as a GUI program. This program was invoked as the **`WIN`** command from MSDOS. The next major revision transformed the **`WIN`** command to an operating system, then known as Windows 95. Since then, there have been numerous revisions, including versions made for server operations (Windows NT). At the time of this writing, Windows 10 rules the market. We present here a generic discussion of Windows that relate to the later versions.

Like all GUIs, Windows was designed to hide the core OS that resides beneath the GUI. There are productivity gains in using the mouse to run programs rather than key in terse commands having difficult-to-remember syntaxes. For instance, you can delete a group of unrelated files faster in the GUI (using the mouse for selection) than in a CUI. However, the CUI-based command **`DEL *.c`**, which deletes all C programs, has no GUI equivalent. A GUI user still needs the CUI-based shell prompt for tasks like these.

If a user account has been set up for you, you need to log in using the user-id and password. Other users may also have separate accounts on the same computer, but *most versions of Windows don't permit concurrent logins that share the CPU and memory.* However, it's possible to start a download and play a piece of music while you browse the Web—all running concurrently. Windows is thus a single-user, multi-tasking system.

2.8.1 The Display and Mouse

A GUI like Windows relies on a display resolution better than the *VGA mode* (640 × 480 pixels). The resolution is typically 1366 × 768 pixels, which is lower than the HDTV minimum (1920 × 1080). The opening screen is called the *desktop* which has a *taskbar* at the bottom (Fig. 2.4). Programs are invoked by clicking on icons placed on the desktop and taskbar. The left-hand corner of the taskbar supports a *Start* menu where you'll find all installed programs.

When you click or double-click on an icon using the left mouse button, Windows invokes a `.exe` file associated with the icon. For instance, when you click on *File Explorer*, Windows runs **`explorer.exe`** located in `C:\Windows`. The left mouse button is also used to select (i) a menu item, (ii) a text section by dragging the mouse. Selected text can then be manipulated by using both buttons.

The right button activates a context-sensitive menu (Fig. 2.5). Many of the items in this menu are also available in the *menu bar* that features at the top of the window of most applications. Cut- or copy-paste operations are easily performed by using

- the left button to select text,
- the right button to copy (*[Ctrl-c]*) or cut (*[Ctrl-x]*) the text,
- the right button at a different location to paste the text (*[Ctrl-v]*).

The keyboard equivalents are shown in parentheses. Efficient mouse handling can result in significant productivity gains, so use the mouse right button wherever you can.

FIGURE 2.4 Windows Desktop

Tip: If you are using a laptop, change the desktop background to black to reduce battery drain. Avoid using screensavers, specially the ones that use a lot of colors.

2.8.2 File Explorer: Manipulating Files and Directories

This is the shell used by Windows to interact with the user for file-related operations. For the same tasks, we used multiple commands in MSDOS *(2.7.3)*. Upon invocation, Explorer displays a two-pane window showing all the drives used by the system (Fig. 2.6). There are two disk file systems (`C:` and `F:`), a DVD drive (`D:`) and an SD card (`E:`). You can access the entire file system of any of these drives either by selecting its icon in the left pane or double-clicking the icon in the right pane.

Windows uses the term *folder* to refer to directories. The right pane of Figure 2.7 shows the files and folders of **`C:\Windows`**. Note that this folder contains the files **`explorer.exe`** and **`notepad.exe`**. You can invoke these applications by double-clicking their respective icons.

When you right-click on a file or folder, the context-sensitive menu allows you to create, delete, rename and copy files and folders (Fig. 2.7). A file or folder can be moved by dragging its icon to the destination folder. It should not take you long to realize that Explorer is a one-stop shop for file manipulation tasks.

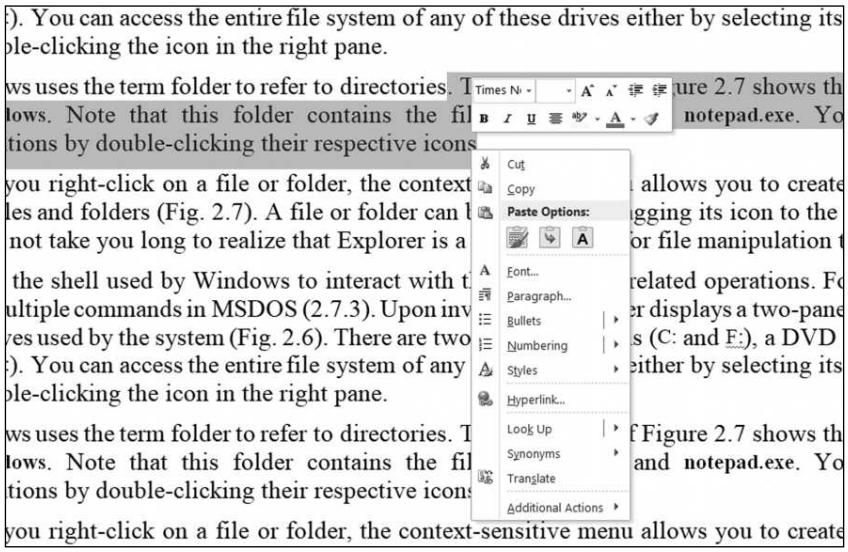


FIGURE 2.5 Context-sensitive Menu (Obtained by right-clicking)

If you have deleted a file and later regretted it, you can still retrieve the file using the *Recycle Bin* icon located on the desktop (or elsewhere). Simply select the file, right-click on it and then select *Restore* to bring it back to the file system.

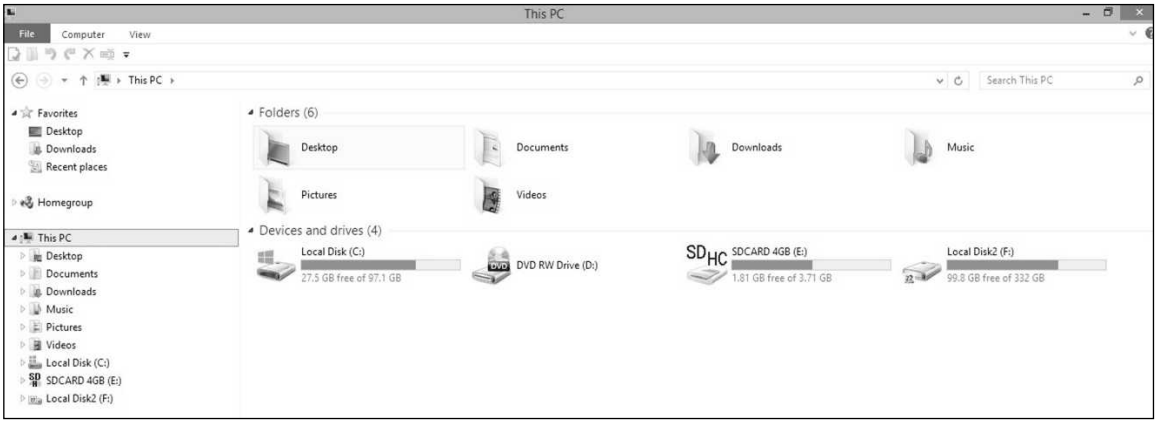


FIGURE 2.6 Windows Explorer (Overall view)

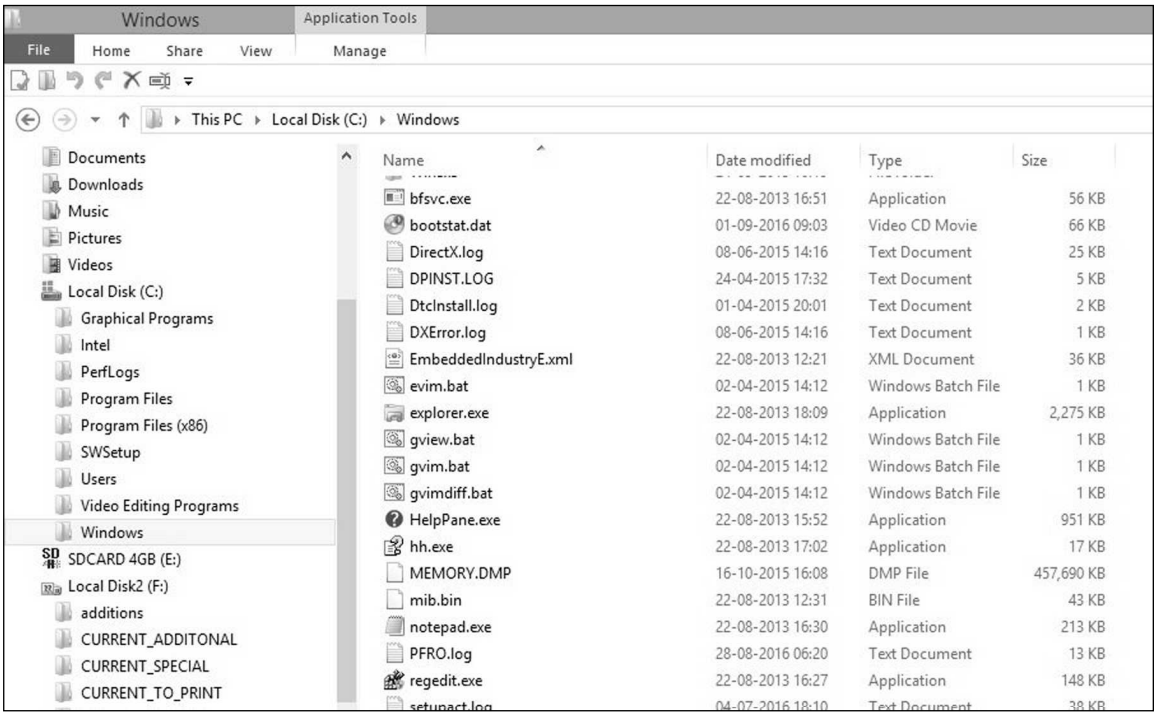


FIGURE 2.7 Windows Explorer (Detailed view)

2.8.3 Utilities and Administrative Tools

Windows supports a set of utilities and administrative tools that show up as a menu when you right-click on the *Start* button in the taskbar (Fig. 2.8). You need administrator privileges to use some of these programs. Your existing user-id may have the privilege or you may need to use a different user-id. Here are the major programs that you may need to use, specially when you are in trouble:

- *Programs and Features* You need this section to uninstall a program.
- *System* It displays the hardware configuration and version of Windows.
- *Network Connections* This section allows you to configure your network interface card, including setting the IP address and gateway.
- *Disk Management* This section allows you to create, delete and format disk partitions.
- *Command Prompt* This represents the MSDOS shell. A separate option creates the shell for use by the administrator.
- *Task Manager* This shows all processes running in the system. If an application freezes, you may have to *kill* it using this facility.
- *Control Panel* A feature available in every Windows implementation. It lets you configure some devices like the mouse, loudspeaker and microphone.

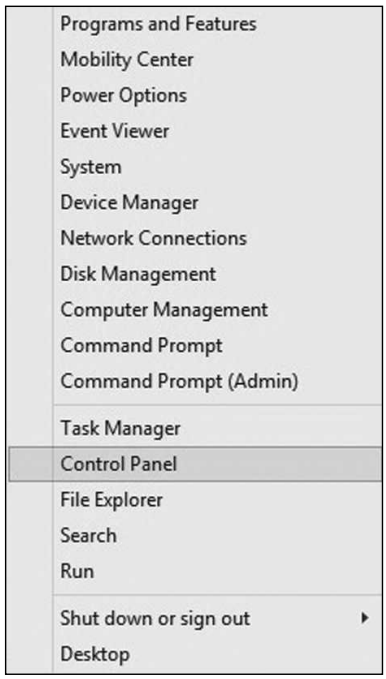


FIGURE 2.8 Start Menu

- *File Explorer* The Windows shell that allows manipulation of files, folders and their attributes.
- *Search* Locates a file or program in the system.
- *Run* Executes a program from the MSDOS command line; useful if you know the program name.
- *Shutdown* Shuts down the system or lets the user log out.

There's a lot that can be done from this *Start* menu, and you must explore them on your own. In the process, you'll discover the redundancies that are built into the system. The same program can often be accessed from multiple places. For instance, *Disk Management* can also be accessed from *Control Panel*, *File Explorer* and the desktop icon labeled *This PC*. By the time you master these features, the next version of Windows would have changed the scheme altogether.

Tip: You must activate the *Windows Update* feature if not already done by default. This allows the automatic installation of pieces of system software that improve the security features of the system, specially the ones related to malware.

2.9 UNIX/LINUX: A PROGRAMMER'S OPERATING SYSTEM

UNIX was created by Ken Thomson and Dennis Ritchie in the 1970s as a proprietary product of Bell Labs. Subsequent enhancements made by computer vendors, the University of California, Berkeley and the Linux community have today made UNIX a robust, powerful and reliable

operating system. Unlike MSDOS and Windows, UNIX is a *true* multi-user and multi-tasking system that runs on hardware ranging from PCs to supercomputers. UNIX/Linux systems form the backbone of the Internet.

The core of the OS is known as the *kernel*, which directly communicates with the hardware. Users interact with a shell (similar to the MSDOS shell) which acts as an interface between the user and kernel. When you enter a command through the keyboard, the shell examines the command line before forwarding it to the kernel for execution. If a program needs to access the hardware, it makes calls to the kernel. These calls (known as *system calls*) are special C functions that have been left out of ANSI C.

The UNIX utilities that handle files and processes are the finest in the business. Linux is a programmer's paradise; it supports compilers for *all* programming languages. UNIX was originally a CUI system where all programs or commands were executed at the shell prompt. The GUI, which was added to the system later, supports a CUI-based terminal window representing the shell. However, most of the command-line programs don't have GUI equivalents.

Tip: Linux users can switch to the CUI mode using *[Ctrl][Alt][F1]* and to the GUI mode using *[Ctrl][Alt][F7]*.

2.9.1 The UNIX File System

Files and directories are related to one another by being part of a file system. Even though MSDOS and Windows have emulated the UNIX file system, the following differences must be noted:

- The separator of a UNIX pathname is a / instead of \. The MSDOS pathname **`a\b\c`** becomes **`a/b/c`** in UNIX. This convention is also followed on the World Wide Web.
- There are no drive prefixes like `C:` and `D:` in the pathname.
- In UNIX, all files in *all* devices attached to a computer are part of a *single* file system with a single root (/ instead of \). So, the pathnames **`/home/sumit/progs`** and **`/home2/kanetkar/progs`** can be on two different disks.
- Unlike in MSDOS/Windows, UNIX filenames are case-sensitive, so **`foo`** and **`FOO`** are two different filenames that can coexist in the same directory.

The third point has far-reaching implications. Unlike with MSDOS/Windows, the UNIX file system integrates multiple disks and file systems into a single file system. These systems may reside on multiple connected machines—perhaps on two different continents! The user still sees a single file system.

2.9.2 Basic Directory and File Handling Commands

As you know well by now, files are containers for storing static information. But UNIX also considers directories and devices as files, and the same command sometimes works with multiple file types. By convention, UNIX commands are in lowercase and the following ones are used for handling directories:

`mkdir progs`	Makes a directory `progs`.
`rmdir progs`	Removes directory `progs`.
`cd`	Changes to the home directory (internal command).
`cd progs`	Changes current directory to `progs`.
`pwd`	Prints (displays) current directory (internal command).

Note that UNIX supports two internal commands for directory navigation, while MSDOS uses a single **`CD`** command (with and without arguments). A separate set of commands handles ordinary files:

`cat foo`	Displays the contents of file `foo`.
`cp foo bar`	Copies file `foo` to `bar`.
`rm foo`	Removes file `foo`.
`mv foo bar`	Renames (moves) file `foo` to `bar`.
`ls`	Lists files and subdirectories in current directory.

If these commands were designed to be used only in these trivial ways, nobody would be using UNIX. All of these commands support several options, and when they are combined with *wild-cards (2.7.4)*, the power of UNIX commands easily becomes apparent. Here are some examples of their advanced usage:

`cp -r foo1 foo2`	Copies an entire directory structure `foo1` to `foo2`.
`mv *.c progs`	Moves all files with `.c` extension to the `progs` directory.
`rm -r *`	The most dangerous command; removes everything in the current directory.
`ls -R`	Lists current directory structure including all subdirectories.
`rm *.????*`	Removes all files with an extension of at least 4 characters.
`cat foo[1-5] > foo6`	Concatenates files `foo1`, `foo2`, ... `foo5` and saves the output in `foo6`.

UNIX supports an elaborate system of wild-cards that make it possible to match a set of filenames with a simple pattern. The preceding examples gave us a glimpse of the usage of the wild-cards, `*`, `?` and `[]`. The wild-cards used by MSDOS represent a small subset of the UNIX set.

Note: The **`cd`** and **`pwd`** commands are internal; they are built into the shell. The shell itself is an external command, which, like **`COMMAND.COM`** of MSDOS, keeps running as long as the user is logged in.

2.9.3 File Ownership and Permissions

There are other file-related commands that you need to know as a programmer. The **`ls`** command which, by default, lists all filenames in the current directory, can be used with a number of options that reveal some of the file's attributes. Let's use **`ls -l`** before we examine the attributes:

```
$ ls -l unix2dos.c                                  $ is the UNIX shell prompt
-rwxr-xrwx 1 sumit users    297 2016-08-17 09:48 unix2dos.c
```

The eight-column output shows the date and time of last modification before the filename. The first column shows the file permissions, and the third and fourth columns show the file's owner and group owner. Ownership is an important issue that relates to file security, so we should know something about it.

Every file has an *owner*. By default, the owner is the creator of the file. A file also has a *group owner* which, by default, is the group to which the user belongs. The set of permissions shown in the previous output apply separately to the owner, group owner and *others*.

These permissions can be manipulated by the **chmod** command. The following command line removes the write permission for a file from the users that belong to the "others" category:

```
chmod o-w unix2dos.c                                    Two arguments
```

When you run the **ls -l** command a second time, you'll find that the first column has changed:

```
$ ls -l unix2dos.c                                      Two arguments; one of them is an option
-rwxr-xr-x 1 sumit users    297 2016-08-17 09:48 unix2dos.c
```

The string `rwx` that was previously seen at the end of the first column has now become `r-x`. You have just protected a file from being modified by "others." This category represents those users who are neither the owner nor the group owner of the file.

2.9.4 grep and find: The UNIX Search Commands

As a programmer, you'll need to locate programs that contain a specific word, say, a C function. The **grep** command locates both the line and file containing it. The following command line looks for the string `malloc` (a C function) in all C programs in the `progs` directory:

```
grep "malloc" progs/*.c
```

You may also need to locate the file itself by specifying its name and the directory from where to commence the search. This work is done by the **find** command. When used in the following way, **find** locates and displays all C programs by completely (recursively) examining the directory **/home/sumit/progs**:

```
find /home/sumit/progs -name "*.c" -print
```

Use the **man** command with the filename to explore the other options that you can use with these commands (**man grep** and **man find**). The possibilities are mind-boggling and it won't take you long to realize that Thomson and Ritchie wrote UNIX for the benefit of programmers.

Takeaway: File permissions apply to three categories of users: *owner*, *group owner* and *others*. The owner is the creator of the file and the group to which the owner belongs becomes the group owner. The rest are known as *others*. In general, others must not be allowed write permission on files and directories.

Tip: If you need to change the programs where a certain function has been used, use the command **grep -l** *name_of_function* ***.c** to obtain the list of filenames containing the function.

2.9.5 Other Utilities

The commands that we take up in this section relate to our UNIX system. When an external command is invoked, the OS creates a *process* for it. The **ps** command shows the processes running on your terminal:

```
$ ps
  PID TTY          TIME CMD
 3826 tty1     00:00:00 bash          The shell
 3990 tty1     00:00:11 vi            An editing program
14367 tty1     00:00:00 ps            The program run to get this output
```

The PID represents the process-id which is unique in the system. If a program misbehaves and you find it difficult to terminate it, then you may have to use the **kill** command.

The following commands also display information related to the system:

`date`	Displays system date and time. (Can be changed by administrator)
`uname -r`	Displays version of the operating system.

The **passwd** command changes the user's password. The system prompts you for the old password before it asks for the new one:

```
$ passwd
Changing password for sumit.
Old Password: *******
New Password: *******
Reenter New Password: *******          Old and new password keyed in
Password changed
```

UNIX is a multi-user system and you may like to know the users currently logged on to the system like you. The output of the **who** command shows three users sharing the machine's CPU and RAM:

```
$ who
romeo      console      May  9 09:31    (:0)
henry      pts/4        May  9 09:31    (:0.0)
steve      pts/5        May  9 09:32    (saturn.heavens.com)
```

How does the shell locate the files representing these commands? The answer lies in the `PATH` variable. The shell searches the list of directories specified in `PATH`. This variable will always have the directories **/bin** and **/usr/bin** because they contain the external UNIX commands meant for all users:

```
$ echo $PATH                                   echo is an internal command
/home/sumit/bin:/usr/local/bin:/usr/bin:/bin:.
```

This concludes our brief discussion on the UNIX operating system. You can try out all the commands on your Linux system if you are keen to use Linux for programming purposes. Linux is an automatic choice of course, but other considerations may lead you to choose Windows instead.

2.10 MICROSOFT WORD

In this and the next two sections, we'll discuss three components of an application software—Microsoft Office. We begin with Microsoft Word (or simply, Word), the application meant for word processing. The executable is named **`winword.exe`**, but you should be able to locate its icon in the *Apps* section of the desktop. Word files have the extension `.doc` or the newer `.docx`. When saving a Word file, keep in mind that an older version of Word may not be able to read a `.docx` file.

The empty window seen on invocation shows most of the software's essential features (Fig. 2.9). You can create a professional-quality document using the four rows of buttons, tabs and menu options. Some versions also feature a giant *Office* button located at the top-left hand corner. The exact frontal view is version-dependent, but the following components show up on selecting *Home* in the *menu bar*:

- *Office button* This button contains the important file-handling options that let you open, save, print and close a file (Fig. 2.10). If you don't find this button, look in the *File* menu of the menu bar for the same options.
- *Title bar* Located right at the top, this bar contains three buttons meant for saving a file, and undoing and redoing an editing action. You may also find a *Quick Access Toolbar*.

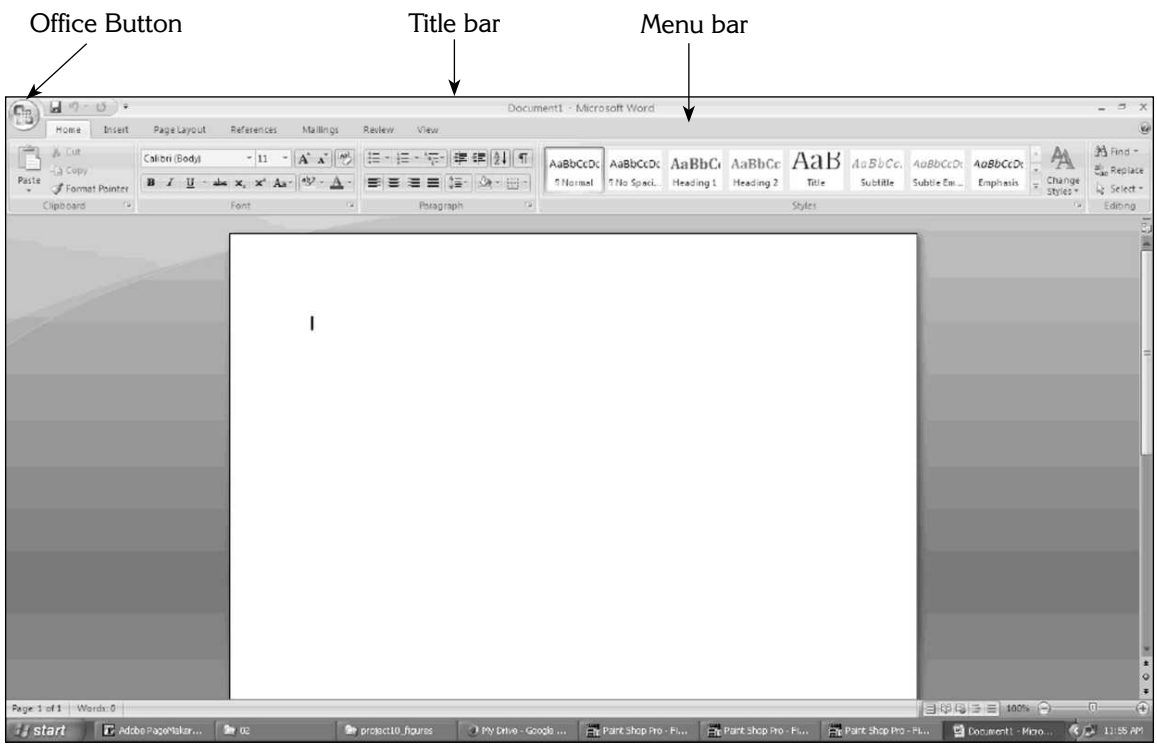


FIGURE 2.9 Microsoft Word: Opening Screen

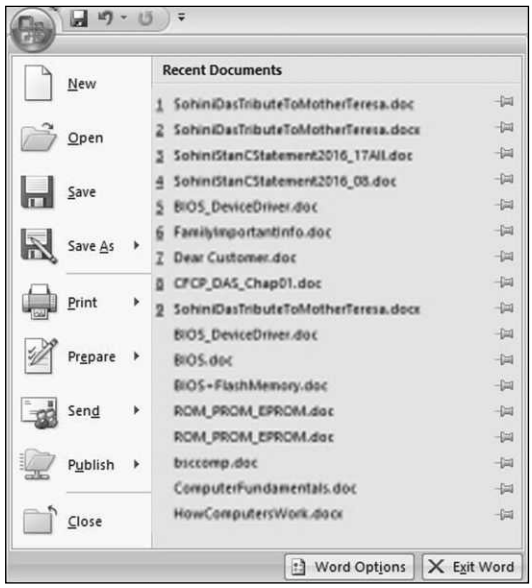


FIGURE 2.10 Office Button

- *Menu bar* This contains text descriptions of the options of the main menu. The left-most option is *Home* whose contents are shown in the two bars below it. When you click on, say, *Insert* or *Page Layout*, the contents in the lower bars change.
- *Formatting toolbars* Below the menu bar are two formatting toolbars containing the buttons needed for formatting a document. Older Word versions would have a single extended toolbar.
- *Status bar* This is located at the bottom of the window. It displays document statistics (like page and word count). On the right is a set of buttons that control the page view. We'll stick to the default *Print Layout* view signified by the first button.

Every item in the menu bar is associated with its own set of buttons and features. For instance, clicking *Insert* shows the buttons for inserting pictures, tables, headers and footers. Also, many of the menu bar items are common to all three Office programs (Word, Excel and Powerpoint).

2.10.1 Creating and Saving a Document

Let's start using this software by creating a new document. We'll key in some text, format it and save the created text as a file. The following steps are involved:

- Select *New* from the Office button or *File>New*. Choose a blank document which should now appear in view.
- Set the page properties before inputting text. Select *Page Layout* from the menu bar and then use the buttons below it to select *Portrait* (Orientation) and *A4* (Size). You are now ready to begin.
- Position the cursor on the blank window and key in text. To erase text, use the backspace key for removing text on the left or *[Delete]* for removing text on the right. You can also delete a text section by dragging the mouse over the text to select it and then use the same keys.

- Select the entire text either by dragging the mouse or using the *[Control-a]* shortcut. Select a font (say, Arial) from the *Format toolbar* and set the font size to, say, 10. Click anywhere on the window to deselect the text and observe the changes you have made. If you don't like the change, click on the undo button in the title bar.
- Save the file by using the *Save As* option from the Office button or *File>Save*. Windows shows the directory where the file would be saved by default. You should save your files in a separate directory that can be accessed easily. Locate the directory from the left pane and change the default filename to a meaningful one. Before you save the file, make sure the file type is `docx` or `doc`.
- Using the Office button, you can print this file using the *Print* option (or *File>Print*) and then close the file with *Close* (or *File>Close*). You can now close Word or open another file.

You have just created a file with some text in it. To reopen the file, use the *Open* option from the Office button (or *File>Open*). You may not see the file in the window, so select the right directory from the left pane and you should then see your file on the right pane.

Note: Office 2013 has replaced the large *Office button* with the *FILE* item in the menu bar. All file-related options are available in the *FILE* menu.

2.10.2 Handling Text

Cut-Copy-Paste Operations Any text-editing software supports facilities for cut-copy-paste operations. The key combinations used by Windows have been discussed in Section 2.8.1, but they also apply to all Windows software including the three components of Microsoft Office:

[Ctrl-c] for copying text

[Ctrl-x] for cutting text

[Ctrl-v] for pasting text

Word also offers alternative mechanisms of performing the above actions using (i) the right mouse button, (ii) the buttons on the left of the formatting toolbar. Right-clicking is generally the most convenient mechanism for these tasks as it involves minimum movement of the mouse.

Aligning and Indenting Text Properly aligned text enhances readability, and Word offers four buttons in the formatting toolbar for "justifying" text (Fig. 2.11). Three of these buttons align the selected text to the left, right and center. The fourth one fully justifies the text which makes it appear both left- and right-aligned. Left-justified text is the easiest to read, but fully-justified text looks cosmetically attractive. Aligned text can also be indented to the left or right by using their respective buttons.

Creating Lists Information is often presented in the form of lists that may be numbered or bulleted. There is a separate set of icons in the formatting toolbar for creating lists (Fig. 2.12). To convert an existing line to a list item, click on the line and then the respective icon to prefix the line with a number or a bullet. Once that is done, the next line automatically acquires the same property. You can terminate the list by pressing *[Enter]* twice.

FIGURE 2.11 Alignment and Indent Buttons

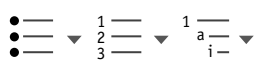


FIGURE 2.12 List Buttons

Paragraph Formatting Word lets you format one or more paragraphs with the *Paragraph dialog box* (Fig. 2.13). This box can be invoked either from the *Paragraph* tab below the formatting toolbar or by right-clicking on the paragraph. This dialog box has two tabs which have a number of useful settings.

Using the *Indents and Spacing* tab, you can adjust the spacing between both lines and paragraphs. You can also align the text and specify the indentation in numerical terms. The most important feature in the tab labeled *Line and Page Breaks* is *Widow/Orphan control* which controls the appearance of a paragraph when it overflows to the next page. This feature prevents

- the first line from appearing at the bottom of a page (orphan).
- the last line from appearing at the beginning of a page (widow).

You can also specify a page break before a paragraph in case you find that it would not be prudent to let a paragraph overflow to the next page. Explore the other options that relate to keeping lines together.

Search and Replace You'll often need to search a document for one or more occurrences of a string. Optionally, you may like to replace the string either selectively or globally. The search and

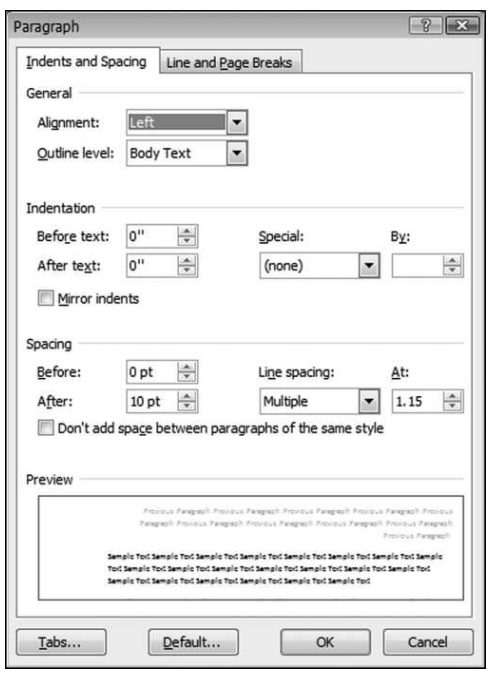


FIGURE 2.13 Paragraph Dialog Box

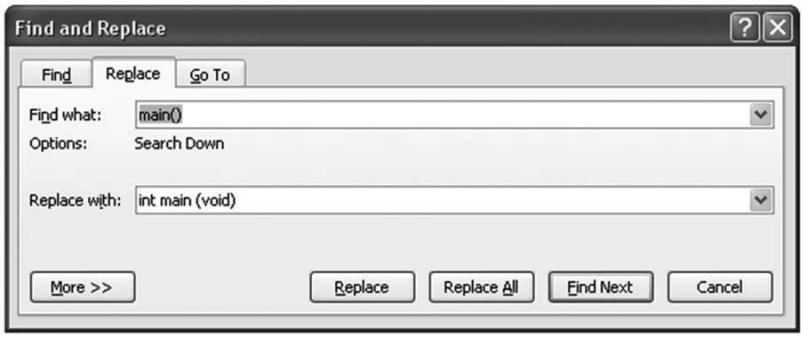


FIGURE 2.14 Find and Replace Box

replace facilities are available as two buttons on the extreme right of the toolbars. Selecting either of them opens a window with three tabs (Fig. 2.14). For simple searches, enter the search string in the window of the *Find* tab and then click on *Find Next* repeatedly to see each instance of the selected text in turn. The *Replace* tab shows an additional box to key in the replaced text.

Checking Spelling and Grammar After you have created a document, you need to check the spelling and grammar. You'll find the *Spelling & Grammar* button in the *Review* item of the menu bar. A correction window appears highlighting the word or phrase that Word doesn't find in its dictionary. A suggestion also appears in a separate window which you may accept or ignore. You may also add the text to the dictionary to let Word know that it has to ignore the text in subsequent encounters.

2.10.3 Font Handling

The Windows OS includes a number of fonts that are available to every application. You can install additional fonts using *Control Panel* of Windows. The font you use for a document must be available on every machine where the document is viewed. Otherwise, Word will select an alternative font which may cause the page to be displayed improperly. For this reason, it is safe to select the standard fonts like *Times Roman* and *Arial* which are available on every system.

Word lets you control font settings with the *Font dialog box* (Fig. 2.15). This box can be invoked from the *Font* tab below the formatting toolbar or by right-clicking on selected text. The top section lets you select the font, its style (normal, bold, italic or bold-italic) and size. By default, the text color is black but you can change it in this dialog box. The *Effects* section allows the use of subscripts and superscripts. You can also convert your text to uppercase in this section.

The second tab in the Font dialog box lets you control the space between characters. You may be forced to use this facility if a text segment has to fit in one line, and font size reduction is not an option.

2.10.4 Inserting Tables

Tabular information makes for good readability and Word supports elegant facilities for handling tables. You can draw a table or create one by providing its specifications. You can subsequently add or remove rows and columns and also adjust their width and height.

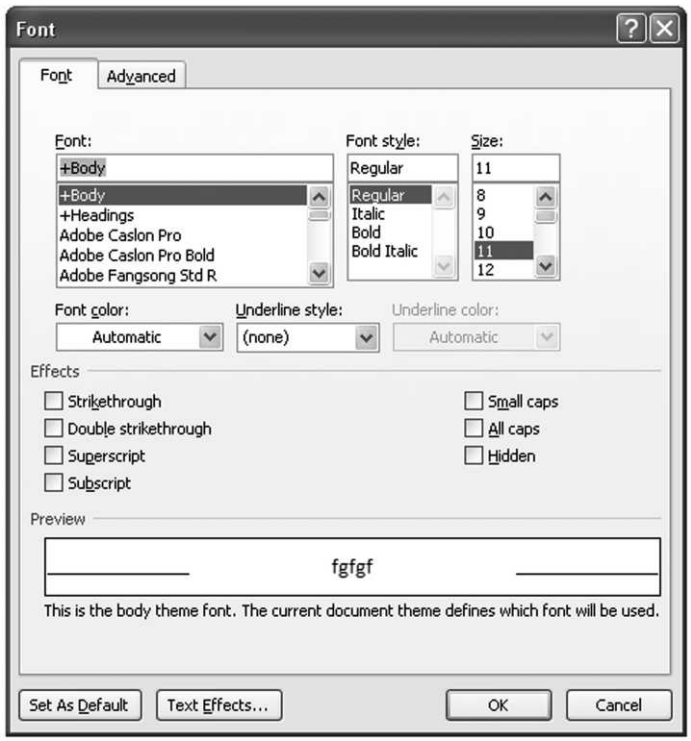


FIGURE 2.15 Font Dialog Box

You can insert a table using the *Insert* option in the menu bar and then selecting the *Table* or *Insert Table* button. A set of boxes show up to let you select the number of rows and columns. To create a table with, say, four rows and five columns, simply select the fifth box in the fourth row. A table is then created with equal spacing between rows and columns. You can now enter text in any cell and use the *[Tab]* key to navigate between columns.

You can change the column width by dragging a column to the left or right. If the row width can't be changed by dragging the row, right-click on the table. The menu that opens up lets you add or remove rows or columns and also change their properties. Select *Table Properties* (by right-clicking on the table) and then change the height of a row or width of a column.

If the default table settings are not good enough for you, you can draw the table yourself using the *Draw Table* option in *Insert Table*. The cursor changes to a pencil. Draw a box representing the table and then draw the horizontal and vertical lines by dragging the mouse from one side of the rectangle to its opposite one. The table thus created can now by further manipulated using the usual facilities that have just been discussed.

2.10.5 Inserting Graphics

Word is no longer confined to handling text. It can include pictures and provide rudimentary editing features of the inserted graphic. To insert a picture, use the *Insert* item of the menu bar and

click on the *Picture* button to open a window. Navigate to the directory and then select the graphic file to place in the document window. Word supports all the commonly used picture formats (like `.jpg`, `.gif` and `.png`).

If the picture needs to be edited, select it first and then click on the new *Format* option that shows up in the menu bar. You can change the brightness and contrast by using the buttons on the top-left of the document window. You can also crop and resize the image by selecting their respective options located at the top-right. Some of these options can be quickly accessed by right-clicking on the picture.

2.10.6 Other Useful Utilities

Using Mail-Merge This facility lets you create a document with static and variable parts. The variable part is represented as *variables* which are replaced with data saved in a list after mail-merge is run. For instance, if you want to send the same letter to multiple recipients, the senders' details could be held in an Excel spreadsheet. The main document has to be linked to this list. The mail-merge tool is available in the *Mailings* item in the menu bar.

Using Macros Macros are shortcuts to frequently used key sequences. After selecting text for which you want to create a macro, first select *Macros* from the *View* item in the menu bar. Initiate recording of the macro by providing it with a name and placing it in the title bar. After the selected text has been processed and the recording has been stopped, you can then repeat the set of recorded actions on any text by clicking on the icon that was created.

Converting to HTML & PDF A `.doc` or `.docx` Word file can be converted to HTML, the format used by documents on the World Wide Web. You simply have to select the file type as `.htm` or `.html` when using *Save As* from the Office button or *File* menu. You can also create an Adobe Acrobat PDF file using the *Acrobat* item on the menu bar. Both HTML and PDF files are smaller than Word files and are thus suitable for use on the Web. A Web browser that is meant to read HTML files can also handle PDF using a special software called a *plugin*.

2.11 MICROSOFT EXCEL

A *spreadsheet* is a document that stores data in tabular form. Microsoft Excel, another component of Microsoft Office, is meant to handle spreadsheets. The program has the name **`excel.exe`** but its icon is easily located on the desktop. Excel treats a spreadsheet file as a *workbook* comprising multiple *worksheets*. Each worksheet is a collection of *cells* arranged as a set of rows and columns. An Excel file has the `.xlsx` or `.xls` (older) extension. Older versions of Excel can't read `.xlsx` files but newer versions can read `.xls` files.

When Excel is invoked, an empty workbook is displayed on the screen (Fig. 2.16). Like in Word, a set of toolbars are seen at the top with many identical buttons. You may not see the large *Office* button, in which case you'll have to work with the *FILE* item in the menu bar. Since the significance of these buttons have been discussed in Section 2.10, we'll mainly focus on the Excel-specific features.

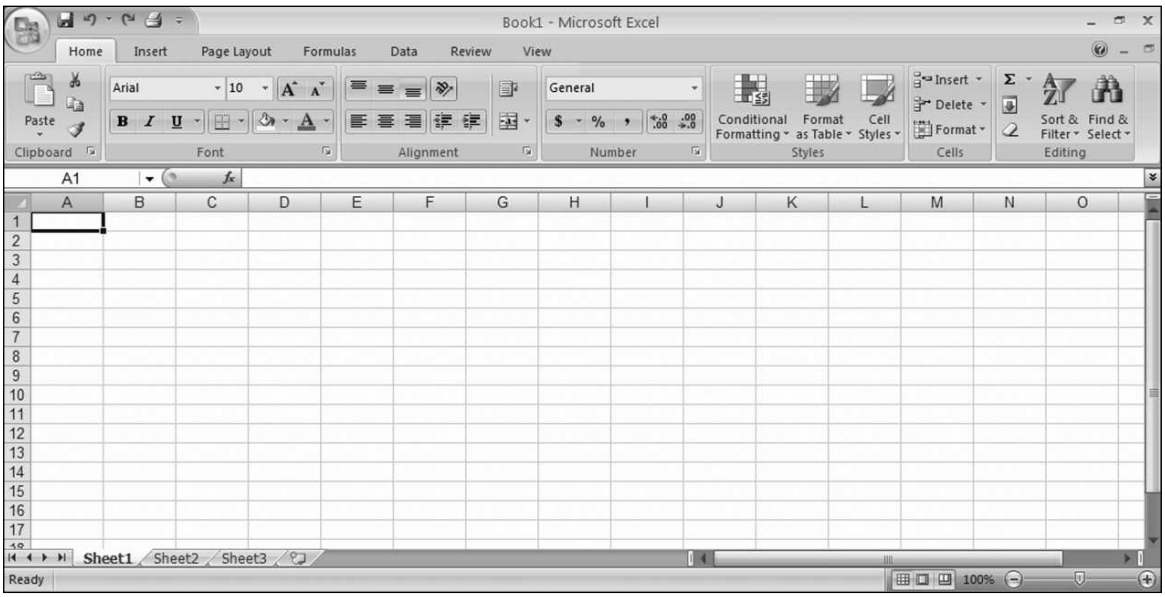


FIGURE 2.16 Microsoft Excel: Opening Screen

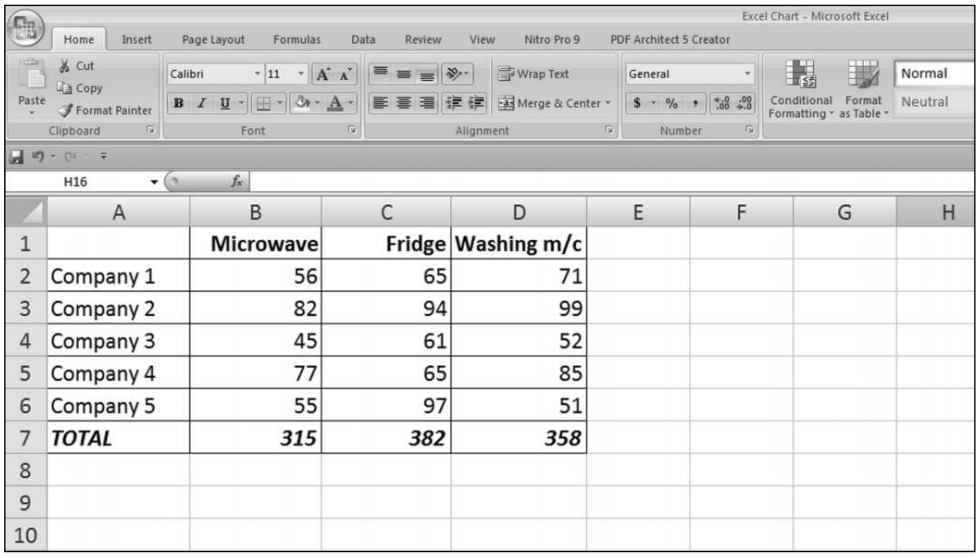
2.11.1 Creating, Saving and Printing a Spreadsheet

Use the *New* option in the Office button (or *File>New*) to select a blank workbook. This workbook shows up as an *apparently* endless stream of identifiable cells formed by numbered rows and alphabetically labeled columns. For instance, D3 represents the cell in the third row and fourth column. The entire worksheet is known by the name *Sheet1*, the name of the first of the three tabs located at the bottom. Excel creates a workbook with three worksheets, but you can add more as and when you need. To add, delete or move a worksheet, right-click on the worksheet tab.

Let's now create a table of data with headers (Fig. 2.17). Use the cursor control keys for navigation as you enter data in each cell. Note that text is left-aligned and a number is right-aligned in a cell. This is an established norm, but you can change the default. We will progressively add features to this sheet, but for now, let's save the keyed-in information. Select *Save As* from the Office button (or *File>Save As*) to save the spreadsheet as an Excel workbook with the `.xls` or `.xlsx` extension. As discussed before *(2.10.1)*, choose a suitable directory so that you have no problems in locating the file later.

This is a small worksheet, so it can be printed straightaway by selecting *Print* from the Office button (or *File>Print*). As a worksheet expands, its width may exceed the normal A4 page width, in which case you may have to change the orientation from the default *Portrait* to *Landscape* in the *Print* menu. If that doesn't help, then print selected portions of the worksheet by activating the *Selection* radio button in the *Print* dialog box.

We have not closed the file yet, so let's select *Close* from the Office button (or *File>Close*) to close the file. To reopen this file for editing, we need to use the *Open* option this time.



	A	B	C	D	E	F	G	H
1		Microwave	Fridge	Washing m/c				
2	Company 1	56	65	71				
3	Company 2	82	94	99				
4	Company 3	45	61	52				
5	Company 4	77	65	85				
6	Company 5	55	97	51				
7	*TOTAL*	*315*	*382*	*358*				
8								
9								
10								

FIGURE 2.17 Adding Data

Tip: Avoid paper wastage by previewing the selected section of the worksheet before you print it. Use *Print Preview* and activate the margins (*Show Margins*). If necessary, drag the margins to prevent overflow and to control the position of the headers.

2.11.2 Formatting Cells

Both the form and content of each cell can be formatted using the *Cells* section of the home screen (Fig. 2.18). The first two buttons let you add or remove individual cells or entire rows and columns. The third button supports an extensive set of cell formatting capabilities that let you do the following:

- Change cell dimensions by setting the height of rows or width of columns. The changes made apply to the entire selected row or column of the current worksheet.
- Change the appearance of the cell itself and its contents using a separate dialog box labeled *Format Cells*. This option is also available on right-clicking one or more cells.

Let's briefly examine the features supported by the six tabs of the Format Cells dialog box (Fig. 2.19):

- *Number* This option formats numeric data by specifying the number of decimal places, the % symbol and currency symbol. If you want to sort dates, use the *Date* option to set their cells to this type.
- *Alignment* By default, text is left-aligned and numbers are right-aligned but you can change these defaults here. Long text also overflows to adjacent cells but you can wrap it around to fit in one cell by checking the box labeled *Wrap text*.

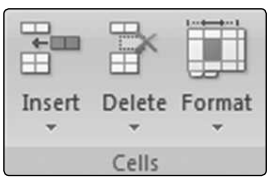


FIGURE 2.18 Cells Section

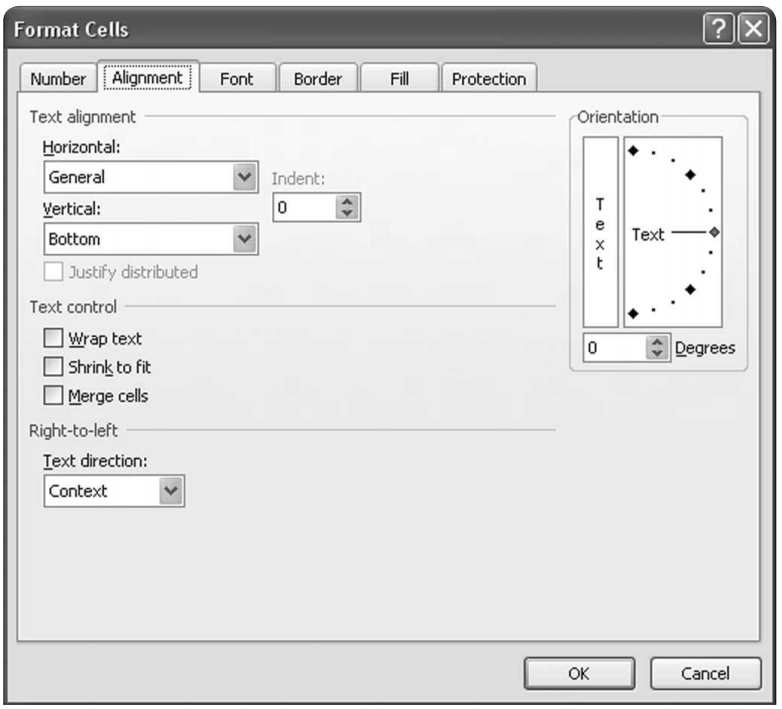


FIGURE 2.19 Format Cells Dialog Box

- *Font* There is no difference in the font-handling mechanism used by Word *(2.10.3)* and Excel, so they are not discussed here. As in Word, many of these features (like bold, italics and font color) are available in the Home screen.
- *Border* Cells that need special attention can be highlighted by providing them with borders. A cell may be fully or partially enclosed by borders and their style and color can be set here.
- *Fill* This option lets you specify the background color of the cell. This color must be carefully chosen so as to enhance readability without causing eye strain.

If multiple contiguous cells need the same treatment, select the range of cells by first clicking on one end of the range and then clicking on the other end with the *[Shift]* key pressed. If the cells are non-contiguous, use the *[Ctrl]* key instead.

Tip: For enhancing readability of large numbers, use the comma separators offered by the *Number* tab.

2.11.3 Computing and Using Formulas

Apart from numbers, Excel can handle expressions and *formulas* (actually, *functions* in programming parlance). Mathematical expressions support the standard arithmetic operators (`+`, `-`, `*`, `/`, `%` and `^`). The expression `1200 * 5 / 12` is valid in Excel.

Operations like summing or averaging a group of numbers are available as formulas. Excel supports the use of `SUM(3+5+7)` in addition to the expression `3 + 5 + 7`. Before you start using them, you need to know the following:

- All expressions and formulas must be prefixed with the `=` symbol. You must use `=1200 * 5 / 12` and `=SUM(12, 14, 15)`, otherwise they would be treated as simple strings.
- An expression or formula can include a cell address. The expression `=A1 + A2 + A3` computes the sum of the contents of three cells. *This also means that the contents of the cell containing this expression will change automatically if any of its three components change.*
- A contiguous set of cells can be specified using the colon (`:`). So, `=AVERAGE(D5:D8)` computes the average of the cells D5 through D8.
- If the cells are non-contiguous, use the comma (`,`) instead. Thus, `=MAX(A5,H7,K4)` evaluates to the maximum of the numbers contained in the three cells.
- The normal precedence rules apply for arithmetic expressions. This means that multiplication takes place first, followed by division, etc. Parentheses must be used to override the default. The expression `=(K3 - 32) * 5 / 9`, which converts Fahrenheit to Celsius, needs the parentheses to work correctly.

Sometimes, directly keying in cell addresses could be tedious, so use the mouse to select cells. Keep the *[Ctrl]* key pressed and click on the cells separately. A glance at the formulas window shows the progressive formation of the expression that Excel would eventually place in the cell.

Apart from the general-purpose formulas (Table 2.2), Excel supports a host of scientific, engineering and financial formulas. You can view the complete list by selecting the *Formulas* item in the menu bar.

Table 2.2 General-Purpose Formulas Supported by Excel

Formula	*Significance*
`AVERAGE(`*n1*`,` *n2*`, ....)`	Computes average.
`COUNT(`*n1*`,` *n2*`, ...)`	Computes number of cells.
`DATE(`*y*`,` *m*`,` *d*`)`	Prints date as *mm-dd-yyyy*.
`MAX(`*n1*`,` *n2*`, ...)`	Selects maximum of numbers.
`MIN(`*n1*`,` *n2*`, ...)`	Selects minimum of numbers.
`SUM(`*n1*`,` *n2*`, ...)`	Computes total of numbers.
`TIME(`*h*`,` *m*`,` *s*`)`	Prints time as *hh:mm* `AM/PM`.
`TODAY()`	Prints today's date as *dd-mm-yyyy*.

2.11.4 Handling Data

Standard Cut-Copy-Paste Operations A single cell or a range of cells can be copied using the standard techniques *(2.10.2)* provided the cells contain simple values—without formulas and cell references. For instance, these techniques won't permit you to copy the formula `=SUM(G40:G42)` and have its *value* pasted at the destination.

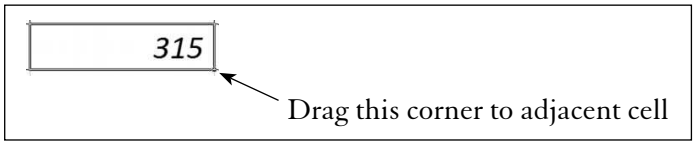


FIGURE 2.20 Copying a Cell

For copying a single cell to its adjacent location, use a shortcut: simply drag the handle at the lower-right corner of the copied cell to one or more adjacent cells (Fig. 2.20). A single value is thus copied to multiple cells.

When copying a range of such "simple" cells, you need to specify a single cell at the destination that signifies the top-left corner of the copied cells. The shortcut technique referred to previously applies but numbers need to be treated carefully. For instance, a row of three numbers can be dragged vertically but not horizontally. A converse reasoning applies to a column of numbers. If you drag a row of numeric cells horizontally, Excel thinks you are trying to complete a sequence and it fills up the blank cells with other values—not the copied ones.

Copying Expressions and Formulas Excel supports *Paste Special* that takes care of cell-reference and formula-related issues (Fig. 2.21). Two features should be noted. First, you can create a *link* between two cells. For instance, you may like to view the contents of a distant cell, say J200, at the top of the worksheet, say A20. Copy the cell J200 in the usual manner, but paste it at A20 using *Paste Link* from the *Paste Special* dialog box. J200 is now linked to A20, which means that changing J200 also changes A20.

The other feature relates to copying a formula. Consider that you have three columns of numbers and one of them has been summed with the `SUM` formula (Fig. 2.22). When you copy this cell

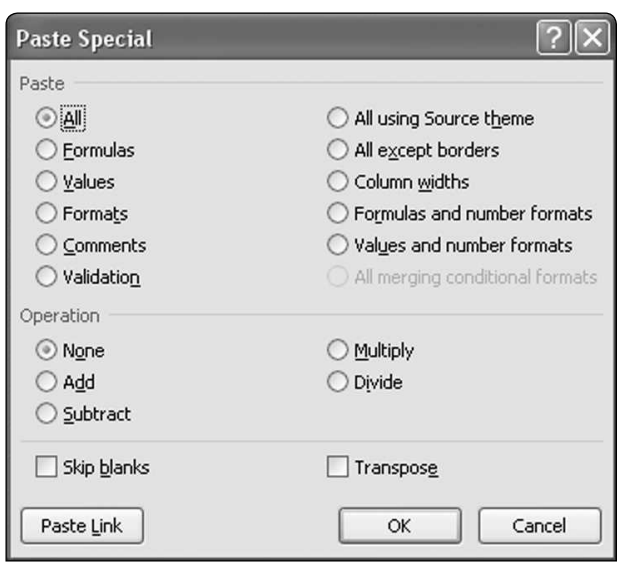
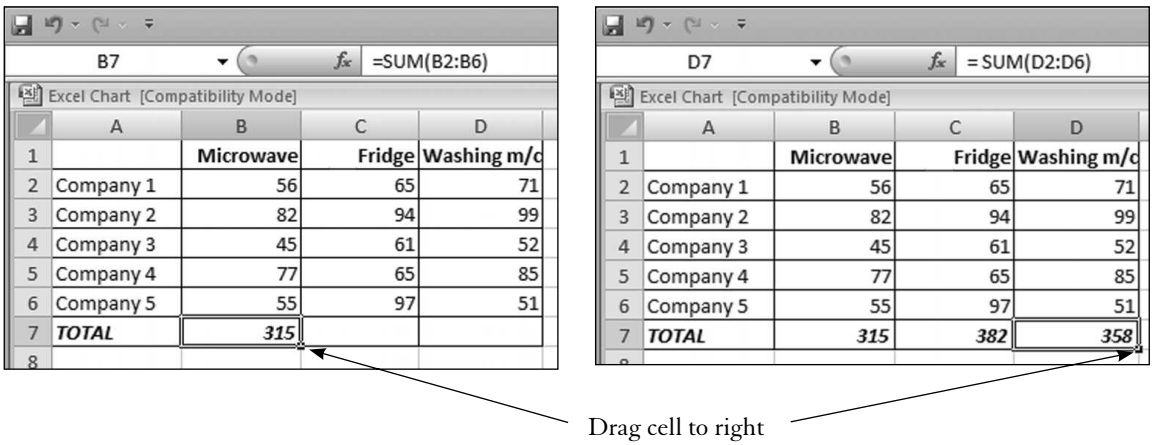


FIGURE 2.21 Paste Special Dialog Box



	A	B	C	D
1		Microwave	Fridge	Washing m/c
2	Company 1	56	65	71
3	Company 2	82	94	99
4	Company 3	45	61	52
5	Company 4	77	65	85
6	Company 5	55	97	51
7	*TOTAL*	*315*		
8				

	A	B	C	D
1		Microwave	Fridge	Washing m/c
2	Company 1	56	65	71
3	Company 2	82	94	99
4	Company 3	45	61	52
5	Company 4	77	65	85
6	Company 5	55	97	51
7	*TOTAL*	*315*	*382*	*358*

FIGURE 2.22 Copying a Formula

containing the formula to its adjacent columns, the data in those columns get summed automatically. In other words, if the formula `=SUM(B2:B6)` is copied from B7 to C7 and D7, Excel pastes the formulas `=SUM(C2:C6)` and `=SUM(D2:D6)` at `C7` and `D7`, respectively. This is normally what we want, but sometimes, you may like to paste the *value* of `=SUM(B2:B6)` rather than the formula. In that case, click on the *Values* radio button at the top of the *Paste Special* dialog box.

Tip: To copy a formula from one location to one or more adjacent locations, click on the lower-right handle of the cell containing the formula and then drag the mouse pointer to the other cells. The formula gets appropriately revised to reflect the new locations.

Filling Cells Excel has an intelligent feature that tries to complete a series of numbers or dates whenever it can. Consider that two adjacent row cells contain the integers 4 and 8. After selecting these two cells and dragging the handle of the right-most cell to, say, three blank cells on the right, these cells have the values 12, 16 and 20 filled in automatically.

Excel treats a date type a little differently. Just drag a single cell right or down; the date gets incremented by one. Drag it left or up, the date gets decremented by one.

Note: The column containing the strings `Company1`, `Company2`, and so forth, in Figure 2.22 was completed by dragging down the first cell. Excel is smart enough to guess your intention even when it sees text data that contains a numeric suffix.

The *Fill* button in the Home screen contains options to set the increment or decrement value. Select *Series* from the drop-down menu and and set the *Linear* button and specify the desired step value. A negative value creates a descending progression of linear values. You can explore the other options on your own.

Sorting Data Sorting is the ordering of data according to certain rules. Excel supports sorting on columns. Text data are sorted in the *ASCII sequence* (Appendix B), where numbers occur first, followed by uppercase and then lowercase letters. Numeric data are sorted by their value. Select the *Sort & Filter* button and choose the direction of sort. You can also sort items of the date type provided they have been formatted as such.

2.11.5 Creating Charts

A picture is worth a thousand words, we are told. This age-old saying applies perfectly to Excel which supports extensive charting tools for graphical depiction of data. Space constraints don't permit discussion on most of these tools, but we'll examine three of them that have withstood the test of time: pie charts, column charts and bar charts. Figure 2.23 depicts all three of them on a single worksheet with the related data placed at the top. The charting tools are available in the *Insert* function of the menu bar.

We use a *pie chart* for making a relative comparison of the elements of a single set of data. The pie chart in the figure shows how five companies share the microwave market. It was created by (i) selecting the columns L and M, (ii) choosing *Pie* from the *Insert* section. After the chart is created, it can be resized by dragging its handles and moved by dragging a blank portion. Right-clicking on a chart lets you add labels like values in each pie segment.

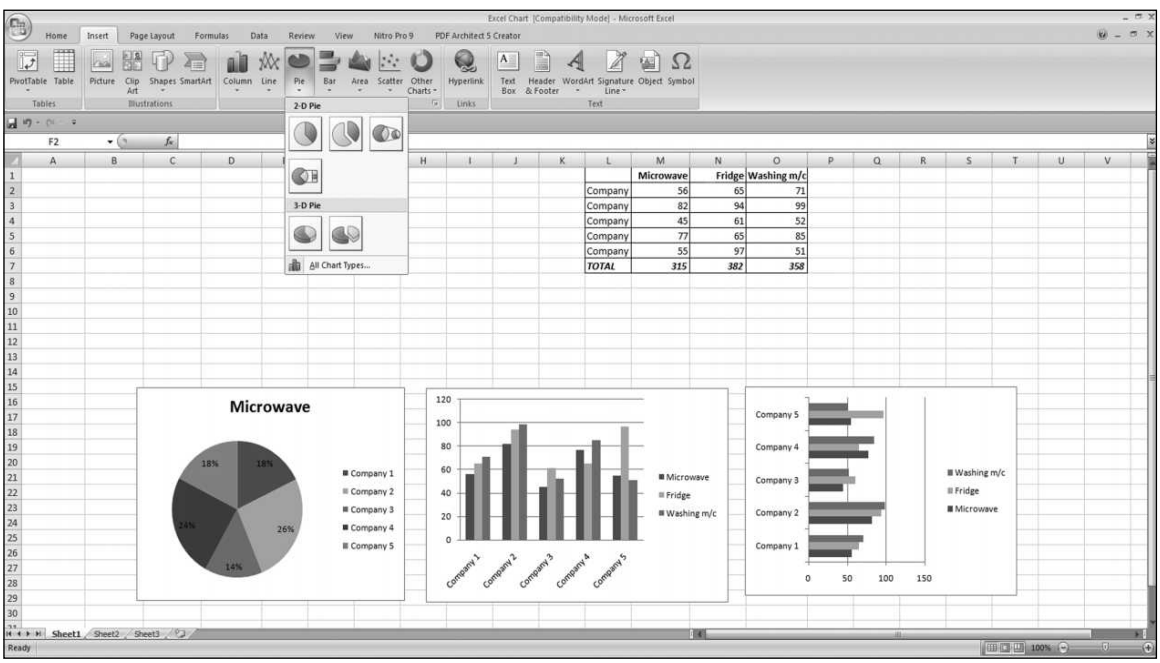


FIGURE 2.23 Excel Charts

When multiple products are involved, the *column chart* and *bar chart* present a more complete picture. To create them, select all the columns from L through O and then choose *Column* or *Bar* from the options offered above. Each of the two charts on the right show (i) how the three products of each company are faring among themselves, (ii) how a product of one company fares vis-a-vis its competitor's. We added the numbers in the bar chart by right-clicking on each of the three segments and selecting the *Add Data* label.

2.12 MICROSOFT POWERPOINT

Powerpoint is the third component of Microsoft Office that is meant to be used for creating and viewing presentations. Many of its features—specially those related to text formatting—are common to Word. Powerpoint files have the extension `.pptx` (or `.ppt` in older versions). Like with its other siblings, the interface and features of Powerpoint have undergone considerable changes over time. Users of Word and Excel should have no difficulty in handling Powerpoint, so we will restrict our discussions on this software to its bare essentials only.

A presentation file comprises a number of *slides* that conform to one or more *templates*. A typical template comprises a *Title* and a *Content* section which are populated simply by keying in text in dotted text boxes (Fig. 2.24). Slides can be added, deleted and duplicated by using the *Slides* view on the left pane. Because a slide can also contain graphics, Powerpoint also supports an *Outline* view on the left pane to show only the text sections of each slide. For the benefit of the presentation designer, the bottom window allows the keying in of notes for each slide. These notes are not displayed in the slide show.

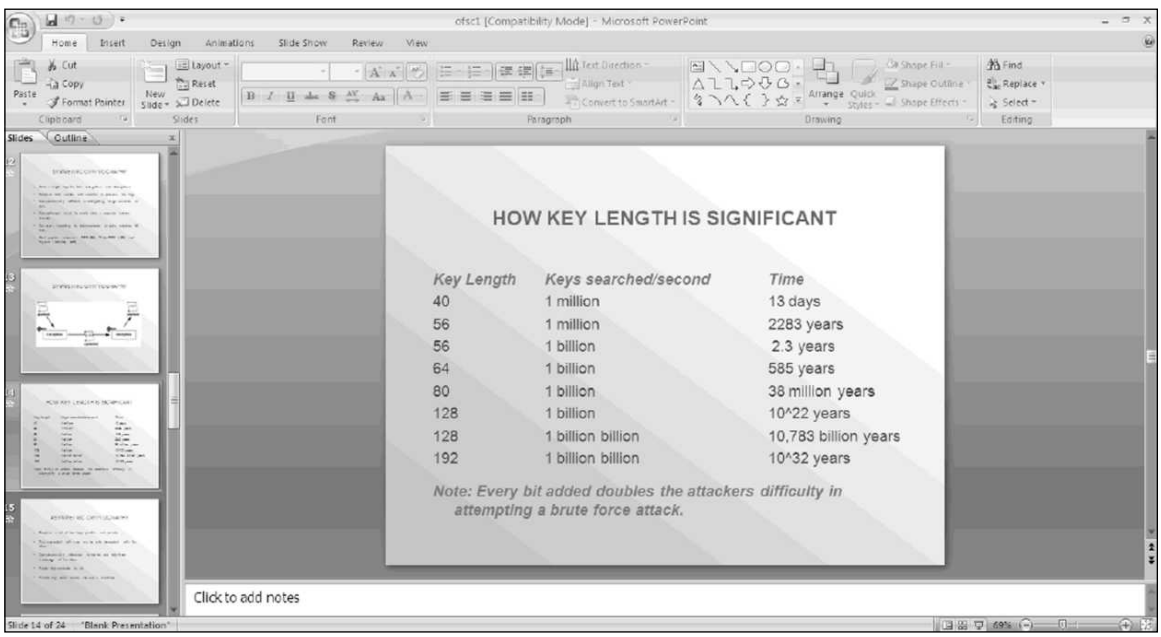


FIGURE 2.24 Microsoft Powerpoint

By default, Powerpoint uses the *Normal view* for display as indicated by the first of the three icons at the bottom. The next icon represents the *Slide Sorter View* which displays the thumbnails of the slides. You can rearrange slides by simply dragging these thumbnails or using the cut-copy-paste facilities available on right-clicking each slide. The presentation can eventually be viewed by clicking on the third icon that represents *Slide Show*, or using the menu option at the top of the screen.

Besides containing formatted text, a slide can feature graphics, audio and video. Each slide can also have (i) *animation* which determines how the text is moved to the screen, (ii) *transition* which controls the way one slide vanishes and the next one appears. The quality of the overall presentation is enhanced by choosing a proper background for each slide.

2.13 UNDERSTANDING A TCP/IP NETWORK

We live in a connected world where computers talk to one another using networking protocols. A *protocol* is a set of rules used by two or more machines to communicate with one another. The standard protocols used on the Internet are collectively known as *TCP/IP.* The acronym expands to *Transmission Control Protocol/Internet Protocol*, but the name is somewhat of a misnomer because it is a collection of several protocols (of which TCP and IP are the most important ones). The Net is running on TCP/IP since 1983.

Unlike our fixed-line telephone system, TCP/IP is a *packet-switching* system. Data are broken into *packets* with the sender's and recipient's addresses written on them. As the packets travel along the network, they encounter routers which look at the addresses to determine the most efficient route a packet has to take to move *closer* to its destination. Packets may move along different routes and arrive out of order. They are reassembled in the correct order at the destination before they finally reach the application.

2.13.1 Hostnames and IP Addresses

In a network, a computer is known as a *host*, sometimes a *node*, and every such host has a *hostname*. Depending on whether a network is part of the Internet, a host in such a network could take on one of the following forms:

```
earth                    A hostname
earth.planets.com        A fully qualified domain name
```

Every host in the network has an address, called the *IP address* which is used by other machines to communicate with it. This address is a series of four dot-delimited numbers (called *octets*) which could typically look like this:

```
92.168.0.1
```

The maximum value of each octet is 255. This address has to be unique not only within the network, but also in all connected networks. And, if the network is hooked up to the Internet, it has to be unique throughout the world. TCP/IP applications can address a host by its hostname as well as its IP address:

```
telnet earth
ftp 92.168.0.1
```

telnet and **ftp** are two important commands used in a network and your Linux system supports both. Modern Windows systems support **ftp** from the command prompt but has the **telnet** command disabled. You can of course enable it if you want *(2.14.2)*.

2.13.2 The TCP/IP Model—The Four Layers

The TCP/IP model handles network communication using a four-layered model (Fig. 2.25). Data are divided into packets and each packet moves from one layer to another. Each layer handles a specific function of the communication process. When sending data, a layer adds information to a packet that helps in routing and reassembly. Sent data move down the following layers:

- *Application layer*, representing the application that generates data. New services are constantly added at this layer. The applications representing Google Chrome or Skype belong to this layer.
- *Transport layer* This layer splits the received data into packets, each packet containing a sequence number. The dominant protocol here is TCP (Transmission Control Protocol), which makes the transmission totally reliable. TCP will retransmit a data packet if acknowledgment of receipt is not received from the other side.
- *Internet layer*, which uses IP (Internet Protocol) as the protocol. IP takes care of addressing by encapsulating each packet with the source and destination addresses (the reason why they are called *IP addresses*).
- *Network access layer*, which finally converts the destination IP address to the MAC address of the network interface card *(1.15.1)*. ARP (Address Resolution Protocol) handles this translation.

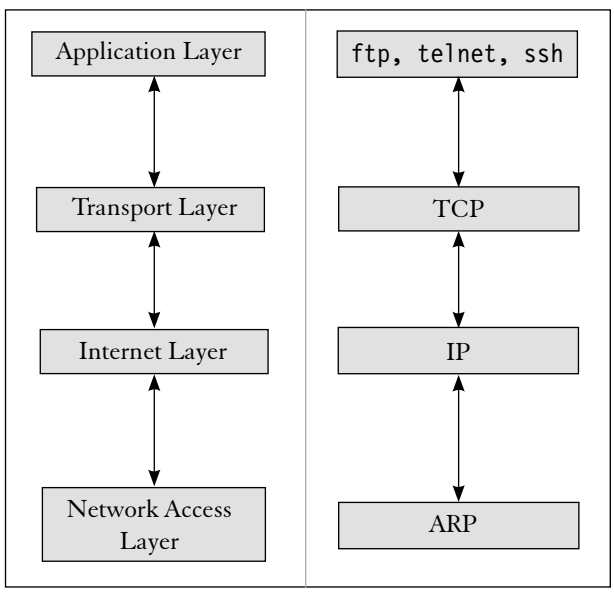


FIGURE 2.25 The TCP/IP Protocol Stack

At the receiving end, the packets move up a similar set of four layers (but in the reverse direction). They are relieved of their headers before they are reassembled. Normally, the set of TCP, IP and ARP protocols handle most of the network traffic.

Note: The 32-bit IP address is not understood by the computer's network interface card, the reason why the IP address has to be finally converted to the MAC address. If you change the network card of a host with another, the MAC address changes, but the IP address remains the same.

2.13.3 DNS: Resolving Domain Names on the Internet

Even though data packets contain IP addresses, we don't *explicitly* use them. For instance, to access Google, we key in its domain name, *www.google.com*, in the URL window of the Web browser. However, the data packets will still contain the IP address of the Google server. Another translation mechanism is at work here; the domain name is converted to the IP address. On the Internet, this translation is performed by the *Domain Name System (DNS)*.

In the Internet namespace, hosts belong to *domains*, which in turn could belong to *subdomains*, and so forth. For instance, if *binghamton* is a subdomain under *edu*, and *cs* a subdomain under *binghamton*, then the host *ralph* could be uniquely described as

ralph.cs.binghamton.edu

This represents a *Fully Qualified Domain Name (FQDN)* of the host *ralph*—something like the absolute pathname of a file. Every domain maintains hostname-IP address mappings of all hosts of that domain in a few *name servers*. When an application tries to contact a host using the FQDN, a name server is first queried to resolve the FQDN to its corresponding IP address. If a name server doesn't have the information, it must be able to contact other name servers, so that the FQDN finally gets resolved to the IP address. The DNS system works perfectly because a name server never gives up.

Note: FQDNs are case-insensitive; you can use *JACK.HILL.COM* in the same way you can use *jACk.HiLL.coM*.

2.14 INTERNET APPLICATIONS

TCP/IP and the Internet operate on the *client-server* principle—the division of labor between two networked computers. An Ftp application is split into a server and a client component—two separate programs residing on two separate machines. The Web server's job is to send an HTML document, which the client software (the browser) formats on the user's screen. In the following sections, we'll have a look at the client side of the major network applications.

2.14.1 Electronic Mail

Electronic mail is possibly the earliest service on the Internet and is still widely used today. It is based on the *Simple Mail Transport Protocol (SMTP)*. The old CUI-based mail clients like **`mailx`** and **`pine`** are no longer in use; most people handle mail on their Web browsers. Sites like Gmail support additional protocols (`POP` and `IMAP`) to enable users to download mail onto their machines.

A sender or recipient of a mail message is identified by their email address which can take one of two forms on the Internet. Here's the first form:

```
romeo@heavens.com
```

All email addresses have the form *name*`@`*domain*, where the `@` acts as the delimiter of *name* and *domain*. The name portion allows the use of all alphanumeric characters, the underscore and hyphen but not spaces.

The second form of the email address shows additional information:

```
juliet floyd <juliet@heavens.com>
```

Here, juliet's email address is embedded within < and > with the full name preceding it. However, only the actual email address—and not the full name—is used to route the message.

Every mail message consists of several lines of header information. A typical message shows at least the first four fields shown below:

```
Subject: creating animations in macromedia director from GIF89a images
   Date: Fri, 08 Nov 2002 15:42:38 +0530
   From: joe winter <winterj@sasol.com>
     To: heinz@xs4all.nl
     Cc: psaha@earthlink.net
```

The subject is optional, but it's good discipline to use the field. The `Date:` and `From:` fields are automatically filled in by the system. The `Cc:` field is used to copy a message to someone else; we call it a *carbon copy*. Some messages will also have a `Bcc:` header where its recipient receives a *blind carbon copy* (a fact not known to the "To:" and "Cc:" recipients). Following the headers is the message body containing the text of the message.

2.14.2 Telnet: Remote Login

You can use the Telnet application to log in to a remote machine. Once you have gained access using the user-id and password, anything you type on your local machine is sent to the remote machine, and your machine just behaves like a dumb terminal (even if it is a PC that you are using Telnet from). Any files that you use or any commands that you run will always be on the remote machine. However, you can't transfer files between two machines using Telnet.

UNIX/Linux systems support a **`telnet`** command but later Windows systems have the facility disabled by default. To enable it, simply switch to *Control Panel* and then select *Turn Windows Features on or off* from the *Programs* menu. You can then invoke the **`telnet`** command from the MSDOS shell.

When using Telnet, data are sent out in clear text which can be easily intercepted. For this reason, Telnet has been superseded by the *Secure Shell (SSH)* facility. If you are concerned about security, use SSH which is discussed in Section 2.14.4.

2.14.3 Ftp: Transferring Files

The Ftp application transfers one or more files and directories in either direction using the *File Transfer Protocol (FTP)*. Both UNIX and Windows offer the **ftp** command for invoking the application. Log in normally as you do with Telnet, and then you can invoke any of the internal commands of **ftp** from the `ftp>` prompt. Here are the commonly used ones:

`put` and `mput`	Upload one or multiple files to the remote machine.
`get` and `mget`	Download one or multiple files from the remote machine.

If files are transferred between machines running different operating systems, you need to be careful when dealing with *text files*. These files contain only printable characters. Text files have different schemes for terminating a line on Apple, UNIX and Windows systems *(9.6)*. Ftp automatically converts the end-of-line character at the time of transfer unless you invoke the **binary** command to disable the conversion.

2.14.4 The Secure Shell

The Telnet and Ftp services are increasingly being disabled because they are inherently insecure. The user-id, password and data are sent across the network in clear text and can be easily intercepted. Confidentiality and integrity are lost because a file being transferred can be read and altered by an outsider.

The *Secure Shell (SSH)* overcomes these limitations by encrypting an entire session. The SSH suite contains the **ssh** and **sftp** commands that let you dispense with **telnet** and **ftp**, respectively. Every UNIX/Linux system supports these commands, but Windows has made it clear that it will do so in the immediate future. Until that happens, you can use the GUI-based SSH applications that are available on the Net and forget **telnet** and **ftp** once and for all.

2.14.5 The World Wide Web

The World Wide Web was conceived by Tim Berners-Lee as a simple mechanism for interconnecting documents spread across the globe. Like Telnet and Ftp, the Web service is based on the client-server model that uses the *Hyper Text Transfer Protocol (HTTP)*. The Web client is represented by a *browser* like Mozilla Firefox or Google Chrome. Web documents are written in the *Hyper Text Markup Language (HTML)* and their files have the extension `.htm` or `.html`.

A Web browser fetches a document residing on Web servers and formats it using the formatting instructions provided in the document itself. However, HTML's real power lies in its *hypertext* capability which links one Web page to another. An HTML page typically contains links to many resources which may be distributed across multiple servers. The client fetches these resources to complete the page display. Today, Web pages have moved beyond displaying simple pictures to the handling of multimedia data (audio and video).

A Web *resource* is described by a *Uniform Resource Locator (URL)*—a form of addressing that combines the FQDN of the site and the pathname of the resource. Here's a typical URL that points to a file:

```
http://www.oracle.com/docs/books/tutorial/information/FAQ.html
```

You'll see this URL in the URL window at the top of the browser. A URL is a combination of the following components:

- The protocol (usually `http://`) used in transferring the resource.
- The FQDN of the host (here, *www.oracle.com*).
- The pathname of the resource (here, `/docs/books/tutorial/information/FAQ.html`).

Normally, we don't specify `http://` when keying in the URL because the browser uses HTTP by default. Unlike C, which produces binary executables from text sources, HTML is not a programming language. Rather, HTML uses tags to "mark up" text. For instance, text enclosed by the `<STRONG>` and `</STRONG>` tags appears in boldface. The `<IMG>` tag specifies the URL of a graphic. Finally, the `<A>` tag signifies a hyperlink that fetches another resource when you click on the link.

Caution: Sites like Gmail and Amazon require user authentication, so they use HTTPS (secure HTTP) instead of HTTP. For these sites, you should see the `https://` prefix in the URL window. All data using HTTPS are encrypted and cannot be intercepted by an outsider. If a Web site asks for your credit card information or your login details, check the URL first. If it doesn't show `https://`, then the site is fraudulent or unreliable. (It can still be even if it shows `https`.)

WHAT NEXT?

Have you observed that we didn't need to change the hardware for running the programs discussed in this chapter? When the application changed, we simply changed the program. We'll soon be writing programs but only after we have grasped the essential computing and programming concepts.

WHAT DID YOU LEARN?

A computer is meant to run programs that contain instructions to act on data. The *source code* of a program is written in a language like C, C++ and Java. This code is passed through a *compiler* to generate *machine code* which eventually runs on the CPU.

A computer needs both *system* and *application software*. System software interacts with the computer hardware while application software relates to the real-life situations that eventually concern the user.

System software comprise the BIOS, operating system and device drivers. The Basic Input Output System (BIOS) performs essential hardware-related tasks on startup. It eventually loads the operating system.

The *operating system* is a type of system software that manages the hardware, allocates memory and provides the essential services needed by application programs. Operating systems use the services of *device drivers* to operate devices.

MSDOS, Windows and UNIX/Linux are some of the commonly used operating systems. All of them have separate commands and programs for handling files and directories. Some commands are internal but most commands are external and exist as separate programs.

Microsoft Office belongs to the category of application software. It comprises the components Word (word processor), Excel (spreadsheet) and Powerpoint (presentation).

Computers use the TCP/IP networking protocols to communicate with one another. A computer has an *IP address* which is encapsulated in the data handled by the computer. All data are sent in *packets* which are reassembled at the destination.

The common network applications are email, Ftp (for file transfer) and Telnet (for remote login). The *secure shell* is superior to them because it encrypts the session. The *HTTP* protocol is used on the World Wide Web to handle HTML documents.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

2.1 Any device powered by a microprocessor needs software to make it work.

2.2 It is possible to generate the source code of a program from its machine code.

2.3 The BIOS makes a preliminary check of the computer's hardware.

2.4 The shell in Windows is represented by the *Explorer* program.

2.5 The OS, and not the device driver, knows the complete details of a hardware device.

2.6 It is possible for a computer with a single processor to *actually* run multiple programs concurrently.

2.7 The internal MSDOS commands are built into the file **`COMMAND.COM`**.

2.8 Both UNIX and Windows use the / as the delimiter in a file pathname.

2.9 The MSDOS command **`DEL *`** deletes all files in the current directory.

2.10 The *Recycle Bin* feature of Windows enables the recovery of deleted files.

2.11 The **`chmod`** command changes the owner of a UNIX file.

2.12 The Domain Name System (DNS) translates IP addresses to their corresponding domain names.

2.13 A mouse click on a Web document may activate a hyperlink to fetch another document.

A2. FILL IN THE BLANKS

2.1 The computer's BIOS belongs to the category of _____ software.

2.2 Programs designed to cause harm belong to the category of _____.

2.3 The BIOS is being replaced with the _____ in modern computers.

2.4 The BIOS setup is saved in a battery-powered _____.

2.5 The process of loading the operating system by the boot loader is called _____.

2.6 When a program needs disk space, the _____ allocates disk blocks for the program to use.

2.7 Users invoke a program from the _____ of the OS.

2.8 For the response to an event to be instantaneous, you need to use a _____ OS.

2.9 Device drivers belong to the category of _____ software.

2.10 **CHKDSK** and **FORMAT** are _____ commands while **DIR** and **COPY** are _____ commands of MSDOS.

2.11 You can use either _____ or _____ to rename an MSDOS file.

2.12 For enhancing security in Windows, you need to activate the _____ feature.

2.13 The core of the UNIX OS is known as the _____.

2.14 The _____ key combination copies text and _____ pastes the copied text in Windows.

2.15 The string `203.55.12.7` signifies the _____ of a host in a TCP/IP network.

2.16 The _____ protocol is used to transfer files between two hosts in a network.

2.17 The World Wide Web uses the _____ protocol and Web document files have the _____ or _____ extension.

A3. MULTIPLE-CHOICE QUESTIONS

2.1 One of the following can't be categorized as system software: (A) UNIX, (B) compilers, (C) Microsoft Word, (D) BIOS.

2.2 A device driver is (A) an electrical switch used to operate a device, (B) a component of the OS, (C) a piece of software invoked by the OS, (D) none of these.

2.3 The operating system is loaded to memory (A) by a user program, (B) before the BIOS is loaded, (C) after the BIOS is loaded, (D) none of these.

2.4 The check for integrity of RAM is made by the (A) BIOS, (B) boot loader, (C) operating system, (D) none of these.

2.5 One of the following is not an operating system: (A) MSDOS, (B) Excel, (C) UNIX, (D) Mac OS.

2.6 A computer needs an operating system because (A) it runs multiple programs, (B) it has to allocate memory for the programs, (C) it has access to the devices, (D) all of these.

2.7 If the computer doesn't boot the OS, it could be because (A) the OS is not installed or is corrupt, (B) the BIOS is wrongly set up, (C) the first sector of the disk doesn't contain the boot loader, (D) any of these.

2.8 Modern Windows filenames use (A) the 8+3 convention, (B) up to 255 characters, (C) up to 256 characters, (D) none of these.

2.9 The MSDOS command for creating a directory is (A) **MKDIR**, (B) **mkdir**, (C) **MD**, (D) A and B, (E) all of these.

2.10 One of the following is not an internal MSDOS command: (A) **DIR**, (B) **SORT**, (C) **TYPE**, (D) **DATE**.

2.11 It is possible for two files, **foo** and **Foo**, to coexist in the same directory in (A) UNIX, (B) Windows, (C) both, (D) neither.

2.12 Font attributes include (A) boldface, (B) underline, (C) italics, (D) all of these.

2.13 You can sum the numbers 5, 10 and 15 in Excel using (A) `=5 + 10 + 15`, (B) `=SUM(5, 10, 15)`, (C) `=SUM(5 + 10 + 15)`, (D) all of these.

2.14 An email address (A) must contain an `@`, (B) may have text in either case, (C) must have at least one dot, (D) all of these.

2.15 The following applications are insecure: (A) Telnet, (B) Ftp, (C) HTTP, (D) all of these.

A4. MIX AND MATCH

2.1 Match the software name/type with their significance:

(A) Excel, (B) Access, (C) Windows, (D) Powerpoint, (E) trojan horse, (F) compiler.

(1) Operating system, (2) presentation, (3) virus, (4) code translation, (5) DBMS, (6) spreadsheet.

2.2 Match the MSDOS commands with their functions:

(A) **`DIR`**, (B) **`CHDIR`**, (C) **`TYPE`**, (D) **`COPY CON`**, (E) **`PATH`**, (F) **`CLS`**.

(1) Creates a file, (2) locates programs, (3) clears screen, (4) lists files, (5) displays file contents, (6) shows current directory.

2.3 Associate the Windows programs and key sequences with their function:

(A) *[Ctrl-c]*, (B) *[Ctrl][Alt][Del]*, (C) *[Ctrl-v]*, (D) *Recycle Bin*, (E) *Explorer*, (F) *Control Panel*.

(1) Views file system, (2) configures devices, (3) copies text, (4) reboots system, (5) restores files, (6) pastes text.

2.4 Match the UNIX/Linux commands with their functions:

(A) **`pwd`**, (B) **`cat`**, (C) **`mv`**, (D) **`ls`**, (E) **`grep`**, (F) **`chmod`**.

(1) Displays file, (2) renames file, (3) changes file permissions, (4) lists files, (5) displays current directory, (6) locates string in file.

2.5 Associate the file extensions with their handling programs:

(A) `.docx`, (B) `.xlsx`, (C) `.html`, (D) `.gif`, (E) `.pdf`.

(1) Web browser, (2) Image viewer, (3) Acrobat Reader, (4) Excel, (5) Word.

2.6 Match the network protocols with their function:

(A) TELNET, (B) HTTP, (C) SMTP, (D) FTP, (E) SSH.

(1) File downloading, (2) mail handling, (3) secure network access, (4) remote login, (5) Web access.

CONCEPT-BASED QUESTIONS

2.1 Explain the features of POST run by the BIOS.

2.2 Explain how the OS enables programmers to write hardware-independent programs. How does the OS handle changes in hardware?

2.3 Why doesn't a microprocessor-based device like a washing machine have an OS installed? Does it make sense to have an OS included in the program itself?

2.4 Explain the difference between multi-programming and multi-tasking systems.

2.5 Explain the significance of the following MSDOS commands: (i) **XCOPY**, (ii) **CHDIR bar**, (iii) **COPY foo1 + foo2 foo**, (iv) **REN *.xls *.xlsx**.

2.6 Why is Windows not considered to be a multi-user system even though multiple users can access a Windows machine?

2.7 Name two differences between UNIX and Windows file pathnames.

2.8 What is the difference between an internal and external command in (i) MSDOS and (ii) UNIX?

2.9 Explain the significance of the following UNIX commands: (i) **cp *.c backup**, (ii) **rm foo[1-5]**, (iii) **ls foo?**, (iv) **cd**.

2.10 What is the significance of PATH in MSDOS/Windows and UNIX? How does its representation differ in the two systems?

2.11 Explain the characteristics of the packet-switching system used in a TCP/IP network.

3 Computing and Programming Concepts

WHAT TO LEARN

- The foundation of numbering systems.
- Techniques of performing arithmetic operations with binary numbers.
- Key features of the octal and hexadecimal numbering systems.
- Special features of fractional and negative numbers.
- Principles of functioning of *logic gates*.
- Programming methodologies like the *top-down* and *bottom-up* strategies.
- Benefits of *procedural* and *structured programming*.
- How to use *algorithms* and *flowcharts* for solving problems.
- Classification of programming languages by *generations*.
- Use of *compilers*, *linkers* and *loaders* in program development.

3.1 NUMBERING SYSTEMS

Computers are meant to compute; that's a given. Humans compute using a numbering system that comprises 10 symbols that we call *digits*. There is nothing magical about the number 10. Our hands have ten fingers, so we created the decimal or *base-10* system, which uses 10 as the *base*. If we had 12 fingers, we would have invented two more symbols to have a *base-12* system.

A computer doesn't have fingers, but it has millions of tiny devices that can be in one of two states. For mathematical convenience, we refer to them as 0 and 1. By manipulating these states, we can create a new set of numbers having only two symbols or digits. We call them *binary* numbers or *base-2* numbers. The binary number 101 represents 5 in the decimal system. A computer understands 101 and can easily add another binary number 11 to it, but it can't make sense of the decimal number 5. Leibniz had foreseen this constraint *(1.3.1)*, but his advocation of the binary system was eventually realized centuries later.

Before examining the other numbering systems used in computer programs, we need to know how a decimal number like 357 is interpreted. Let's split the number into its 3 digits and assign a *weight* to each digit:

Hundreds		*Tens*		*Units*	*Digit Labels*
3		5		7	Digits
10^2		10^1		10^0	Weights
300	+	50	+	7	Weighted values

The right-most digit, called the *least significant digit (LSD)*, has the weight 10^0, which is 1. (The value of any number to the power 0 is always 1.) You can count from 0 and up to 9 using this single digit. When the number gets larger, a second digit is required to represent it. This digit has a higher weightage (10^1) than the one on its right. So, the digits 5 and 7 collectively evaluate to $5 \times 10^1 + 7 \times 10^0 = 57$. The number 357 is actually $3 \times 10^2 + 5 \times 10^1 + 7 \times 10^0 = 357$. We'll learn to apply these principles to other numbering systems.

Computer scientists have discovered that it's often both convenient and efficient to use as base a number which is an integral power of 2. In this text, you'll encounter the *octal* (base-8) and *hexadecimal* (base-16) number systems. While the octal system uses a subset of the digits of the decimal system (0 to 7), six new symbols are needed by the hexadecimal system (0 to 9 and A through F). Table 3.1 shows how the decimal numbers between 0 and 15 are represented in the four standard systems.

In the following sections, we'll examine the binary, octal and hexadecimal systems. We'll perform the basic arithmetic operations on them and convert numbers between these systems. These operations are based on consistent principles, so knowing one system thoroughly makes it easier to understand the others. We begin with the binary numbering system.

TABLE 3.1 The First 15 Positive Integers

Decimal	*Binary*	*Octal*	*Hexadecimal*
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Takeaway: The *base* represents the number of symbols used to represent the digits. The greater the base, the more compact is the representation of a number in that system. The decimal number 15 is represented by 2 digits in base-10, 4 digits in base-2 (1111), but a single digit in base-16 (F).

3.2 THE BINARY NUMBERING SYSTEM

A binary number uses only two types of digits, 0 and 1. These digits are called *binary digits*, or abbreviated to *bits*. Although binary numbers are suitable for computers, they are not suitable for general use because they run out of bits quickly. This is revealed by a look at the first eight integers in their binary and decimal forms:

Decimal	0	1	2	3	4	5	6	7
Binary	0	1	10	11	100	101	110	111
Number of Bits	1	1	2	2	3	3	3	3

Note that one bit can represent up to the decimal number 1. Two bits extend this figure to 3 (11), while 3 bits can represent up to 7 (111). The maximum number that can be represented with 8 bits is $2^8 - 1 = 255$, which needs only 3 digits in decimal. In general, the maximum number that can be represented with n bits is $2^n - 1$.

By simply looking at a number, you can't determine its base. You can't say with certainty that 10101 is a binary number because it is also a valid decimal, octal or hexadecimal number. The standard technique employed to remove this ambiguity is to enclose the number within parentheses and use the base as the subscript or suffix. Thus, binary 10101 can be unambiguously written as $(10101)_2$ or (10101)2. Similarly, (9999)16 is a hex number even though it uses only decimal digits.

Note: C uses the symbols 0 and 0x as prefixes to indicate octal and hexadecimal numbers, respectively. For instance, when C encounters the numbers 077 and 0x4F, it treats them as 77 in octal and 4F in hex, respectively.

3.2.1 Converting from Binary to Decimal

How does one compute the decimal value of a binary number? Simply multiply each digit by its assigned weight and then sum the weighted values. To see how this is actually done, split the binary number 10101 into its constituent bits and align them with their weights and weighted values:

1	0	1	0	1	*Bits (A)*
2^4	2^3	2^2	2^1	2^0	*Weights (B)*
16	0	4	0	1	*Weighted values (A) × (B)*

Note that the weights increase proportionately as the digits are traversed from right to left. Adding the products in the third row (16 + 0 + 4 + 0 + 1) yields 21, which is the decimal value of 10101. Nothing could be simpler than that!

Note: The weights are an increasing power of the base, i.e., the number of symbols used by the numbering system to represent its digits. An octal number would have the weights 8^0, 8^1, 8^2, and so forth.

3.2.2 Converting from Decimal to Binary

Conversion from decimal to binary calls for employing a totally different technique. Divide the decimal number repeatedly by 2 and collect all the remainders. The binary equivalent is the *reversed* sequence of these remainders. The following examples employ this method for converting the decimal numbers 6, 19 and 32 to binary. The first line shows the division to be performed. Subsequent lines display the progressive quotient and remainder:

2|6		2|19		2|32	
3	Rem 0	9	Rem 1	16	Rem 0
1	Rem 1	4	Rem 1	8	Rem 0
0	Rem 1	2	Rem 0	4	Rem 0
Binary:	110	1	Rem 0	2	Rem 0
		0	Rem 1	1	Rem 0
		Binary:	10011	0	Rem 1
				Binary:	100000

Let's examine the first example. Division is performed three times:

(i) 6 is divided by 2. This produces the quotient 3 and remainder 0.
(ii) 3 is divided by 2. Both quotient and remainder are 1.
(iii) 1 is divided by 2. The quotient is 0 and remainder is 1.

The remainders are thus generated in this sequence: 0 1 1. Reverse this sequence to 110 and you have the binary equivalent of 6. Note from all three examples that division is terminated when the quotient is 0 and the remainder is 1.

3.2.3 Binary Coded Decimal (BCD)

Many devices like pocket calculators have LED displays where each digit is represented by a set of seven segments. Such devices store numbers in *binary code decimal (BCD)*, where each decimal digit is stored in binary form. For instance, the number (254)10, which is 11111110 in "pure" binary (8 bits), is represented as 0010 0101 0100 in BCD (12 bits). Here are a few more examples:

Decimal	*Binary*	*Binary Coded Decimal (BCD)*
9	1001	1001
14	1110	0001 0100
32	100000	0011 0010
127	1111111	0001 0010 0111

Any decimal digit can be represented with 4 bits, so *packed BCD* uses 4 bits (half a byte or one *nibble*) for each digit. The third column shows the packed BCD representation. *Unpacked BCD* takes up 8 bits (one byte or two nibbles) for each digit. The last example suggests that a three-digit decimal number like 127, which is represented by 7 digits in binary, takes up 12 bits in packed BCD and 24 bits in unpacked BCD.

Note that BCD (packed or unpacked) takes up more space because each of the decimal digits is codified separately. This wastage can be justified in situations where fractional numbers are involved. A decimal number having a fractional component can be accurately converted only to BCD and not to binary.

3.3 NEGATIVE BINARY NUMBERS

In arithmetic, we use the - symbol on the left of the most significant digit (MSD) to signify a number as negative. So, binary -1111 is mathematically equivalent to decimal -15. But computers are designed to handle only 0s and 1s; there's no room for a third symbol. Computers handle negative numbers by using the left-most bit to indicate the sign. This bit is known variously as the *Most Significant Bit (MSB)*, *sign bit* or *high-order bit*. Positive numbers have the MSB set to 0 and negative numbers have it set to 1. The remaining bits are used for the *magnitude* (value) of the number.

Positive one-byte integers use all 8 bits for the magnitude and none for the sign. This makes it possible to represent the 256 numbers between 0 and 255 (2^8 - 1). However, if a number *can* be negative, then the MSB must be reserved for the sign. The range of numbers then changes from -127 to +127 or -128 to +127, depending on the scheme used by the computer to store negative numbers. It's the programmer's job to tell the computer whether it should treat a number or a numeric variable as *signed* or *unsigned*.

There are mainly three schemes for storing the magnitude of a negative number. Let's briefly examine them with reference to byte-sized (8 bits) integers.

3.3.1 Sign-and-Magnitude Representation

In this system, the left-most bit (MSB) is allocated for the sign as usual: 0 for positive and 1 for negative numbers. The remaining 7 bits are reserved for the magnitude or value. For instance, the number (127)10 is represented as 01111111 in binary, where the MSB or high-order bit is set to 0. To negate this number to -127, set this bit to 1 and the number then becomes 11111111. Here are two more examples:

Decimal	*Binary*	*Decimal*	*Binary*
6	0000110	56	00111000
-6	1000110	-56	10111000

Using a byte-sized integer, the sign-magnitude system can represent values between -127 and +127, but it has an inherent flaw: it creates two zeroes. The number 00000000 is positive zero and 10000000 is negative zero. Two zeroes create problems when comparing numbers, the reason why this system is not used today.

3.3.2 One's Complement

In this system, a negative number is represented as the *one's complement* of the positive number. This means inverting all bits of the number, i.e. changing 0s to 1s and 1s to 0s. The significance of the MSB remains the same—0 for a positive number and 1 for negative numbers. The following examples show how the one's complement of (15)10 and (85)10 are derived:

Decimal	*Binary*	*Decimal*	*Binary*
15	00001111	85	01010101
-15	11110000	-85	10101010

The one's complement of zero (00000000) is 11111111, which is -0. The problem of positive and negative zero remains here as well. The one's-complement system was once used by IBM and PDP for some of their machines. Today, it plays the role of an "assistant" to the two's-complement system.

Note: Testing for a zero in both sign-magnitude and one's complement forms becomes tedious because one has to test for both +0 and -0.

3.3.3 Two's Complement

This is the most widely accepted system for representing negative numbers. The two's complement of a number is simply its one's complement with a 1 added to it. The following example illustrates the point:

Decimal +15	00001111	
-15 in one's complement form	11110000	*High-order bit set*
Adding 1 to one's complement	+1	
-15 in two's complement form	11110001	*High-order bit set*

How does two's complement handle the "two-zero" problem? In the one's complement system, -0 is represented as 11111111. When you add 1 to it, an overflow is generated which is ignored in the two's complement system:

Decimal +0	00000000	
-0 in one's complement form	11111111	
Adding 1 to one's complement	+1	
	100000000	*Overflow to be ignored*
-0 in two's complement form	00000000	*Same as +0*

You'll soon learn that binary addition of 1 and 1 creates 10 (2 in decimal), where the 1 is carried over to the left bit. That's how adding 1 to 11111111 reset all of these 1s to 0s but it also resulted in a carry of 1. By ignoring this carry, two's complement ensures that there is only one zero in the system. The range of numbers changes slightly; it's -128 to 127 compared to -127 to 127 that is possible with the one's complement system.

With negative binary numbers encroaching on half of the domain of the unsigned ones, how does one interpret a binary number, say, 11111111 that has its MSB set? The answer is one of the following:

(i) 255 if the number is *declared* as unsigned, i.e., a positive number.
(ii) -127 in sign-magnitude form.
(iii) -0 in one's complement form.
(iv) -1 in two's complement form.

The answer in (i) depends on us—the way we *declare* the number in a program. The other answers depend on the computer—the way it internally stores negative numbers. Declaration of a number belongs to a later topic *(5.2.2)*.

Takeaway: By simply looking at a binary number that has its MSB or high-order bit set, you can't determine its actual value. If the number is declared to the computer as unsigned, then the *entire* number is the actual value. However, if the number is signed, the value would depend on the system used by the processor to store negative numbers.

3.4 BINARY ARITHMETIC

We have just added 1 to a binary number without knowing the formal rules related to binary addition. If you thought that there is nothing special in binary arithmetic, then you are absolutely right. The rules of arithmetic were not based on a specific numbering system. The standard "carry" and "borrow" techniques used for decimal arithmetic apply equally to binary arithmetic (or, for that matter, any arithmetic).

3.4.1 Binary Addition

How do you add two decimal integers? Right-align the numbers and then sum the columns from right to left. If the sum of one column can't be held in a single digit, carry the extra digit to the next column. The following example shows the carry-over generated by columns 2 and 3 when adding the decimal numbers 398 and 56:

```
 11          Carry over digits
 398
+ 56
 454         Sum
```

The addition of 8 and 6 takes place first. It generates a carry of 1 (from 14) which is shown at the top of column 2. This column, which uses the carry, generates yet another carry which is used by column 1.

Binary addition is no different except that (i) digits are restricted to 0 and 1, (ii) addition of 1 and 1 generates a carry of 1 (1 + 1 = 10). Here are three examples of binary addition; the first one doesn't create a carry but the others do:

	11	1111	*Carry over bits*
1010	1110	1111	
+ 101	+ 111	+ 1	
1111	10101	10000	*Sum*
	1234	1234	*Column ruler*

The first example computes 1111 or (15)10 as the sum. This is based on the principle that the sum of 0 and 1 evaluates to 1. The second example features two carry operations. Column 3 generates

a carry from 1 + 1. This carry is then embedded in the expression 1 + 1 + 1 of column 2, which creates 11 as the sum. Another carry is thus generated for column 1 which sums 1 and 1 to 10. The third example generates a carry for every column summed. That's how (1111)2 is incremented to (10000)2.

3.4.2 Binary Subtraction (Regular Method)

There are two ways of performing binary subtraction—the normal way and using the two's complement form. The normal way is to simply right-align the numbers and then subtract the *subtrahend* from the *minuend* while moving from right to left. If the subtrahend is larger than the minuend, you borrow a 1 from the left column. Consider the following example that features decimal subtraction:

81	*Borrowed digit*
967	*Minuend*
- 483	*Subtrahend*
484	*Difference*

Subtraction in column 2 requires a 1 to be borrowed from column 1. *This 1 has a weight equal to the base*, i.e., 10. It is thus multiplied by 10 and added to 6, yielding 16 from where 8 is subtracted. Because the previous column has "lent" this 1 to the next column, the minuend 9 gets reduced to 8.

The same principles are followed in binary subtraction where 0 - 1 = 1 after borrowing 1 from the previous bit. Also, a borrowed bit has a weight of 2 and not 10. Consider the following three examples of which two feature borrowing:

	0111	01111	*Borrowed bit*
1111	10101	10000	*Minuend*
- 101	- 111	- 1	*Subtrahend*
1010	1110	1111	*Difference*
1234	12345	12345	*Column ruler*

The first example needs no explanation. In the second example, borrowing begins from column 4. The borrowed 1 is multiplied by the base (2) and then the subtrahend 1 is subtracted from it. The minuend of column 3 (which has now become 0) borrows from column 2. But column 2 can only lend after it has borrowed from column 1. The third example shows how continuous borrowing reduces (10000)2 by 1 to (1111)2.

3.4.3 Binary Subtraction Using Two's Complement

We have known since high school that 7 - 3 is the same as 7 + -3. In other words, subtraction is the same as addition of the minuend with the negative representation of the subtrahend, i.e., its two's complement. Subtraction using two's complement is easily achieved by following these simple steps:

- Ensure that both minuend and subtrahend have the same number of bits by padding one of them with 0s if necessary.
- Convert the subtrahend first to its one's complement, i.e. by inverting all of its bits. Add 1 to the one's complement to obtain the two's complement.

- Add the two's complement of the subtrahend to the minuend and ignore the carry, i.e. the extra bit that is produced.

Using these techniques, let us redo the examples of Section 3.4.2, using the two's complement method this time:

1111	10101	10000	*Minuend (A)*
101	111	1	*Subtrahend (B)*
0101	00111	00001	*Padding subtrahend with 0s*
1010	11000	11110	*One's complement of subtrahend*
1011	11001	11111	*Adding 1 to one's complement (C)*
+ 1111	+ 10101	+ 10000	*Unchanged minuend (A)*
11010	101110	101111	*(A) + (C)*
1010	1110	1111	*Actual difference after ignoring carry and leading zeroes*

The annotations should be adequate to understand how the subtraction process is performed using this technique. Note that (A) + (C) generates a carry in each of the three examples. This is not a coincidence; the carry is generated only when the difference is positive. The actual difference is shown in the last row after removing the carry bit. We removed the leading zero only for cosmetic effect.

3.4.4 Binary Multiplication

Binary multiplication is probably the simplest of all arithmetic operations when using the well-known *partial products* method. This technique requires (i) multiplying a number A with each digit of number B, (ii) shifting each subsequent row by one digit space to the left and (iii) summing the partial products to obtain the final product. The rule to remember here is that $1 \times 1 = 1$, and the other operations (1×0, 0×1 and 0×0) lead to zeroes. Consider the following three examples:

101 × 10	1011 × 101	1111 × 11	*Number 1* *Number 2*
000 101	1011 0000	1111 1111	
1010	1011	101101	
	110111		
1234	123456	123456	*Column ruler*

Note from each of these examples that a row of partial product contains either zeroes or a copy of the first number. The first example is the simplest with no carry produced during the summing of the rows. The second example shows a single carry generated in the third column. It's the third example that we need to examine closely as it generates four carry operations. Let's examine the summation of each column starting from column 6 and moving left from there:

- Column 6: $1 + 0 = 1$, no carry.
- Column 5: $1 + 1 = 10$, generates a carry.

- Column 4: 1 + 1 + 1 (carry) = 11, also generates a carry.
- Column 3: 1 + 1 + 1 (carry) = 11, also generates a carry.
- Column 2: 0 + 1 + 1 (carry) = 10, also generates a carry.
- Column 1: Holds only the carry.

3.4.5 Binary Division

The division operation involves dividing the *dividend* by the *divisor* to create a *quotient* and optionally a *remainder*. Like in decimal division, we use the *long division* method to perform binary division. Compared to the other arithmetic operations, division is a little complex because it involves the use of multiplication and subtraction as well. So, let's have a look at the technique that we have used in school to divide the decimal number 1035 by 11:

	094	*Quotient*
Divisor 11	1035	*Dividend*
	0	
	103	*3 brought down from dividend*
	99	
	45	*5 brought down from dividend*
	44	
	1	*Remainder*

Because the divisor (11) has two digits, we first try to divide the first two digits of the dividend (10). That is not possible, so we place a zero as the first digit of the quotient and bring down the next digit (3) from the dividend. Division of 103 by 11 creates a quotient of 9 and a remainder of 4. The 9 is suffixed to the overall quotient placed at the top. This process goes on until all digits of the dividend have played their part. Note that each digit of the quotient is created by multiplication and all remainders are formed by subtraction.

Let's use this technique in binary division. Make sure that you are comfortable with binary multiplication and subtraction before taking up the following examples. We'll discuss the first example so that you have no problems understanding the other two. Note that two of these examples produce remainders:

110	0111	011101
10|1101	11|10101	101|10010101
10	0	0
10	101	1001
10	11	101
01	100	1000
0	11	101
1 *Remainder*	11	111
	11	101
	0 *No remainder*	100
		0
		1001
		101
		100 *Remainder*

The first row of each example shows the quotient which forms progressively with each multiplication. The second row shows the divisor and dividend. For the first example, let's break up the entire division process into the following steps:

- Divide the first two digits of the dividend (11) by the divisor (10). This creates the quotient 1 and remainder 1. The quotient goes to the top to form the first digit of the ultimate quotient.
- Bring down the next digit of the dividend (0) and suffix it to the remainder. The new dividend becomes 10. Division of 10 by 10 creates the quotient 1 and remainder 0. The intermediate quotient now has the value 11.
- Bring down the final digit of the dividend (1). The new dividend is now 01, or simply, 1. Further division is not possible, so the remainder remains at 1 and the quotient acquires 110 as its final value.

You should now have no problem in understanding the other examples.

3.5 THE OCTAL NUMBERING SYSTEM (BASE-8)

The octal numbering system uses 8 (a power of two) as the base. The permissible (octal) digits are 0 to 7, so the first 8 octal integers appear identical to the corresponding decimal ones:

Octal	0	1	2	3	4	5	6	7	10	11	12	13	14
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12

While one digit can represent up to the octal number 7, two digits extend this figure to 77 (decimal 63). Using the formula $2^n - 1$, the maximum decimal number that can be represented with three octal digits is $8^3 - 1 = 511$.

3.5.1 Converting from Octal to Decimal

The principles of converting a number between two number systems have already been discussed with reference to the binary and decimal systems *(3.2.1 and 3.2.2)*. For an octal number, the assigned weights are an increasing power of 8 as digits are traversed from right to left. To convert the octal value 357 to decimal, multiply each octal digit by its assigned weight and then sum the weighted values:

3	5	7	*Octal digits (A)*
8^2	8^1	8^0	*Weights (B)*
64×3	8×5	1×7	*Weighted values = (A) × (B)*

Adding the products in the third row (192 + 40 + 7) yields (239)10, the decimal value of (357)8.

3.5.2 Converting from Decimal to Octal

For converting a decimal number to binary, we repeatedly divided the number by 2 and reversed the remainders *(3.2.2)*. For decimal to octal conversion, we'll use the same technique but use 8 as the divisor instead of 2. The following examples convert the decimal numbers 256, 999 and 4095 to octal:

8|256		8|999		8|4095	
32	Rem 0	124	Rem 7	511	Rem 7
4	Rem 0	15	Rem 4	63	Rem 7
0	Rem 4	1	Rem 7	7	Rem 7
		0	Rem 1	0	Rem 7
	Octal: 400		Octal: 1747		Octal: 7777

The dividend in the first example is perfectly divisible by 8, but the last example shows a remainder of 7 with every division. In every case, division terminates when the quotient is zero. You then need to collect the remainders and reverse them to obtain the octal representation of these numbers.

3.5.3 Converting from Octal to Binary

Conversion between two bases, both of which are powers of 2, is simple to handle. Because an octal digit can be represented by a 3-bit binary number, any octal number can be converted to binary simply by converting each of the octal digits to binary. The following examples show how easy the conversion is:

Octal	*Binary*	
7	111	
15	001 101	*Binary 101 is 5 in octal/decimal*
467	100 110 111	*Binary 111 is 7 in octal/decimal*

For ease of understanding, the binary representation of each octal digit is shown as a separate group and leading zeroes have been added for reasons of symmetry.

3.5.4 Converting from Binary to Octal

The logic that was used to convert an octal number to binary needs to be reversed for converting a binary number to octal. Break up the binary number into groups of 3 bits each and convert each group into its corresponding octal digit. If the total number of bits is not a multiple of 3, pad the binary number with leading zeroes:

Binary	*Adjusted Binary*	*Octal*	*Decimal*
1111	001 111	17	15
10100101	010 100 101	245	165
10111001101	010 111 001 101	2715	1485

In all of the examples, zero-padding was required. The decimal values of the numbers have also been shown in the last column. You'll find it easier to derive these decimal values from their octal rather than binary equivalents.

3.6 THE HEXADECIMAL NUMBERING SYSTEM (BASE-16)

This system uses 16 types of digits to represent numbers. These digits comprise the numerals `0` to `9` and the letters `A` to `F`. The next digit after `9` is `A` and the maximum (16th) digit is `F`. Here is a comparative presentation of the top 18 decimal and hexadecimal integers:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12

The decimal number 255 is represented in hex by FF, which uses only two digits. Because of their compactness, hexadecimal numbers are seen in computer documentation. The C language recognizes both octal and hex numbers.

3.6.1 Converting from Hexadecimal to Decimal

The assigned weights for base-16 numbers are an increasing power of 16. To convert the hex value 3AF to decimal, multiply each hex digit by its assigned weight and then sum the weighted values:

3	A	F	*Hex digits (A)*
16^2	16^1	16^0	*Weights (B)*
256×3	16×10	1×15	*Weighted values = (A) × (B)*

Adding the products in the third row (768 + 160 + 15) yields 943, which is the decimal value of (3AF)16. Note that A has the decimal value 10 and F has the value 15. Consider a few more examples:

Hexadecimal	*Weighted Values*	*Decimal*
AB	$10\times16^1 + 11\times16^0$	171
111	$1\times16^2 + 1\times16^1 + 1\times16^0$	273
F0A	$15\times16^2 + 0\times16^1 + 10\times16^0$	3850

We know that a 32-bit number is capable of representing 4 billion. The same number can be represented with only 8 hex digits (FFFFFFFF = 4 billion in decimal).

3.6.2 Converting from Decimal to Hexadecimal

Conversion of a base-10 number to base-16 should be a no-brainer by now. Divide the base-10 number repeatedly by 16, collect the reminders and reverse them. The following examples convert the decimal numbers 200, 1503 and 65529 to hex:

16|200		16|1503		16|65529	
12	Rem 8	93	Rem 15	4095	Rem 9
0	Rem 12	5	Rem 13	255	Rem 15
		0	Rem 5	15	Rem 15
				0	Rem 15
	Hex: C8		Hex: 5DF		Hex: FFF9

The first example generates a final remainder of 12, which translates to C in the hex system. So, (200)10 = (C8)16. The second example shows two remainders of 15 and 13, which translate to F and D, respectively. The third example generates a remainder of 15 three times in a row to convert (65529)10 to (FFF9)16. It takes a little time to get used to treating letters as digits, but things do get better with practice.

3.6.3 Converting between Hexadecimal and Octal Systems

Because both systems use a base that is a power of 2, you need to tweak the conversion principles *(3.2)* that have been used earlier. For conversion from hex to octal, you need to do the following:

- Break up each hex digit into its corresponding 4-digit binary.
- Combine all of these bits and split them again on groups of 3 bits.
- Convert each group of 3 bits to its corresponding octal.
- Pad one or more zeroes to each octal digit to maintain the 3-bit structure.

Let's apply these principles to the hex numbers D, FFF and 9FA. The first example needs two zeroes to be padded:

Hex	D	F F F	9 F A
Binary	1101	1111 1111 1111	1001 1111 1010
Regrouped Binary	001 101	111 111 111 111	100 111 111 010
Octal equivalent	1 5	7 7 7 7	4 7 7 2

Let's now reverse the previous process to perform octal-hex conversion. This time, break up each octal digit to its 3-bit binary, group these bits and split them again on 4 bits before converting each group to hex:

Octal	1 5	7 7 7 7	4 7 7 2
Binary	1 101	111 111 111 111	100 111 111 010
Regrouped Binary	1101	1111 1111 1111	1001 1111 1010
Hex equivalent	D	F F F	9 F A

We have got back the original numbers. Note that the 2 in 4772 has to be represented as 010 to maintain the 3-bit structure.

3.6.4 Converting between Hexadecimal and Binary Systems

The conversion techniques needed here should be quite obvious as both systems use bases that are a power of 2. To convert a base-16 number to base-2, simply break up each hex digit to its corresponding 4-bit binary, using leading zeroes if necessary. We have already carried out this exercise as an intermediate step in hex-octal conversion, so the following examples should be quite clear:

Hex	F 1 F	1 F 0 A
Binary	1111 0001 1111	0001 1111 0000 1010
Simplified Binary	111100011111	1111100001010

Note the importance of zero padding in both of these examples. Both 1 and 0 had to be written as 0001 and 0000, respectively, for the conversion to be correct. However, we removed the leading zeroes from the converted number in the second example without losing accuracy.

Conversion from binary to hex requires splitting the binary number into groups of 4 bits and then converting each group into its corresponding hex. Let's reverse the previous conversion to get back the hex forms of the numbers:

Binary	111100011111	1111100001010
Grouped Binary	1111 0001 1111	0001 1111 0000 1010
Hex	F 1 F	1 F 0 A

There's nothing special about this conversion; the lines of the example prior to the previous one have simply been reversed with some change in labels.

Takeaway: The mechanism of splitting each digit into binary before regrouping the bits works only when both numbers have bases that are a power of 2. If one of them is not, then the number with the larger base needs to be repeatedly divided by the other. In that case, the converted number is represented by the remainders arranged in reverse.

3.7 NUMBERS WITH A FRACTIONAL COMPONENT

In everyday life, we are used to handling base-10 numbers with a fractional component (like 123.456). In general, numbers with any base can have a fractional component which is interpreted and converted using techniques similar to those used for whole numbers. It would be helpful to recall these techniques and note the change needed in one of them:

- For conversion to decimal (base-10), assign weights normally to each digit and then sum the weighted values.
- For conversion from decimal, use the long division method. For fractional numbers, this technique needs modification.
- When both bases can be represented as powers of 2, split each digit as usual into binary and regroup the bits (3 for octal, 4 for hex).

Let us now separately re-examine the conversion techniques for these three categories. We'll often focus only on the fractional component because conversion of the integral portion has already been discussed.

3.7.1 Converting Binary, Octal and Hexadecimal Fractions to Decimal

This first category of conversion is identical to that used for whole numbers. You are aware that the digits of a whole number, when traversed from right to left, are assigned weights that are an increasing power of the base. The same principle applies to the fractional portion, so the first decimal digit has a weight of 10^{-1}, followed by 10^{-2}, and so on. This is how the decimal number 123.456 is related to its components:

Digits	1		2		3	.	4		5		6	
Weights	10^2		10^1		10^0	.	10^{-1}		10^{-2}		10^{-3}	
Weighted values	100×1		10×2		1×3	.	0.1×4		0.01×5		0.001×6	
Computed values	100	+	20	+	3	+	0.4	+	0.05	+	0.006	

Final value = 123.456

Converting a Binary Fraction to Decimal To convert the binary fractional number (1011.101)2 to a decimal fraction, use the same technique but with 2 as the base instead of 10. Note that 2^{-1} is 0.5 and 2^{-3} is 1/8, i.e. 0.125:

$$(1011.101)2 = 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 1 + 2^0 \times 1 \; . \; 2^{-1} \times 1 + 2^{-2} \times 0 + 2^{-3} \times 1$$
$$= 8 + 0 + 2 + 1 + 0.5 + 0.0 + 0.125$$
$$= 11.625$$

Converting an Octal Fraction to Decimal To convert the octal fraction (0.375)8 to a decimal fraction, use 8 as the base. In this case, the converted fraction needs 9 decimal places for accurate representation:

$$(0.375)8 = 8^{-1} \times 3 + 8^{-2} \times 7 + 8^{-3} \times 5$$
$$= 1/8 \times 3 + 1/64 \times 7 + 1/512 \times 5$$
$$= 0.494140625$$

Converting a Hexadecimal Fraction to Decimal The base here is 16, so the fraction (0.F2)16 is converted to decimal in the following manner:

$$(0.F2)16 = 16^{-1} \times 15 + 16^{-2} \times 2$$
$$= 1/16 \times 15 + 1/256 \times 2$$
$$= 0.9453125$$

This number needs seven places of decimal for accurate representation. While conversion to decimal is always accurate, in some cases "premature" rounding off may be necessary for convenience of handling.

3.7.2 Converting a Decimal Fraction to Binary, Octal and Hexadecimal

Decimal fractions are converted using separate techniques for the integral and fractional components:

- The integral portion is converted by repeatedly dividing it by the base of the converted number (2, 8 or 16), and then reversing the remainders *(3.2.2)*.
- The fractional portion is converted by repeatedly *multiplying* it by the new base and collecting only the integral portion of the product. The multiplication sequence continues until the fractional portion becomes zero or is terminated at some point by human intervention.

Converting a Decimal Fraction to Binary To convert the decimal number (12.875)10 to a binary fraction, repeatedly divide 12 by 2 and repeatedly multiply 0.875 by 2. The division stops when the quotient drops to zero, and the multiplication stops when the fractional portion becomes zero:

Integral Component	*Fractional Component*
2|12	0.875 × 2 = 1.75 = 1 + 0.75
6 Rem 0	0.75 × 2 = 1.50 = 1 + 0.50
3 Rem 0	0.50 × 2 = 1.00 = 1 + 0.00 *Stops here*
1 Rem 1	Fractional value: 0.111
0 Rem 1	
Integral Value: 1100	Final value: 1100.111

The first technique is familiar to us; repeated division of 12 by 2 yields 0011 as a set of remainders, which when reversed becomes 1100, the binary equivalent of the integral portion. The fractional portion, 0.875, is treated differently; it is repeatedly multiplied by 2 and involves the following steps:

- The first iteration creates the product 1.75. The 1 becomes the first fractional digit but it is ignored for subsequent computation. The number 0.75, and not 1.75, is moved down for the next iteration.
- The second iteration multiplies 0.75 by 2 to create the product 1.50. The 1 forms the second fractional digit. The number 1.50 is then reset to 0.50 and moved down for the next iteration.
- Finally, 0.50 × 2 yields 1.00 with no fractional portion. Further multiplication is thus halted. At this point, the third fractional digit is also 1.

The fractional value obtained in the three steps is 0.111 which is the binary equivalent of (0.875)10. This value is appended to the converted integral portion, so (12.875)10 = (1100.111)2.

Converting a Decimal Fraction to Octal In this example, we'll focus only on the fractional component to convert (0.1640625)10 to octal. Multiply this number repeatedly by 8, and at each stage, collect the integral portion of the product before truncating it:

```
0.1640625 × 8 = 1.3125 = 1 + 0.3125
0.3125 × 8 = 2.5 = 2 + 0.50
0.50 × 8 = 4.00 = 4 + 0.00                    Stops here
```

Three rounds of multiplication created the whole numbers 1, 2 and 4, which represent the fractional part in octal. Thus, (0.1640625)10 = (0.124)8.

Converting a Decimal Fraction to Hexadecimal Let's now convert the same number, (0.1640625)10, to hex. This time, the multiplier is 16:

```
0.1640625 × 16 = 2.625 = 2 + 0.625
0.625 × 16 = 10.00 = 10 + 0.00 = A
```

The product obtained in the second round is 10, which is A in hex. Thus, (0.1640625)10 = (0.2A)16.

For the last two examples, the number 0.1640625 was carefully chosen (source: Wikipedia) so that multiplication terminates after two or three rounds. In most cases, multiplication of this type creates patterns of repetitive behavior that continue indefinitely. In such an event, terminate this process when the desired level of accuracy has been attained. Unlike in integer conversion, fractional conversion is seldom accurate.

3.7.3 Converting between Binary, Octal and Hexadecimal Fractions

The third category of conversion is the simplest of them all because the bases are all powers of two. Conversion here doesn't involve repeated division or multiplication, but simply a rearrangement and regrouping of bits. You are quite familiar with the techniques involved here *(3.5.3 and 3.5.4)*.

Conversion Between Binary and Octal Fractions To convert the number (1001.1001)2 to octal, we need to create groups of 3 bits because an octal digit is created from 3 bits. The bit allocation and their regrouping are shown in the following:

Binary	1001.1001	
Adjusted Binary	001 001 . 100 100	*Right-padding crucial here*
Octal equivalent	1 1 . 4 4	

Note that the integer and fractional portions have identical bit patterns but their bit allocations are different. The integral portion is left-padded (001 instead of 1) and the fractional portion is right padded (100 instead of 1) before they are converted to octal. Thus, (1001.1001)2 = (11.44)8.

For reverse conversion, say, (57.123)8 to binary, we split each octal digit into 3 bits and then simply combine these bits:

Octal	57.123
Binary	101 111 . 001 010 011

As long as each octal digit can be split into 3 bits, no extra padding with zeroes is required here. So, (57.123)8 = (101111.001010011)2.

Conversion Between Binary and Hexadecimal Fractions The key task here is to organize the bits in groups of four because one hexadecimal digit can be represented by 4 bits. The following examples represent binary-hex and hex-binary conversions:

	Binary		*Hexadecimal*
Original	10111001.1101011	Original	0.A3F
Adjusted Binary	1011 1001 . 1101 0110	Split Binary	0.1010 0011 1111
Converted Hex	B 9 . D 6	Combined Binary	0.101000111111

Using these simple regrouping techniques, (10111001.1101011)2 is converted to (B9.D6)16, and (0.A3F)16 is converted to (0.101000111111)2.

Conversion Between Octal and Hexadecimal Fractions For conversion from octal to hex, split the octal fraction into groups of 3 bits before regrouping them into groups of 4 bits. The reverse is true for hex-to-octal conversion. The following examples convert (0.776)8 to hex and (0.4F8)16 to octal:

	Octal to Hex		*Hex to Octal*
Original Octal	0.776	Original Hex	0.4F8
Split Binary	0. 111 111 110	Split Binary	0. 0100 1111 1000
Regrouped Binary	0. 1111 1111 0000	Regrouped Binary	0. 010 011 111
Converted Hex	0. F F 0	Converted Octal	0. 2 3 7

Thus, (0.776)8 is equivalent to (0.FF)16 and (0.4F8)16 converts to (0.237)8. Conversions of these types are indeed simple but remember that they apply to only to numbers whose bases are powers of 2.

Takeaway: There are two ways of converting numbers from one base to another. The first method involves repeated division (and repeated multiplication for the fractional component) and can be used only when one of the bases is *not* a power of 2. When both bases are powers of 2, then grouping and regrouping of bits represent the simplest solution.

3.8 ASCII CODES

Even though a computer eventually handles only numbers, what it takes in as keyboard input are characters and not numbers. Also, a computer writes the number 128 to the screen as three separate characters `1`, `2` and `8` which visually give us the impression of a number. A computer handles characters by their numeric codes. The standard codification scheme used today is the *ASCII code* (ASCII—American Standard Code for Information Interchange). There have been other systems like EBCDIC, but ASCII is now the defacto standard for character codification.

The *standard ASCII* set comprises 128 characters where each character is represented by 7 bits ($2^7 = 128$). This set includes the English alphabetic letters (both upper and lower), the numerals 0 to 9, several symbols and some non-printable characters. Table 3.2 shows the ASCII values of some of these characters. The letter '`A`' has the ASCII value 65 while '`a`' has the value 97. Digits have even lower values. The character `'1'` has the ASCII value 49, so when you key in `1`, it's not binary 1 that goes to the CPU but the binary value of 49. The complete 7-bit standard ASCII list is presented in Appendix B.

Because of the constraints imposed by 7 bits, it was not possible to include the key symbols of the major European languages in standard ASCII. Several companies made their own extensions to standard ASCII by incorporating an eighth bit which made it possible to represent 256 characters ($2^8 = 256$). Finally, ISO (International Standards Organization) came up with a standardized codification scheme that used 8 bits. Their scheme is officially called *ISO 8859-2* or *ISO Latin-1*, but is often loosely (and incorrectly) termed as *extended ASCII*. Your PC or laptop supports the ISO 8-bit set.

Note: C programs that sort character data use the ASCII code list to determine the sort order. It is thus helpful to know that numerals come first, followed by uppercase and then lowercase letters in this list.

TABLE 3.2 Selected ASCII Codes (Complete List in Appendix B)

ASCII Values	*Character*
0	NUL
1-26	*[Ctrl-a]* to *[Ctrl-z]*
48-57	0 to 9
65-90	A to Z
97-122	a to z

3.9 LOGIC GATES

We know the rules of binary arithmetic, but how does a computer actually perform addition and multiplication? How does it sum two binary numbers 1 and 1 to obtain 10? The answer lies in the use of *logic gates*, which are the basic building blocks of digital systems. Logic gates are based on Boolean algebra, a system that owes its origin to George Boole. *Boolean expressions* use variables

that have *logical* rather than actual values. This fits perfectly with our binary system whose values 0 and 1 can be treated as logical values for the purpose of evaluating a boolean expression.

A logic gate is a miniaturized electronic circuit having one or two lines for logical input and a single line representing the logical output. Both input and output can have one of two values, 0 or 1. The relationship between input and output is determined by the type of the gate (like AND, OR) and its associated *truth table*. This table shows the possible input combinations and the output for each of these combinations. Since most logic gates have two inputs, their truth tables show four possible input-output combinations.

Integrated circuits and microprocessors contain millions of these gates. Acting in tandem, they handle all computational and decision-making tasks required by computer programs. There are basically three and a total of seven types of logic gates which are identified by standard symbols. In this section, we examine the truth tables of all the following seven gates:

- The AND gate and its complement NAND (Not AND).
- The OR and XOR gates and their complements, NOR (Not OR) and XNOR (Not XOR).
- The NOT gate.

Except for the NOT (inverter) gate which has a single line for input, all the other gates have at least two lines for input. We will, however, restrict the number of input lines to two except for the NOT gate which by definition has only one input line.

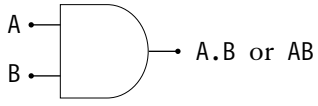
3.9.1 The AND, OR and XOR Gates

The AND and OR logical functions are quite common in the programming world. The logic of these functions can be found in the truth tables of their gates. The XOR gate is a simple variant of the OR gate.

The AND Gate As shown in Figure 3.1, the AND gate is represented by its own unique symbol along with two input lines (A and B) and one output line (C). The output of this gate is represented in the following manner:

```
C = A.B or C = AB                    Dot is the AND operator
```

As the associated truth table shows, the output is 1 (true) only when both inputs are 1. Note that the boolean expression for the AND gate is represented by A.B or simply AB (without the dot).

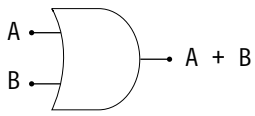


Truth Table

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

FIGURE 3.1 The AND Gate

The OR Gate For the two-input OR gate (Fig. 3.2), the output is 1 when either of the inputs is 1. The boolean expression for the OR gate is A+B. Note that the output is 1 even when both inputs are 1, the reason why this device is also known as the inclusive-OR gate.

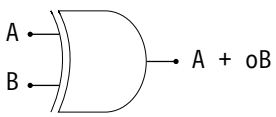


Truth Table

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE 3.2 The OR Gate

The XOR Gate The XOR gate (or exclusive OR gate) behaves differently from the OR gate only when both inputs are 1 (Fig. 3.3). When both A and B are 1, the output is 0 for XOR but 1 for OR. This means that the output in XOR is 0 when both A and B are either 0 or 1. The boolean expression for the OR gate is A+oB.



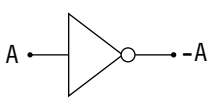
Truth Table

A	B	A+oB
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 3.3 The XOR Gate

3.9.2 The NOT Gate

Of the seven gates discussed here, the NOT or inverter gate is the only one which has a single input line (Fig. 3.4). The output of this gate is simply the inverse of its input. Thus, 1 becomes 0 and vice versa.



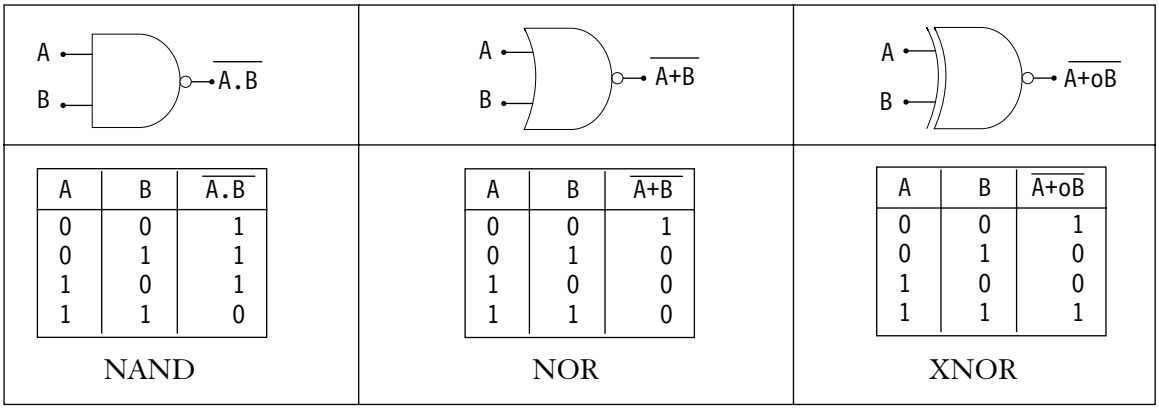
Truth Table

A	-A
1	0
0	1

FIGURE 3.4 The NOT (or Invertor) Gate

3.9.3 The NAND, NOR and XNOR Gates

The AND, OR and XOR gates have complements that use the "N" prefix. NAND inverts the logic used by AND, while NOR and XNOR have the same relationship with OR and XOR, respectively. The "N" gates use the same symbols as their "normal" counterparts but with a small bubble at the output end (Fig. 3.5). Their boolean expressions use the bar symbol at the top to indicate negation of the entire expression.



A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

NAND

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

NOR

A	B	$\overline{A+oB}$
0	0	1
0	1	0
1	0	0
1	1	1

XNOR

FIGURE 3.5 The NAND, NOR and XNOR Gates

These complementary gates also have reversed truth tables. This means that the functionality of NAND can be achieved by simply connecting the output of the AND gate to a NOT gate. The NAND and NOR gates are also known as *universal gates* because the other gates can be implemented using either of these two gates.

Note: In electronics, the logical values, 0 and 1, are implemented by applying a low voltage (around 0 volts) or a high voltage (around 5 volts) to a transistor. Most digital circuits use the "positive logic" system where 0 (false) represents a low voltage and 1 (true) represents a high voltage.

3.10 PROGRAMMING METHODOLOGIES

Programs today are far more complex than they were a few decades ago. Like languages, programming methodologies have also evolved to progressively offer improved design techniques for creating programs that are easy to understand, free of bugs and convenient to modify. Wherever possible, programs should also be designed to be *reusable*—even in other projects. Adherence to these objectives speeds up software development resulting in programs that are efficient and easily maintainable.

Most modern languages are *procedural*; they enable the execution of a section of code by invoking the name assigned to the section. They also have separate names for these sections—like *routines*, *subroutines*, *procedures* or *functions*. Another set of languages (like C++ and Java) added the *object* flavor to change the programming paradigm altogether. These features allow the division of code into separate *modules* using the following programming methodologies to choose from:

- A problem can be approached from the top to first visualize the "big picture" before breaking it up into separate components (top-down programming).
- The individual components can be designed first without fully knowing the big picture. The components are later integrated to form the complete system (bottom-up programming).
- The problem can be conceived as a collection of objects where each object encapsulates both data and the method used to access the data (object-oriented programming—OOP).

Once the programming methodology is finalized, a suitable language has to be selected that will provide an optimum solution to the problem. Not all languages support object-oriented programming even though the vast majority of them support both the top-down and bottom-up mechanisms. In the following sections, we'll briefly review these concepts.

3.10.1 Top-Down Programming

Top-down programming is a style that reverse-engineers the finished product to decompose it to smaller, manageable or *reusable* modules. You can then invoke a module by its name to carry out the task that it represents. These modules are further broken down into sub-modules and the process continues until the lowest-level modules are elemental enough to be easily implemented. If the module is designed to be reusable, it can be invoked by other programs as well. These modules are known as *procedures* or *functions*.

Modules reduce complexity by encouraging the programmer to design a program in terms of its building blocks rather than actual code. For instance, a program can have separate modules that read a file, validate the data, perform calculations and finally print a report. In a C program, the top-down programming approach visualizes the entire work as comprising the following functions:

```
read_file();
validate_data(month, year);                    Data passed to function
result = process_calc();                       Data returned by function
print_report();
```

The top-down approach breaks up the complete system into *black boxes* whose contents are not known initially. The design of each black box is undertaken only when there is complete clarity on the input a black box receives and the output it produces. The top-down approach permits the independent development of modules by separate groups of programmers and developers. The lower-level modules are invoked (or called) by the higher level ones, and a simple "main" program calls all of these modules.

The downside to the top-down technique is that programming cannot begin until the entire system, or, at least a substantial portion of it, has been decomposed into its subsystems. Testing can thus be carried out only after a significant portion of the system has been designed.

3.10.2 Bottom-Up Programming

Bottom-up programming is based on concepts that are opposite to the ones used by the top-down approach. Instead of visualizing the big picture first and then moving down to the sub-component level, the bottom-up style advocates the design of the low-level components first. These components are then pieced together—often across multiple levels—to form the complete system. The advantage of this approach is that the design group can start coding some of the modules even if the complete system has not been fully understood.

Bottom-up programming is usually adopted when off-the-shelf components in the form of reusable code are already available. It's common for a new system to be similar to one that has previously been tested and implemented. It turns out that many of the modules of the older system can be reused in the new system with little or no modifications.

Most software development usually combines a mix of both top-down and bottom-up approaches. In both systems, modules are arranged in a hierarchical structure where the top-level module "calls" a lower-level one. This is possible only when the programming language supports this "calling" feature. Procedural languages like C support this feature. C calls a function by its name to execute a block of code that is associated with that name.

3.10.3 Object-Oriented Programming

Both procedural and non-procedural languages treat a program as a set of instructions that act on data. *Object-Oiented Programming (OOP)* treats an application as a collection of *objects* instead. No separate instructions are needed because these objects encapsulate both data and instructions (called *methods*) to access and manipulate the data. Because objects are reusable and easily interact with one another, object-oriented programming is used for large and complex projects.

The procedural mindset needs to change to perceive an application in terms of objects. For instance, a shopping-cart in an online application is represented in procedural programming as a collection of several variables and procedures as standalone entities. In OOP, the cart is an object that contains (i) data represented by the item description, their price and quantity, (ii) the methods that would display, add or delete items, change their quantities or shipping details. Another application can reuse this object with or without modifications.

The object is initially designed as a *class* which is simply a template from which multiple objects can be created. Apart from encapsulation, the OOP model also relies on *inheritance* and *polymorphism* that help make programs clear and flexible. The top-down and bottom-up approaches work in OOP as well, but the focus is always on the class or object.

Note: It is possible to use an object-oriented language like C++ and Java as a procedural language. However, this defeats the very purpose of using OOP.

3.11 STRUCTURED PROGRAMMING

Structured programming strives to achieve clarity, faster development times and easy maintainability of programs. It is an offshoot of procedural programming but goes beyond the use of subroutines or functions to create modular programs. Structured programs use *control blocks* (a group of statements that are treated as a single statement) and loops (the repetition of one or more statements). These constructs are meant to replace the age-old GOTO statement whose use has now been deprecated (somewhat questionably though).

The foundations of structured programming can be traced back to the mathematicians Bohm and Jacopini, who proved in their *structure theorem* that any computing problem can be solved by using only the following constructs:

- *Sequencing* (the execution of statements in sequence).
- *Decision making* (selection from among alternatives).
- *Repetition* (running one or more statements in a loop).

Dijkstra followed subsequently by advocating the removal of the *GOTO* statement from all programs. It was felt that use of this statement leads to unstructured ("spaghetti") code that makes programs difficult to understand and modify. All modern languages have the three essential constructs though the *GOTO* statement has not been completely done away with.

Most languages support one or more of the *WHILE, FOR and UNTIL* loops to implement repetition. The purists advocate a single point of exit from these loops. For instance, the loop used in C, **`while (c < 50)`**, terminates only at the point of testing the condition (`c < 50`). However, the architects of C have included the **`break`** keyword which abnormally terminates the loop, thus providing an alternative point of exit. Opinion is divided on whether this feature violates the philosophy of structured programming.

3.12 ALGORITHMS

Two of the most useful tools needed by a programmer at the pre-programming stage are the algorithm and its pictorial counterpart, the flowchart. An *algorithm* is a finite set of clear and unambiguous steps that must be followed to solve a computing problem. The programmer eventually converts the algorithm to a program in *any* language. *Any computing problem, regardless of its complexity, can be represented by a program only if the problem can also be represented by an algorithm*.

An algorithm follows no established standards in the choice of keywords, but the symbols used for computation (like + and -) and comparison (like > and <) are used universally. The *structure theorem (3.11)* mandates the use of three types of constructs (sequence, repetition and decision), so let's have a look at their related keywords that will be used in the book. Steps are numbered using a convention followed by the BASIC interpreter language, but you are free to adopt one that you find comfortable to work with.

3.12.1 Sequencing

We begin with a simple algorithm that requires a set of steps to be performed sequentially. The following algorithm accepts two integers from the keyboard and computes the sum of their squares:

```
10 start
20 input n1, n2                    Keyboard input assigns two variables
30 sum = n1 * n1 + n2 * n2
40 print sum                       Displays output on the screen
50 stop
```

We'll follow the convention of using the **`input`** statement for accepting input from the keyboard and the **`print`** statement for displaying output. The computation is carried out using three variables, `n1`, `n2` and `sum`. All of our algorithms begin with **`start`** and end at the **`stop`** statement. A programming language will have its own keywords representing **`input`** and **`print`** though it may use the same variable names.

3.12.2 Decision Making

Real-life situations are complex and can't be handled by sequencing alone. We often need to make decisions based on the outcome of tests made on the data. The most common decision-making construct used in programming languages is the **if** or **IF** statement, and we'll use it for our algorithms as well. The following algorithm accepts an integer from the user, checks whether it is greater than 47 and prints a suitable message:

```
10 start
20 input n
30 if n > 47 then                          n > 47 is a relational expression
      print the number
   else
      print an error message
   endif                                   Terminates the if statement
40 stop
```

This is a typical **if-then-else-endif** construct that implements two-way decision making. The condition `n > 47` is a *relational expression* that evaluates to either a true or false value. For multi-way decision making, you need to "nest" multiple **if** statements in a single **if** construct (**if-then-else-if-then** ..... **endif**).

3.12.3 Repetition

How does one calculate the sum of the first 100 integers? We can use the repetitive feature to add a number to a variable and repeat the exercise 100 times by incrementing the number each time. The most commonly used repetitive construct is the **while** statement. Some languages also support variants like the **for**, **until**, **do-while** and **repeat** constructs. The following algorithm does the job:

```
10 start
20 set n to 1
30 set sum to 0
40 while n <= 100                          Condition fails when n reaches 101
      add n to sum                         Both this statement and following ...
      add 1 to n                           ... are executed 100 times.
50 print sum
60 stop
```

This task needs the initialization of two variables, `n` and `sum`, which are set to 1 and 0, respectively. The **while** construct (a loop) first tests the condition `n <= 100`. The two statements following **while** are executed repeatedly until `n` changes to 101 when the relational expression evaluates to false. In these 100 iterations, every integer between 1 and 100 has been added to `sum`. Note that the entire process takes place in a non-interactive manner because no external input is supplied.

Algorithms are useful tools for a programmer provided they are small; a single-page algorithm is easily converted to a program. Because programs are designed to be modular, it makes sense to have separate algorithms for each module. However, the language of an algorithm is closer to a programming language than it is to a spoken language. This makes comprehension of an algorithm difficult for the end-user. In such situations, a diagrammatic representation could be a better option.

3.13 FLOWCHARTS

Flowcharts are used in industry to pictorially depict the flow of manufacturing and business processes. In the programming world, a flowchart presents a graphical view of an algorithm for ultimate conversion to a program. It clearly depicts the steps and traces the movement of the associated data. Programmers must feel equally at home with both flowcharts and algorithms. However, because of its intuitive property, a flowchart convinces the end-user more easily than an algorithm does.

A flowchart comprises a standard set of graphic symbols that represent the various elements of a program. Each symbol contains text that describes precisely the action that needs to be taken. The symbols have the following attributes:

- Rectangular box () This box specifies all processing activities (except decision making) like setting variables and evaluating arithmetic expressions.
- Parallelogram () All input/output operations (the **`input`** and **`print`** statements used in algorithms) are performed here.
- Diamond () This shape is meant for two-way decision making. True and false outcomes emanate from the bottom and right corners, respectively. Since repetition also involves decision making, the diamond is also used as a component of loops.
- Ellipse or rectangle with curved edges () They are reserved for the start and stop activities.

These symbols are connected by lines having arrows at one end, which serve to indicate the direction pursued by program logic. These arrows move in three ways:

- Down for sequential flow.
- Down and right for decision making.
- Down and up for repetitive behavior.

A few examples that follow will clear things up.

3.13.1 Sequencing

Sequencing (the first of the three tenets of the structure theorem) is represented by rectangular boxes for data processing and a parallelogram for input/output activities. Figure 3.6 shows the flowchart for computing and displaying the sum of the squares of three numbers input from the keyboard.

Programs manipulate data using variables but only after they have been *declared*. The first parallelogram accepts three numbers from the keyboard and the second one prints the computed value of `sum`.

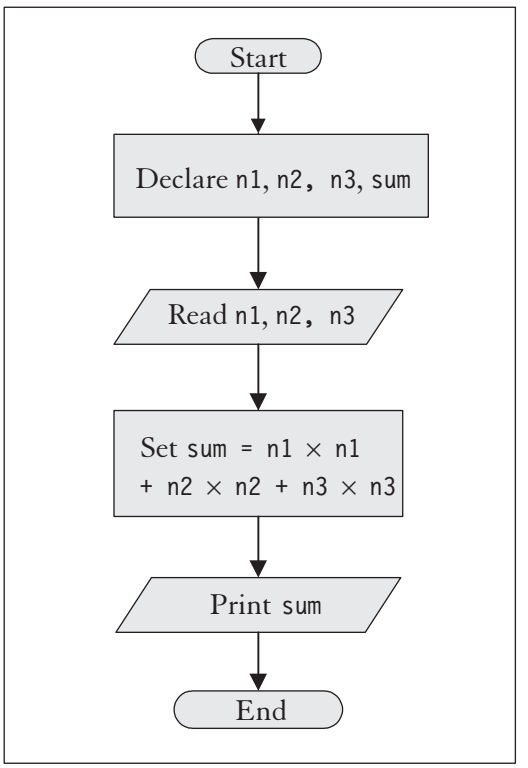


FIGURE 3.6 Sequencing

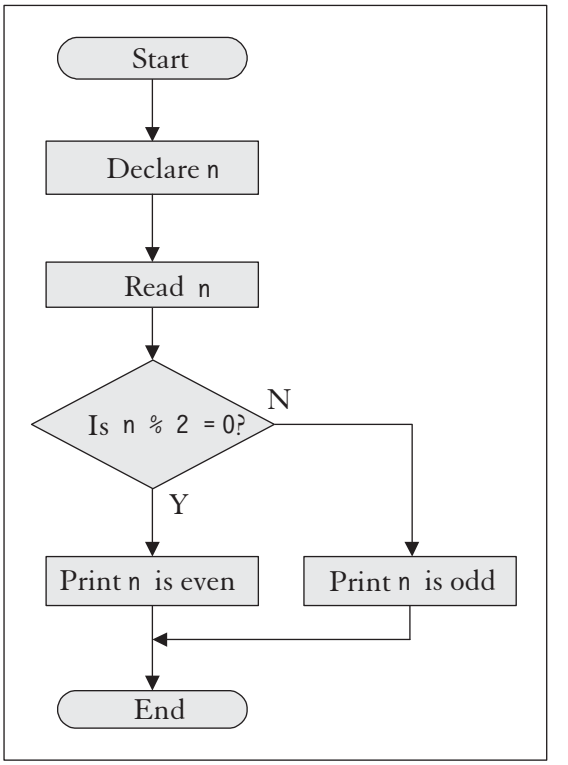


FIGURE 3.7 Decision Making

3.13.2 Decision Making

Decision making is represented by the diamond which accepts input from the top corner. For two-way decision making (say, for representing the condition `x > 5`), two arrows emanate from the bottom- and right-corner of this diamond. A third arrow is required for the left corner to represent situations where three outcomes are possible. Figure 3.7 shows the flowchart for determining whether a number is even or odd.

The `%` (modulus) operator is used to compute the remainder of a division, so `n % 2` evaluates to 0 or 1, while `n % 3` evaluates to 0, 1 or 2. Complex decisions that need multiple **`IF`** statements are represented by connecting multiple diamonds. Section 7.7 features a situation where multiple outcomes are generated from a nested **`IF`** structure.

3.13.3 Repetition

There is no standard symbol for representing repetition; a loop structure created by combining sequential and decision-making symbols repeatedly executes a set of instructions. Because the loop has to eventually terminate, a conditional check must be made either at the beginning or end of the iterative process. The flowchart shown in Figure 3.8 clearly depicts the methodology to be adopted for computing the factorial of a user-input number.

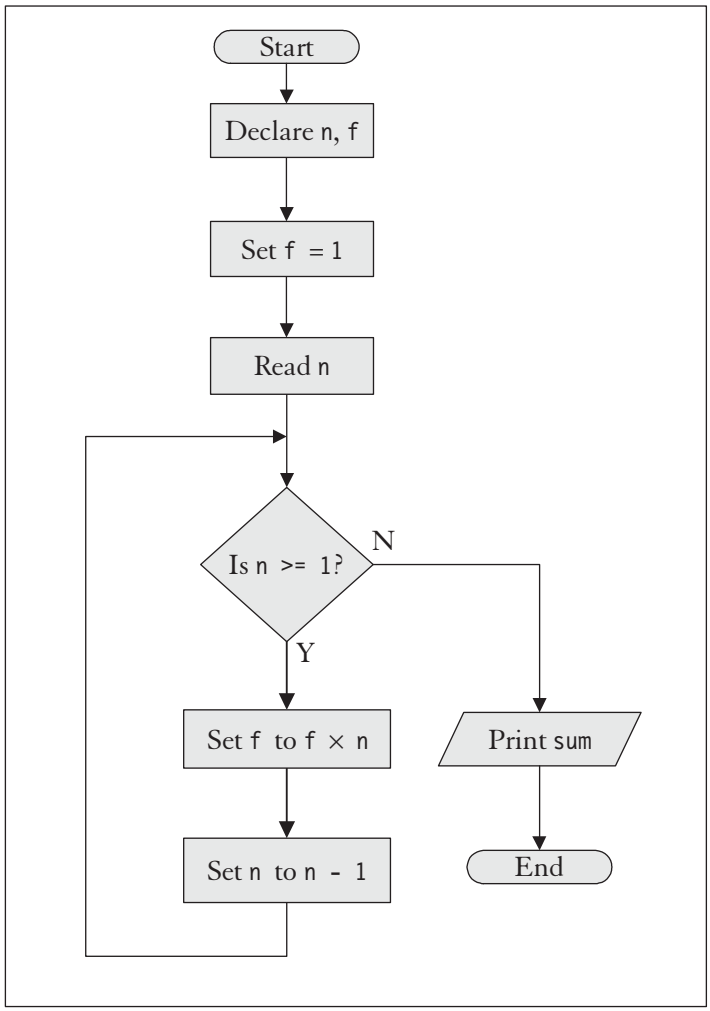


FIGURE 3.8 Repetition

The simplest method of computing the factorial (`f`) of a number (`n`) is to repeatedly multiply `n` by the next lower number until the multiplicand becomes 1. For `n = 6`, the progressive values of `f` thus become 6, 6×5, $6 \times 5 \times 4$, and so forth. This means that the value of `n` must be checked before `f` is computed. Note how a separate arrow at the side creates a loop. The decision to remain in or exit the loop is made at the beginning of the loop.

Both algorithms and flowcharts help in converting a computing problem to a program. Even though a flowchart is more intuitive, programmers must be comfortable with algorithms which take up less space and are easier to modify than flowcharts. Both tools are useful for debugging programs but flowcharts are often the preferred choice for creating program documentation.

3.14 CLASSIFICATION OF PROGRAMMING LANGUAGES

In the beginning, computers were programmed in their native machine language using binary numbers that represented machine instructions, data and specific memory locations. Programming with 0s and 1s was both tedious and unproductive, so better solutions had to be found. For over half a century, computer programming has evolved to create several languages that may be classified according to the following types and *generations*:

- Assembly languages (2nd generation, 2GL)
- High-level languages (3rd generation, 3GL)
- Fourth generation languages (4GL)

Machine code represents the first generation language (1GL). This is the actual code executed by the CPU and is consequently the most efficient of all languages. *Nothing can run faster than a program written in machine language*. Programs written in other languages must eventually be transformed to machine code (by compilation, for instance). However, the machine code of one computer will not run on another because different machines have different instruction sets which have different binary codes. In programming parlance, we say that machine code is not *portable*.

A higher generation represents three features: greater power, convenience of programming and reduced proximity to hardware. For example, a statement of a 3GL language (like C) easily replaces several instructions of a 2GL language (power). A 3GL statement is also closer to our spoken languages than a 2GL statement (convenience). Finally, a 3GL statement makes less assumptions about the underlying hardware than a 2GL instruction does (reduced proximity). We'll examine these generations in the forthcoming sections.

3.14.1 Assembly Language (2GL)

Assembly language (2GL) improves upon machine language by replacing 0s and 1s with intuitive keywords and symbols. You can say that assembly code is merely an easy-to-understand representation of machine code bound by a one-to-one relationship. This means that a single assembly instruction translates to a single machine (code) instruction. A special program named *assembler* translates assembly source code to machine code which is then run on the CPU.

Assembly language programmers use words like **mov** and **add**, rather than their equivalent machine codes, to represent key operations. Here are two commonly used assembly instructions:

```
mov al, num1
add al, num2
```

This code is hardware-dependent, so it won't work on a machine that uses a different processor. On the plus side, assembly code is most efficient because it directly accesses the hardware (close proximity). For this reason, it is considered a *low-level* language. The hardware vendor specifies the instruction set in assembly language for a particular machine.

Before high-level languages made their appearance, assembly language was used to code all types of programming tasks. The now obsolete MSDOS operating system and spreadsheet Lotus 1-2-3 were written in assembly language. Today, this language operates in a restricted domain—mainly for tasks that require direct access to hardware.

3.14.2 High-Level Languages (3GL)

A *high-level language (3GL)* moves closer to spoken languages and away from the hardware. It uses a small set of *keywords* along with many symbols found on the keyboard. 3GL has a one-to-many relationship with 2GL, which means that a single high-level *statement* corresponds to multiple assembly instructions. The following snippet of 3GL source code adds two numbers:

```
int n1 = 7;
int n2 = 9;
int n3 = n1 + n2;
printf("Sum of 7 and 9 is %d\n", n3);                    This is C!
```

Like an assembler, another special program—the *compiler*—converts the source code of high-level language programs to machine code. The machine code is held in a separate *executable* file which is then run on the CPU. 3GL programs make no assumptions about hardware and usually conform to a standardized syntax. Thus, the above source code will work on *most* machines, but would require recompilation every time it encounters a different hardware.

Most of the high-level programming languages were developed in the last fifty years. BASIC, Fortran and COBOL ruled the 3GL world before the arrival of C. Object-oriented languages, which also belong to the 3GL family, like C++, C# and Java, followed thereafter. Today, high-level languages are used to code commercial and scientific applications as well as system software like operating systems.

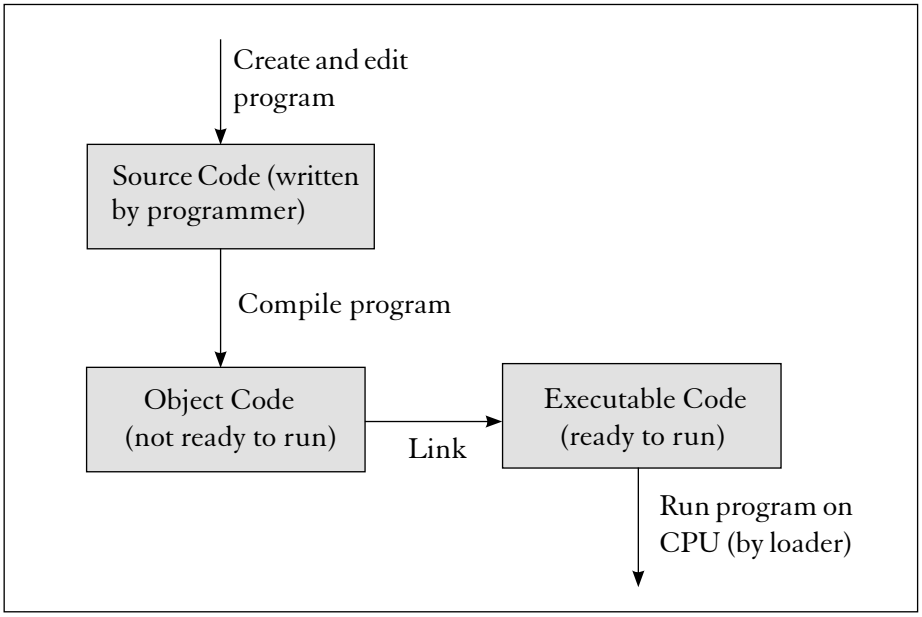


FIGURE 3.9 Role of the Compiler, Linker and Loader

Takeaway: Assembly language is not portable, either at source code or machine code level. However, when a program written in a high-level language (like C) is moved to a machine with a different processor, the source code normally needs no modification. But this code must be recompiled to generate the machine code for that machine.

3.14.3 4GL Languages

4GL languages continue the mission of high-level languages by moving even further away from the hardware and closer to natural languages. 4GL statements are very powerful; one 4GL statement often replaces an entire 3GL program. Compared to 3GL languages, programming in 4GL involves less rigor as is evident from the following 4GL statement which selects specific items from a database:

```
SELECT NAME, COMPOSITION, YEAR FROM COMPOSERS
       WHERE NAME = 'BEETHOVEN' OR NAME = 'MAHLER';
```

This is a statement in *Structured Query Language (SQL)*, a popular 4GL language used for several database products like Oracle and DB2. You need a hundred lines of C code to achieve the same task. Deployment of 4GL in areas once ruled by the 3GL world has resulted in very significant improvements in productivity and cost. All database products are bundled with a 4GL facility that can query, update and print information stored in the database. From a learning point of view, however, 4GL languages are not as challenging as the lower generation ones.

Note: To prepare you for what lies ahead, we'll henceforth refrain from using the term *instruction* for a high-level language, but use the word *statement* instead. We'll refer to machine and assembly language instructions and C statements.

3.15 COMPILERS, LINKERS AND LOADERS

Programmers using 3GL languages like C and C++ need to understand the roles of the compiler and linker in the compilation phase and the loader for executing the program (Fig. 3.9). These programs belong to the category of system software because they have knowledge of hardware built into them. However, they can be invoked directly by the user.

The *compiler* begins the task of translating the source code to machine or object code. It allocates memory addresses to variables and instructions but when functions or procedures are encountered, the compiler simply creates a table of *unresolved references*. The object codes for these functions are available externally in *libraries* or *archives* but the compiler goes no further than providing the references.

For instance, a C program frequently makes calls to the **`printf`** and **`scanf`** functions. The object code for these functions are available in a separate library, but the compiler doesn't look up the library. It's the *linker*—another separate program—that completes the unfinished work of the compiler. The linker looks at the table of unresolved references created by the compiler and then extracts the machine code from libraries or other files to create a single standalone executable. This executable can then be executed from the shell of the operating system.

When a user invokes the executable created by the linker, the *loader* reads the file and loads the instructions and data to memory. It then initiates the execution of the first instruction of the program. Unlike the compiler and linker, which are separate programs, the loader is part of the operating system. There is no separate program representing the loader that you can type in.

The linking and loading phases vary across compilers and operating systems. Linking need not necessarily occur at compile time. It can also be performed during loading or during program execution. Windows supports *Dynamically Linked Libraries (DLLs)* that allow linking to be done during program execution. UNIX/Linux systems support *shared objects* that perform a similar job. In either case, the library code is not permanently linked into the executable file.

Note: Many 3GL compilers (including C) perform the conversion of source code to machine language using two separate programs. The assembler converts the source code to assembly language, while the compiler converts the assembly code to machine language. The definition of the compiler changes here, but we'll address the issue in Section 4.2.3.

3.16 COMPILED vs INTERPRETED LANGUAGES

So far, we have classified languages by generation (2GL, 3GL and 4GL)—by their proximity to spoken languages and inverse proximity to hardware. We have also noted that programs need to be compiled to machine code before execution. Can we then say that all languages are compiled languages? No, a language can also be *interpreted*, where each statement of a program is converted and executed on the CPU before the next statement is taken up. Programs that perform both conversion and execution on the fly are known as *interpreters*. A language belonging to any generation may be interpreted or compiled depending on the way it is implemented.

BASIC is a 3GL language and some of us learned it decades ago using an interpreter. Every time a BASIC program was run, its interpreter had to be invoked. In most cases, no separate file is created for the code that runs on the CPU. This is not the case with a compiler or linker which creates a standalone executable—a separate file that runs independently. In fact, an executable can even run on another (compatible) computer that doesn't have the compiler installed. We prefer compilers to interpreters because compilers are flexible and versatile. Moreover, compiled code runs faster than interpreted code.

The distinction between interpreters and compilers are a little blurred today. Some modern-day languages like Java, C#, Perl and Python use a combination of both compilation and interpretation. The compiler creates an intermediate code, often called the *byte code* which is then executed by an interpreter. These languages differ in their handling of the byte code. Java, C# and Python create the intermediate code in a file, but only once (unless the program is modified), while Perl creates it in memory every time the program is executed.

Takeaway: A compiler creates machine code in a file but takes no further part in its execution. An interpreter normally creates no separate file but is involved in both conversion and execution. Any language can be interpreted or compiled or use a mix of both.

WHAT NEXT?

We now have all the ingredients in place for focusing on a specific third generation language that remains useful to this day. We'll now progressively unravel the features of the C language and learn to write intelligent C programs.

WHAT DID YOU LEARN?

The binary number system uses two digits (0 and 1). The C language also supports the *octal* (0 to 7) and *hexadecimal* systems (0 to 9 and A to F). The value of a number is the sum of the weighted values of the individual digits.

A decimal integer can be converted to binary, octal or hex by repeatedly dividing the number by the base and reversing the remainders. For numbers whose base is a power of 2, conversion is done by regrouping the bits.

Negative binary numbers are commonly represented by adding 1 to the *one's complement* of the positive number to generate its *two's complement*. Binary subtraction is performed by adding the two's complement of the subtrahend with the minuend and ignoring the carry.

Fractional numbers are handled by extending the weightage to use negative indices. A decimal fraction is converted to another base by repeatedly multiplying it by the new base and collecting only the integral portion of the product. However, if both bases are a power of 2, conversion is done by regrouping the bits.

In the ASCII code list, numbers come before uppercase letters, followed by lowercase ones. All printable characters are found in 7-bit ASCII (0 to 127).

All computation in the computer is carried out using *logic gates*, which usually have two inputs and one output. A *truth table* determines the output for various combinations of the input.

Top-down programming involves the decomposition of the entire problem into smaller manageable units. *Bottom-up programming* advocates the design of the low-level components before connecting them to form the complete system.

Structured programming is based on the thesis that any programming problem can be solved by using only three types of constructs that perform *sequencing*, *decision making* and *repetition*.

Algorithms specify a set of unambiguous instructions that are codified to solve a programming problem. A *flowchart* represents the graphical view of an algorithm.

Programming languages can be classified by *generations* which represent their proximity to spoken languages and reverse proximity to the hardware. C is a third generation language (3GL).

A *compiler* translates the source code to machine code containing *unresolved references*. A *linker* adds the code of functions to the machine code to create a standalone executable.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

3.1 The base of a number signifies the number of symbols or digits needed to represent it.

3.2 A number represented in binary can sometimes use more bits than its equivalent BCD form.

3.3 The one's complement of a number is obtained by inverting its bits and adding 1 to it.

3.4 There is only one zero in the two's complement system and two zeroes in the one's complement system.

3.5 For converting a number from one base to another, the repeated division technique is used only when one of the bases is not a factor of 2.

3.6 The output of an XOR gate is 0 if both inputs are 1.

3.7 All logic gates can be derived from the AND and OR gates.

3.8 It is not possible to start coding for a project without knowing it completely.

3.9 Any computing problem can be converted to a program even if the problem can't be represented by an algorithm.

3.10 Both assembler and compiler convert the source code to machine code.

3.11 One high-level language statement corresponds to multiple assembly language instructions.

3.12 The loader is a separate program in the OS that is invoked by the user to run a program.

A2. FILL IN THE BLANKS

3.1 Computing is efficient if the base of a number is a power of _____.

3.2 The maximum number that can be represented with 8 bits is _____.

3.3 The number $(255)_{10}$ is represented as _____ in BCD.

3.4 A negative binary number is formed by using all but the left-most bit to represent the _____.

3.5 All printable characters can be represented by _____ ASCII.

3.6 The output of the OR and XOR gates differ only when both inputs are _____.

3.7 A _____ language allows the naming of a section of code and invoking the code by that name.

3.8 _____ programming advocates the breaking down of the entire project into smaller modules.

3.9 The condition `x > 2` is a _____ expression which can have either a ____ or _____ value.

3.10 A flowchart is a pictorial representation of an _____.

3.11 The keywords, **`mov`** and **`add`**, are usually seen in _____ language.

3.12 The object code produced by a compiler must be processed by a _____ to create a standalone executable.

3.13 An _____ translates and executes each statement of the source code before retrieving the next statement.

A3. MULTIPLE-CHOICE QUESTIONS

3.1 The number 7011 can be (A) binary, (B) octal, (C) decimal, (D) hexadecimal, (E) B, C and D, (F) B and C.

3.2 The binary code of (66)10 is (A) 1000010, (B) 1000110, (C) 1011011, (D) 0111101.

3.3 Inheritance is a feature of (A) procedural programming, (B) non-procedural programming, (C) object-oriented programming, (D) none of these.

3.4 Program modularity (A) reduces the size of code, (B) makes code reusable, (C) makes maintenance easier, (D) B and C, (E) A, B and C.

3.5 Any programming problem can be solved using (A) repetition, (B) sequence and decision, (C) either A or B, (D) A and B.

3.6 Compared to assembly language, a third-generation language is (A) more powerful, (B) easier to program, (C) hardware independent, (D) A and B, (E) A, B and C.

3.7 Assembly language is not portable at the (A) source code level, (B) machine code level, (C) both, (D) none of these.

3.8 An unresolved reference indicates that (A) the program contains a syntax error, (B) a function with a wrong name has been invoked, (C) the code for a function has not been included in the object code of the main program, (D) none of these.

A4. MIX AND MATCH

3.1 Match the terms with their significance/attribute:

(A) XNOR, (B) functions, (C) diamond, (D) library, (E) OOP, (F) loop.

(1) Class, (2) repetition, (3) logic gate, (4) procedural language, (5) linker, (6) flow chart.

CONCEPT-BASED QUESTIONS

3.1 Convert the decimal number -130 to binary in (i) sign-magnitude form, (ii) one's complement form, (iii) two's complement form.

3.2 Evaluate the following expressions containing binary numbers: (i) 110011 + 101111, (ii) 1100111 - 11101, (iii) 101 X 1110, (iv) 110011 / 111

3.3 Convert the following integers from one base to another:

(i) (753)8 = (???)10, (ii) (491)10 = (???)8, (iii) (AF8)16 = (???)2, (iv) (65450)10 = (???)16, (v) (10111011)2 = (???)8.

3.4 Convert the following fractional numbers from one base to another:

(i) (10111.1001)2 = (???)10, (ii) (107.47)8 = (???)10, (iii) (107.47)10 = (???)16, (iv) (1111.111)2 = (???)8, (v) (0.1AF)16 = (???)2.

3.5 Develop an algorithm to sum the digits of a number.

3.6 Develop an algorithm and flowchart for sorting three integers.

3.7 What is the difference between a high-level and low-level language? Provide an example for each of these types.

3.8 Explain the difference between an assembler, compiler and interpreter.

4 A Taste of C

WHAT TO LEARN

- The three-phase compilation process that transforms a C source program to machine code.
- Use of *sequence*, *decision making* and *repetition* in programs.
- Handling simple errors generated by the compiler.
- Role of *functions* in making programs modular.
- Working knowledge of the **`printf`** and **`scanf`** functions.
- Macro view of the features of the C language.
- Commonly used styles that make programs readable and maintainable.
- Standardization efforts that have made C stable and acceptable.

4.1 THE C LANGUAGE

The C language was created by Dennis Ritchie of AT&T (now Lucent) fame in the 1970s. C was created with an unusual objective—not to replace Fortran or BASIC—but to rewrite the code for the UNIX operating system. In those days, operating systems were written in assembly language, but Ritchie's out-of-box thinking produced a language that was portable, compact and fast. In retrospect, we find that he achieved the best of both worlds. Today, the UNIX/Linux operating systems run on practically any hardware, while C is widely favored for programming.

But the C originally created for UNIX systems is significantly different from the one used on Windows computers (often, Visual Studio) or iPhones (Objective C). Ritchie's C could make direct calls (called *system calls*) to the UNIX operating system. The UNIX OS does all of its work using a standardized set of these system calls. However, non-UNIX systems don't use most of these calls, so UNIX C programs containing these calls would fail to compile on other systems. But C had too many advantages to be ignored for this reason alone.

When C went commercial, vendors made numerous changes to the language to let it run on other platforms. The original C gradually lost its identity until the American National Standards

Institute (ANSI) stepped in to standardize the language. The ANSI standard removed the system calls from UNIX C and retained a subset of the original language. This subset is now common to all systems, and in this book, we'll stick to this subset, which will be variously referred to as ANSI C, C89 or C99, the nomenclature used by ANSI.

C is a simple but powerful third-generation language. Its rich library of *functions* enables the development of modular applications. Its support of a variety of constructs for decision making (**`if`** and **`switch`**) and repetition (**`while`**, **`for`** and **`do-while`**) also makes it suitable for structured programming. But the language derives its power from the vast arsenal of *operators* that permit low-level operations like (i) direct access to memory, and (ii) manipulation of every bit in memory. In this respect, C is somewhat akin to a second-generation language like assembly language.

Today, C is widely adopted for developing a diverse range of both application and system software. Compilers for many languages (including C) are written in C. Since C programs are compact and efficient, it is the preferred language for software running on small devices like mobile phones and tablets. C has replaced BASIC and Pascal for introducing students to the programming world. It has also influenced the development of object-oriented languages like C++, C# and Java. It pays to know C because the language is not going to go away in a hurry.

Note: C can be considered to be both a high- and low-level language. While it is sufficiently low-level to enable access to specific memory locations, it is also a high-level language with all the standard constructs and operators found in any 3GL language.

4.2 THE LIFE CYCLE OF A C PROGRAM

In this section, we discuss the role of the agencies involved in making a C program run successfully. A program is conceived, created, transformed and eventually executed (Fig. 4.1). Errors invariably occur during (i) compilation on account of faulty use of syntax (grammar), (ii) execution because of erroneous program design. Before we discuss the compilation mechanism, it is necessary to discuss two important features of C that significantly impact the compilation process.

4.2.1 Header Files and Functions

Ritchie wanted C to be compact, so two features—directives and functions—were kept "outside" the language. For instance, most C programs have the line **`#include <stdio.h>`** at the beginning of a program. This is simply a *directive* to place the contents of the *header file*, `stdio.h`, in the program. A header file contains data used by multiple programs, so rather than include the data in the program itself, a one-line directive keeps things simple and manageable. Also, by judicious division of labor, C ensures that this line is processed by a *preprocessor* so that the compiler doesn't see it at all.

In C, most work is performed by *functions*. A function is called by its name to execute a set of statements. If multiple programs need to perform this task, then they can call the same function. This is possible only if the machine code for the function is available centrally and made accessible to all. For instance, when the compiler encounters the **`printf`** function, it knows that the code for **`printf`** is available elsewhere. Division of labor works here as well; the compiler takes a hands-off approach and leaves the job of including the function code to the *linker*.

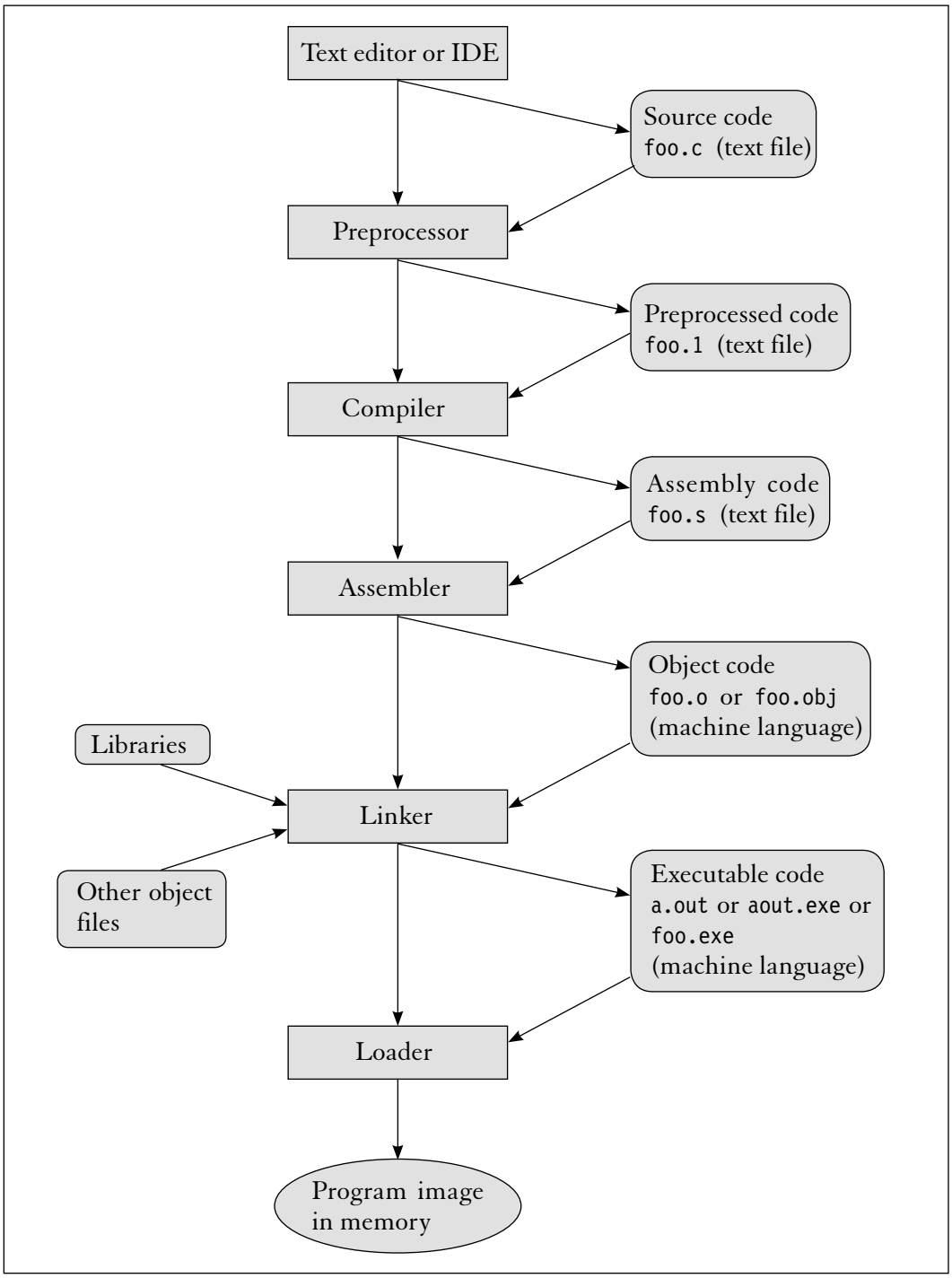


FIGURE 4.1 The Life Cycle of a C Program

4.2.2 Editing the Program

Before writing a program, you need to first identify its input and output and then develop the program logic. Once you have the algorithm or flowchart ready, key in the source code using an editing software. This editor may be provided by the operating system or it may be a component of the compiling software. Save the code in a file having the `.c` extension using a meaningful name (like **`celsius2fahrenheit.c`** instead of names like **`a.c`** or **`foo.c`**).

For a program to compile successfully, its source code must be *syntactically* correct, i.e. it must conform to the rules of the language. Make a small mistake (like dropping the **`;`** in a C statement) and your program will refuse to compile. Every time one or more errors are generated, you have to re-invoke the editor to carry out the changes. The file containing the source code must be backed up to allow for future modifications. If you lose this file, you cannot reverse-engineer the machine code to retrieve the source code.

4.2.3 The Three-Phase Compilation Process

As just noted, a C program depends on information maintained externally, so the compilation procedure in C is somewhat different from other languages. Compilation in C is a three-step process that involves the use of three separate programs:

- The *Preprocessor* This program acts on lines that begin with a **`#`**. They occur mostly at the beginning of a program and are called *preprocessor directives*. A directive may insert the contents of a header file at that location (like **`#include <stdio.h>`**) or perform a substitution (like **`#define PI 3.142`**) in the program itself. The preprocessor thus *modifies* the source code to make it suitable for use in the next phase. This modification is carried out on a copy; the original file containing the source code is left undisturbed.
- The *Compiler* This program checks the source code after it has been modified by the preprocessor. If the compiler detects one or more syntax errors in this code, it displays them and stops further processing. Otherwise, the compiler creates intermediate *object code*. This code is incomplete and not ready to run because it doesn't contain the code for the standard C functions (like **`printf`** and **`scanf`**). The compiler merely leaves *unresolved references* to these functions *(3.15)*.
- The *Linker* This program acts on the unresolved references left behind by the compiler. The linker does this by adding the machine code of functions available in separate libraries to the object code created by the compiler. The resultant code is a standalone executable that is run by the program loader. *The linker fills the holes left by the compiler.*

Even though every C distribution contains the three programs, they may or may not be invoked separately (unless you are handling an application that comprises multiple programs). Generally, these programs are invoked by one principal program, but it will stop in its tracks if it encounters syntax errors in the source code. For instance, the linker won't be invoked if the compiler detects syntax errors, which means that the executable won't be created. If that happens, you have to edit the source file and repeat this three-phase cycle.

Does the compiler directly translate the source code to machine code? Some compilers (like the Linux GCC compiler) find it more convenient to perform translation to assembly language first. A separate program (the *assembler*) then converts the assembly language code to machine code. The intermediate file is removed once the final machine code is created. However, GCC also supports an option that retains the assembly code for use by assembly language programmers.

Note: The term *compiler* is used in two ways in C. It could mean the program that does the job specified in the second phase, or represent a collective term that performs all of the three tasks. The true meaning should be evident from the context, but when it is not, we'll use the term *compiling software* to refer to the collective meaning.

4.2.4 Executing the Program

The default name of the standalone executable is system-dependent—**`a.out`** on UNIX/Linux systems, and **`aout.exe`** or **`AOUT.EXE`** on Windows. (Visual Studio names the executable after the source file, so **`foo.c`** is compiled to **`foo.exe`**.) This file can be executed by running **`./a.out`** or **`.\aout`** from the operating system's shell. If you are using C in a GUI environment (like Visual Studio), you have to use the mouse to select menu options that perform all of the compilation and execution tasks.

It is possible for a program to pass the compilation and linking phases and still fail in execution. Programs fail at *runtime* (i.e., when the program is actually run) on account of *semantic errors* that arise out of mistakes in design. For instance, a division by zero makes the program abort with an error. Use of `=` instead of `==` will also yield unexpected results. Faulty logic that doesn't account for all possibilities can make a program behave erratically as well. The solution is the same; edit the source code and repeat the compilation process before re-execution.

You might argue that if the contents of `stdio.h` are made available by a simple **`#include`** directive, why not a value (like `3.142` or `PI`) that is required by multiple programs involving circles? That's right, C doesn't encourage you to store the value of `PI` in every program that computes the area of a circle. You can keep this information in a header file and include it as well.

Takeaway: A program can have syntax errors (on account of wrong use of its grammar) or semantic errors (because of faulty program logic). The former is detected during compilation and the latter during program execution.

4.3 KNOW YOUR C COMPILING SOFTWARE

Before you start writing programs, you need to be familiar with the C compiling software (the entire set of programs) installed on your system. Over the years, the footprint of this software has grown in both size and convenience to provide a complete programming environment. This environment generally comprises the following components:

- The main programs—the preprocessor, compiler and linker.
- The standard header files interpreted by the preprocessor.

- The standard library containing the object code of all functions that are part of ANSI C (like **printf**).
- An editor to edit programs and header files.
- A debugger to locate semantic (runtime) errors in programs.

Compiling software is available in both CUI and GUI flavors. Command-line or CUI systems offer separate programs for editing, processing and debugging. These programs are invoked by name and they support several *options* or *switches* that change their default behavior. We won't need to use these options in this book though (except in Chapter 17).

GUI-based systems offer an *Integrated Development Environment (IDE)* where all actions are invoked by the point-and-click mechanism of the mouse. IDEs are menu-based which dispenses with the need to invoke commands by name. Behind the hood, however, these mouse clicks internally invoke standalone programs. Generally, IDE systems also offer a CUI mode for those (this author included) who prefer to invoke commands by name.

4.3.1 GCC (Linux)

Linux systems support a CUI system called GCC, the free-to-use package from GNU. It is shipped with every Linux distribution but may not be installed by default. It comprises the following standalone programs: **gcc** (compiler), **cpp** (preprocessor), **ld** (linker) and **gdb** (debugger). GCC doesn't offer an editor, but Linux supports two powerful editors that are used by most people: **vim** and **emacs**. To check whether GCC is installed on your system, run the **gcc** command from a shell:

```
$ gcc                          The $ signifies the shell prompt
gcc: no input files            gcc exists
```

This message from the **gcc** compiler points out that a filename needs to be specified. If the GCC package is not installed, you'll get the message `gcc: command not found`. **gcc** also acts as a frontend to **cpp** and **ld**, so you don't normally have to bother about running the preprocessor and linker separately. By default **gcc** creates the executable **a.out** which has to be executed as **./a.out**.

4.3.2 Microsoft Visual Studio (Windows)

Unlike in Linux, C compiling software is not native to the Windows operating system. But numerous third-party packages are available including the GCC package that has also been ported to this platform. For this book, we'll refer to the popular GUI-based Visual Studio package, where the *Build* menu in the menu bar offers all the necessary processing options. The essential programming activities and their associated menu options are listed below:

- Compile the program with *Build* > *Compile*.
- Invoke the linker with *Build* > *Build*.
- Execute the program with *Build* > *! Execute*.

Unlike GCC, Visual Studio supports a built-in editor having limited capabilities. It also permits command-line operation from an MSDOS shell. The commands to compile and link are **cl.exe** and **link.exe**, respectively. Like **gcc**, **cl.exe** directly invokes the linker (**link.exe**) to create the

executable. Unlike other Windows compilers that generate **aout.exe** as the default executable, Visual Studio names the executable after the source filename. The source file **first_prog.c** is compiled to **first_prog.exe**.

Note: Windows executables have the .exe extension. Depending on the compiler used, the program **foo.c** is converted to the executable **aout.exe** or **foo.exe**. Also note that Windows filenames are not case-sensitive, so **foo.c** and **FOO.C**, which are treated as separate files by Linux, will create conflicts when moved to Windows.

4.4 first_prog.c: UNDERSTANDING OUR FIRST C PROGRAM

Our first program (Program 4.1—**first_prog.c**) sequentially executes a set of statements that perform a simple arithmetic operation using two variables. The output is displayed on the screen. This program is not designed to pause for taking user input from the keyboard; all data are provided in the program itself. A cursory glance reveals a typical two-part structure comprising the *preprocessor* and *body* sections. The spirit of C is evident in the annotations and program output.

```
/* first_prog.c: Declares and assigns variables.
                 Also features some simple computation. */

/* Preprocessor Section */
#include <stdio.h>            /* printf needs this file */
#define PI 3.142              /* PI is a constant */

/* Program Body */
int main(void)                /* This is where the action begins */
{
   /* Variable declarations */
   int radius = 5;            /* Integer type */
   float area;                /* Floating point type */

   /* Other Statements */
   printf("All numbers are stored in binary\n");
   printf("Radius of circle = %d\n", radius);
   area = PI * radius * radius;
   printf("Area of circle having radius %d = %f\n", radius, area);

   /* The return statement explicitly terminates a program */
   return 0;
}
```

PROGRAM 4.1: **first_prog.c**

```
All numbers are stored in binary
Radius of circle = 5
Area of circle having radius 5 = 78.550003
```

PROGRAM OUTPUT: **first_prog.c**

4.4.1 Program Comments

Like every language, C supports comments in the source code. Comments help in making program maintenance easier, so a sensible programmer provides them at key points in a program. They are enclosed by the character sequence `/*` on the left and `*/` on the right. Comments can be single- or multi-line and they can also be placed after program statements. This program shows all of these instances.

Comments in no way affect the size or efficiency of the executable because they are removed by the compiler. While excessive use of comments is discouraged, complex logic must be properly documented in the program itself. Judicious use of comments will help you understand your own program when you look at it months or even years later.

4.4.2 Preprocessor Section

The preprocessor section that follows the initial comment lines contains two lines that begin with a # but don't end with a semicolon. They are not C statements because all C statements are terminated with a semicolon. (For confirmation, look at the lines that follow.) These lines, called *preprocessor directives*, generally appear before the occurrence of the word **`main`**. When the program is compiled, the *preprocessor* acts on these lines before the compiler sees them.

The first directive (**`#include <stdio.h>`**) tells the preprocessor to replace the line with the contents of the file **`stdio.h`**. This *header file*, which contains information required by the **`printf`** function, is shipped with every C distribution.

The next directive (**`#define PI 3.142`**) instructs the preprocessor to globally replace all occurrences of `PI` in the program with `3.142`. The compiler thus doesn't see `PI` but only `3.142`. Unlike `radius` and `area` which are variables, `PI` is a *symbolic constant*. It occurs once in the body—in the expression that calculates the area of a circle. We will have to understand why `PI` was not declared as a variable.

Note: Preprocessor directives don't end with a semicolon because technically they are not C *statements*. In fact, the compiler, which acts after the preprocessor has completed its job, doesn't see any comment lines, **`#define`** or **`#include`** directives.

4.4.3 Variables and Computation

The preprocessor section is followed by the program body, which begins with the word **`main`**. The body represents the meat of a C program where all work is actually done—assigning variables, making decisions, running loops and invoking function calls. The body contains *statements* that end with a semicolon. Here, it begins with two variable declarations:

```
int radius = 5;            /* Integer type */
float area;                /* Floating point type */
```

C is a *typed* language, which means all data have types assigned to them. The first statement declares a variable named `radius` having the type `int` (integer). The assignment symbol, `=`, is an *operator* which assigns the value 5 to `radius`. The next statement declares another variable named

area of type float (floating point). These C *keywords*, int and float, are two of the frequently used data types in C. At this stage, we have no idea what the initial value of area is.

The next statement (area = PI * radius * radius;) uses the multiplication operator, *, to calculate the area of a circle. The result is then assigned to the variable area. Note that the compiler doesn't see PI, but 3.142, since the preprocessor has already made this substitution. We have now seen the use of two members (= and *) of C's vast arsenal of operators.

Note: All variable declarations should be made at the beginning—after the preprocessor directives and before the statements.

4.4.4 Encounter with a Function: printf

C uses **printf** to output messages on the screen. Unlike int and float, which are C keywords, **printf** is a *function*. A C function is easily identified by the pair of parentheses that follow its name. The first **printf** uses a single *argument*—the string "All numbers are stored in binary\n". Note that **printf** doesn't print \n which it interprets as an *escape sequence*. In this case, \n moves the cursor to the next line, so the next **printf** can display output on a separate line.

The next two **printf** statements use two or three arguments delimited by commas. In each case, the first argument *must* be the string to be printed. But if this string contains a *format specifier* (like %d), then the next argument is printed at that location. %d is used to print integer values while %f handles floating point numbers. In these two statements, %d is the format specifier for radius, while %f is the specifier for area.

Note: **printf** will not insert a newline unless you explicitly direct it to do so. This feature enables printing of long strings with separate **printf** statements.

The main Function Is **printf** the only C function we find in this program? No, we should have known that the word **main** is actually a function; it too is followed by a pair of parentheses. For now, let's simply note that every program has a **main** function which is *automatically* invoked to execute the statements in the program body. We will discuss this function in Section 11.15.

4.4.5 The return Statement

Finally, we meet the **return** statement which is usually the last statement in the program body. Program execution terminates when this statement is encountered inside the program body. After we have acquired knowledge of functions, we'll use **return** inside functions also. For now, we'll ignore the significance of the value 0 but not fail to use **return** with this value.

With the limited knowledge acquired so far, we have somewhat understood how **first_prog.c** works. But for a genuine first-hand feel, we also need to key in the program and run it to obtain the output shown below the program listing. We'll do this after we have examined the software environment that we use to compile and run our programs.

4.5 EDITING, COMPILING AND EXECUTING first_prog.c

Make a separate directory on your system for the C programs that you'll develop. Using a text editor, key in this program to create the source file, **first_prog.c**. We use a text editor instead of a word processing software like Microsoft Word because C source programs are text, and not binary, files. Using the following guidelines would stand you in good stead:

- Indent the text as shown in the program listing. Proper indentation helps in both understanding and maintenance, so use the *[Tab]* (⇆)key to provide indentation.
- Don't fail to provide the **;** which is seen at the end of most lines.
- Don't terminate statements beginning with **#define** and **#include** with a semicolon.
- C is case-sensitive, so don't use uppercase except where shown.
- Don't use single quotes where double quotes have been specified, and vice versa.

After you have input your program, save it as **first_prog.c**. You are now ready to start the compilation process. Compiler-specific information follows next, so you will need to refer to either Section 4.3.1 or Section 4.3.2.

⚠ **Caution:** Remember to terminate every C statement with the semicolon. A compiler generates errors if it doesn't find it even though it may not properly identify the location where the semicolon should have been placed.

4.5.1 Using gcc (Linux)

To input your program, you may choose the **vim** or **emacs** editors, two of the most notable and powerful editors supported by Linux. (Versions for Windows are also freely available.) After saving the file, use the **gcc** command to compile **first_prog.c**:

```
$ gcc first_prog.c
$ _                                    Prompt returns; compilation successful
```

By default (as above), **gcc** first invokes the preprocessor (**cpp**), then does the job of compiling itself, and finally calls the linker (**ld**) to create the executable—all in a single invocation. The silent return of the prompt indicates that no syntax errors have been encountered and that the default executable file, **a.out**, has been created. Run the **ls** command now to verify the presence of this file.

You can now invoke the executable in this manner to produce the output shown after the program listing (Program 4.1):

```
$ ./a.out
All numbers are stored in binary
Radius of circle = 5
Area of circle having radius 5 = 78.550003
```

The output is correct, which means there are no semantic errors in your program. In case you see syntactical error messages, note the line numbers of the offending locations from these messages. Correct the source code and recompile the program. Handling error messages is taken up in Section 4.6.

4.5.2 Using Visual Studio (Windows)

As mentioned previously, Visual Studio supports both GUI and CUI modes of operation. Key in your program using the built-in editor of Visual Studio or Windows Notepad, and save the file as **`first_prog.c`**. The sequence of steps for each of these modes is explained next.

GUI Mode The *Build* menu in the Menu bar has all the necessary options for compilation and execution. In this mode, you need to invoke the compile and link phases separately:

- *Preprocess and Compile* Use *Build* > *Compile first_prog.c* and observe the window pane at the bottom. If you see compilation errors, note their line numbers, edit the source file and repeat the compilation. You may ignore the warnings for the time being. On successful compilation, you should see the message `first_prog.obj - 0 error(s)`, signifying that the object file, **`first_proj.obj`**, has been created. This file contains machine code but is not ready to run because it contains unresolved references *(3.15)*.
- *Link* Linking is done with *Build* > *Build first_prog.exe*, when you should see the message `first_prog.exe - 0 error(s)`. The executable, **`first_prog.exe`** has been created.

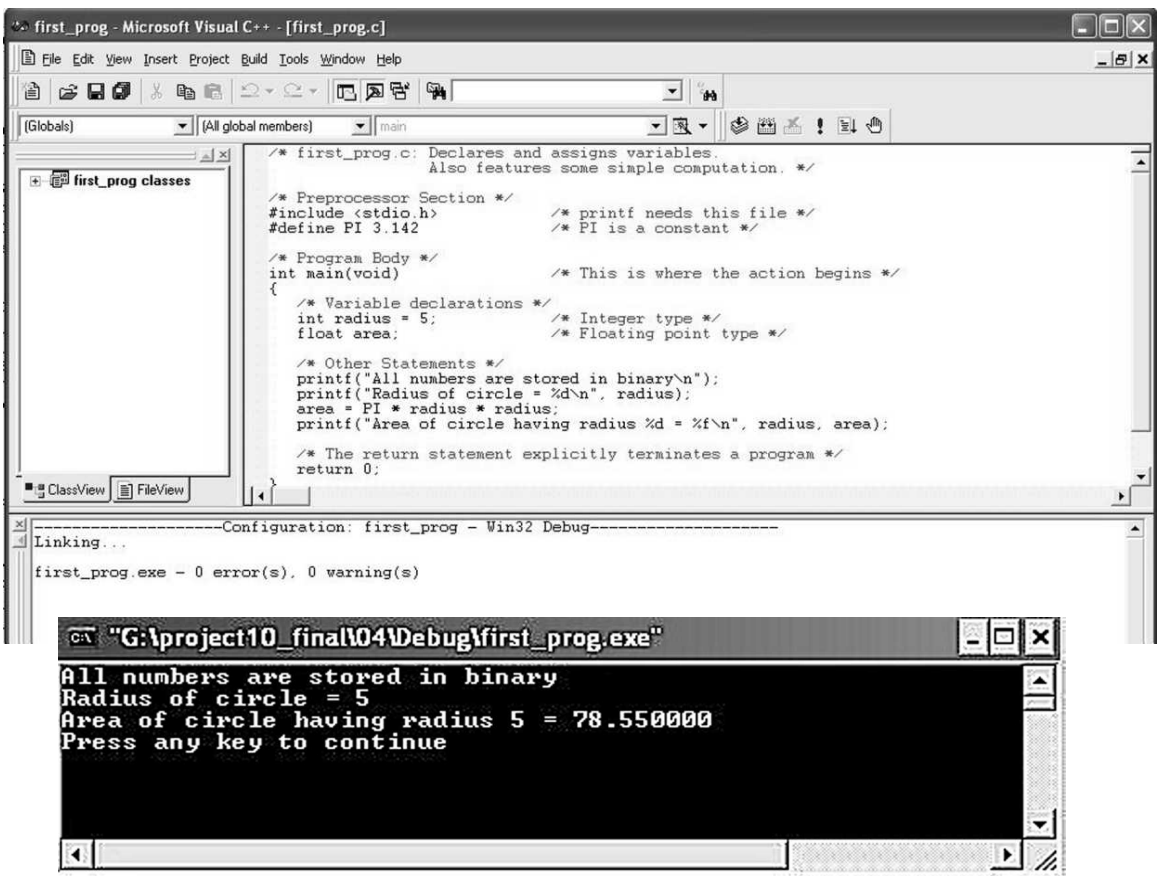


FIGURE 4.2 Visual Studio: Compiling and Executing a Program

You can now run this executable with *Build > ! Execute first_prog.exe*. A new window will open up with the output of the program. The screen dumps of these activities are shown in Figure 4.2.

Tip: Visual Studio offers shortcuts for all three activities: *[Ctrl][F7]* for compiling, *[F7]* for linking and [Ctrl][F5] for executing. Also, you'll find separate icons representing these activities in one of the toolbars.

CUI Mode Invoke the MSDOS shell or command prompt (using *Run > cmd*) and then invoke the following command:

```
C:\> cl first_prog.c
/out:first_prog.exe
first_prog.obj
```

The program **cl.exe** invokes **link.exe** to create the executable, which can then simply be run like this:

```
C:\> .\first_prog                                    Note the .\
All numbers are stored in binary
Radius of circle = 5
Area of circle having radius 5 = 78.550003
```

The output is correct, which means there are no semantic errors in the program. In case you encounter errors, locate the line numbers provided in the error messages to edit the source code. Recompile the program and re-execute **first_prog.exe**. Handling error messages is taken up in Section 4.6.

Note: The standalone executable (**a.out** or **first_prog.exe**) can now run on this and similar machines *even if the C compiler is not installed on these machines*.

4.6 HANDLING ERRORS AND WARNINGS

What you see is often not what you expected to see—at least not in the initial phase of the learning process. Errors are inevitable but you must able to detect and correct them. Programmers often forget to add the semicolon, and if you do that in the first **printf** statement of **first_prog.c**, this is how **gcc** reacts on Linux:

```
$ gcc first_prog.c
first_prog.c: In function main:
first_prog.c:18: error: expected ; before printf
```

There seems to be no ambiguity in the message; the compiler spots a missing semicolon on line 18. But, unexpectedly perhaps, line 18 contains a semicolon:

```
printf("Radius of circle = %d\n", radius);
```

Did the compiler get the line number wrong? Not really; it's just that the missing **;** on line 17 lured the compiler into assuming that lines 17 and 18 constituted a single statement. Since the statement terminator is a semicolon, the compiler keeps scanning until it finds one.

Visual Studio also displays a similar message, but it generates a warning as well:

```
first_prog.c(16) : warning C4244: '=' : conversion from 'double ' to 'float ',
                    possible loss of data
first_prog.c(18) : error C2146: syntax error : missing ';'
                    before identifier 'printf'
Error executing cl.exe.
first_prog.obj - 1 error(s), 1 warning(s)
```

This warning will make sense after you have learned that, like variables, constants have data types *(5.1.2)*. Here, the constant `PI` has the type `double`, which on this machine is longer than `float`. The compiler is worried at the possible loss of information that could be caused by assigning a value of a larger type (`double`) to a variable (`area`) of smaller type (`float`). A warning is not an error and the code will still compile. However, you should know what you are doing, so after some time, you should be able to handle these warnings as well.

The omission of the **`;`** at the end of a C statement is an example of a *syntax error*. A syntax error occurs when a statement fails to conform to the rules of the C language. The compiler doesn't generate the executable when such an error occurs. The previous error message helped us locate the offending line without any problem, but some errors could be a little difficult to locate.

However, even if you manage to successfully clear the compilation and linking phases, your program may still not behave in the manner intended. You may have entered `=` instead of `==` (both are valid C operators), or `65` instead of `65.0`. These are *semantic errors* which signify errors in meaning. For instance, **`if (x == 5)`** is not the same as **`if (x = 5)`**, even though both are syntactically correct. Semantic errors are encountered only at *runtime*, i.e., when the program is actually run. The compiler doesn't detect them; it's your job to do that—using a debugging program if necessary.

Tip: Syntax errors are normally located early, but semantic errors can often be difficult to identify. These errors may not show up in every invocation of the program simply because some data may cause problems but some may not. Pay attention to the warnings; they often provide valuable clues.

4.7 `second_prog.c`: AN INTERACTIVE AND DECISION-MAKING PROGRAM

You are aware that structured programming requires a language to support (i) decision making and (ii) repetition. Our next program (Program 4.2) uses the **`if`** statement for making decisions. It also uses the **`scanf`** function to implement interactive behavior. The program takes an integer from the user, validates the input and prints the ASCII character having that value. Four invocations of the program reveal important information that you would like to keep in mind at all times.

Note the **`#include`** statement which should always be included whenever we use functions like **`printf`** and **`scanf`**. When the program encounters **`scanf`**, it pauses to take user input. **`scanf`** then *converts* and saves this input value in the variable `your_number`.

What is striking here is the use of the `&` prefix for the variable `your_number` when used as an argument to **`scanf`**. By design, C functions can't *directly* assign keyboard input to variables. Section 4.8.2 explains the necessity of conversion and why **`scanf`** needs the `&` even though **`printf`** doesn't.

```
/* second_prog.c: Uses decision making to validate user input.
                  Introduces the scanf function for taking user input. */
#include <stdio.h>
int main(void)
{
   int your_number;

   /* Take user input first */
   printf("Enter an integer greater than 47: ");        /* No \n here */
   scanf("%d", &your_number);          /* Note that scanf requires the & */

   /* Decision making statement */
   if (your_number > 47) {             /* Beginning of control block */
      printf("The number you entered is %d\n", your_number);
      printf("ASCII value of '%c' = %d\n", your_number, your_number);
   }                                   /* End of control block */
   else
      printf("You entered a number less than 48\n");

   return 0;
}
```

PROGRAM 4.2: **second_prog.c**

```
Enter an integer greater than 47: 48                    First invocation
The number you entered is 48
ASCII value of '0' = 48

Enter an integer greater than 47: 65                    Second invocation
The number you entered is 65
ASCII value of 'A' = 65

Enter an integer greater than 47: 97                    Third invocation
The number you entered is 97
ASCII value of 'a' = 97

Enter an integer greater than 47: 47                    Fourth invocation
You entered a number less than 48
```

PROGRAM OUTPUT: **second_prog.c**

The input value is now validated with a decision making construct—the **if** statement. This statement uses a boolean expression (`your_number > 47`) that yields a true or false value. This value determines whether or not the statements in the following *control block* are executed. This block is delimited by a matched pair of curly braces ({ and }).

Like **if**, many C constructs as well as functions work with blocks. If you enter a number that is less than 48, the condition is not satisfied (false) and the statement following the **else** keyword is executed. This is evident from the fourth invocation of the program.

There are two things to note in the second **printf** statement inside the control block related to the **if** statement. **printf** prints the same variable twice but with different format specifiers (%c and %d). The %c specifier prints the character representation of the value stored in your_number, while %d prints its ASCII value. The letter 'A' has the ASCII value 65 while 'a' has the value 97. Digits have even lower values. Also note the use of single quotes inside double quotes, which lets the single quotes be interpreted literally.

Note: The ASCII value of a lowercase letter is 32 more than its uppercase counterpart. All ASCII characters are not available on your keyboard, but they can be generated by a C program.

4.8 TWO FUNCTIONS: printf AND scanf

You have been introduced to two important functions so far: **printf** and **scanf**. They belong to the I/O (Input/Output) family of functions. Even though these functions are extensively used in this book, a detailed examination of them is postponed to Chapter 9. However, we need to know how to call these functions. We also need not know how they work.

printf and **scanf** complement each other; **printf** displays output while **scanf** accepts input. Both use a *control string* to determine how variables and expressions are printed or assigned. Unlike most C functions that accept a fixed set of arguments, *the argument list for these functions is variable*. In this book, we have adopted the custom of beginning a discussion of important functions with their syntax (or usage, which we'll later call *prototype*).

4.8.1 printf: Printing to the Terminal

The **printf** function is the most convenient mechanism of displaying (printing) formatted output on the screen. It needs the file stdio.h, so you should include it every time you use this function. The function can be used in two ways:

Form 1	Form 2
`#include <stdio.h>` `printf(`*string*`);`	`#include <stdio.h>` `printf(`*control string*`,` *arg1*`,` *arg2*`, ...);`

Form 1 is used to print a simple string which may contain escape sequences (like \n). The entire string represents a single argument to **printf**:

```
printf("All characters have ASCII values\n");          A single argument
```

Form 2 is used to display the values of variables or expressions. The first argument in this case is a *control string* which contains one or more format specifiers (like %d). The remaining arguments, *arg1*, *arg2*, etc. represent the variables to be printed using the information provided in the control string. The ellipsis (three dots) at the end of the argument list signifies a variable number of items in the list.

This is how **printf** is used with two arguments to print a variable of type int (say, total which is set to 10):

```
int total = 10;
printf("Total = %d\n", total);                                    Prints Total = 10
```

A format specifier acts as a placeholder, which is replaced with the value of its matching variable in the output. Multiple format specifiers must have a matching number of arguments. The following one has three format specifiers for printing three variables:

```
float length = 10, breadth = 7, height = 5;
printf("Length = %f, Breadth = %f, Height = %f\n", length, breadth, height);
```

The format specifiers, `%d` and `%f`, are used to print a decimal integer and floating point number, respectively (Table 4.1). You will also use `%c` for printing a single character and `%s` for printing a string. There are many more specifiers but knowledge of these four will suffice for the time being.

Caution: Errors arise when the number of format specifiers don't match the number of variables. For instance, if you remove the variable `height` from the previous **`printf`** statement but not its format specifier, then you could have a junk value printed at that location. The compiler may issue a warning but it will not generate an error. C probably thinks that you have done that deliberately.

TABLE 4.1 Essential Format Specifiers Used by **`printf`** and **`scanf`**
(See Tables 9.1 and 9.3 for expanded list)

Format Specifier	*Represents*	*Used by Data Type*
`%c`	Character	`char`
`%d`	Integer	`short`, `int`, `long`, `long long`
`%f`	Floating point	`float`, `double`, `long double`
`%s`	String	Array of type `char`

4.8.2 scanf: Input from the Keyboard

The **`scanf`** function complements the role of **`printf`**. It takes input from the keyboard and assigns values *after conversion* to variables *arg1*, *arg2*, etc. using information from *control string*. The syntax of **`scanf`** is presented in the following:

```
#include <stdio.h>
scanf(control string, &arg1, &arg2, ...);
```

Like **`printf`**, **`scanf`** uses format specifiers like `%d`, `%f`, etc. in the control string, with roughly the same significance. But `&`*arg1*, `&`*arg2*, etc. don't represent variables even though *arg1* and *arg2* are variables. The argument `&`*arg1* signifies a *memory address* that contains the value of *arg1*. The following snippet of code extracted from **`second_prog.c`** shows the use of `&` in **`scanf`**:

```
int your_number = 0;
printf("Enter an integer greater than 47: ");
scanf("%d", &your_number);
printf("The number you entered is %d\n", your_number);
```

HOW IT WORKS: *How Does* `scanf` *Work?*

When you key in, say, 75, in response to the previous **`scanf`** call, **`scanf`** doesn't read the input as the integer 75 but as two separate characters `'7'` and `'5'`. **`scanf`** knows (because of `%d`) that you meant to specify an integer, so it first converts `'7'` and `'5'` to the binary equivalent of the number 75. Because of a special property of C functions, **`scanf`** has to peek into the *memory location* of a variable, say, `your_number`, to place this converted value there. The `&` in `&your_number` provides the memory address of `your_number`. **`printf`** doesn't need the `&` because it evaluates a variable and doesn't assign one.

The first **`printf`** prompts for an integer, and the one you key in is converted to an integer and saved by **`scanf`** in the variable `your_number`. The inset (*How It Works*) explains the necessity of conversion and the role of the `&` operator.

Caution: It is a common semantic error to omit the `&` in the variable list of **`scanf`**. The compiler will generate a warning but will still proceed with the creation of the executable. The program, however, won't work.

Takeaway: It is impossible to assign a variable using a function that takes a variable name as a function argument. You have to use the `&` prefix.

4.9 `third_prog.c`: AN INTERACTIVE PROGRAM FEATURING REPETITION

We have considered sequence (Program 4.1) and decision making (Program 4.2). This program (Program 4.3) features repetition, the third tenet of structured programming. Using two invocations of **`scanf`**, the program accepts a character and a number, and then runs a **`while`** statement (a loop) to print that character as many times as the number. To keep the program short and simple, validation of user input has been avoided.

This time we see three variable assignments in one line. This is legal because C requires statements to be terminated with a semicolon and doesn't care whether they are on separate lines or not. The variable `counter` is initialized (to `0`) but not the other variables which are assigned by **`scanf`**.

Program execution pauses twice—at the two **`scanf`** statements. The first **`scanf`** saves the character entered in the variable `letter`. The next **`scanf`** converts and saves the input in the variable `your_number`. Expectedly, the format specifiers (`%c` and `%d`) are different for the two calls. You need to remind yourself that variables in **`scanf`** must have the `&` prefix.

The **`while`** loop repeats the statements in its body. Like **`if`**, **`while`** uses a *control expression* to determine how often these statements are repeated (if at all). Each loop *iteration* prints one user-input character before incrementing the value of `counter`. Control then reverts to the top of the loop where the condition, i.e., control expression (`counter < your_number`) is rechecked. If the condition is met (true), the loop body is repeated, otherwise the loop is terminated. Because **`printf`** doesn't use a `\n` here, the character is printed multiple times but in one line.

```
/* third_prog.c: Uses repetition to print a character multiple times. */

#include <stdio.h>

int main(void)
{
   int your_number; int counter = 0; char character;

   /* Take user input first */
   printf("Enter the character to repeat: ");
   scanf("%c", &character);
   printf("Number of times to repeat '%c'?: ", character);
   scanf("%d", &your_number);
   printf("Printing '%c' %d times\n", character, your_number);

   /* Repetitive statement */
   while (counter < your_number) { /* Beginning of control block */
      printf("%c", character);
      counter = counter + 1;       /* Adds 1 to current value of counter */
   }                               /* End of control block */
   printf("\n");                   /* Takes cursor to next line */

   return 0;
}
```

PROGRAM 4.3: **third_prog.c**

```
Enter a character to repeat: *
Number of times to repeat '*'?: 20
Printing '*' 20 times
********************                                    20 asterisks
```

PROGRAM OUTPUT: **third_prog.c**

The statement `counter = counter + 1;` makes no sense in algebra but it is the standard method used by programming languages to increment a variable. For every iteration of the loop, 1 is added to the existing value of `counter` and the resultant sum is saved in the same variable. That is why a variable that is used for counting must be initialized to zero.

Note: Be prepared to see true and false values practically everywhere—constants, variables, expressions and functions. In C, anything that evaluates to a numeric value can be interpreted as true or false.

4.10 FUNCTIONS

C depends heavily on functions for doing all work. A *function* is a name that represents a group of statements. The inputs provided to a function are known as its *arguments*. For instance, the function **`printf("Dolly");`** has `"Dolly"` as its sole argument. If C doesn't offer a function for a specific task, you can create one, which we'll be doing in our next program.

A C function is like a black box. We need to simply know what it does and not how it does it. If a function uses arguments, we must know the number and data type of the arguments. Even though ignorance of the inner working of **printf** in no way affects our ability to use it, we need to treat user-defined functions differently. We must know how they actually work, otherwise we'll have a tough time maintaining them.

The machine codes of a group of functions are maintained in a separate file called a *library*. In C, one library is special—the *standard library*, which contains the machine code for **printf** and **scanf**. When we compile a program containing **printf**, the linker extracts its code from this library and includes it in the program executable (Fig. 4.1).

Of all C functions, the **main** function is special for two reasons. First, it must be present in every executable C program (as `int main`). Second, unlike functions like **printf** whose definition (i.e., implementation) is hidden from us, **main** is defined in every program. This definition begins with the opening curly brace ({) and ends with the closing brace (}). The body of the program (which doesn't include the preprocessor directives) is actually the body of **main** and a program executes this body when it encounters **main**.

Takeaway: We don't need to know how the standard C functions work. But we must invoke them with the correct number and type of arguments. If a function definition specifies two arguments of type `int` and `float`, then we must invoke the function precisely in that way.

Note: It is often said that preprocessor directives (like **#define**) and functions (like **printf**) are not part of the C language. This distinction apparently made sense when C was in its infancy. Today, C is a standardized language; the ANSI standard includes preprocessor directives and standard functions. It makes no sense now to consider them as being "outside" the language.

4.11 fourth_prog.c: A PROGRAM CONTAINING A USER-DEFINED FUNCTION

Now that we know something about functions, let's create a program having a user-defined function. This program (Program 4.4) uses a standard formula to compute the interest on a term deposit. The formula is used by the function **calc_interest**, which is called with two sets of the following data: principal, interest rate and term. The function is first *declared*, then *defined* and finally *called* (or i*nvoked)*. Let's examine each of these phases.

The *declaration* of the function specifies its name, the number and data type of its arguments, and the data type of the value *returned*. **calc_interest** needs three arguments of type `int`, `float` and `int`, and you have to call it exactly as specified. All functions don't necessarily return values, but **calc_interest** does; it returns a value of type `float` (the first word of the declaration). Note the **;** that terminates this declaration.

We now move on to the *definition* of **calc_interest** located below the body of **main**. The opening line of the definition doesn't have the **;** terminator, but it is otherwise a replica of the declaration. The compiler compares the definition to the declaration and outputs an error in case their

```
/* fourth_prog.c:  A simple interest calculator using a function. */

#include <stdio.h>

/* Function declaration or prototype */
float calc_interest(int principal, float interest_rate, int years);

int main(void)
{
   float amount;

   /* Use calc_interest function */
   amount = calc_interest(100, 6, 2);
   printf("Interest on 100 for 2 years = %f\n", amount);
   amount = calc_interest(200, 6, 3);
   printf("Interest on 300 for 3 years = %.2f\n", amount);

   return 0;
}

/* Function definition */
float calc_interest(int principal, float interest_rate, int years)
{
   float interest_amount;
   interest_amount = principal * (interest_rate / 100) * years;
   return interest_amount;
}
```

PROGRAM 4.4: **fourth_prog.c**

```
Interest on 100 for 2 years = 12.000000
Interest on 300 for 3 years = 36.00
```

PROGRAM OUTPUT: **fourth_prog.c**

arguments don't match. The definition of **calc_interest** specifies (i) three arguments and their data types, (ii) a set of statements for computing the interest, (iii) the mechanism of returning this value using the **return** statement.

The body of **main** calls **calc_interest** twice but with a different set of arguments. In each case, the return value is saved in the variable `amount` before it is printed. Note that **printf** is a true formatter; it can print up to two decimal places (`%.2f`). We'll later learn to print the return value of a function without saving it to a variable.

By using a function to compute the interest, we have benefited in two ways. First, we have made the code *reusable*. We can call this function multiple times in the same program or in multiple programs without replicating the code. Second, code maintenance is also easier because if the formula changes, the modification has to be carried out at a *single* location.

Note: The C language was developed on the premise that lower-level functions would do all the work and the **main** function (or other high-level functions) would invoke them. If a function is designed as an independent unit, it can be stored in a separate library for use by multiple applications. In that case, the linker must know the name and location of this library.

4.12 FEATURES OF C

Even though we have seen only a handful of C constructs in the four sample programs of this chapter, we have been exposed to most of the key features of the language. C shares many features with other programming languages, but it has some special characteristics of its own. The language was developed by Ritchie for his own use, so some surprises are inevitable:

- *Compact and efficient* C has a small vocabulary comprising 30 keywords, but much of its power is derived from its exploitation of all symbols found on your keyboard. C executables are small in size, which becomes an advantage when developing applications for hand-held devices having limited resources.
- *A portable language* C runs on a variety of devices ranging from mobile phones to supercomputers. Since C makes no assumptions about the hardware, an ANSI-compliant C program can be ported to other platforms with minor or no modifications.
- *A free-form language* A directive or statement can begin from any position in a line. Previously, C required the **#define** statement to begin from column 1, but that restriction has been removed by ANSI C.
- *A strongly typed language* C supports a large number of data types that allow a programmer to choose one that doesn't waste memory. There are as many as five types for integers and three for floating point numbers. However, C doesn't support string or boolean data types. Strings are implemented in C by using an array of characters having the data type `char`.
- *A case-sensitive language* C keywords and identifier names (like variables, for instance) are case-sensitive. All keywords have been designed in lowercase, so you can use only **while** and not **WHILE** as a loop statement. Variables can be designed in either case as long as you remember that `pie` and `Pie` are two separate variables.
- *Statements are terminated with a semicolon.* The concept of a *line* has little significance in C. A line can comprise multiple statements as long as they are separated by semicolons. This stipulation also applies to constructs that use curly braces to define a body. For instance, the following **while** loop that would normally occur in four lines can also use a single line:

```
while (count < number) { printf("%c", '*'); count = count + 1; }
```

  Compare this with a similar statement used in the previous program, **third_prog.c**, and you'll have no doubt that the above form is not pleasant to look at.
- *C is extensible because of function libraries.* The standard library contains a host of useful functions for handling a variety of programming tasks. A programmer needs to use this library for routine tasks like string manipulation and for doing complex arithmetic. You too can create a library of functions, distribute it and thereby extend the language.

- *C permits creation of cryptic code.* By virtue of its vast collection of operators, C programs can be quite cryptic. C programmers sometimes use this opportunity to outsmart their peers only to discover later that their "smart" code is barely comprehensible.
- *C trusts the programmer.* The language assumes that you know what you are doing. C assumes that you know the difference between `9` and `9.0` because they can produce different results. Also, if you have defined an array of 5 elements, C allows you to access the non-existent "sixth" element and beyond. C assumes that you have a specific reason for accessing the "sixth" element, which may not be true.

These features will gradually unfold themselves as we examine the remaining chapters of this book. C is a lean but powerful language, but it also assumes that the programmer is intelligent and vigilant. It's very easy to commit semantic errors in C because the compiler just won't notice them. Modern compilers, however, are smart enough to point out that a variable has been declared but not used or that **`printf`** doesn't have a matching number of arguments. Even today, these lapses are often displayed as warnings and not errors.

4.13 SOME PROGRAMMING TIPS

If you look at the programs featured in this book, you'll notice that they have a common cosmetic style. Cosmetics is important in programming because proper use of it makes a program both readable and maintainable. In this book, we have adopted a set of styles that are presented below as a general recommendation:

- *Choose filenames with care.* It is common for starters to choose filenames like **`a.c`**, **`b.c`**, and so on, but it pays to use meaningful names. For instance, the filename **`decimal2binary.c`** should leave no doubt in your mind about the objective of the program.
- *Use a prefix for a filename belonging to an application.* Adoption of a suitable prefix helps identify, list and back up files with minimum effort. An application for a tourist agency, for instance, could have filenames like **`tourism_bill.c`**, **`tourism_quotes.c`**, etc.
- *Comment your programs.* Use the `/* text */` feature to document the objectives at the beginning of a program or section. The usage and purpose of a user-defined function should be noted in its declaration or definition.
- *Choose meaningful names for variables.* Names like `tax_rate`, `month`, `years_left` help in understanding program logic. Use of single-letter names like `x` or `y` should be avoided in most cases. Beginning programmers often tend to adopt the latter approach to save keystrokes but end up spending more time on their programs than they ought to.
- *Use whitespace liberally.* The term *whitespace* collectively refers to the space, tab and newline characters. If C allows the use of one whitespace character at any place, you can use several of them without violating any syntax. The statement `counter = counter + 1;` is easier to read than `counter=counter+1;` although both are syntactically correct. Your programs are going to get longer, so use blank lines to divide the code into separate sections.

- *Maintain a consistent style for curly braces.* You have used a set of curly braces to define a control block for the **main** function, the **if** and **while** statements. There are two accepted styles for them:

  In the first form, the opening brace is placed in the next line and the closing brace is vertically aligned with it:

  ```
  int main(void)
  {
     body
  }
  ```

  The second form places the opening brace on the right of the construct. The closing brace lines up with the identifier that uses these braces:

  ```
  while (condition is true) {
     body
  }
  ```

  This book uses the first form for all functions (including **main**) and the second form for everything else (loops, decision-making, structures, etc.). Choose the style or mix that most suits you and follow it consistently. Programs containing nested loops (one loop within another) can be very hard to understand if their braces are not lined up properly.

- *Indent program statements.* Program statements need to be indented properly. You'll see two levels of indentation in the programs, **second_prog.c** and **third_prog.c**. While the body of **main** is indented one tab to the right, the statements in the control blocks of the **if** and **while** statements are preceded by two tabs.

You may find these suggestions trivial, but experience will tell you otherwise. You are better off by not waiting for that to happen. Adopt these suggestions from day one because they provide clarity to programs and thereby enhance the productivity of a programmer.

Tip: The tab, an ASCII character that you'll find on the left of your keyboard (having the symbol ⇆), expands to eight spaces by default. For complex programs that feature multiple levels of indentation, deeply nested statements could easily overflow to the next line. To maintain readability, configure your Tab key (if possible) to expand to three or four spaces.

4.14 C89 AND C99: THE C STANDARDS

The original definition of C appeared in the first edition of the text book *The C Programming Language* written by Kernighan and Ritchie in the 1970s. The version of C that conformed to the book was known as *K&R C*. As UNIX systems spread, so did the use of C, and as vendors developed C compilers, they included features not found in K&R C. C thus got progressively fragmented which led to the situation where a C program developed on one machine wouldn't run on another.

In 1989, the American National Standards Institute (ANSI) formally adopted a C standard called C89. This standard was adopted by the International Standards Organization (ISO) in 1990. C89 introduced a number of features not found in K&R C. These include a different way of declaring functions, the induction of additional keywords, and specific guidelines for setting the minimum size of data types (like `int`, `char`, etc.).

As hardware advanced from 8-bit computers (the original IBM PC) to 32- and 64-bit machines, the language had been continuously undergoing changes by the incorporation of new features. As business and academic institutions adopted these features, the need for a new standard was increasingly felt. In 1999, ANSI adopted the C99 standard, which was approved by ISO in 2000.

Many features found in the C99 standard were already offered by pre-C99 compilers. In fact, commenting using //, the `long long` data type and function *inlining* have been available in GCC long before C99 came into being. The C99 committee wisely decided to take into account these enhancements. This made the transition to C99 a lot easier for those organizations who have already been using many of its features. C11 (published in 2011) is the current C standard, which we'll occasionally refer to in this book.

This book is based on C89 because that is the version used by most C programmers today. However, the C99 features are also discussed in separate asides. Before writing programs, you will have to decide whether you want C89- or C99- compliance because only then can you make your programs portable. C was developed as a portable language and we'd better let it remain that way.

WHAT NEXT?

Before we examine the statements supported by C, we need to thoroughly understand the properties of data that are used with these statements. The next chapter pursues this topic and paves the way for further examination of data in expressions.

WHAT DID YOU LEARN?

A C program is first passed through a *preprocessor*, which processes lines beginning with a #. A *compiler* converts the preprocessed code to machine code but which is not ready to run. The *linker* adds the code of functions to the compiled output to create a standalone executable.

Any program logic can be implemented by using statements that feature *sequence*, *decision making* (like the **`if`** statement) and *repetition* (like the **`while`** statement).

The compiler detects *syntax errors* in a program. The executable is created only when the program is free from such errors. *Semantic errors* are detected only at runtime.

Functions are used to codify a set of statements that are used repeatedly. A function *declaration* specifies the number and type of arguments and the *definition* contains the implementation. It is necessary to only know *how* to call a standard function.

printf and **scanf** are two important functions of the standard library. Both convert their input data, but **printf** displays output while **scanf** saves converted data in variables.

C is a compact and portable language. It allows you to extend the language by adding libraries. Many errors get unnoticed by the compiler because C expects the programmer to be vigilant.

C89 and C99 are the two major standards in use today. Any project must comply with one of them.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

4.1 C is a high-level language with some low-level features.

4.2 A C program can obtain program-related information from another file.

4.3 Both statements and preprocessor directives end with a semicolon.

4.4 A symbolic constant like `WIDTH` is evaluated by the compiler.

4.5 The compiler doesn't invoke the linker if it detects an error in the source code.

4.6 Syntax errors are detected by both the compiler and linker.

4.7 Every executable C program must contain the **main** keyword.

4.8 The compiler generates an executable file only if there are no syntax errors.

4.9 Too many comments in a program should be avoided because they increase the size of the executable.

4.10 The standard library contains the object code of the built-in functions of ANSI C.

4.11 The **printf** function must have at least one format specifier like `%d`.

4.12 The expression `&x` used in **scanf** represents the memory address of the variable `x`.

4.13 Every function returns a value.

A2. FILL IN THE BLANKS

4.1 C was originally developed for the _____ operating system.

4.2 A C program is acted upon by a preprocessor, _____ and _____ to create an executable.

4.3 The string **#include <stdio.h>** is a preprocessor _____.

4.4 In the sequence **#define WIDTH 15**, `WIDTH` is a _____.

4.5 The symbols _____ and _____ are used to enclose comments in a C program.

4.6 _____ errors are detected on compilation but _____ errors are generated at runtime.

4.7 The body of a C program begins with the keyword _____ and ends with the _____ statement.

4.8 The * and = represent two _____ supported by C.

4.9 The sequence `x > 5` represents a _____ expression.

4.10 A Windows C executable file has the ____ extension.

4.11 Program execution is terminated when the _____ statement is encountered.

4.12 A program pauses to take input when it encounters the _____ function.

4.13 The sequence `"%f %d"` is known as a _____.

A3. MULTIPLE-CHOICE QUESTIONS

4.1 The preprocessor (A) acts on lines beginning with #, (B) modifies a copy of the source code, (C) doesn't process C statements, (D) all of these.

4.2 A string followed by `()` represents (A) a function, (B) library, (C) keyword, (D) none of these.

4.3 When the compiler encounters the **`printf`** function, it (A) creates the object code for **`printf`**, (B) extracts the object code from a library, (C) leaves the job to the linker, (D) none of these.

4.4 The line **`#define WIDTH 15`** (A) defines a variable named `WIDTH`, (B) represents a preprocessor directive, (C) replaces `WIDTH` with 15 everywhere in the program, (D) A and B, (E) B and C.

4.5 If the compilation process generates only warnings, (A) the executable will not be created, (B) the executable will be created but it will not run, (C) the program may still run without any problems, (D) none of these.

4.6 A control block is enclosed within (A) `(` and `)`, (B) `{` and `}`, (C) `[` and `]`, (D) none of these.

4.7 Before a function is called, it must be (A) declared, (B) defined, (C) both.

4.8 A library contains the (A) source code of programs, (B) source code of functions, (C) object code of programs, (D) object code of functions, (E) none of these.

CONCEPT-BASED QUESTIONS

4.1 Explain the role of the linker in program compilation.

4.2 Explain how functions make code modular, reusable and maintainable.

4.3 Explain how the **`main`** function differs from other functions like **`printf`** and **`scanf`**.

4.4 Discuss the similarities and differences between the **`printf`** and **`scanf`** functions.

4.5 Name five important features of C.

PROGRAMMING & DEBUGGING SKILLS

4.1 Correct the syntax errors in the following program:

```
#define LENGTH = 10;
#include <stdio.h>;
int main
{
```

```
    int width = 5;
    int area = LENGTH X width;
    printf(Area = %d/n, area);
    return 0;
}
```

4.2 Locate the syntax errors in the following code segment:

```
int X, int Y;
scanf('%d', X);
if x < 0
   printf('Only positive integers\n');
else {
   Y = X * X;
   printf('Value of Y = ', Y);
}
```

4.3 Write a program that accepts two integers from the keyboard and prints their average. Replace the sequence **`int main(void)`** with **`main()`** and drop the **`return`** statement. Does the program still compile and execute properly?

4.4 The following code section compiles correctly but doesn't work properly. What is wrong with it?

```
int x;
while (x < 10)
   printf("This statement is executed ten times\n");
printf("Quitting ...\n");
```

4.5 Write a program that uses a **`while`** loop to print the integers from 1 to 10. If you replace the < operator with <=, what else do you need to do to obtain the same output?

4.6 What is the following program meant to display? Now try out the following:

(i) Remove the function declaration. Does the program compile?

(ii) Change the function arguments from `int` to `float`. How does the output change and why?

```
#include <stdio.h>
float func(int x, int y);           /* Function declaration */
int main(void)
{
   float result;
   result = func(5, 8);             /* Function return value saved here ... */
   printf("%f\n", result);
   printf("%f\n", func(5.5, 8.5)); /* ... but not here */
   return 0;
}
float func(int x, int y)            /* Function definition */
{
   float z;
   z = (x + y) / 2.0;               /* The / is operator for division */
   return z;
}
```

5 Data—Variables and Constants

WHAT TO LEARN

- Concept of data held in *variables* and *constants*.
- Classification of data into *fundamental* and *derived* types.
- Attributes of the *integer* and *floating point* data types.
- Why the `char` data type is special.
- How attributes of constants differ from variables.
- Overview of *arrays* and *strings* as derived data types.

5.1 PROGRAM DATA

We deal with data in everyday life. Data often represent the attributes of real-life objects—like name, weight, quantity or price. Because computer programs are often closely linked with real-life situations, they need data too. In fact, it is impossible to devise a meaningful program that doesn't need data. Even though instructions make a program do all the work, it's the data that make the same program behave in different ways.

A program must know the address in memory of every piece of data that it handles. When a program encounters the variable `x`, it can evaluate this variable only if it knows the memory location of `x`. As a programmer, you need not know the address of this location or the register where it is moved to because the compiler provides this information in the executable. Keep in mind that you are learning to use C and not assembly language.

5.1.1 Variables and Constants

Program data occur in the form of *variables* and *constants*. A variable has a name that is assigned a value. This value is stored in memory and can change while the program is running. We used the variables `area`, `your_number` and `counter` in the programs of Chapter 4. The name used to access a variable is also known as an *identifier*. (Arrays and functions also have names, hence they too are identifiers.)

Constants represent data that don't change. In the expression `(fahrenheit - 32) / 9.0`, both `32` and `9.0` are constants. The first argument used with **`printf`** is often a string constant, say, `"Printing ASCII list"`. Like variables, constants also occupy space in memory. These constants have no names, but later in the chapter we'll encounter constants that have names. We'll also have a look at variables that behave like constants.

5.1.2 Data Types

Because both numeric and string data are stored as binary numbers, how does a program distinguish between them? C is a *typed* language, which means that every data item—variable or constant—has a type associated with it. To consider variables first, its type is clearly visible in the declaration:

```
int counter;
```

The compiler identifies the type as `int`, i.e. integer, and allocates *at least* 2 bytes of memory for the variable named `counter`. `int` is one of several data types in C. The highlight on "at least" signifies the inability of C to specify exact sizes for variables and constants (and we'll soon know why). All constants (like `100`, `3.142`, `'A'`, etc.) also have data types and we must know them.

5.1.3 Data Sources

A program obtains its data from different sources, and the C language has different constructs for handling these sources. We have already dealt with the first two of the following three sources in Chapter 4:

- Embedded in the program (as variables and constants).
- User input captured by functions like **`scanf`** (a feature of interactive programs).
- External files that are read by file-related functions supported by C.

There is at least one more source of data (command-line arguments), but we'll postpone discussions on it until we have understood arrays and strings. Files are also taken up later, so we will stick to the first two data sources for the time being.

5.2 VARIABLES

A *variable* is a name given to a memory location that is accessed by that name to fetch or set a value at that location. Variable usage is either mandatory or compelling. For instance, if a **`while`** loop has to execute a finite number of times, you need a variable to determine when the loop will terminate (mandatory). On the other hand, if the rupee-dollar conversion rate is used multiple times in a program, then this value is better stored in a variable (compelling). Here, the variable behaves like a constant while the program is running.

5.2.1 Naming Variables

The rules for framing variable names apply to all identifiers, which include arrays and functions. A variable name can begin with a letter or the underscore (_) character, but the remaining part of the name can comprise only the following characters:

- Letters (both lower- and uppercase)
- Digits
- The underscore (_) character. Note that this is different from the hyphen (-) which is available on the same key as the _ on your PC or laptop.

A variable name thus cannot begin with a digit. The names `rate1` and `_rate1` are valid but not `1rate`. Variable names are also case-sensitive; `rate1` and `RATE1` are two separate variables. Here are some examples of valid identifier names:

```
x   ABC     name024    sampling_speed     _month    __file
```

Under C89 (the first C standard from ANSI), the first 31 characters of an identifier are significant. The remaining characters are ignored by the compiler. The following keywords used in C cannot be used as variable names:

```
auto      continue  enum      if        short     struct    unsigned
break     default   extern    int       signed    switch    void
case      do        float     long      sizeof    typedef   volatile
char      double    for       register  static    union     while
const     else      goto      return
```

The C language also uses a number of variables but they are in uppercase. To distinguish user-defined variables from the ones used by C, you should declare variables only in lowercase.

Tip: Use meaningful lowercase names for variables. Names like `x` and `abc` are valid, but they don't intuitively reveal what they represent—unlike variables like `sampling_speed` and `conversion_rate`.

5.2.2 Declaring Variables

A variable must be *declared* before use. In most cases, the declaration is made at the beginning of the program (mostly after **`main`**, but sometimes before **`main`** for global variables). The following statement declares a variable named `bits_per_sec`:

```
int bits_per_sec;                          ; shows declaration is a statement
```

Here, the variable `bits_per_sec` has the type `int`, one of the integer types supported by C. On seeing this type, the compiler allocates *at least* two bytes in memory for `bits_per_sec`. It also notes the starting address of that memory location so it knows where to look up when encountering expressions that use the variable. We have still not explained the significance of the highlight on the words "at least."

Using the comma as delimiter, multiple declarations can be made in a single statement provided the type is the same:

```
int bits_per_sec, sampling_speed;          Variables must have same type
```

On modern machines, an `int` variable uses 4 bytes of memory, which is also the size of the word. However, when the CPU extracts these 32 bits from memory it must also know that these bits have

to be interpreted as a *single* binary number. This knowledge is vital because a chunk of 4 bytes can be interpreted as an integer, a floating point number or a three-character (not four) string.

5.2.3 Assigning Variables

The previous declarations did not assign initial values to the variables. To *initialize* a variable, use the = operator. One way is to separate the declaration from the assignment:

```
int bits_per_sec;                    First declare
bits_per_sec = 0;                    Then assign
```

Alternatively, you can combine both activities:

```
int bits_per_sec = 0;                Both in one statement
```

For declaring and initializing multiple variables *of the same type*, use the comma as delimiter. The following statements declare and initialize all variables to the same value:

```
float amount1, amount2, amount3;
amount1 = amount2 = amount3 = 1000.00;
```

However, to assign separate values, use the comma in the initialization as well:

```
amount1 = 500.50, amount2 = 600.75, amount3 = 200.25;
```

You can even combine the declaration and initialization:

```
float amount1 = 500.50, amount2 = 600.75, amount3 = 200.25;
```

The value of an uninitialized integer variable is indeterminate (i.e., junk); it can't be assumed to be zero. The value is actually one that was resident in memory when the program was executed. *Never use an uninitialized variable in an expression!*

Tip: Even though a variable declaration can occur anywhere in a program as long as it precedes usage, it is good programming practice to bunch all declarations immediately *after* **`main`**. However, global variables (differerent from the ones discussed here) have to be declared *before* **`main`** *(14.8)*.

5.3 `intro2variables.c`: DECLARING, ASSIGNING AND PRINTING VARIABLES

Program 5.1 declares four variables, assigns them values and then prints them. Two variables are of type `int` and two have the type `float`. The program also shows how a variable is assigned from keyboard input.

Out of the four variables, only one is uninitialized (`float_x1`). No wonder this variable displays junk the first time it is printed (`-0.000029`). `float_x1` is later assigned with (i) =, (ii) **`scanf`**. This function halts execution to let you key in a floating point (real) number. In both cases, the value assigned or keyed in is not *exactly* the one you see in the **`printf`** output. Floating point numbers can't always be exactly represented in memory.

```
/* intro2variables.c: Declares, assigns and prints variables.
                      Also assigns a variable by keyboard input. */
#include <stdio.h>

int main(void)
{
   /* Variables must be declared before they are used */
   int int_x1, int_x2, int_x3 = 64;       /* Partial initialization */
   int_x1 = int_x2 = -22;                 /* Simultaneous initialization */
   float float_x1;                        /* Uninitialized variable */
   float float_x2 = 30.48;

   printf("int_x1 = %d, int_x2 = %d, int_x3 = %d\n",
           int_x1, int_x2, int_x3);
   printf("float_x1 = %f, float_x2 = %f\n", float_x1, float_x2);

   /* Once declared, a variable can be assigned anywhere .... */
   float_x1 =  1203.67;
   printf("float_x1 = %f\n", float_x1);

   /* ... and also reassigned, this time with the scanf function */
   printf("Now key in any real number: ");
   scanf("%f", &float_x1);
   printf("float_x1 = %f\n", float_x1);

   return 0;
}
```

PROGRAM 5.1: **intro2variables.c**

```
int_x1 = -22, int_x2 = -22, int_x3 = 64
float_x1 = -0.000030, float_x2 = 30.480000
float_x1 = 1203.670044
Now key in any real number: 1234.56789
float_x1 = 1234.567871
```

PROGRAM OUTPUT: **intro2variables.c**

Caution: If you don't initialize a numeric variable in your program, it will have a junk value to begin with. Never assume this value to be zero. Not all variable types have junk initial values though.

5.4 DATA CLASSIFICATION

C is a small language but it is generous in its offering of data types from where you should be able to select the right type and subtype. These types can be broadly categorized as follows:

- Fundamental (like `int`, `float`, etc.)
- Derived (like arrays and pointers)
- User-defined (structures, the `typedef` and enumerated types)

The derived types are variations of the fundamental types. C also allows you to define your own data types for handling complex data. However, the language doesn't have a separate type for strings. In this chapter, we'll discuss the fundamental types and briefly understand how C implements a string.

5.4.1 The Fundamental Types

In our previous programs, we have used only the fundamental data types—explicitly for variables and implicitly for constants. (A constant like `32` or `'A'` also has a fundamental data type.) The fundamental types can be further divided into the following three subtypes:

- Integer (the keywords `short`, `int`, `long` and `long long`)
- Floating point (the keywords `float`, `double` and `long double`)
- Character (the keyword `char`)

The declaration of any variable of the fundamental types must begin with one of these keywords. The subtype determines two things: the amount of memory allocated for the variable or constant and the scheme used for storage. Unlike integer data, a real number like `32.5` is *not* stored as a single contiguous number but as a combination of two integers *(5.8)*.

Constants also have types, but unlike variables, their type information is conveyed *implicitly* to the compiler. When the compiler looks at constants like `3`, `3.0` or `"Jelly Bean`", it interprets `3` as an `int`, `3.0` as a `double`, and `"Jelly Bean"` as an array of type `char`. The upcoming discussions on the three subtypes apply fully to variables and partially to constants.

5.4.2 Type Sizes: The `sizeof` Operator

The compiler knows the size of every subtype even though K&R C (the first definition of the language) deliberately refrained from specifying *fixed* sizes for them. To enable C to run on all types of hardware, K&R C prescribed only *minimum* sizes along with a relationship between them. This explains the use of the words "at least" in Sections 5.1.2 and 5.2.2.

HOW IT WORKS: Declaration vs Definition

We have *declared* variables before using them, but it can also be said that we have *defined* them. A *declaration* of an object makes its type information available to the compiler but it doesn't create memory for the object. The compiler simply knows how to store and interpret the object in memory. It's the *definition* that actually allocates space in memory for the object.

We have not bothered so far to make this distinction because the statement `int x;` not only makes type information of `x` available to the compiler (declaration) but also creates space in memory for `x` (definition). When you declare a variable, you also define it.

However, declaration and definition represent separate activities for a function or a structure. You have seen in Section 4.11 how a function is first declared before **`main`** and defined after **`main`**.

ANSI's endorsement of the K&R C prescription (the first C definition from Kernighan and Ritchie) of minimum sizes needs to be kept in mind when developing portable applications. Even though ANSI specifies the minimum size of `int` as 2 bytes, your machine will in all probability have a 4-byte `int`. And if portability is not a concern, there is no reason why you shouldn't take advantage of this enlarged size.

The **`sizeof`** operator evaluates the size of all data types—the fundamental types, pointers, arrays and constants. This is how you can find the size of the `int` data type on your computer:

```
printf("Size of int: %d\n", sizeof(int));
```

This should print 4 on your PC. Note that **`sizeof`** is not a function even though it looks like one. It is an operator (like = and +) which works on an *operand*, which here is enclosed within parentheses in function-like manner. The next program uses **`sizeof`** both with and without parentheses.

Takeaway: There are no fixed sizes for the fundamental data types, but only ANSI-specified minimum sizes. This was done to protect the applications that were developed prior to the ANSI standard.

5.5 `sizeof.c`: DETERMINING THE SIZE OF THE FUNDAMENTAL DATA TYPES

Program 5.2 uses **`sizeof`** to determine the size of all the fundamental data types on your computer. It operates on two constants as well. The last **`printf`** statement which checks for the `long long` type has been commented. To know whether your compiler supports this data type inducted by C99 (the next ANSI standard after C89), uncomment this line. The output is system-dependent, the reason why it is shown separately for Linux (GCC) and Windows (Visual Studio).

The size of `char` is always 1 byte because, by definition, it was designed to hold the entire character set of the machine. Observe that both `int` and `long` are 4 bytes wide on this machine. We have also used **`sizeof`** on the constants `34` and `3.142`. Do an integer and floating point constant need 4 and 8 bytes, respectively? We'll soon find out.

If **`sizeof`** is an operator and not a function, does it need the parentheses? That depends on what the operand is. If the operand is a fundamental type (like `int` and `float`), parentheses are necessary. Otherwise, they are optional as shown in the two statements featuring the constants `34` and `3.142`.

Note: Unlike other operators that evaluate their operands during runtime, **`sizeof`** is a compile-time operator. The result of its operation is known to the compiler at the time of compilation, which implies that the value is built into the executable.

5.6 THE INTEGER TYPES

An *integer* is a contiguous collection of digits without a decimal point but it may contain the `-` prefix. For instance, 65536 and -7 are integers. C uses the `int` keyword to declare an integer variable. Using the `short` and `long` qualifiers in addition, C offers a total of four types of integers, whose abbreviated and expanded keywords are shown in the following list:

- `int`
- `short` or `short int` (`short` is a qualifier to `int`)
- `long` or `long int` (`long` is a qualifier to `int`)
- `long long` or `long long int` (`long long` is a qualifier to `int`)

```
/* sizeof.c: Displays the size of the fundamental data types
             using the sizeof operator. */
#include <stdio.h>

int main(void)
{
   printf("Size of char: %d byte\n", sizeof(char));
   printf("Size of short: %d bytes\n", sizeof(short));
   printf("Size of int: %d bytes\n", sizeof(int));
   printf("Size of long: %d bytes\n", sizeof(long));
   printf("Size of float: %d bytes\n", sizeof(float));
   printf("Size of double: %d bytes\n", sizeof(double));
   printf("Size of long double: %d bytes\n", sizeof(long double));

   /* Now use sizeof on 2 constants -- parentheses optional here */
   printf("Size of 34: %d bytes\n", sizeof 34);
   printf("Size of 3.142: %d bytes\n", sizeof(3.142));

   /* Uncomment the following line to see if long long is supported
      by your compiler. It may lead to a compiler error. */

   /* printf("Size of long long: %d bytes\n", sizeof(long long)); */

   return 0;
}
```

PROGRAM 5.2: **`sizeof.c`**

Linux (GCC)	*Windows (Visual Studio)*

`Size of char: 1 byte`
`Size of short: 2 bytes`
`Size of int: 4 bytes`
`Size of long: 4 bytes`
`Size of float: 4 bytes`
`Size of double: 8 bytes`
`Size of long double: 12 bytes`
`Size of 34: 4 bytes`
`Size of 3.142: 8 bytes`
`Size of long long: 8 bytes`
	`Size of char: 1 byte`
	`Size of short: 2 bytes`
	`Size of int: 4 bytes`
	`Size of long: 4 bytes`
	`Size of float: 4 bytes`
	`Size of double: 8 bytes`
	`Size of long double: 8 bytes`
	`Size of 34: 4 bytes`
	`Size of 3.142: 8 bytes`
	long long not supported before Visual Studio 2013

PROGRAM OUTPUT: **`sizeof.c`** (Linux and Windows)

This is how we declare two variables of type `int` and `long`:

```
int simply_int;
long simply_long = 50000;
```

The last type in the list (`long long`) has been available with many compilers before it was included in C99. However, Visual Studio supports it only from their 2013 version. In the following sections, we'll examine these types, know their minimum sizes along with the relationship between these sizes.

Note: The - in -7 is an operator; it multiplies 7 by -1.

5.6.1 The ANSI Stipulation for Integers

The ANSI standard came into being after C applications had already penetrated the commercial world. In its endeavor to create a standard that also protected these applications, ANSI *consciously* left the sizes of these data types vague. However, it specified two rules:

- A minimum size for the types:

  `short` — 2 bytes (16 bits)
  `int` — 2 bytes (16 bits)
  `long` — 4 bytes (32 bits)
  `long long` — 8 bytes (64 bits)

  Modern `int` *is 4 bytes*

- A relationship between the types that can be represented in this form:

  `char <= short <= int <= long <= long long`

The first rule explains why we used the words "at least" when discussing memory allocation for variables. The second rule implies that a `short` cannot be smaller than `char`, an `int` cannot be smaller than `short`, and so on. These two rules make it possible for `int` and `long` to have the same size on one machine, while `short` and `int` may have the same size on another. We included the `char` data type here because `char` is essentially an integer type even though it is almost exclusively used to represent characters.

Takeaway: You can't be assured of seeing a fixed size for any type, but ANSI guarantees a minimum size. It also specifies a relationship between the sizes.

5.6.2 Signed and Unsigned Integers

By default, the four `int` types are *signed*, which means that they can handle both positive and negative integers. This default behavior can be overridden by using the `unsigned` qualifier before the keyword. For instance, the statement

```
unsigned int count;
```

allows `count` to store only positive integers. Because signed data types use one bit to store the sign, the positive side of the range of a signed type is half of its unsigned type *(3.3)*. For instance, the two

bytes used by unsigned short can represent values between 0 and 2^{16} - 1, i.e. 65,535. This range changes to -32,768 to 32,767 (-2^{15} to 2^{15} - 1) for signed short.

Table 5.1 displays the essential attributes of these integers, their unsigned maximum values and the format specifiers used by **printf** and **scanf** for handling them. Note that the unsigned format specifier has the d replaced with u. Thus, %ld becomes %lu, %d becomes %u, and so forth.

TABLE 5.1 Attributes of Integers

Subtypes	*ANSI-mandated Minimum Size*	*Unsigned Maximum Value*	**printf/scanf** *Format Specifier (signed/unsigned)*
short	2	65,535	%hd / %hu
int	2	65,535	%d / %u
long	4	4,294,967,295 (4 billion)	%ld / %lu
long long	8	18,446,744,073,709,551,615	%lld / %llu

5.6.3 The Specific Integer Types

The short Type Abbreviated from short int, the minimum and typical size for this type is 2 bytes. For the range it offers (-32,768 to 32,767), short is suitable for positive integral entities like age, marks and family size. If you need an integer greater than 32,767 but less than 65,535, use unsigned short.

The int Type This is the standard integer type that reflects the word size of your machine. Like short, its minimum size is 2 bytes, but since most computers today are 32-bit machines, int is also 32 bits (4 bytes). A signed int can handle numbers up to 2 billion (on both +ve and -ve sides), while the unsigned type can hold a number up to 4 billion. The popular and often misused %d format specifier actually applies to int.

The long Type The long type (abbreviated from long int) is meant to hold numbers larger than int. However, that has not always been the case on 32-bit systems. In most cases, int and long are the same size—4 bytes, but on some systems you could see a long of 8 bytes. Also, most 64-bit programming models use 8-byte long integers. An 8-byte long integer can handle values in the quintillion range.

The long long Type This type was added by C99 to the language to accommodate 64-bit integers. On most systems that support this type, long long is 8 bytes. Since long in most 64-bit models is already set to 8 bytes, it's a little too early to say what the eventual size of long long would be.

Note: The basic format specifier for the signed integer data type is %d. A *modifier* is needed for short (h), long (l) and long long (ll). For unsigned types, replace %d with %u with the same modifiers as considered necessary. The complete list of these specifiers used by **printf** and **scanf** can be found in Chapter 9.

5.6.4 Octal and Hexadecimal Integers

Apart from decimal integers (base-10), C also supports octal (base-8) and hexadecimal (base-16) numbers. C identifies them by a special prefix:

- When a `0` (zero) prefixes a number comprising the digits 0 to 7, C interprets the number as octal. Thus, `072` is an octal integer having the value (58)10 ($7 \times 8^1 + 2 \times 8^0 = 58$). The **`printf`**/**`scanf`** duo uses `%o` to handle octal integers.
- When a number comprising the digits 0 to 9 and A to F is preceded by `0x` or `0X`, C interprets the number as hexadecimal. Thus, C understands `0x72` as a base-16 number having the value (114)10 ($7 \times 16^1 + 2 \times 16^0$). Use the `%x` or `%X` format specifier to handle hex integers.

Systems programmers need to be familiar with these numbering systems because many programs output numbers in these forms.

5.7 `all_about_int.c`: UNDERSTANDING INTEGERS

Program 5.3 partially summarizes our knowledge of the four integer data types. It demonstrates how **`printf`** converts data across the different numbering systems. The program also shows the consequences of data overflow and how use of wrong format specifiers affect the conversion.

Two of the variables are signed and one is unsigned. Since `short_x` is signed, it uses 15 bits, so 32767 is the maximum value it can hold. This is confirmed by the octal representation which shows the largest octal digit (7) occupying all five slots. The right format specifiers have been used in the first three **`printf`** statements.

Next, we create an overflow situation by adding 1 to `short_x`. Signed `short` has a range of -32,768 to 32,767. The number 32768 resulting from the addition exceeds the maximum positive value by 1. This causes an overflow which sets the sign bit to 1 and the remaining bits to 0. The resultant number (1000 0000 0000 0000)2 represents the minimum value for a signed `short` (-32768).

HOW IT WORKS: Which Integer Type to Choose?

The ANSI stipulation leads to overlap of the type sizes. Some machines have the same size for `short` and `int`, or `int` and `long`. How does one then determine the right type to use for an integer? If your application needs to be portable, the following guideline should help:

If you need 2 bytes and both `short` *and* `int` *have this size on your machine, then use* `short`*. Your program won't waste memory when it is moved to a 32-bit machine where* `int` *expands to 4 bytes. However, if you need 4 bytes, and both* `int` *and* `long` *have this size, then use* `long` *because* `int` *can reduce to 2 bytes when the application moves to a smaller machine.*

The above rule of thumb will protect your application but it is not guaranteed to prevent wastage. If you use `long` because of the guarantee of getting 4 bytes, then be prepared to get 8 bytes instead on 64-bit machines.

The last two **printf** statements use wrong format specifiers. We first printed a negative integer with %u, the specifier meant for unsigned int. We then printed the new value of int_x (123456) with %hd, the specifier meant for short. This number also exceeds the maximum value that can be handled by short. Both statements output junk values.

Note: The last two **printf** statements use symbols like %%d and %%u to print the strings %d and %u, respectively. Because % is interpreted in a special manner (the specifier prefix) by **printf**, the only way to literally print a % is to prefix it with another.

```
/* all_about_int.c: Assigns and displays integer values.
                    Shows effects of overflow and wrong format specifier. */
#include <stdio.h>

int main(void)
{
   short short_x = 32767;              /* Signed -- uses 15 bits */
   int int_x = -65;                    /* Signed -- uses 31 bits */
   unsigned long long_x = 400000;      /* Unsigned -- uses 32 bits */

   printf("short_x: Decimal: %hd, Octal: %ho, Hex: %hx\n",
           short_x, short_x, short_x);
   printf("int_x: %d\n", int_x);
   printf("long_x = %lu\n", long_x);

   /* Data too large for type -- causes overflow */
   short_x = short_x + 1;              /* 16-bit value can't be held in 15 bits */
   printf("short_x after incrementing: %hd\n", short_x);

   /* Wrong selection of format specifier */
   printf("int_x printed with %%u = %u\n", int_x);
   int_x = 123456;
   printf("int_x printed with %%d = %d, with %%hd = %hd\n", int_x, int_x);

   return 0;
}
```

PROGRAM 5.3: **all_about_int.c**

```
short_x: Decimal: 32767, Octal: 77777, Hex: 7fff
int_x: -65
long_x = 400000
short_x after incrementing: -32768
int_x printed with %u = 4294967231
int_x printed with %d = 123456, with %hd = -7616
```

PROGRAM OUTPUT: **all_about_int.c**

5.8 THE FLOATING POINT TYPES

A floating point number is a number with a decimal point, which divides it into integral and fractional components. The Fortran and Pascal languages as well as the scientific community understand it as a *real number*. A floating point number is suitable for representing prices, averages and ratios. It is also used for storing very large and small numbers like the number of stars in the universe or the percentage of argon in the atmosphere.

There are two ways of representing floating point numbers. The first way is the one we commonly use—a number with a decimal point. Also, as the last two items of the following examples show, you may omit digits before or after the decimal point:

```
44.1        -0.3048     .00075       7.                    7. is different from 7
```

The second way is to use the *exponential* or scientific notation. This notation uses two components—*mantissa* and *exponent*— separated by the letter `E` or `e`. It is expressed in this general form:

```
mantissa E exponent                                       Can use e instead of E
```

The *mantissa*, which is an integer or real number, is multiplied by 10 with the exponent as its power. For instance, 44.1 is equivalent to $4.41X10^1$, so it is expressed as 4.41E1 using exponential notation. Here, 4.41 is the mantissa and 1 is the exponent.

The *exponent* signifies the number of places the decimal point needs to shift to the right to convert a number to the conventional form. The point *floats* (hence, the term floating point number) and the exponent controls this float. If you increase the value of the exponent, the point has to move to the left to keep the number unchanged. So, 44.1 can be expressed as 4.41E1, 0.441E2 and 0.0441E3, as well as 441E-1. Consider some more examples:

Number	*Exponential Notation*	*Remarks*
`0.3048`	`3.048E-1`	This is 3.048×10^{-1}
`.00075`	`7.5e-4`	This is 7.5×10^{-4}
`7.`	`7.0e0` *or* `7e0`	This is 7.0×10^{0}

A floating point number has a *precision*, which signifies the number of significant digits used in its representation. For instance, the value of `PI` is `3.142` when expressed with 4-digit precision, and `3.142857` when 7-digit precision is used. Your selection of the floating point type will often be determined by the precision you need.

A floating point number used in a C program can thus be declared in two ways:

```
float price = 123.45;                          5 digits of precision
double factor = 1234.56789E10;                 9 digits of precision
```

Unlike integers, floating point numbers exist only in signed forms in C. They also use a different storage scheme. The exponent and mantissa are laid side-by-side as two separate binary numbers in

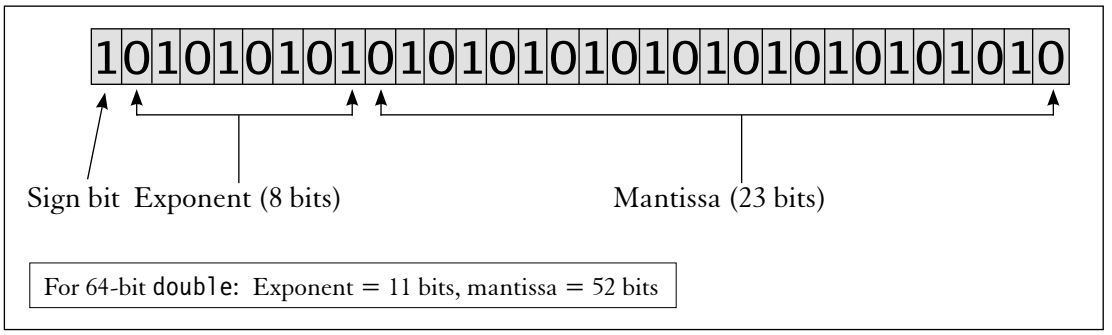


FIGURE 5.1 Memory Storage Scheme for 32-bit Floating Point Number (`float`)

memory (Fig. 5.1). One bit is reserved for the sign in the exponent and one is hidden in the mantissa. Type information tells the compiler the number of bits to use for the mantissa and exponent.

Because of the way they are stored, floating point numbers can't always be represented accurately in memory. Also, the result of an expression like `2 / 3` is in itself inexact, so its evaluated value can't be stored without loss of information. Two floating point numbers should never be tested for equality.

Note: The size of the exponent determines the range and the size of the mantissa determines the precision of a floating point number.

5.9 THE SPECIFIC FLOATING POINT TYPES

While integers are classified by their size, floating point numbers are classified by their precision. C supports three types of floating point numbers whose keywords are noted below:

- `float` (single precision)
- `double` (double precision)
- `long double` (quad precision)

Unlike with integers, ANSI specifies a *single* minimum range for all of them. This range for a positive number is 10^{-38} to 10^{+38}, but it must also satisfy the following condition:

range of `float` <= range of `double` <= range of `long double`

Though the specification permits all three types to have the same range, that is hardly ever the case. Either `double` or `long double` (or both) will have a higher range than `float`. The precision specified by ANSI and the range are now discussed with reference to the specific types.

The `float` Type ANSI requires a `float` to represent at least 6 significant digits within its minimum range of 10^{-38} to 10^{+38}. This means that it must be able to handle at least the first 6 digits in a number like 77.789654.

The `double` Type For this type, ANSI specifies a greater precision—at least 10 digits. The `double` type usually allows a very high range of values—10^{-308} to 10^{+308}.

The `long double` Type ANSI provides this type with the same precision as `double`. For systems that support this type, `long double` uses 8 to 16 bytes.

Floating point computing is slower than integer computing. The `double` and `long double` types may take up a lot of memory on your system, so don't use them unless you need a range and precision greater than what is offered by `float`. Recall that in the program **`sizeof.c`** *(5.5)*, `double` and `long double` were seen to be 8 and 12 bytes wide on a Linux system using GCC.

Table 5.2 displays the attributes of these floating point types and the format specifiers used by **`printf`** and **`scanf`** for handling them. Note that **`printf`** uses `%f` for both `float` and `double`, and prints 6 decimal places by default. For very large or small numbers, use the `%e` specifier which displays in exponential notation. **`printf`** handles `long double` using the `L` modifier with `%f` (`%Lf` or `%Le`).

Note: When you are not sure whether `%f` can display a value meaningfully, use `%g`. **`printf`** then chooses either `%f` or `%e` for display. It automatically removes all fractional digits if the value is integral.

TABLE 5.2 Attributes of Floating Point Numbers

Subtypes	*Minimum Precision*	*Size*	*Typical Values for*		**`printf/scanf`** *Format Specifier*
	(Digits)	*(Bytes)*	*Mantissa (Bits)*	*Exponent (Bits)*	
`float`	6	4	23	8	`%f`, `%e`, `%g`
`double`	10	8	52	11	`%lf`, `%le`, `%lg`
`long double`	10	8-16	112	15	`%Lf`, `%Le`, `%Lg`

5.10 `all_about_real.c`: UNDERSTANDING FLOATING POINT NUMBERS

The attributes of floating point data types are demonstrated in Program 5.4. The program first establishes the equivalence of the normal and exponential notation by displaying data using the three format specifiers. It then determines the precision of the `float` and `double` data types using a very large number. Finally, it makes a floating point computation after taking keyboard input. Uncomment the lines pertaining to `long double` if the code is run on a Linux system.

The program first initializes three "pi" variables, using the exponential notation for two of them. The first **`printf`** displays 6 decimal places for the `%f` and `%e` specifiers but removes the trailing zeroes for `%g`.

Next, a very large number comprising 20 significant digits is assigned to the variables `float_x` and `double_y`. Note that, on this machine, a `float` and `double` have a precision of 8 and 17 significant digits, respectively. Both values are higher than the ANSI-specified minimum (6 and 10).

Finally, we take keyboard input to make a simple computation using three `float` variables. Note the use of `'` (single quote) and `%%` in the last two **`printf`** statements. As observed before, **`printf`** interprets `%%` as a single `%`.

```
/* all_about_real.c: Assigns and displays the floating point types.
   Determines the precision of float and double on your machine. */

#include <stdio.h>

int main(void)
{
   float pi1 = 3.142, pi2 = 0.3142E1, pi3 = 3142E-3;
   float  float_x    = 12345.678901234567890;      /* 20 significant digits */
   double double_y = 12345.678901234567890;      /* Same */
   float sensex1 = 25341.86;
   float sensex2, gain;

   printf("pi1 = %f, pi2 = %e, pi3 = %g\n", pi1, pi2, pi3);
   printf("float_x = %.15f\n", float_x);
   printf("double_y = %.15f\n", double_y);

   /* Uncomment these 2 lines if your computer supports long double */

   /* long double ldouble_z = 12345.678901234567890;
   printf("ldouble_z = %.20Lf\n", ldouble_z); */

   printf("Enter today's value of Sensex: ");
   scanf("%f", &sensex2);
   gain = (sensex2 - sensex1) / sensex1 * 100;
   printf("Sensex returns since March 31, 2016: %f%%\n", gain);

   return 0;

}
```

PROGRAM 5.4: **all_about_real.c**

```
pi1 = 3.142000, pi2 = 3.142000e+00, pi3 = 3.142
float_x = 12345.678710937500000                    Accuracy to 8 significant digits
double_y = 12345.678901234567093                   Accuracy to 17 significant digits
Enter today's value of Sensex: 28743
Sensex returns since March 31, 2016: 13.421039%
```

PROGRAM OUTPUT: **all_about_real.c**

Takeaway: A floating point data type is selected based on the precision it can handle. The `float` and `double` types handle 6 and 10 digits of precision, respectively.

5.11 char: THE CHARACTER TYPE

C supports the `char` data type which is just wide enough to hold the values of the computer's character set. Our PCs use the 8-bit ASCII character set, so `char` by definition is one byte wide. Even though `char` is essentially an integer type, we are discussing it separately for two reasons:

- The numeric value stored in a `char` variable can easily be displayed in its equivalent character form and vice versa.
- Strings owe their origin to `char`. A string is simply an array of type `char`.

The first point implies that the value `65` stored as a `char` can be printed as `A` by **`printf`** using `%c`. Conversely, the character `A` input from the keyboard can be saved in a `char` variable as `65` by **`scanf`** using `%c`. Data of type `char` can also be represented in two other ways:

- By a character enclosed within single quotes (e.g. `'A'`).
- By a special escape sequence like `'\c'`, but only a handful of non-printable characters can be represented by an escape sequence.

Like the other integer types, `char` occurs in signed and unsigned forms. The signed form can hold values between -128 to 127, while the range for the unsigned type is 0 to 255. Unlike the other integer types, which are signed by default, the default type of `char` is system-dependent. So, if you need to store an 8-bit ASCII character, specify the data type explicitly as `unsigned char`.

5.11.1 char as Integer and Character

If `char` is unsigned, then every character of the 8-bit ASCII set can be assigned to a `char` variable. The following examples demonstrate that any integer type can be used to display a character with `%c`. The last statement should sound a beep as the BEEP character (*[Control-g]*) has the ASCII value 7:

```
char let1 = 'A'; short let2 = 97; char let3 = '9'; int let4 = 7;
```

Code	*Output*
`printf("'%c' = %d\n", let1, let1);`	`'A' = 65`
`printf("'%c' = %d\n", let2, let2);`	`'a' = 97`
`printf("'%c' = %d\n", let3, let3);`	`'9' = 57`
`printf("'%c' = %d\n", let4, let4);`	`'' = 7`

The number `7` is stored in memory as `(0000 0111)2`. But `'7'` is a character, whose ASCII value (55) is stored as `(0011 0111)2`. When you input `7` from the keyboard in response to a **`scanf`** call, it's `'7'` that is sent to the keyboard buffer *(4.8.2—How It Works)*. The `%d` specifier tells **`scanf`** to convert `'7'` to `7` (if the intent was to use an integer).

You can use octal and hexadecimal numbers too. Simply precede the number with `0` and `0x`, respectively:

```
char c = 0101;
char c = 0x41;
```

$101 = 1 \times 8^2 + 0 + 1 \times 8^0 = 65$

$41 = 4 \times 16^1 + 1 \times 16^0 = 65$

Beginners often make the mistake of using double quotes (`"A"` instead of `'A'`). Double-quoted characters (like `"A"` or `"Internet of things"`) represent a *string*, which is not a fundamental data type. C understands a string as an *array* (a derived data type) of type `char`, but more of that later.


Note: In C, `'A'` is different from `"A"`. The former is a character and the latter is a string. This string has two characters (and not one), which may puzzle you now but not after you have read Section 5.15.

5.11.2 Computation with char

Since any single-quoted character actually represents an integer, it can be used in arithmetic operations. The following assignments are all valid, but the third one has a message to convey:

```
char letB, letD, leta, digit4;
letB = 'A' + 1;                    letB is now 'B'
letD = letB + 2;                   letD is now 'D'
leta = 'A' +  32;                  leta is now 'a' but only in ASCII
digit4 = '0' + 4;                  digit4 is now '4'
```

Should we use `'A'` or `65`, its ASCII value? The question is significant because on EBCDIC systems (designed by IBM), `'A'` evaluates to 193 and not 65. If you have portability in mind, use the character rather than the numeric representation.


Note: Because letters and numerals occur in a contiguous sequence (but in separate blocks) in the ASCII list, it's convenient to sort words and lines containing these characters. A simple **`while`** loop can print a list of all letters or numerals.

5.11.3 char as Escape Sequence

C supports the *escape sequence* as a synonym for a non-printable character. An escape sequence is a two-character combination comprising a \ (backslash) and a lowercase letter. The \ forces the character following it to be treated specially. From Table 5.3, we observe that `\t` (the tab) expands to eight spaces on your terminal. The newline (`\n`) is universally used in **`printf`** to advance the cursor to the next line. The `\f` initiates a page break.

But what about those non-printable characters that can't be represented by an escape sequence? You need not worry because the \ can also be used with the ASCII value of *any* character provided this value is represented in octal or hexadecimal. Simply prefix the octal value with a \ and the hex value with `\x`:

```
char bell = '\7';                  \07 not permitted
char bell = '\x7';                 \0x7 not permitted
printf("Wake up\7");               Terminal beeps
```

The characters `'`, `"` and \ are printable, but they play a special role in framing strings and character constants. So how does one represent them literally, i.e., without their special meaning? Escape them with a \ wherever necessary:

```
char single_quote = '\'', backslash = '\\';
printf("\\ and \" are special\n" );        Prints \ and " are special
```

This mechanism, however, doesn't work with the `%` prefix of the format specifier. You can't print `%` using `\%`; you need to use `%%`.

Note: You can use a leading zero to represent an octal or hexadecimal character, but only when the character is embedded in a string. Thus, `"Wake up\a"`, `"Wake up\07"` and `"Wake up\7"` are equivalent.

TABLE 5.3 Escape Sequences

Escape Sequence	*Significance*
`\a`	Bell
`\b`	Backspace
`\c`	No newline (cursor in same line)
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\\`	Backslash
`\`*n*	ASCII character represented by the octal value *n*, where *n* can't exceed 0377 (decimal value 255)
`\x`*n*	ASCII character represented by the hex value *n*, where *n* can't exceed 0xFF (decimal value 255)

5.12 `all_about_char.c`: UNDERSTANDING THE `char` DATA TYPE

Let's now use our knowledge of `char` in Program 5.5. This program first establishes whether the `char` type is signed or unsigned on our machine. It then uses two **`while`** loops to print all the uppercase letters and numerals in a portable manner. We'll also learn to use some of the escape sequences.

Observe that even though the variable `schar` was assigned 250, it shows up as -6, which means that the MSB (most significant bit) for `schar` is set to 1. However, the value of unsigned `uchar` shows up correctly. It's clear that this system uses signed `char` by default, which means the maximum positive value it can handle is 127. Note that these `char` variables were assigned integers and also printed as integers without any problem.

The ASCII list in Appendix B shows letters and numerals arranged sequentially. A **`while`** loop thus becomes an automatic choice for printing all of them. For the first loop, the variable `letter` moves from `'A'` to `'Z'` before the loop terminates. For the second loop, `numeral` moves in reverse sequence from `'9'` to `'0'`. Using `'A'`, `'Z'`, `'0'` and `'9'` instead of their ASCII equivalents makes the code portable across all systems.

The space after `%c` and the absence of `\n` in **`printf`** ensures that the elements are printed in a single line and separated by a space. We have used a different technique this time of incrementing and decrementing a variable (with `++` and `--`).

```
/* all_about_char.c: Handling the char data type and escape sequences. */

#include <stdio.h>

int main(void)
{
   char schar = 250;
   unsigned char uchar = 250;
   char letter = 'A'; int numeral = '9';

   printf("schar = %d\n", schar);   /* Know char default on your system */
   printf("uchar = %d\n", uchar);

   /* Printing the entire alphabet in uppercase */
   while (letter <= 'Z') {          /* Using 'Z' instead of its ASCII value */
      printf("%c ", letter);        /* makes the code portable. */
      letter++;                     /* This is letter = letter + 1 */
   }

   printf("\n\t\tNext line starts after 16 spaces and also beeps\07\n");

   /* Printing all digits of the base-10 system in reverse */
   while (numeral >= '0') {         /* This code is also portable because */
      printf("%c ", numeral);       /* '0' is used instead of ASCII value. */
      numeral--;                    /* This is numeral = numeral - 1 */
   }
   return 0;
}
```

PROGRAM 5.5: **all_about_char.c**

```
schar = -6                                   Machine uses signed char by default
uchar = 250
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
                Next line starts after 16 spaces and also beeps
9 8 7 6 5 4 3 2 1 0
```

PROGRAM OUTPUT: **all_about_char.c**

Between the two loops, a **printf** statement demonstrates the use of the \ and escape sequences. A \t shifts the cursor by eight spaces, so using two of them offsets the line by 16 spaces. If your terminal supports the beep sound, you'll hear a beep as well (\07).

Note: You must know the distinction between `'A'`, `'APE'`, `"A"` and `"APE"`. `'A'` is a character constant whose ASCII value (65) is stored as a binary number. `'APE'` is not a valid character constant because only the last character (`'E'`) is stored. `"A"` and `"APE"` are legal strings, occupying *two* and *four* characters in memory, respectively. Why an extra character for the strings? Read on.

5.13 DATA TYPES OF CONSTANTS

So far, our focus has been on variables. We are yet to know what the data types of constants are. Is `32` an `int` or a `short`? Is `'B'` stored as a `char` or an `int`? Is `5.5` treated as a `float` or a `double`? We need to know the answers to all of these questions before we take up expressions in Chapter 6.

Constants represent data that don't change in the program. They also have types even though they are not formally declared. Using a set of rules, the compiler detects the data type of constants by *simple examination*. C supports a wide range of constants—both unnamed and named—which may be grouped into the following categories:

- Unnamed constants belonging to the fundamental data types. The constants `12`, `-34.5` and `'A'` belong to this category.
- Named or symbolic constants that are defined with **`#define`** at the beginning of the program. You have seen one of them used in Program 4.1.
- Variables that behave like constants when the `const` qualifier is added to the variable declaration.
- String constants like `"Nougat is here"` belong to the derived data type.

C specifies a default type for a constant, but this default is often changed by the compiler or the programmer. This can happen for the following reasons:

- The size of the constant is too large to be accommodated in the default type. For instance, the number 5000000000 is too large for an `int`, the default type for integer constants.
- You may like to force the constant to have a different type.
- The type can automatically change when a constant occurs in an expression where at least one component has a higher type.

In the following sections, we'll examine all of the constant types except string constants, discussions on which are postponed to Chapter 13.

5.13.1 Constants of the Fundamental Types

Integer Constants C specifies `int` (signed) as the default type for an integer constant. The compiler treats `655` as an `int` even though the number easily fits in a `short`. If a constant doesn't fit in its default type, the compiler tries the next higher type using the following sequence:

`int` ⟶ `unsigned int` ⟶ `long` ⟶ `unsigned long` ⟶ `long long` ⟶ `unsigned long long`

When a constant is used in an expression, we may sometimes want it to have a specific type (coercion). C offers special symbols that can be suffixed to these constants:

`U` or `u`— `unsigned int`

`L` or `l`— `signed long`

`UL` or `ul`— `unsigned long`

`LL` or `ll`— `long long`

`ULL` or `ull`— `unsigned long long`

For instance, the compiler, which otherwise sees 3 as an int, treats 3L as a long. There are no constants of type short. Next time you see the constants 540U or 450L, remember to treat them as unsigned int and signed long, respectively.

Floating Point Constants C treats a real constant as a double which takes up 8 bytes on most systems. Constants like 3.142 and 1789.456 have the type double even though a float (4 bytes) is wide enough for them. However, unlike integer constants which can only be promoted to long and long long, real constants can be both promoted and demoted:

- The F or f suffix demotes a real constant to float. The constants 3.142f and 1789.456F have the type float.
- The L or l (el) suffix promotes a real constant to long double, which may take up to 16 bytes.

These suffixes work with the exponential form also (like 3E4F). Demoting a constant to float often makes sense because it leads to savings in memory space and computation time.

Character Constants Any character or escape sequence enclosed in single quotes represents a character constant. The characters 'A', '\t' and '\7' are character constants having the ASCII values 65, 11 and 7, respectively. Strangely enough, C treats character constants as type int rather than char. This means the constant 'A' is stored in a space of 4 bytes (size of int).

Takeaway: The default type for an integer constant is int. For a real number, the default is double. A character constant is stored as an int. C doesn't support a constant of type short.

5.13.2 Variables as Constants: The const Qualifier

A constant that is used multiple times in a program is often stored as a variable. If the value is changed later, then the code needs to be modified at a single location, thus making maintenance easier. The constant itself should be stored as a variable using the **const** qualifier:

```
const float euro_dollar = 1.325;
```

The **const** qualifier protects a variable from being rewritten by marking it read-only. You'll often see this qualifier in the syntax of many C string-handling functions.

5.13.3 Symbolic Constants: Constants with Names

Storing a constant in a variable has a drawback. Every time a variable is encountered, the program has to access its memory location. Memory access takes time, so C offers *symbolic constants* to avoid this access. We have used PI as a symbolic constant in a **#define** directive of the preprocessor *(4.4)*. The euro-dollar conversion problem has a better solution in the following **#define** directive:

```
#define euro_dollar 1.325f                    No semicolon — not a C statement
```

The preprocessor replaces euro_dollar with 1.325 *everywhere* in the program. The compiler thus never gets to see euro_dollar but sees 1.325 instead. (The f suffix ensures that the default double

is not used for the constant.) If the value of a constant doesn't change elsewhere in the program or when the program is running, why use a variable at all? However, the topic appropriately belongs to Chapter 17 which discusses preprocessor directives.

Tip: If you need to use a constant repeatedly in a program, either define it as a symbolic constant or declare it as a variable using the **const** qualifier.

5.14 sizeof_constants.c: PROGRAM TO EVALUATE SIZE OF CONSTANTS

If you are concerned about possible misuse of memory, you must know the size of the constants you use in your programs. We have used the **sizeof** operator with types *(5.5)*, so let's now use Program 5.6 to repeat the exercise with constants.

The program first lists the size of every fundamental type supported on this machine to enable their comparison with constants. Observe that the constant 34 occupies 4 bytes, the size of an `int`. We know that a `char` constant is stored as an `int` too, so the constant `'A'` occupies 4 bytes. However, the `char` variable storing this constant uses just 1 byte.

```
/* sizeof_constants.c: Evaluates the size in bytes of constants. Also shows
                       the effect of the L and F suffixes on size. */
#define LOOP_VAR 50
#include <stdio.h>

int main(void)
{
   char letter = 'A';
   printf("Size in bytes of the fundamental data types on this machine:\n");
   printf("int: %d, long: %d, float: %d, double:% d, long double: %d\n",
           sizeof(int), sizeof(long), sizeof(float),
           sizeof(double), sizeof(long double));
   printf("\nSize of constants on this machine:\n");
   printf("Size of 34 = %d bytes\n", sizeof 34);
   printf("Size of 'A' = %d bytes\n", sizeof 'A');
   printf("Size of letter which is set to 'A' = %d byte\n", sizeof letter);
   printf("Size of 34L = %d bytes\n", sizeof 34L);
   printf("Size of 3.142 = %d bytes\n", sizeof 3.142);
   printf("Size of 3.142F = %d bytes\n", sizeof(3.142F));
   printf("Size of 3.142L = %d bytes\n", sizeof(3.142L));
   printf("Size of LOOP_VAR = %d bytes\n", sizeof(LOOP_VAR));

   /* Note that the size of this string constant is NOT 8! */
   printf("Size of string \"Lollipop\" = %d bytes\n", sizeof("Lollipop"));
   return 0;
}
```

PROGRAM 5.6: **sizeof_constants.c**

```
Size in bytes of the fundamental data types on this machine:
int: 4, long: 4, float: 4, double: 8, long double: 12

Size of constants on this machine:
Size of 34 = 4 bytes
Size of 'A' = 4 bytes
Size of letter which is set to 'A' = 1 byte
Size of 34L = 4 bytes
Size of 3.142 = 8 bytes
Size of 3.142F = 4 bytes
Size of 3.142L = 12 bytes
Size of LOOP_VAR = 4 bytes
Size of string "Lollipop" = 9 bytes
```

PROGRAM OUTPUT: **sizeof_constants.c**

Let's now examine the effect of suffixing the constants. Both `34` and `34L` take up 4 bytes because `int` and `long` have the same size on this machine. The program also confirms that the real number `3.142` is stored as a `double` by default (8 bytes), as a `float` (4 bytes) when the `F` suffix is used, and as a `long double` (12 bytes) when the `L` suffix is used.

The last two **`printf`** statements show how **`sizeof`** evaluates constants not belonging to the fundamental types. **`sizeof`** doesn't see the symbolic constant `LOOP_VAR` but sees the number `50`, which is an integer constant. Also note that the string `"Lollipop"` takes up 9 and not 8 characters. We'll finally explain this anomaly in the next section.

5.15 ARRAYS AND STRINGS: AN INTRODUCTION

We often have to deal with a group of related data. For instance, we may need to store the monthly rainfall for an entire year. Rather than create 12 variables, we use an *array* to store the 12 values. An array is a collection of data items belonging to a single type. We need to know some of the basics of arrays and strings now because they feature in bits and pieces in subsequent chapters. However, arrays are discussed in detail in Chapter 10, so you won't lose much by skipping this section.

The following statement declares an array having the name `month`. It is capable of holding 12 values, each of type `int`:

```
int month[12];                                    Uninitialized array
```

Assuming a 4-byte `int`, the compiler uses this declaration or definition to allocate a contiguous chunk of 48 bytes of memory. Each *element* of the array is accessed by the notation `month[`*n*`]`, where *n* is an integer representing the *index*. The value of *n* here lies between 0 and 11. The first element is accessed as `month[0]` and the last element is `month[11]`.

At this stage, the array elements are uninitialized, which means that they have junk values. You can perform the initialization at the time of declaration by assigning a set of comma-delimited values enclosed by curly braces:

```
int month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

However, if initialization is not done at the time of declaration, the elements have to be assigned separately. An array element is interpreted as a variable, so we can display its value with **printf**:

```
month[0] = 31; month[1] = 28; month[2] = 31;  .....
printf("month[1] = %d\n", month[1]);                    Prints 28
```

C arrays can hold data of any type, ranging from the fundamental types to pointers and structures. They are very useful for handling large amounts of data because a simple **while** loop can be employed to access every array element.

An array of type char has special significance in C. Every programming language needs to handle strings, and C does this by tweaking an array of type char. Let's declare and initialize an 8-element array named msg:

```
char msg[8] = {'A', 'n', 'd', 'r', 'o', 'i', 'd', '\0'};
```

The name msg represents the string Android, which takes up seven slots in the array and one more for the NUL character (shown as '\0'). NUL has the ASCII value 0 (Appendix B). Without it, msg would simply be an array of type char. Any function that operates on a string must know where it begins and where it ends. The array name provides the start point and the NUL provides the end point. The program that follows has more to say about both.

5.16 intro2arrays.c: GETTING FAMILIAR WITH ARRAYS

We conclude this chapter with Program 5.7, which works with two arrays, int_array and char_array. The program prints the elements of both arrays using different techniques. Some important conclusions can be drawn from the use of these techniques.

Like variables, arrays can be initialized either in the declaration (as in char_array), or assigned later, using separate statements for each element (as in int_array). The program employs a **while** loop to print the elements of int_array. The loop iterates three times, printing each element in the process. Observe that the tab separates each element because of the presence of \t in the **printf** control string.

A different approach is used to handle char_array. This array is initialized with 7 characters and terminated with NUL. When **printf** encounters the name char_array and also sees %s, it prints the contents of the entire array. The name char_array tells **printf** where the first element of the array is located in memory. **printf** starts printing from that location and stops when it encounters NUL. In the process, the string "Kitkat" is printed.

The last two **printf** statements conclusively prove that both char_array and the string "Kitkat" contain 7 characters. This implies that a string n characters long needs a char array of $(n + 1)$ elements. To understand how the entire string could be printed using simply the array name, we need to understand *pointers*. We'll make a detailed examination of arrays, pointers and strings in three separate chapters.

```
/* intro2arrays.c: Introduces elementary concepts of arrays. Shows that
                   a string is actually an array of characters. */
#include <stdio.h>
#define LOOP_MAX 3

int main(void)
{
   /* Declare the array index and two arrays of type int and char */
   short index = 0;
   int int_array[3];                /* Elements uninitialized*/
   char char_array[7] = {'K', 'i', 't', 'k', 'a', 't', '\0'};

   /* Integer array: Initialize and print array elements */
   int_array[0] = 111;
   int_array[1] = 222;
   int_array[2] = 333;

   printf("Elements of int_array: ");
   while (index < LOOP_MAX) {       /* Compiler sees 3, not LOOP_MAX */
      printf("%d\t", int_array[index]);
      index++;                      /* Synonym for index = index + 1 */
   }
   printf("\n");                    /* Add a newline now */

   /* Character array: Print first element and then see what happens
      when you use %s with the array name */
   printf("char_array[0] = %c\n", char_array[0]);
   printf("char_array actually stores the string \"%s\"\n", char_array);
   printf("Size of char_array: %d\n", sizeof(char_array));
   printf("Size of \"Kitkat\": %d\n", sizeof("Kitkat"));

   return 0;
}
```

PROGRAM 5.7: **intro2arrays.c**

```
Elements of int_array: 111  222  333
char_array[0] = K
char_array actually stores the string "Kitkat"
Size of char_array: 7
Size of "Kitkat": 7
```

PROGRAM OUTPUT: **intro2arrays.c**

Takeaway: A string is simply an array of type `char` that is terminated by the NUL character (`'\0'`). When **`printf`** is used with `%s` and the name of the array as its matching argument, it prints each array element until it encounters the one containing NUL.

WHAT NEXT?

Apart from being used as standalone objects, variables and constants also feature as components of expressions that use *operators*. The attributes of these data items often change—or are forced to change—when they are used in expressions. The next chapter explores these operators and expressions.

WHAT DID YOU LEARN?

Program data occur as *variables* and *constants* having different types and sizes. ANSI has specified the minimum size of the fundamental types (integer, floating point and character) and the relationship between the sizes.

A variable is *declared* by specifying its name and type. Declaration automatically *defines* a variable by allocating memory for it. A variable may be initialized at the time of declaration or may be assigned later.

The **`sizeof`** operator evaluates the size of a data type, variable or constant at compile time.

There are four types of integer variables (`short`, `int`, `long` and `long long`) where the type on the right cannot be shorter than the one on its left. The `long long` type has been included by C99 for use on 64-bit systems.

There are three types of floating point variables (`float`, `double` and `long double`). They can also be expressed in *exponential* form using a *mantissa* and *exponent*. A programmer selects the type based on the *precision* required.

The `char` data type is always 1 byte wide because it was designed to hold the entire character set of the machine. A `char` can be represented by an integer, a single quoted character or an *escape sequence*.

The data types of constants are determined by simple examination. By default, integer and character constants have the type `int`, while real constants have the type `double`. The type may be promoted or demoted by suffixing them with letters like `F`, `L` or `LL`.

C supports the *array* for containing multiple data items having the same type. The language also treats a string as an *array* of type `char` terminated by the NUL character (`'\0'`) which has the ASCII value 0.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

5.1 An identifier must begin with a letter or underscore but can contain numerals elsewhere.

5.2 A constant doesn't have a name.

5.3 A variable name can contain a hyphen except at the beginning.

5.4 An uninitialized variable does not necessarily have the value 0.

5.5 `int` and `float` are fundamental data types.

5.6 The size of a `char` is one or two bytes long.

5.7 A fractional number is also known as a real number.

5.8 The choice of the floating point type is determined by the range that is needed.

5.9 `char` data can be represented by a character enclosed within double quotes.

5.10 It is possible for the `float`, `double` and `long double` data types to have the same range.

5.11 The constant `9L` is the integer 9 stored as a `long`.

5.12 An array stores data of the same type.

5.13 The array index 12 represents the 12th element.

5.14 An array of type `char` must contain the NUL character at the end.

A2. FILL IN THE BLANKS

5.1 It is not possible to use `default`, `char` and `int` as variable names because they are _____ words.

5.2 Variable _____ makes type information available to the compiler, while the _____ allocates memory for the variable.

5.3 The compiler determines the data type of a _____ by simple examination.

5.4 The _____ operator evaluates the size of a data type, variable or a constant.

5.5 The minimum size of an `int` is _____ bytes and that of a `long` is _____ bytes.

5.6 C identifies a number with the _____ and _____ prefixes as octal and hexadecimal, respectively.

5.7 The `float` and `double` data types handle _____ and _____ digits of precision, respectively.

5.8 The constant `'S'` is stored as an _____.

5.9 The symbols `'\c'` represent an _____ _____.

5.10 The array is a _____ data type.

A3. MULTIPLE-CHOICE QUESTIONS

5.1 When the compiler sees a variable of type `int`, it allocates (A) 2 bytes, (B) 4 bytes, (C) at least 2 bytes, (D) depends on the machine.

5.2 The compiler uses type information of a variable to determine (A) the number of bytes to allocate, (B) the way the bytes are interpreted, (C) A and B, (D) none of these.

5.3 A string is (A) derived from the `char` type, (B) an array of characters, (C) not a user-defined type, (D) all of these.

5.4 The size of a `long` is (A) greater than or equal to an `int`, (B) less than or equal to an `int`, (C) greater than an `int`, (D) none of these.

5.5 A signed `char` variable can be assigned (A) an integer not exceeding 255, (B) the constant `"2"`, (C) an integer between -127 and 128, (D) none of these.

5.6 The array `int arr[20]` needs (A) 20 bytes, (B) 20 `X sizeof(int)` bytes, (C) 40 bytes, (D) 80 bytes.

5.7 The symbols `'\0'` represent the (A) octal value of 0, (B) the NUL character having the ASCII value 0, (C) a single-character string, (D) none of these.

CONCEPT-BASED QUESTIONS

5.1 List the fundamental data types along with their ANSI-stipulated minimum sizes. Why are these sizes not fixed by ANSI?

5.2 Even though the size of the fundamental data types may vary across systems, the `char` data type is always one byte wide. Explain.

5.3 Name at least three sources of data that are available to a program.

5.4 Spot the valid variable names in the list: (i) `bit-rate`, (ii) `upload_speed`, (iii) `__width`, (iv) `a1`, (v) `2017_month`.

5.5 When does it make sense for a constant to be stored in a variable? How does **#define** differ from the **const** keyword?

5.6 Explain the difference between *declaration* and *definition* of a variable. Can you declare a variable without defining it?

5.7 What is the difference between a string and an array of type `char`? How many array elements does the string `"abcd"` use?

5.8 Will the following code segment print properly? Explain with reasons.

```
char stg1[6] = {'N', 'o', 'u', 'g', 'a', 't'};
char stg2[] = "Nougat";
printf("%s\n", stg1);
printf("%s\n", stg2);
```

5.9 Since `'A'` is actually an integer, will **`printf("%c", 'A');`** display the same output as **`printf("%d", 'A');`**? Explain with reasons.

5.10 Between using 6 variables and an array of 6 elements, what considerations determine the choice of one scheme in preference to the other?

PROGRAMMING & DEBUGGING SKILLS

5.1 Why does the following code section print a negative integer? How do you rectify it?

```
short x = 40000;
printf("x = %hd\n", x);
```

5.2 Write only assignment statements to swap the values of two variables x and y using a temporary variable `t`.

5.3 What will the following program display on a computer with a 2-byte `short` and 4-byte `int`? Provide the arithmetic to support your answer. Also change the program to print correctly.

```
#include <stdio.h>
int main(void)
{
   short s = 40000;
   int i = 2200000000;
   printf("s = %hd, i = %d\n", s, i);
   return 0;
}
```

5.4 Write a program that accepts a positive integer from the user and prints the value in decimal, octal and hex.

5.5 Write a program that assigns the maximum values of an unsigned `short` (typically, 2 bytes) and unsigned `int` (typically, 4 bytes) to two variables. What is the output when you print one more than this maximum value with **`printf`**? Explain with reasons.

5.6 Study the following program and answer the following questions:

(i) What does this program display?

(ii) Is there any difference between the two lines of output?

(iii) Why was the second **`printf`** statement provided?

(iv) If both loops performed an additional iteration, what would be the next item to be printed?

```
#include <stdio.h>
int main(void)
{
   char c = '0'; short s = 0;
   while (c <= '9') {
      printf("%c ", c);
      c = c + 1;
   }
   printf("\n");
   while (s <= 9) {
      printf("%hd ", s);
      s++;
   }
   return 0;
}
```

5.7 Write a program that uses two **`while`** loops to populate a 6-element `long` array with the first 6 positive integers and a 7-element `char` array with the first 6 (not 7) letters of the English alphabet. After printing both arrays, set the 7th element of the `char` array to `'\0'` and then reprint this array. Explain your observations.

6 Operators and Expressions

WHAT TO LEARN

- Attributes of *operators* and their role in evaluating *expressions*.
- Significance of the *arithmetic* operators.
- How data types of variables and constants are *implicitly* changed.
- Role of *casts* in *explicit* conversion of data types.
- Using operator *precedence* and *associativity* in evaluating expressions.
- Significance of the *assignment* operators (`+=`, `-=`, `*=`, etc.).
- The increment (`++`) and decrement (`--`) operators and their *side effects*.
- Boolean evaluation of expressions using *relational* and *logical* operators.
- Use of the *conditional* and **`sizeof`** operators.

6.1 EXPRESSIONS

Variables and constants represent data in their most elemental form. These data objects become useful when combined with one another to form *expressions*. The components of an expression are linked to one another by symbols called *operators*. For instance, `a + b / 7` is an expression where `+` and `/` are operators. The expression has a *value* which here is evaluated arithmetically.

An arithmetic expression is not the only type of expression you'll encounter in C. There are other types—like relational and logical expressions. For instance, `a > b` is not an arithmetic expression but it still has a value. The following examples illustrate the diversity of expressions used in C:

C Construct	*Expression*	
`interest = principal * rate / 100;`	`principal * rate / 100`	*(arithmetic)*
`if (age > 18)`	`age > 18`	*(relational)*
`while (answer == 'y')`	`answer == 'y'`	*(relational)*
`if (age > 15 && age < 25)`	`age > 15 && age < 25`	*(logical)*

(*Continued*)

(*Continued*)

C Construct	*Expression*	
`while (count-- >= 0)`	`count-- >= 0`	*(relational)*
`answer = 'y';`	`answer = 'y'`	*(assignment; also an expression)*
`++count;`	`++count`	*(expression without operators)*
`if (count)`	`count`	*(Ditto; simplest expression)*

Except for the last one, all of these expressions use one or more operators. The simplest expression, however, uses no operators. This means that the variable `count` and the constant `5` are two valid expressions. This all-embracing view of expressions may seem trivial but it has important consequences, as you will gradually discover.

Every C expression has a *value* and the significance of this value depends on the nature of the expression. For an arithmetic expression (like `a + b`) it's the computed value that interests us. But relational expressions (like `age > 18`) evaluate to either of two boolean values, which we interpret as *true* (value > 0) or *false* (value = 0).

6.2 OPERATORS

Operators form the connecting link between variables and constants in expressions. An operator acts on one or more *operands*. For instance, in the expression `x - 5`, the `-` symbol is an operator, which acts on the operands `x` and `5`. C supports a rich collection of operators—richer than any language known previously. These operators can be grouped into the following types:

- Arithmetic operators (like `+`, `-`, etc.)
- Assignment operators (like `=`, `+=`, `-=`, `*=`, etc.)
- Increment and decrement operators (`++` and `--`)
- Relational operators (like `>`, `<`, `<=`, etc.)
- Logical operators (`&&`, `||` and `!`)
- Bitwise operators (like `&`, `|`, `>>`, `<<`, etc.)
- The comma, **`sizeof`** and cast operators

An operator may be *unary*, *binary* or *ternary*. A unary operator has one operand. For instance, the `-` symbol in the negative constant `-5` is a unary operator which multiplies its operand by -1. (The unary `-` is different from the binary `-` which performs subtraction.) The `++` in `count++` is a unary operator which increments its operand by one.

Most operators are of the binary type which needs two operands. The assignment operator `=` in `x = 5` is a binary operator. Arithmetic operators like `*` and `/` and relational operators like `>` and `<` are also binary operators. C has one ternary operator which uses the `?` and `:` symbols.

When an expression contains multiple operators, the order of their evaluation becomes important. For instance, when evaluating the expression `x + y * z`, does multiplication occur before addition? For this purpose, C specifies two attributes for each operator:

- The *precedence* (priority) ensures that a high-priority operator is acted upon before a low-priority one. In the expression `x + y * z`, the * has a higher precedence than +, so multiplication will be done first.
- The *associativity* determines the direction of evaluation (left-to-right or right-to-left). In the expression `x * y / z`, it is associativity and not precedence that determines that multiplication will be done first. You may think that it doesn't matter, but if `y` and `z` are integers, then it definitely matters *(6.10.3)*.

These two attributes will be examined in detail later in the chapter. It's important that you understand them *completely* because a little learning in this area would certainly be a dangerous thing.

Note: The constant `5` is an expression but `5;` is a valid statement. Since many functions *return* a value, they can also be considered as expressions. The **`printf`** function returns a value and so can be used anywhere an expression is used. Our first program, **`expressions.c`**, clearly demonstrates this.

6.3 `expressions.c`: EVALUATING EXPRESSIONS

Program 6.1 features a set of **`printf`** statements that print the values of expressions. One of these expressions is **`printf`** itself. Finally, the program executes a variable and constant as statements. Because these statements have no effect on the program, the compiler may generate warnings that may be ignored.

Let's examine the five **`printf`** statements that follow the initial one. The first four of them use binary operators, but the last one uses a unary one. For identifying the statements, we'll use the second argument to **`printf`** as the subhead in the following discussions:

- `distance/mileage` This is a simple arithmetic expression whose value is obtained by floating-point division. The `%.2f` specifier overrides the default decimal width of 6.
- `mileage = 20` This expression also has a separate value, which is the value of the operand on the right side of `=`. So, both `mileage` and the expression `mileage = 20` have the value 20.
- `answer == 'y'` The `==` operator checks for equality by comparing `answer` to `'y'`. This is a relational expression that evaluates to either 0 (false) or 1 (true). It is obviously false.
- `answer = 'y'` Unlike the previous expression, this expression is an assignment and has the value `'y'`, which in ASCII is 121.
- `++mileage` The `++` unary operator increments a variable so **`printf`** outputs the value 21.

A possible jolt could come from the last **`printf`** statement which contains another **`printf`** as its second argument *but without the semicolon*. Observe that both **`printf`** functions in the statement have been executed, while the value 10 is also printed. You'll soon know that **`printf`** can be interpreted as an expression, whose return value (10) is the number of characters printed (9 from `"Honeycomb"` and 1 from `\n`).

The last two statements contain harmless expressions that are executed as statements. These expressions evaluate to 21 and 420, respectively. They are valid C statements even if they do nothing.

```
/* expressions.c: Evaluates value of expressions. Establishes that
   an assignment or function like printf is also an expression. */

#include <stdio.h>

int main(void)
{
   int mileage = 17; float distance = 420; char answer = 'n';

   printf("Printing value of expressions:\n");
   printf("distance / mileage has the value %.2f\n", distance / mileage);
   printf("mileage = 20 has the value %d\n", mileage = 20);
   printf("answer == 'y' has the value %d\n", answer == 'y');
   printf("answer = 'y' has the value %d\n", answer = 'y');
   printf("++mileage has the value %d\n", ++mileage);

   /* Surprise! printf is also an expression. */
   printf("This printf expression has the value %d\n", printf("Honeycomb\n"));

   /* Expressions as statements */
   mileage;                    /* Perfectly valid statement */
   420;                        /* Same, so ignore the warnings */

   return 0;
}
```

PROGRAM 6.1: **expressions.c**

```
Printing value of expressions:
distance / mileage has the value 24.71
mileage = 20 has the value 20
answer == 'y' has the value 0
answer = 'y' has the value 121                    ASCII value of 'y'
++mileage has the value 21
Honeycomb
This printf expression has the value 10
```

PROGRAM OUTPUT: **expressions.c**

Takeaway: An expression which is not a function evaluates to either an absolute or a boolean value (0 or 1). For a function, its evaluated value is the value returned by it on execution. However, the **printf** function not only returns a value but also has the *side effect* of printing something. It is strange that printing, the main task of **printf**, is "relegated" to a "side effect!"

6.4 ARITHMETIC OPERATORS

We begin our discussion on the individual operators with the five arithmetic ones: +, -, *, / and % (Table 6.1). Your electronic calculator uses the first four in identical manner. The %, known as the *modulus* operator, is used in C for computing the remainder of an integer division. C permits some flexibility in arithmetic operations, but it places some restrictions as well.

6.4.1 The +, -, * and /: The Four Basic Operators

The operators + and - are meant to do what everyone knows. Multiplication and division are handled by * and /. These four binary operators support a mix of integer and floating point operands which must have the right data types. Here are some situations that can cause problems:

```
256 * 256       Value is 65536; can't hold in a short
22 / 7          Value is 3 not 3.142!
22.0 / 7        Value is 3.142857; 7 converted to 7.0
5 / 0           Program aborts; division by zero not permitted
```

As evident from the first example, multiplication is the most common source of data overflow problems. The second example is an eye-opener; it evaluates to 3 because integer division leads to truncation of the decimal portion. The next example shows that an expression is evaluated using floating point arithmetic only if at least one operand is of the floating point type. Finally, your programs must have adequate checks to ensure that the denominator is never zero.

6.4.2 The %: The Modulus Operator

The other arithmetic operator we consider is the %, known as the *modulus* operator. This operator evaluates the remainder of a division between two integers. The expression `10 % 7` (10 modulo 7) evaluates to 3. Consider a few more examples:

```
55 % 2          Value is 1; suitable for odd or even number check
21 % 7          Value is 0; suitable for day of the week check
5 % 8.0         Error; only integers allowed
-14 % 5         Implementation dependent; usually evaluates to -4
```

As seen from the third example, the operation fails if one of the operands is a real number. The % provides a simple solution to some common tasks like checking for even or odd numbers and determining whether a year is a leap year or not.

TABLE 6.1 The Arithmetic Operators

Operator	*Significance*
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (integers only)

6.5 computation.c: MAKING SIMPLE CALCULATIONS

Let's use the arithmetic operators in our next program (Program 6.2). The program takes as input an even number from the user and computes the sum of all integers from 1 to the value that is input. The computation uses an interesting technique that is documented in the program itself. A validation check ensures that you don't key in odd integers. The program is invoked three times but with only one successful outcome.

```
/* computation.c: Uses arithmetic operators to compute the sum of the
   first n integers where n is even. Quits program if n is not even. */
#include <stdio.h>

#define FIRST_NUMBER 1
int main(void)
{
   int last_number, pairs, result, sum_pair;

   printf("Enter the last number: ");
   scanf("%d", &last_number);
   if ((last_number % 2) == 1) {          /* Can't do without % */
      printf("Not an even number\n");
      return 0;                           /* Terminates program */
   }

   /* Example: For calculating sum of first 100 integers, first divide the
      integers into pairs. Sum of 1 and 100 = 101, sum of 2 and 99 = 102, etc.
      Final sum = Sum of each pair * number of pairs */

   sum_pair = FIRST_NUMBER + last_number;
   pairs = sum_pair / 2;
   result = pairs * sum_pair;

   printf("Sum of first %d integers = %d\n", last_number, result);
   printf("Difference between first and last integer = %d\n",
          last_number - FIRST_NUMBER);

   return 0;
}
```

PROGRAM 6.2: **computation.c**

```
Enter the last number: 5
Not an even number
```

```
Enter the last number: 100
Sum of first 100 integers = 5050
Difference between first and last integer = 99
```

```
Enter the last number: -6
Sum of first -6 integers = 10                          Meaningless
Difference between first and last integer = -7         This is correct
```

PROGRAM OUTPUT: **computation.c**

A symbolic constant (FIRST_NUMBER) makes an appearance in this program. The preprocessor replaces FIRST_NUMBER with 1 at the two locations of its occurrence in the program. You can change the starting point by changing this value. Note that we can't check whether a number is even or odd without the % operator. The program documents the logic behind the algebraic formula sum = n × (n + 1) / 2. The validation check rejects odd integers but it is not good enough to prevent the input of negative and floating point numbers.

Note: While integer division causes truncation of the fractional component, the truncated portion can be obtained by using the % operator. You then need two operations to get both quotient and remainder.

6.6 AUTOMATIC OR IMPLICIT TYPE CONVERSION

When numbers of different data types are used in an expression, the compiler converts a lower type to a higher type using the concept of *implicit promotion*. C finally converts the type of the expression itself to the highest type it finds there. Consider this simple expression that involves an assignment:

```
float pi = 22 / 7.0;
```

Here, the three variables and constants have three different data types. Recall that `7.0` has the type `double` and `22` is of type `int` *(5.13.1)*. The compiler doesn't think that the types are incompatible and proceeds with type conversion in the following sequence:

- It looks at the operands of the / operator and finds `7.0` to have a higher type than `22`. It converts `22` to `double` as well, so the resultant expression also has the type `double`. This means that `sizeof(22 / 7.0) = sizeof(double)`.
- It then looks at the operand on the left of the `=`. This is a variable of type `float`, which has a lower type than `double`. But since C can't change the type of a variable that has been declared with a specific type, it *demotes* the type of the entire expression on the right to `float`. This can sometimes lead to loss of information but it won't happen here.

The preceding example featured a simple expression, but for complex expressions containing multiple data types, the compiler uses the following sequence for conversion:

1. All `char` and `short` operands are converted to `int` or `unsigned int`, if considered necessary.

2. For the remaining operands of every operator, if they have two different types, the lower type is converted to the higher type using the following sequence:

   `long double` ⟶ `double` ⟶ `long long` ⟶ `long` ⟶ `int`

   This means the compiler first looks for `long double`, and if it finds one, it converts its matching operand to `long double`. It repeats this exercise for the next lower type (`double`) and continues until all operands have been examined. The scanning cannot go beyond `int` because `char` and `short` have already been converted.

3. Eventually, the entire expression is converted to the highest type found. And if the expression is assigned to a variable, the type of the expression is changed to match the type on the left of the `=`. This can lead to either promotion or demotion (loss of information).

The term "conversion" is a misnomer because type changes are made only to the *copies* of the operands. Also, in the process of conversion, if the default (signed) type cannot hold the data even after promotion, then the unsigned type is chosen.

6.7 implicit_conversion.c: PROGRAM TO DEMONSTRATE IMPLICIT CONVERSION

This two-part program (Program 6.3) confirms some of the assertions made in Section 6.6. The first part features the automatic conversion of `short` and `char` to `int`. The second part shows how to use the implicit conversion feature for preventing truncation in integer division. The third statement in this part conveys a simple message: our knowledge of type conversion is not complete yet.

Part 1 The **`sizeof`** operator clearly reveals the outcome of type conversion. The expression `char1 * char2` is seen to have a width of 4 bytes (size of `int`), which means that `char1` and `char2` are individually promoted to `int` in an expression. A similar reasoning also applies to the `short` variables.

```
/* implicit_conversion.c: Shows automatic conversion of char, short and int. */
#include <stdio.h>

int main(void)
{
   char  char1 = 1, char2 = 1;
   short short1 = 1, short2 = 1;
   int int1 = 9, int2 = 5;

   /* 1. char and short automatically promoted to int */
   printf("Size of char1 = %d\n", sizeof char1);
   printf("Size of char1 * char2 = %d\n", sizeof(char1 * char2));
   printf("Size of short1 = %d\n", sizeof short1);
   printf("Size of short1 * short2 = %d\n", sizeof(short1 * short2));

   /* 2. Automatic conversion of int to float. First 2 cases prevent
         truncation in integer division, but not 3rd one. Why? */

   /* a. Write one integer as a real number */
         printf("1. 9.0f/5 evaluates to %f\n", 9.0f / 5);
   /* b. Multiply by 1.0 before division of 2 int variables */
         printf("2. 1.0*int1/int2 evaluates to %f\n", 1.0 * int1 / int2);
   /* c. Multiply by 1.0 after division */
         printf("3. int1/int2*1.0 evaluates to %f\n", int1 / int2 * 1.0);

   return 0;
}
```

PROGRAM 6.3: **`implicit_conversion.c`**

```
Size of char1 = 1
Size of char1 * char2 = 4                    Both variables promoted to int
Size of short1 = 2
Size of short1 * short2 = 4                  Both variables promoted to int
1. 9.0/5 evaluates to 1.800000
2. 1.0*int1/int2 evaluates to 1.800000
3. int1/int2*1.0 evaluates to 1.000000
```

PROGRAM OUTPUT: **`implicit_conversion.c`**

Part 2 Truncation in the evaluation of the expression `9 / 5` can be prevented by converting one operand to a real number (say, `9.0f` instead of `9`). The other operand is then implicitly converted to `float`. This technique doesn't work with variables, so in the next two cases, the multiplier `1.0` is used. Observe that the expression evaluates correctly when `1.0` is placed at the beginning but not when it is placed at the end! It seems something went wrong with the *order of evaluation* of the expression *(6.10.3)*.

Takeaway: There are two ways of dividing two integers correctly (i.e., without truncation). If at least one integer is a constant, convert it to a `float` (like using `9.0f` or `9.0F` instead of `9`). When both are variables, artificially induct the constant `1.0` at the *beginning* of the expression. There's a third solution which is also the best one *(6.8)*.

6.8 EXPLICIT TYPE CONVERSION—THE TYPE CAST

The technique of using the `F`, `L` and `LL` suffixes to change the data type of constants was discussed in Section 5.13.1. This technique obviously doesn't apply to variables. The ultimate conversion tool for *explicit conversion* or *coercion* is the *type cast* or *cast*. It has the following syntax:

(*data type*) *variable, constant or expression*

A cast is a unary operator having two components: the name of the data type (`int`, `float`, etc.) enclosed in parentheses, followed by the data item that needs conversion. The data item can be any variable, constant or expression of any type. For instance, `(float)` represents a cast, and `(float) x` evaluates `x` as a `float` even though `x` may actually be of type `int`.

The familiar expression `9 / 7` can now be evaluated as a `float` without using any artificial techniques:

```
int x = 9, y = 7;
float z1 = (float) x / y;          Cast works with variables ...
float z2 = (float) 9 / 7;          ... and constants.
```

The cast i.e., `(float)`, re-evaluates its operand, `x` or `9`, as a `float`. The computation is then performed using floating point arithmetic. As with the conversion techniques discussed previously, a cast can't change the type of its operand; it simply copies it to a different type.

We don't use a cast only to preserve information. To compute the sum of three numbers after discarding their fractional components, use a separate cast for each number:

```
float f1 = 1.5, f2 = 2.5, f3 = 3.7;
int sum = (int) f1 + (int) f2 + (int) f3;          sum is 6
```

Unlike the previous type conversion schemes, a cast can also use an entire expression as its operand. The expression is first evaluated according to its own rules and then re-evaluated using the cast. This is how the sum of three real numbers is computed before discarding the fractional component from the sum:

```
float f1 = 1.5, f2 = 2.5, f3 = 3.7;
int sum = (int) (f1 + f2 + f3);          sum is 7
```

These examples suggest that the cast operation occurs before other operations. For instance, when the compiler encounters the expression `(int) f1 + f2`, it evaluates `f1` as an `int` *before* adding it to `f2`. Is the *priority* of the cast higher than addition? We'll soon find out.

6.9 casts.c: PROGRAM TO DEMONSTRATE ATTRIBUTES OF A CAST

Program 6.4 demonstrates three features of casts. First, it uses a cast to perform floating point arithmetic with two integers. Second, it establishes that the cast operation occurs before division. Finally, it shows how **`sizeof`** evaluates a variable before and after a cast. You need to ask yourself why **`sizeof`** needs parentheses in the second instance even though the operand is not an expression *(6.19.2)*.

Why can't we get rid of the `(float)` cast by declaring `int1`, `int2` and `int3` as `float` variables instead of `int`? Yes, we can do that, but these variables could be used elsewhere as `int`, so declaring them as `float` simply to suit a single operation doesn't make sense.

```
/* casts.c: Demonstrates the use of casts. */
#include <stdio.h>

int main(void)
{
   int int1, int2, int3;
   float average;

    printf("The value of pi is %.8f\n", (float) 22 / 7);

   printf("Enter three integers separated by a space: ");
   scanf("%d %d %d", &int1, &int2, &int3);
   average = (float) (int1 + int2 + int3) / 3;

   printf("Average of the three integers = %f\n", average);
   printf("Size of int1 = %d\n", sizeof int1);

   /* Parentheses for sizeof not required above but required below */
   printf("Size of (short) int1 = %d\n", sizeof ((short) int1));

   return 0;
}
```

PROGRAM 6.4: **`casts.c`**

```
The value of pi is 3.14285707            8 decimal places because of %.8f
Enter three integers separated by a space: 3 5 6
Average of the three integers = 4.666667
Size of int1 = 4
Size of (short) int1 = 2
```

PROGRAM OUTPUT: **`casts.c`**

6.10 ORDER OF EVALUATION

Proper use of data types and conversion techniques are not enough to evaluate expressions correctly. You also need to use a specified *order of evaluation*. You probably remember the "BODMAS" acronym from your school days, which specified that division is performed before addition (D before A in BODMAS). C too has a set of rules for all the 40 operators of the language. These rules, which we'll now discuss, are based on two operator attributes: *precedence* and *associativity*.

6.10.1 Operator Precedence

Every operator has a *precedence* which signifies the priority of its operation. The precedence is often interpreted in this book in terms of *levels*. An operator belonging to a higher level has a greater precedence or priority than a lower-level operator. The levels for some of the operators that we have already discussed can be represented in the following manner:

Level 1 — (and)
Level 2 — + (unary), - (unary)
Level 3 — `sizeof`
Level 4 — (*cast*)
Level 5 — *, / and %
Level 6 — + (binary) and - (binary)
Level 7 — =

The * and / operators belong to a higher level than + and -. So, the expression `32 + 9 / 5` is evaluated by performing the division before the addition. Note that the cast operator has a lower precedence than **`sizeof`** but a higher precedence than the arithmetic operators. The complete table of operator precedence has over 15 levels (Table 6.2 and Appendix A), but the above data collated from the table is adequate to explain how C handles the following expression:

```
fahrenheit = 32 + -40.0 * 9 / 5;
```

This represents the well-known formula that converts temperatures between Celsius and Fahrenheit. The constant `-40.0` represents the temperature in Celsius. This is how C evaluates the expression:

1. Among these operators, the unary operator, -, has the highest level, so C first multiplies `40.0` with `-1`.

2. The sub-expression, `-40.0 * 9 / 5` is then evaluated because the * and / have a higher priority than the binary +. Here, we have two operators belonging to the same level. For reasons of associativity that will be explained later, C performs the multiplication before the division, but before doing so, it promotes the type of `9` to `double` (the type of `-40.0`).

3. The sub-expression at this stage has the value -360.0 which has to be divided by 5. C now promotes the type of `5` (by default, `int`) to `double` also. Division of -360.0 by 5.0 yields -72.0.

4. C now performs the addition operation, but only after promoting the type of `32` to `double`. The final result is 32.0 + -72.0 = -40.0.

5. The = operator has the lowest precedence in our list, so the result of the previous operation is finally assigned to `fahrenheit`. If `fahrenheit` is of type `double`, the right-hand side needs no conversion, but if it is a `float`, then the expression will be converted to `float` before assignment. This could lead to truncation but not with the value that we are working with.

6.10.2 Using Parentheses to Change Default Order

Our abridged list of levels shows the parentheses having the highest precedence. Thus, their use at any location changes the default order of evaluation at that location. If we were to rework the previous equation to convert a given temperature in Fahrenheit (say, -40F) to Celsius, we need to use parentheses:

```
celsius = (-40 - 32) * 5 / 9
```

Here, the expression `-40 - 32` is evaluated first (but after multiplying `40` by `-1`). In the absence of parentheses, `-32` would have been multiplied by `5` first. This would have resulted in erroneous evaluation.

Note: The expression `-40 - 32` needs two operations for evaluation and not one. The first `-` is the unary minus which multiplies `40` by `-1`. `32` is next subtracted from the result. However, C treats `-40 -32` and `-40 - 32` as equivalent.

6.10.3 Operator Associativity

Every operator has an *associativity* which determines the direction of evaluation when an operand is shared by two operators *having the same precedence*. This attribute can take on two values—right-to-left (R-L) or left-to-right (L-R). To understand the implications of this property, consider the following statements that use the same operators and operands but arranged in a different sequence:

```
printf("%.2f\n", -40.0 * 9 / 5 + 32);          Prints -40.00
printf("%.2f\n", 9 / 5 * -40.0 + 32);          Prints -8.00
```

The first statement prints the correct value. Here, the * and / have the same precedence and also share an operand (`9`). From Table 6.2, it is seen that * and / have left-to-right associativity. This means that the operand `9` will be operated upon from the left. Thus, multiplication will be done first, followed by division.

But why does the second **`printf`** output a different result? Here, it's the `5` that is shared by the same operators, / and *. Because of L—R associativity, division occurs first, but this also leads to truncation. So the final result is $1 \times -40.0 + 32 = -8.0$. However, we could have prevented truncation by using the cast `(float) 9`.

Now comes the inevitable question that has often been answered incorrectly. How is the expression `a * b + c / d` evaluated where the equal-priority operators, * and /, don't share an operand? All we can say is that the addition will be done last. The outcome is implementation-dependent even though the final result won't be affected. Associativity has no role to play here.

Takeaway: Operator associativity becomes a determining factor only when two operators having the same precedence share an operand. When equal-priority operators don't share an operand, the outcome is implementation-dependent but it won't lead to an error. An exception can be made when using the logical operators *(6.16)*.

Note: Table 6.2 lists all operators featured in the 17 chapters of this book. The bitwise operators (discussed in Chapter 17) are included in the complete version of the table (Appendix A).

TABLE 6.2 Operator Precedence and Associativity (Complete table in Appendix A)

Operator	*Significance*	*Associativity*
`( )`	Function call	L-R
`[ ]`	Array subscript	
`.`, `->`	Member selection (for structures)	
`++`, `--`	Postfix increment/decrement	
`++`, `--`	Prefix increment/decrement	R-L
`+`, `-`	Unary	
`!`	Logical NOT	
`sizeof`	Size of data or type	
(*data type*)	Cast	
`*`, `&`	Dereference/address (for pointers)	
`*`, `/`, `%`	Arithmetic	L-R
`+`, `-`	Arithmetic (binary)	L-R
`<`, `<=`, `>`, `>=`	Relational	L-R
`==`, `!=`	Relational	L-R
`&&`	Logical AND	L-R
`||`	Logical OR	L-R
`? :`	Conditional (ternary)	R-L
`=`, `+=`, `-=`, `*=`, `/=`, `%=`	Assignment	R-L
`,`	Comma (expression delimiter)	L-R

6.11 `order_of_evaluation.c`: PRECEDENCE AND ASSOCIATIVITY

Our next program (Program 6.5) confirms the observations made on operator precedence and associativity in the preceding sections. The first part shows what the relative precedence of six operators are and how they can be changed using parentheses. The second part represents an excellent programming tip related to associativity.

```
/* order_of_evaluation.c: Demonstrates precedence and associativity. */
#include <stdio.h>

int main(void)
{
   float f, celsius;
   short year = 2014, years_left;

   /* 1. Precedence issues */
   f =  7 * 8 + 9 - 16 / 4 ;           /* 56 + 9 - 4 = 61 */
   printf("1. %.2f\n", f);

   f =  7 * (8 + 9) - 16 / 4 ;         /* 7 * 17 - 4 = 115 */
   printf("2. %.2f\n", f);

   years_left = 4 - year % 4;          /* No parentheses required */
   printf("3. years_left = %hd\n", years_left);

   /* 2. Associativity issues */
   printf("4. Enter temperature in Celsius: ");
   scanf("%f", &celsius);

   printf("5. %.2fC = %.2fF (Correct)\n", celsius, celsius * 9 / 5 + 32);
   printf("6. %.2fC = %.2fF (Incorrect)\n", celsius, 9 / 5 * celsius + 32);

   return 0;
}
```

PROGRAM 6.5: **order_of_evaluation.c**

```
1. 61.00
2. 115.00
3. years_left = 2
4. Enter temperature in Celsius: 40
5. 40.00C = 104.00F (Correct)
6. 40.00C = 72.00F (Incorrect)
```

PROGRAM OUTPUT: **order_of_evaluation.c**

Part 1 This part features three simple expressions that use the arithmetic operators. It's now a given that * and / have a higher precedence than + and -. But in the expression used for a leap year check, note that `year % 4` is computed before the subtraction because % has a higher precedence than -.

Part 2 Refer to Part 2 of Program 6.3 (**implicit_conversion.c**), where we noted that the constant `1.0` must be placed at the *beginning* of the expression to prevent truncation in division. We have achieved the same result here by placing the variable `celsius` before `9 / 5`. The explanation is simple: since / and * have the same precedence and L-R associativity, `celsius * 9` is evaluated first (but only after `9` has been converted to `double`).

Tip: If you encounter an expression in the form `b / c` or `b / c * a`, where `b` and `c` are of type `int` and `a` is of type `float`, you can easily prevent truncation of the decimal part. For `b / c`, either multiply the expression by `1.0` at the beginning or use a cast for `b` or `c`. For `b / c * a`, rearrange the operands to have `a * b / c`, in which case a cast is not necessary.

6.12 ASSIGNMENT OPERATORS

Let's now resume our examination of the essential C operators by taking up the assignment and increment/decrement operators. They are closely related to the arithmetic ones except that they have the *side effect* of changing their operand. In this section, we discuss the operators related to assignment (Table 6.3). These binary operators evaluate the expression on their right and assign the result to the variable on the left.

6.12.1 The =: The Simple Assignment Operator

The = operator assigns the expression on its right to a variable on its left. The variable can be of any type—including array elements and pointers—but we'll begin with a simple assignment:

```
int max_tries = 5;
```

Is this *assignment* similar to the algebraic *equation* `max_tries = 5`? No, because in C, the left side of the = must be a variable and not an expression like this one:

```
max_tries - 5 = 0;                    Makes perfect sense in algebra but not in C
```

You can have different types for the two operands, with or without adverse consequences. The type of the expression is promoted or demoted to match the type of the variable:

```
float amount = 100;                   100 converted to float; no problem
int balance = 100.5;                  100.5 converted to int; truncation occurs
```

The = has a very low precedence and R-L associativity (Table 6.2). Thus, low precedence allows an expression containing virtually any operator to be evaluated before assignment. You should now understand the significance of the following statement that has been used a number of times before:

```
count = count + 1;                    Makes perfect sense in C but not in algebra
```

The + has a higher precedence than =, so addition is performed first with the current value of `count`. The evaluated value is then assigned to the variable `count` on the left of the =. In the process, `count` is incremented by one.

6.12.2 The Other Assignment Operators

The other operators of this family use the = in combination with each of the arithmetic operators. There are thus five of them: `+=`, `-=`, `*=`, `/=` and `%=` (Table 6.3). Each of these two-character operators performs a simple arithmetic operation on a variable on its left using an expression on its right, *and then stores the result in the same variable.* This is how we use the += operator to add 5 to the existing value of `count`:

```
count += 5;                           Equivalent to count = count + 5
```

This statement represents two separate tasks performed with a single operator. The += evaluates the expression `count + 5` before assigning the result to `count`. The evaluation is considered to be the "main" task and the assignment is interpreted as a *side effect*. The other two-character assignment operators (like -=) and the operators, ++ and --, also have side effects. Like the =, all of these operators have low precedence and R-L associativity.

TABLE 6.3 The Assignment Operators

Operator	*Example*	*Equivalent to*
=	`count = 5;`	-
+=	`count += 3;`	`count = count + 3;`
-=	`count -= 3;`	`count = count - 3;`
*=	`count *= 4;`	`count = count * 4;`
/=	`count /= 4;`	`count = count / 4;`
%=	`count %= 5;`	`count = count % 5;`

6.13 ++ and --: INCREMENT AND DECREMENT OPERATORS

A programmer frequently needs to increment or decrement a variable by one—especially inside a loop. C provides shortcuts for performing these two operations: the unary ++ and -- operators. Each operator is offered in two varieties which are shown below for the ++ operator:

```
count++;                    Both expressions are identical to ...
++count;                    ... count += 1; and count = count + 1;
```

In either case, `count` is incremented by one. The ++ suffix used in the first example is known as a *postfix* operator. The second example shows the ++ used as a *prefix* operator. We have identical versions for the -- operator also. Both statements below decrement `count` by one:

```
count--;                    -- is a postfix operator
--count;                    -- is a prefix operator
```

The value of `count` changes every time these expressions are invoked. Even though there is a difference between the prefix and postfix versions, they can be used interchangeably when used in a standalone mode as above. They have L-R associativity but they don't have the same precedence. The postfix operators have a higher precedence than the prefix ones (Table 6.2).

6.13.1 Side Effect of ++ and --

Both ++ and -- have a side effect which shows the essential difference between the prefix and postfix versions. For instance, in `count++`, the variable is evaluated *before* reassignment to the same variable. For `++count`, evaluation takes place *after* the assignment. Consider these examples:

```
count = 1;
printf("count = %d\n", count++);        Prints 1 before incrementing count to 2
count = 1;
printf("count = %d\n", ++count);        Increments count to 2 before printing 2
```

After execution is complete, `count` in each case is set to *2* even though **`printf`** displays different values. The difference lies in the timing of the side effect that causes `count` to change its value. This timing is crucial when using a **`while`** loop in the following ways:

```
count = 10;
while (count-- > 0)                    Loop executes 10 times
count = 10;
while (--count > 0)                    Loop executes 9 times
```

In Chapter 8, you'll acquire the skill to handle this difference of 1 that arises between the prefix and postfix operations.

6.13.2 When Things Can Go Wrong

You must not prefix or postfix a variable multiple times in function arguments or in the same expression. Enthusiastic programmers sometimes go overboard and use **printf** like this:

```
i = 11;
printf("The first 3 integers exceeding 10 are %d, %d, %d\n", i, i++, i++);
```

While one would expect the numbers 11, 12 and 13 to be printed, the results obtained on two systems were found to be different. There seems to be no definite consensus on the sequence of handling the **printf** arguments:

```
The first three integers exceeding 10 are 13, 12, 11        GCC
The first three integers exceeding 10 are 13, 13, 13        Visual Studio
```

A similar observation can be made on using ++ or -- multiple times with the same variable in an expression. The following statements are meant to calculate the sum of the squares of the first three integers:

```
int i = 3, sum;
sum = (i * i) + (--i * i) + (--i * i);
printf("%d\n", sum);                   Should it print 9 + 4 + 1 = 14?
```

We used parentheses for clarity; the precedence of the operators used in the expression don't require their use. On the author's Linux system, `sum` evaluated to 5 when 14 was expected which means that the evaluation did not start from the left.

Caution: The rules of C don't specify the order of evaluation when ++ and -- are used multiple times on the same variable in function arguments or an expression. Because the order of evaluation is implementation-dependent, you must not use these operators in these two situations.

6.14 computation2.c: USING THE ASSIGNMENT AND ++/-- OPERATORS

Program 6.6 exploits the abbreviative features of the special arithmetic operators to offer two important utilities. Using a **while** loop, the program computes (i) the sum of the first 100 integers, (ii) the result of an integer raised to a certain power. But before doing that, this three-part program demonstrates the way the assignment operators are used.

Part 1 This part uses the five assignment operators on the variable `a`. Note the automatic generation of the serial number for every line printed. Every time `sl++` is printed, the side effect increments the value of `slno`.

```
/* computation2.c: Uses the assignment, ++ and -- operators to compute
   (i) sum of first 100 integers (ii) power of a number. */

#include <stdio.h>
#define LAST_INT 100

int main(void)
{
   int a = 3, slno = 1;               /* slno prints the serial no */
   int inta = 0, total1 = 0;
   int intb = 0, base, power, total2 = 1;

   /* 1. The assignment operators */
   printf("%d. Initial value of a = %d\n", slno++, a);
   a += 5;                                        /* a is now 8, which ... */
   printf("%d. a += 5 = %d\n", slno++, a);       /* ... is confirmed here  */
   printf("%d. a /= 2 = %d\n", slno++, a /= 2);
   printf("%d. a *= 6 = %d\n", slno++, a *= 6);
   printf("%d. a -= 3 = %d\n", slno++, a -= 3);
   printf("%d. a %%= 6 = %d\n", slno++, a %= 6);

   /* 2. Sum of first 100 integers */
   while (inta < LAST_INT + 1) {
      total1 += inta;                /* Can also use total1 += inta++ ... */
      inta++;                        /* ... and eliminate this line */
   }
   printf("%d. Total of first %d integers = %d\n", slno++, LAST_INT, total1);

   /* 3. Power of a number */
   printf("%d. Enter two numbers: ", slno++);
   scanf("%d%d", &base, &power);
   while (++intb < power + 1)          /* No { and } required because ... */
      total2 *= base;                  /* ... a single statement is executed */
   printf("%d. %d to the power %d = %d\n", slno++, base, power, total2);

   return 0;
}
```

PROGRAM 6.6: **computation2.c**

```
1. Initial value of a = 3
2. a += 5 = 8
3. a /= 2 = 4
4. a *= 6 = 24
5. a -= 3 = 21
6. a %= 6 = 3
7. Total of first 100 integers = 5050
8. Enter two numbers: 2 8
9. 2 to the power 8 = 256
```

PROGRAM OUTPUT: **computation2.c**

Part 2 Unlike the technique used in Program 6.2, this program uses a **while** loop to calculate the sum of the first 100 integers. The loop repeats as long as `inta` is less than `LAST_INT + 1`. To know the sum of the first 200 integers, simply change the **#define** directive.

Part 3 This part calculates the power of a number ($base^{power}$), where `base` and `power` are integers taken from user input. Because the `*=` repeatedly multiplies `base` by itself, `total2` progressively acquires the following values: 1, `base`, `base * base`, `base * base * base`, and so on. Note that this **while** loop increments `intb` and checks its value in the same expression. Amazing power of C!

6.15 RELATIONAL OPERATORS AND EXPRESSIONS

Two of the tenets of structured programming—decision making and repetition—are implemented using relational expressions. A *relational expression* comprises two expressions connected by a *relational operator*. This expression can have only boolean values (true or false) and takes the following general form:

exp1 relop exp2 *relop signifies a relational operator*

Here, *exp1* or *exp2* can be any expression. *relop* can be any of the relational operators shown in Table 6.4 (like `>`, `<`, `==`, etc.). For instance, `x > y` is a relational expression and has the value 1 or 0 depending on whether `x` is greater than `y` (1) or not (0). Consider this code snippet:

```
int x = 5, y = 7;
printf("%d, %d\n", x > y, x < y);          Prints 0, 1
```

These two possible values are determined by the *relation* `x` has with `y`; hence the term "relational." The value `0` or `1` is of type `int`.

Takeaway: Unlike arithmetic expressions which are evaluated arithmetically, relational expressions are evaluated *logically*. An expression here can take on only true (one) or false (zero) values.

TABLE 6.4 The Relational Operators

Operator	*Expression*	*Has the Value 1 if*
`<`	`x < y`	x is less than y, 0 otherwise.
`<=`	`x <= y`	x is less than or equal to y, 0 otherwise.
`>`	`x > y`	x is greater than y, 0 otherwise.
`>=`	`x >= y`	x is greater than or equal to y, 0 otherwise.
`==`	`x == y`	x is equal to y, 0 otherwise.
`!=`	`x != y`	x is not equal to y, 0 otherwise.

6.15.1 Using the Operators

Relational expressions are used as *control expressions* with all decision making and loop constructs (Chapters 7 and 8) to determine control flow. All control expressions must be enclosed within parentheses as shown in the following examples:

Construct	*Value of Control Expression is 1 if*
`if (your_number < 11)`	`your_number` is less than 11
`while (count <= your_number)`	`count` is less than or equal to `your_number`
`while (response == 'y')`	`response` is equal to `'y'`
`if (index != 0)`	`index` is not equal to zero
`if (index = 0)`	Wrong! Sets `index` to 0

The third example shows the use of the == operator (in a **`while`** loop) to test equality. Beginners often make the mistake of using = (the assignment operator) instead of ==. For instance, the following statement is used wrongly:

```
if (index = 0)                    No test here, changes value of index
```

The expression `index = 0` is not a relational one but it still evaluates to 0 because of the assignment it makes by using a wrong operator. The compiler may flag a warning but not an error because it has no reason to think you have made a mistake even though you have.

Caution: Don't use the = operator to check for the equality of two expressions because you'll be making an assignment instead. The == is the equality operator used in C.

Note: Even though we tend to think of 1 and 0 as true and false values, respectively, the term *true* actually represents any non-zero value. Thus, the minimalist expression 5 is always true.

6.15.2 Precedence and Associativity

The relational operators have a lower precedence than the five arithmetic operators, but they rank higher than the assignment operators. They also have L-R associativity. You'll often find the = combined with the relational operators, == and !=, in this manner:

```
(x = y) == z                      Without (), y == z would be evaluated first
(x = y) != z                      Without (), y != z would be evaluated first
```

Each expression is evaluated after making an assignment followed by a relational test. Because = has a lower precedence than either == or !=, use of the parentheses ensures that y is assigned to x before x is compared to z. The following example represents a practical application of this expression:

```
(c = getchar()) != EOF
```

This expression, which owes its origin to Kernighan and Ritchie, is commonly used inside a **`while`** loop to repeatedly invoke the **`getchar`** function to read a character from a file or the terminal. Here, the return value of **`getchar`** is assigned to c, which is then checked for non-equality with a symbolic constant. This expression has the side effect of reading a character every time it is evaluated.

6.16 THE LOGICAL OPERATORS

In real-life programming, decisions can't always be made by making simple relational tests. C handles complex decision making using the following three *logical operators*:

- `&&` represents logical AND
- `||` represents logical OR
- `!` represents logical NOT

The first two operators are binary while the third one is unary. These operators are meant to be used with relational expressions even though any expression that evaluates to a value is a valid operand for them.

6.16.1 The && & || Operators

The logical operators, `&&` and `!!`, evaluate two expressions *exp1* and *exp2* individually as true or false and then use these values to evaluate the combined or compound expression as true or false. The syntax of both operators are shown below:

exp1 `&&` *exp2*	*exp1* and *exp2* evaluate to true or false
exp1 `||` *exp2*	*Ditto*

`&&` and `||` are also known as the logical AND and logical OR operators, respectively. They conform to the truth tables of the AND and OR gates, respectively (Figs. 3.1 and 3.2). Thus, the value of the compound expression is true when

- in the case of `&&`, both *exp1* and *exp2* individually evaluate to true.
- in the case of `||`, at least one of them is true.

Here's an example of a compound expression that uses the `&&` with two relational expressions as its operands:

`a > 0 && b > 0`	*A compound expression*

If `a` has the value 0 and `b` has the value 5, the resultant expression evaluates to false because the first expression is false. Now, let's replace the `&&` in the previous expression with `||`:

```
a > 0 || b > 0
```

Assuming the same values of `a` and `b`, the compound expression is true because one of the expressions is true (`b > 0`). In both cases, evaluation will begin from the left but it will stop mid-way if the outcome of the entire expression has already been determined. For the `&&`, evaluation stops if `a > 0` is false. For the `||`, evaluation stops if `a > 0` is true.

Both operators have a precedence lower than the relational operators but higher than the assignment ones (Table 6.2). Thus, no parentheses are needed to enclose the relational expressions. Between themselves, the `&&` has a higher priority than the `||`, but both have L-R associativity.

Chapters 7 and 8 fully exploit the power of the logical operators. It could still be helpful to know how they are used in a program, say, by an **`if`** statement:

```
if (a == 0 && b == 0) {
   printf("Both integers are zero\n");
   return 1;                                    Intelligent use of return value
}
```

A compound expression can include more than one logical operator. The following expression evaluates to true when all of the relational expressions also evaluate to true:

```
a <= 0 && b <= 0 && c <= 0
```

Compound expressions like the one above are useful only when we don't want to know the precise reason of failure; it's either `a`, `b` or `c`, that's all we care to know. Sometimes, you may need to make separate decisions for each of three possible outcomes. In that case, you need to use a separate **`if`** statement with each expression.

6.16.2 The ! Operator

The next logical operator we examine is the unary `!` representing the NOT operation. This operator negates the logical value of its operand. True becomes false and vice versa as is the case with the NOT gate (Fig. 3.4). For instance, the two expressions on the following lines negate each other:

```
!(a > 0)          a > 0
!(b == 1)         b == 1
```

The `!` has a higher precedence than the relational and logical operators (Table 6.2). Without the parentheses, `!a` would represent the logical negation of `a`.

Table 6.5 presents some constructs that use the `!` alongside their equivalent constructs that don't use it. The expression `!(a > 0)` signifies "`a` not greater than 0", which is the same as `(a <= 0)`. Normally, an expression without the `!` is easier to read than one with it, so you should avoid using it to the extent you can.

TABLE 6.5 Use of Relational Operators with and without !

With Logical !	*Without Logical* !
`if (!(a > 0))`	`if (a <= 0)`
`if (!(a >= 1))`	`if (a < 1)`
`while (!(a != 0))`	`while (a == 0)`
`if (a !> 0)`	Error; `!>` is an invalid operator
`if (!a == 1)`	Not an error; but not what you meant

6.17 THE CONDITIONAL OPERATOR

C supports the only ternary operator—the *conditional operator*—which uses the symbols `?` and `:`. These two symbols are used with three expressions to implement a rudimentary form of decision making. The operator uses the following formal syntax:

```
exp1 ? exp2 : exp3                              Entire line is an expression
```

exp1, *exp2* and *exp3* represent three expressions, where *exp1* is generally a relational one. The composite expression evaluates to *exp2* if *exp1* evaluates to true, or *exp3* otherwise. Here's how you can use this operator to make a simple decision:

```
rate = total > 1000 ? 0.95 : 0.90;
```

If `total` is greater than zero, then `0.95` is assigned to `rate`, otherwise `0.90` is assigned. Nothing could be simpler than that!

The ternary conditional operator has a very low precedence—just above the assignment operators. It has R-L associativity. This operator is taken up again in Section 7.14 for displaying messages using the expression property of **`printf`**.

6.18 `binary_outcomes.c`: RELATIONAL, LOGICAL AND CONDITIONAL OPERATORS AT WORK

Program 6.7 demonstrates the use of the operators that evaluate expressions to logical values. It accepts three integers as user input, and in the first two parts, conducts relational and logical tests on them. The final part features the conditional operator which uses the **`printf`** function as two operands. The program is invoked twice and the output is shown side by side.

The first two sections feature the `>`, `<` and `==` relational operators along with the `&&` and `||` logical operators. Observe that the expressions evaluate to either 0 or 1.

The third section uses the ternary operator `?:` in two ways. In the first invocation, `i4` is assigned either of two values (`50` or `100`) depending on the outcome of the expression `i1 > i2`. The same relation is used in the second invocation simply to evaluate two **`printf`** expressions. Note that both the side effect and return value of **`printf`** have been printed by another **`printf`**!

6.19 OTHER OPERATORS

We have completed our examination of the operators belonging to the well-defined families (arithmetic, relational and logical). Another family, comprising the bitwise operators, are discussed in Chapter 17. Before we draw the curtains on this chapter, let's examine two more operators, one of which has been used before and is included here for completeness.

6.19.1 The Comma (,) Operator

The comma (`,`) is a binary operator that evaluates two or more expressions from left to right and returns the value of the right-most expression. The syntax is shown below:

exp1, *exp2*
exp1, *exp2*, *exp3*, ...

The comma *operator* used here is different from the comma *separator* used to delimit variable declarations or argument lists of functions. Here, the comma operates on *exp1*, *exp2*, and so forth, and discards every expression in the list except the last one. So, the following expression

```
a = 3, b = 5
```

```
/* binary_outcomes.c: Uses operators that work with logical values. */
#include <stdio.h>

int main(void)
{
   int i1, i2, i3, i4;

   printf("Enter three integers: ");
   scanf("%d %d %d", &i1, &i2, &i3);

   /* 1. Relational Operators */
   printf("i1 = %d, i2 = %d, i3 = %d\n", i1, i2, i3);
   printf("i1 > i2 = %d\n", i1 > i2);
   printf("i2 < i3 = %d\n", i2 < i3);
   printf("i2 == i3 = %d\n", i2 == i3);

   /* 2. Logical Operators */
   printf("i1 > i2 && i2 == i3 = %d\n", i1 > i2 && i2 == i3);
   printf("i1 < i2 || i2 != i3 = %d\n", i1 < i2 || i2 != i3);

   /* 3. Conditional Operator */
   i4 = i1 > i2 ? 50 : 100;
   printf("i1 > i2 ? 50 : 100 -- has value %d\n", i4);
   printf("%d\n",
          i1 > i2 ? printf("Note i1 > i2\n") : printf("Note i1 <= i2\n"));

   return 0;
}
```

PROGRAM 6.7: **binary_outcomes.c**

```
Enter three integers: 7 3 3
i1 = 7, i2 = 3, i3 = 3
i1 > i2 = 1
i2 < i3 = 0
i2 == i3 = 1
i1 > i2 && i2 == i3 = 1
i1 < i2 || i2 != i3 = 0
i1 > i2 ? 50 : 100 -- has value 50
Note i1 > i2
13
```

```
Enter three integers: 5 7 9
i1 = 5, i2 = 7, i3 = 9
i1 > i2 = 0
i2 < i3 = 1
i2 == i3 = 0
i1 > i2 && i2 == i3 = 0
i1 < i2 || i2 != i3 = 1
i1 > i2 ? 50 : 100 -- has value 100
Note i1 <= i2
14
```

PROGRAM OUTPUT: **binary_outcomes.c**

evaluates to 5. The evaluated value may be assigned to a variable in which case the parentheses are needed to override the low precedence of the comma (Table 6.2). As the following examples indicate, without parentheses, x would be assigned the first item in the list:

```
x = (3, 5);                          x is 5
x = 3, 5, 7;                         x is 3—Wrong
x = (3, 5, printf(Hello\n"))         x is 6; side effect prints message
```

Like with the **goto** keyword, there is skepticism about the utility of this operator because C can do without it. However, the comma is useful in at least one situation where one of the expressions produces a side effect. Consider the following code that sums all integers input from the keyboard until 0 is input:

```
while (scanf("%d", &x), x > 0)                    Loop terminates when x is 0
   sum += x;
printf("Sum: %d\n", sum);                          Statement outside the loop
```

The control expression used by **while** contains a comma-operated expression. Every time the loop iterates, the comma discards the return value of **scanf**, but the side effect of **scanf** reads the next integer from the keyboard. The loop terminates when the expression `x > 0` (the right-most expression) becomes false.

6.19.2 The sizeof Operator

As we have seen on numerous occasions, this unary operator works at compile time to evaluate the size in bytes of any data type or expression. There are two possible syntaxes for this operator:

```
sizeof(data type)
sizeof expression                                  Parentheses optional here
```

Table 6.6 shows the typical values obtained when **sizeof** is used with various data types and expressions. The last two entries need to be studied carefully. Because **sizeof** has a higher precedence than the arithmetic operators, parentheses are needed to allow the evaluation of `3.142 * 4` (a `double`) to take place first.

sizeof is very useful in situations where you want a chunk of memory to be allocated, say, for 20 integers of type `int`. Rather than assume 4 as the size of `int` and ask for 80 bytes, you can make your program portable by asking for `20 * sizeof(int)` bytes instead. You'll encounter **sizeof** in Chapter 16 when dynamically allocating memory using the **malloc** and **calloc** functions.

TABLE 6.6 Determining Size of Data Types and Expressions with **sizeof**

Statement	*Prints*	*Remarks*
`printf("%d\n", sizeof(int));`	4	
`printf("%d\n", sizeof(float));`	8	
`printf("%d\n", sizeof 3.142);`	8	Real constant stored as `double`
`printf("%d\n", sizeof 'A');`	4	Character constant stored as `int`
`printf("%d\n", sizeof "Kitkat");`	7	Counts NUL character also
`printf("%d\n", sizeof 3.142 * 4);`	32	Wrong, parentheses needed even ...
`printf("%d\n", sizeof (3.142 * 4));`	8	... though operand is an expression

WHAT NEXT?

Real-life programs are seldom represented by a set of sequential statements. Programs need to make decisions and perform repetitive action. The next two chapters examine the constructs that disrupt sequential behavior and thus make C suitable for structured programming.

WHAT DID YOU LEARN?

Operators connect variables and constants to form *expressions*. They have either absolute or boolean values (0 or 1). A function that returns a value is also an expression.

Arithmetic operators include the `%` (modulo) which computes the remainder of a division of two integers. Each arithmetic operator (like + and -) can be combined with the = to form an *assignment operator* (like += and -=).

A division of two integers truncates the decimal portion. This can be prevented by using at least one operand of the floating point type.

An operand with a lower type is *implicitly* promoted to the type of the other operand. All `char` and `short` operands are automatically promoted to `int`.

The *cast* is the most powerful type-changing tool. It can *explicitly* promote or demote the type of any variable, constant or expression.

The order of evaluation of an expression is determined by operator *precedence*. When an operand is shared by two operators having the same precedence, the sequence of evaluation is determined by operator *associativity*.

The increment and decrement operators, ++ and `--`, and the assignment operators produce *side effects*, which change the operand itself.

Expressions using *relational* operators (like >) evaluate to boolean values. These values also determine the boolean value of a combined or compound expression that uses *logical* operators (like &&).

The *conditional* expression (`? :`) evaluates to the value of one of two expressions depending on the boolean value of a third expression.

The *comma* operator evaluates its operands to the value of the right-most expression. The compile-time operator, **`sizeof`**, determines the size in bytes of a data type or expression.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

6.1 The sequence `x + 5` is an expression but not `x = 5`.

6.2 A function cannot be treated as an expression even if it returns a value.

6.3 The sequence `5;` is a valid statement in C.

6.4 For an expression to be evaluated using floating-point arithmetic, at least one operand must be of the floating point type.

6.5 For evaluating the expression `5 * 10 - 12`, both precedence and associativity of the operators need to be considered.

6.6 Associativity is invoked when two operators in an expression have the same precedence.

6.7 The constants `'Z'` and `30000` take up the same amount of storage.

6.8 The expressions `15 / 9` and `15.0 / 9` have different values.

6.9 The operand of **`sizeof`** must always be enclosed within parentheses.

6.10 The parentheses in **`sizeof(int)`** are optional.

6.11 The `-` in `-5` has the same priority as the `-` in the expression `4 - 3`.

6.12 The value of the expression `2, 3, x = 5` is 5.

A2. FILL IN THE BLANKS

6.1 The * and / represent binary operators but ++ is a _____ operator.

6.2 The expression `x = 10` has the value (A) 1, (B) 0, (C) 10, (D) none of these.

6.3 The order of evaluation of an expression is determined by the _____ and _____ of the operators.

6.4 The **`printf`** function not only returns a value but also has a _____ _____.

6.5 In an expression, the `char` and `short` operands are converted to _____.

6.6 A relational expression returns either a _____ or _____ value.

6.7 The expression `k += 1` is the same as _____ and _____.

6.8 The `!=` operator negates the _____ operator.

6.9 The `?:` represents the only _____ operator in C.

6.10 For `int arr[20]`, the expression `sizeof arr / sizeof(int)` evaluates to _____.

A3. MULTIPLE-CHOICE QUESTIONS

6.1 The expressions x++ and ++x can be used interchangeably (A) always, (B) never, (C) sometimes.

6.2 If `x = 3`, the statement **`printf("%d %d %d", ++x, ++x, x--);`** displays (A) `4 4 3`, (B) `4 5 4`, (C) `4 5 5`, (D) undefined value.

6.3 If x has the value 5, the expression `x > 5` has the value (A) 5, (B) 0, (C) 1, (D) no value.

6.4 The expression `7.5 % 5` evaluates to (A) 2, (B) 2.5, (C) an illegal operation, (D) an implementation-dependent value.

6.5 For evaluating the expression `x * y - a / b`, (A) multiplication will be done first, (B) division will be done first, (C) subtraction will be done first, (D) subtraction will be done last.

6.6 The expression `1.1 * 55L + 2 / 0.75F` has the data type (A) `double`, (B) `long double`, (C) `long`, (D) `float`.

6.7 The expression `printf("Hello\n")` has the value (A) 0, (B) 6, (C) 5, (D) 7.

6.8 The expression `2 / 5 + 5 / 2` has the value (A) 2.5, (B) 2.9, (C) 0, (D) none of these.

6.9 The statement `if (x = 5)` (A) doesn't cause a compilation error, (B) assigns 5 to x, (C) doesn't cause a runtime error, (D) all of these, (E) none of these.

6.10 The expression `5 ? 6 : 7` evaluates to (A) 6, (B) 7, (C) 1, (D) 0.

A4. MIX AND MATCH

6.1 Match the operators with their types:

(A) +=, (B) `%`, (C) `&&`, (D) `<=`, (E) `--`

(1) relational, (2) assignment, (3) decrement, (4) logical, (5) arithmetic.

CONCEPT-BASED QUESTIONS

6.1 What is the significance of the the two `-` operators in the expression `x - -5`?

6.2 What are the advantages of a *cast* compared to automatic type conversion?

6.3 If `x = 9` and `y = 5`, describe three ways of evaluating the expression `x / y` without truncating the fractional part.

6.4 What is the size of the constant `1.0`? How can you make it store a smaller number of bytes?

6.5 Explain the concept of the side effect of operators and functions. Name three operators that have side effects.

6.6 Explain the difference in output of the following statements:

```
printf("%d\n", 3, 5);
printf("%d\n", (3, 5));
```

6.7 Will the following code work? Explain with reasons.

```
int x;
printf("Enter an integer: ");
scanf("%d", &x);
x % 2 ? printf("Odd") : printf("Even");
```

6.8 Explain the circumstances in which the entire expression is *not* evaluated:

(i) `a > b || c < d`

(ii) `a > b && c < d`

PROGRAMMING & DEBUGGING SKILLS

6.1 Write a program that computes and prints the quotient and remainder of a division of two integers that are accepted from the keyboard with **`scanf`**.

6.2 Write a program that accepts the dimensions of a rectangle as an integer and floating point number and prints the area and perimeter.

6.3 Using the temperature-conversion formula *C/5 = (F-32)/9*, write a program that accepts a temperature (in decimal) in Fahrenheit and prints the converted temperature in Celsius without using a cast. Modify the program to use a cast and note your observations.

6.4 Write a program that accepts an integer from the keyboard and prints `"ODD"` or `"EVEN"` depending on its value.

6.5 Write a program that accepts the time taken as hours, minutes and seconds as 3 integers and prints the total number of seconds.

6.6 Write a program to accept the file size in MB (megabytes) and the download speed in Mbps (megabits/sec) and calculate the time in seconds that would be spent to download the file. (Note it's megabytes and megabits.)

6.7 Using the formula *area = (base X height) / 2*, write a program that accepts the base and height of a triangle as integers and prints the area as a real number. (Use of the multiplier 1.0 is not permitted.)

6.8 The **`scanf`** function returns the number of items read successfully. Why doesn't the following code run properly when two integers are input?

```
int x, y, num;
if (num = scanf("%d, %d", &x, &y) != 2)
          printf("Error");
```

6.9 Write a program to swap two variables without using a third variable.

7 Control Structures—Decision Making

WHAT TO LEARN

- Significance of the *control expression* for making decisions.
- Use of the **`if-else`** construct for two-way decision making.
- Nesting of **`if-else`** constructs for multi-way decision making.
- Indentation and pairing issues with nested constructs.
- Using a *conditional expression* for simple two-way decisions.
- Usefulness of the **`switch`** construct for testing equality.
- Whether the **`goto`** statement deserves the ill-reputation that it has acquired.

7.1 DECISION-MAKING CONCEPTS

We are frequently confronted with the need to make a decision on what to do next. Decisions enable devices or applications to behave in different ways depending on the data they encounter. A decision can be two-way or multi-way, and one decision can lead to further decision making. Here are some real-life situations that involve making decisions:

- Blocking access to an ATM when a user fails to enter correct PIN in three attempts.
- Switching off the geyser when the required temperature has been attained.
- Setting the motor speed of a washing machine depending on the type of wash chosen.
- Checking the residual battery charge of a cellphone to activate a notification LED.

All programming languages support at least one construct that supports *selection*, i.e., decision making. This construct specifies what to do if one or more conditions are met, and also what to do otherwise. Complex situations are handled by chaining together a number of such constructs. As noted before, sequence, selection and repetition can solve any problem that is capable of being described unambiguously.

Execution of a decision-making construct interrupts the default sequential flow of program execution. It causes control to branch to a different point in the program from where it *should* return to resume execution from the next statement. This may not happen if this transfer of control is caused by *improper* use of the *GOTO* statement. Structured programs, therefore, invoke procedures or functions from where the return is automatic.

7.2 DECISION MAKING IN C

The C language recognizes that, depending on the nature and complexity, different situations need different decision-making tools to handle them. Consequently, C addresses selection issues with the following three constructs:

- The **`if`** statement supported by the optional **`else`** keyword.
- The **`switch`** statement.
- The conditional operator *(6.17)*.

The **`if-else`** construct, which is common to all languages, takes up most of the chapter space. The construct handles complex decisions by combining a number of simple **`if-else`** constructs. In certain situations, the **`switch`** is better suited for multi-way decision making. The conditional operator (**`?:`**) uses one-liners to handle very simple tasks. You have seen it used in Section 6.17.

7.2.1 The Control Expression

Some C constructs like the **`if-else`** and **`while`** statements use the true or false value of a *control expression* to determine control flow *(6.15.1)*. They require this expression to be enclosed by parentheses. The control expression is normally a relational or logical expression as shown by the following examples:

```
if (amount > 20000)
if (answer == 'Y')
if (age >= 60 && sex == 'F')
```

But a control expression can also be any expression that returns any value including 0. Also, because any non-zero value is true in C, the following expressions are also valid even if they are neither relational nor logical:

```
if (total)          True if total is greater than 0.
if (count++)        True if count is greater than 0 before postfixing.
if (5)              Always true.
```

When framing these expressions, you need to know the relative precedence and associativity of their operators. Although discussed before, an essential subset of these operators is presented here for quick reference (Table 7.1). Even though the `%` doesn't belong to this subset, it has been included because it appears in some of the programs in this chapter. The operators are arranged in decreasing order of their precedence.

Note: The relational operators have a higher precedence than the logical ones. Also remember that the AND operation is performed before OR.

TABLE 7.1 Operators Used for Decision Making (in order of precedence)

Operator(s)	*Significance*	*Associativity*
`!`	Logical NOT	R-L
`%`	Modulus	L-R
`<, <=, >, >=`	Relational	L-R
`==, !=`	Relational (Equality)	L-R
`&&`	Logical AND	L-R
`||`	Logical OR	L-R
`?:` (ternary operator)	Conditional	R-L

7.2.2 Compound Statement or Block

When using constructs like **`if`** and **`while`**, you'll need to address a group of statements as a single unit. This unit, which is flanked on either side by a matched pair of curly braces, is called a *compound statement*. It is also known as a *control block*, or, simply, *block*. Here's a compound statement drawn from Section 6.5:

```
if ((last_number % 2) == 1) {                    Control block begins
   printf("Not an even number\n");
   return 0;
}                                                Control block ends
```

The **`printf`** and **`return`** statements form a compound statement which is manipulated by **`if`** as a single statement. Whenever two or more statements are affected by a control expression, they *must* be enclosed in curly braces to form a compound statement. It is also permissible to use a compound statement wherever a single statement is allowed. That is sometimes done to provide clarity.

It could be helpful to keep in mind that a block changes the *scope* (i.e. visibility) of a variable declared inside it. Such a variable is not visible outside the block or in other blocks. These visibility issues will be addressed in Chapter 11.

7.3 THE `if` STATEMENT

The **`if`** statement is the most commonly used selection construct in any programming language. It has basically two forms—with and without the **`else`** keyword. The other forms discussed later are derived from these two forms. The first form (Fig. 7.1) specifies the action if *expression* evaluates to true. The syntax shows the usage for both simple and compound statements.

`if (`*expression is true*`)`
  statement`;`
	`if (`*expression is true*`) {`
	  statement1`;`
	  statement2`;`
	  ...
	`}`

FIGURE 7.1 The `if` Statement (Form 1)

Execution of this statement begins by evaluating *expression*. If this value is true, then one or more statements are executed. Curly braces are not required for a single statement but they are needed for a compound statement. Here's a simple example:

```
if (hours_left > 6)                                    if syntax requires use of parentheses
   rate = 25;                                          Next statement should be indented
printf("Refund amount = %f\n", rate * amount);
```

Here, `25` is assigned to `rate` if `hours_left` is greater than 6. Note that the **`printf`** statement is executed unconditionally and has nothing to do with the binary outcome of the control expression. The difference in indentation reveals our true intention and is consistent with what the compiler also sees.

A compound statement needs curly braces at its two ends. The following compound statement comprises three simple ones:

```
if (amount > 10000) {
   printf("You can't withdraw more than 10000 with this debit card\n");
   printf("Key in an amount not exceeding 10000\n");
   scanf("%d", &amount);                                   if ends at this ;
}
```

Note there is no semicolon following the } symbol. In the absence of curly braces, only the first **`printf`** would have been executed conditionally.

Note: When used with a single statement, **`if`** ends at the semicolon used as statement terminator. For a compound statement, **`if`** ends at the terminating **`;`** of the last statement.

Caution: All control expressions used by the decision-making and loop constructs must be enclosed within parentheses. The compiler will generate an error if you use `if count > 0` or `while count++ <= MAX_COUNT`.

7.4 `average_integers.c`: AVERAGE CALCULATING PROGRAM

Our first program (Program 7.1) calculates the average of two non-zero integers keyed in by a user, but considers only their absolute values after validation. This means that the **-** sign is stripped off from a negative integer. The task has been achieved by using one relational (<) and one logical operator (||). The program is run thrice.

The program first assigns two integer variables, `inta` and `intb`, with input from the keyboard. It then uses a logical expression as the control expression for **`if`** to find out if either of the integers is zero. It's good programming practice to first handle those situations that lead to errors. If program flow moves beyond the first **`if`** statement, it means that we have two valid integers in our hands.

Once the validation test has been cleared successfully, the next step is to use two **`if`** statements to compute the absolute values of the two integers. The simplest way to do that is to use the unary **-** operator, which simply multiplies its operand by -1. We use the `(float)` cast to compute the average of these two integers, because without it, the division would result in truncation of the fractional part.

```
/* average_integers.c: Calculates average of the absolute values of two
                       integers. Quits program if one integer is zero. */
#include <stdio.h>

int main(void)
{
   int inta, intb;
   printf("Enter two non-zero integers: ");
   scanf("%d %d", &inta, &intb);

   if (inta == 0 || intb == 0) {
      printf("At least one integer is zero\n");
      return 1;              /* Good to return nonzero for invalid entry */
   }

   /* Assigning absolute values to inta and intb */
   if (inta < 0)
      inta = -inta;          /* Using the - unary operator */
   if (intb < 0)
      intb = -intb;

   /* If we have come here, both inta and intb must be positive */
   printf("Average of absolute value of the integers = %.2f\n",
          (float) (inta + intb) / 2);
   return 0;
}
```

PROGRAM 7.1: **average_integers.c**

```
Enter two non-zero integers: 7 0                          First invocation
At least one integer is zero
```

```
Enter two non-zero integers: 5 10                         Second invocation
Average of absolute value of the integers = 7.50
```

```
Enter two non-zero integers: -13 8                        Third invocation
Average of absolute value of the integers = 10.50
```

PROGRAM OUTPUT: **average_integers.c**

This program features two **return** statements, and interestingly, the first one returns the value 1 instead of 0. Does returning 1 instead of 0 make any difference? Yes, it does but only if the *next* program takes advantage of this value. It will take a while before you work with cooperative programs, but it's good to know that this is the way **return** was designed to be used (see Inset—*How It Works*).

Note that we didn't use separate validation routines for the two integers `inta` and `intb`. Instead, we used the following logical expression:

```
if (inta == 0 || intb == 0)
```

This is acceptable for this introductory program since we don't need to know, in the event of failure, who the offender is. Sometimes, we need to pinpoint the cause of failure, in which case you need to use two relational expressions with individual **if** statements (as **if (int a == 0)** and **if (intb == 0)**), and specify separate paths for each of them.

HOW IT WORKS: Why `return 1` *instead of* `return 0`*?*

The **`return`** statement, when placed in the body of **`main`**, terminates a program and transmits a value to its *caller*. The caller in most cases is the operating system which saves this value until the next program is run. In applications comprising multiple programs that depend on one another, it is often necessary for one program to know whether a previous program has completed successful execution. The return value provides this information. On operating systems like UNIX and Linux, a return value of 0 signifies success. On these systems, a programmer uses a non-zero value with **`return`** to indicate failure!

7.5 `if-else`: TWO-WAY BRANCHING

None of the **`if`** statements of the previous program, **`average_integers.c`**, *explicitly* specifies the action to take when its control expression evaluates to false. The default action is to proceed sequentially and move on to the next statement following the **`if`** statement. The second form of the **`if`** construct permits two-way branching using the **`else`** keyword (Fig. 7.2). The statements following **`else`** are executed when the control expression fails.

```
if (expression is true)
    statement;
else
    statement;
```

```
if (expression is true) {
    statements;
        ...
}
else {
    statements;
        ...
}
```

FIGURE 7.2 The `if-else` Statement (Form 2)

Consider a store that offers a weekend discount of 10% for purchases over Rs 1000, and 5% otherwise. The code that computes the amount payable can be represented in this manner:

```
if (total > 1000)                                   total previously declared as float
   total = total * 0.90;                            Or total *= 0.90;
else
   total = total * 0.95;                            Or total *= 0.95;
printf("The total amount payable is Rs %.2f\n", total);
```

Because of proper indentation, it doesn't take any effort to know that the **`printf`** statement is not affected by the **`else`** clause and is executed unconditionally.

7.6 `leap_year_check.c`: PROGRAM TO CHECK LEAP YEAR

Let's now use both forms of the **`if`** statement in our next program (Program 7.2). This program subjects a user-input integer to a leap year check. A leap year is divisible by 4, so `year % 4` is 0 for a leap year. This program ignores the special check made for years that signify a change of century (like 1800, 1900, etc.) but it indicates when the next leap year will occur.

```
/* leap_year_check.c: Checks for leap year using the if-else structure.
                      Doesn't make the check for century. */
#include <stdio.h>
int main(void)
{
   short year, years_left;
   printf("Enter year for leap year check: ");
   scanf("%hd", &year);
   if (year < 0) {
      printf("Invalid year\n");
      return 1;
   }
   if (year % 4 == 0)                  /* No parentheses required */
      printf("Year %hd is a leap year.\n"
             "Next leap year is after 4 years.\n", year);
   else {
      years_left = 4 - year % 4;       /* No parentheses required */
      printf("Year %hd is not a leap year.\n"
             "Next leap year is %hd.\n", year, year + years_left);
   }
   return 0;
}
```

PROGRAM 7.2: **leap_year_check.c**

```
Enter year for leap year check: 2009                    First invocation
Year 2009 is not a leap year.
Next leap year is 2012.
```

```
Enter year for leap year check: 2012                    Second invocation
Year 2012 is a leap year.
Next leap year is after 4 years.
```

```
Enter year for leap year check: 1900                    Third invocation
Year 1900 is a leap year.                               This is incorrect
Next leap year is after 4 years.
```

PROGRAM OUTPUT: **leap_year_check.c**

We have not used parentheses in the expression `year % 4 == 0` because the `%` has a higher precedence than `==` (Table 7.1). The modulus operation is thus performed first. The **`else`** part contains the code for handling a non-leap year. Parentheses are left out in the expression `4 - year % 4` also because the `%` has a higher priority than `-`.

In the last two **`printf`** statements, the first argument actually comprises two concatenated strings. C allows this concatenation *(9.12.2)*, but note that the combined string is actually a single argument even if the strings are located on two physical lines.

Tip: Always look for potential errors right at the beginning of the program. It's pointless proceeding with program execution with erroneous data. Use the **`if`** statements to check for them and then use **`return`** with specific values to prematurely terminate the program.

7.7 MULTI-WAY BRANCHING WITH `if-else-if` ...

Nothing prevents us from using a second **`if`** statement (in any form) inside another **`if`** statement. When the second **`if`** is inducted in the "if" section, we have a *nested* **`if`** structure (**`if-if-else`**). When the same is done in the "else" section, we have a *ladder* (**`if-else-if`**) instead. Both nested and ladder structures belong to the domain of multi-way decision making. The nested form is discussed in Section 7.10. The ladder structure is discussed here (Fig. 7.3).

```
if (expression is true)
   statement;
else if (expression is true)
   statement;
      ...
else
   statement;
```

```
if (expression is true) {
   statements;
      ...
}
else if (expression is true) {
   statements;
      ...
}
      ...
else {
   statements;
      ...
}
```

FIGURE 7.3 The Ladder **`if-else-if`** Statement (Form 3)

The test on the control expression is carried out sequentially from the top downwards, and terminates as soon as it evaluates to true. The last **`else`** statement takes care of "the rest" and is executed only when all of the previous tests fail. C imposes no restriction on the number of **`else-if`** sections that can be used in this manner.

Let's now use this construct to compute the tariff for the 4G Internet service offered by a mobile operator. The tariff is Rs 255 for up to 1 GB, Rs 455 for up to 2 GB, Rs 755 for up to 4 GB and Rs 995 thereafter (simplified from live data). This logic is easily implemented using the **`if-else-if`** form:

```
if (usage <= 1)
   tariff = 255;
else if (usage <= 2)
   tariff = 455;
else if (usage <= 4)
   tariff = 755;
else                                        "The rest"
   tariff = 995;
```

This chained construct uses three different relational expressions that enable the variable `tariff` to have one of four possible values. When using constructs that implement multi-way branching,

you must ensure that the expressions are mutually exclusive. This means that two expressions must never evaluate to true in a single traversal of the structure.

```
/* irctc_refund.c: Computes refund amount on ticket cancellation. Negative
   value for hours_left is valid. Maximum price of ticket = Rs 10,000 */
#include <stdio.h>
int main(void)
{
   short hours_left, rate;
   float ticket_price, refund_amount;
   printf("Enter price of ticket: ");
   scanf("%f", &ticket_price);
   printf("Number of hours before train departure: ");
   scanf("%hd", &hours_left);
   if (ticket_price <= 0) {
      printf("Price can't be negative\n");
      return 1;
   }
   else if (ticket_price > 10000) {
      printf("Price can't exceed Rs 10,000.\n");
      return 1;
   }
   else if (hours_left > 48)
      rate = 0;
   else if (hours_left > 6)
      rate = 25;
   else if (hours_left > -2)            /* 2 hours after departure */
      rate = 50;
   else
      rate = 100;                       /* No refund */
   refund_amount = ticket_price * (100 - rate) / 100;
   printf("Refund amount = %.2f\n", refund_amount);
   return 0;
}
```

PROGRAM 7.3: **irctc_refund.c**

```
Enter price of ticket: 11000
Number of hours before train departure: 45
Price can't exceed Rs 10,000.
```

```
Enter price of ticket: 1000
Number of hours before train departure: 4
Refund amount = 500.00
```

```
Enter price of ticket: 1000
Number of hours before train departure: -1
Refund amount = 500.00
```

PROGRAM OUTPUT: **irctc_refund.c**

7.8 irctc_refund.c: COMPUTES TRAIN TICKET CANCELLATION CHARGES

Program 7.3 computes the ticket cancellation charges of the IRCTC railway reservation system. This charge, computed as a percentage of the total fare, depends on the number of hours left for departure of a train. The program takes the ticket price and number of hours as input and displays the refund amount. The following table shows the cancellation charges:

Cancellation Time	*Cancellation charge*
More than 48 hours before departure	0%
Between 6 and 48 hours before departure	25%
Less than 6 hours before departure and up to 2 hours *after* departure	50%
More than 2 hours *after* departure	100%

The maximum price of a ticket is set to Rs 10,000. The program uses an **if-else-if** ladder construct both to validate the input and to compute the cancellation charge. Because cancellation is also permitted up to two hours *after* departure, the variable `hours_left` turns negative in the last **else-if** section of the construct.

Refer to the example on 4G Internet tariffs that was presented prior to this program. The control expression there used the same variable name (`usage`) throughout the **if-else** construct. However, this construct in the current program uses two different variables (`ticket_price` and `hours_left`) in the control expressions. Even though we managed to do everything in one place, the first invocation demonstrates that it was not the right thing to do. The validation check on `ticket_price` should have been conducted before taking input for `hours_left`.

7.9 atm_operation.c: CHECKS PIN BEFORE DELIVERY OF CASH

Program 7.4 represents an ATM cash-dispensing application that uses the **if-else-if** ladder structure. The program validates the PIN (a code) that you input and allows withdrawal of an amount not exceeding Rs 20,000. The entire code section runs in a loop, so the user has unlimited chances to key in data. This program has a number of important features, so spend some time to study it in depth.

The program uses the construct **while (1)** to set up an infinite loop, one that never terminates. You may not know much about loops, but you surely know that the expression `(1)` is always true. The loop will thus never terminate unless some statement inside the loop forces an exit. Here, either of the **return** statements forces termination of the loop.

There are two **if-else-if** ladder structures, one nested inside the other. The "outer" one handles PIN code validation. The PIN is accepted only if it comprises four digits and matches the symbolic constant `USER_PIN`. If validation fails, the remaining statements are ignored and execution "falls through" to hit the curly brace that pairs with the opening brace of the **while** loop. The entire set of instructions are then repeated for the user to re-enter the PIN.

When the program has successfully validated the PIN, control moves on to the "inner" ladder which prompts for the amount. A value of 0 for `amount` provides an escape route for the program to terminate using **return 1;**. The program also terminates when `amount` is assigned a valid value.

```
/* atm_operation.c: Validates PIN with USER_PIN and ensures 4-digit PIN
   is input. Also limits maximum withdrawal amount to Rs 20,000 */
#define USER_PIN 7534
#include <stdio.h>
int main(void)
{
   int atm_pin, amount;
   while (1) {                                    /* Block 1 begins */
      printf("Enter PIN: ");
      scanf("%d", &atm_pin);
      if (atm_pin == 0)
         printf("PIN can't be zero.\n");
      else if (atm_pin > 9999)
         printf("PIN should not be more than 4 characters.\n");
      else if (atm_pin < 1000)
         printf("PIN should not be less than 4 characters.\n");
      else if (atm_pin != USER_PIN)
         printf("Incorrect PIN.\n");
      else {                                      /* Block 2 begins */
              printf("Enter amount to withdraw: ");
              scanf("%d", &amount);
              if (amount > 20000)
                 printf("Can't withdraw more than Rs 20000.\n");
              else if (amount == 0)
                 return 1;
              else {                              /* Block 3 begins */
                 printf("Processing transaction, take cash.\n");
                 return 0;
              }                                   /* Block 3 ends */
      }                                           /* Block 2 ends */
   }                                              /* while loop ends */
   return 0;              /* Will this statement ever be executed? */
}
```

PROGRAM 7.4: **atm_operation.c**

```
Enter PIN: 0
PIN can't be zero.
Enter PIN: 75344
PIN should not be more than 4 characters.
Enter PIN: 7535
Incorrect PIN.
Enter PIN: 7534
Enter amount to withdraw: 25000
Can't withdraw more than Rs 20,000.
Enter PIN: 7534
Enter amount to withdraw: 15000
Processing transaction, take cash.
```

PROGRAM OUTPUT: **atm_operation.c**

Note: The **return 0;** statement at the bottom of the program is never executed because the program exits through the paths provided by the other **return** statements placed in the inner ladder. This **return** statement placed at the bottom should be removed.

7.10 MULTI-WAY BRANCHING WITH NESTED if (if-if-else)

We have seen the induction of an **if-else** section in the **else** part of the main **if** statement. When the same section is inducted in the main **if** part itself (Fig. 7.4), we have a *nested* **if** (**if-if-else**) structure. Like with the ladder, this is not a separate feature of the language but is derived from the **if** syntax.

```
if (expression is true)
   if (expression is true)
      if (expression is true)
         statement;
            ...
      else
         statement;
   else
      statement;
else
   statement;
```

FIGURE 7.4 The Nested **if-if-else** Statement (Form 4)

The figure shows the indentation you must adopt when using nested **if** structures. Without proper indentation, it is easy to visually pair an **if** with the wrong **else**. The pairing scheme is quite simple: The innermost **if** pairs with the innermost **else**, the immediate outer **if** pairs with the immediate outer **else**, and so on until the outermost **if** pairs with the outermost **else**. The symmetry breaks down when you drop an **else** for one of the **if**s, but more of that soon.

Let's look at a simple example that demonstrates the usefulness of this derived construct. Consider the need to print the sum of three positive integers obtained from user input, but only after validation. Here's the code snippet that handles the logic:

```
if (a > 0)
   if (b > 0)
      if (c > 0)

         /* Cleared the validity check */
         printf("Sum of three integers = %d\n", a + b + c);
      else
         printf("c is not a positive integer\n");
   else
      printf("b is not a positive integer\n");
else
   printf("a is not a positive integer\n");
```

When code is presented in this way, there should be no problem in pairing an **if** with its **else**. The first three **if** statements represent the AND condition (`a > 0 && b > 0 && c > 0`), so the first **printf** signifies the action to take when all three expressions evaluate to true. The other **printf** statements are associated with the **else** clauses and they tell us what happens when each of the variables is less than or equal to zero.

In some cases, we need to drop some of the **else** keywords, or even all of them. The latter is easy to handle, but we have to handle the other situations with care. Before we do that, let's have a look at a program which contains a symmetric nested **if** structure.

7.11 right_angle_check.c: PROGRAM TO CHECK PYTHAGORAS' THEOREM

Our knowledge of geometry tells us that the square of the hypotenuse of a right-angled triangle is equal to the sum of the squares of the other two sides. This means that given three numbers `a`, `b` and `c`, if `a * a + b * b` is equal to `c * c`, then a right-angled triangle can be formed with these three sides, where the largest side represents the hypotenuse.

Program 7.5 runs an infinite **while** loop to accept three integers (`a`, `b` and `c`) from the keyboard. It allows a user multiple chances to key in non-zero values, and then tries all possible combinations of `a`, `b` and `c` to see if one of them fits the formula. It's good to know how nested **if** constructs work even though a better and more intuitive approach would be to use the **if-else-if** ladder.

The program first validates the integers for non-zero values; it terminates when all of them are zero. After successful validation, control moves to the nested construct comprising four **if** statements. The first **if** (the outermost one) allows only positive integers to pass through. If this test fails, the matching **else** at the bottom uses **printf** to display a non-specific error message. Because this entire construct runs in a loop, control then moves up to the beginning of **while** to accept the next set of integers.

Once the three integers pass the second validation test, the program tries out three possible combinations (`a-b-c`, `a-c-b` and `b-c-a`) that will *not* fit the theorem. In other words, if all three relational expressions evaluate to true, then a right-angled triangle *cannot* be formed with any combination of these integers. If one of the expressions fails, its corresponding **else** section uses **printf** to tell us that a right-angled triangle *can* be formed, and it also tells us what the hypotenuse is.

7.12 PAIRING ISSUES WITH if-if-else NESTED CONSTRUCTS

We didn't face any pairing problems in the previous program (Program 7.5) because the nested **if** construct had an equal number of **if** and **else** clauses. It was easy to see which **else** paired with which **if**. Sometimes, program logic may not need one or more **else** clauses. A small code fragment having an **else** missing is shown in Figure 7.5 with both misleading and correct indentation.

The indentation of the **else** part in the form shown on the left is deceptive. It gives the impression that the **else** is paired with the first **if**, which it is not. The form on the right correctly shows the **else** paired with the inner **if**. Even though the compiler doesn't look at indentation, we can't afford to ignore it.

```
/* right_angle_check.c: Uses nested if statements. Doesn't catch
   exact cause of failure. Formula used: a*a + b*b = c*c */
#include <stdio.h>
int main(void)
{
   short a, b, c;
   while (1) {
      printf("Enter three integers a b c: ");
      scanf("%hd %hd %hd", &a, &b, &c);
      if (a == 0 && b == 0 && c == 0) {
         printf("All values zero. Quitting ...\n");
         return 1;
      }
      /* RAT represents right-angled triangle */
      if (a > 0 && b > 0 && c > 0)
         if (a * a + b * b != c * c)         /* Whether RAT can't be formed */
            if (a * a + c * c != b * b)      /* Ditto */
               if (b * b + c * c != a * a)   /* Ditto */
                  printf("RAT not possible.\n");
               else
                  printf("RAT with %hd as hypotenuse.\n", a);
            else
               printf("RAT with %hd as hypotenuse.\n", b);
         else
            printf("RAT with %hd as hypotenuse.\n", c);
      else
         printf("At least one input is invalid.\n");
   }
}
```

PROGRAM 7.5: **right_angle_check.c**

```
Enter three integers a b c: 5 0 9
At least one input is invalid.
Enter three integers a b c: 3 5 4
RAT with 5 as hypotenuse.
Enter three integers a b c: 5 2 6
RAT not possible.
Enter three integers a b c: 6 10 8
RAT with 10 as hypotenuse.
Enter three integers a b c: 0 0 0
All values zero. Quitting ...
```

PROGRAM OUTPUT: **right_angle_check.c**

Misleading Indentation	*Correct Indentation*

`if (a > 0)`
`  if (b > 0)`
`  valid_int = 'y';`
`else`
`  valid_int = 'n';`
	`if (a > 0)`
	`  if (b > 0)`
	`  valid_int = 'y';`
	`  else`
	`  valid_int = 'n';`

FIGURE 7.5 Code with a Missing `else`

But what if you actually wanted an **`else`** part for the first **`if`** and not for the second? There are two solutions to this problem. One is to use curly braces to eliminate ambiguity. The other is to use a dummy **`else`** containing a solitary semicolon. The **`;`** placed by itself on a line signifies a *null* statement—one that does nothing. The two solutions are placed side-by-side in Figure 7.6 for you to choose the one you prefer. The curly brace solution is the one that is preferred by most.

Null Command Solution	*Curly Brace Solution*

`if (a > 0)`
`  if (b > 0)`
`  valid_int = 'y';`
`  else`
`  ;`
`else`
`  printf("a is <= 0\n");`
	`if (a > 0) {`
	`  if (b > 0)`
	`  valid_int = 'y';`
	`}`
	`else`
	`  printf("a is <= 0\n");`

FIGURE 7.6 Two Solutions for Missing `else`

7.13 `leap_year_check2.c`: PROGRAM USING THE `if-if-else` STRUCTURE

Let's conclude our discussions on the **`if`** statement by improving a previous leap year checking program. The revised version (Program 7.6) includes the special check for the "century" years (like 1900, 2000, etc.). We use a symmetric nested **`if-if-else`** structure where there is no **`else`** missing. The use of the variable `is_leap_year` should be an eye-opener because of the remarkable way it has been used.

You are aware that all years that signify a change of century are not leap years even though they are all divisible by 4. Years like 1900 and 2000 have to be divisible by 400, which means that 2000 is a leap year but 1900 is not. To identify a leap year, we must follow this course of action:

1. Check whether the number is divisible by 4. If it is not, the number is not a leap year and the program terminates.
2. If the previous check succeeds, divide the number by 100 to determine whether the year represents a change of century. If it is not, then the number represents a leap year and the program terminates.
3. If we have come here, it means that we are dealing with a century year. Now check whether the number is divisible by 400. If it is, the number is a leap year and the program terminates.
4. If we have come here, the number doesn't represent a leap year.

```
/* leap_year_check2.c: Checks for leap year using nested ifs.
                       This one makes the check for century. */
#include <stdio.h>
int main(void)
{
   short year, years_left;
   char is_leap_year = 'n';

   printf("Enter year for leap year check: ");
   scanf("%hd", &year);

   if (year % 4 == 0)                /* Includes 1900, 2000, 2008 */
      if (year % 100 == 0)           /* Example year 1900 */
         if (year % 400 == 0)        /* Example year 2000 */
            is_leap_year = 'y';
         else                        /* Example year 1900 */
            years_left = 4;
      else                           /* Example year 2008 */
         is_leap_year = 'y';
   else                              /* Not a leap year */
      years_left = 4 - year % 4;

   /* We now know whether it is a leap year or not */
   if (is_leap_year == 'y')
      printf("%d is a leap year.\n", year);
   else
      printf("%d is not a leap year. Next leap year: %d.\n",
             year, year + years_left);

   return 0;
}
```

PROGRAM 7.6: **leap_year_check2.c**

```
Enter year for leap year check: 2016
2016 is a leap year.
```

```
Enter year for leap year check: 2013
2013 is not a leap year. Next leap year: 2016.
```

```
Enter year for leap year check: 1900
1900 is not a leap year. Next leap year: 1904.
```

```
Enter year for leap year check: 2000
2000 is a leap year.
```

PROGRAM OUTPUT: **leap_year_check2.c**

The preceding pseudo-code is implemented by using a set of three nested **if** statements. Each time they detect a leap year, the char variable is_leap_year is set to 'y'. For non-leap years, the variable years_left is set to the number of years left for the next leap year. Once the final status of is_leap_year is known, a simple **if-else** construct is employed to output a suitable message.

C doesn't support a boolean variable but the variable is_leap_year is used as one. It is set to 'y' at two locations and to 'n' at two others. After all checks on year have been completed, we know what the value of is_leap_year is. We can then make a decision *from a single location*. Using a variable as a flag in this way represents a classic programming technique of avoiding code redundancy. Note that the variable is_leap_year has been suitably named to indicate the type of value it holds.

Tip: If your program has multiple checks that require a common action to be taken, rather than take action at every check point, simply set a flag there. After all checks have been made, test this flag and then take the necessary action. This technique not only reduces the size of your code but also makes maintenance easier because if you later decide to change the action, it's only a single location that needs this change.

HOW IT WORKS: How to Use a Logical Expression to Solve the Leap Year Problem

We can solve the leap year problem using a logical expression. A year is a leap year if the following compound condition is met:

Is the year (divisible by 4 AND not divisible by 100) OR divisible by 400?

The parentheses have been provided for ease of understanding. Here's the code that does the leap-year job in one shot:

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
   printf("%d is a leap year\n", year);
else
   printf("%d is a not leap year\n", year);
```

Look up Table 7.1 for the precedence of the %, ==, &&, != and || operators to convince yourself that no parentheses are required for enclosing the relational expressions in the compound logical expression.

7.14 THE CONDITIONAL EXPRESSION (?:)

The conditional expression was touched upon briefly in Section 6.17. This construct uses two symbols (? and :) as a ternary operator. The operator acts on three expressions using the following formal and functional syntaxes:

exp1 ? *exp2* : *exp3* — *Entire line is an expression*

condition ? *value1* : *value2*

The combination of the three expressions itself constitutes a larger expression—the *conditional expression*. This expression has the value *exp2* if *exp1* evaluates to true, and *exp3* otherwise. Even if this construct has been discussed in Chapter 6 as a special form of expression, it is also important for decision making. In fact, this operator easily replaces those **if-else** statements that merely set a variable. Consider the following **if** statement which is shown side-by-side with the ternary version:

if-else *Version*	*Equivalent Conditional Expression*

`if (total > 1000)`
`  rate = 0.90;`
`else`
`  rate = 0.95;`
	`rate = total > 1000 ? 0.90 : 0.95;`

Using a conditional expression, the leap year problem becomes even shorter. Simply set the variable `is_leap_year` to `'y'` or `'n'` depending on the value of a logical expression. Use parentheses if they help to provide clarity:

```
char is_leap_year;
is_leap_year = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
               ? 'y' : 'n';
```

Now, **printf** and **scanf** are also expressions because both return a value, so can we use **printf** in this way?

```
count = inta < 0 ? printf("Invalid value\n") : printf("Valid integer\n");
```

Yes, this is a valid statement. In the process of evaluation of **printf**, the side effect (printing) also shows up. The variable `count` stores the return value of **printf**, i.e., the number of characters printed. Note that the **;** at the end is the terminator of the assignment statement. The first **printf** doesn't have a terminator, so why should the second **printf** have one?

7.15 THE switch STATEMENT

We use the equality test so often that C has a special construct to handle it. It's the **switch** statement which implements multi-way branching with a compact and structured construct. If you need to match 10 integer values for making 10 different decisions, use **switch** whose syntax is shown in Figure 7.7.

```
switch (exp) {
   case value1 : statements1;
                 break;
   case value2 : statements2;
                 break;
          ...
       default : statements;
}
```

FIGURE 7.7 Syntax of **switch** Statement

switch first evaluates *exp* which must be an integer or character variable or constant. It next tries to match *exp* with *value1*. If the match succeeds, **switch** executes one or more statements represented by *statements1*. If the match fails, **switch** attempts to match *exp* with *value2* and so on until control moves to the keyword **default** (if present). This option is thus invoked when all previous matching operations fail. For this purpose, **switch** is assisted by three keywords—**case**, **break** and **default**. The entire set of options and their associated action are enclosed in curly braces.

Every **case** represents an option that specifies the statements to be executed in case matching is successful. In most cases, a **case** is followed by **break**. When it is encountered, further matching is halted and control moves past the end of **switch** to the statement following the }. In the absence of **break**, however, control "falls through" to the next **case** option for the next **break**, or past the end of **switch** if it doesn't find one. Program 7.9 clearly shows us why **switch** was designed to behave in this unusual manner.

Here's a simple code fragment that shows the working of **switch**. It validates a user-input integer for the values 1 and 2:

```
printf("Enter a 1 or 2: ");
scanf("%d", &response);
switch (response) {
   case 1: printf("You entered 1\n");
           break;
   case 2: printf("You entered 2\n");
           break;
  default: printf("Invalid option\n");
}
```

No break required ...
... will break anyway

The **default** keyword (if present) doesn't need a **break** since control will anyway break out of the construct at the point where it could have occurred. However, **switch** operates with the following restrictions:

- *exp* can be any expression as long as it evaluates to an integer or character constant like `1`, `2` or `'y'`.
- The labels *value1*, *value2*, etc. can only be integer or character constants or constant expressions (like `3 + 5`). Floating point values and variables are not permitted here.
- The label is always followed by a colon, and the statements associated with it must be terminated by semicolons.
- The **default** label is optional, but if present, is usually placed as the last option (not mandatory though).
- Two or more labels can have one **break** statement. This feature gives **switch** the power of the logical OR operator.
- It is permissible to have an empty **case** (one without any statement).

Can a relational expression be used as *exp*? Sure it can, but the evaluation would yield only one of two values. You can use **switch** to match the labels `0` and `1`, but then **if** is ideally suited for this task. Program 7.9 uses a relational expression for *exp*.

7.16 calculator.c: A BASIC CALCULATOR PROGRAM USING switch

Program 7.7 simulates a rudimentary calculator that performs the basic arithmetic operations with two operands. These operands are keyed in as integer or floating point numbers. The operator is selected from a list. The entire sequence runs in a loop which can be terminated only by pressing the interrupt key (usually, *[Ctrl-c]*).

```
/* calculator.c: A rudimentary calculator using switch-case.
   Truncates operands to integers before using the % operator. */

#include <stdio.h>

int main(void)
{
   int operator;
   float left_operand, right_operand;

   while (1) {                          /* Infinite loop */
      printf("Enter left operand: ");
      scanf("%f", &left_operand);
      printf("Enter right operand: ");
      scanf("%f", &right_operand);
      printf("1. '+' 2. '-' 3. '*' 4. '/' 5. '%%'\n");
      printf("Enter a valid number for operator: ");
      scanf("%d", &operator);

      switch (operator) {
        case 1: printf("%f\n", left_operand + right_operand);
                break;
        case 2: printf("%f\n", left_operand - right_operand);
                break;
        case 3: printf("%f\n", left_operand * right_operand);
                break;
        case 4: printf("%f\n", left_operand / right_operand);
                break;
        case 5: printf("%d\n", (int) left_operand % (int) right_operand);
                break;
       default: printf("Illegal operator\n");
      }
   }
   return 0;
}
```

PROGRAM 7.7: **calculator.c**

```
Enter left operand: 5
Enter right operand: 8
1. '+' 2. '-' 3. '*' 4. '/' 5. '%'
```

```
Enter a valid number for operator: 3
40.000000

Enter left operand: 10.5
Enter right operand: 20.3
1. '+' 2. '-' 3. '*' 4. '/' 5. '%'
Enter a valid number for operator: 1
30.799999

Enter left operand: 20
Enter right operand: 40
1. '+' 2. '-' 3. '*' 4. '/' 5. '%'
Enter a valid number for operator: 6
Illegal operator
[Ctrl-c]                                    Terminates program abnormally
```

PROGRAM OUTPUT: **calculator.c**

Three **scanf** statements make the operator and its operands available to the **switch** construct. Each **case** option, which is dedicated to one operator, prints the result of the operation. The **break** in the first five **case** options ensure that control doesn't fall through once a match is found. Note that casts are needed for the modulus operator which works only with integer operands. The program is flawed; there should have been a separate option (say, 0 or 9) that would have allowed the program to exit gracefully using a **return** statement.

7.17 mobile_tariffs.c: PROGRAM TO COMPUTE CHARGES FOR 4G SERVICES

The next program (Program 7.8) lets you evaluate the 4G Internet plans offered by a mobile operator. The data is taken from a code fragment presented in Section 7.7. The program offers a menu of four plans to choose from, and then uses **switch** to assign the data offered (in GB) and plan value to two variables. It then displays the per-unit tariff for the selected plan. The program runs in an infinite loop which terminates when the **return** statement is encountered.

```
/* mobile_tariffs.c: Computes the cost per GB of data consumed from among
   four 4G Internet plans. Uses a multi-line printf to display a menu. */
#include <stdio.h>
int main(void)
{
   short option, data_limit, tariff;

   while (1) {
      printf("\nChoose from the following 4G plans:\n\n"
             "\t1. 1 GB for Rs 255\n \t2. 2 GB for Rs 455\n"
             "\t3. 4 GB for Rs 755\n \t4. 10 GB for Rs 995\n"
             "\t9. Exit"
             "\nEnter choice: ");
      scanf("%hd", &option);
```

```
      switch (option) {
         case 1: data_limit = 1; tariff = 255;
                 break;
         case 2: data_limit = 2; tariff = 455;
                 break;
         case 3: data_limit = 4; tariff = 755;
                 break;
         case 4: data_limit = 10; tariff = 995;
                 break;
         case 9: printf("Quitting ...\n");
                 return 0;
        default: printf("Invalid option\n");
      }

      if (option >= 1 && option <= 4) {
         printf("You chose %hd GB for Rs %hd.\n", data_limit, tariff);
         printf("Cost per GB = Rs %.2f\n", (float) tariff / data_limit);
      }
   }                            /* Closes while */
}
```

PROGRAM 7.8: **mobile_tariffs.c**

```
Choose from the following 4G plans:

   1. 1 GB for Rs 255
   2. 2 GB for Rs 455
   3. 4 GB for Rs 755
   4. 10 GB for Rs 995
   9. Exit
Enter choice: 4
You chose 10 GB for Rs 995.
Cost per GB = Rs 99.50

Choose from the following 4G plans:

   1. 1 GB for Rs 255
   2. 2 GB for Rs 455
   3. 4 GB for Rs 755
   4. 10 GB for Rs 995
   9. Exit
Enter choice: 9
Quitting ...
```

PROGRAM OUTPUT: **mobile_tariffs.c**

Observe how a multi-line menu of five options is created by a single **printf** statement with the aid of the \t and \n escape sequences. The user's choice is used by **switch** to set the variables data_limit and tariff. This program terminates gracefully when the user selects 9.

7.18 date_validation.c: A PROGRAM TO VALIDATE A DATE

Program 7.9 uses two **switch** constructs to validate a date that is input in *dd/mm/yyyy* format. The first **switch** validates the month and the second **switch** validates the day. The program has two interesting features—the omission of **break** in some **case** options, and the **continue** statement inside **switch** even though **continue** is not a part of the **switch** construct.

```
/* date_validation.c: Validates a date that is input in dd/mm/yyyy format.
   Uses continue inside switch and drops break from some labels. */

#include <stdio.h>

int main(void)
{
   short day, month, year, max_days;

   while (1) {
      printf("Enter date in dd/mm/yyyy format: ");
      scanf("%hd/%hd/%hd", &day, &month, &year);

      /* Month validated & max_days set here */
      switch (month) {
         case 2  : max_days = (year % 4 == 0) &&
                   (year % 100 != 0 || year % 400 == 0) ? 29 : 28;
                   break;
         case 4  :                 /* Fall-through from case 4 to case 11 */
         case 6  :
         case 9  :
         case 11 : max_days = 30;  /* Applies to case 4, 6, 9 and 11 */
                   break;
         default : if (month > 12) {
                      printf("Illegal month\n");
                      continue;
                   }
                   max_days = 31;  /* For the remaining 7 months */
      }

      /* Day validated here */
      switch (day > max_days) {    /* Relational expression used here */
         case 1 : printf("Illegal number of days\n");
                  continue;
         case 0 : printf("Valid date\n");
      }
      return 0;                    /* Program terminates if date valid */
   }                               /* Closes while loop*/
}                                  /* Closes main */
```

PROGRAM 7.9: **date_validation.c**

```
Enter date in dd/mm/yyyy format: 31/06/2016
Illegal number of days

Enter date in dd/mm/yyyy format: 30/13/2015
Illegal month

Enter date in dd/mm/yyyy format: 29/02/1900
Illegal number of days
Enter date in dd/mm/yyyy format: 29/02/2000

Valid date
```

PROGRAM OUTPUT: **date_validation.c**

The program may look daunting but it is actually quite simple. It first validates the month after extracting it from the input saved by **scanf**. The first **switch** divides the 12 months into three categories: February, the four 30-day months and the seven 31-day months. February is first checked for a leap year using a logical expression that has been discussed before.

Next, the 30-day months are handled by four **case** options, three of which specify no action and have no **break**. For these three options, control simply falls through until it encounters **break** in the **case** labeled 11. The assignment `max_days = 30` that is made here applies to the previous three options also. This would not have been possible had **switch** not supported the default fall-through behavior on a missing **break**.

The remaining seven 31-day months are handled by the **default** label which contains the first of two **continue** statements. Unlike **break**, **continue** is not part of the **switch** syntax. It is used in all loops to restart the next *iteration* (term used to describe the traversal of all statements of a loop) from the top of the loop. In this program, **continue** simply restarts the **while** loop. `max_days` can now be safely set to 31.

The second **switch** simply checks whether the day that was input exceeds `max_days` or not. A **continue** is found here as well to give the user another chance in case an invalid day was input. The program terminates normally through **return 0;** when a valid date is keyed in.

Takeaway: Both **break** and **continue** terminate a **switch** except that **continue** works only when **switch** is enclosed by a loop. **break** moves control to the statement following the **switch** terminator (the closing control brace), while **continue** moves control to the top of the loop to begin the next iteration.

7.19 THE "INFINITELY ABUSABLE" goto STATEMENT

Both **break** and **continue** can be considered as restricted versions of the *GOTO* statement offered by languages like Fortran and BASIC. However, C also supports a **goto** statement that lets control branch to a different point in the program *without returning to the point in the program from where it was executed.* A lot of negative hype has been built around the statement ever since Kernighan and Ritchie suggested that this "infinitely abusable" construct should be used "sparingly, if at all." Before we learn our own lessons, let's consider a small program (Program 7.10) that uses **goto**.

```
/* goto.c: Demonstrates the use of goto. */
#include <stdio.h>
int main(void)
{
   float ticket_price, rate = 50;
   printf("Enter price of ticket: ");
   scanf("%f", &ticket_price);

   if (ticket_price <= 0 || ticket_price > 10000)
      goto exit_from_here;
   else
      rate = 100;

   printf("Rate = %.2f\n", rate);
   return 0;

exit_from_here:
   printf("Invalid ticket price\n");
   return 1;
}
```

PROGRAM 7.10: **goto.c**

```
Enter price of ticket: 5000
Rate = 100.00
Enter price of ticket: 12000
Invalid ticket price
```

PROGRAM OUTPUT: **goto.c**

In this program, the **goto** statement moves control to the label `exit_from_here`. This label can be placed anywhere in the program but is often found at the end. When the validation on `ticket_price` fails, control branches to that label and executes all statements found below it. There can be another **goto** here as well, but placing one here would be an abuse of the facility.

It is true that the task performed by **goto** can be achieved by other means. But a **goto** helps in taking control out of a deeply nested loop, i.e., when one loop is enclosed by multiple outer loops. The **break** statement can't do that because it terminates only the current loop. However, the use of **goto** should be reserved only for the rarest of rare cases.

Tip: You'll sometimes find that multiple code sections require some cleanup to be done (like closing of files) before termination of a program. This cleanup job could be performed at a single location by using **goto** to reach that location. This analogy is similar to the one used to justify the use of flags in a program *(7.6)*.

WHAT NEXT?

Decision making alone won't solve all problems. We also need to use the mechanism that allows one statement to be executed repeatedly, creating a different effect each time it is executed. You had a glimpse of the infinite loop in this chapter. The next chapter makes a detailed examination of all loops—finite and infinite.

WHAT DID YOU LEARN?

Decisions are made in C by evaluating a *control expression* that is used by the **`if`** statement and all loops. A control expression is generally a relational or logical expression that returns one of two values (true or false).

A single **`if-else`** structure is used for two-way decision making, but a number of them can be chained to form *nested* or *ladder* structures that can make multi-way decisions.

Nested and ladder structures can be hard to interpret if an **`if`** is not aligned properly with its corresponding **`else`**. A missing **`else`** may need to be explicitly shown with a *null* command or extra curly braces.

The conditional expression using the **`?:`** ternary operator can be used to handle simple **`if-else`** problems.

The **`switch`** statement is a compact structure used only for equality tests. Every **`case`** option needs a **`break`** to prevent "fall-through" behavior. In some cases, this behavior is needed.

The **`goto`** statement causes control to branch to a different point in the program from where it doesn't return automatically. It may be used only in rare situations.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

7.1 A decision-making construct disturbs the sequential flow of a program.

7.2 The statement **`if (1)`** is not a valid C construct.

7.3 An **`if`** statement need not have an **`else`** clause.

7.4 The control expression of the **`if`** statement cannot be a function.

7.5 Every **`if`** construct can be replaced with **`switch`**.

7.6 A variable declared inside a control block is not visible outside the block.

7.7 Two floating point numbers may not compare equal even if they appear to be so.

7.8 It doesn't matter whether you use **`return 0;`** or **`return 1;`** to terminate a program.

7.9 Multiple **case** options in **switch** can have a single **break**.

7.10 The keyword **default** must be used as the last option of a **switch** construct.

A2. FILL IN THE BLANKS

7.1 The sequence (x > 3) in **if (x > 3)** is known as a _____ _____.

7.2 A compound statement must be enclosed by _____ _____.

7.3 A solitary **;** placed on a line signifies a _____ statement.

7.4 The _____ keyword prevents control from falling through in a **switch** construct.

7.5 The expression matched by **switch** must evaluate to an _____ value.

7.6 The _____ statement doesn't return to the place from where it was executed.

7.7 A conditional expression comprises a _____ operator that uses the symbols _____ and _____.

A3. MULTIPLE-CHOICE QUESTIONS

7.1 Of the two operators, && and ||, the && has a (A) higher priority, (B) lower priority, (C) same priority.

7.2 Use of curly braces ({}) for enclosing the body of the **if** statement is (A) optional, (B) recommended, (C) compulsory when executing at least two statements, (D) compulsory in a nested or ladder **if** structure, (E) C and D.

7.3 Consider the following code:

```
int x = 5;
if (x++ == 10)
   x = 0;
else
   ++x;
```

The final value of x is (A) 5, (B) 6, (C) 7, (D) implementation-dependent.

7.4 Consider the following code:

```
int x = 5;
if (x = 6)
   x = 0;
else
   x = 10;
```

The value of x is (A) 10, (B) 0, (C) 5, (D) 6.

7.5 Select the odd operator out: (A) &&, (B) ||, (C) !=, (D) !.

7.6 Consider the following code:

```
int x = 5;
x = x == 5 ? 1 : 10;
```

The value of x is (A) 1, (B) 5, (C) 10, (D) implementation-dependent.

CONCEPT-BASED QUESTIONS

7.1 Why doesn't the construct **`if (c = 'Y')`** work correctly?

7.2 Of the following two forms of the **`if`** statement, when will you use one in preference to the other?

```
if (x > 0)
  if (y > 0)
     if( z > 0)
```

```
if (x > 0 && y > 0 && z > 0)
```

7.3 When is the use of **`switch`** preferred to the **`if`** statement? Can **`switch`** handle all decision-making situations?

7.4 Why is the **`return`** statement often used with different values in an **`if-else`** or **`switch`** construct?

PROGRAMMING & DEBUGGING SKILLS

7.1 Locate the errors in the following code segment:

```
if x > 5
   printf("Number must not exceed 5\n");
   return 1;
else
   printf("Number is valid\n");
```

7.2 What is wrong with the following code segment?

```
if (x > 0) || (y > 0)
   printf("At least one number is positive\n");
else if (x <= 0) && (y <=0)
   printf("Both numbers are zero or negative\n");
```

7.3 Write a program using **`if`** that accepts three integers from the keyboard and prints the maximum and minimum values found.

7.4 Write a program using **`if`** that accepts a user-input floating point number and prints (i) the largest integer that is smaller than the number, (ii) the smallest integer that is greater than the number.

7.5 The Olympic Games have been held every four years since 1896 except for the years between 1940 and 1947. Write a program that accepts an integer and determines whether it represents an "Olympic year."

7.6 A power utility charges the following rates:

Units	Rate/Unit
First 25 units	Rs 4.89
Next 35 units	Rs 5.40
Next 40 units	Rs 6.41
Beyond 100 units	Rs 7.18

Write a program that accepts the number of units consumed and prints the total charges payable.

7.7 Write a program that accepts a character from the keyboard and prints whether the character is alphabetic, numeric or neither. The program should also print whether the character is lower- or uppercase. (HINT: Use the `%c` specifier of **`scanf`**.)

7.8 Write a program that accepts an integer from the user and prints "Odd" or "Even" without using the **`if`** and **`switch`** statements.

7.9 Consider the following rates charged by a mobile operator for data consumption in *integral* units of GB:

Data (GB)	Rate
1 GB	Rs 148
2 GB	Rs 255
3 GB	Rs 355
4 to 6 GB	Rs 455
7 GB onwards	Rs 700

Write a program using **`switch`** that accepts an integer from the keyboard and prints the corresponding rate after validation.

7.10 Detect the flaws in the following code segment:

```
float x = 5.5;
switch (x) {
    case 5.5: printf("Number is 5.5\n");
     default: printf("Some other number\n");
}
```

7.11 Write a program using **`switch`** that accepts an integer between 1 and 7 and prints whether the number represents a weekend (Saturday or Sunday) or not (Sunday = 1).

7.12 Write a program using **`switch`** that checks a user-input integer representing the month number and prints the number of days in that month. The program must combine multiple **`case`** options wherever possible.

7.13 Write a program using **`switch`** that checks a user-input character for a vowel or consonant and prints a suitable message. The program must first check whether the entered character is alphabetic (lower or upper).

8 Control Structures—Loops

WHAT TO LEARN

- Principles of entry and exit in loops.
- Concept of the *key variable* used in a control expression and loop body.
- Working of the **`while`** construct as an entry-controlled loop.
- Working of the **`do-while`** construct as an exit-controlled loop.
- Significance of three expressions for controlling iterations in a **`for`** loop.
- *Nesting* of all loops.
- Interrupting loop processing with **`break`** and **`continue`**.

8.1 LOOPING BASICS

Many activities that we perform daily are repetitive in nature. Repetition makes us breathe, moves automobiles and drives production lines. In most cases, the repetition framework also includes the mechanism to halt itself. Repetition also extends to areas that computer programs handle with ease. Consider the following tasks that involve repetition:

- Computation of average—requires repeated addition.
- Computation of the power of a number—requires repeated multiplication.
- Initialization of an array—requires repeated assignment.
- Decimal-to-binary conversion—requires repeated division.
- Drawing a line—requires repeated printing of a character.

In addition to decision making, a language must be able to repeatedly execute a a set of statements. Constructs that do that are known as *loops*, and C offers three of them—**`while`**, **`do-while`** and **`for`**. Before we get down to the specifics, we need to know in generic terms how a loop works. Figure 8.1 shows the working of a generic loop which here prints the entire English alphabet in uppercase.

1. Initialize key variable (x) outside loop.
   (say, `x = 65`)
2. Test control expression at loop beginning.
   (Is `x <= 91`?)
3. If answer to 2 is yes, perform steps 4 to 6.

   Beginning of Loop

   4. Execute loop body to print one letter.
      (`printf("%c ", x)`)
   5. Assign the next higher value to key variable.
      (`x = x + 1`)
   6. Go back to 2 to re-test control expression.

   End of loop

7. If answer to 2 is no (i.e. `x > 91`), continue execution after loop
   (say, `printf("\nJob over\n")`).

FIGURE 8.1 How a Loop Works

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Job over
```

Output of Pseudo-Code

Before a loop begins, a *key variable* is initialized (1). This variable is used to evaluate a control expression (2). If the evaluation is true (3), the *loop body* is executed (4 to 6), otherwise control moves to the next statement after the loop (6). The key variable is then changed in the loop body (5) before control moves back to the beginning of the loop (2). The control expression is now re-checked (2) before the next *iteration* (traversal or repeat) can begin. If the loop is designed to run a finite number of times, the control expression will eventually turn false and cause the loop to terminate.

Even though all of the three loops offered by C broadly conform to the above scheme, there are minor differences between them. First, the initialization may be done in the loop construct itself (**`for`** loop) instead of being done outside. Second, the control expression may be evaluated at the end (**`do-while`** loop) instead of at the beginning. Finally, a control expression can't prevent a **`do-while`** loop from performing at least one iteration even though it can prevent the other loops from running at all.

A loop iteration can either be suspended (to start a new one) or terminated altogether without re-testing the control expression. This is achieved with the **`continue`** and **`break`** keywords that can be invoked anywhere in the loop body. You have already used this feature in one program of the previous chapter, so you'll find it a convenient alternative to the traditional method of using the control expression to determine loop continuity.

A loop is generally designed to execute a finite number of times, but many applications need a loop to run all the time. Such loops are known as *infinite loops*. We have seen one of them in the previous chapter (**while (1)**). Many Internet services use infinite loops to check whether a mail has been received or a user has keyed in text in a messaging application. However, even infinite loops can be terminated by other means.

8.2 THE while LOOP

You have seen the **while** loop at work in some of the previous chapters. The syntax, featuring both single and compound statements in the loop body, is shown in Figure 8.2. Don't forget to enclose the control expression in parentheses.

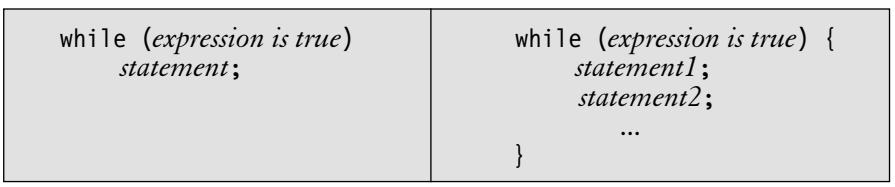


FIGURE 8.2 The while Statement

The **while** loop is virtually identical to the generic loop that was depicted in Figure 8.1. An *expression* containing a key variable is evaluated to true or false before loop iteration. If false, *the loop is not entered at all*, otherwise the body of the loop is executed. After execution, control reverts to the top to test *expression* that will determine whether the next iteration can begin. Since the key variable changes with every iteration, eventually its value causes *expression* to turn false and terminate the loop.

8.2.1 while_intro.c: An Introductory Program

Let's examine a small program (Program 8.1) that computes the sum of the first 10 integers and prints both a progressive and final sum. The key variable here is i, which is used three times in a **while** loop. Since this loop is finite, i will eventually exceed 10. The control expression will then turn false and prevent loop re-entry. The progressive sum is printed inside the loop and the final sum outside.

The **while** loop is an *entry-controlled loop*, which means that loop entry is not guaranteed. If the control expression is false to begin with, then the loop is not entered at all. This is also true of the **for** loop, but not the **do-while** loop which is an *exit-controlled loop*. The control expression in a **do-while** loop is tested at the end of the loop, which means that initial loop entry is guaranteed but not its subsequent iterations.

8.2.2 The Control Expression

The control expression is the nerve center of any loop. C lets you use any expression (relational, logical or constant) as long as it evaluates to an integer value. Any control expression used with **if** in Chapter 7 will work in identical manner with all loops. Here are some examples of usage of these expressions:

```
/* while_intro.c:  A simple while loop that prints the sum of the
                   first 10 integers. Also prints progressive sum. */

#include <stdio.h>

#define NO_OF_INTEGERS 10
int main(void)
{
   short i = 1;                    /* Key variable initialized */
   short sum = 0;
   printf("Progressive sum shown below:\n");

   while (i <= NO_OF_INTEGERS) {   /* Key variable tested */
      sum += i;
      i++;                         /* Key variable updated */
      printf("%hd ", sum);
   }
   /* Loop has terminated */
   printf("\nSum of first %d integers = %hd\n", NO_OF_INTEGERS, sum);

   return 0;
}
```

PROGRAM 8.1: **while_intro.c**

```
Progressive sum shown below:
1 3 6 10 15 21 28 36 45 55
Sum of first 10 integers = 55
```

PROGRAM OUTPUT: **while_intro.c**

```
while (reply == 'y')                          Relational expression
while (++num <= power)                        As above, but with a difference
while (base > 0 && power >= 0)                Logical expression
while (quot)                                  Expression is a variable
while (1)                                     Expression is a constant
while (printf("Enter an integer: "))          Expression is a function
```

The second expression is an eye-opener; it contains the updating of the key variable (`num`) in the control expression itself. The statement **`while (quot)`** is the same as **`while (quot > 0)`**. By the same token, **`while (1)`** is always true. The last expression using **`printf`** is also true because it evaluates to the number of characters printed.

8.2.3 Updating the Key Variable in the Control Expression

As discussed before, most loop applications are based on the *initialize-test expression-update key variable* model:

- Initialize the key variable.
- Test it in the control expression.
- Update the key variable in the loop body.

However, we can sometimes update the key variable in the control expression itself and get rid of one statement in the loop body. The following code related to the program **`while_intro.c`** will also work:

```
short i = 0, sum = 0;                                   i initialized to 0 not 1
while (++i <= 10) {                                     ++i instead of i
   sum += i;
   printf("%hd ", sum);
}
printf("\nSum of first 10 integers = %hd\n", sum);      Prints 55
```

The updating of the key variable `i` has now been moved to the control expression, but this movement needs the initial value of `i` to be changed as well.

Note: When you change the key variable in the control expression to one having prefix or postfix operators, you also need to change either the relational operator (say, > to >=) or the initial value of the key variable.

8.3 THREE PROGRAMS USING `while`

Before we continue exploring this loop in C, let's examine three simple programs that use simple control expressions. All of these programs are revised in subsequent chapters to use arrays and functions. Two of the later versions also use the *recursive* technique in contrast to the *iterative* approach of loops that is pursued here.

8.3.1 `factorial.c`: Determining the Factorial of a Number

The factorial of a positive integer n (denoted by $!n$) is the product of all positive integers less than or equal to n. Thus, $!4 = 4 \times 3 \times 2 \times 1 = 24$. Program 8.2 uses a **`while`** loop to compute the factorial of an integer supplied by the user. Section 11.13.1 shows an alternative way of solving the same problem using a function.

```
/* factorial.c: Program to determine factorial of a number
                        without using a function. */
#include <stdio.h>
int main(void)
{
   short num1, num2;
   unsigned int factorial = 1;     /* unsigned doubles the maximum value */
   printf("Factorial of which number? ");
   scanf("%hd", &num1);
   num2 = num1;                    /* Saving num1 before it changes */
   while (num1 >= 1) {
      factorial *= num1;
      num1--;
   }
   printf("Factorial of %hd = %d\n", num2, factorial);
   return 0;
}
```

PROGRAM 8.2: **`factorial.c`**

```
Factorial of which number? 4
Factorial of 4 = 24
```

```
Factorial of which number? 6
Factorial of 6 = 720
```

```
Factorial of which number? 1
Factorial of 1 = 1
```

PROGRAM OUTPUT: **factorial.c**

8.3.2 extract_digits.c: Program to Reverse Digits of an Integer

Program 8.3 performs the dual task of reversing the digits of an integer and printing their sum. Using 10 as the divisor, each digit is extracted from the right using the / and % operators. The program has the demerit of not saving the digits, so you can't perform any other task with them once printing is complete.

```
/* extract_digits.c: Reverses the digits of an integer. Prints and sums
                     the digits using the % and / operators. */
#include <stdio.h>
int main(void)
{
   unsigned int number;
   short last_digit, sum = 0;

   printf("Key in an integer: ");
   scanf("%d", &number);

   while (number != 0) {
      last_digit = number % 10;              /* Extracts last digit */
      number = number / 10;                  /* Removes last digit */
      sum += last_digit;
      printf("%hd  ", last_digit);
   }
   printf("\nSum of digits: %hd\n", sum);
   return 0;
}
```

PROGRAM 8.3: **extract_digits.c**

```
Key in an integer: 13569
9  6  5  3  1
Sum of digits: 24
```

PROGRAM OUTPUT: **extract_digits.c**

In every iteration of the **while** loop, the / operation removes the last digit and the % operation saves this digit. This is how the values of the variables last_digit and number change with every loop iteration:

	Iteration 1	*Iteration 2*	*Iteration 3*	*Iteration 4*	*Iteration 5*
last_digit	9	6	5	3	1
number	1356	135	13	1	0

The program is inflexible; we can't use this technique to print the digits in the sequence they occur in the number. We need to save every digit, not with five variables, but with an *array* of five elements. We'll revisit the program in Section 10.7.1.

8.3.3 fibonacci_ite.c: Printing and Summing the Fibonacci Numbers

Every student of computer science would have encountered the Fibonacci sequence of numbers at some point in their learning curve. In this sequence, which starts with the values 0 and 1, each subsequent term is the sum of the previous two. This relationship can be seen in the first 12 terms of this sequence:

```
0    1    1    2    3    5    8    13    21    34    55    89
```

Here, 55 is the sum of its two predecessors, 34 and 21, while 21 is the sum of 13 and 8. Program 8.4 prints and sums *n* terms of the sequence where *n* is provided by the user. The program employs a **while** loop that runs *n* times. In each iteration, the sum of the variables prev1 and prev2 are saved in sum and printed. Note that the first two terms are generated from the logic which has not treated them in an exceptional manner. This has been made possible by careful selection of the initialized values of four variables.

```
/* fibonacci_ite.c: Uses the iterative technique to generate terms of
                    the Fibonacci series and compute the sum of terms. */
#include <stdio.h>
int main(void)
{
   short num;
   unsigned long prev1 = 0, prev2 = 0, sum = 1, sum_terms = 0;
   short i = 0;
   printf("Enter last term of series: ");
   scanf("%hd",&num);
   while (i++ < num) {
      printf("%ld ", sum);            /* Prints each term */
      sum_terms += sum;               /* Sums all terms */
      prev1 = prev2;
      prev2 = sum;
      sum = prev1 + prev2;            /* Sums previous two terms */
   }
   printf("\nSum of first %hd terms = %ld\n", num, sum_terms);
   return 0;
}
```

PROGRAM 8.4: **fibonacci_ite.c**

```
Enter last term of series: 15
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
Sum of first 15 terms = 1596
```

PROGRAM OUTPUT: **fibonacci_ite.c**

The Fibonacci sequence can also be generated by using a *recursive* function, i.e., a function that calls itself repeatedly. Section 11.14.3 examines this function and its demerits compared to the iterative approach used here.

8.4 LOADING THE CONTROL EXPRESSION

Repetition creates opportunities for using functions as control expressions. For instance, the **getchar** function, a member of the I/O family of functions, fetches a character from the keyboard and returns its ASCII value. The **putchar** function uses this value to display the character. Consider the following code snippet:

```
int c;
c = getchar();                  Gets the first character
while (c != EOF) {              EOF is end-of-file
   putchar(c);                  Displays the character
   c = getchar();               Fetches the next character
}
```

This appears to be a simple way of retrieving a string of text from keyboard input. But because **getchar** returns the ASCII value of the fetched character, you can move the **getchar** statement itself to the control expression:

```
while ((c = getchar()) != EOF)
   putchar(c);
```

This works perfectly and looks clean as well because a single **getchar** is used. The `!=` has a higher precedence than the `=`, so parentheses must be provided around the assignment (`c = getchar()`). This ensures that the return value of **getchar** is assigned to `c` before `c` is compared to EOF (end-of-file).

Many functions return a value. **scanf** returns the number of items successfully read and **printf** returns the number of characters written, so these functions can also be used as control expressions:

```
while (scanf("%d", &num) == 1)
while (printf("Enter the amount: "))
```

Both **getchar** and **putchar** are discussed in Chapter 9 which examines the major I/O functions. The **scanf** and **printf** functions are also examined in the same chapter.

8.4.1 Merging Entire Loop Body with the Control Expression

A previous example merged both calls to **getchar** to a single call in the control expression. But **putchar** also returns a value (that of the character written), so why not move it too to the control expression? Yes, we can do that as well:

```
while ((c = getchar()) != EOF && putchar(c))
  ;                                          Null statement
```

The value of the expression `putchar(c)` is `c`. This value is true for all ASCII characters except the NUL character, so the **`putchar`** expression is always true. This compound expression will turn false only when the input to **`getchar`** is exhausted. Since this loop has no body, don't forget to follow the **`while`** statement with a semicolon, preferably on a separate line.

Tip: Don't merge too many statements with the control expression if that destroys clarity. Program clarity is more important than compactness. In the preceding case, you should stop after moving **`getchar`** to the control expression. The **`putchar`** statement is better left in the loop body.

8.4.2 When You Actually Need a Null Body

There are situations when you actually need to have a null body. When **`scanf`** is used with the `%c` format specifier in a loop, whitespace characters often get trapped in the buffer. Subsequent invocations of **`scanf`** get these trapped characters rather than fresh user input. So, before you call **`scanf`** again, call **`getchar`** repeatedly to empty the buffer:

```
scanf("%c", &reply);                    \n generated by [Enter] remains in buffer
while (getchar() != '\n')               \n removed from buffer
   ;                                    No further action needed
```

Here, **`getchar`** continues reading from the buffer until it encounters the newline. This ensures that the next **`scanf`** call operates with an empty buffer. A null body is mandatory because the control expression itself has done the job of clearing the buffer.

Caution: When using **`while`** (or any loop) without a formal body, remember to place a semicolon immediately below it to indicate that the loop runs a null command. If you fail to provide the **`;`**, the statement following **`while`** will be considered as the body of the loop, which obviously is not intended.

8.5 NESTED `while` LOOPS

Just as an **`if`** statement can contain another **`if`** statement, nothing prevents us from using a second **`while`** inside an outer **`while`** construct. Figure 8.3 presents two views of the structure of a nested **`while`** loop. The view on the right highlights the way it is normally used. It shows the locations of initialization and updating of key variables of both loops.

The nested loop starts with the initialization of *key variable1*, followed by evaluation of *expression1*. If the result is true, the outer loop is entered. Before entry into the inner loop, *key variable2* is initialized. If and after *expression2* evaluates to true, the inner loop is entered. When this loop terminates, *key variable1* of the outer loop is updated before control moves to the top of this loop for re-evaluation of *expression1*.

Thus, each iteration of the outer loop leads to multiple iterations of the inner loop. The three programs that follow make use of this one-to-many relationship that exists between the inner and outer **`while`** loops.

Syntax Form	Usage Form

`while (`*expression1*`) {`
`  ...`
`  while (`*expression2*`) {`
`  ...`
`  }`
`}`
	initialization1
	`while (`*expression1*`) {`
	`  ...`
	initialization2
	`while (`*expression2*`) {`
	`  ...`
	Update key variable2
	`}`
	Update key variable1
	`}`

FIGURE 8.3 Structure of a Nested `while` Loop

8.5.1 `nested_while.c`: Printing a Multiplication Table

Consider printing a table of *x* rows and *y* columns. This is done by printing all *y* columns of one row before taking up the next row. The task is completed with the printing of the *y*th column of the *x*th row. A nested **`while`** construct is thus the perfect choice for handling two-dimensional data like tables and arrays as the following program (Program 8.5) shows.

```
/* nested_while.c: Uses nested while loop to print multiplication table
   for positive integers up to 5. Maximum size of multiplier = 12 */
#define ROWS 5
#define COLUMNS 12
#include <stdio.h>
int main(void)
{
   int x = 0, y;                    /* Key variable x initialized here ... */
   while (x++ < ROWS) {             /* ... and updated in control expression */
      y = 0;                        /* Key variable y initialized here ... */
      while (y++ < COLUMNS)         /* ... and updated in control expression */
         printf("  %3d", x * y);
      printf("\n");
   }
   return 0;
}
```

PROGRAM 8.5: **`nested_while.c`**

```
  1   2   3   4   5   6   7   8   9  10  11  12
  2   4   6   8  10  12  14  16  18  20  22  24
  3   6   9  12  15  18  21  24  27  30  33  36
  4   8  12  16  20  24  28  32  36  40  44  48
  5  10  15  20  25  30  35  40  45  50  55  60
```

PROGRAM OUTPUT: **`nested_while.c`**

8.5.2 half_pyramid.c: Printing a Half-Pyramid with Digits

You need to use nested loops to print predictable two-dimensional patterns like pyramids and half-pyramids. Consider the following pattern that uses digit-characters where the digit used is also the sequence number of the row:

```
1
2 2
3 3 3
4 4 4 4
```

Even though the **for** loop is an automatic choice for tasks such as the one above, the next program (Program 8.6) uses two **while** loops to print this half-pyramid. The number of rows to be printed is taken as user input. Observe that one key variable (col) is updated in the control expression but the other (row) is not. That is because row is used in both control expressions.

```
/* half_pyramid.c: Uses a nested while structure to print a half-pyramid. */
#include <stdio.h>
int main(void)
{
   short row = 1, col, rows_max;
   printf("Number of rows? ");
   scanf("%hd", &rows_max);
   while (row <= rows_max) {
      col = 1;                    /* Key variable col initialized here ... */
      while (col++ <= row)        /* ... and updated in control expression */
         printf("%d ", row);      /* Prints row number in each row */
      row++;
      printf("\n");               /* Moves to next row */
   }
   return 0;
}
```

PROGRAM 8.6: **half_pyramid.c**

```
Number of rows? 5
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

PROGRAM OUTPUT: **half_pyramid.c**

Since there is no limit on the depth of nesting (i.e., the number of inner loops possible), you can handle multi-dimensional data quite easily.

Takeaway: In a nested **`while`** structure there is a one-to-many relationship between the outer and inner loop. For every iteration of the outer loop, the inner loop must complete its set of iterations before the next iteration of the outer loop can begin.

8.5.3 `power.c`: Computing Power of a Number

Program 8.7 uses an outer loop to take two integers (x and y) as user input. It then uses an inner loop to compute the value of x raised to the power y. The program handles only positive integers but quits when the user keys in a zero (for the first number) or a negative integer (for both numbers).

```
/* power.c: Computes power of multiple sets of numbers using a nested
            while loop. Repeatedly multiplies a number by itself. */
#include <stdio.h>
int main(void)
{
   short num, base = 1, power = 1;
   unsigned long total;

   while (base > 0 && power >= 0) {
      printf("Enter base and power (0 0 to exit): ");
      scanf("%hd%hd", &base, &power);

      if (base <= 0 || power < 0) {          /* Invalid Input */
         printf("Invalid Input. Quitting ...\n");
         return 1;
      }

      num = 0; total = 1;           /* Initialization before inner loop */
      while (++num <= power)
         total *= base;             /* Multiply base by itself */
      printf("%d to the power %d = %lu\n", base, power, total);
   }
   return 0;
}
```

PROGRAM 8.7: **`power.c`**

```
Enter base and power (0 0 to exit): 2 0
2 to the power 0 = 1
```

```
Enter base and power (0 0 to exit): 2 10
2 to the power 10 = 1024
```

```
Enter base and power (0 0 to exit): 0 0
Invalid Input. Quitting ...
```

PROGRAM OUTPUT: **`power.c`**

The outer **while** loop repeatedly prompts for two integers. After performing the necessary validation, the program passes on valid numbers to the inner loop. For each iteration of this loop, `total` is assigned the values `base`, `base * base`, `base * base * base`, and so forth until the key variable `num` equals `power` that is set by user input.

Note: The maths library supports the **pow** function which does the same job as this program. Chapter 12 features a program that uses this function. For a program containing **pow**, the compilation process is different: the linker must be *explicitly* invoked to include the code of **pow** in the executable.

8.6 USING while WITH break AND continue

The **while** loops seen so far used the control expression to determine whether it should iterate or terminate. However, some situations demand immediate suspension of the current iteration to either break out of the loop or restart a new iteration. C supports two keywords—**break** and **continue**—that can perform these two tasks (Fig. 8.4).

When the program sees **break**, control breaks out of the loop to resume execution from the statement immediately below the end of the loop. On the other hand, **continue** doesn't terminate a loop but suspends loop execution to resume the next iteration. Both situations usually occur when a variable or expression used in the loop acquires a value that makes further execution of statements undesirable.

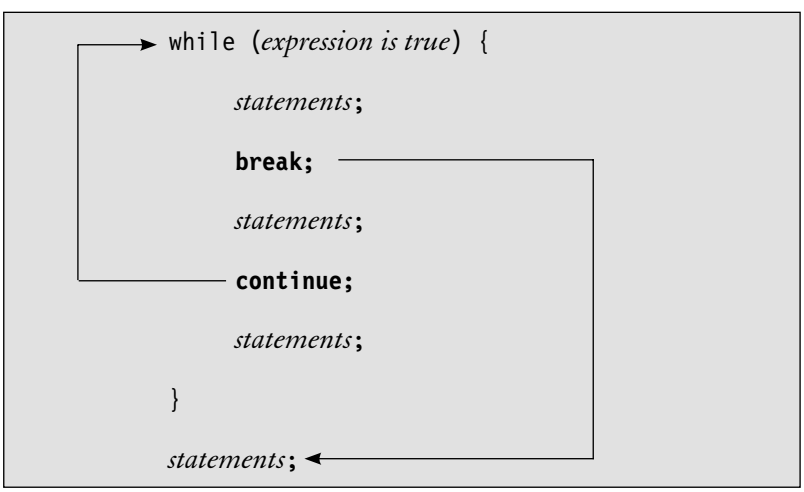


FIGURE 8.4 How **continue** and **break** Behave in a **while** Loop

The next program (Program 8.8) uses an infinite **while** loop to enable repeated entry of three positive numbers. It uses **continue** to ask for a new set of numbers in case at least one of them is found to be negative. If the numbers are all positive, then the **break** statement terminates the loop. The **printf** statement below the closing brace of **while** then prints the sum of the numbers. The program also terminates when zero is input for all three numbers.

```
/* break_continue.c: Uses break and continue in a loop. */
#include <stdio.h>
int main(void)
{
   short a, b, c;
   while (1) {
      printf("Enter three integers a b c: ");
      scanf("%hd %hd %hd", &a, &b, &c);

      if (a == 0 && b == 0 && c == 0) {
         printf("All values zero. Quitting ...\n");
         return 1;
      }
      else if (a < 0 || b < 0 || c < 0) {
         printf("Negative integer not allowed\n");
         continue;                /* Goes to while (1) */
      }
      else
         break;                   /* Goes to next printf statement */
   }
   printf("Sum of %d %d %d: %d\n", a, b, c, a + b + c);
   return 0;
}
```

PROGRAM 8.8: **break_continue.c**

```
Enter three integers a b c: 3 4 -5
Negative integer not allowed
Enter three integers a b c: 3 4 5
Sum of 3 4 5: 12
Enter three integers a b c: 0 0 0
All values zero. Quitting ...
```

PROGRAM OUTPUT: **break_continue.c**

Structured programming theorists disapprove of the use of **break** and **continue**. They feel that loops should provide a single point of exit (where the control expression is tested). But this arrangement comes at a price; a number of variables need to be set and tested at various points in the loop. After you have used **break** and **continue**, you may not be willing to pay that price.

8.7 prime_number_check.c: MORE OF break AND continue

We end our discussions on the **while** loop with Program 8.9, which checks whether a number is prime or not. A *prime* number is one that is not divisible by any number other than itself and 1. The program accepts multiple numbers from the keyboard and terminates when zero is input. The exercise is carried out in a nested **while** loop structure that uses **break** and **continue**. The program is grouped into four sections.

```
/* prime_number_check.c: Determines whether a variable is prime or not.
                         Uses a nested loop with break and continue. */
#include<stdio.h>
int main(void)
{
   int num, divisor;
   char is_prime;

   /* (A) Number input and validation */
   printf("Enter numbers for prime test (0 to quit): ");

   while (scanf("%d", &num) == 1) {
      if (num < 0) {
         printf("Invalid input\n");
         continue;
      }
      else if (num == 0)
         break;                          /* Go to (D) */

      /* (B) Check for prime */
      divisor = 2;
      is_prime = 'y';                    /* Assume number is prime */
      while (divisor < num) {
         if (num % divisor == 0) {
            is_prime = 'n';              /* Number is not prime */
            break;                       /* Go to (C) */
         }
         divisor++;
      }

      /* (C) After break in (B) */
      is_prime == 'y' ? printf("%d is prime\n", num) :
                  printf("%d is not prime, divisible by %d\n", num, divisor);
   } /* End of outer loop */

   /* (D) After break in (A) */
   printf("Quitting ...\n");
   return 0;
}
```

PROGRAM 8.9: **prime_number_check.c**

```
Enter numbers for prime test (0 to quit): 19 49 -5 137 1234567 0
19 is prime
49 is not prime, divisible by 7
Invalid input
137 is prime
1234567 is not prime, divisible by 127
Quitting ...
```

PROGRAM OUTPUT: **prime_number_check.c**

(A) The outer **while** loop uses an unusual control expression—the **scanf** function which returns the number of items read. Since **scanf** has been used here with a single format specifier, it returns 1 in every successful invocation. Negative numbers are handled by **continue**, which halts further processing and moves control back to the top (A) to read the next number. The **break** statement handles a zero by terminating this outer loop and switching control to (D) from where the program terminates. Both **continue** and **break** make it impossible to execute the remaining loop statements once validation fails.

(B) To determine whether a number *n* is prime, it has to be repeatedly divided (in the inner loop) by all values between 2 and *n* - 1. We assume here that a number is prime unless proved otherwise, so `is_prime` (a flag) is initially set to `'y'`. This inner loop breaks in either of the following situations:

- the modulo operation (`num % divisor`) evaluates to 0 which means that the number is not prime. `is_prime` is then set to `'n'`. **break** terminates this loop.
- `divisor` is one less than `num`. The loop terminates normally without changing the default value of `is_prime` (`'y'`).

(C) At this point we know the value of `is_prime`. A conditional expression that uses two **printf** expressions takes the final call on the prime property of the number. Note that these expressions are used here only for their side effect.

Note: The **break** statement terminates only the current loop, so if there is an enclosing loop, **break** moves control to the next statement of that loop. If nesting is too deep, it becomes difficult to come out of the innermost loop without quitting the program (with **return;**). A **goto** would then be the most appropriate statement to use.

8.8 THE do-while LOOP

C supports a **do-while** loop which functionally differs from **while** in only one respect: the control expression is tested at the *end* of the loop (Fig. 8.5). This means that a **do-while** loop is guaranteed to perform at least one iteration. The loop begins with the **do** keyword and ends with the **;** in the last line which contains the **while** keyword. This is in contrast to the **while** loop which terminates at the **;** of its only or last statement.

```
do
   statement;
while (expression is true);
```

```
do {
     statement1;
     statement2;
        ...
} while (expression is true);
```

FIGURE 8.5 The **do-while** Statement

A **do-while** loop is an *exit-controlled loop* and not an entry-controlled one. Any **while** loop can be replaced with a **do-while** loop only if the application requires the loop to be executed at least once. A previous program section using a **while** statement (Program 8.1) can thus be replaced in this way:

```
short i = 1, sum = 0;
do {
   sum += i;
   i++;
} while (i <= 10);
printf("Sum of first 10 integers = %hd\n", sum);
```

Note the ; at the end of the control expression (`i <= 10`). The incrementing of the key variable, `i`, can be moved to the control expression (which would also require `<=` to be replaced with `<`). Since that move leaves a single statement in the body, you can drop the braces too. Many programmers prefer to use braces even if they enclose a single statement. Here are the two forms:

```
do
   sum += i;
while (i++ < 100);
```

```
do {
   sum += i;
} while (i++ < 100);
```

Select the form you find comfortable to work with and use it consistently if you have to use a **do-while** loop at all. We'll now develop a useful application using this loop.

8.9 decimal2binary.c: COLLECTING REMAINDERS FROM REPEATED DIVISION

Program 8.10 uses a **do-while** loop to convert a decimal number to binary *but without reversing the bits*. Recall that the reversed bit pattern actually represents the equivalent binary number *(3.2.2)*. The program also displays the number of bits needed to store the binary number.

```
/* decimal2binary.c: Uses a do-while loop to convert a decimal number
                     to binary but without reversing the digits. */
#include <stdio.h>

int main(void)
{
   long quot;
   short count = 0, rem = 0;

   printf("Enter a decimal integer: ");
   scanf("%ld", &quot);

   do {
      rem = quot % 2;            /* Either 0 or 1 */
      quot /= 2;                 /* Quotient progressively decreases */
      printf("%hd ", rem);
      count++;
   } while (quot > 0);

   printf("\nNumber of iterations = %hd\n", count);

   return 0;
}
```

PROGRAM 8.10: **decimal2binary.c**

```
Enter a decimal integer: 35
1 1 0 0 0 1                                   Binary of 35 is 100011
Number of iterations = 6
```

```
Enter a decimal integer: 128
0 0 0 0 0 0 0 1
Number of iterations = 8
```

PROGRAM OUTPUT: **decimal2binary.c**

The number obtained from user input is subjected to repeated division and modulo operation in a **do-while** loop. The loop terminates when `quot`, which progressively decreases with each iteration, eventually drops to zero. Because the generated bits are not saved, the program can't reverse them. To correctly represent the binary equivalent of the number, we need to use an array to store the bits and then read the array backwards. We'll improve this program in Section 8.12.

8.10 THE for LOOP

Like **while** and **do-while**, the **for** loop also uses the *key variable/control expression* paradigm to determine loop entry and continuity. It is a compact construct that gathers three expressions at one place (Fig. 8.6). Any **while** loop can be replaced with a **for** loop, and for many, the latter is the preferred loop.

```
for (exp1; exp2; exp3)
    statement;
```

```
for (exp1; exp2; exp3) {
    statement1;
    statement2;
        ...
}
```

FIGURE 8.6 The **for** Statement

The **while** loop works on the principle of initializing a key variable before the loop, testing it in the control expression and updating the key variable in the body. The three expressions in **for** do the same work but at one place. The initialization is done by *exp1*, the key variable check is done by *exp2*, and updating of the key variable is done by *exp3*. These expressions, which are delimited by semicolons, determine whether a loop will run, and if so, how many times.

The left side of Figure 8.7 shows the use of a **for** loop to print the ASCII values of the uppercase letters. The right side shows the equivalent code using **while**. All three expressions in **for** are not evaluated in every loop iteration, so let's understand the sequence of operations used by **for** in this program:

1. The first expression (`chr = 65` and *exp1* in syntax) initializes the key variable (`chr`).
2. The second expression (`chr < 91` and *exp2*), which is the control expression for this loop, is evaluated. If the evaluation is true, the loop is entered.

```c
#include <stdio.h>
int main(void)
{
   int chr;
   for (chr = 65; chr < 91; chr++)
      printf("%c=%d ", chr, chr);
   return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
   int chr = 65;
   while (chr < 91) {
      printf("%c=%d ", chr, chr);
      chr++;
   }
   return 0;
}
```

PROGRAM OUTPUT:

```
A=65 B=66 C=67 D=68 E=69 F=70 G=71 H=72 I=73 J=74 K=75 L=76 M=77 N=78 O=79 P=80
Q=81 R=82 S=83 T=84 U=85 V=86 W=87 X=88 Y=89 Z=90
```

FIGURE 8.7 Difference in Form Between **for** (left) and **while** (right)

3. If the result of evaluation in Step 2 is true, the **printf** statement is executed.
4. After an iteration has been completed, control moves to the third expression (`chr++` and *exp3*) where the key variable is incremented.
5. Control now goes back to Step 2 where the control expression is re-tested. The loop starts the next iteration if `chr` is still less than 91.
6. `chr` will eventually have the value 91. The control expression will then turn false and prevent loop re-entry. The **return** statement terminates the program.

Like **while**, **for** is an entry-controlled loop because *exp1* and *exp2* jointly determine whether the loop will be entered at all. This loop is special because of the flexibility of expression usage. Any of these three expressions can comprise comma-separated multiple expressions, or it can be dropped altogether. Using this feature, you can dismantle the set of three expressions to provide a "while"-like look to **for**:

```c
int chr = 65;                                   exp1
for (; chr < 91;) {                             exp2
   printf("%c=%d ", chr, chr);
   chr++;                                       exp3
}
```

Here, *exp1* and *exp3* have been moved out, leaving their slots with null expressions. The semicolons are mandatory even if the expressions are absent. Like in **while**, some statements of the loop body can also be moved to the control expression.

Takeaway: In the first iteration of a **for** loop, the sequence is *exp1* followed by *exp2*. In subsequent iterations, it is *exp3* followed by *exp2*. *exp1* is evaluated only once—before the loop is entered.

8.10.1 ascii_letters.c: A Portable ASCII Code Generator

Program 8.11 generates the *machine* values of the lowercase letters. It prints on each line the values of nine letters in the form *letter=value*. The program is fully portable because, unlike the previous program which works with specific ASCII values, this program works with the character representation of the letters. As a consequence, the program will run without modification on both ASCII and EBCDIC systems.

```
/* ascii_letters.c: A portable version of the ASCII list generator.
                    Runs without modification on EBCDIC machines. */
#include <stdio.h>

int main(void)
{
   short chr, count = 0;
   short diff = 'a' - 'A';        /* Will be different in EBCDIC and ASCII */

   for (chr = 'A'; chr <= 'Z'; chr++) {
      printf("%c = %3hd ", chr + diff, chr + diff);
      if (++count == 9) {         /* Nine entries in one line only */
         printf("\n");
         count = 0;
      }
   }
   return 0;
}
```

PROGRAM 8.11: **ascii_letters.c**

```
a =  97 b =  98 c =  99 d = 100 e = 101 f = 102 g = 103 h = 104 i = 105
j = 106 k = 107 l = 108 m = 109 n = 110 o = 111 p = 112 q = 113 r = 114
s = 115 t = 116 u = 117 v = 118 w = 119 x = 120 y = 121 z = 122
```

PROGRAM OUTPUT: **ascii_letters.c**

The machine values of `A` and `a` in ASCII are 65 and 97, respectively. Machines that support other character sets (like EBCDIC) have different values. For the program to be independent of the character set, we need to store the difference of these two values in a separate variable (`diff`). We can then derive the lowercase value of a letter from knowledge of its uppercase value. The program is portable because instead of using `chr + 32`, we have used `chr + diff`.

The **`for`** loop initializes the key variable `chr` to `'A'` before it evaluates the control expression `chr <= 'Z'`. Since this is true and will remain true for the next 25 iterations, the loop is entered. Next, **`printf`** prints the value of `'a'` in a space 3 characters wide (because of `%3hd`) before beginning the next iteration. A simple equality test in the loop ensures the printing of nine entries in each line.

After completing one iteration, control moves to the top where `chr` is incremented and the control expression re-evaluated. The entire cycle is repeated until the value of `chr` exceeds `'Z'`, when the loop terminates. However, in every ninth iteration, **`printf`** prints a newline and `count` is reset to zero.

Tip: When designing programs that change the case of letters, follow the preceding technique of storing the difference between a lowercase and uppercase letter in a variable, and then use this value for case conversion. Your programs won't be portable if you derive the value of `'a'` by adding 32 to `'A'` simply because this relation is true only for ASCII systems.

8.10.2 `print_terms.c`: Completing a Series of Mathematical Expressions

You can use a **`for`** loop to complete a series comprising expressions (terms) where a fixed relation exists between two consecutive terms. Consider the following series comprising the squares of consecutive integers where the odd terms are added and even terms are subtracted:

```
1 - 4 + 9 - 16 + 25 ...
```

The nth term in the series is represented as n^2. The following program (Program 8.12) completes the series up to n terms where n is set by the user. The program also evaluates the complete expression by choosing the + and - operators correctly. The annotations adequately explain the working of the program.

```
/* print_terms.c: Program to complete a series of squares of integers.
                  Adds even terms but subtracts odd terms. */
#include<stdio.h>
int main(void)
{
   short n, n_max, sum = 0;
   printf("Enter max number of terms: ");
   scanf("%hd", &n_max);
   for (n = 1; n <= n_max; n++) {
      if (n % 2 == 0)                /* For even integers ... */
         sum += n * n;               /* ... add to sum */
      else                           /* For odd integers ... */
         sum -= n * n;               /* ... subtract from sum */
      printf("%hd ", n * n);         /* Displays square of integer */
   }
   printf("\nSum = %hd\n", sum);
   return 0;
}
```

PROGRAM 8.12: **`print_terms.c`**

```
Enter max number of terms: 5
1 4 9 16 25
Sum = -15
```

```
Enter max number of terms: 10
1 4 9 16 25 36 49 64 81 100
Sum = 55
```

PROGRAM OUTPUT: **`print_terms.c`**

8.11 for: THE THREE EXPRESSIONS (*exp1*, *exp2*, *exp3*)

Each component of the combined expression *exp1*; *exp2*; *exp3* need not be a single expression. You can use a comma-operated expression to represent any of these expressions. You can drop any expression or take the extreme step of dropping all of them! The following paragraphs discuss some of the ways these three expressions can be tweaked to our advantage.

8.11.1 Moving All Initialization to *exp1*

To appreciate the usefulness of the comma operator, have a look at the two initializations made before the loop in the program **ascii_letters.c** (Program 8.11):

```
diff = 'a' - 'A';
count = 0;
```

Since *exp1* is evaluated only in the first iteration, there's no reason why we can't move these assignments to *exp1*. The **for** statement should now look like this:

```
for (diff = 'a' - 'A', count = 0, chr = 'A'; chr <= 'Z'; chr++)
```

The value of *exp1* (containing the two comma operators) is now 'A' (the value of the right-most expression), but that is of no consequence to us in this program. What matters to us is that the three expressions together can be treated as *exp1*.

8.11.2 Moving Expression Statements in Body to *exp3*

Let's now consider the program, **while_intro.c** (Program 8.1), which adds the first 10 integers using a **while** loop. The equivalent code using **for** is shown below:

```
short i, sum;
for (i = 1, sum = 0; i <= 10; i++) {
   sum += i;
   printf("%hd ", sum);
}
printf("\nSum of first 10 integers = %hd\n", sum);              Prints 55
```

The loop body contains two statements, which along with i++ (*exp3*), constitute the body of the equivalent **while** loop. So if i++ can represent *exp3*, why not sum += i? Yes, we can move this operation to *exp3*, but we also need to change two things: initialize i to 0 (instead of 1) and change the relational operator to < (instead of <=):

```
for (i = 0, sum = 0; i < 10; ++i, sum += i)
   printf("%hd ", sum);                                        Also prints 55
```

But then **printf** is also an expression, so why leave it out?

```
for (i = 0, sum = 0; i < 100; ++i, sum += i, printf("%hd ", sum))
   ;                                                           Also prints 55
```

This **for** loop has no body as shown by the null statement (the solitary ;) on a separate line. This ; is necessary, otherwise the subsequent **printf** statement will be considered as the body of the loop, which it is not.

8.11.3 Dropping One or More Expressions

Program 8.13 uses the **scanf** function to convert a line of keyboard input from lower- to uppercase. This function, when used with %c, saves the fetched character in a char variable. The **for** loop in this program has a component missing, while the other two components are unrelated.

```
/* lower2upper.c: Converts a line of keyboard input to uppercase. */
#include <stdio.h>
int main(void)
{
   char c, diff;
   for (diff = 'a' - 'A'; scanf("%c", &c) == 1; ) {
      if (c >= 'a' && c <= 'z')
         printf("%c", c - diff);
      else
         printf("%c", c);
   }
   return 0;
}
```

PROGRAM 8.13: **lower2upper.c**

```
dont forget to use the & prefix with the variables used in scanf.
DONT FORGET TO USE THE & PREFIX WITH THE VARIABLES USED IN SCANF.
[Ctrl-d]
```

PROGRAM OUTPUT: **lower2upper.c**

The expression *exp2* of this **for** loop behaves in an unusual manner. Every iteration of the loop runs **scanf** to assign the next input character to the variable c. The loop thus reads a complete line of input and terminates when it encounters EOF. Because there is no key variable here, we don't need to use *exp3* at all. Its absence is indicated by the solitary **;** at the end of the complete expression. Like Program 8.11, this program is also portable.

Note: The return value of the **scanf** expression acts as a key variable in this program. Its side effect is also utilized by the program.

8.11.4 The Infinite Loop

Depending on the nature of the problem, any of the three expressions can be dropped. As a special case, all of them can be dropped to create an infinite loop:

```
for (;;)                    Infinite loop; like while (1)
```

Though this isn't quite intuitive, the absence of all expressions signifies a perpetually true condition. Since *exp2* needs to be permanently true, the statement **for (;1;)** also creates an infinite loop, but **for (;0;)** doesn't. Like in the **while** loop, any infinite loop can be terminated with **break**.

Note: In most cases, the **for** loop uses the same variable as the three components of the combined expression. However, this is not the case in Program 8.13; the variables **diff** and **c** are completely unrelated to each other. Be prepared to see **for** loops where exp1, exp2 and exp3 are represented by three different variables.

8.12 decimal2binary2.c: CONVERTING A DECIMAL NUMBER TO BINARY

Program 8.14 is an improved version of its predecessor (Program 8.10). It *saves* in an array the remainders generated by repeated division of a number by 2. The program then reverse-reads the array to generate the binary equivalent of the decimal integer. The program uses two **for** loops for saving and reading the bits.

```
/* decimal2binary2.c: Stores remainders generated by repeated integer
   division in an array and then reverse-reads the array. */

#include <stdio.h>

int main(void)
{
   unsigned long quot;
   char binary_digits[80];       /* Adequate for remainders of 80 divisions */
   short rem, index;

   printf("Enter a decimal integer: ");
   scanf("%lu", &quot);

   for (index = 0; quot > 0; index++, quot /= 2) {
      rem = quot % 2;
      binary_digits[index] = rem == 1 ? '1' : '0';
   }
                                        /* All remainders stored ... */
   for (index--; index >= 0; index--)   /* ... Now reverse-read the array. */
      printf("%c ", binary_digits[index]);

   return 0;
}
```

PROGRAM 8.14: **decimal2binary2.c**

```
Enter a decimal integer: 8
1 0 0 0
```

```
Enter a decimal integer: 35
1 0 0 0 1 1
```

```
Enter a decimal integer: 128
1 0 0 0 0 0 0 0
```

PROGRAM OUTPUT: **decimal2binary2.c**

Repeated division by 2 in the first **for** loop generates the values 0 and 1 of type `short`. Because the array, `binary_digits`, is of type `char`, these `short` values are transformed to `char` using a conditional expression. After the first **for** loop has completed its run, `binary_digits` has all the remainders. A second **for** loop now reads this array backwards by repeatedly decrementing the value of `index` which was set by the previous loop.

8.13 NESTED for LOOPS

One **for** loop can enclose any number of inner **for** loops. For reasons of compactness, nested **for** loops are often preferred to nested **while** loops. For instance, the multiplication table created with a nested **while loop** in Section 8.5.1 can also be printed with a compact nested **for** loop structure. In the following equivalent construct, for each value of `x` (1 to 5), `y` can take on 12 values (1 to 12):

```
int x, y;
for (x = 1; x <= 5; x++)  {
   for (y = 1; y <= 12; y++)
      printf("  %3d", x * y);
   printf("\n");
}
```

Nested **for** loops are the perfect tool to use for printing patterns. We have used nested **while** loops to print a half-pyramid (Program 8.6), so let's use a nested **for** loop to print a complete pyramid with asterisks. Program 8.15 prints the following pattern when the number of rows is set to 6 by user input:

```
     *
    ***
   *****
  *******
 *********
***********
```

```
/* pyramid.c: Prints a pyramid pattern with the * using nested for loops. */
#include <stdio.h>
int main(void)
{
   short i, j, k, rows;
   printf("Number of rows? ");
   scanf("%hd", &rows);
   for (i = 1; i <= rows; i++) {
      for (j = 1; j <= rows - i; j++)
         printf(" ");             /* Prints spaces before pattern */
      for (k = 0; k != 2 * i - 1; k++)
         printf("*");
      printf("\n");
   }
   return 0;
}
```

PROGRAM 8.15: **pyramid.c**

Each line is printed by increasing the number of asterisks by 2 and decreasing the number of leading spaces by 1. This logic is captured by the control expressions of two **for** loops in Program 8.15. The expression `j <= rows - 1` controls the number of spaces, and the expression `k != 2 * i - 1` controls the number of asterisks.

8.14 USING for WITH break AND continue

The keywords, **break** and **continue** , apply to all loops including **for**. **break** terminates the current loop and moves control to the statement following the end of the loop. **continue** suspends the current iteration to start the next iteration. Figure 8.8 shows small code fragments that highlight their usage in the **for** loop.

You have already experienced the convenience of using **break** and **continue** in a **while** loop. It's clear that, without them, one would need to use additional code to bypass subsequent processing. You would need to set special flags to force the control expression to change normal loop behavior *(8.6)*. A previous caveat applies here as well: a **break** statement in the inner loop can't take control out of the enclosing loop.

```
for (div = 3; div < num; div += 2) {
   if (num % div == 0) {
      is_prime = 'n';
      break;
   }
}
```

```
for (;;) {
   printf("Number for prime test: ");
   scanf("%d", &number);
   if (number < 0) {
      printf("Invalid input\n");
      continue;
   }
      ... rest of code ...
```

FIGURE 8.8 Use of **break** and **continue** in a **for** Loop

8.15 all_prime_numbers.c: USING break AND continue

The final program of this chapter (Program 8.16) generates all prime numbers not exceeding a user-set limit. In Program 8.9, we determined whether a number is prime by dividing it by *all* smaller integers. That program unnecessarily included even numbers but the current program skips them. For reasons of convenience, the program doesn't print 2, the first prime number.

This program considers only *odd* values of both divisor and dividend. The outer **for** loop cycles through odd values of the dividend and the inner loop does the same for the divisor. The program works on the principle that a number is prime unless proved otherwise.

```
/* all_prime_numbers.c: Uses a nested for loop to generate all prime numbers
                        not exceeding a user-set limit. Starts from 3. */
#include <stdio.h>
int main(void)
{
   int num, max_num, div;
   char is_prime;

   printf("Maximum number to be tested for prime: ");
   scanf("%d", &max_num);

   for (num = 3; num <= max_num; num += 2) { /* Tests odd integers only */
      is_prime = 'y';
      for (div = 3; div < num; div += 2) {   /* Divisor is also odd */
         if (num % div == 0) {
            is_prime = 'n';                  /* Number is not prime so ... */
            break;                           /* no further check required */
         }
      }
      if (is_prime == 'y')                   /* If prime number found ... */
         printf("%d ", num);                 /* ... print it */
   }
   printf("\n");
   return 0;
}
```

PROGRAM 8.16: **all_prime_numbers.c**

```
Maximum number to be tested for prime: 100
3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

PROGRAM OUTPUT: **all_prime_numbers.c**

WHAT NEXT?

All work in C is performed by functions, and we must know the ones that interact with the terminal. The next chapter makes a detailed examination of the essential input/output (I/O) functions.

WHAT DID YOU LEARN?

A loop generally operates by first initializing a *key variable*, testing it in the control expression and then updating it in the loop body.

A loop may run a finite or infinite number of times. An *infinite loop* is created if the control expression never turns false.

The **while** loop is an *entry-controlled* loop where the control expression is tested at the beginning of the loop.

The **do-while** loop is an *exit-controlled* loop where the control expression is tested at the end of the loop. This loop is guaranteed to execute at least once.

The **for** loop is an entry-controlled loop which uses three expressions to determine loop entry and exit. Any or all of these expressions can be dropped. An infinite loop is created by dropping all expressions.

Some of the statements in the loop body can be moved to the control expression.

All loops can be *nested*. For every iteration of the outer loop, the inner loop iterates multiple times.

A loop can be prematurely terminated with **break** and a new iteration can be initiated with **continue**. In both cases, the remaining statements of the loop are ignored.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

8.1 The **while** and **for** loops may not be entered at all.

8.2 Every loop is designed to eventually terminate.

8.3 All control expressions that work with **if** will also work with **while**.

8.4 Every **for** loop can be replaced with a **do-while** loop.

8.5 All three components of the control expression of a **for** loop are evaluated at the beginning of every iteration.

8.6 Every **while** loop can be replaced with a **for** loop.

8.7 The presence of all three components, *exp1*, *exp2* and *exp3*, in the control expression of a **for** loop is mandatory.

8.8 In a nested loop structure, the outer loop has a one-to-many relationship with the inner loop.

A2. FILL IN THE BLANKS

8.1 The _____ keyword inside a loop terminates the loop, while the _____ keyword resumes the next iteration.

8.2 **while** and **for** are _____ loops but **do-while** is an _____ loop.

8.3 The statements **while (1)** and **for (;;)** signify an _____ loop.

8.4 The statement **for (; scanf("%d", &x) == 1;)** will run an infinite loop as long as the input to **scanf** is an _____.

8.5 A _____ statement in the inner loop of a nested loop will terminate the program.

A3. MULTIPLE-CHOICE QUESTIONS

8.1 Pick the odd item out: (A) **if-else**, (B) **while**, (C) **for**, (D) **do-while**.

8.2 A loop terminates when the control expression evaluates to (A) true, (B) false, (C) 0, (D) B and C.

8.3 If `x = 0`, the construct **`while (x++ < 5)`** will execute (A) 0 times, (B) once, (C) 4 times, (D) 5 times.

8.4 If `x = 0`, the construct **`while (++x < 1)`** will execute (A) 0 times, (B) once, (C) 4 times, (D) 5 times.

8.5 In the following code segment, how many times is the **for** loop executed?

```
for (i = prod = 0; (prod = i * i <= 25); i++)
   printf("hello\n");
```

(A) 4, (B) 5, (C) 6, (D) infinite.

8.6 For a nested loop, a **break** in the innermost loop (A) terminates the program, (B) terminates the innermost loop, (C) terminates all loops, (D) none of these.

CONCEPT-BASED QUESTIONS

8.1 Why are the inner set of parentheses necessary for the following construct?

```
while ((c = getchar()) != EOF)
```

8.2 Explain why the statement **`while (printf("Name: "))`** runs an infinite loop. What is the value of the control expression here?

8.3 Explain whether the sequences **`for (;1;)`** and **`for (-5;1;-5)`** create an infinite loop.

8.4 Is this statement legal? If so, what is the output?

```
for (printf(""); printf("x"); printf(""))
   ;
```

PROGRAMMING & DEBUGGING SKILLS

8.1 Write a program using **while** that accepts real numbers from the keyboard, truncates their decimal portion (if any) and prints the sum of their squares. The loop must terminate when the sum has exceeded 100,000.

8.2 Write a program using **while** to print a rectangle whose dimensions and the line-drawing character are set by user input.

8.3 Write a program that uses a **while** loop to print the following 4-line pattern:

```
   a
  b b
 c c c
d d d d
```

8.4 Correct the following code snippet:

```
short i = 0;
do
   printf("hello\n");
   i++;
while i < 9
```

8.5 Write a *portable* program using **do-while** that uses **getchar** to accept a string of letters in either case from the keyboard and then reverses the case of the string. The program must be independent of the character set used by the machine.

8.6 Write a program that uses a **for** loop to print the following pattern:

```
   1
  2 3
 4 5 6
7 8 9 10
```

8.7 Develop a program using **while** (outer loop) and **for** (inner loop) to print all prime numbers between two integers that are input by the user. The program must repeatedly prompt for the pair of integers until the user keys in at least one zero.

8.8 Write a program that calculates the maximum and minimum values of a set of positive integers. The program should prompt for each integer and print the two values after the user enters 0 or after 10 integers have been input.

8.9 Write a program using a **for** loop that determines the number of students from user input before accepting the total marks for each student. Using the guide A: 90-100, B: 80-89, C: 60-79, D: 0-59, print a statement that shows the number of students placed in each of these categories.

8.10 Write a program using **while** that converts a binary integer to decimal without using an array.

8.11 Design the skeleton of a program that uses a nested loop. For a certain condition (say, `x = 0`), control must break out of both loops using **break** twice. Is the **goto** statement a better alternative here?

9 Terminal Input/Output Functions

WHAT TO LEARN

- How *end-of-file (EOF)* and *buffering* issues affect the behavior of input/output functions.
- Fetching and writing a character with the **getchar** and **putchar** functions.
- Performing the same tasks with the **getc**/**fgetc** and **putc**/**fputc** functions.
- Converting text files between UNIX and MSDOS systems.
- How the special files—*stdin*, *stdout* and *stderr*—are used with programs.
- Reading and formatting character data with **scanf**.
- Printing individual data items as a character string with **printf**.

9.1 I/O FUNCTION BASICS

Unlike other languages, C depends on functions for carrying out even the most trivial tasks. C supports no keywords for taking input from the keyboard or for displaying output on the terminal. Functions that carry out input/output operations consider data as a stream of *characters* irrespective of the data type they actually represent. Thus, some of the I/O functions discussed in this chapter have to perform type conversion of data before they reach the destination.

Most I/O functions have to contend with two issues—EOF and buffering of data. Input functions often need to check for an *end-of-file (EOF)* condition. The term "file" is misleading because EOF actually indicates the end of data. There's no character named EOF in the ASCII character set (Appendix B). For keyboard input, UNIX/Linux systems use *[Ctrl-d]* and MSDOS/Windows systems use *[Ctrl-z]* to *indicate* EOF to a program.

Functions like **getchar** and **scanf** are often used in a loop which is terminated on EOF, i.e., when no further input is forthcoming or no error is encountered:

```
while ((c = getchar()) != EOF)
while (scanf("%d", &day) != EOF)
```

In a program, EOF is represented as a symbolic constant defined in `stdio.h`. Since any value between 0 and 255 is a valid return value for **`getchar`** and **`scanf`**, ANSI requires EOF to have a value that is *outside* the range of the character set of the machine. Typically, it is -1 but ANSI doesn't specify that it must be so.

The second issue is related to the buffering of data. A *buffer* is a block of memory allocated for temporary storage of data. When you input data to a program, say, using **`scanf`**, the data is first kept in this buffer before it is transmitted to the program. Similarly, when a program outputs data, say, using **`printf`**, the data is first written to a buffer before it is transferred to the output device (Fig. 9.1). Both operations require *flushing* of the buffer, the mechanism by which the contents of the buffer are transferred to the final destination.

FIGURE 9.1 The Role of the Buffer in I/O Operations

An input buffer can contain more data than the function needs in a single invocation. For instance, when you key in the string `yes`, four characters (including the trailing newline generated by *[Enter]*) are stored in the buffer. If your program needs only the `y`, then you must clear the remaining elements from this buffer. Clearing the buffer is an important task in C programming, and if you don't do that job properly, programs won't behave in the way they are meant to.

In this chapter, we will discuss some of the important I/O functions, many of whom have been used in previous chapters. These functions can be grouped into two categories:

- Character-handling functions (like **`getchar`**, **`fgetc`**, **`putchar`** and **`fputc`**).
- Formatted I/O functions (**`printf`** and **`scanf`**). These functions also perform conversion of data between character and other data types.

The other I/O functions will be taken up in subsequent chapters. Wherever necessary, we list, for each function, its prototype (i.e., syntax), the header file that must be included at the top of the program and the return value.

Takeaway: The term EOF signifies both a condition that is checked by the input functions as well as a character used by the operating system to indicate that condition. This is done by pressing *[Ctrl-d]* (UNIX/Linux) or *[Ctrl-z]* (MSDOS/Windows). The ASCII list doesn't include a character named EOF.

9.2 CHARACTER INPUT WITH `getchar`

Prototype: `int getchar(void);`
Header file: `stdio.h`
Return value: The character read or EOF when end-of-file or error occurs.

HOW IT WORKS: The C Standard Library

The *C standard library* is a collection of *archives* and *header files*. An archive is a disk file that contains the precompiled object code of several functions. The prototypes of these functions are maintained in a set of header files (like `stdio.h`). The location of these files on disk are known to the compiler. When the preprocessor sees the angular brackets (<>) in **`#include <stdio.h>`**, it knows the possible places where the file `stdio.h` can be found.

The standard library, which provides most of our daily needs, is functionally divided into a number of sections. I/O-handling functions like **`printf`** and **`scanf`** use `stdio.h`. Arithmetic functions have their prototypes listed in `math.h`. The file `ctype.h` is used by functions that determine the class to which a character belongs. Every function discussed in this chapter needs `stdio.h`.

The **`getchar`** function fetches the next character from the input buffer that has previously been filled with keyboard input. It is a character-oriented function, which means it treats input (say, the string `123`) as three characters even if they represent a number. **`getchar`** uses no arguments (the word `void` in the prototype) and returns the machine value of this character as an `int`. There are mainly three uses of this function:

- *Pause a Program.* A call to **`getchar`** pauses a program which resumes operation after the user has pressed *[Enter]*.
- *Accept a single-character response.* In this situation, the return value of **`getchar`** is saved and later used for decision making by an **`if`** or **`switch`** statement:

```
int c;
c = getchar();
if (c == 'y') {
    ...
```

  If you key in `y` and press *[Enter]*, these two characters are saved by the input buffer. **`getchar`** fetches the `y` and returns it as an `int`, but it leaves behind the newline character in the buffer. If you run **`getchar`** again, *the program will not pause this time because the buffer is not empty*. However, this second call will remove the newline and clear the buffer.

- *Fetch a group of characters by running in a loop.* You are familiar with the following code segment that was seen in Section 8.4:

```
while ((c = getchar()) != EOF)
   putchar(c);                          Writes fetched character to display
```

  In this sequence, **`getchar`** fetches all characters from the buffer before it encounters EOF, when the loop terminates. Every fetched character is written to the terminal using the **`putchar`** function.

Takeaway: Whenever the input buffer needs to be read or cleared, **`getchar`** should be run in a loop. The return value must be saved if the fetched character is needed for further processing but not when it is simply cleared.

Note: `getchar` must return an `int`, and not `char`, because it can also return EOF which must not represent a valid character value. Even though zero signifies a false value, it is still a legitimate value (the NUL character) of the ASCII character set. Thus, the value 0 can't be used to determine whether the function has encountered EOF or an error. That's why EOF is usually defined as -1.

9.3 CHARACTER OUTPUT WITH `putchar`

Prototype: `int putchar(int c);`
Header file: `stdio.h`
Return value: The character written (*c*) or EOF on error.

The **`putchar`** function complements the role of **`getchar`** by writing one character to the terminal. This character is accepted as an argument of type `int`. The function is generally used in this simple way:

```
int c = 'A';                                    char data assigned to int—OK
putchar(c);
```

The argument to **`putchar`** is often taken from the return value of **`getchar`**. **`putcha`**r returns the same argument as an `int` or EOF on error. Unlike **`getchar`**, however, **`putchar`** is generally used for its side effect (i.e., for writing to the output device) and not for its return value. There are no buffer-related issues with **`putchar`** that can cause problems.

You'll find **`putchar`** quite useful to output non-printable characters. For instance, *[Ctrl-g]*, represented by the escape sequence `'\a'`, can be output by **`putchar`** to sound a beep:

```
putchar('\a');                                  Equivalent to putchar(7);
```

When dealing with single characters, **`getchar`** and **`putchar`** are preferred to their formatted cousins, **`scanf`** and **`printf`**. The other functions of this family (**`getc`**/**`putc`** and **`fgetc`**/**`fputc`**) also provide the same functionality as **`getchar`** and **`putchar`**. They are discussed later in the chapter.

9.4 `retrieve_from_buffer.c`: A BUFFER-RELATED ISSUE

When you input a string, say, `yes` (followed by *[Enter]*) in response to a **`getchar`** call, the string `yes\n` is sent to the buffer. However, **`getchar`** returns only the first character (`y`), while the rest of the string remains in the buffer. Program 9.1 shows how to retrieve the contents of the buffer by using **`getchar`** and **`putchar`** inside a loop.

The program first invokes **`getchar`** simply to pause the program. **`getchar`** empties the buffer by fetching the newline character without saving it. In the next invocation, **`getchar`** returns with the first character (`'A'`) of the two-word string. The remainder of the string (`mazon drones\n`) is now in the buffer. Repeated invocation of **`getchar`** and **`putchar`** inside the **`while`** loop fetches and displays these characters. The loop terminates when it encounters the `\n` at the end of the string. The input buffer is now empty.

```
/* retrieve_from_buffer.c: Uses getchar to retrieve the remnants
                           of a string from the buffer. */
#include <stdio.h>
int main(void)
{
   int c;
   printf("Press [Enter] to continue ...");
   getchar();              /* Gets newline generated by [Enter] */
   printf("Enter a string: ");
   c = getchar();          /* Only first character held in c ... */
   putchar(c);             /* ... remaining characters still in buffer */
   printf("\nExtracting remaining characters from buffer ...\n");
   while ((c = getchar()) != '\n')
      putchar(c);
   putchar ('\n');         /* Takes cursor to next line */
   return 0;
}
```

PROGRAM 9.1: **retrieve_from_buffer.c**

```
Press [Enter] to continue ... [Enter]
Enter a string: Amazon drones
A                                          putchar prints first character from buffer
Extracting remaining characters from buffer ...
mazon drones
```

PROGRAM OUTPUT: **retrieve_from_buffer.c**

9.5 THE STANDARD FILES

You could be surprised to learn that **getchar** and **putchar** have been designed to work *only* with special files. The reason why they worked without problem in the previous program is that C (like UNIX) treats the keyboard, terminal and printer as *files*. Every C program has access to three special files that remain open for the entire duration of the program:

- *stdin*, the standard input file (default: keyboard).
- *stdout*, the standard output file (default: terminal).
- *stderr*, the standard error file (default: terminal).

These files have a *flexible* association with the default physical device. By default, *stdin* is associated with the keyboard, so a program that reads standard input takes input from the keyboard. Likewise, a write to *stdout* shows up on the terminal.

UNIX/Linux and MSDOS/Windows systems support the < and > symbols which are used to *change* the default files. For instance, a program containing **getchar** or **scanf** can use the feature of *input redirection* to take input from the file `foo1` instead of the keyboard:

```
aout < foo1                                    Use a.out for Linux
```

When used in this way, the program no longer pauses because it obtains its input from `foo1`. In a similar manner, a program containing **`putchar`** or **`printf`** can use the feature of *output redirection* to save its output to a disk file:

```
aout > foo2
```

Programs that use the < and > redirection symbols have to be executed from the shell prompt. The next program uses the services of the operating system to read one file and write another one without using a file-related function (like **`fopen`**, **`fread`**, etc.).

9.6 `unix2dos.c`: PROGRAM TO CONVERT A UNIX FILE TO MSDOS FORMAT

To understand this program, we need to know how text files are represented in different operating systems. A *text* file contains a sequence of printable characters grouped into *lines*, where each line is terminated by a platform-specific character sequence:

- CR-LF for MSDOS/Windows systems
- LF for UNIX/Linux systems
- CR for Mac OS X (Apple computers)

CR (carriage return—ASCII value 13) is represented by the escape sequence `\r`. LF (linefeed or newline—ASCII value 10) is the same as `\n`. Program 9.2 uses these escape sequences to convert a UNIX text file to MSDOS format. It copies the contents of the input file except that it writes a CR before it writes LF.

```
/* unix2dos.c: Converts a UNIX file to MSDOS.
                        Writes a CR (\r) before every LF (\n). */
#include <stdio.h>
int main(void)
{
   int c;
   while ((c = getchar()) != EOF) {
      if (c == '\n')              /* If character is LF ... */
         putchar('\r');           /* ... write CR first */
      putchar(c);                 /* Write every character seen in input */
   }
   return 0;
}
```

PROGRAM 9.2: **`unix2dos.c`**

This program must be run from the command prompt using the < and > redirection symbols to take input from the file `foo1` and write the output to `foo2`. Populate a file named `foo1` on a UNIX/Linux system with some text in multiple lines and then run the C executable:

```
$ foo1 < a.out > foo2          Linux shell; $ is the prompt
C:\> foo1 < aout > foo2        Windows shell; C:> is the prompt
```

Now compare the file sizes of `foo1` and `foo2` using the **`ls -l`** or **`DIR`** command. `foo1` should be smaller than `foo2`—by one character for each line. Even though **`getchar`** took input from `foo1` and **`putchar`** wrote to `foo2`, the program itself makes no mention of these two files. The connections to `foo1` and `foo2` are made by the operating system and not by the program.

Takeaway: UNIX/Linux systems use LF as the line terminator, while MSDOS/Windows systems use CR-LF. Thus, a text file comprising ten lines becomes smaller by 10 characters when moved from Windows to Linux after conversion.

Note: All operating systems may not support the redirection feature that use the < and > symbols. These symbols are not part of the C language and are not seen by the program when it executes.

9.7 OTHER CHARACTER-I/O FUNCTIONS

The **`getchar`** and **`putchar`** functions were designed to work *only* with `stdin` and `stdout`, respectively. The standard library supports another two sets of functions that also read and write a character:

- **`fgetc`** and **`fputc`**
- **`getc`** and **`putc`**

Unlike **`getchar`** and **`putchar`**, the preceding functions are meant to be used with *any* file—including a disk file. We'll now see how these functions handle `stdin` and `stdout` and wait until Chapter 15 to know how they handle disk files.

9.7.1 The fgetc and fputc Functions

I/O functions that have the "f" prefix are designed to work with files. Both **`fgetc`** and **`fputc`** use an argument that identifies the file. When used with the arguments, `stdin` or `stdout`, these functions replicate the behavior of their **`getchar`**/**`putchar`** equivalents, as shown in the following:

```
c = fgetc(stdin); —— Same as c = getchar();
fputc(c, stdout); —— Same as putchar(c);
```

Program 9.3 converts the case of letters using the **`fgetc`**/**`fputc`** duo and the **`toupper`** function. The latter belongs to a family of functions that determines the class of a character (like alphabetic, digit, etc.). Every function of this family needs the header file `ctype.h`.

9.7.2 The getc and putc Macros

C also supports the **`getc`** and **`putc`** *macros*, which replicate the functionality of the **`fgetc`** and **`fputc`** functions. (We'll explain macros in a bit.) Both sets use the same type and number of arguments and return the same values. The previous program will run normally when **`fgetc`** and **`fputc`** are replaced with **`getc`** and **`putc`**, respectively.

```
/* lower2upper2.c: Uses fgetc and fputc and makes use of the
                   toupper function to convert text to uppercase. */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int c;
   while ((c = fgetc(stdin)) != EOF)
      fputc(toupper(c), stdout);
   fputc('\a', stdout);                              /* The BELL character */
   return 0;
}
```

PROGRAM 9.3: **lower2upper2.c**

```
redmi note 4
REDMI NOTE 4
iphone 6 plus
IPHONE 6 PLUS
[Ctrl-d] or [Ctrl-z]                                 EOF
... Terminal beeps ...
```

PROGRAM OUTPUT: **lower2upper2.c**

A C *macro* is an alias that behaves like a function without incurring the cost of a function call. It is defined using the **#define** preprocessor directive. The downside to functions is that they use a memory *stack* for its arguments and variables *(11.13.2—How It Works)*. But since a macro doesn't use this stack, the **getc** macro often runs faster than the **fgetc** function. You should also know that **getchar** and **putchar** are also implemented as macros. Macros are examined in Chapter 17.

Tip: Use **getc**/**putc** or **fgetc**/**fputc** if you need to handle disk files for I/O. For performing I/O with the keyboard and terminal, use **getchar**/**putchar**.

9.7.3 The ungetc Function

The **ungetc** function reverses the action of **getc** by putting back the most recently read character to the input stream. This may appear queer but there are situations when you need to examine the next character before you decide what to do with the current one. The next read operation on the character stream will then fetch the character that was just pushed back. The **ungetc** function for `stdin` is used in this way:

```
ungetc(c, stdin);
```

How do you replace the CR-LF line terminator with LF for converting an MSDOS file to UNIX? The correct technique is to look ahead: when you encounter CR, fetch the next character. If this character is not LF, use **ungetc** to push it back to the input stream so it can be read normally. Program 9.4 achieves this task with the **getc** and **putc** macros.

```
/* dos2unix.c: Converts an MSDOS file to UNIX.
                        Removes the CR character before every newline. */
#include <stdio.h>
int main(void)
{
   int c;
   while ((c = getc(stdin)) != EOF) {
      if (c == '\r')                          /* If \r, then ... */
         if ((c = getc(stdin)) != '\n')       /* ... check next character */
            ungetc(c, stdin);                 /* Put it back if not \n */
         else
            putc('\n', stdout);
      else
         putc(c, stdout);
   }
   return 0;
}
```

PROGRAM 9.4: **dos2unix.c**

Since we are using `stdin` and `stdout`, we need to adopt the redirection technique discussed previously to execute the program:

```
aout < foo1 > foo2                                    a.out for Linux
```

After you have read Chapter 15, you should be able to modify this program and its inverse counterpart (**unix2dos.c**) to handle disk files directly without using the < and > symbols. Unlike `stdin` and `stdout`, disk files have to be opened explicitly to be read or written. You'll find these programs useful for moving files across diverse platforms.

The functions and macros that we have discussed so far read and write data as *characters*. They cannot identify or handle specific data types in the character stream. The rest of this chapter examines the **scanf** and **printf** functions that convert character data into specific data types and vice versa. Though both functions have been used extensively in previous chapters, there's a lot that we still need to know about them.

9.8 FORMATTED INPUT: THE scanf FUNCTION

Prototype: `int scanf(`*cstring*`, &`*var1*`, &`*var2*`, ...);`
Header file: `stdio.h`
Return value: Number of items successfully matched or EOF if end of input is encountered before conversion of first item.

scanf is the main member of a family of input functions that includes **fscanf** and **sscanf**. Unlike most functions, **scanf** uses a *variable* number of arguments. The first argument (*cstring*) is a *control string* containing one or more *format specifiers* (like `%d`). The remaining arguments (&*var1*, &*var2*, etc.)

represent—not variables—but their *addresses*. The number of variable addresses must equal the number of format specifiers. Figure 9.2 shows a **scanf** statement used with three format specifiers and three variable addresses.

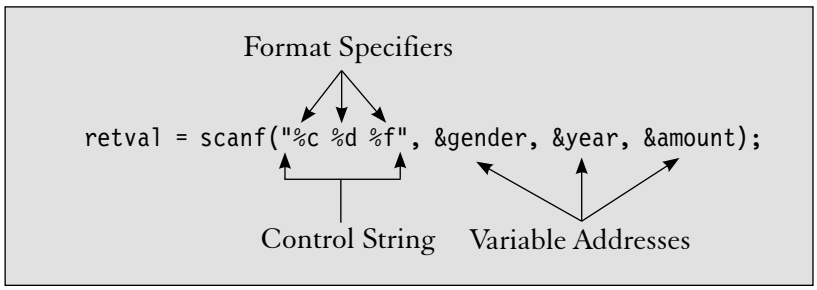


FIGURE 9.2 Components of **scanf**

scanf uses the information in the control string to divide the input into one or more *fields*. Thus, if the control string is `"%c %d %f"` and **scanf** encounters the string `M 1986 37.4`, it performs the following tasks:

- Saves the character `M` (`%c`) to the variable `gender` without performing any conversion.
- Converts the string `1986` to the integer 1986 (`%d`) and saves it in the variable `year`.
- Converts the four-character string `37.4` to the floating point value 37.4 (`%f`) and saves it in the variable `amount`.

If **scanf** encounters a mismatch between the type specified in the format specifier and that found in the data, it stops further processing. **scanf** then returns the number of items that were successfully read (if at all). The remaining characters, including the one that failed to match, are left in the buffer. The next invocation of **scanf** or any other input function will fetch these characters. It is often necessary to run **getchar** in a loop for clearing this "unclean" buffer.

Caution: If you use `x` instead of `&x` as a **scanf** argument, **scanf** will interpret the current value of `x` as an address and will overwrite the contents of memory at that address. The behavior of the program then becomes unpredictable; it can even crash.

Program 9.5 runs **scanf** in a loop to accept a `char`, `int` and `float` value from standard input. **scanf** successfully matches the input in the first two loop iterations but fails in the next two. The buffer is cleared before the next iteration using a technique that was discussed in Section *8.4.2*.

scanf can easily frustrate a user if its data-matching rules are not fully understood, so let's examine each invocation separately:

Invocation 1—Matching is perfect.

Invocation 2—**scanf** successfully matches the integer `4096` with `%f`.

Invocation 3—**scanf** truncates `3.142` when matching it with `%d`. `3` is wrongly assigned to `i` and `0.142` to `f`.

```
/* intro2scanf.c: Accepts a char, int and float.
                  Shows how easy it is for scanf to fail. */
#include <stdio.h>
int main(void)
{
   char c; int i = 1; float f;
   while (i) {                        /* Loop terminates when i = 0 */
      printf("Enter a character, int and float: ");
      scanf("%c %d %f", &c, &i, &f);
      printf("You input %c, %d, %f\n", c, i, f);
      while (getchar() != '\n')       /* Clears the buffer */
         ;
   }
   return 0;
}
```

PROGRAM 9.5: **intro2scanf.c**

```
Enter a character, int and float: Y 12 3.142
You input Y, 12, 3.142000
```

```
Enter a character, int and float: Y 15 4096
You input Y, 15, 4096.000000
```

```
Enter a character, int and float: N 3.142 12.75
You input N, 3, 0.142000
```

```
Enter a character, int and float: Yes 4096 15.75
You input Y, 3, 0.142000                    Prints previous values of d and f
```

```
Enter a character, int and float: 0 0 0
You input 0, 0, 0.000000
```

PROGRAM OUTPUT: **intro2scanf.c**

Invocation 4—**scanf** assigns `Y` to `c` all right, but it is unable to match the remainder of the string (`es`) with `%d`. **scanf** thus returns after matching only one item and leaves the values in the variables `i` and `f` unchanged.

The inner **while** loop having a null body clears the buffer at the end of each iteration, so **scanf** always operates with an empty buffer. Why not comment these two lines and then run the program?

Takeaway: scanf truncates a real number when matching it with `%d` even though it can match an integer with `%f`. Also, **scanf** can't match multiple characters with `%c`.

9.9 scanf: THE MATCHING RULES

A number of rules determine the way **scanf** matches input values with format specifiers. Most of the rules also have exceptions. By default, **scanf** splits input on whitespace (space, tab and newline) even if the control string doesn't contain whitespace. Before doing anything else, you must commit the following rule to memory:

Whitespace in the control string matches zero or more leading whitespace characters in the input.

The emphasis here is on the word "leading." This means that both `%d%d` and `%d %d` match the strings `12 45` and `12 45`. However, this rule doesn't apply to `%c`, whose matching techniques are discussed in Section 9.10.2.

There are two more issues that impact matching. First, if the control string contains non-whitespace characters, the same characters must be present in the input. For instance, `%d:%d:%d` matches `12:30:45` and `12: 30: 45` but not `12 :30 :45` because two values contain trailing whitespace.

Second, if a field width is specified (as in `%2d`), **scanf** tries to match two characters of the specified data type. Thus, `%2d%2d%2d` splits the string `311214` into the values `31`, `12` and `14` and saves them in three `int` variables. That's how one can extract the components of a date from a six-character string containing digit-characters.

Tip: For easier matching, provide whitespace on both sides of the non-whitespace character in the control string. Thus, `%d : %d` matches `12:34`, `12 :34`, `12: 34` as well as `12 : 34`. The same is true for `%c` and `%f`.

9.9.1 Mismatches and Return Value

scanf returns the number of items successfully matched and assigned to variables. For example, if the control string `%4d%c%f` successfully matches the input, **scanf** returns the value 3. If a mismatch occurs, **scanf** returns the number of items successfully matched until the occurrence of the mismatch. However, if EOF (the condition) occurs before the first item is matched, **scanf** returns EOF (the value—usually -1).

Mismatches can also occur for other reasons. If **scanf** finds a letter when a digit was expected, it returns the character to the buffer before the function itself returns to the caller. Wrong field width is also a possible cause of mismatch. When using `%4d` to match a six-digit number, **scanf** will only match the first four digits. Even if a subsequent call can read the remaining two digits, it would be reading the wrong item. Both of these situations are demonstrated in the following examples:

Control string: `"%4d %d"`

Input String	*Matches*	*Return Value*	*Remarks*
`1234 5678`	`1234 5678`	2	Nothing in buffer
`123456 789`	`1234 56`	2	`789` remains in buffer
`123r 5678`	`123`	1	`r 5678` remains in buffer
`r123 5678`	Nothing	0	Entire string remains in buffer

Takeaway: **scanf** returns 0 if it fails to match any item. But if **scanf** encounters EOF before reading the first item, it returns the value of EOF (usually -1). It is better to specify EOF (rather than -1) in your programs.

9.9.2 scanf_retval.c: Program to Extract Numbers from a String

Program 9.6 extracts integers embedded in an input character stream and adds them. It shows the skillful exploitation of the return value in handling every possible mismatch that may occur. As the output shows, as long as there are numerals embedded in a string, they will be extracted, converted and added. It just doesn't matter what the other characters are.

This program which terminates when EOF is sent from the keyboard, checks for three possible return values: 1, 0 and EOF (-1). **scanf** returns 1 when it successfully separates an integer from the rest of the string. It successfully extracts 3 from 3-three and 5 from 5(five). When **scanf** meets the non-numeric characters, it returns 0. Because **getchar** runs in a loop, it has no problem in ridding the buffer of these troublemakers.

```
/* scanf_retval.c: Uses scanf to add integers embedded in a string.
                   Employs the return value to handle mismatches. */
#include <stdio.h>
int main(void)
{
   short num, total = 0;
   int retval;

   printf("Enter integers to sum, EOF to quit: ");

   while ((retval = scanf("%hd", &num)) != EOF)
      if (retval == 1)                        /* Right input */
         total += num;
      else if (retval == 0)                   /* Wrong input ... */
         getchar();                           /* ... so clear buffer */

   printf("Total: %d\n", total);

   return 0;
}
```

PROGRAM 9.6: **scanf_retval.c**

```
Enter integers to sum, EOF to quit: 1 3 5 [Ctrl-d]
Total: 9
Enter integers to sum, EOF to quit: Add 3 to 5 and also 100. [Ctrl-d]
Total: 108
Enter integers to sum, EOF to quit: "1", 3-three & 5(five) [Ctrl-d]
Total: 9
Enter integers to sum, EOF to quit: 1.3.5 [Ctrl-d]
Total: 9
```

PROGRAM OUTPUT: **scanf_retval.c**

9.10 scanf: THE MAJOR FORMAT SPECIFIERS

scanf formats the input stream using format specifiers in the control string. A format specifier begins with a % followed by a *conversion code* (like the d in %d). The following generalized form also shows the optional components between the % and the conversion code:

% * *field width* *modifier* *conversion code*

The specifier %*4hd contains all optional components—the asterisk, field width (4) and modifier (h). These components have the following significance:

- The * flag is used for skipping a field.
- The *field width* specifies the *maximum* number of characters to be matched. Thus, %4d matches at most four digit-characters.
- A *modifier* changes the data type of the variable. You have used most of these modifiers on several occasions: h, l, L or ll.

An exhaustive list of format specifiers (including modifiers and a flag) used by **scanf** is shown in Table 9.1. We will now discuss the major format specifiers.

TABLE 9.1 Format Specifiers Used by **scanf**

Integer	**short**	**int**	**long**	**long long**
Signed Decimal	%hd	%d	%ld	%lld *(C99)*
Unsigned Decimal	%hu	%u	%lu	%llu *(C99)*
Octal	%ho	%o	%lo	%llo *(C99)*
Hex	%hx	%x	%lx	%llx *(C99)*
Decimal/Octal/Hex	%hi	%i	%li	%lli *(C99)*
Floating Point	**float**	**double**	**long double**	
Normal or exponential	%f, %e, %g	%lf, %le, %lg	%Lf, %Le, %Lg	
Synonyms	%F, %E, %G	%lF, %lE, %lG	%LF, %LE, %LG	
Other Format Specifiers	***Matches***			
%c	Character (char)			
%s	Character string comprising a single word			
%p	Pointer			
%n	Nothing, but saves count of characters matched in a variable			
%%	%			
%[...] *(scan set)*	Any character in list			
%[^...] *(scan set)*	Any character not in list			
Flag	***Significance***			
*	Skips a field			

9.10.1 Handling Numeric Data (%d and %f)

Integer and floating point data are represented by the conversion codes `d` and `f`, respectively, with or without modifiers. Note the following attributes of their format specifiers:

- `%d` matches `7`, but it truncates `7.25` and leaves the string `.25` in the buffer.
- `%f` matches both an integer and a real number expressed in decimal or exponential notation. Thus, it matches `7`, `713.5` and `7.135E2`. In this respect, `%f` differs with its peer in **`printf`**.

A field width *n* matches up to *n* digits of an integer. For a floating point number, *n* includes the decimal point. So, when matching `%5f` with 3.14285, **`scanf`** selects `3.142` and returns `85` to the buffer. Unlike with **`printf`**, it is not possible to specify the width of the decimal portion by using, say, `%6.2f`.

Caution: Never use **`scanf`** with `%d` or `%u` to read a floating point number. **`scanf`** reads the integral part and returns the decimal portion to the buffer.

9.10.2 Handling Character Data (%c)

`scanf` uses `%c` to match a single character *which can be a whitespace character*. If you enter `y` and *[Enter]* in response to **`scanf("%c", &x)`**, the `\n` remains in the buffer. If the next **`scanf`** call uses `%d` or `%f`, the `\n` (which is now leading whitespace) will be ignored. But if the specifier is another `%c`, then the second **`scanf`** call will match the `\n` and return without pausing. That's why the buffer must be cleared between successive runs of **`scanf`** with `%c`.

While `%d%d%d` or `%f%f%f` ignores leading whitespace, `%c%c%c` does not. Surprisingly, `%c%c%c` matches three or more spaces, the string `abc`, but not `a b c`. However, `%c %c %c` matches `abc`, `a b c`, `a□□b□□c` (two spaces separating the characters), but it doesn't match only spaces. Table 9.2 shows the use of `%c` in matching one or more characters. The last three examples show the effect of using non-whitespace characters in the format specifier.

TABLE 9.2 Matching `char` Data Type in **`scanf`**

Format Specifier	*Input String*	*Matches*
`"%c"`	`y`	`y` (`\n` in buffer)
`"%c"`	`yes`	`y` (`es\n` in buffer)
`"%c%c%c"`	`abc`	`abc`
`"%c%c%c"`	`a b c`	`a b` (space, `c\n` in buffer)
`"%c %c %c"`	`abc`	`abc`
`"%c %c %c"`	`a b c`	`a b c`
`"%c:%c:%c"`	`a:b:c`	`a:b:c`
`"%c : %c : %c"`	`a:b:c`	`a:b:c`
`"%c : %c : %c"`	`a : b : c`	`a : b : c`

9.10.3 Handling String Data (%s)

We know a string as an array of characters terminated by NUL *(5.15)*, but that is not how **scanf** understands a string. To **scanf**, a string is simply a *token* or *word*, i.e. a set of one or more non-whitespace characters. **scanf** reads this word with the %s format specifier and the name of the array as its matching argument. Thus, if you have defined an array named stg as

```
char stg[20];                                   Array must be of type char
```

then you can fill up the array by using **scanf** like this:

```
scanf("%s", stg);                               stg not &stg
```

scanf ignores leading whitespace and reads until it encounters a whitespace character or EOF. It saves the word in the array stg and tags a NUL character at the end. This means that the array must have an extra element for NUL. Note that **scanf** doesn't use the & prefix when %s is used. The name of the array itself provides the address that **scanf** needs.

The word-based approach used by **scanf** makes it unsuitable for reading a multi-word string. To match three words, **scanf** needs three %s specifiers (a single one for **printf**) and the addresses of three arrays. There are two solutions to this problem and one of them is discussed in Section 9.11.1.

Takeaway: printf and **scanf** differ in their treatment of %s. This specifier matches a single word in **scanf** but one or more words in **printf**.

9.10.4 scanf_char_string.c: Characters and Strings as Input

Program 9.7 uses %c and %s to handle character and string input. It also uses the field width to split a string into three. The **getchar** sequence is used twice to clear the buffer, so it makes sense to use a macro (FLUSH_BUFFER) in this program.

Because whitespace in the control string ("%c %c %c") matches zero or more occurrences of leading whitespace in the data, we could key in both ABC and a b c. The final invocation of **scanf** uses "%3s%3s%3s" to split the string sepoctnov into three separate strings. Each array has an extra byte to let **scanf** append the NUL at the end.

9.11 scanf: OTHER FEATURES

Our coverage of the **scanf** function would not be complete without a discussion on the (i) the scan set, (ii) the only flag supported by it, and (iii) some more format specifiers. These topics are examined in the following sections.

9.11.1 The Scan Set Specifier

The *scan set* specifier comprises one or more characters enclosed by [and]. For instance, %[aeiou] represents a scan set that matches only vowel characters. However, a caret at the beginning of the

```
/* scanf_char_string.c: Uses the %c and %s format specifiers to read
   individual characters and split a string into three. */
#include <stdio.h>
#define FLUSH_BUFFER while (getchar() != '\n');

int main(void)
{
   char c1, c2, c3;
   char month1[4], month2[4], month3[4];

   while (printf("Enter three characters: ")) {
      if (scanf("%c %c %c", &c1, &c2, &c3) == EOF)
         break;
      FLUSH_BUFFER
      printf("ASCII values: %c = %d, %c = %d, %c = %d\n",
             c1, c1, c2, c2, c3, c3);
   }

   printf("\nEnter 3 month names as a nine-character string: ");
   scanf("%3s%3s%3s", month1, month2, month3);
   FLUSH_BUFFER
   printf("Months split as %s | %s | %s\n", month1, month2, month3);

   return 0;
}
```

PROGRAM 9.7: **scanf_char_string.c**

```
Enter three characters: ABC
ASCII values: A = 65, B = 66, C = 67
Enter three characters: a b c
ASCII values: a = 97, b = 98, c = 99
Enter three characters: [Ctrl-d]
Enter 3 month names as a nine-character string: sepoctnov
Months split as sep | oct | nov
```

PROGRAM OUTPUT: **scanf_char_string.c**

set reverses its usual meaning. Thus, `%[^aeiou]` matches only non-vowel characters. A scan set can include escape sequences and ranges (like `0-9`) as shown in the following examples:

```
char stg[10];
scanf("%[aeiou]", stg);          Matches only vowels.
scanf("%[0-9a-zA-Z]", stg);      Matches only alphanumeric characters.
scanf("%[^,]", stg)              Matches all characters except the comma.
scanf("%[^\n]", stg);            Matches an entire line—useful feature.
```

scanf stops scanning the moment it encounters a character not belonging to the set. It automatically pads a NUL at the end of the array which must be declared to include one extra element. The last two examples are made use of in Program 9.8 which splits and manipulates strings input from the keyboard.

```
/* scanf_char_string2.c: Uses the %c and %s format specifiers
                         and the scan set to rearrange text. */
#include <stdio.h>
#define FLUSH_BUFFER while (getchar() != '\n');

int main(void)
{
   char c1, c2, c3;
   char month1[4], month2[4], month3[4], lastname[20], firstname[20];

   while (printf("Enter three characters: ")) {
      if (scanf("%c %c %c", &c1, &c2, &c3) == EOF)
         break;
      FLUSH_BUFFER
      printf("ASCII values: %c = %d, %c = %d, %c = %d\n",
                c1, c1, c2, c2, c3, c3);
     }

   printf("\nEnter 3 month names as a nine-character string: ");
   scanf("%3s%3s%3s", month1, month2, month3);
   FLUSH_BUFFER
   printf("Months split as %s | %s | %s\n", month1, month2, month3);

   printf("Last name, First Name: ");
   scanf("%[^,], %[^\n]", lastname, firstname);
   printf("Name is %s %s\n", firstname, lastname);

   return 0;
}
```

PROGRAM 9.8: **scanf_char_string2.c**

```
Enter three characters: ABC
ASCII values: A = 65, B = 66, C = 67
Enter three characters: a b c
ASCII values: a = 97, b = 98, c = 99
Enter three characters: [Ctrl-d]
Enter 3 month names as a nine-character string: sepoctnov
Months split as sep | oct | nov
Last name, First Name: Ritchie, Dennis
Name is Dennis Ritchie
```

PROGRAM OUTPUT: **scanf_char_string2.c**

scanf first runs in a loop and uses the control string `"%c %c %c"` to match both `ABC` and `a b c`. It then uses `"%3s%3s%3s"` to split a nine-character word into three components. Finally, **scanf** uses two different scan sets to split a line containing two comma-delimited strings. The first scan set matches a group of characters not containing a comma, while the second one matches all characters except the newline. Together, they make it possible to reverse the first and last names.

Note: Even though **scanf** can read an entire line using the scan set [^\n], C also offers an alternative mechanism to handle this task using two functions—**fgets** and **sscanf**. **fgets** reads a line into an array of characters and **sscanf** splits this line into multiple fields. This important mechanism is examined in Chapter 13.

9.11.2 The * Flag

The * flag, placed between the % and the conversion code, skips a field. For instance, %*d skips a field of type int. As shown in the following, a skipped field doesn't have the address of the variable as argument:

```
int number; float amount;
printf("Key in an int, char and float: ");
scanf("%d%*c%f", &number, &amount);                    %*c skips char
```

This is a useful feature to have when using **scanf** (or **fscanf**) to read a file. If each line of a file has six fields and only four require to be read, we can use the * to skip two fields. The * has a different significance in **printf** *(9.13)*.

9.11.3 The %p, %n and %% Specifiers

The %p specifier matches a pointer, a special data type which is explored in Chapter 12. For now, it suffices to know that %p matches a number that represents a memory address.

The %n specifier doesn't match anything, but uses its matching argument to store in an int variable the number of characters that are input. This useful validation feature is used in the following example to check whether a six-character string has been input:

```
int count, num;
printf("Enter a 6-digit number: ");
scanf("%d%n", &num, &count);
if (count != 6)
   printf("Not a 6-digit number.\n");
```

Since the % has special significance in the control string, there must be a way to indicate to the program that the character has to be treated literally. This is done by repeating the symbol:

```
scanf("%d%%", &rate);                          Input number must have % suffix
```

This call expects a number to be input, followed by a % with or without whitespace separating them. If you need to match two %s, use %%%%.

The **scanf** cousins, **sscanf** and **fscanf**, will be examined in Chapters 13 and 15, respectively. Many of the format specifiers used by **scanf** are also used by **printf**, though not always with identical meaning. Even though **printf** doesn't have any unique format specifiers to offer, it supports a set of flags that are not used by **scanf**.

9.12 FORMATTED OUTPUT: THE `printf` FUNCTION

Prototype: `int printf(`*cstring*`, `*var1*`, `*var2*`, ...);`
Header file: `stdio.h`
Return value: Number of characters printed or a negative value on error.

`printf` is the primary member of a group of formatted output functions that also includes **`fprintf`** and **`sprintf`**. Like **`scanf`**, it uses a variable number of arguments where the first argument (*cstring*) represents a control string. If *cstring* contains one or more format specifiers, it must be followed by a matching number of *data items* (*var1*, *var2*, etc.). A data item can be a variable, expression or constant. Figure 9.3 shows a **`printf`** statement containing three format specifiers and three matching variables.

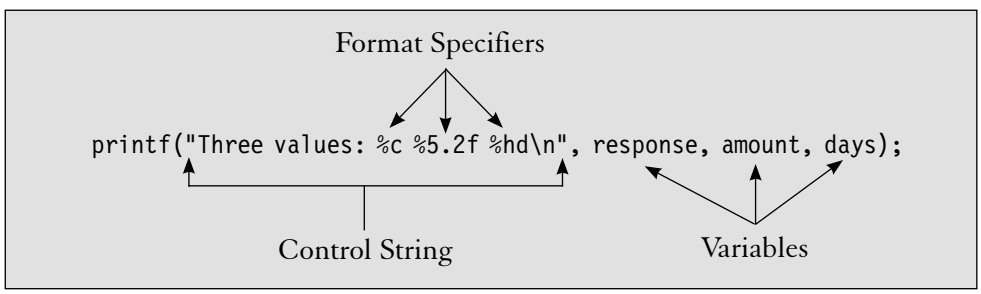


FIGURE 9.3 Components of **`printf`**

`printf` uses information in the control string to replace all format specifiers with values of their matching arguments. Because these values can be of different data types, they are converted to strings before replacement. The entire string is then displayed on the terminal. For instance, the **`printf`** statement in Figure 9.3 converts the floating point value of `amount` (`%5.2f`) and integer value of `days` (`%hd`) to characters before displaying them.

Like in **`scanf`**, a format specifier is a combination of `%` and a conversion code (like `d` or `f`). This code is optionally prefixed by a modifier to handle the data sub-types (like `h` in `%hd`) and field width (like `80` in `%80s`). **`printf`** supports all conversion codes and modifiers used by **`scanf`** (Table 9.3), sometimes with modifications. A **`printf`** format specifier can also include a *precision*.

The number of data items must equal the number of format specifiers. If they don't, the compiler generates only a warning. **`printf`** ignores the surplus data items but it prints junk values for surplus format specifiers:

```
printf("%d %d\n", x, y, z);          Doesn't print value of z
printf("%d %d\n", x);                Prints junk for the second %d
```

Mismatches in data types may also occur between format specifiers and values. The compiler merely issues a warning when you use `%d` to print a string or `%s` to print a number. Every data item that is backed by a format specifier is written even though their data types may not match.

9.12.1 printf and scanf Compared

printf and **scanf** share some common features. Both use a control string to determine how the data have to be converted. They also use the same format specifiers and modifiers (with few exceptions). However, they differ in a few ways:

- **scanf** converts a string to the intended data types, but **printf** does the reverse. For instance, when **printf** matches the number 23 with %d, it converts the integer 23 to the string 23.
- The control string in **printf** may not have any format specifiers at all. We often use **printf** to output a simple string (say, "Hello world\n").
- **printf** data items are variables, expressions or constants, but **scanf** uses only addresses. There's only one format specifier in **printf** that signifies an address (%n).
- Even though **printf** uses the same modifiers (h, l, ll and L), it uses %f and not %lf to print a double. The compiler converts a float argument to double, so %f is good enough to print both.
- Each of the floating point specifiers used in **scanf**—%e, %f and %g—matches a number expressed in decimal or exponential notation. **printf**, however, is specific: %f for numbers with a decimal point, %e for exponential notation, and %g to let the function choose the format.
- While %s in **printf** matches any string, the same specifier in **scanf** matches a *word* as a group of non-whitespace characters.
- A **printf** specifier supports a *precision* field (like .2 in %8.2f). Precision doesn't apply to **scanf**.
- **printf** supports numerous *flags* (like + in %+6d). **scanf** knows only one flag (the *, which skips a field).

printf is a simpler statement to use primarily because it has none of the whitespace-related problems that often frustrate users of **scanf**.

9.12.2 Printing Plain Text

The control string contains text to be printed, with or without format specifiers. **printf** treats whitespace literally and supports the escape sequences shown previously in Table 5.3. The following statement prints the control string which has two escape sequences but no format specifier:

```
printf("\tNo format specifier in this string\n");
```

For printing long text, **printf** supports two options but the one we have adopted in this text is to concatenate multiple strings by placing one after the other—on different lines, if necessary:

```
printf("It's not only printf that allows string concatenation. All string "
       "handling functions accept this mechanism of joining strings.\n");
```

Because the characters % and \ are treated specially, you can force them to be interpreted literally by repeating the character; %% prints a % and \\ prints a \:

```
printf("%% and \\ are special.\n");
```

Prints % and \ are special.

By now, you should be comfortable in using **printf** conversion codes and their modifiers. We'll thus focus on the field width, precision and flags in the rest of this chapter. But before doing that, let's take up a program which demonstrates some of the features that have been discussed.

TABLE 9.3 Format Specifiers and Flags Used by **printf**

Integer	**short**	**int**	**long**	**long long**
Signed Decimal	%hd	%d, %i	%ld	%lld *(C99)*
Unsigned Decimal	%hu	%u	%lu	%llu *(C99)*
Octal	%ho	%o	%lo	%llo *(C99)*
Hex	%hx	%x	%lx	%llx *(C99)*
Floating Point	**float**	**double**	**long double**	
Normal	%f	%f	%Lf	
Exponential	%e, %E	%le, %lE	%Le, %LE	
Exponential or normal	%g, %G	%lg, %lG	%Lg, %LG	
Other Format Specifiers	***Prints***			
%c	Character (char)			
%s	Character string (not restricted to one word)			
%p	Pointer			
%n	Nothing, but saves count of printed characters in a variable.			
%%	%			
Flag	***Significance***			
-	Left-justifies output.			
+	Prefixes a + to a non-negative number.			
Space	Prefixes a space to a non-negative number.			
#	Depends on conversion code: With o, prepends 0 to number. With x/X, prepends 0x or 0X to number. With g/G, prints trailing zeroes. With e/E/f/F/g/G, prints number with decimal point.			
0	Pads field with zeroes when its size is less than field width.			
*	Field width specified by corresponding argument.			

9.12.3 printf_special.c: A Second Look at the Data Types

This three-part program (Program 9.9) demonstrates some features of **printf** that may have been overlooked previously. The first part creates two overflow situations and shows the consequences of using an unequal number of arguments and format specifiers. An extra specifier in the first **printf** shows a value of zero, but i2 has been printed only once in the second **printf** statement. The compiler may issue two warnings here.

```
/* printf_special.c: A second look at printf conversion codes, overflow
                     situations, mismatches and escape sequences. */
#include <stdio.h>
int main(void)
{
   short s1 = 32767, s2 = s1 + 1;
   int i1 = 2147483647, i2 = i1 + 1;
   float f = 3.142857;
   printf("short: s1 = %hd, s2 = %hd, s3 = %hu\n", s1, s2);
   printf("int: i1 = %d, i2 = %d\n", i1, i2, i2);
   printf("float: f = %f, f = %e, f = %g\n", f, f, f);
   printf("Mismatch: Printing %hd using %%f: %f\n", s1, s1);
   printf("Mismatch: Printing %f using %%d: %d\n", f, f);
   printf("Repetition required for %%, \\n and \\t.%n\n", &i1);
   printf("Number of characters printed by previous printf:\n"
          "%i in decimal, %o in octal and %x in hex\n", i1, i1, i1);
   return 0;
}
```

PROGRAM 9.9: **printf_special.c**

```
short: s1 = 32767, s2 = -32768, s3 = 0
int: i1 = 2147483647, i2 = -2147483648
float: f = 3.142857, f = 3.142857e+00, f = 3.14286
Mismatch: Printing 32767 using %f: 2.000000
Mismatch: Printing 3.142857 using %d: 1073741824
Repetition required for %, \n and \t.
Number of characters printed by previous printf:
37 in decimal, 45 in octal and 25 in hex
```

PROGRAM OUTPUT: **printf_special.c**

The second part shows the effect of mismatches between the data types and their format specifiers. The compiler may issue two warnings here as well. The third part shows how to literally interpret the \ and %. Observe the use of the %n format specifier in displaying the number of characters output by the immediately preceding **printf** statement. This is the only specifier that signifies an address (&i1).

9.13 printf: USING FIELD WIDTH AND THE * FLAG

printf prints a field in a space that is wide enough to accommodate it. But it also uses a default width that can be changed. For instance, a value matched with %6d is printed in a space 6 characters wide unless the value is wider than 6 characters, in which case the actual width is used. The following examples show the effect of field width on three data types:

Format Specifier	*Value*	*Remarks*
`%3d`	`12345`	**`printf`** ignores 3 and uses a width of 5.
`%5d`	`12`	Prints 3 spaces before `12`.
`%4f`	`1234.5678`	Prints `1234.567749`.
`%14f`	`1234.5678`	Prints 3 spaces before `1234.567749`.
`%2s`	`"Dolly\n"`	**`printf`** uses entire width to print `Dolly`.

`printf` uses right-justification when the specified width is larger than the actual width. This means `"%5d"` prints `12` as □□□`12` (three spaces before `12`). Using a flag, we can left-justify the output. The third example (`%4f`) uses a width of 11 instead of the specified 4 because, by default, floating point numbers are printed with 6 decimal places. This default can be changed by using a precision, which is discussed in Section 9.14.

Sometimes, you won't know how large the number or string is, so you just can't specify a field width in advance. C solves the problem with the * flag and its matching variable. The * is typically used in this way:

```
printf("%*d", n, x);                    2 variables but only 1 format specifier
```

The `*` informs the compiler that the field width for the variable `x` is stored in the variable `n`, and will be determined only at runtime. You'll recall that the * in **`scanf`** has a totally different meaning even though in both cases the number of items don't match the number of format specifiers. For every * used in **`scanf`**, the number of items is one less, and for **`printf`**, it is one more.

Takeaway: For floating point numbers, the field width doesn't specify the number of decimal places to be printed. All floating point numbers are printed using six decimal places unless a precision is used.

9.14 `printf`: USING PRECISION

A format specifier can also specify a *precision* that follows the field width, if present. It is a combination of a dot and an integer that are optionally suffixed to the field width. For instance, the specifiers `%.3f` and `%20.10s` use a precision of 3 and 10, respectively. The significance of precision depends on the data type it is applied to. The following examples feature a ruler framed with a repeated set of 10 digits which you'll find helpful to check the alignment and width of the output.

9.14.1 Precision with Floating Point Numbers

For floating point values using `%e` or `%f`, precision signifies the number of decimal places. Thus, `%.2f` prints a value with 2 digits after the decimal point. Precision can also be combined with the field width. Thus, `%12.5f` specifies a minimum total width of 12 including the decimal point, with 5 digits after the decimal point. For `%g`, the precision signifies the maximum number of significant digits. The following examples show how precision and width control the formatting of real numbers:

float f = 3.1428571428;		
Format Specifier	*Output*	*Remarks*
Ruler ⟶	123456789012345	
%10f	3.142857	Total width 10.
%15f	3.142857	Total width 15.
%8.3f	3.143	Total width 8, 3 places after decimal.
%.10f	3.1428570747	10 places after decimal.
%.5e	3.14286e+00	5 places after decimal.
%.5g	3.1429	5 significant digits.

The first two examples specify a field width but no precision. Thus, both values have been printed using the default 6 decimal places. The remaining four examples use precision. The specifier %8.3f restricts the decimal places to 3 while maintaining the total width at 8. To print 10 digits after the decimal point, we need to use precision .10. Precision retains its usual meaning when using %e but not when %g is used. Note that %.5g prints 5 significant digits.

9.14.2 Precision with Integers

For integers, both precision and field width specify the minimum width of the field but with one difference. If the number is not large enough to meet the precision, leading zeroes are used for padding. For example, %.5d prints the number 25 as 00025. Consider the following examples:

int i = 4096;		
Format Specifier	*Output*	*Remarks*
Ruler ⟶	1234567890	
%2d	4096	Overrides specified field width.
%10d	4096	Prints with 6 spaces before number.
%.10d	0000004096	Zeroes used to maintain field width.
%10.8d	00004096	Zeroes and spaces used to maintain field width.

printf ignores %2d and uses the entire width to print the number 4096. But when the number is smaller than the specified width, it is printed right-justified within the overall field width. When only precision is used, all leading spaces are replaced with zeroes. However, when a field width is also specified, the difference between width and precision is made up with leading spaces to maintain the overall width.

9.14.3 Precision with Character Strings

For character strings, the precision simply specifies the number of characters to print. The specifier %.10s allows only the first 10 characters to be printed. When a field width is also included, say, %30.10s, the first 10 characters are printed, right-justified, using a total width of 30 places. These observations are highlighted in the following examples:

```
char stg[30] = "Significance of field width";
```

Format Specifier	*Output*
Ruler ⟶	12345678901234567890123456789012345678901234567890
%40s	Significance of field width
%.12s	Significance
%20.12s	Significance

When the field width is greater than the string length, leading spaces are used for padding. When precision .12 is used, only the first 12 characters of the string are printed. To adjust the position of this substring, we need to combine precision with field width. The third example uses %20.12s to print the first 12 characters, right-justified, within the overall width of 20. You can use this feature to extract a substring from a larger string:

```
char month[10] = "January";
printf("%.3s\n", month);                                    Prints Jan
```

It seems that the term "precision" used in C is somewhat of a misnomer except when it is used with a floating point number. However, the preceding examples show that both field width and precision are needed to control the appearance of the output.

9.15 printf: USING FLAGS

As if field width and precision were not enough, **printf** supports a number of *flags* which enable even finer control of the output. A flag is placed immediately after the % and before the field width (if present). If you need to left-justify output or pad it with zeroes instead of spaces, use a flag. As Table 9.3 lists all of them with adequate explanation, we'll discuss them briefly:

- *The - Flag* This flag left-justifies the output but this obviously makes sense when - is used with a field width. Thus, %-30s ensures that the string begins at column 1. If the string is shorter than 30 characters, trailing spaces are added to maintain the total field width at 30. We usually don't left-justify numbers.
- *The + Flag* You won't expect the + to right-justify output because that is the default anyway. This flag is interpreted literally; it adds a + before every positive number. C offers this feature to let a positive and negative number use the same width. For example, %+6d prints the number 123 as +□□□123.
- *The Space Flag* If seeing a + in every positive number seems awkward, you can use the space flag instead. For instance, the specifier %□d adds a space before a positive number but none before a negative one. Like the +, the space flag ensures that both 123 and -123 use the same width when printed.
- *The 0 Flag* This flag can only be used with numeric values. For integers, it behaves like precision; it pads the output with zeroes if required. So, %010d is the same as %.10d. The 0 is the only mechanism that lets you pad a floating point number with zeroes. Thus, %010.3f prints 123.456 as 000123.456.

- *The # Flag* This flag is used with octal and hexadecimal integers, as well as real numbers. It prepends a `0` and `0x` when used as `%#o` and `%#x`, respectively. For its effect on floating point numbers, look up Table 9.3. Note that the `#` has a dual role to play when used as `%#g` or `%#G`.

The significance of the * flag has already been discussed in Section 9.13. While the # bears no relation to the field width, and the * is used to determine it, the other flags are most effective when used with a field width. You'll find these flags useful when building tables with properly-aligned columns.

9.16 printf_flags.c: USING THE FLAGS

We round up our discussions on **printf** with Program 9.10, which illustrates the use of all flags. The flags are applied to three data types—`int`, `float` and the character string. The flag-bearing **printf** statements are numbered and references to these numbers are made in the explanations that follow. This program also contains a ruler against which you can check the width and alignment of the output.

The integers `256` and `-256` are first printed without applying a flag to show the default width they use (1 and 2). The negative sign in `-256` increases the width by one; this causes misalignment

```
/* printf_flags.c: Demonstrates use of the +, -, space, 0, # and * flags.
                   Adjust the position of the scale if necessary. */
#include <stdio.h>

int main(void)
{
   int i = 256, n = 0;
   float f = 3.142;
   char stg[30] = "Significance of flags";

   while (n++ < 26)
      printf(" ");
   printf("12345678901234567890123456789012345\n"
          "-----------------------------------------------------------\n");

   printf(" 1. No flags (%%d)\t: %d\n", i);
   printf(" 2. No flags (%%d)\t: %d\n", -i);
   printf(" 3. + flag (%%+d)\t: %+d\n", i);
   printf(" 4. Space flag (%% d)\t: % d\n", i);
   printf(" 5. Space flag (%% d)\t: % d\n", -i);
   printf(" 6. - flag (%%-10d)\t: %-10d|\n", i);
   printf(" 7. - flag (%%-30s)\t: %-30s|\n", stg);
   printf(" 8. 0 flag (%%010d)\t: %010d\n", i);
   printf(" 9. # flag (%%#x)\t: %#x\n", i);
   printf("10. # flag (%%#5.0f)\t: %#5.0f\n", f);
   printf("11. * flag (%%*d)\t: %*d\n", 15, i);

   return 0;
}
```

PROGRAM 9.10: **printf_flags.c**

```
                          123456789012345678901234567890123456
------------------------------------------------------------
 1. No flags (%d)        : 256
 2. No flags (%d)        : -256
 3. + flag (%+d)         : +256
 4. Space flag (% d)     :  256
 5. Space flag (% d)     : -256
 6. - flag (%-10d)       : 256       |
 7. - flag (%-30s)       : Significance of flags         |
 8. 0 flag (%010d)       : 0000000256
 9. # flag (%#x)         : 0x100
10. # flag (%#5.0f)      :    3.
11. * flag (%*d)         :            256
```

PROGRAM OUTPUT: **printf_flags.c**

in the output of these two integers. The next three statements apply the + and space flags to the same integers. When the + flag is used, a + is printed before 256 (3), which restores the alignment. The space flag simply prepends a space to a positive integer (4) but not a negative one (5).

The - flag changes the default justification to left (6 and 7). Aligning a value with the beginning column causes trailing spaces to be added if the specified field width is longer than the actual width. We have used the | as a marker to establish this for an integer and a string (6 and 7). When using **printf** to create tables, we normally use right-justification for numbers and left-justification for strings.

Finally, let's consider the 0, # and * flags. The 0 flag is used to prepend zeroes to maintain the field width, but it applies only to numbers (8). The # is responsible for printing a hexadecimal number with the 0x prefix (9). For a floating point number, the # is used with %f to compulsorily print a dot even if .0 precision is used (10). Finally, the * is supported by its matching argument, 15, to print 256 in a field width of 15 spaces (11). Since 15 can be replaced with a variable, we can design programs where the field width is determined only at runtime.

Note: The * is the only mechanism that lets you specify the field width as a variable.

WHAT NEXT?

The remaining I/O functions will be discussed after we have completely understood strings (an array of characters). We now need to learn to use an array to handle multiple data items of one type. That would prepare us for exploring the world of pointers and strings.

WHAT DID YOU LEARN?

Input functions need to properly handle EOF and buffering of data. Data not required by a program remain in a buffer which must be cleared before the next function call reads new data.

The **getchar** and **putchar** functions read and write one character in each invocation without converting the data.

C treats all devices as *files* and every C program has access to the files *stdin*, *stdout* and *stderr*. By default, these files are associated with the keyboard and terminal, but they can be changed using the < and > symbols of the operating system.

The **fgetc**/**fputc** functions can also perform the same functions as **getchar**/**putchar** but they use `stdin` and `stdout` as additional arguments.

The **getc**/**putc** *macros* can also be used for the same tasks. The **ungetc** function puts back a character to the input stream.

The **scanf** function uses *format specifiers* in a *control string* to read data items. The function specifies the memory addresses for these items. **scanf** reads only one word with `%s` and an entire line using `%[^\n]` as the *scan set specifier*.

scanf returns the number of items successfully read but it puts back unmatched data to the buffer.

The **printf** function also uses format specifiers in a control string, but it converts data into character form before printing. It can write a multi-word string using `%s`. Using field width, precision and flags, the formatting of data can be tightly controlled to meet any requirements.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

9.1 There is no character signifying EOF in the ASCII list.

9.2 A three-digit integer is fetched by **getchar** as three separate characters.

9.3 The **fgetc** and **fputc** functions work only with disk files.

9.4 The sequence **foo1 < aout.exe > foo2** works only on computers that have a C compiler.

9.5 If **scanf("%d", &x)** encounters the string `abcd1234`, it leaves the entire string in the buffer.

9.6 The statement **scanf("%f%%", &rate);** matches a real number followed by `%%`.

9.7 A **printf** function may not contain a format specifier.

9.8 The statement **printf("%8.3f", f);** prints the value of `f` in a space 8 characters wide with 3 digits after the decimal point.

A2. FILL IN THE BLANKS

9.1 EOF is usually defined as _____ in the file _____.

9.2 Keyboard input is first stored in a _____ before it reaches the program.

9.3 The precompiled object code of a library function is stored in an _____.

9.4 A UNIX text file is _____ than its converted MSDOS file.

9.5 **`printf`** and **`scanf`** use a _____ number of arguments.

9.6 `stdin`, `stdout` and `stderr` represent three _____ that remain _____ as long as the program is running.

9.7 The _____ function puts back a character into the input stream.

9.8 The format specifier `%[^\n]` is used by the _____ function to match an entire _____.

9.9 The `%s` format specifier matches a _____ in **`scanf`** and a _____ in **`printf`**.

A3. MULTIPLE-CHOICE QUESTIONS

9.1 The character representing EOF is (A) *[Ctrl-d]* for UNIX and *[Ctrl-z]* for MSDOS, (B) the same for MSDOS and UNIX, (C) *[Ctrl-z]* for UNIX and *[Ctrl-d]* for MSDOS, (D) *[Ctrl-z]* for both systems.

9.2 For converting a UNIX file to MSDOS/Windows format, the LF character must be (A) replaced with CF, (B) succeeded by CR, (C) preceded by CR, (D) succeeded by another LF.

9.3 The **`scanf`** function returns (A) the number of characters read, (B) the number of items successfully read, (C) 1 on success, (D) none of these.

9.4 The statement **`scanf("%f", price);`** (A) assigns a real number to `price`, (B) stores a real number at the address of `price`, (C) generates a compiler error, (D) interprets `price` as an address and attempts to store the input at the address.

9.5 The statement **`scanf("%d%*ld%f), &days, &price);`** (A) generates a compilation error, (B) generates a runtime error, (C) reads an `int` and `long` into `days` and `price`, (D) reads first and third data items into `days` and `price`, respectively.

9.6 Pick the odd item out: (A) `%f`, (B) `%u`, (C) `%d`, (D) `%ld`.

9.7 The **`printf`** function returns (A) the number of words written, (B) the number of characters written, (C) 1 on success, (D) none of these.

9.8 The statement **`printf("%f%f\n", f1, f2, f3);`** (A) generates a compilation error, (B) prints `f1` and `f2` only, (C) prints `f1`, `f2` and junk, (D) none of these.

A4. MIX AND MATCH

9.1 Match the line-terminating characters with the operating system:

(A) carriage return (CR), (B) linefeed (LF), (C) carriage return-linefeed (CR-LF).

(1) UNIX, (2) MSDOS, (3) Mac OS X

9.2 Match each format specifier with the corresponding data type:

(A) `%d`, (B) `%f`, (C) `%ld`, (D) `%[^\n]`, (E) `%o`.

(1) octal integer, (2) scan set, (3) real number, (4) signed integer, (5) long integer.

CONCEPT-BASED QUESTIONS

9.1 Why does the **`getchar`** function return an `int` instead of a `char`?

9.2 What does the following sequence achieve? What is the significance of the `;` at the end of the second line?

```
scanf("%c", &reply);
while (getchar() != '\n') ;
```

9.3 What is the limitation of the following statement? Which format specifier will you use to overcome it?

```
scanf("%s", arr);
```

9.4 Why does **`scanf`** use addresses of variables as arguments even though **`printf`** doesn't?

9.5 What is the difference in the data conversion techniques employed by **`scanf`** and **`printf`**?

PROGRAMMING & DEBUGGING SKILLS

9.1 Write a program that uses **`getchar`** to accept any string but uses **`putchar`** to print only the uppercase letters separated by spaces.

9.2 Modify the preceding program in C9.1 to use only **`getc`** and **`putc`**. How can you run this program from the shell of the operating system?

9.3 Use **`getchar`** and **`switch`** in a program to accept from the keyboard a string comprising only letters and then print the number of vowels and consonants in the string.

9.4 Develop a menu-based program that first prompts for an integer and then presents three options of printing that number in octal, decimal or hex.

9.5 Write a program that uses **`scanf`** to accept a time in *HH*`:`*MM*`:`*SS* 24-hour format. The three components must be validated before they are printed in the new format: *HH* `hours`, *MM* `minutes`, *SS* `seconds`.

9.6 The statement **`printf("%d%f\n");`** doesn't print the six-character string enclosed within quotes. Correct the statement.

9.7 Write a program that repeatedly asks for a letter, validates the input and quits when the user enters 0. The program should reverse the case of the letter before printing it and its ASCII value. (Make sure the buffer doesn't create problems.)

9.8 Write a program to accept a floating point number from the keyboard. Print the same number in the following formats: (i) total width of 10 characters, (ii) as in (i) but with 3 places after decimal, (iii) exponential notation, (iv) as in (iii) but with 3 places after decimal.

9.9 The following sequence is meant to print the string `Jan`, but it doesn't. What change needs to be made for obtaining the correct output?

```
char s[10] = "JanFebMar";
printf("%3s", s);
```

9.10 Write a program to print a truncated string that is defined in the program. The width of the truncated portion must be set by the user.

9.11 Use the * flag of **`printf`** in a program to print the pattern shown below. The number of rows in the pattern (here, six) must be accepted from user input.

```
*          *
 *        *
  *      *
   *    *
    *  *
     **
```

10 Arrays

WHAT TO LEARN

- Techniques of initializing a single-dimensional array.
- Populating an array with **`scanf`**.
- Inserting and deleting elements in an array.
- Reversing an array with and without using a second array.
- Understanding and implementing the *selection sort* algorithm.
- Searching an array using *sequential* and *binary search*.
- Handling two- and three-dimensional arrays.
- Applying the concepts of two-dimensional arrays to manipulate matrices.
- The C99 feature of determining the size of an array at runtime.

10.1 ARRAY BASICS

Variables can't handle voluminous data because a hundred data items need a hundred variables to store them. The solution is to use an *array*, which is an ordered set of data items stored contiguously in memory. Each data item represents an *element* of the array and is accessed with an *index* or *subscript*. For instance, the elements of an array named `signal` are accessed as `signal[0]`, `signal[1]`, `signal[2]`, and so forth. The first subscript is 0 and the last subscript is one less than the size of the array.

The elements of an array have a common data type (like `int`, `char`, etc.). An array must be declared with its type and size to enable the compiler to allocate a contiguous chunk of memory for all array elements. This is how a array named `signal` having 32 elements is declared:

```
int signal[32];
```

The first element is accessed as `signal[0]` and the last element is `signal[31]` (not 32). The subscript is either zero, a positive integer or an expression that evaluates to an integer value. An array element can be assigned to a variable (say, `int x = signal[2];`), which means we can use `signal[2]` wherever

we use x. By the same logic, &signal[2] is also a valid **scanf** argument like &x. It is easy to cycle through all array elements by using the array subscript as a key variable in a loop.

C doesn't offer *bounds checking* for arrays, i.e. it doesn't validate the index used to access an array element. For instance, the compiler will not generate an error if you attempt to access signal[35] even though the array has 32 elements. A negative index (as in signal[-5]) doesn't generate an error either. However, these accesses create problems at runtime because the program accesses an area of memory that was not allocated for the array. C trusts the programmer, so you'll have to do the bounds checking yourself.

The data type of an array is not restricted to the fundamental ones. Even derived and user-defined types like pointers and structures can be stored in arrays. C also supports multi-dimensional arrays where an element is handled with multiple subscripts. Since there is virtually no limit to the number of indexes and dimensions an array can have, arrays in C are very powerful data structures that can easily gobble up a lot of memory.

Because arrays can save a lot of data using a single identifier, there are a number of useful things you can do with them. You can add or delete array elements, reverse an array, sort and search it as well. You can also make a frequency count of the items in an array. Furthermore, many of the routines we develop in this chapter deserve to be converted to functions, a task that we'll attempt in Chapter 11.

Note: C doesn't complain if you use a negative index or one that exceeds its size. For instance, rate[-2] will clear the compilation phase but it will evaluate to an unpredictable value at runtime. It may destroy another variable, i.e., a part of the program you are running.

10.2 DECLARING AND INITIALIZING AN ARRAY

Like a variable, an array has to be declared before use. The declaration of an array specifies its name, type and size, which is normally specified as a constant or symbolic constant. The following statement declares a single-dimensional array named signal having 32 elements, each of type int:

```
int signal[32];                    Size specified explicitly
```

This declaration also *defines* the array by allocating the required memory where the last segment of memory is used by signal[31]. The **sizeof** operator, when used on an array, returns the total usage of all elements, so the above definition allocates 32 × sizeof(int) bytes of memory—typically, 128 bytes. Figure 10.1 depicts the memory layout of a single-dimensional array of 10 elements.

Arrays of other primitive data types are declared in a similar manner:

```
short months[12];                  Unlikely to change size
float rate[10];
char name[SIZE];                   Uses symbolic constant
```

The naming rules for variables *(5.2.1)* apply in identical manner to arrays. This means that the name of an array can comprise only letters, numerals and the underscore character, but it cannot begin with a digit. Also, case is significant, so months[12] and MONTHS[12] are two separate arrays.

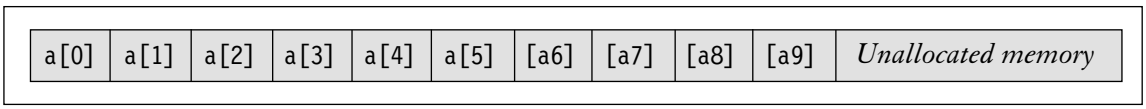


FIGURE 10.1 Layout in Memory of the Array `a[10]`

Tip: Arrays are often resized, so it makes sense to use a symbolic constant to represent the size (like in `char name[SIZE];`). When the size changes, simply change the **#define** directive that sets `SIZE`. You don't need to disturb the body of the program.

10.2.1 Initializing During Declaration

A simple declaration doesn't initialize an array (unless the keyword **static** is also used), so at this stage the elements have junk values. An array can be initialized on declaration by enclosing a comma-delimited list of values with curly braces. Two of the following declarations explicitly specify the size, but the third one does it only implicitly:

```
short months[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
float rate[10] = {12.5, 13.7, 5};
char type[] = {'A', 'n', 'd', 'r', 'o', 'i', 'd', '\0'};
```

In the first declaration, the number of values matches the size of the array. The second declaration performs a *partial* initialization by assigning values to the first three elements only. *When that happens, the remaining values are automatically initialized to zeroes* (NUL for an array of type `char`).

The third declaration uses an empty pair of brackets (`[]`) to *implicitly* specify the size of the array. This flexibility benefits the programmer and also doesn't bother the compiler as it can easily compute the size by counting eight values on the right. This array is a string because it contains the terminating NUL (`'\0'`). We'll discuss the string as an array of characters in Chapter 13.

10.2.2 Initializing After Declaration

You may not know all initial values at compile time, or you may need to reassign values later in the program. The technique using curly braces won't work when values are assigned later, in which case, each element has to be assigned individually:

```
months[0] = 31;
months[1] = 28.3;          Decimal portion truncated
rate[4] = 3;               OK, 3 converted to float
type[5] = '\0';            The NUL character having ASCII value 0
```

A loop is often used to initialize an entire array if all elements are assigned the same value or values obtained by evaluating an expression. The array index is usually the key variable of the loop. The following example on the left initializes all elements of the array, `rate`, to `12.5`, while the one on the right assigns an expression that contains the index itself:

```
short i;
float rate[10];
for (i = 0; i < 10; i++)
   rate[i] = 12.5;
```

```
short i;
float rate[SIZE];
for (i = 0; i < SIZE; i++)
   rate[i] = i * 1.1;
```

The symbolic constant, SIZE, must have been previously defined as **#define SIZE 10**. To print all elements, a similar two-line **for** loop should do the job:

```
for (i = 0; i < 10; i++)
   printf("%f ", rate[i]);
```

```
for (i = 0; i < SIZE; i++)
   printf("%f ", rate[i]);
```

The initialization rules for uninitialized and partially initialized arrays apply to arrays that are defined inside a function like **main**. In Chapter 11, you'll find out how arrays are automatically initialized either by use of additional keywords or by defining them outside a function.

10.3 array_init.c: INITIALIZING AND PRINTING ARRAYS

Program 10.1 features three arrays having different types and degrees of initialization. The program displays the contents of these arrays, and in doing so, establishes the consequences of using uninitialized and partially initialized arrays.

```
/* array_init.c: Shows effects of (i) no initialization, (ii) partial
                 initialization. Also initializes one array. */
#include <stdio.h>
#define SIZE 7

int main(void)
{
   short i;
   short short_arr[SIZE];
   int int_arr[SIZE] = {100, 200};
   long long_arr[] = {10, 20, 30, 40, 50, 60, 100};

   printf("Size of short_arr = %d\n", sizeof short_arr);
   printf("Size of long_arr = %d\n", sizeof long_arr);

   printf("short_arr contains    ");
   for (i = 0; i < SIZE; i++)
      printf("%5hd  ", short_arr[i]);       /* Uninitialized */

   printf("\nint_arr contains       ");
   for (i = 0; i < SIZE; i++)
      printf("%5d  ", int_arr[i]);          /* Partially initialized */

   printf("\nlong_arr contains      ");
   for (i = 0; i < (sizeof long_arr / sizeof(long)); i++)
      printf("%5ld  ", long_arr[i]);        /* Fully initialized */

   for (i = 0; i < SIZE; i++)
      short_arr[i] = i * 2;                 /* Initializing short_arr ... */
   printf("\nshort_arr now contains ");
   for (i = 0; i < SIZE; i++)
      printf("%5hd  ", short_arr[i]);       /* ... and printing it */

   return 0;
}
```

PROGRAM 10.1: **array_init.c**

```
Size of short_arr = 14
Size of long_arr = 28
short_arr contains    -24588   2052  13928  -16479  -31319   2052  -8352
int_arr contains         100    200      0       0       0      0      0
long_arr contains         10     20     30      40      50     60    100
short_arr now contains     0      2      4       6       8     10     12
```

PROGRAM OUTPUT: **array_init.c**

Observe that **sizeof** evaluates the memory allocated for all array elements. The following observations are made for the arrays used in the program:

- short_arr initially has junk values because it was declared without initialization. It is later assigned the value of an expression that is related to its index.
- int_arr is partially initialized. The first two elements are initialized, so the remaining five elements are automatically set to zero.
- The size of long_arr is *implicitly* declared. The compiler determines it by counting seven values. This size is also computed by dividing the total space used by the array (**sizeof long_arr**) by the size of the long data type (**sizeof(long)**).

Generally, functions that accept an array as argument also need the number of elements as another argument. This value should be passed to the function as the expression **sizeof** *array* / **sizeof**(*data type*) rather than a constant. Even if you change the size of the array later, you don't need to change this expression.

Takeaway: An uninitialized array contains junk values, but a partially initialized array has the remaining elements set to zero or NUL. Also, **sizeof** for an array evaluates to the total space taken up by all elements of the array.

10.4 scanf_with_array.c: POPULATING AN ARRAY WITH scanf

Using the **scanf** function in a loop, we can populate an array with keyboard input. Program 10.2 takes advantage of the return value of **scanf** to determine the end of input. The **while** loop in this program terminates when **scanf** returns 0, and that happens when EOF or any non-digit character (say, q) is keyed in.

You can input as many as 20 integers—on separate lines if necessary—but every invocation of **scanf** reads one integer and leaves the remaining input in the buffer. The number of values input is available in the expression i - 1, which determines the number of times the second loop will iterate to print all array elements that were populated.

10.5 BASIC OPERATIONS ON ARRAYS

As the previous program clearly shows, an array element can be treated as a variable for all practical purposes. An element can thus form a component of any arithmetic, assignment, relational or logical

```
/* scanf_with_array.c: Populates an array with user input. */
#include <stdio.h>
int main(void)
{
   short i = 0, num;
   short arr[20];
   printf("Key in some integers followed by EOF: ");
   while (scanf("%hd", &arr[i++]) == 1)              /* Populates array */
      ;
   num = i - 1;
   for (i = 0; i < num; i++)
      printf("arr[%hd] = %hd, ", i, arr[i]);
   return 0;
}
```

PROGRAM 10.2: **scanf_with_array.c**

```
Key in some integers followed by EOF: 11 22 [Enter]
44 55 77 [Enter]
99
EOF                                    Press [Ctrl-d] or [Ctrl-z] to terminate loop
arr[0] = 11, arr[1] = 22, arr[2] = 44, arr[3] = 55, arr[4] = 77, arr[5] = 99,
```

PROGRAM OUTPUT: **scanf_with_array.c**

expression. The following examples drawn from programs in this chapter reveal the various ways of using an array element in an expression:

```
arr[i] = arr[i + 1];
temp = arr[i];
arr[c]++;
if (arr[i] > arr[j])
if (num < arr[0] || num > arr[SIZE - 1])
```

The [and] in the element `arr[i]` are actually a set of operators that serve to evaluate the element. These symbols have the highest precedence—at the same level as the parentheses. Thus, in the expression `arr[c]++`, `arr[c]` is evaluated before the increment operation is applied.

10.5.1 `insert_element.c`: Inserting an Element in an Array

To insert an element in an array at a specified location, we need to shift to the right all elements from that location to the end of the array. Program 10.3 takes user input to initialize an array and then takes further input to insert an element at a specified location. Finally, it displays the changed contents of the array. The program is documented well enough to require further elaboration.

```
/* insert_element.c: Inserts element at specified location in an array. */
#include <stdio.h>
int main (void)
{
   short i, j, num, new_value, arr[20];
   printf("Number of integers to input? ");
   scanf("%hd", &num);
   printf("Key in %hd numbers: ", num);
   for (i = 0; i < num; i++)
      scanf("%hd", &arr[i]);
   printf("Enter desired index at insertion and inserted value: ");
   scanf("%hd%hd", &j, &new_value);
   /* Section for shifting elements to right and insertion of element */
   for (i = num; i > j; i--)            /* arr[num] is last element plus one */
      arr[i] = arr[i - 1];              /* Moves a group of elements to right */
   arr[i] = new_value;                  /* Inserts new value at index j */
   for (i = 0; i <= num; i++)           /* Now <= because of an extra element */
      printf("%hd ", arr[i]);
   return 0;
}
```

PROGRAM 10.3: **insert_element.c**

```
Number of integers to input? 6
Key in 6 numbers: 11 22 33 55 66 77
Enter desired index at insertion and inserted value: 3 44
11 22 33 44 55 66 77
```

PROGRAM OUTPUT: **insert_element.c**

10.5.2 delete_element.c: Deleting an Element from an Array

Deletion of an element is equally simple and can be achieved with a similar but inverse line of reasoning. Just move to the specified location as before, and then shift to the *left* all elements from that location to the end of the array. Program 10.4 achieves this task. This time we use the technique discussed in Section 10.4 which populates an array without knowing the number of elements needed. This program too is well-documented, so no further explanation is provided.

Large arrays are often sorted for faster access. Insertion and deletion of elements in a sorted array require additional programming effort. Because a user can't specify the insertion or deletion point for a sorted array, the program has to locate it. After you have learned to search an array, these tasks will be within your grasp.

```
/* delete_element.c: Deletes element at specified location in an array. */
#include <stdio.h>
int main (void)
{
   short i, j, num, arr[20];
   printf("Key in numbers and press [Ctrl-d]:\n");
   num = 0;
   while (scanf("%hd", &arr[num]) == 1)
      num++;                          /* Number of items read = num */
   printf("\nEnter index at deletion point: ");
   scanf("%hd", &j);
   for (i = j; i < num - 1; i++)      /* Now num - 1 because of deleted element */
      arr[i] = arr[i + 1];            /* Moves a group of elements to left */
   for (i = 0; i < num - 1; i++)
      printf("%hd ", arr[i]);
   return 0;
}
```

PROGRAM 10.4: **delete_element.c**

```
Key in numbers and press [Ctrl-d]:
11 22 33 44 55 66 77
[Ctrl-d] or [Ctrl-z]
Enter index at deletion point: 3
11 22 33 55 66 77
```

PROGRAM OUTPUT: **delete_element.c**

10.6 REVERSING AN ARRAY

This section discusses two techniques of reversing an array. This is easily achieved with two arrays, but since large arrays can take up a lot of space, we must also be able to perform the task using a single array. Both techniques are discussed in the following sections.

10.6.1 Reversing with Two Arrays

The simplest method of reversing the contents of an array is to use a second array to hold the new values. The sequence goes like this: read the first element of *array1* and save it to the last element of *array2*, save the next element of *array1* to the one previous to the last element of *array2*, and so on. This simple code segment does the job:

```
short i, size = 7, short brr[7];
short arr[7] = {1, 3, 6, 9, 11, 22, 5};
for (i = 0; i < size; i++)                    /* Reversing array */
   brr[size - 1 - i] = arr[i];
```

```
for (i = 0; i < size; i++)                          /* Printing reversed array */
   printf("%d ", brr[i]);
```

OUTPUT: 5 22 11 9 6 3 1

In this and the next example, we have chosen to use a variable (`size`) to store the number of elements of the array. A serious programmer, however, will use **`#define SIZE 7`** instead. Since this size is one more than the value of the last subscript, we need to subtract one from it before assigning an element of the first array to the second array.

10.6.2 Reversing with a Single Array

An array of *n* elements can be reversed without using a second array. This technique uses an intermediate variable to reverse the first and *n*th (last) element, the second and (*n* - 1)th element, and so on. This process is carried out *n* / 2 times. If *n* is even, the exchange is completed for all elements, but if *n* is odd, the middle element is unaffected. This is the case anyway, so program logic doesn't depend on whether *n* is even or odd. The following code does the job:

```
short i, j, temp, size = 7;
short arr[7] = {1, 3, 6, 9, 11, 22, 5};
for (i = 0, j = size - 1; i < (size / 2); i++, j--) {
   temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}
for (i = 0; i < size; i++)                          /* Printing reversed array */
   printf("%d ", arr[i]);
```

OUTPUT: 5 22 11 9 6 3 1

The comma operator in *exp1* and *exp3* of the combined expression of the first **`for`** loop is responsible for the lean loop body. This loop swaps two values using `temp` as the intermediate variable. The subscript `i` is incremented from 0 to `(size - 1) / 2`, while `j` is simultaneously decremented from the last subscript `(size - 1)`. The original array is, however, lost in the process. Because this technique saves memory, we'll henceforth use it for any job related to array reversal.

10.7 TWO PROGRAMS REVISITED

Some of the previous programs used a loop to generate one or more values in every iteration. Because all of these values couldn't be saved, action on each value was taken the moment it was generated and before the next iteration could begin. Now that we have an array to store all values, we can use these values even *after* the loop has completed execution. Let us now revisit two of these programs to make them flexible enough to handle multiple requirements.

10.7.1 `extract_digits2.c`: Saving Digits of an Integer

Program 10.5 is an improvement over its predecessor (Program 8.3), which extracts the digits from an integer. This program, however, stores all extracted digits in an array and then reverse-reads

the array to print the digits in the normal sequence. With minor modifications to the code, you can manipulate these digits in any manner you want. You can print them in any sequence, add or multiply them.

```
/* extract_digits2.c: Modifies a previous program to save the digits of an
                      integer in an array for manipulation in any manner. */
#include <stdio.h>

int main(void)
{
   unsigned int number;
   short i = 0, digit_arr[10];

   printf("Key in an integer: ");
   scanf("%d", &number);

   while (number != 0) {
      digit_arr[i++] = number % 10;          /* Saves last digit */
      number = number / 10;                  /* Removes last digit */
   }

   i--;
   while (i >= 0)                            /* Printing the digits ... */
      printf("%hd ", digit_arr[i--]);        /* ... in normal sequence */
   printf("\n");

   return 0;
}
```

PROGRAM 10.5: **extract_digits2.c**

```
Key in an integer: 1357926
1 3 5 7 9 2 6
```

PROGRAM OUTPUT: **extract_digits2.c**

10.7.2 decimal2anybase.c: Converting from Decimal to Any Base

The concepts behind the reversal of an array can be gainfully employed in the task of converting a number from one base to another. In Chapter 8, decimal-to-binary conversion was attempted twice by reversing-reading an array of remainders. Program 10.6 extends the scope by converting from decimal to *any* base not exceeding 16. Instead of reverse-reading an array, the program reverses the array itself using the technique discussed previously.

The program uses two arrays of type `char`. `digit_arr` stores the remainders and `digits` stores the character representation of all hex digits. For decimal and octal numbers, the program needs to use only the first 10 and 8 elements, respectively, of this array.

The number to be converted and the required base are first taken from user input. This number is repeatedly divided by the base. Using `remainder` as subscript, the program extracts the appropriate

digit-letter from digits. For instance, if remainder is 15 from a division by 16, digits[remainder] is 'F'. This value is stored in digit_arr, which is finally reversed by the second **for** loop. The last **for** loop simply reads this converted array.

```
/* decimal2anybase.c: Extends decimal2binary.c to convert a number to any base.
                      Reverses an array without using a second one. */
#include <stdio.h>
int main(void)
{
   char digit_arr[80];
   char digits[16] = {'0','1','2','3','4','5','6','7','8','9',
                      'A','B','C','D','E','F'};
   short i, j, imax, temp, base, remainder;
   long quotient;

   printf("Enter number to convert and base: ");
   scanf("%ld%hd", &quotient, &base);
   for (i = 0; quotient > 0; i++) {       /* Divide repeatedly */
      remainder = quotient % base;
      quotient /= base;
      digit_arr[i] = digits[remainder];   /* Select digit from array */
   }
   imax = i;                              /* Number of digits to be printed */

   /* Reverse the array before printing the digits */
   for (i = 0, j = imax - 1; i < imax / 2; i++, j--) {
      temp = digit_arr[i];
      digit_arr[i] = digit_arr[j];
      digit_arr[j] = temp;
   }
   for (i = 0; i < imax; i++)
      printf("%c ", digit_arr[i]);

   return 0;
}
```

PROGRAM 10.6: **decimal2anybase.c**

```
Enter number to convert and base: 12 2
1 1 0 0                                          2^3 + 2^2 = 12

Enter number to convert and base: 127 2
1 1 1 1 1 1 1                                    Seven 1s—127

Enter number to convert and base: 255 8
3 7 7                                            3×8^2+7×8^1+7×8^0 = 255

Enter number to convert and base: 255 16
F F                                              15×16^1+15×16^0 = 255
```

PROGRAM OUTPUT: **decimal2anybase.c**

10.8 SORTING AN ARRAY (SELECTION)

Sorting refers to the ordering of data in ascending or descending sequence. Large lists like a telephone directory must be sorted for them to be useful. Arrays are often sorted either to present information in a specified sequence or to fulfill a requirement of other operations (like a search). The technique used in sorting numbers is different from the one used in sorting strings. In this section, we'll discuss the theory and programming techniques related to the sorting of numeric data. Sorting of character strings is examined in Chapter 13.

There are several algorithms for sorting arrays. Three of the common ones are known as *bubble sort*, *selection sort* and *insertion sort*. Because they have some common features, knowledge of one algorithm will help you understand the others. In this section, we'll examine the algorithm for selection sort before we develop a program that implements this algorithm.

Consider a numeric array, `arr`, which needs to be sorted in ascending order. Sorting is done by repeatedly scanning the array in multiple passes, where the start point of each pass progressively moves from `arr[0]` to `arr[1]`, `arr[2]`, and so forth. If a number is found to be lower than the start element of that pass, then the two numbers are interchanged. Thus, at the end of each pass, the start element has the lowest number among the elements encountered in that pass. A total of n - 1 passes are required for an array of n elements, where each pass encounters one fewer element compared to its previous pass.

Let's now implement this algorithm in Program 10.7, which sorts a set of integers input from the keyboard. The sorting operation is carried out with a nested **`for`** loop. The outer loop determines the start point of each pass, while the inner loop scans the elements following the start element. We'll examine the exact sequence of events that occur in the first two iterations and the last iteration of the outer loop.

Outer loop: First iteration The outer loop begins the scan by selecting the first element as the start point (`i = 0`). Next, the inner loop scans all remaining elements starting from the second element (`j = i + 1`). In each iteration, it compares the value of the scanned element to the start element and interchanges them if the start element is greater than the other. For this purpose, it makes use of the temporary variable, `tmp`. After the inner loop has completed its scan, control reverts to the outer loop for the second iteration.

Outer loop: Second iteration The outer loop now selects the second element as the start point (`i = 1`). The inner loop now performs the scan for all elements except the first and second (`j = 2`), comparing each element to the second element and interchanging them if the relational test is satisfied.

Outer and inner loops: Last iteration With every iteration of the outer loop, the start point progressively shifts from the first element to the one preceding the last element. At this point, the inner loop iterates only once by comparing this element to the last one and interchanging if necessary. Both loops have now completed their run and the sort action has been completed.

For your benefit, the program contains an additional **`for`** loop which prints the contents of the array after the inner **`for`** loop completes a single pass. The output shows how each pass moves the lowest number encountered in that pass to the left. Eventually, the last line of diagnostic output shows a sorted array in place.

```
/* sort_selection.c: Sorts a one-dimensional array using a nested for loop.
                     Algorithm used: selection sort. */
#include <stdio.h>

int main(void)
{
   short i, j, k, imax, tmp;
   short arr[100];

   printf("Key in some integers (q to quit): ");
   for (i = 0; scanf("%hd", &arr[i]) == 1; i++)
      ;

   printf("Status of array after each iteration ...\n");

   imax = i;                              /* Number of elements used */
   for (i = 0; i < imax - 1; i++)  {      /* Last element not considered */
      for (j = i + 1; j < imax; j++)      /* Scanning remaining elements */
         if (arr[i] > arr[j]) {           /* If start element is larger ... */
            tmp = arr[i];
            arr[i] = arr[j];              /* ... interchange the ... */
            arr[j] = tmp;                 /* ... elements */
         }

      /* This for loop displays the array contents after each pass.
         You can remove it after you have understood selection sort. */
      for (k = 0; k < imax; k++)          /* Displaying progress */
         printf("%3hd", arr[k]);
      printf("\n");
   }

   printf("Final sorted output ...\n");

   for (i = 0; i < imax; i++)
      printf("%3hd", arr[i]);
   printf("\n");

   return 0;
}
```

PROGRAM 10.7: **sort_selection.c**

```
Key in some integers (q to quit): 55 4 27 1 66 34 11 q
Status of array after each iteration ...
  1 55 27  4 66 34 11
  1  4 55 27 66 34 11
  1  4 11 55 66 34 27
  1  4 11 27 66 55 34
  1  4 11 27 34 55 66
Final sorted output ...
  1  4 11 27 34 55 66
```

PROGRAM OUTPUT: **sort_selection.c**

Note: Since a character string doesn't have a numeric value (even though its individual characters have), sorting a set of strings is based on a different principle altogether. Starting with the first character, the ASCII value of each character of *string1* is compared to its peer in *string2*. The sort stops when the first mismatch occurs. We need to use **strcmp** (a string-handling function) to make this comparison, so the topic will be visited in Chapter 13.

10.9 array_search.c: PROGRAM TO SEQUENTIALLY SEARCH AN ARRAY

Searching is a common operation performed on arrays. We often search an array for a number. Depending on the application, we may ignore this number or increment a counter if the number exists in the array. If it doesn't exist, we may add the number to the array. The simplest search technique sequentially scans all array elements and terminates the search when a match is found. A more efficient technique uses a sorted array to conduct a binary search. Both techniques are discussed in this chapter.

Program 10.8 sequentially searches an array that can hold up to 128 integers. A user first keys in a number of integers before searching the array for a particular integer. In case a match is found, the program prints the array element along with the index. This technique is adequate for small arrays.

With **scanf** running in a loop, a set of six integers are keyed in before the input is terminated with `q`. Using the notation `&arr[i++]`, **scanf** saves these integers in the array `arr`. **getchar** then performs its usual cleanup job to get rid of the `q` and other non-numeric characters that may remain in the system buffer.

Next, a search routine runs in a nested **for** loop to search this array for another user-input integer. If a match is found, the variable `is_found` is set to `'y'` and the inner loop is terminated with **break**. If no match is found, the loop terminates normally and `is_found` remains unchanged at `'n'`. Control in either case moves down to the conditional expression which evaluates the status of `is_found` and selects the appropriate **printf** expression. The next iteration of the outer loop then prompts the user for another integer.

10.10 BINARY SEARCH

Sequential search is slow and is acceptable only for searching small arrays. When working with large data volumes, we need to use the *binary search* mechanism. In this scheme, the array is sorted before it is subjected to a search. A binary search *halves* the search domain every time it is invoked. The number of loop iterations needed is thus reduced, making binary search more efficient than sequential search. Every serious programmer should understand this search mechanism and know how to implement it in their programs.

The principle used in binary search is simple, and you may have already used it in a number guessing game. You ask a person to guess a number between 1 and 100. You then compute the mid-point of this range (50), and then ask whether the number lies in the lower or upper half of the range. The next question halves the selected range even further, and in only a few guesses, the range reduces to a single number—the one that was guessed. The range could typically follow this progression:

```
1-100,  50-100,  50-75,  63-75,  63-69,  66-69,  66-68,  67-68,  67
```

```
/* array_search.c: Saves in an array some integers from the keyboard.
                   Also sequentially searches the array. */
#include <stdio.h>

int main(void)
{
   int i = 0, imax, num, arr[128];
   char is_found;

   /* Input the integers */
   printf("Key in some integers (q to quit): ");
   while (scanf("%d", &arr[i++]) == 1)   /* Becomes 0 with non-digit */
      ;

   /* Remove the 'q' and other trailing characters from input */
   while (getchar() != '\n')
      ;
   imax = i - 1;                         /* Save i before it is reused */

   /* Search the array for a number */
   for (; printf("Integer to search (q to quit): ") ;) {
      if (scanf("%d", &num) == 0)        /* When non-digit is input */
         break;
      is_found = 'n';                    /* Set default to 'n' */
      for (i = 0; i < imax; i++) {       /* Cycle through array elements */
         if (num == arr[i]) {            /* If number found in array ... */
            is_found = 'y';
            break;                       /* ... abandon search */
         }
      }
      is_found == 'n' ? printf("%d not found\n", num) :
                        printf("%d stored in arr[%d]\n", num, i);
   }
   return 0;
}
```

PROGRAM 10.8: **array_search.c**

```
Key in some integers (q to quit): 5 10 15 20 25 31 q
Enter an integer to search for: 20
20 stored in arr[3]
Enter an integer to search for: 21
21 not found
Enter an integer to search for: q
```

PROGRAM OUTPUT: **array_search.c**

When using the binary search technique, we didn't need more than nine guesses to arrive at the number 67. On the other hand, a sequential search could require up to 100 guesses. We'll now understand the algorithm for binary search on arrays before developing a program that implements this algorithm.

10.10.1 The Binary Search Algorithm

Let's search a sorted array A of n elements for the number x. Let *start* and *end* represent the two ends (indexes) of the search domain. Because this domain shrinks by half every time a search is made on the array, the *start* and *end* points progressively move closer to each other. The search algorithm thus involves the following steps:

1. Set the initial values of *start* to 1 and *end* to n.
2. Check whether *start* <= *end* in every iteration. If this condition fails, go to Step 7.
3. Set the mid-point of the search domain as *mid* = (*start* + *end*) / 2 and compare the value of x to A[*mid*].
4. If $x < A$[*mid*], the number belongs to the lower half of the range, which means that *end* has to be shifted to the left. Set *end* = *mid* - 1 and go to Step 2.
5. If $x > A$[*mid*], the number belongs to the upper half of the range, which means that *start* has to be shifted to the right. Set *start* = *mid* + 1 and go to Step 2.
6. If you have come here, it means that $x = A$[*mid*]. The search is successful and the algorithm is terminated.
7. If you have come here (from Step 2), the search is unsuccessful and the algorithm is terminated.

Every time we move to Step 2—either from Step 4 or Step 5—the difference between *start* and *end* shrinks. For a search to be successful, *start* must remain equal to or less than *end*. If *start* eventually exceeds *end*, it means that the search has failed. Since this algorithm discards half of the data in every iteration, binary search is more efficient than sequential search, where efficiency is related to the number of iterations required to complete a lookup. We'll consider some sample figures after we have discussed the next program.

10.10.2 `binary_search.c`: Implementation of the Algorithm

Program 10.9 is a simple implementation of the algorithm that was just discussed. It searches a sorted array of 10 elements that are initialized at the time of declaration. The program uses an outer **`while`** loop to repeatedly accept an integer from the keyboard, and then uses an inner **`while`** loop to perform a lookup of the array. For your benefit, the program prints the adjusted *start* and *end* points (SP and EP) in every iteration.

The first **`if`** construct inside the outer **`while`** loop validates `num`, the number that is input. Like some of the previous programs, this program also ends when the input is not a number. The **`continue`** statement in the **`else-if`** section prevents the remaining statements from being executed if the number falls outside the permitted range.

The binary search algorithm is implemented in the inner **`while`** loop. Before this loop is entered, the variables `start` and `end` are set to the first and last index of the array, respectively. Every iteration of the loop sets a new mid-point (`mid`) by averaging `start` and `end`. The element with `mid` as index is then compared to `num`, the search value. If this comparison fails in one iteration, either `start`

or end is adjusted, bringing the two closer to each other. This also changes mid which thus moves closer to the search target.

The output shows how the start and end points are adjusted with every iteration. If the search eventually succeeds, the loop terminates abnormally using the **break** route. However, if the number doesn't exist in the array, the loop terminates normally because of the eventual failure of the control expression. Observe how the number 66 could be found in the first iteration itself, while both 22 and 87 needed four iterations before 22 was found but not 87.

```
/* binary_search.c: Uses the binary search mechanism for searching a sorted
                    array. Progressively narrows the search domain by
                    adjusting the start and end points of search. */
#include <stdio.h>
#define SIZE 10

int main(void)
{
   short num, start, mid, end;
   short arr[SIZE] = {11, 22, 33, 44, 55, 66, 77, 88, 99, 110};

   /* Search the array for a number */
   while (printf("Integer to search (q to quit): ")) {
      if (scanf("%hd", &num) == 0)       /* When non-digit is input */
         break;
      else if (num < arr[0] || num > arr[SIZE - 1]) {
         printf("Number outside range\n");
         continue;
      }

      start = 0; end = SIZE;
      while (start <= end) {
         mid = (start + end) / 2;
         printf("[SP: %hd, EP: %hd]  ", start, end);
         if (arr[mid] < num)
            start = mid + 1;             /* Advance the start point */
         else if (arr[mid] > num)
            end = mid - 1;               /* Move back the end point */
         else                            /* Search successful ... */
            break;                       /* ... array index = mid */
      }

      /* We now know whether the search was successful or not */
      if (arr[mid] == num)                /* From previous break */
         printf("\nNumber found at index %hd\n", mid);
      else                                /* From normal loop termination */
         printf("\nNumber not found\n");
   }
   return 0;
}
```

PROGRAM 10.9: **binary_search.c**

```
Integer to search (q to quit): -10
Number outside range
Integer to search (q to quit): 22
[SP: 0, EP: 10]  [SP: 0, EP: 4]  [SP: 0, EP: 1]  [SP: 1, EP: 1]
Number found at index 1
Integer to search (q to quit): 66
[SP: 0, EP: 10]
Number found at index 5
Integer to search (q to quit): 33
[SP: 0, EP: 10]  [SP: 0, EP: 4]
Number found at index 2
Integer to search (q to quit): 87
[SP: 0, EP: 10]  [SP: 6, EP: 10]  [SP: 6, EP: 7]  [SP: 7, EP: 7]
Number not found
Integer to search (q to quit): q
```

PROGRAM OUTPUT: **binary_search.c**

Using the binary search technique, we needed a maximum of four iterations to complete a lookup of an array of 10 elements. The corresponding figure for a sequential search is 10. The power of binary search can be assessed from the following rule:

The number of iterations needed to search a sorted array of n elements is the number of bits needed to represent n in binary.

Thus, a 1000-element array needs 10 iterations (compared to 1000 for sequential search). The gulf between the two widens as the size of the array increases, showing how enormously efficient binary search can be. Now think of this: a sorted array of 4 billion elements can be searched in only 32 iterations!

Note: The array must be sorted for binary search to work.

10.11 count_chars.c: FREQUENCY COUNT OF DATA ITEMS

We often need to know the frequency of occurrence of each item in the data. For instance, a calligraphist may like to know the usage of letters in a text book. A demographic expert may want to study the age-wise distribution of people. A weather expert may like to know the number of days a city has experienced humidity exceeding 90%. These tasks can't be handled by variables because you need too many of them. Arrays are useful counting devices and can handle these jobs easily.

A single array can count the frequency of occurrence of several quantities using one of two techniques. The simplest implementation uses the quantity itself as the index of the array (provided it is integral and positive). Whenever this quantity occurs in the data, the program merely increments the element having that quantity as the index. The other technique is to use a two-dimensional array to store both the number and its count. The following program uses the first technique and Section 10.13 examines the second one.

```
/* count_chars.c: Counts frequency of occurrence of individual characters.
                  Index of array is the ASCII value of a character. */
#include <stdio.h>
#define SIZE 128

int main(void)
{
   short c, i;
   short arr[SIZE] = { 0 };       /* Remaining elements also initialized */

   printf("Enter a line of text and press [Enter]\n");
   while ((c = getchar()) != '\n')
      arr[c]++;

   for (i = 0; i < SIZE; i++)
      if (arr[i] > 0)
         printf("%c: %hd\n", i, arr[i]);

   return 0;
}
```

PROGRAM 10.10: **count_chars.c**

```
Enter a line of text and press [Enter]
aeiouiioeaiou[Enter]
a: 2
e: 2
i: 4
o: 3
u: 2
```

PROGRAM OUTPUT: **count_chars.c**

Program 10.10 uses a small array of 128 elements to count the number of occurrences of individual characters. Each character, which has an integral value, is fetched using **getchar**. Even though the size of the array is 128, the PC keyboard has far fewer characters. Thus, many elements of the array would be unaffected by the program. The output is printed in the format *letter*: *count*.

The return value of **getchar** forms the index of the array. For instance, the element `arr[97]` counts the frequency of occurrence of the letter `'a'`, which has 97 as its ASCII value. The statement `arr[97]++;` adds one to the existing count whenever `'a'` is encountered in the input. The **for** loop prints only those elements that have a count greater than zero.

10.12 TWO-DIMENSIONAL (2D) ARRAYS

C supports multi-dimensional arrays to represent tables and complex data structures. Each element of such an array is accessed with multiple subscripts. A two-dimensional array needs two subscripts and a three-dimensional array needs three subscripts. Like a single-dimensional array, a multi-dimensional array occupies a contiguous region of memory.

A two-dimensional (2D) array, `arr`, takes the form `arr[i][j]`, where the subscript *i* represents the number of rows and *j* represents the number of columns. The following statement declares a 2D array without initializing it:

```
short table[3][4];
```

Because a 2D array is actually a set of rows and columns, the symbolic constants, `ROWS` and `COLUMNS`, are often used to represent its size. Internally though, a 2D array can be viewed as an array of single-dimensional arrays. This means that `table` represents a set of three single-dimensional arrays of four elements each. Alternatively, it can be considered as three rows having four columns in each row. Figure 10.2 depicts the layout in memory of an array having these dimensions. Knowledge of this organization will help us understand pointers when the concept is applied to arrays.

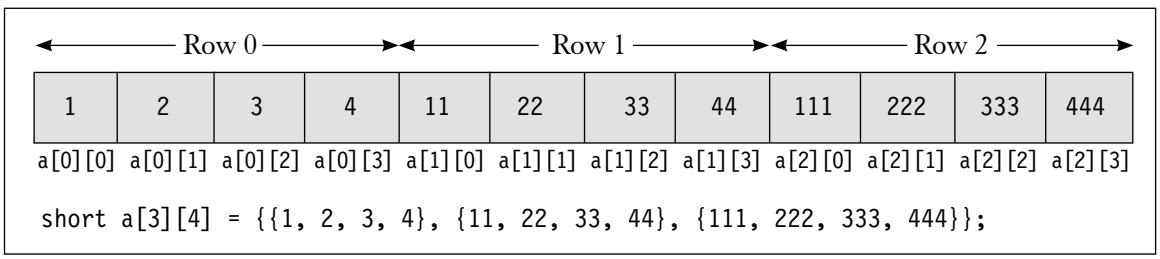


FIGURE 10.2 Layout in Memory of the 2D array `a[3][4]`

Note from Figure 10.2 that a total of 12 memory slots (typically using 24 bytes) are allocated for this array. The first four cells are used by the four columns of row 0, followed by four columns each of rows 1 and 2. This means that, when memory is scanned sequentially, a column changes faster than its row. Thus, the array can be completely accessed by using the row and column as key variables in an outer and inner loop, respectively.

10.12.1 Full Initialization During Declaration

We can employ the usual technique of using curly braces for initialization of a 2D array at the time of declaration, except that you may also have to use a set of inner braces to group the columns of each row. Here's how `table` is declared and initialized with 15 values:

```
short table[3][5] = { {0, 1, 2, 3, 4},          Row 0
                      {10, 11, 12, 13, 14},     Row 1
                      {20, 21, 22, 23, 24} };   Row 2
```

The columns of each row should be enclosed by a set of inner curly braces and placed on a separate line. It then becomes easy to determine the value of an element by visual inspection. For instance, `table[1][2]` is 12 and `table[2][0]` is 20. Observe the comma that follows the closing curly brace of each row and not the number on its left. For a fully initialized array (like the one above), we can also do without the inner braces:

```
short table[3][5] = {0, 1, 2, 3, 4, 10, 11, 12, 13, 14, 20, 21, 22, 23, 24};
```

It could be difficult to relate values to subscripts when the inner braces are dropped, so you should use them wherever possible—specially for large arrays. When all values are known, C permits dropping the subscript for the row:

```
short table[][5] = { {0, 1, 2, 3, 4},
                     {10, 11, 12, 13, 14},
                     {20, 21, 22, 23, 24} };
```

The inner braces and subscript for the row can be dropped only if their omission doesn't lead to ambiguity. This is not an issue for a fully initialized array like the one above, so we can drop the braces and the compiler will still be able to correctly determine the number of rows.

10.12.2 Partial Initialization During Declaration

C permits partial initialization of an array. The braces in that case are necessary because without them there is no way the compiler can figure out how the values are organized among the rows and columns. Consider the following declarations:

```
short table1[3][5] = { {0, 1, 2}, {10, 11}, {20} };
short table2[ ][5] = { {0, 1, 2}, {10, 11}, {20} };
```

For both arrays, the three elements of the first inner group belong to row 0, the next two belong to row 1 and the last element belongs to row 2. The remaining elements are initialized to zeroes as shown below in the row-column view of the data:

```
 0  1  2  0  0        Row 0
10 11  0  0  0        Row 1
20  0  0  0  0        Row 2
```

Because of the presence of the braces, the compiler correctly computes the number of rows in the declaration of `table2`. Omission of the inner braces would confuse the compiler when it looks at the following data:

```
short table2[3][5] = { 0, 1, 2, 10, 11, 20 };
```

The compiler would now assign the first five elements to row 0 and the last element to row 1. It would also incorrectly compute the number of rows for `table2`. The row-column view would then change to this:

```
 0  1  2 10 11        Row 0
20  0  0  0  0        Row 1
 0  0  0  0  0        Row 2
```

The message is loud and clear. Use inner braces whether initialization is partial or complete.

10.12.3 Assignment After Declaration

Like with single-dimensional arrays, elements can also be assigned values after declaration. The assigned value can be a constant, variable or expression which can include another array element:

```
table[1][4] = 14;
table[2][7] = x;
table[2][4] = table[1][4] + x;
```

You can also use **scanf** to assign a value to a 2D array element. Don't forget to use the & prefix:

```
scanf("%hd", &table[2][3]);
```

scanf can also be used in a nested loop construct to populate an entire 2D array. The statement **scanf("%hd", &table[i][j]);** does the job but it could be inconvenient to key in data without knowing which element it is meant for.

Takeaway: C uses the inner curly braces to assign a set of values to a specific row of a 2D array. If you partially assign a row, then you must use the braces because otherwise the compiler would simply assign the values to all of the columns of a row before taking up the next row.

10.12.4 Assignment and Printing Using a Nested Loop

You need a nested loop to assign a single value or expression to an entire 2D array. The outer loop changes the row and the inner loop changes the column. Program 10.11 fully initializes `table` to an expression. The same nested loop also prints the contents of the array.

```
/* print_2d_array.c: Populates and prints a 2D array. */
#define ROWS 3
#define COLUMNS 5
#include <stdio.h>
int main(void)
{
   short i, j;
   short table[ROWS][COLUMNS];
   for (i = 0; i < ROWS; i++) {
      for (j = 0; j < COLUMNS; j++) {
         table[i][j] = i * 10 + j * 2;
         printf("%5hd", table[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

PROGRAM 10.11: **print_2d_array.c**

```
    0    2    4    6    8
   10   12   14   16   18
   20   22   24   26   28
```

PROGRAM OUTPUT: **print_2d_array.c**

10.13 count_numbers.c: USING A 2D ARRAY AS A COUNTER

Program 10.12 uses a 2D array to implement a frequency counter. The program accepts a set of integers as user input and sequentially assigns them to the first row of the array arr, but only after checking that the number doesn't already exist there. The corresponding column of the next row maintains a count for the number. For instance, the first number is assigned to arr[0][0] and its count is saved in arr[1][0].

```
/* count_numbers.c: Uses a 2D array as a frequency counter.
                    First row stores number, second row stores frequency. */
#include <stdio.h>

int main(void)
{
   short i, imax = 0, tmp;
   short arr[2][50] = {{0}};          /* All array elements initialized */
   char is_found;

   printf("Key in some integers (q to quit): ");
   while (scanf("%hd", &tmp) == 1) {
      is_found = 'n';
      for (i = 0; i < imax; i++) {    /* Check whether number exists */
         if (arr[0][i] == tmp) {      /* If number found in array ... */
            arr[1][i]++;              /* ... then increment its count */
            is_found = 'y';
            break;                    /* Abandon further checking */
         }
      }
      if (is_found == 'n') {          /* If number not found in array ... */
         arr[0][i] = tmp;             /* ... add number to array ... */
         arr[1][i] = 1;               /* ... and set its count to 1 */
         imax++;                      /* Increase limit of search */
      }
   }

   /* Print array contents in pairs */
   for (i = 0; i < imax; i++)
      printf("%5hd = %2hd\n", arr[0][i], arr[1][i]);

   return 0;
}
```

PROGRAM 10.12: **count_numbers.c**

```
Key in some integers (q to quit): 11 33 22 -1 0 33 666 33 22 0 33 q
   11 =  1
   33 =  4
   22 =  2
   -1 =  1
    0 =  2
  666 =  1
```

PROGRAM OUTPUT: **count_numbers.c**

Each number that is input is sequentially looked up in the first row of the array. This is how the program responds to the result of the search:

- If the number is found in that row, then its corresponding column in the second row is incremented (`arr[1][i]++;`). The **`for`** loop is then terminated with **`break`** for the next number to be keyed in.
- If the number doesn't exist in the first row, it is assigned to the next free element of that row. The count in the corresponding column of the second row is then set to 1. The variable, `imax`, is then incremented for use by the next number to be added.

After all numbers have been input, the expression `imax - 1` evaluates to the number of elements used by the first row of `arr`. The last **`for`** loop uses this value to print the key-value pairs of only those array elements that have been utilized.

Note that even though the highest number input was 666, only 12 elements of the array were used. Program 10.10 (which used the number itself as the array index) would have needed an array of 666 elements. That program will still fail here because it can't handle negative numbers, which this program does with ease.

10.14 MULTI-DIMENSIONAL ARRAYS

Multi-dimensional arrays in C can go beyond two dimensions. Every increase in dimension increases the number of subscripts by one, with the subscript on the right changing faster than the one on the left. A three-dimensional (3D) array can be treated as an array of arrays of arrays. In this case, only those elements accessed with the right-most subscript—with the other subscripts remaining unchanged—occupy consecutive memory cells.

While the principles of memory allocation and initialization remain unchanged for multi-dimensional arrays, visualization of an array becomes difficult when the number of dimensions exceeds three. A 3D array is declared in this manner:

```
int max_temp[YEARS][MONTHS][DAYS];
```

This array can store the maximum daily temperature, grouped by month, for multiple years. Three loops are needed to initialize the elements:

```
for (i = 0; i < YEARS; i++)
   for (j = 0; j < MONTHS; j++)
      for (k = 0; k < DAYS; k++)                    If all values are 0, simply ...
         max_temp[i][j][k] = 0;                     ... initialize one element to 0
```

The symbolic constants representing the subscripts, `MONTHS` and `DAYS`, would have the values 12 and 31, respectively. The month changes after every 31 elements, while the year changes after 31×12 elements. Because all months don't have 31 days, special logic has to be built into a program to skip those elements that point to invalid days.

10.15 USING ARRAYS AS MATRICES

In mathematics, you would have encountered the *matrix* as a rectangular array of numbers or expressions. A matrix is represented by a name, say, A, having the dimensions $m \times n$. Each element of the matrix is represented by the notation, say, $a_{i,j}$, where the subscripts i and j cannot exceed m and n respectively. This is how a 3×4 matrix named A is represented in mathematics:

$$A = \begin{bmatrix} a1,1 & a1,2 & a1,3 & a1,4 \\ a2,1 & a2,2 & a2,3 & a2,4 \\ a3,1 & a3,2 & a3,3 & a3,4 \end{bmatrix}$$

A two-dimensional array can easily implement a matrix except that the array index starts from 0 and not 1. Thus, the matrix element $a_{1,1}$ is the same as the array element `a[0][0]`. In the general case, $a_{m,n}$ is equivalent to `a[`m `- 1][`n `- 1]`. We'll now examine three programs that use 2D arrays to implement the most commonly performed operations on matrices.

10.15.1 `matrix_transpose.c`: Transposing a Matrix

A matrix is *transposed* by swapping its rows and columns. Thus, the first row becomes the first column; the last row becomes the last column. Program 10.13 transposes the matrix represented by the 2D array, `mat1`, and saves the output in the array, `mat2`.

```
/* matrix_transpose.c: Transposes a matrix;
                       converts rows to columns and vice versa. */
#include <stdio.h>
#define COLUMNS 3
#define ROWS 3

int main(void)
{
   short mat1[ROWS][COLUMNS] = {{1, 2, 3},
                                {4, 5, 6},
                                {7, 8, 9}};
   short mat2[ROWS][COLUMNS];
   short i, j;

   for (i = 0; i < ROWS; i++)
      for (j = 0; j < COLUMNS; j++)
         mat2[i][j] = mat1[j][i];

   printf("Transposed matrix:");
   for (i = 0; i < ROWS; i++) {
      putchar('\n');
      for (j = 0; j < COLUMNS; j++)
         printf ("%4hd", mat2[i][j]);
   }
   return 0;
}
```

PROGRAM 10.13: **`matrix_transpose.c`**

```
Transposed matrix:
   1   4   7
   2   5   8
   3   6   9
```

PROGRAM OUTPUT: **matrix_transpose.c**

10.15.2 matrix_add_subtract.c: Adding and Subtracting Two Matrices

Two matrices can be added or subtracted if both of them have the same number of rows and the same number of columns. Program 10.14 adds or subtracts two matrices, `mat1` and `mat2`, depending on the option chosen. Note the conditional expression that uses two **`printf`** expressions as the operands of the ternary operator, `?:`. As in Program 10.8, these expressions are used only for their side effects.

```
/* matrix_add_subtract.c: Adds/subtracts two initialized matrices. */
#include <stdio.h>
#define COLUMNS 4
#define ROWS 3
int main(void)
{
   short mat1[ROWS][COLUMNS] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
   short mat2[ROWS][COLUMNS] = {{12, 11, 10, 9}, {8, 7, 6, 5}, {4, 3, 2, 1}};
   short mat3[ROWS][COLUMNS];
   short i, j, choice;
   printf("Enter 1 for addition or 2 for subtraction: ");
   scanf("%hd", &choice);
   for (i = 0; i < ROWS; i++)
      for (j = 0; j < COLUMNS; j++)
         switch (choice) {
             case 1: mat3[i][j] = mat1[i][j] + mat2[i][j];
                     break;
             case 2: mat3[i][j] = mat1[i][j] - mat2[i][j];
                     break;
            default: printf("Illegal choice\n");
                     return 1;
         }
   (choice == 1) ? printf("Matrix after addition:")
                 : printf("Matrix after subtraction:");
   for (i = 0; i < ROWS; i++) {
      putchar('\n');
      for (j = 0; j < COLUMNS; j++)
         printf ("%4hd", mat3[i][j]);
   }
   return 0;
}
```

PROGRAM 10.14: **matrix_add_subtract.c**

```
Enter 1 for addition or 2 for subtraction: 3
Illegal choice
```

```
Enter 1 for addition or 2 for subtraction: 1
Matrix after addition:
  13  13  13  13
  13  13  13  13
  13  13  13  13
```

```
Enter 1 for addition or 2 for subtraction: 2
Matrix after subtraction:
 -11  -9  -7  -5
  -3  -1   1   3
   5   7   9  11
```

PROGRAM OUTPUT: **matrix_add_subtract.c**

10.15.3 matrix_multiply.c: Multiplying Two Matrices

Matrix multiplication is not as straightforward as addition and subtraction. An $m \times n$ matrix can only be multiplied with an $n \times p$ matrix, which means that the number of columns of the first matrix must equal the number of rows of the second matrix. The resultant matrix has the dimensions $m \times p$.

Matrix multiplication is performed by using the *dot-product* technique. The dot-product of two sets of numbers containing the same number of elements is the sum of the products of the corresponding elements. For matrices, the dot-product applies to the row elements of the first matrix and the column elements of the second matrix (Fig. 10.3).

1 2 3 4 5 6	6 5 4 3 2 1	12 9 6 30 23 16 48 37 26
Matrix 1	**Matrix 2**	**Matrix 3**

FIGURE 10.3 Multiplying Two Matrices

The data for the matrices are obtained from the output of Program 10.15. The first element of matrix 3 (12) is evaluated by computing the dot-product of the first row of matrix 1 and the first column of matrix 2. For the second element (9), the dot-product is applied to the same row of matrix 1 and the second column of matrix 2. Thus, the first row of matrix 3 is evaluated in the following manner:

```
1 × 6 + 2 × 3 = 12      (mat3[0][0])
1 × 5 + 2 × 2 =  9      (mat3[0][1])
1 × 4 + 2 × 1 =  6      (mat3[0][2])
```

The second and third rows of matrix 3 are computed in a similar manner. Program 10.15 performs this dot-product evaluation where the matrix values are supplied by user input. The program also performs a compatibility test of the two matrices.

```
/* matrix_multiply.c: Multiplies two matrices. */
#include <stdio.h>
int main(void)
{
   short mat1[10][10], mat2[10][10];
   short mat3[10][10] = {{0}};     /* Initializes all elements to zero */
   short rows1, cols1, rows2, cols2, rows3, cols3, i, j, k;

   printf("Enter number of rows and columns of matrix 1: ");
   scanf("%hd %hd", &rows1, &cols1);

   printf("Enter number of rows and columns of matrix 2: ");
   scanf("%hd %hd", &rows2, &cols2);

   if (cols1 != rows2) {
      printf("Matrices not compatible; multiplication not possible\n");
      return 1;
   }

   printf("Resultant matrix has the size %hd X %hd\n",
           rows3 = rows1, cols3 = cols2);

   /* Assign and display values for matrix 1 */
   printf("\nMatrix 1: Key in %hd items, %hd items for each row:\n",
           rows1 * cols1, cols1);

   for (i = 0; i < rows1; i++)
      for (j = 0; j < cols1; j++)
         scanf("%hd", &mat1[i][j]);

   printf("Entered values for matrix 1: ");
   for (i = 0; i < rows1; i++) {
      putchar('\n');
      for (j = 0; j < cols1; j++)
         printf ("%4hd", mat1[i][j]);
   }

   /* Assign and display values for matrix 2 */
   printf("\n\nMatrix 2: Key in %hd items, %hd items for each row:\n",
           rows2 * cols2, cols2);

   for (i = 0; i < rows2; i++)
      for (j = 0; j < cols2; j++)
         scanf("%hd", &mat2[i][j]);

   printf("Entered values for matrix 2: ");
   for (i = 0; i < rows2; i++) {
      putchar('\n');
      for (j = 0; j < cols2; j++)
         printf ("%4hd", mat2[i][j]);
   }
```

```
/* Multiply the matrices and display values for matrix 3 */
   for (i = 0; i < rows3; i++)
      for (j = 0; j < cols3; j++)
         for (k = 0; k < cols1; k++)
             mat3[i][j] += mat1[i][k] * mat2[k][j];

   printf("\nComputed values for matrix 3: ");
   for (i = 0; i < rows3; i++) {
      putchar('\n');
      for (j = 0; j < cols3; j++)
         printf ("%4hd", mat3[i][j]);
   }
   return 0;
}
```

PROGRAM 10.15: **matrix_multiply.c**

```
Enter number of rows and columns of matrix 1: 3 2
Enter number of rows and columns of matrix 2: 2 3
Resultant matrix has the size 3 X 3
Matrix 1: Key in 6 items, 2 items for each row:
1 2 3 4 5 6
Entered values for matrix 1:
   1   2
   3   4
   5   6
Matrix 2: Key in 6 items, 3 items for each row:
6 5 4 3 2 1
Entered values for matrix 2:
   6   5   4
   3   2   1
Computed values for matrix 3:
  12   9   6
  30  23  16
  48  37  26
```

PROGRAM OUTPUT: **matrix_multiply.c**

10.16 VARIABLE LENGTH ARRAYS (C99)

Finally, let's briefly discuss one of the most beneficial products to emerge from the C99 standard. This is the *variable length array (VLA)* which can be declared *after* obtaining the size of the array at runtime. In Program 10.16, the array `arr` is declared only after its size is obtained from a call to **`scanf`**.

The output confirms that `size` has been rightly set to `n`. Note that you are able to set the size of an array at runtime without changing the program at all. The expression `sizeof(arr) / sizeof(arr[0])` evaluates to the number of array elements *(10.3)*.

```
/* variable_length_array.c: Declares an array with its size determined
                            at runtime. Works only in C99. */
#include <stdio.h>
int main(void)
{
   short i, n, size;
   printf("Enter the number of elements in array: ");
   scanf("%hd", &n);           /* Index of array determined at runtime */
   short arr[n];                                /* Can't do this in C89 */
   size = sizeof(arr) / sizeof(arr[0]);         /* size is the same as n */
   printf("Size of this array is set at runtime to %hd\n", size);
   printf("Key in %hd integers: ", n);
   for (i = 0; i < size; i++)
      scanf("%hd", &arr[i]);
   for (i = 0; i < size; i++)
      printf("%hd ", arr[i]);
   return 0;
}
```

PROGRAM 10.16: **variable_length_array.c**

```
Enter the number of elements in array: 6
Size of this array is set at runtime to 6
Key in 6 integers: 11 33 55 77 22 44
11 33 55 77 22 44
```

PROGRAM OUTPUT: **variable_length_array.c**

Note: In C89, space is dynamically created at runtime by the **malloc**, **calloc** and **realloc** functions. Even though C99 offers a simpler alternative, we need to know how these three functions work. They are discussed in Chapter 16.

WHAT NEXT?

We are not done with arrays yet. We have to learn to interpret a *string* as an array. We must also be able to pass arrays as arguments to *functions*. That will be possible after we have understood the relationship arrays have with *pointers*. We'll examine functions before we take up pointers and strings.

WHAT DID YOU LEARN?

An *array* is a set of contiguous memory cells that can store multiple data items having the same data type. The compiler doesn't complain if an array element is accessed with an illegal index.

An uninitialized array contains junk values. A partially initialized array sets the uninitialized elements to zero or NUL.

You can insert an element at a specific location by moving the remaining elements to the right. For deletion, the remaining elements are moved to the left. An array can be reversed with or without using a second array.

An array can be sorted using different algorithms. The *selection sort* mechanism scans an array in multiple passes and progressively moves the start point of the next search closer to the end point.

An unsorted array can only be searched sequentially. *Binary search* is faster but it works only on sorted arrays. Binary search halves the search domain in every pass that scans the array.

Two-dimensional (2D) arrays are interpreted as an array of single-dimensional arrays. A 2D array can be used as a frequency counter.

A 2D array can also be used as a *matrix*. Two compatible matrices can be added or subtracted. An $m \times n$ matrix can only be multiplied with an $n \times p$ matrix to produce an $m \times p$ matrix.

In a multi-dimensional array, the right subscript changes faster than the left subscript.

C99 supports a *variable length array (VLA)*, which lets you determine the size of an array at runtime.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

10.1 The elements of an array must have the same data type.

10.2 The compiler issues an error when you access an array with a negative index.

10.3 If an array is partially initialized, the uninitialized elements have junk values.

10.4 The size of an array cannot be changed while the program is running.

10.5 It is not possible to reverse an array without using a second array.

10.6 When declaring and initializing a 2D array, the number of columns need not be specified.

10.7 In C, there is no specified limit on the number of dimensions an array can have.

10.8 A 5×3 matrix can be multiplied by a 3×5 matrix to output a 5×5 matrix.

A2. FILL IN THE BLANKS

10.1 The elements of an array are laid out _____ in memory.

10.2 All array elements can be printed by using the _____ as the key variable in a loop.

10.3 The `[ ]` symbols in the array `arr[ ]` represent an _____.

10.4 _____ search on an array is faster than sequential search.

10.5 A two-dimensional array can be interpreted as an _____ of arrays.

10.6 The array `a[i][j]` can be interpreted as `i` _____ and `j` _____.

10.7 Assuming a 4-byte `int`, the `int` array `a[10][5]` occupies _____ bytes of memory.

10.8 In the memory layout of the array `a[5][10]`, the element `a[3][0]` is preceded by _____.

10.9 In C99, it is possible to specify the size of an array at _____.

A3. MULTIPLE-CHOICE QUESTIONS

10.1 If an array is declared without specifying its size, (A) it must also be initialized, (B) the declaration is illegal, (C) the size can be specified later, (D) none of these.

10.2 The initialized values of an array are enclosed by (A) `[]`, (B) `()`, (C) `{}`, (D) none of these.

10.3 If `arr` is declared as an `int` array of 10 elements, the statement **`scanf("%d", arr);`** (A) fails on compilation, (B) fails at runtime, (C) works perfectly, (D) produces implementation-dependent behavior.

10.4 The maximum number of iterations needed to conduct a binary search on a sorted array of 1000 elements is (A) 999, (B) 256, (C) 32, (D) 10.

10.5 The statement `int a[5][5] = {{1}};` (A) sets all elements to 1, (B) sets `a[0]0]` to 1 and the remaining elements to zeroes, (B) sets only `a[0][0]` to 1, (D) fails on compilation.

10.6 For initialization of a 2D array, the inner braces are (A) mandatory for partial initialization, (B) optional for partial initialization, (C) optional, (D) mandatory.

CONCEPT-BASED QUESTIONS

10.1 If an integer array has 100 elements, will you adopt the same technique for initializing all of its elements to 0 and, say, 10?

10.2 Explain what is meant by saying that the size of an array can be set explicitly or implicitly.

10.3 If `arr` is an array of type `int`, what is the significance of the expression `sizeof arr / sizeof(int)`? What is the advantage of using this expression?

10.4 How is the number of loop iterations needed to search a sorted array related to its size?

10.5 If the array `arr` is defined in the following manner, what will be the value of `arr[2][0]` and why?

```
short arr[4][6] = { 0, 1, 2, 3, 4,
                    10, 11, 12, 13, 14,
                    20, 21, 22, 23, 24 };
```

PROGRAMMING & DEBUGGING SKILLS

10.1 Correct the following code segment to compile and execute correctly:

```
int i;
short arr[] = {1, 2, 3, 4};
for (i = 1; i <= sizeof arr / sizeof short; i++)
   printf(" %hd\n", arr[i]);
```

10.2 Write a program that prints the maximum and minimum values stored in an integer array containing 10 elements. The array is populated with user input.

10.3 Write a program that declares and initializes a 10-element array with arbitrary values. The program should accept an integer from the keyboard, delete all matching array elements and print the modified array in reverse sequence.

10.4 Use **`switch`** in a **`do-while`** loop to set the elements of the array, `month[13]`, to the maximum number of days of each month. Print the array in the format `month[`*n*`]` = *value*. (Ignore `month[0]`.)

10.5 Write a program to input a set of integers and store the values in an array after discarding duplicates. Print the array and a count of the number of duplicates found.

10.6 Modify the preceding program in C10.5 to store odd and even integers between 0 and 99 in two separate arrays but after discarding duplicates. Merge the two arrays into a third array by running a loop which looks at both arrays and creates a sorted list but without using a sorting algorithm.

10.7 Write a program using a 26-element integer array to print the number of occurrences of each lowercase letter fetched from the keyboard using **`getchar`**. Also print the number of discarded characters in the input.

10.8 Write a program that populates a 2D integer array of 3 rows and 4 columns with user input. The program must discard duplicates.

10.9 Correct the following statement which leads to compilation errors:

```
short arr[3][] = { {0, 1, 2, 3, 4},
                   {10, 11, 12, 13, 14},
                   {20, 21, 22, 23, 24}; }
```

10.10 Write a program that compares for equality two 2D arrays having 3 rows and 5 columns. Print the unequal elements in the form `element[`*m*`][`*n*`]` = *value1*, *value2*.

10.11 Write a program that initializes a 24-element integer array, `arr`, with arbitrary values and then uses that data to populate a (i) 2D array, `brr[4][6]`, (ii) 3D array, `crr[2][3][4]`. Also, print the 2D array.

10.12 Write a program to compute the sum of all rows and columns of a 3×4 matrix which is populated by user input.

10.13 Write a program to check whether the two diagonals of a 3×3 matrix contain the value 1, and the remaining elements have the value 0. The matrix is populated by user input.

11 Functions

WHAT TO LEARN

- Techniques of *declaring*, *defining* and invoking a function.
- Exploring various ways of handling the *return value* of a function.
- Mechanism of transfer of data from *arguments* to *parameters*.
- Whether C passes arguments *by value* or *by reference*.
- Significance of *side effect* of a function.
- Passing single- and two-dimensional arrays as function arguments.
- Consequences of calling one function from another.
- Techniques of using a function in a *recursive* manner.
- Using *storage classes* to determine the *scope* and *lifetime* of a variable.

11.1 FUNCTION BASICS

C programmers decompose a complex task into independent and manageable modules called functions. Code sections that are used repeatedly—even by other programs—are also implemented as functions. It is easier to develop and maintain a C program that delegates most of its work to separate functions compared to one that uses a monstrous **`main`** function to do everything. While functions like **`printf`** and **`scanf`** are meant to be treated as black boxes, you must know how to create functions of your own and integrate them into the main program.

A *function* is a statement that represents a named body of program code. When it is called or invoked, the code associated with the function is executed. A function can optionally (i) accept one or more values as *arguments*, and (ii) report the outcome of its action by *returning* a single value to the caller. A function is called mainly in one of these two ways:

```
message_of_day();                    No argument, returns no value
fahrenheit = c2f(40.2);              One argument, returns a value
```

A function is easily identified from the matched pair of parentheses that follow its name. The `message_of_day` function is neither passed an argument nor does it return a value. But the **`c2f`** function accepts an argument (`40.2`) and also returns a value, which here is saved in the variable `fahrenheit`. This saved value can be used later in the program. While all functions don't accept arguments or return a value, the vast majority of them do.

A function must be declared and defined before it is called or invoked. The *declaration* specifies how to invoke the function correctly. The *definition* provides the implementation, i.e. , the body of code that will be executed on invocation. Unlike **`message_of_day`** which behaves in identical manner every time it is invoked, the behavior of **`c2f`** is determined by the value of the argument (here, `40.2`) that is passed to it.

The functions you create can also call other functions. A function can even call itself, a property that has important applications in the programming world (like calculating the factorial of a number). For all of the functions we have used so far, we had **`main`** as the caller. This is a special function that every standalone C program must have because a program commences execution by calling **`main`**. We have more to say on the **`main`** function later in this chapter.

In this chapter, we'll create some functions from scratch and rewrite some of our previous programs using them. In each case, the implementation (i.e., definition) of a function occurs in the same file that calls it. So a function defined in one program can't be reused in another program without replicating the declaration and definition. We'll use the techniques mandated by ANSI for creating functions and ignore the techniques originally proposed by K&R C. Chapter 17 examines the techniques of isolating functions into separate files so they can be used by all programs.

Takeaway: A function must be declared and defined before it can be used. The declaration specifies the usage and the definition provides the code that is executed.

11.2 first_func.c: NO ARGUMENTS, NO RETURN VALUE

Let's now consider our first program, (Program 11.1), that features a simple function named **`message_of_day`**. The function is declared before **`main`**, called from the body of **`main`** and defined after **`main`**. The program prints four lines—two each by **`main`** and the function.

The function, **`message_of_day`**, is declared right after the preprocessor directive and before **`main`** in this manner:

```
void message_of_day(void);                    Function declared
```

The declaration or prototype doesn't specify what **`message_of_day`** does, but only how to invoke it. Since every declaration is also a statement, it must be terminated with a semicolon. This function uses no arguments (second `void`) and returns no value (first `void`) and must thus be called in this manner:

```
message_of_day();                             Function called or invoked
```

What **`message_of_day`** is designed to do is specified in its definition that follows **`main`**. By convention, all function definitions are located after **`main`**, even though they may be placed before **`main`** as well.

```
/* first_func.c: Uses a function that has no arguments and returns nothing. */
#include <stdio.h>
/* Function declaration or prototype */
void message_of_day(void);                          /* Try deleting this statement */
int main(void)
{
   printf("Invoking function ...\n");
   message_of_day();                                /* Function invocation or call */
   printf("Returned from function.\n");
   return 0;
}
/* Function definition or implementation */
void message_of_day(void)                           /* The header */
{
   printf("This function uses no arguments and returns nothing.\n");
   printf("The next program uses one argument and returns a value.\n");
   return;
}
```

PROGRAM 11.1: **first_func.c**

```
Invoking function ...
This function uses no arguments and returns nothing.
The next program uses one argument and returns a value.
Returned from function.
```

PROGRAM OUTPUT: **first_func.c**

The definition contains both a *header* and a body enclosed by a pair of curly braces. This body comprises two **printf** statements and a **return** statement without a value.

The compiler finds the function declaration consistent with the definition and invocation. When **message_of_day** is called, program control moves to the function body where it executes all statements placed there. The **return;** statement in the function moves control back to the caller (i.e., **main**). Execution then continues with the next statement following the function call.

The **return;** statement in the function has the same significance as the one we have all along used in **main**. It terminates the function and returns control to the caller. However, this function doesn't return a value (**return;** and not **return 0;**). Even though a **return** here is not necessary, it's good programming practice to include it in every function even though it may return nothing.

Takeaway: A function name occurs in at least three places in a program. The declaration occurs at the beginning, the invocation takes place in the body of **main**, and the definition is placed after **main**. The compiler will output an error if their forms do not match.

Note: We don't need to provide the definition for functions of the standard library, but we still need their declaration. The file `stdio.h` contains the declarations for **`printf`** and **`scanf`**, the reason why we have included this file in every program so far.

11.3 THE ANATOMY OF A FUNCTION

A function is an identifier, which implies that the rules related to the naming of variables and arrays also apply to functions *(5.2.1)*. Thus, the name can comprise letters, digits and the underscore character where the first character cannot be a digit. Case is significant, so, **`area`** and **`AREA`** are two separate functions. Let's now examine the three main attributes of functions: declaration, definition and invocation.

11.3.1 Declaration, Prototype or Signature

A *declaration* is an authentic statement that tells the compiler how the function is invoked. The declaration is also known as *prototype* or *signature*. Since it is placed before **`main`**, the compiler sees it first and then compares it to the invocation and definition. For this comparison, the compiler determines whether the function

- uses any arguments, and if so, the number of such arguments along with their data types.
- returns a value, and if so, its data type.

This creates four possible situations—with or without arguments, and with or without return value. The following syntax and supporting example show how to declare a function that neither has an argument nor a return value:

```
void function_name(void);
void message_of_day(void);                    Used in first_func.c
```

The first `void` indicates that the function doesn't return a value. The second `void` signifies that the function uses no arguments. However, most functions use one or more arguments and also return a value using the following generalized syntax:

```
return_type function_name(type1 arg1, type2 arg2 ...);
```

Here, *return_type* represents the data type of the return value. The function arguments are represented as *arg1*, *arg2* and so on, where each argument is preceded by its data type, *type1*, *type2* and so forth. An argument can be a variable, constant or expression. Here's an example:

```
double area(float length, float breadth);
```

The **`area`** function accepts two arguments each of type `float` and returns a value of type `double`. Since the compiler simply matches the type and number of arguments in the declaration with those used in the definition, it doesn't need to know these variable names, and C lets you omit them:

```
double area(float, float);                    Not recommended
```

The prototype has been compacted at the expense of readability. (You can't do the same in the definition.) The declarations are the first things we see when viewing the source code. A single-line declaration should tell us what the function does and what input it needs. By using meaningful names for both the function and its arguments, we can avoid using separate comment lines.

Note: A function prototype or signature serves as a good reference and prevents you from wrongly invoking a call. Never fail to include it even if you don't encounter an error on its exclusion.

11.3.2 Definition or Implementation

The function *definition* or *implementation* specifies what the function does when invoked. It comprises a *header* and a body where all the work gets done. For a function that accepts arguments and returns a value, the definition takes this generalized form:

```
return_type function_name(type1 arg1, type2 arg2 ...)      Header
{                                                          Beginning of body
    statements;
    return expression;
}                                                          End of body
```

The header (the first line) is virtually identical to the declaration except that there's no semicolon as terminator. The body is represented by a simple or compound statement that is compulsorily enclosed within curly braces. The **return** statement transmits the value of *expression* back to the caller. The **area** function that we just declared would have the following as its definition:

```
double area(float length, float breadth)
{
   double product;                                         Local variable of function
   product = length * breadth;
   return product;
}
```

This header contains a comma-separated list comprising the two *parameters*, `length` and `breadth`. These parameters, which are actually variables, are assigned by their corresponding *arguments* used in the invocation. That is, if the function is invoked as **`rect_area = area(len, width);`**, the values of the arguments, `len` and `width`, are passed to the parameters, `length` and `breadth`, respectively.

The body declares a local variable named `product` to save the result of a computation. The **return** statement finally passes this result back to the caller of the function. The caller often uses a variable to save this value before it disappears after termination of the function. The **area** function doesn't print anything but simply returns a value.

For functions that don't return a value (like **`message_of_day`**), **return** is used (if at all) without *expression*. In that case, *return_type* must be specified as `void` both in the declaration and definition.

11.3.3 Invocation or Call

Once a function has been declared and defined, you know what it does and how it does it. You can then call it as many times as you want, either from **main** or from any other function. If the function doesn't return a value, it is invoked simply like this:

```
message_of_day();
```

For functions that return a value, the method of invocation is often different. The programmer must invoke the function in a way that enables the program to "see" the value. Typically, the return value is saved in a variable or used as an argument to another function:

```
rect_area = area(10.0, 20.0);                    Saves 200.0 in rect_area
printf("Area = %f\n", area(10.0, 20.0));         area evaluated first
```

The compiler normally takes a serious view of type mismatches, but it is a trifle lenient when it looks at the invocation. If **area** is called with two `int`s rather than two `float`s, the compiler won't complain. It is your job to take care of any possible data loss when invoking a function with non-exact data types in its arguments.

There is one more mismatch that you need to take care of. If a function that returns a value (say, `float`) in the function body but doesn't specify a return type in the declaration, the compiler assumes that the function returns an `int`. C11 is strict though; it requires the return type to be explicitly specified.

Note: Both **printf** and **scanf** return a value, but more often than not, we ignore this value. We use these functions mainly for their side effect. However, you would have noticed that in some cases *(9.10.4, 10.4, 10.5.2, 10.10.2)*, these functions have been used both for their return value and side effect.

HOW IT WORKS: How Standard Library Functions Are Declared and Defined

The signatures for functions of the standard library are distributed across several "include" files (having the extension `.h`) that are read by the preprocessor. One of them, `stdio.h`, contains—either directly or indirectly—the prototypes of all functions used for I/O operations. We include this file simply because we always use an I/O function in our programs. When we use the string-handling functions, we'll include `string.h`.

The definitions for these functions are available in compiled form in a library from where they are extracted by the linker during the compilation process. For all ANSI C functions, we need their compiled code and not the source code. This is consistent with the black-box approach that we need to adopt for functions "designed by others."

11.4 c2f.c: ONE ARGUMENT AND RETURN VALUE

Program 11.2 uses a function named **c2f** to convert a temperature from Celsius to Fahrenheit using the well-known formula that was used in Chapter 6. This function accepts the temperature in Celsius as argument and returns the converted value in Fahrenheit to **main**.

The **c2f** function has been invoked in two ways: (i) by saving its return value, and (ii) as an argument to **printf**. When **c2f** is invoked, the value of its *argument*, `celsius`, is passed to its corresponding *parameter*, `f_celsius`. The function also uses a local variable (`fheit`) to return the converted value to **main**. In the first invocation of **c2f**, this value is assigned to a variable. The next invocation uses **c2f** as an expression in **printf**.

```
/* c2f.c: Uses a function that accepts an argument and returns a value. */
#include <stdio.h>
float c2f(float cgrade);                    /* Function declaration */

int main(void)
{
   float celsius, fahrenheit;

   printf("Enter a temperature in Celsius: ");
   scanf("%f", &celsius);
   fahrenheit = c2f(celsius);               /* Function called here */
   printf("%.2f Celsius = %.2f Fahrenheit\n", celsius, fahrenheit);
   printf("%.2f Celsius = %.2f Fahrenheit\n", celsius, c2f(celsius));

   return 0;
}
float c2f(float f_celsius)                  /* Function definition */
{
   float fheit;                             /* Local variable of function */
   fheit = f_celsius * 9 / 5 + 32;
   return fheit;                            /* Value returned to main */
}
```

PROGRAM 11.2: **c2f.c**

```
Enter a temperature in Celsius: 40.5
40.50 Celsius = 104.90 Fahrenheit                    From 2nd printf statement
40.50 Celsius = 104.90 Fahrenheit                    From 3rd printf statement
```

```
Enter a temperature in Celsius: 0
0.00 Celsius = 32.00 Fahrenheit
0.00 Celsius = 32.00 Fahrenheit
```

```
Enter a temperature in Celsius: -40
-40.00 Celsius = -40.00 Fahrenheit
-40.00 Celsius = -40.00 Fahrenheit
```

PROGRAM OUTPUT: **c2f.c**

Note: The name used for the function argument (`celsius`) need not be different from the one used for the parameter (`f_celsius`). Even though Section *11.5.2* explains why that is so, you should still use different names to avoid confusion.

11.5 ARGUMENTS, PARAMETERS AND LOCAL VARIABLES

A function uses its own set of variables for carrying out its tasks. In the definition of **c2f** (Program 11.2), they are seen in the parameter (`f_celsius`) and function body (`fheit`). These variables are created in a separate region of memory that is almost exclusively reserved for functions. When a function terminates, this space is deallocated by the operating system and made available for the next function call. The following sections explain how these factors affect the accessibility and lifetime of variables used in a function.

11.5.1 Parameter Passing: Arguments and Parameters

To understand the concept of parameter passing, we need to look at function arguments from two viewpoints—the caller and called function. Consider the following function definition and invocation taken from the previous program, **c2f.c**:

```
float c2f(float f_celsius) {...}                    Definition
fahrenheit = c2f(celsius);                          Invocation
```

When the caller (here, **main**) invokes a function, it assigns values to the *arguments* or *actual arguments* of the function. The called function accepts these values into its *parameters* or *formal arguments*. When we invoke **c2f** as shown above, the value of `celsius` (the argument) is assigned to `f_celsius` (the parameter). This transfer of value from argument to parameter is known as *parameter passing*.

Unfortunately, the terms *argument* and *parameter* are often used interchangeably. In this text, however, *argument* means the "actual argument" and *parameter* means the "formal argument" to a function.

Note: An *argument* and *parameter* represent two sides (the caller and called function) of the same coin. A function call causes data to flow from the argument to the parameter. The reverse is, however, not possible (though we can simulate the reverse flow using pointers).

11.5.2 Passing by Value vs Passing by Reference

We must now clearly understand how arguments are passed to parameters of the called function. There are some conflicting points of view that have often been fiercely debated by the C community. The question is whether arguments are passed *by value* or *by reference*, or whether there is a mix of both. We'll attempt to resolve this conflict in this section.

In C, all function arguments are *passed by value*, i.e., they are *copied* to their corresponding parameters. *There is no exception to this principle.* When **c2f** is called, the *value* of `celsius` is copied to `f_celsius`. Since a parameter resides in its own memory space, `celsius` and `f_celsius` occupy separate memory locations (Fig. 11.1). You can't subsequently change `celsius` by changing `f_celsius`, i.e., you can't change an argument by changing its copy, the parameter.

It is thus not necessary to maintain separate names for arguments and parameters in the declaration, definition and invocation. In the program **c2f.c**, we could have used the same variable name (`celsius`) in the function definition as well.

Unlike C++, C doesn't support passing by reference, where both argument and parameter occupy the same memory location. However, in C, a variable (`p`) can store the memory address of another variable (`x`), and you can use `p` to *indirectly* change the value of `x`. Thus, if you invoke a function with `p` as argument, its copied parameter will still contain the address of `x`. You can then use this copy of `p` in the function to change `x` that is defined in the caller. Working inside a function you can change a variable defined outside the function!

Some like to think of the above feature as passing by reference or "the effect" of passing by reference, but we'll not take this view and continue to maintain that C passes arguments by value only. Let's now summarize our observations in the following manner:

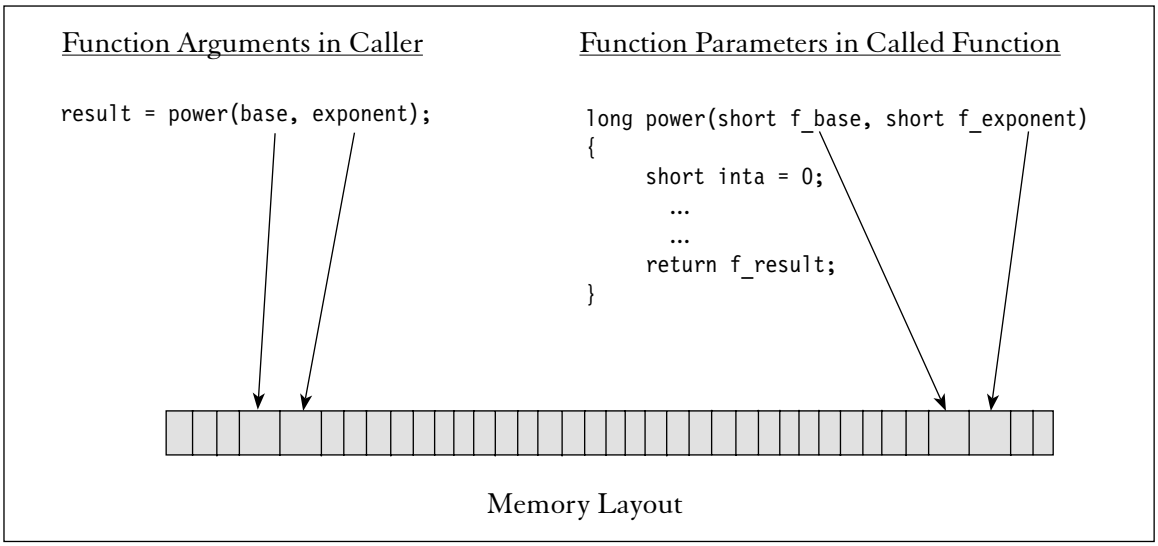


FIGURE 11.1 How Function Arguments and Parameters Are Organized in Memory

When a function is called, its arguments are copied to the parameters. But if the function changes a parameter, then the change is not seen in the caller. However, if an argument contains the address of a variable, then the corresponding parameter of the argument can be used to change that variable.

The preceding paragraphs relating to the use of memory addresses provided a gentle introduction to pointers, touted as the strongest feature of C. We'll unravel the mystery of pointers in Chapter 12, but we'll encounter the topic in a disguised form when we use arrays as function arguments in Section *11.8*.

Note: Even if the changed value of a parameter is not seen by the caller, a function can make it available through the **`return`** statement. However, a function can return only one value, so you can't return multiple parameters in this manner.

11.5.3 Local Variables

A function has its own set of *local* variables that are defined in its implementation. For instance, the **`c2f`** function uses a local variable, `fheit`, to compute the result it returns to its caller. Unlike parameters, local variables cannot be assigned by the caller. However, parameters and local variables have the following features in common:

- Their names don't conflict with identical names defined in the caller or any other function.
- They can be accessed only in the function they are defined in. They are neither *visible* in the caller nor in any other function.
- They last as long as the function is active. This means that they are created every time the function is called.

From the above and as shown in Figure 11.1, we can conclude that every function call has a separate area earmarked for it. This area is deallocated after the function terminates. But since **main** is the first function to be called and the last to terminate, its variables have a longer *lifestime* compared to variables and parameters of other functions. Visibility and lifetime are important attributes of variables that we need to address, and soon we will.

11.5.4 swap_failure.c: The "Problem" with Local Variables

Program 11.3 demonstrates the limited visibility of function parameters and local variables. It makes an unsuccessful attempt to swap the values of two variables that are passed as arguments to a function. In the process, it establishes that the area of memory allocated to function arguments is separate from that used by the parameters and local variables of a function.

```
/* swap_failure.c: Establishes the impossibility of swapping two values
                   using parameters and local variables of a function. */
#include <stdio.h>
void swap(short x, short y);              /* Function returns no value */
int main(void)
{
   short x = 1, y = 100, temp = 0;
   printf("In main before swap: x = %hd, y = %hd\n", x, y);
   swap(x, y);
   printf("In main after swap: x = %hd, y = %hd\n", x, y);
   printf("In main temp seen as %hd\n", temp);
   return 0;
}
void swap(short x, short y)               /* x and y local to swap */
{
   short temp;                            /* Local variable */
   temp = x;
   x = y;
   y = temp;                              /* Values swapped inside function */
   printf("In swap after swap: x = %hd, y = %hd\n", x, y);
   printf("In swap temp set to %hd\n", temp);
   return;
}
```

PROGRAM 11.3: **swap_failure.c**

```
In main before swap: x = 1, y = 100
In swap after swap: x = 100, y = 1
In swap temp set to 1
In main after swap: x = 1, y = 100
In main temp seen as 0
```

PROGRAM OUTPUT: **swap_failure.c**

To reinforce our point, we have used identical variable names (x, y and temp) in both calling and called function. The **swap** function interchanged the values of x and y all right, but it changed the parameters and not the original arguments. In **main**, x and y remain unchanged at their initial values (1 and 100) even after swapping. The function also failed to change the the value of temp in **main** by changing its own local variable of the same name.

Because the variables of a function have their own memory space, changes made to them can't impact the variables used in the caller. However, we had observed in Section 11.5.2 that a function *can* change the value of a variable defined in its caller only if it knows the address of that variable. Using this property, we'll modify this **swap** function in Chapter 12 to make it finally work. You'll soon find that this restriction also vanishes when the argument is an array.

11.6 THE RETURN VALUE AND SIDE EFFECT

For a function that returns a value, C considers the transmission of this value to be its primary task. Any other action performed by the function body is considered to be the *side effect*. Displaying a message, fetching a line of text, evaluating an expression or updating an external file are some of the common side effects seen in functions. From this viewpoint, the printing of a string with **printf** and accepting keyboard input by **scanf** are considered by C to be side effects.

Some functions return a value while others don't. A function that returns a value is evaluated as an expression, whose value is the return value of the function. We often save this return value for further processing or directly use it as an argument to another function. We have done both using the **c2f** function:

```
fahrenheit = c2f(celsius);
printf("%.2f Celsius = %.2f Fahrenheit\n", celsius, c2f(celsius));
```

If a function that returns a value also has a side effect, it is for the programmer to determine whether the function is to be used for its side effect, or return value, or both. In all of the programs we have used so far, **printf** and **scanf** have always been used for their side effect. It's only in a handful of cases that we have taken advantage of their return value as well:

```
while (scanf("%hd", &arr[i++]) == 1)                    (10.4)
while (printf("Integer to search (q to quit): "))        (10.10.2)
```

Here, **scanf** is used both for reading keyboard input (side effect) and also detecting the end of input (return value). Similarly, **printf** is used for displaying a message (side effect) and to return a true value so that the **while** loop runs forever.

There are times when you would want the return value to be interpreted as true or false. For instance, if you want a function to conduct a leap year check in the following manner:

```
if (is_leap_year(2016))                    Function may return either 0 or 1
```

it means that the **is_leap_year** function simply needs to return a 0 or 1. The possible definition of this function could then be this:

```
int is_leap_year(short year)
{
   if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
      return 1;                              /* Returns true */
   else
      return 0;                              /* Returns false */
}
```

A function that doesn't return a value can't be used in any of the ways shown previously. Because the **message_of_day** function *(11.2)* doesn't return a value, you can neither assign it to a variable nor use it in a control expression. A function that belongs to this category can't have any side effect because whatever it does is its only effect. Most functions of the standard library return a value but there are three that don't: **abort**, **exit** and **assert**.

11.7 prime_number_check2.c: REVISED PROGRAM TO CHECK PRIME NUMBERS

Program 11.4 modifies a previous version (Program 8.9) to demonstrate some features of the return value. It takes a number from a user, uses one function (**is_not_prime**) to check for prime and then another one (**to_continue**) to determine whether to continue or not. Since the algorithm for determining a number for prime has already been discussed, we'll confine our discussions mainly to the way the return values of the two functions are captured and used.

```
/* prime_number_check2.c: Modifies a previous version to use a function for
   determining a prime number. Shows intelligent use of return value. */
#include<stdio.h>
int is_not_prime(int num);
int to_continue(void);
int main(void)
{
   int number, divisor;
   while (printf("Number to test for prime: ")) {
      scanf("%d", &number);
      while (getchar() != '\n') ;          /* Note ; is on same line this time */
      divisor = is_not_prime(number);      /* Is 0 for prime number ... */
      if (divisor > 0)                     /* ... the lowest factor otherwise */
         printf("%d is not prime, divisible by %d\n", number, divisor);
      else
         printf("%d is prime\n", number);
      if (!to_continue())                  /* If not true, i.e. is 0 */
         break;
   }
   return 0;
}
```

```
int is_not_prime(int num)
{
   int divisor = 2;
   while (divisor < num) {
      if (num % divisor == 0)
         return divisor;                  /* divisor is non-zero */
      divisor++;                          /* Try next higher number */
   }
   return 0;                              /* Executed when number is prime */
}

int to_continue(void)
{
   char answer;
   printf("Wish to continue? ");
   answer = getchar();
   if (answer == 'y' || answer == 'Y')
      return 1;
   else
      return 0;
}
```

PROGRAM 11.4: **prime_number_check2.c**

```
Number to test for prime: 13
13 is prime
Wish to continue? y
Number to test for prime: 377
377 is not prime, divisible by 13
Wish to continue? n
```

PROGRAM OUTPUT: **prime_number_check2.c**

The **printf** function always returns a non-zero value (the number of characters written), so it is a valid control expression for an infinite loop. The number input with **scanf** is checked for prime by the function **is_not_prime**. The return value is saved in the variable `divisor` because it is subsequently used twice.

Once a number has been established as prime or otherwise, the **to_continue** function is invoked to determine whether the user wants to input another number. The function definition shows 0 and 1 as the only possible return values, which implies that the function can be used as a logical expression. When this expression is used with the NOT operator (!), the **while** loop in **main** is exited only if **to_continue** evaluates to false.

11.8 USING ARRAYS IN FUNCTIONS

Functions also use arrays as arguments. You can pass both an individual array element and the name of an array as argument to a function. The principles of parameter passing apply to arrays too, but

they affect array elements and entire arrays in opposite ways. Let's first consider this function call that uses an array element as argument:

```
validate(month[2]);
```

This is a simple pass by value; the value of `month[2]` is copied to its respective parameter—probably a simple variable—in the function. You can't thus modify this element by changing its copy inside the **`validate`** function. However, this is not the case when the name of the array is passed as an argument:

```
init_array(month, 12);
```

Here, `month` does *not* signify the complete array, so the entire array is not copied inside the function. Instead, `month` signifies the *address of its first element* (i.e., `&month[0]`), and it is this address that is copied. The function can then compute the address of the remaining elements using simple arithmetic. This means that the function can change every element of `month` even though it is defined in the caller!

However, there is one problem. Even though **`init_array`** has been passed an address, it has no way of knowing that this address refers to an array. But if the size of the array is also passed as a separate argument, **`init_array`** can then use this size to identify a block of memory *and treat this block as an array*. It can then compute the addresses of the remaining "elements" without straying beyond the boundaries of the "array." A classic case of deception but one that works perfectly!

Finally, in a declaration or definition of the function that uses an array name (say, `month`) as argument, the array name is specified as `month[]` or `month[`*n*`]`, where *n* is any integer. But the function call uses only the array name, as the following declarations and invocation show:

```
void init_arr(short month[], short size);          Declaration uses name + []
void init_arr(short month[12], short size);        This declaration is also OK
init_arr(month, 12);                               Invocation uses simply name
```

In the second example, the compiler doesn't interpret the subscript 12 as the size of the array. It simply looks at the `[]` to know that it is dealing with an array. It thus makes no difference whether you use `month[]`, `month[12]`, or `month[65000]` in the declaration (or header of definition). But the compiler has no way of knowing the size of this array, so the size must be passed as a separate argument.

If the array contains a string (an array of type `char` terminated by `'\0'`), a function *can* determine the size of the string without knowing the size of the array. Strings are discussed in Chapter 13.

Takeaway: Array elements, when used as function arguments, are copied to their respective parameters in the usual way. But when the name of an array is passed as a function argument, what is copied is the address of the first element, and not the entire array. However, because the function knows this address, it can change the entire array provided it also knows the number of array elements.

11.8.1 input2array.c: Passing an Array as an Argument

Program 11.5 uses a function to populate an array with keyboard input. This function uses the name of the array and its size as the two arguments. After the function completes execution, the program prints the values of all array elements. The output shows that the initialization of the "array", f_arr, inside the function has actually changed arr in **main**, thus confirming our assertion that the array itself is never copied inside a function.

Since arr is the same as &arr[0], both &arr[0] and &f_arr[0] represent the address of the same array element. But f_arr is not an array even if the declaration and definition suggest that it is one (f_arr[]). The **input2array** function, however, treats f_arr as an array and uses the other argument (SIZE or num) to populate it correctly. Thus, changing f_arr inside the function actually changes arr outside the function.

Let's now face the truth once and for all: *Both* arr *and* f_arr *are pointers*. The declaration could have specified short * f_arr instead of short f_arr[], and it would have made no difference to the existing implementation of the function. (See it for yourself by making this change.) But hold

```
/* input2array.c: Uses a function to fill up array with keyboard input. */
#include <stdio.h>
#define SIZE 6
void input2array(short f_arr[], short num);    /* Note the [] */
int main(void)
{
   short arr[SIZE], i;
   input2array(arr, SIZE);                     /* Array name as argument */
   for (i = 0; i < SIZE; i++)
      printf("arr[%hd] = %hd ", i, arr[i]);
   printf("\n");
   return 0;
}
void input2array(short f_arr[], short num)
{
   short i;
   printf("Key in %hd integers: ", num);
   for (i = 0; i < num; i++)
      scanf("%hd", &f_arr[i]);                 /* No array named f_arr actually */
   return;
}
```

PROGRAM 11.5: **input2array.c**

```
Key in 6 integers: 1 5 20 100 1024 4096
arr[0] = 1 arr[1] = 5 arr[2] = 20 arr[3] = 100 arr[4] = 1024 arr[5] = 4096
```

PROGRAM OUTPUT: **input2array.c**

your breath until Chapter 12 when we revisit arrays. We'll discover that array notation in a function declaration is merely a convenience, and that a function need not use this notation to access the array in the function body.

Arrays can take up a lot of space, so it just doesn't make sense for a function to create a copy and then lose it immediately after the function terminates. Besides, functions that sort and search an array can do the job faster if they are spared the burden of creating a copy. True, there remains the risk of inadvertently changing the array from the function, but the benefits of this design far outweigh the drawback.

11.8.2 The `static` keyword: Keeping an Array Alive

A function can create an array as a local variable and return any element to the caller. This is reflected in the following definition of a function that retrieves from its own database (a local array) the maximum number of days available in a month:

```
short days_in_month(short index)
{
   short month[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
   return month[index];
}
```

But what if a function returns the name of the array, i.e., the address of the first element? Can the array then be accessed outside the function? No, that won't be possible because even if the calling function can "see" this address, the array itself ceases to exist after the function has completed execution. We'll provide a solution to this problem after addressing a performance issue related to the use of arrays.

Every call to **`days_in_month`** creates the `month` array, initializes it and finally destroys it. Function calls have overheads, which can be quite severe when used to create large arrays. Better performance could be obtained by creating the array in **`main`**, but then the function would lose its generality. We would then need to create this array in every program using this function. A better solution would be to declare the array as **`static`** inside the function:

```
static short month[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Unlike a normal variable, a **`static`** variable is created and initialized once—when the function is first called. Thus, the `month` array will retain all values between successive calls to **`days_in_month`**. And if this function is modified to return the starting address of `month` instead, the caller can access all elements without repeatedly calling **`days_in_month`**! We'll make this possible using pointers in Chapter 12. The **`static`** keyword, which affects the lifetime of any variable (not limited to arrays), is examined in Section 11.17.2.

11.9 `merge_arrays.c`: MERGING TWO SORTED ARRAYS

Program 11.6 uses the services of three functions to merge two sorted arrays. One function takes care of initializing the two arrays with sorted data. Another one performs the actual task of merging the data. The third function simply prints the sorted data.

```
/* merge_arrays.c: Merges two sorted arrays. Uses three functions to
                   (i) initialize, (ii) merge and (iii) print arrays. */
#define SIZE1 5
#define SIZE2 6

#include <stdio.h>

void input2array(short a[], short num);
void merge_arrays(short a[], short i, short b[], short j, short c[]);
void print_array(short a[], short num);

int main(void)
{
   short arr[SIZE1], brr[SIZE2], crr[SIZE1 + SIZE2];

   input2array(arr, SIZE1);
   input2array(brr, SIZE2);

   merge_arrays(arr, SIZE1, brr, SIZE2, crr);

   printf("After merging: ");
   print_array(crr, SIZE1 + SIZE2);
   return 0;
}

void input2array(short a[], short num)
{
   short i;
   printf("Key in %hd integers: ", num);
   for (i = 0; i < num; i++)
      scanf("%hd", &a[i]);
   return;
}

void merge_arrays(short a[], short m, short b[], short n, short c[])
{
   short i = 0, j = 0, k = 0;
   while (i < m && j < n) {
      if (b[j] > a[i])
         c[k] = a[i++];           /* Assigns c[] higher of two element values */
      else
         c[k] = b[j++];           /* Ditto */
      k++;                        /* Move to next element of c[] */
   }

   /* Once here, it means one array has been totally assigned to c[] */
   if (i >= m)                    /* If it is a[] ... */
      while (j < n)
         c[k++] = b[j++];         /* ... assign remaining elements of b[] */
   else if (j >= n)               /* If it is b[] ... */
      while (i < m)
         c[k++] = a[i++];         /* ... assign remaining elements of a[] */
   return;
}
```

```
void print_array(short a[], short num)
{
   short i;
   for (i = 0; i < num; i++)
      printf("%d ", a[i]);
   printf("\n");
   return;
}
```

PROGRAM 11.6: **merge_arrays.c**

```
Key in 5 integers: 11 33 55 77 99
Key in 6 integers: 22 44 66 88 95 111
After merging: 11 22 33 44 55 66 77 88 95 99 111
```

PROGRAM OUTPUT: **merge_arrays.c**

The compiler permits the use of `SIZE1 + SIZE2` to set the size of the third array because the value of this expression is known at compile time. The action begins by calling the **input2array** function twice to populate two arrays. Remember to key in an ascending list of integers.

The **merge_arrays** function uses five arguments to merge the two arrays. The size is specified for `a` and `b`, but not for `c` because its corresponding argument, `crr`, is adequately sized. The first **while** loop compares elements of both arrays before assigning them to `c` in the right order. The loop terminates when `c` has all elements of either `a` or `b`. The **if** statement then runs a **while** loop to assign `c` with the leftover elements of the other array. The **print_array** function finally prints the merged array.

This program represents a classic case of *modularization (3.11)*. Here all work has been assigned to individual modules, and the lean **main** section simply invokes them. This is what you must be prepared to do as your programs get progressively large. Note that the program is easily maintainable; you can easily spot the module that requires change.

11.10 PASSING A TWO-DIMENSIONAL ARRAY AS ARGUMENT

A function can also work with the name of a 2D array as an argument. Like with a 1D array, a 2D array is specified differently in the declaration (and definition) and invocation. In the former case, you may drop the first subscript (rows), but you can't drop the second subscript (columns). The following declarations for the **print_2d_array** function are thus valid:

```
void print_2d_array(int arr[3][5], int rows);
void print_2d_array(int arr[][5], int rows);
```

Without knowledge of the number of columns, it would be impossible for the compiler to know when one row ends and another begins, considering that the rows are placed next to one another in memory. In other words, the compiler must know that element `arr[0][4]` is followed by `arr[1][0]` in memory.

However, you can use only the array name when invoking the function:

```
print_2d_array(arr, 3);                    3 should be replaced with a variable
```

Program 11.7 prints a 2D array using a function. The number of columns is set as a symbolic constant, but the number of rows is obtained by evaluating an expression (total space used / space used by each row). This number (60 / 20 = 3) is then passed as the second argument to the function, **print_2d_array**. This is the correct way of passing the number of rows to a function when this number is not explicitly specified in the program.

Observe that the subscript for the row is missing in the function declaration but not in the definition. This was done simply to prove that the compiler doesn't look at the row subscript at these two locations. Because this subscript is passed as a separate argument, the program can print multiple array types provided they have the same number of columns which is fixed at compile time.

```
/* print_2d_array2.c: Uses a function to print a 2D array. */
#define COLUMNS 5
#include <stdio.h>
void print_2d_array(int arr[][COLUMNS], int rows);
int main(void)
{
   int arr[][COLUMNS] =
       { {1, 3, 5, 7, 9}, {11, 33, 55, 77, 99}, {111, 333, 555, 777, 999}};
   short m_rows = sizeof(arr) / sizeof(arr[0]);
   printf("Number of rows = %hd\n", m_rows);
   print_2d_array(arr, m_rows);
   return 0;
}
void print_2d_array(int f_arr[3][COLUMNS], int f_rows)
{
   int i, j;
   for (i = 0; i < f_rows ; i++ ) {
      for (j = 0; j < COLUMNS; j++)
         printf("%3d ", f_arr[i][j]);
      printf("\n");
   }
   return;
}
```

PROGRAM 11.7: **print_2d_array2.c**

```
Number of rows = 3
  1   3   5   7   9
 11  33  55  77  99
111 333 555 777 999
```

PROGRAM OUTPUT: **print_2d_array2.c**

11.11 CALLING A FUNCTION FROM ANOTHER FUNCTION

Just as **main** calls a function, a function can also call another function. The called function can in turn call another function, and so on. In this scenario, the returns of all intermediate functions are kept pending until the innermost function returns. The remaining functions then return in the reverse sequence of their invocation. Ultimately, the outermost function returns control to **main** (Fig. 11.2).

As long as the return of a function is kept pending, its parameters and local variables continue to occupy space in memory. Freeing of memory begins with the return of *fn* and continues progressively as the other functions return. However, the return value, if any, of every function is captured by its caller before the called function is extinguished.

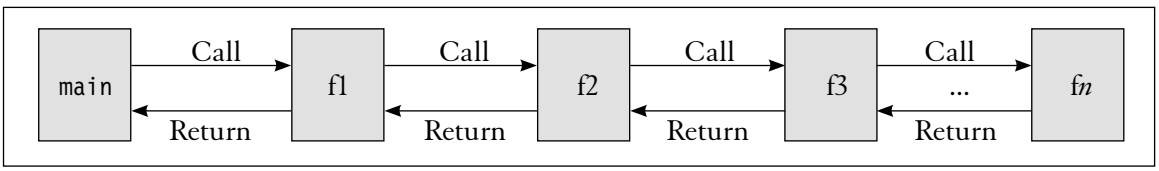


FIGURE 11.2 Functions Calling Other Functions

11.11.1 time_diff.c: Program to Compute Difference Between Two Times

Let's now take up Program 11.8, which computes the difference between two times input by the user, each in *hh mm* format. The **main** function calls the **time_diff** function, which computes the elapsed time in minutes. This function in turn calls the **absolute** function to ensure that a negative value is converted to positive before it is returned to **main**. The function call sequence is shown below:

main ⟶ time_diff ⟶ absolute

Even though the **absolute** function is called from **time_diff**, it must be declared before **main**. This is because a function can't be declared in the body of another function. **scanf** accepts the two times as four variables, which are then passed as arguments to **time_diff**. After **time_diff** returns, its return value is printed. Note that **main** can't return until **time_diff** returns, which can do so only after **absolute** returns.

The **time_diff** function converts the two times to minutes elapsed since 00:00 hours (on the same day) before they are subtracted. (The UNIX clock is maintained as the number of seconds elapsed since January 1, 1970, 00:00 hours.) Because the times are not validated, a negative value is possible, so a call to the **absolute** function is made. This function simply returns the absolute value of the number using a conditional expression.

Note: Most implementations of C allocate a fixed chunk of memory (the *stack*) for function variables and parameters. This space, which is usually adequate for multiple invocations of functions, is progressively eaten up as one function calls another, which in turn calls another, and so on. The space may be totally consumed when a function calls itself by *recursion* and fails to terminate. The functioning of the stack is taken up later in the chapter.

```
/* time_diff.c: Calculates the difference between start and end times.
                Calls one function from another function. */
#include <stdio.h>

int time_diff(short h1, short m1, short h2, short m2);
int absolute(short a_mins);

int main(void)
{
   short mins, hours1, hours2, mins1, mins2;

   printf("Enter start & endtimes as HH MM HH MM: ");
   scanf("%hd%hd%hd%hd", &hours1, &mins1, &hours2, &mins2);

   mins = time_diff(hours1, mins1, hours2, mins2);      /* Calls function1 */

   printf("Difference between the two times = %hd minutes\n", mins);

   return 0;
}

int time_diff(short h1, short m1, short h2, short m2)   /* Function1 */
{
   short net_mins;
   net_mins = h1 * 60 + m1 - (h2 * 60 + m2);
   return absolute(net_mins);                           /* Calls another function */
}

int absolute(short a_mins)
{
   return (a_mins > 0 ? a_mins : a_mins * -1);
}
```

PROGRAM 11.8: **time_diff.c**

```
Enter start & endtimes as HH MM HH MM: 12 35 7 30          12:35 and 7:30
Difference between the two times = 305 minutes
Enter start & endtimes as HH MM HH MM: 14 45 6 15          Negative value
Difference between the two times = 510 minutes
Enter start & endtimes as HH MM HH MM: 2 30 0 0
Difference between the two times = 150 minutes
```

PROGRAM OUTPUT: **time_diff.c**

Takeaway: Because each function is allocated separate space in the stack, no conflicts can occur between variable names used in the calling and called functions.

11.11.2 power_func.c: Computing the Sum of a Power Series

Consider the task of computing the value of an expression that represents the sum of a series of n terms where the nth term is expressed as n^n. The expression is formally represented in the following manner:

$1^1 + 2^2 + 3^3 + 4^4 + \ldots + n^n$

HOW IT WORKS: When `exit` *Is Better than* `return`

Except when used in **main**, **return** doesn't terminate a program. When functions are nested and you need to quit the *program* from an inner function, it is impractical to use **return**. In such situations, use the **exit** function which bypasses the function hierarchy including **main**, and switches control to the operating system. The situation is analogous to the use of **goto** rather than **break** when coming out of a deeply nested loop.

The **exit** function is invoked with a return value as argument and needs the file `stdlib.h`, so you must include this file when using **exit**. The call **exit(0);** made from any function simply terminates the program (after performing the necessary cleanup operations). UNIX-C programmers (including this author) prefer to use **exit** rather than **return** for terminating a program.

For this purpose, we need to reuse the code for computing the power of a number that we have developed previously (Program 8.7). The following program, (Program 11.9), uses the same code—this time as the **power** function—to evaluate each term of the series. Another function, **compute_sum**, calls this function repeatedly to compute the final sum.

```
/* power_func.c: Computes the sum of n terms where the nth term is n^n.
                 Calls the power function from another function. */
#include <stdio.h>

long power(short f_base, short f_exponent);
unsigned long compute_sum(short f_terms);

int main(void)
{
   short m_terms;

   printf("Enter number of terms to sum: ");
   scanf("%hd", &m_terms);
   printf("\nSum of %hd terms = %ld\n", m_terms, compute_sum(m_terms));

   return 0;
}

unsigned long compute_sum(short f_terms)
{
   short i = 1, base = 1;
   unsigned long sum = 0;
   while (i++ <= f_terms) {
      sum += power(base, base);    /* Each term evaluated and summed here */
      printf("%ld  ", sum);        /* Prints progressive sum */
      base++;
   }
   return sum;
}
```

```
long power(short f_base, short f_exponent)
{
   short i = 0;
   long result = 1;
   while (++i <= f_exponent)
      result *= f_base;                        /* Multiply base by itself */
   return result;
}
```

PROGRAM 11.9: **power_func.c**

```
Enter number of terms: 3
1  5  32
Sum of 3 terms = 32
Enter number of terms: 5
1  5  32  288  3413
Sum of 5 terms = 3413
```

PROGRAM OUTPUT: **power_func.c**

11.12 sort_bubble.c: ORDERING AN ARRAY USING BUBBLE SORT

After having used the selection sort algorithm (Program 10.7), let's learn to use the bubble sort technique of ordering an array. For an ascending sort and beginning from the left, two adjacent elements are compared, and if the left element is greater than the right one, the two elements are swapped. With each pass, the lower values from the right "bubble" their way to the left. The entire array is sorted using *n* - 1 passes where *n* represents the size of the array. Program 11.10 features a number of interesting techniques that you should incorporate in other programs.

The program uses the **bubble_sort** function which, in turn, invokes the **compare** function to compare two adjacent elements. The fourth argument of **bubble_sort** is unusual: it determines whether the multi-row output representing the progress of sort should be printed. Use of properly named symbolic constants (like ASCENDING and PRINT_YES) rather than actual numbers makes it easy to understand what the constants actually represent.

The **compare** function performs a simple task; it returns a true or false value from the comparison of two array elements. Also, depending on whether **bubble_sort** uses ASCENDING or DESCENDING as its third argument, **compare** uses either `x < y` or `y > x` as the expression for comparison. By tweaking this function, you can handle complex sorting conditions—say, sorting only odd numbers.

Because we opted for printing the progress of sort (PRINT_YES instead of PRINT_NO), we have a clear picture of the working of the bubble sort algorithm. Observe how the lowest value (0) needs 11 passes (one less than the size of the array) to move from the extreme right to the extreme left. In many cases, you'll find that a lesser number of passes will suffice.

```
/* sort_bubble.c: Uses bubble sort in a function which invokes
                  another function to determine sort order. */
#include <stdio.h>
#define ASCENDING 0
#define DESCENDING 1
#define PRINT_YES 1
#define PRINT_NO 0

void bubble_sort(short sarr[], short size, short order, short flag);
void print_array(short sarr[], short size);
int compare(short x, short y, short order);

int main(void)
{
   short arr[] = {111, 77, 30, 5, 7, 60, 222, 55, 80, 15, 40, 0};
   short size = sizeof(arr) / sizeof(arr[0]);

   printf("Array contents before sorting:\n");
   print_array(arr, size);
   printf("Array status after each iteration of outer loop:\n");

   bubble_sort(arr, size, ASCENDING, PRINT_YES);

   printf("Array contents after sorting:\n");
   print_array(arr, size);

   return 0;
}

void bubble_sort(short sarr[], short size, short order, short flag)
{
   short i, j, temp;
   for (i = 0; i < size - 1; i++)  {
      for (j = 0; j < size - i - 1; j++)
         if (compare(sarr[j], sarr[j + 1], order)) {
            temp = sarr[j];
            sarr[j] = sarr[j + 1];
            sarr[j + 1] = temp;
         }
      if (flag == PRINT_YES)              /* Set flag = PRINT_NO if no ... */
         print_array(sarr, size);         /* ... display of progress needed */
   }
   return;
}

void print_array(short sarr[], short size)
{
   short i;
   for (i = 0; i < size; i++)
      printf(" %3hd", sarr[i]);
   printf("\n");
   return;
}
```

```
int compare(short x, short y, short order)
{
   return (order == ASCENDING) ? (x > y) : (x < y);
}
```

PROGRAM 11.10: **sort_bubble.c**

```
Array contents before sorting:
111  77  30   5   7  60 222  55  80  15  40   0
Array status after each iteration of outer loop:
  77  30   5   7  60 111  55  80  15  40   0 222
  30   5   7  60  77  55  80  15  40   0 111 222
   5   7  30  60  55  77  15  40   0  80 111 222
   5   7  30  55  60  15  40   0  77  80 111 222
   5   7  30  55  15  40   0  60  77  80 111 222
   5   7  30  15  40   0  55  60  77  80 111 222
   5   7  15  30   0  40  55  60  77  80 111 222
   5   7  15   0  30  40  55  60  77  80 111 222
   5   7   0  15  30  40  55  60  77  80 111 222
   5   0   7  15  30  40  55  60  77  80 111 222
   0   5   7  15  30  40  55  60  77  80 111 222
Array contents after sorting:
   0   5   7  15  30  40  55  60  77  80 111 222
```

PROGRAM OUTPUT: **sort_bubble.c**

11.13 RECURSIVE FUNCTIONS

Loops use the iterative technique to repeatedly update a set of data with the same set of instructions. Repetition can also be achieved by *recursion*, a form of cloning that uses instructions to specify themselves. In C, a recursive function calls itself, which on return delivers to its caller its share of the final solution. This incremental approach is akin to that followed by a loop, so the same problem can often be solved using both techniques.

A recursive function must not be allowed to run forever because the program will eventually run out of memory. That will happen if the function doesn't specify a condition that eventually terminates the process, as reflected in the following code where **main** calls itself:

```
#include <stdio.h>
int main(void)
{
   static long count;
   printf("Call number: %ld\n", count++);
   main();
   return 0;
}
```

```
OUTPUT:
Call number: 1
Call number: 2
  ...
Call number: 523171
Call number: 523172
Segmentation fault
```

Because the **static** variable count is defined and initialized just once, it can track the number of calls made to **main**. You are aware that function variables and parameters are saved in a separate region of memory called the stack. With every recursive call, the stack grows in size, and if this growth goes unchecked, the stack will eventually overflow. (See Inset entitled "How Functions Use the Stack" in Section 11.13.2.) In this program, after **main** has called itself 523,172 times, stack overflow occurs and terminates the program with a "segmentation fault." This is not the way recursive functions are meant to be used.

A recursive function is typically used with one or more arguments that represent a subset of the problem. Each call contains a part of the solution which cannot be delivered to its caller until return of the call it has made to itself. The function typically includes a check for a *terminating condition* which at some point changes to stop further recursion. The calls then return in reverse sequence of their invocation, moving closer to the final solution with every return.

11.13.1 factorial_rec.c: Using a Recursive Function to Compute Factorial

Let's consider the familiar recursive function that computes the factorial of a number. This is simply the product of all integers from 1 to the number. For instance, 6!, the factorial of 6, is $6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$. But 6! can also be expressed as $6 \times 5!$ in the same way as 5! is expressed as $5 \times 4!$. We can thus express the factorial of a number in this manner:

$$\text{factorial}(n) = n \times \text{factorial}(n - 1)$$

Recursion is at work here because the factorial of a number can be described in terms of the factorial of another number. Our task now is to define a function that repeatedly calls itself until $n = 1$. This represents the terminating condition that stops further calls from being made. Program 11.11 uses this recursive technique.

Let's now examine what the **factorial** function actually delivered in its six invocations. The final solution (the value of 6!), which is stored in the variable result, is determined by the return values of all recursive calls. This value can thus be made available only after these calls have returned. The first call made to **factorial(6)** returns the following value:

```
6 * factorial(5)
```

This call is kept pending because its return value depends on the return value of **factorial(5)**. A call to **factorial(5)** is made next, but this call can't return either because it has to make a call to **factorial(4)**. In this way, all recursive calls are kept pending until a call is made to **factorial(1)**. The terminating condition (n == 1) evaluates to true here, so **factorial(1)** returns 1 without making any further calls. This enables **factorial(2)** to return, followed by **factorial(3)**, and so on until **factorial(6)** returns with the final answer.

```
/* factorial_rec.c: Computes factorial of a number by recursion. */
#include <stdio.h>
long factorial(short n);
short count;                        /* A global variable; seen everywhere ... */
int main(void)
{
   short num;
   printf("Factorial of which number? ");
   scanf("%hd", &num);
   printf("!%d = %ld\n", num, factorial(num));
   printf("Count = %hd\n", count);              /* ... both here ... */
   return 0;
}

long factorial(short n)
{
   long result;
   count++;                                     /* ... and here. */
   if (n == 1)                                  /* Terminating condition */
      result = 1;
   else
      result = factorial(n - 1) * n;            /* Recursive call */
   return result;
}
```

PROGRAM 11.11: **factorial_rec.c**

```
Factorial of which number? 6
!6 = 720
Count = 6
```

PROGRAM OUTPUT: **factorial_rec.c**

For the first time in this text, we have defined a variable (count) *before* **main**. This makes count a *global* variable and not a local one. If count had been declared in either **main** (the normal location so far) or **factorial**, it would not have been visible in the other function. As discussed in Section 11.16.1, a global variable can be accessed by all functions in the program.

Note: The variable count is used by the **factorial** function and yet is not declared there. In the process, the function has lost its reusability because every program that uses it must have count defined as a global variable before **main**.

11.13.2 Recursion vs Iteration

Now let's consider the iterative version of this problem. This version simply multiplies all positive integers not greater than the number. The **factorial** function here employs a **for** loop which computes !6 with six iterations:

```
long factorial(short n)
{
   short i;
   long result = 1;
   for (i = n; i > 0; i--)
      result *= i;
   return result;
}
```

HOW IT WORKS: How Functions Use the Stack

Most C implementations use a *stack*, a data structure occupying a fixed region of memory, to execute functions (Fig. 11.3). When a function is called, its parameters and local variables are *pushed* onto the stack. If this function calls another function, then another set of parameters and variables are stacked on top of the previous set. At any point of time, it's the top-most set of variables and parameters that are active.

When a function completes execution, its parameters and variables are *popped* out of the stack and the portion of memory allocated to the function is deallocated. The process is repeated for the next function, whose parameters and variables now feature at the top of the stack. The stack thus operates in a LIFO (last in-first out) mode. The first entrants to the stack are the local variables of the **`main`** function. They are also the last to leave when the program, i.e. **`main`**, completes execution.

The limited size of the stack becomes a serious limitation when using recursive functions. If the stack can't accommodate any more function calls, then *stack overflow* occurs, which causes the program to crash. It's easy to create this overflow by calling **`main`** repeatedly *(11.13)*.

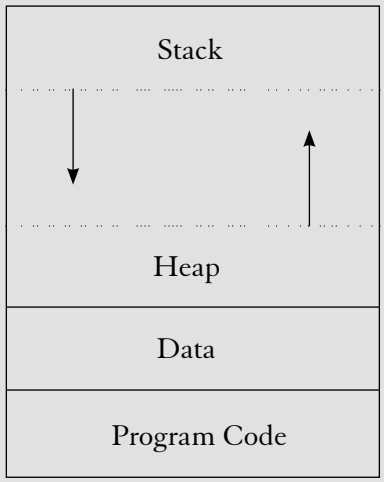


FIGURE 11.3 Organization of the Stack in Memory

For computing 6!, the recursive version makes six function calls but the iterative version makes a single call. Each function call is allocated its own set of local variables and parameters in the stack. So, for computing !6 using the recursive technique, six separate sets of the variables, n and result, are *pushed* onto the stack. In contrast, the iterative solution pushes a single set of the variables, i, n and result onto the stack.

As noted before, the overheads associated with a function call could be significant when the number of calls is very high. For this reason, it is often felt that the iterative solution is a better choice. This is, however, not the case here because you can't make too many recursive calls with this function anyway. If **sizeof(long)** is 8 bytes on your machine, 25! will easily exceed the maximum value possible for this data type.

11.14 THINKING IN RECURSIVE TERMS

Like a loop, a recursive function repeats a set of instructions, but it takes a little time initially to understand problems in terms of recursion rather than iteration. The idea is actually simple if you consider that the changes caused by each loop iteration can also be achieved by passing and returning different values in each function call. The following guidelines should be helpful:

- Divide the problem into appropriate subsets that can be subjected to the same treatment. For instance, 6! can be computed by applying the same instruction to each of its subsets, i.e., 5!, 4!, etc.
- Connect the subsets in a way that enables the solution to develop in a progressive manner. For n!, the statement `return n * factorial(n - 1);` connects the subsets, and every invocation of the statement takes us closer to the solution.
- Identify the *base case* (the final call) where the terminating condition evaluates to true. At this point, the return of all pending calls is initiated. For n!, the base case occurs when n = 1.

These techniques are unique to recursive calls, but otherwise a recursive function call is fundamentally no different from a function calling another function. The downside of making too many function calls apply to recursive calls in the usual manner. Let's now discover recursion in some common problems that are normally solved by iteration.

11.14.1 Adding Array Elements Recursively

To add the elements of an array, a function (iterative or recursive) needs three inputs—the name of the array and its two indexes that define the range of the elements to be summed. This is easily handled by recursion because the sum of *n* elements is simply the first element + the sum of the remaining (*n* - 1) elements. Shown below is the definition of the **add_array** function where the parameters, first and last, signify the two indexes:

```
short add_array(short arr[], short first, short last)
{
   if (first == last)
      return arr[last];                  /* Base case -- returns only element */
   else
      return arr[first] + add_array(arr, first + 1, last);
}
```

The second **return** statement provides the connecting link between successive calls. Each call to **add_array** increments first by one, so the next array element becomes the first element for that call. In the base case, the termination condition (first == last) evaluates to true and returns the last element. This initiates the return of the other pending calls, where each call adds its "first element" to this value. Eventually, when the first call returns, all elements defined by first and last have been added. This is how the function can be invoked from **main**:

```
#include <stdio.h>
short add_array(short arr[], short first, short last);
int main(void)
{
    short arr[10] = {1, 3, 5, 7, 9, 12, 14, 16, 18, 20};
    short first = 0, last = 9;
    printf("Sum of elements %hd to %hd = %hd\n",
            first, last, add_array(arr, first, last));
    return 0;
}
```

The program outputs 105, the sum of all ten elements. For an array of a few hundred elements, adopting an iterative solution won't result in significant savings. But before adding a million elements, you have to choose between using a million recursive calls or a single iterative one. (The choice is actually obvious.)

11.14.2 Computing the Power of a Number Recursively

We developed an iterative version of the **power** function in Program 11.9. This function can also be implemented using recursion because the *n*th power of a number can be described in terms of the (*n* - 1)th power of the same number. In other words, power(2, 10) = 2 * power(2, 9). This relationship is implemented with the following code:

```
long power(short base, short exponent)
{
    long result;
    if (exponent == 1)
       result = base;
    else
       result = power(base, exponent - 1) * base;
    return result;
}
```

Because each call to **power** reduces the exponent by one, repeated calls will eventually lead to the base case, **power(2, 1)**, where the terminating condition stops further recursion. Here, the second assignment to result provides the connecting link between successive calls.

11.14.3 Using Recursion to Compute Fibonacci Numbers

Section 8.3.3 adopted an iterative approach for deriving the terms of the Fibonacci sequence. To recall briefly, a Fibonacci sequence starts with the numbers 0 and 1, but every subsequent term is formed by summing the previous two. Here are the first 10 terms of the sequence:

```
0  1  1  2  3  5  8  13  21  34  55
```

This sequence can also be derived by recursion because of the following relationship that exists between three consecutive terms of the sequence:

fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)

The following function, **fibonacci_rec**, returns the sum of the previous two terms by calling itself twice in the **return** statement:

```
unsigned long fibonacci_rec(short num)
{
   if (num < 3)                                    /* Excludes first two terms */
      return 1;
   else {
      return fibonacci_rec(num - 1) + fibonacci_rec(num - 2);
   }
}
```

The function is called from a loop in **main** which prints each term in every iteration of the loop:

```
for (i = 1; i <= num; i++)
   printf("%ld ", fibonacci_rec(i));
```

Recursion is a loser here because 500 million recursive function calls are needed to obtain the first 40 terms. (You can verify this figure by setting up a counter inside the function after having defined it globally.) The iterative technique needs just one function call and 40 iterations!

Caution: Recursion can lead to two problems that are not encountered in iteration. Because of the way the function stack is implemented, too many calls can lead to either stack overflow or increased execution time. In most cases, the iterative approach is the way to go.

11.15 THE main FUNCTION

We now meet a special function that is present in every C executable—the **main** function. This function is the ultimate ancestor of all functions called in a program. Program control is divided between **main** and the other functions called by it. A program terminates when **main** terminates, and this happens either by invoking **return**, **exit** or by "falling through" to the end.

Unlike user-defined functions, **main** is never declared in a program. The definition of **main** is, however, provided by the programmer. When you invoke a program (say, **a.out** or **aout.exe**), the operating system uses the program loader to run **main** *without arguments*. This is signified by the now-familiar header:

```
int main(void)                                  Program run without arguments
```

The int signifies the data type of the return value. The word void indicates that **main** is called without arguments. The caller of **main** is normally the operating system, so the value returned by **main** is available in the operating system as one of its variables.

Variables declared inside **main** are allocated space from the same memory bank that other function variables and parameters draw upon. Thus, these variables are also local in nature and are not accessible to other functions. But they have a longer lifespan because **main** terminates after all other functions have terminated. We'll revisit **main** in Chapter 13 to understand how it is invoked with arguments and how its header changes accordingly.

11.16 VARIABLE SCOPE AND LIFETIME

When working with functions, you'll face situations where two or more functions need to access the same variable. You have already faced that situation with Program 11.11, where the variable `count` was used by both **main** and the **factorial** function. Sometimes, you'll need to hide a variable even within the same function. Further, the same variable may also be used by multiple program files. C offers a number of features that adequately address these issues.

Apart from having a data type, a variable has space and time attributes. The space attribute signifies the *scope* or *visibility* of the variable. This is actually the region of the program where the variable is visible and can be accessed. A variable may have one of four possible scopes: *block*, *function*, *file* and *program*. A variable having block scope is visible in the block in which it is declared. A variable having file scope is visible in the entire file.

The time attribute determines the *lifetime* of the variable, i.e., its time of birth and death. A variable declared inside a function lives as long as the function is active. One declared before **main** is alive for the duration of the program.

11.16.1 Local and Global Variables

Variables declared inside a function are visible only in the function. They have local *scope*, the reason why they are called *local variables*. Variables declared outside a function are known as *global variables*. The scope of a global variable extends from the point of its declaration to the rest of the file. When a variable is declared before **main**, it is truly global across the entire program (which may be spread across multiple files). We declared a global variable (`count`) in Program 11.11, and could access it both in **main** and a function called from **main**.

We'll now examine the lifetime and initialization of variables. A global variable has a lifetime over the complete program. By default, it is initialized to zero at the start of program execution. On the other hand, a local variable lives as long as its function lives. It has no default value, so it would be wrong to assume zero or NUL as its default value.

11.16.2 Variables in a Block

Visibility, however, gets affected when a variable is declared inside a block. Though we have not exploited this concept yet, be aware that you can create a block, enclosed by curly braces, anywhere in a program. In the code segment shown in Figure 11.4, `x` is a global variable, `y` is a local variable of **main** and `z` is a local variable of the inner block.

While `x` can be accessed in the entire program, `y` is visible only in **main**, including the inner block. The visibility of `z` is limited to the inner block in which it is declared. Had this block contained nested blocks, then all of the variables would have been accessible in the nested blocks too.

```
int x = 2;                          /* Global variable */
int main(void)
{
   int y = 22;                      /* Local variable of main */
   {
      int z = 222;                  /* Local variable of block */
      /* x, y and z visible here */
   }                                /* End of block */
   /* x and y visible here, but not z */
}

int func(void)
{
   int p = 5;                       /* Local variable of func */
   /* x and p visible here, but not y and z */
}
```

FIGURE 11.4 Scope of Global and Local Variables

11.16.3 Variable Hiding

If an inner block re-declares a variable, then its new value *hides* both the global value and the old value assigned in the outer block (Fig. 11.5). A variable can be hidden by creating another one having the same name in memory. Hiding doesn't overwrite the previous value, which becomes visible when the inner block is exited.

```
int x = 2;                      /* Global variable */
int main(void)
{
   int y = 22;                  /* Local variable of main */
   {
      /* x and y visible here but only until they are re-declared */
      int x = 4;                /* Hides the value 2 */
      int y = 44;               /* Hides the value 22 */
      int z = 88;               /* Nothing to hide */
   }
   /* Original values of x and y visible here, but z is undefined */
}
```

FIGURE 11.5 Re-Defining Variables

The inner block this time re-declares (or re-defines) two variables, x and y. When the inner block is exited, none of the values that were seen inside the block are seen outside it. The old values of x and y become visible once more.

Takeaway: A local or global variable ceases to be visible in an inner block if that block also declares a variable with the same name. However, the original variable becomes visible when the block is exited.

11.17 THE STORAGE CLASSES

The scope, lifetime and initial value of a variable are determined by its *storage class*. So far, we have used the default storage class for variables (automatic), but you also need to know the following storage classes supported by C:

- automatic (**auto**)
- static (**static**)
- external (**extern**)
- register (**register**)

The storage class also determines the way the compiler allocates memory for a variable. The keywords shown in parentheses are mutually exclusive, and only one of them can be used as a qualifier in a variable declaration. Table 11.1 makes a comparison of the key features of the four storage classes.

11.17.1 Automatic Variables (auto)

If a function is invoked repeatedly, its local variables are created and destroyed automatically as many times as the function is called. For this reason, local variables of a function are also known as *automatic variables*. The keyword **auto** used as a qualifier in a declaration makes a variable an automatic one. We have never used it though because, by default, a local variable has auto as its storage class. But we can also explicitly specify a variable as automatic:

```
int main(void)
{
   auto short count = 0;                    auto can be omitted
   count++;                                 Always has the value 1
    ...
}
```

An automatic variable has function scope and a junk default value. As seen above, you can't use an automatic variable to count the number of times a function is called because every call initializes the variable to the same value. A global variable partially solves the problem since it is initialized once, but by depending on a variable that is defined outside **main**, a function loses its status as a self-contained unit. C has a simple solution to this problem which is discussed next.

Note: Automatic variables are pushed onto the stack when the function is called and popped out after execution is complete *(11.13.2—Inset—How It Works)*. For a variable to retain its value for the duration of the program, it must not be stored in the stack, i.e., it can't have the **auto** storage class.

11.17.2 Static Variables (static)

If a local variable is assigned the *static* storage class, then it will retain its value between function calls. A static variable (keyword: **static**) is created and initialized just once, when the function containing its declaration is executed. It remains alive (is static) even after the function terminates.

This is how we declared a static variable in Section 11.13 to count the number of times the **main** function was called within **main**:

```
static long count;                          Default value is zero
```

By default, a static variable is initialized to zero, but even if it is initialized explicitly, this is done just once. The following definition initializes count to 5 and increments it with every call made to the function:

```
void f(void)
{
   static long count = 5;
   count++;                                 count incremented with every call
}
```

For some reason, if you want a static variable to be initialized every time the function is called, then this is the way to do it:

```
static long count;                          First initialized to zero ...
count = 5;                                  ... and then assigned the value 5
```

The **static** qualifier can also be used with a global variable (defined before **main**), in which case the variable is accessible only in the current file. The static attribute doesn't affect the initialization property because a non-static global variable is initialized once anyway.

A function may also be declared as **static**, and that would make the function inaccessible in other files.

Takeaway: A local variable of a function is created and extinguished every time the function is called and terminated. This default **auto** feature can be overridden with the **static** qualifier in which case the variable retains its value across multiple invocations of the function.

11.17.3 External Variables (extern)

A global or external variable is defined outside a function and is visible everywhere—even in other files that constitute the program. However, a variable declared after **main** is not visible in **main**:

```
int main(void)
{
   ...
   return 0;
}                                           End of main
int x = 10;                                 Not visible in main
```

To make matters worse, a variable declared at the end of the file is not visible anywhere—even in functions that are defined prior to the variable declaration. To make the variable visible at selected locations in the file, you need to assign the *extern* storage class to the variable. In the current scenario, for x to be visible in **main**, it must be declared there with the **extern** qualifier:

TABLE 11.1 Storage Classes of Variables

Storage Class	*Visibility*	*Lifetime*	*Default Value*	*Remarks*
`auto`	In function/block where declared	Duration of function/block	Junk	Default class for variable declared in function/block
`static` (in function)	In function/block where declared	Duration of program/block entry	Zero	Retains value between multiple function calls
`static` (outside function)	From point of declaration to end of file	Duration of program	Zero	Retains value across multiple files
`extern`	In function/block, file or multiple files	Duration of program	Zero	Must be declared at one place without `extern`
`register`	In function/block where declared	Duration of function/block	Junk	Stored in CPU register for faster access

```
int main(void)
{
   extern int x;                                   x declared here ...
   ...
   return 0;
}
int x = 10;                                        ... and defined here
```

The **`extern`** keyword indicates that the variable `x` is merely *declared* here and that the *definition* has been made elsewhere (here, after **`main`**). This means storage for `x` is allocated only when it is defined, and the **`extern`** statement simply intimates type information to the compiler. *There is only one variable* `x` *here using a single storage location*. Without **`extern`**, `x` would be a local variable of **`main`** and would have no relation with the one that is defined later.

Takeaway: If a variable `x` is not visible in a function or a block, use the **`extern`** qualifier to make it visible there. Also, while a regular variable is declared and defined simultaneously, an extern variable can only be declared. It must be defined "elsewhere."

11.17.4 `extern.c`: Using `extern` to Control Variable Visibility

Program 11.12 explores the visibility of two variables (`x` and `y`) and an array (`arr`) which are defined after **`main`**. `x` and `arr` are visible in **`main`** because of **`extern`**, but `y` is not. The `y` in **`main`** is an uninitialized automatic variable that doesn't share storage with the `y` defined after **`main`**. Note that `x` declared in the block is by default not visible after the block is exited, the reason why it had to be re-declared with **`extern`**.

```
/* extern.c: Uses extern to access in main variables defined below main.
             Makes rare distinction between declaration and definition. */
#include <stdio.h>
int main(void)
{
   {
      extern int x;                        /* Declares x without defining it */
      extern int arr[];                    /* Declares arr without defining it */
      int y;                               /* Separate local variable */
      arr[0] = 10; arr[1] = 20;
      printf("Initial value of x = %d\n", x);
      printf("Initial value of y = %d\n", y);
      printf("Elements of arr = %d, %d\n", arr[0], arr[1]);
      x = 10;
   }
   extern int x;                    /* Re-declares x without defining it */
   printf("New value of x = %d\n", x);
   return 0;
}
int x = 5;                          /* Defines x but without using extern */
int y = 50;                         /* Not seen in main */
int arr[2];                         /* Defines arr but without using extern */
```

PROGRAM 11.12: **extern.c**

```
Initial value of x = 5
Initial value of y = -1209016480                    Uninitialized value
Elements of arr = 10, 20
New value of x = 10
```

PROGRAM OUTPUT: **extern.c**

The extern feature is the only way multiple files can share a variable so that the updates to the variable are visible at all desired locations. Security would be compromised if a simple global variable is used without using **extern**.

Note: When using **extern**, there must be a single definition of the variable that doesn't use the **extern** keyword.

Tip: If a variable has to be made accessible in selected functions, define the variable after all functions and then use the **extern** qualifier in only those functions that need to access the variable.

11.17.5 Register Variables (register)

All program variables are stored in primary memory which is faster than disk but slower then registers *(1.9.4)* directly connected to the CPU. The *register* storage class (keyword: **register**) permits a variable to be stored in one of the high-speed CPU registers. A register variable is declared inside a function or block in the following manner:

```
register int x;
```

The compiler will *attempt* to store the variable x in one of the registers, failing which it will use primary memory. Because there are a small number of these registers, only those variables that are frequently accessed (say, a key variable of a loop) should be declared as register variables. The scope, lifetime and initial value of a register variable is the same as those for an automatic variable (Table 11.1).

Because a register normally has a size of one word, not all variables can be assigned this storage class. Generally, **register** is restricted to char and int variables. However, you can't obtain the address of a register variable using &x because registers don't use the addressing system used by primary memory.

WHAT NEXT?

We failed to swap two variables using a function. We are yet to explain why **scanf** needs the & prefix. We used our scant knowledge of memory addresses to change array elements, but we couldn't really assess how important these addresses are in C. The next chapter solves all of these problems and opens up a new world of opportunities. C is all about pointers.

WHAT DID YOU LEARN?

A function is invoked with or without *arguments* and may or may not return a value. A function may be used for its return value or side effect, or both.

A function *declaration* specifies its usage and the *definition* provides the code to be executed. Functions of the standard library are declared in "include" files and their definitions are available as compiled code in *libraries* (archives).

When a function is invoked, its *actual arguments* are passed *by value* to its *formal arguments* (parameters). The function cannot change the actual arguments by changing the parameters. C doesn't support passing *by reference* even though it can use memory addresses to create that effect.

The name of an array represents the address of the first element. A function using an array name as an argument also needs to know (i) the number of elements for a 1D array, and (ii) the number of rows for a 2D array.

A function can call another function, which in turn can call another, and so forth. The called functions return in the reverse order of their invocation.

A *recursive* function calls itself until the *terminating condition* halts further invocations. The *iterative* approach is often preferable to the recursive technique because the former doesn't cause a *stack overflow*.

A *local* variable (**auto**) is visible only in the function or block where it is defined. It is extinguished when the function terminates, unless the **static** qualifier is used. A *global* variable is visible in the entire program.

An **extern** variable is defined once but can be made visible at other locations using proper declarations.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

11.1 A function must either accept one or more arguments or return a value.

11.2 The compiler compares the definition of a function with its declaration.

11.3 A function without arguments need not use the () suffix.

11.4 A function that returns a value can be treated as an expression.

11.5 A function can't both return `void` and also use `void` in the argument list.

11.6 A function can call other functions except itself.

11.7 The function body must be enclosed within curly braces even if the body comprises a single statement.

11.8 It is impossible for a function **f(x)** to change x because x is destroyed after the function terminates.

11.9 A function can be used for its side effect or return value or both.

11.10 A **while** loop can run forever, but not a recursive function.

11.11 A program that uses **int main(void)** cannot be run with an argument.

11.12 Every time the keyword **extern** is used, separate storage is allocated for the variable.

A2. FILL IN THE BLANKS

11.1 The _____ of a function determines its usage while the _____ provides the code that will be executed.

11.2 A function that doesn't return a value must be declared as _____.

11.3 A _____ function repeatedly calls itself.

11.4 The file `stdio.h` contains the _____ of the standard I/O functions.

11.5 Pass-by-value implies that actual arguments are _____ to the formal arguments of a function.

11.6 The parameters and local variables of a function are stored in a region of memory called the _____.

11.7 A _____ variable defined inside a function is created only once irrespective of the number of calls made to the function.

11.8 The _____ statement terminates the current function, but the _____ function terminates the program no matter where it is invoked from.

11.9 The **main** function is never _____ in a program.

11.10 The **extern** keyword is used to _____ a variable. The variable must be _____ elsewhere.

11.11 By default, a variable defined inside a function belongs to the _____ storage class.

11.12 The _____ keyword is used to store a variable in high-speed memory directly connected to the CPU.

A3. MULTIPLE-CHOICE QUESTIONS

11.1 The statement **return sum;** in a function (A) terminates the program after transmitting the value of sum to the operating system, (B) transmits the value of sum to the caller of the function, (C) terminates the function, (D) B and C, (E) A, B and C.

11.2 The declaration of a function (A) must precede its usage in **main**, (B) must be followed by its definition, (C) must use the same parameter names as the definition, (D) A and B, (E) none of these.

11.3 A local variable of a function (A) can be transmitted to the caller using **return**, (B) is destroyed before it can be returned, (C) cannot be transmitted using **return**, (D) none of these.

11.4 When the name of an array is passed as an argument to a function, (A) the array is copied inside the function, (B) the address of the first array element is copied, (C) the function can determine the size of the array using **sizeof**, (D) none of these.

11.5 A variable declared outside a function is (A) visible inside **main**, (B) not visible in any function, (C) visible from the point of its declaration to end of program, (D) none of these.

11.6 If a variable is declared inside a function as **static**, (A) it is initialized only once, (B) it retains its value between multiple function calls, (C) it is destroyed only on program termination, (D) all of these.

CONCEPT-BASED QUESTIONS

11.1 Explain how the compiler acts on the declaration, definition and invocation of a function in a program.

11.2 Explain the circumstances when the return value of the following functions may or may not be ignored: **printf**, **scanf**, **getchar**.

11.3 Explain why the names of the arguments used to invoke a function don't conflict with the names used for the parameters in the function definition.

11.4 Examine the following code snippet and explain why **printf** prints different values in **main** and **func**:

```
int main(void)
{
   short arr[100];
   printf("Size of array = %hd\n", sizeof(arr));
   func(arr);
   return 0;
}
void func(short f_arr[])
{
   printf("Size of array = %hd\n", sizeof(f_arr));
   return;
}
```

11.5 How is passing the element `arr[5]` different from passing the name, `arr`, to a function? Explain whether the function can change the contents of `arr` defined in the caller.

11.6 Explain the circumstances that determine whether a variable is declared before **main**, inside **main** or after **main**.

11.7 Explain the concept of variable hiding using a suitable example.

PROGRAMMING & DEBUGGING SKILLS

11.1 Identify the errors in the following program:

```
#include <stdio.h>
void func(void)
int main(void)
{
   func(void);
   return 0;
}
void func(void)
{
   printf("Elementary penguin\n");
   return;
}
```

11.2 Write a program containing a function named **volume** which returns the volume of a sphere. The function is passed the radius as a single argument.

11.3 Modify the preceding program in C11.2 so that the **volume** function handles a cube and a sphere. Incorporate **#define** statements that would enable the function to be invoked in the following two ways:

```
float radius, length, vol;
vol = volume(radius, SPHERE);
vol = volume(length, CUBE);
```

11.4 Write a program that uses a function named **take_input** to let a user input an *unspecified* number of integers (not exceeding 100) before hitting EOF. The values should be stored in an array defined in **main** and the function is passed the name of this array as argument. How is this program superior to Program 11.5 in Chapter 11?

11.5 Create a function named **print_array** that prints the contents of an array. The function must take an additional argument that determines whether the array is printed in forward or reverse sequence.

11.6 Write a program that uses a function named **copy_array** to copy an initialized array.

11.7 Write a program that uses a function named **copy_array2** to copy a 2D array that is defined and initialized in **main**.

11.8 Write a program that uses a function named **day_diff** to compute the number of days between two days *of the same year* (both days inclusive). The days are input in the format *DD:MM* and don't relate to a leap year.

11.9 Devise a function named **time2download** that accepts the Internet speed in Mbps and file size in MB as two arguments. The function should return the download time in hours as a floating point number. (Note: Speed uses bits and file size uses bytes.)

11.10 The volume of a cylinder is the product of the area of the base (PI * r * r) and the height. Write a program that accepts the radius and height from the user and invokes a function named **volume** which in turn invokes a function named **area**.

11.11 Write a program that draws a rectangle using the **draw_rectangle** function. This function invokes the **draw_line** function for drawing the horizontal sides. Both functions need an additional argument that determines the character used for drawing.

11.12 Write a program that uses recursive functions to (i) compute the sum of array elements, (ii) print the elements.

11.13 Write a program that displays the Fibonacci numbers up to *n* terms using (i) a recursive function named **fibonacci_rec**, (ii) an iterative function named **fibonacci_ite**. The value of *n* is set by user input. The program must track and print the total number of recursive calls and loop iterations made by the two functions. As you increase the value of *n, what observations can you make from the results?*

12 Pointers

WHAT TO LEARN

- Accessing memory for fetching and updating the stored value.
- Significance of *indirection* and *dereferencing* when applied to a pointer.
- Using pointers as function arguments to change variables defined outside the function.
- Safety features associated with pointers used as function arguments.
- Using function parameters to return multiple values.
- Relationship between arrays and pointers.
- Special features of *pointer arithmetic* and its limitations.
- Significance of the *null* and *void* pointers.
- Advantage of using an array of pointers for sorting operations.
- The implicit and explicit use of a pointer to a pointer.
- Use of **`const`** to protect a pointer and its pointed-to value from modification.

12.1 MEMORY ACCESS AND THE POINTER

A program evaluates a variable by accessing its memory location and retrieving the value stored there. Because this process is hidden in our programs, you may be surprised to know that a program executable doesn't contain any variable names at all. Instead, it contains their addresses as components of instructions. Before the advent of 3GL languages, programmers had to explicitly specify memory locations for data, while taking care that these locations didn't conflict with one another. Today, the same job is done by the compiler.

C permits a *partial* return to those old days by allowing every *legal* area of memory to be directly accessed. (We say "legal" because not every memory location is accessible by our programs.) The language allows us to obtain the memory location of every variable, array or structure used in a program. We can then use the address of this location to access—and even change—the value stored there.

But why should one want to *indirectly* access a variable when one can do that directly? The simple answer is that sometimes direct access is *not* possible. Recall that a function copies a variable that is passed to it as an argument, which means that it can't *directly* access the original variable. If access to individual memory locations is permitted instead, then we can do the following:

- Use the address to change any variable defined anywhere—even in places where the variable is not visible but its address is.
- Make a function return multiple values by passing multiple addresses as function arguments.
- Access or change the elements and members of bulky objects like arrays and structures without copying them inside a function. All you need is knowledge of the address of the first element or member of the object.

A *pointer*, the most powerful and misunderstood feature of C, easily performs all of the preceding tasks. Pointers are hidden in arrays and strings even though we have cleverly managed to use these objects—often without adequate understanding of what we were doing. But it's in this chapter that we need to examine the pointer in all its manifestations and understand why C can't do without them. C without pointers is like a mechanic without the toolkit.

Note: Two values are associated with any variable—the address of its memory location and the value stored at the location.

12.2 POINTER BASICS

A *pointer* is a variable with a difference. It contains, not a simple value, but the address of another variable or object. So, if a pointer `p` is assigned the address of an `int` variable `x` which is set to `5`, then

```
p = &x;                          p now points to x
```

where `&x` evaluates to the memory address of `x`. The pointer `p` must have been declared previously as

```
int *p;                          Declares pointer variable p
```

C offers two unary operators, the `&` and `*`, for handling pointers. The `&` is used to obtain the address of a variable. The `*` is used for two purposes and one of them is to declare the pointer. When `p` is assigned the address of `x`, `p` is said to *point* to `x`, which means you can use `p` to access and change `x`.

If `p` is visible at some location of a program but `x` is not, you can still access `x` at that location by *dereferencing* `p` to obtain the value at the address pointed to, i.e. `x`. The expression `*p` (second use of `*`) evaluates to `x`:

```
printf("x = %d\n", *p);          Prints x = 5
```

While `*p` represents the value of `x`, it can also be used on the left-hand side of an assignment to change the value of `x`:

```
*p = 10;                         x is now 10
```

The power of a pointer lies in the preceding statement. The assignment `*p = 10` is the same as `x = 10`. A pointer thus provides access to two values—the address it stores and the value stored at the address. The type of this pointer is not `int`, but *pointer to* `int`, which we also refer to as `int *`.

Even though a pointer has an integer value, it supports a limited form of arithmetic. Adding 1 to a pointer value of 100 yields 101 when the pointed-to variable is of type `char`, and 104 when the pointed-to variable is of type `int`. Using pointer arithmetic, we can navigate memory and update the contents at any legal memory location. You can change the value stored in the pointer or the value pointed to:

`p++`	*Adds 1 to value of pointer*
`(*p)++`	*Adds 1 to value pointed to*

Like arrays, pointers belong to the category of derived data types. Since every variable has an address, we can define pointers for every data type, including arrays and structures. Furthermore, nothing in C prevents us from creating a pointer to another pointer.

Even though a pointer is usually a variable, any expression that evaluates to an address in memory can be used where a pointer value is expected. Arrays and strings can also be treated as pointers for most purposes. The name of an array evaluates to the address of the first element, while a double-quoted string evaluates to the address of the first character of the string.

Note: A pointer has a data type that is aligned with the data type of its dereferenced value. If a pointer `p` points to a variable of type `int`, then the type of `p` is `int *`, i.e., pointer to `int`. It would also be helpful to think of `*p` as a value having the type `int`.

Takeaway: A pointer value can originate from three sources: (i) any variable name prefixed with the & operator, (ii) the name of an array, and (iii) a string enclosed within double quotes.

12.3 `intro2pointers.c`: A SIMPLE DEMONSTRATION OF POINTERS

Program 12.1 establishes the minimal assertions that have been made so far about pointers. The uninitialized value of a pointer, `p`, having the type `int *`, is first printed in hex with the `%p` format specifier. It is then made to point to an `int` variable `x`. Observe that both `*p` and `x` evaluate to `420`. But when `*p` is assigned the value `840`, the value of `x` also changes to `840`. This shows that the pointer can be used to change the value pointed to.

12.4 DECLARING, INITIALIZING AND DEREFERENCING A POINTER

A pointer variable is declared (and automatically defined) before use. Declaration specifies the type and allocates the necessary memory for the pointer. The following statement declares a pointer variable named `p` that can store the address of an `int` variable:

`int *p;`	`p` *can't be dereferenced now*

By convention, the `*` is prefixed to the variable, but you can place it anywhere between `int` and `p` (say, `int * p` or `int* p`). The pointer `p`, which now contains an unpredictable value, can be used

```
/* intro2pointers.c: A simple example of a pointer in action. */
#include <stdio.h>
int main(void)
{
   int x = 420;
   int *p;                                          /* p has type int * */
   printf("Value of x = %d\n", x);
   printf("Uninitialized value of pointer p = %p\n", p);
   p = &x;                                          /* p stores address of x */
   printf("Initialized value of pointer p = %p\n", p);
   printf("Value at address pointed to by p = %d\n", *p);
   printf("This is the same value as x: %d\n", x);
   *p = 840;                                        /* Same as x = 840; */
   printf("New value of x = %d\n", x);
   return 0;
}
```

PROGRAM 12.1: **intro2pointers.c**

```
Value of x = 420
Uninitialized value of pointer p = 0xb7f1df60          %p prints in hex
Initialized value of pointer p = 0xbfa1b63c            Address of x
Value at address pointed to by p = 420
This is the same value as x: 420
New value of x = 840
```

PROGRAM OUTPUT: **intro2pointers.c**

only after it is made to point to a legal memory location. This can be the address of an `int` variable (say, `x`), which is obtained by using the `&` operator:

```
int x = 5;
p = &x;                                          p stores the address of x
```

The `&` operator evaluates its operand (`x`) as an address. Expectedly, we can combine the declaration and assignment in this manner:

```
int *p = &x;                                     int* p = &x; is also OK
```

`p` has the data type `int *`, or *pointer to* `int`, which simply means that the value pointed to has the type `int`. Similarly, a `float` variable is pointed to by a pointer of type `float *` or pointer to `float`. However, the size of the pointer itself has no relation to the size of the variable pointed to. A pointer has a fixed size and its value is printed with the `%p` format specifier of **`printf`**.

After a pointer has been assigned a valid address with `p = &x;`, we can use the concept of *indirection* to obtain the value of `x` from `p`. The expression `*p` evaluates to `x`:

```
printf("Value of x is %d\n", *p);                Prints the value 5
```

The *, used as a unary operator in dereferencing or indirection, operates in the following sequence:

- It evaluates its operand (p) to obtain the stored address (&x).
- It then fetches the value (x) stored at that address.

Hence, *p is synonymous with x and can be used wherever x is used. Thus, *p changes x when used in the following way:

```
*p = 10;                    Same as x = 10;
```

As long as p points to x, any change made to x can be seen by p, and vice versa. *This is the most important feature of a pointer.* p thus has two values—a direct (&x) and an indirect one (*p). The dereferenced value of p can also be assigned to another variable or used in an arithmetic expression:

```
int y;
y = *p;                     y is also 10
y = *p + 1;                 Same as y = x + 1;
```

Just as x is a variable on the left side and a value on the right side, the same is true of *p. The relationship between a pointer and the variable it points to is depicted in Figure 12.1.

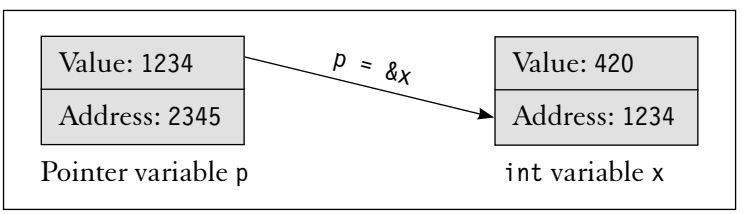


FIGURE 12.1 Pointer and the Variable Pointed to

Note: C needs more operators than the computer keyboard can provide, so sometimes the same symbol has to take on multiple roles. The *, the operator used in multiplication, is also used for declaring a pointer (`int *p;`) and for dereferencing it (`int y = *p;`).

Note: The statement `int *p;` can be interpreted in two ways:

(i) p is of type `int *` (pointer to `int`).

(ii) *p is of type `int`.

This flexibility explains the two ways the * is used. Because of (i), many programmers prefer to write the declaration as `int * p;` instead of `int *p;`.

12.5 pointers.c: USING TWO POINTERS

Program 12.2 uses two pointers p and q of type pointer to `short` (`short *`), and two `short` variables x and y. The first part, where both p and q point to x, shows the effect dereferencing p and q has on the value of x. The second part demonstrates the effect on *q when p is made to point to y.

```
/* pointers.c: Uses two pointers to point to same and different variables. */
#include <stdio.h>
int main(void)
{
   short x = 100, y = 300;
   short *p = &x;
   short *q = p;                    /* Both p and q point to x */
   printf("When x = 100, *p = %hd, *q = %hd\n", *p, *q);
   *p = 150;                        /* Same as x = 150; */
   printf("When *p = 150, *q = %hd, x = %hd\n", *q, x);
   *q = 200;                        /* Same as x = 200; */
   printf("When *q = 200, *p = %hd, x = %hd\n", *p, x);
   p = &y;                          /* q now points to y */
   printf("\np now points to y, *p = %hd, y = %hd\n", *p, y);
   printf("But *q is still %hd\n\n", *q);
   return 0;
}
```

PROGRAM 12.2: **pointers.c**

```
When x = 100, *p = 100, *q = 100
When *p = 150, *q = 150, x = 150
When *q = 200, *p = 200, x = 200
p now points to y, *p = 300, y = 300
But *q is still 200
```

PROGRAM OUTPUT: **pointers.c**

At the time of declaration, the address of x is directly assigned to p and indirectly to q. The first three **printf** statements show how changing *p or *q also changes x, which means that *p, *q and x are equivalent in all respects. In the second part, the linkage between p and x is broken by making p point to y. q is now no longer a copy of p because it continues to point to x.

You can now understand why a pointer needs to have a data type. Without it, you can access the address it contains all right, but you can't dereference the pointer. The type `short *` in the declaration of p and q indicates that two bytes have to picked up from the address stored in each pointer and these bytes have to be interpreted as a value of type `short`.

Takeaway: The assignment `p = &x` sets up a reciprocal relationship between two variables. x can be changed by modifying *p, and vice versa. This relationship is non-existent in the assignment `x = y;` that involves two ordinary variables. Here, *subsequently* changing one variable doesn't change the other.

12.6 IMPORTANT ATTRIBUTES OF POINTERS

Because of the sheer power of indirection, pointers are remarkably effective in solving complex problems. For this reason, they are used almost everywhere except in the most trivial applications. Admittedly, it takes some time for a beginner to get used to these simple but tricky concepts. However, adoption of the following features and guidelines will stand you in good stead:

- *The data type of a pointer is different from the type of the variable it points to.* If the latter is of type `char`, then the type of a correctly defined pointer is `char *`, i.e., pointer to `char`. Also, the data type of a variable or constant can be transformed into the data type of the pointer using a cast. For instance, the expression `(int *) 32657` transforms an `int` to a pointer to `int`.
- *The size of a pointer is generally fixed and doesn't depend on the size of the object it points to.* A pointer, being an address, represents an integer value that is wide enough to address every byte of memory on a machine. Exceptions notwithstanding, a pointer has the same size as the native word length, which often has the size of an `int`. But the pointer itself is not an `int`.
- *A pointer can be assigned the value 0.* The address 0 is also represented by the symbolic constant `NULL`, and a pointer that has this value is known as a *null pointer*. C guarantees that the address `NULL` can never point to an object used by the program.
- *An uninitialized or null pointer must not be dereferenced.* An uninitialized pointer contains a garbage value which may point to a memory location that is not accessible by your program. Your program may either behave unpredictably or terminate abnormally if such a pointer is dereferenced.
- *Adding 1 to a pointer variable doesn't necessarily add 1 to the value stored by the pointer.* The result depends on the data type of the pointer. If a pointer stores the address 1000, then adding 1 to it results in 1001 for a pointer of type `char *`, 1004 for `int *` and 1008 for `float *`. This addition belongs to the realm of pointer arithmetic, which forms the basis of the relationship between pointers and arrays.
- *A pointer can snap its current link and point to another variable.* This has already been established by the previous program (Program 12.2). For instance, if a pointer `p` points to the variable `x`, and you are done with `x`, then you can use the same pointer to point to another variable `y` (Figure 12.2).

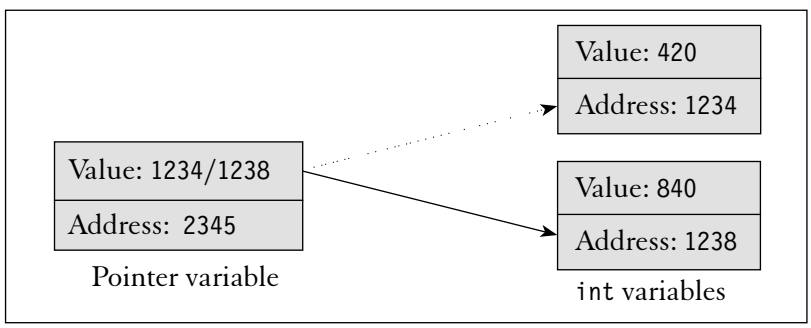


FIGURE 12.2 Pointer to Two Different Variables

- *A pointer variable can be assigned to another pointer variable provided they are of the same type* In the following example, both `p` and `q` point to `x`:

```
int x = 10;
int *p = &x;
int *q = p;
```

This has also been established in the preceding program (Program 12.2). Dereferencing `p` and `q` yield the same value `x` (Fig. 12.3). You can also change the value of `x` by changing either `*p` or `*q`. This is exactly what happens when a pointer is passed as an argument to a function.

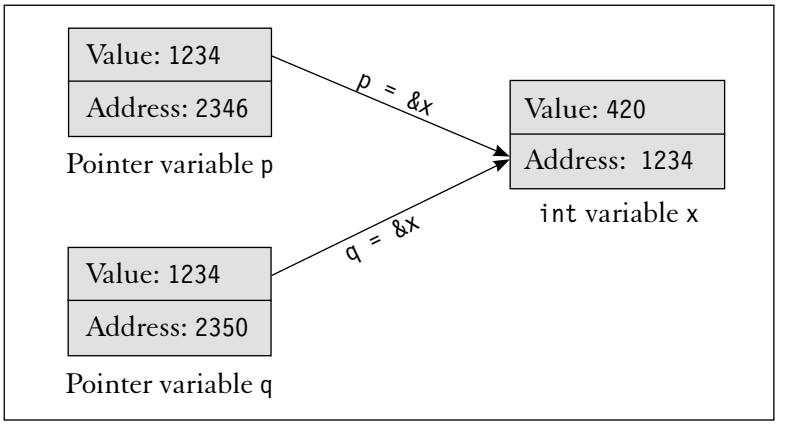


FIGURE 12.3 Two Pointers to the Same Variable

- *A pointer can point to another pointer.* This property is implicitly exploited in a multi-dimensional array or an array of pointers to strings. The pointer pointed to can, in turn, point to another pointer, and so on.

Program 12.3 demonstrates three features of pointers. It uses three ordinary variables of type `char`, `short` and `int` along with three matching pointer variables. The program eventually crashes without executing the final **`printf`** statement.

(A) Here, **`sizeof`** is used to compare the size of the pointer to the size of the variable it points to. Note that the data type of the pointer doesn't determine its own size (4 bytes on this Linux system) even though it determines the size of the dereferenced objects (`*p_char`, `*p_short` and `*p_int`).

(B) When 1 is added to each of the three pointer variables, the resultant values increase not by 1, *but one unit of the data type*. It's only the value of `p_char` that advances by 1; `p_short` and `p_int` increase their values by 2 and 4, respectively. Pointer arithmetic will be examined in Section 12.9.

```
/* pointer_attributes.c: Highlights (i) difference between size of pointer and
                 its dereferenced object, (ii) effect of adding 1 to a pointer,
                 (iii) effect of dereferencing a null pointer. */
#include <stdio.h>

int main(void)
{
   int i = 97;        char c = 'A';        short s = 97;
   int *p_int = &i;   char *p_char = &c;   short *p_short = &s;

   /* (A) Size of pointer compared to size of dereferenced object */
   printf("Size of: p_char = %d, *p_char = %d\n",
           sizeof p_char, sizeof *p_char);
   printf("Size of: p_short = %d, *p_short = %d\n",
           sizeof p_short, sizeof *p_short);
   printf("Size of: p_int = %d, *p_int = %d\n",
           sizeof p_int, sizeof *p_int);

   /* (B) Adding 1 to a pointer */
   printf("p_char = %p,  p_char + 1 = %p\n",  p_char, p_char + 1);
   printf("p_short = %p, p_short + 1 = %p\n", p_short, p_short + 1);
   printf("p_int = %p,   p_int + 1 = %p\n",   p_int, p_int + 1);

   /* (C) Dereferencing a null pointer - Program will crash here!!!!!!! */
   p_short = NULL;                          /* Pointer snaps its current link */
   printf("Dereferenced value of a null pointer = %hd\n", *p_short);
   return 0;
}
```

PROGRAM 12.3: **pointer_attributes.c**

```
Size of: p_char = 4, *p_char = 1
Size of: p_short = 4, *p_short = 2
Size of: p_int = 4, *p_int = 4
p_char = 0xbfc22833,  p_char + 1 = 0xbfc22834              Increases by 1
p_short = 0xbfc22830, p_short + 1 = 0xbfc22832             Increases by 2
p_int = 0xbfc22834,   p_int + 1 = 0xbfc22838               Increases by 4
Segmentation fault                                                    Linux
pointer_attributes.exe has encountered a problem and needs to close.  Visual Studio
```

PROGRAM OUTPUT: **pointer_attributes.c**

(C) The pointer variable, `p_short`, is unhooked from the variable `s` and assigned the value `NULL`. The program terminated prematurely when this pointer was dereferenced, and without executing the final **`printf`** statement. GCC (Linux) and Visual Studio output different messages (both shown in output), but the meaning is clear: you cannot dereference a null pointer.

Caution: It is a common mistake to dereference a pointer that doesn't point to an object or an accessible memory location. For instance, the following sequence may not update the contents at the address stored in `p`, but instead may cause the program to crash:

```
int *p, x;
*p = 1;                                    Should have done p = &x; first
```

An uninitialized pointer points to "somewhere in memory," which can be an illegal memory location. Assign the pointer an address of an object before you dereference the pointer.

12.7 POINTERS AND FUNCTIONS

The preceding programs demonstrated the basic features of pointers and highlighted the importance of using them safely. But they comprised manufactured situations that actually provided no justification for using pointers at all. It's only when you use functions that you realize that C needs pointers for the following reasons:

- Functions pass their arguments by value. Because the *values* of arguments are copied to function parameters, these copies can't be used to change the originals.
- A function uses the **return** statement to return a single value. It can't return multiple values using this mechanism.

We can overcome both hurdles by using pointers as function arguments. Let's overcome the first hurdle first. Program 12.4 passes the address of a variable x to the **change_x** function which uses this address to double the existing value of x. The function is invoked twice to return the changed value of x in two different ways. Observe the declaration of the **change_x** function; it uses a pointer to short as an argument.

```
/* change_x.c: Uses a pointer as an argument to a function
                   to change a variable defined in main. */
#include <stdio.h>

short change_x(short *fx);

int main(void)
{
   short x = 100;
   short *px = &x;

                                                        /* Return value ... */
   change_x(px);                                        /* ... not saved here ... */
   printf("New value of x: %hd, %hd\n", *px, x);
   printf("New value of x: %hd\n", change_x(px));       /* ... but captured here */

   return 0;
}

short change_x(short *fx)
{
   *fx *= 2;
   return *fx;
}
```

PROGRAM 12.4: **change_x.c**

```
New value of x: 200, 200
New value of x: 400
```

PROGRAM OUTPUT: **change_x.c**

The pass-by-value mechanism *(11.5.2)* applies here in the usual manner. The pointer px, which points to x, is copied to the parameter fx. It's true that fx can't change px, but that doesn't concern us here. We want fx to change x instead. Because both px and fx point to x, changing *fx inside the function updates x defined in **main**. In other words, x has been changed through a copy of the pointer.

x is changed yet again in the program, but this time through the return value of the function. So which route should one adopt? And, do we really need to use px instead of &x? We'll answer this question soon. But now that we have discovered a new face for the pass-by-value principle, we need to exploit the same principle in a function to return multiple values.

12.7.1 swap_success.c: Making the swap Function Work

The concept of passing arguments by value hit a roadblock in the program, **swap_failure.c** (Program 11.3), which failed to swap the contents of two variables passed as arguments to the **swap** function. That version of the function swapped, not the original arguments, but their copies. Program 12.5 solves the swapping problem using addresses of variables as arguments.

```
/* swap_success.c: Swaps contents of two variables using a function that
                   accepts pointers to these variables as arguments. */
#include <stdio.h>
void swap(short *fx, short *fy);
int main(void)
{
   short x = 1, y = 100;                  /* Need to swap these two values */
   short *px = &x, *py = &y;
   printf("In main before swap      : x = %3hd, y = %3hd\n", x, y);
   swap(px, py);
   printf("In main after first swap  : x = %3hd, y = %3hd\n", x, y);
   /* Don't really need to use px and py; &x and &y will do. */
   swap(&x, &y);
   printf("In main after second swap: x = %3hd, y = %3hd\n", x, y);
   return 0;
}
void swap(short *fx, short *fy)           /* fx and fy are pointers */
{
   short temp = *fx;              /* Saves original value of x */
   *fx = *fy;                     /* Copies y to x in main */
   *fy = temp;                    /* Copies original value of x to y in main */
   return;
}
```

PROGRAM 12.5: **swap_success.c**

```
In main before swap      : x =   1, y = 100
In main after first swap  : x = 100, y =   1
In main after second swap: x =   1, y = 100
```

PROGRAM OUTPUT: **swap_success.c**

The first invocation of **swap** uses two pointer variables, px and py, that point to x and y, respectively. As both px and fx contain the address of x, changing *fx updates x and changing *fy updates y. The power of pointers shows up in this statement:

```
*fx = *fy;
```

The statement says this: *Fetch the value from the address stored in* fy *and assign it to the address stored in* fx. This action effectively assigns the value of y to x outside the function. The **swap** function works this time because it uses pointers as function arguments to change the value of variables declared outside the function. This shows the power of indirection!

The first invocation of **swap** used two pointer variables, px and py, as arguments. But **swap** simply needs two addresses having the proper data type. We thus don't need to define the pointers, px and py, at all. The second call to **swap** passes the addresses of x and y to reverse the effect of the previous swap operation.

Note: Because the **swap** function is passed the address (also called *reference*) of two variables, there's a view that C can also use pass-by-reference, or at least, create the "effect" of pass-by-reference *(11.5.2)*. In reality, it's the programmer, and not C, who creates this effect by using an address as a function argument and *dereferencing its copy* inside the function. Make no mistake: C uses pass-by-value at all times, including when using pointers.

Takeaway: For a function that uses a regular variable or expression as an argument, there is a one-way flow of data from the argument to the parameter. But when the argument is a pointer and the function changes its pointed value, the changed value is transmitted from the parameter to the argument. This represents a two-way communication using the *argument-parameter* route.

12.7.2 Using Pointers to Return Multiple Values

Let's now closely examine the two functions that we last used. The **change_x** function (in **change_x.c**) returned one value (*fx), but it also made the same value available in the parameter. On the other hand, the **swap** function (**swap_success.c**) returned nothing (Fig. 12.4). But if **swap** could change two variables x and y that are defined outside the function, is it not true that, in a sense, **swap** "returns" two values? Do we compulsorily need to use **return** to return values?

There is merit in this argument. Pointer variables used as function arguments act like containers which the function uses to "deposit" data. **change_x** uses one container while **swap** uses two. But **change_x** also offers an extra option (the **return** route) that we need to take note of. Now let's consider the following **scanf** statement taken from Program 11.8:

```
scanf("%hd%hd%hd%hd", &hours1, &mins1, &hours2, &mins2);
```

```
short change_x(short *fx)
{
   *fx *= 2;
   return *fx;
}
```

```
short x = 1, y = 100;
swap(&x, &y);
```

FIGURE 12.4 Segments from **change_x.c** and **swap_success.c**

Here, **scanf** "returns" five values—one through its normal return route (which is not captured here), and the other four through the pointers used as arguments. In that case, if we design a function to return three values, should one of them use the **return** route and the other two use pointers as arguments? Before we answer that question, we need to take up a program that uses this mechanism of using multiple pointers as arguments to a function.

Takeaway: When pointers are used as function arguments, the term "return" acquires a wider meaning. Apart from its usual meaning (the value passed by the **return** statement), the term also refers to the values placed in the addresses passed as arguments to the function.

12.7.3 sphere_calc.c: Function "Returning" Multiple Values

Program 12.6 takes the radius of a sphere as input and computes its total surface area and volume. Both values are "returned" through two pointers used as arguments to a function. This time we use the **pow** function of the math library to compute two expressions.

The **sphere_area_vol** function requires three arguments. The first argument (`radius`) provides the input. The other two are addresses of `area` and `vol`, two variables defined in **main**. These addresses are copied to the parameters `f_area` and `f_vol` inside the function. When the function computes `*f_area`, it updates the value at the address represented by `&area`. This is the variable `area` itself. Similarly, `*f_vol` updates `vol`. In this way, the function returns two values using the pointer route.

Note: The first argument in this program provides for one-way transmission of information. But the remaining arguments (being pointers) are used for two-way communication.

Would it have made sense to return one of these values, say, `area`, through the **return** statement? The function would then need two, and not three, arguments. Nothing stops us from doing that but the answer would actually depend on the requirements of the application. Generally, a function uses the *pointer-argument* route when at least two values have to be returned. The **return** statement is then used to perform error checking. This is what we have done with our **sphere_area_vol** function of the preceding program.

Note: The program must be compiled with the `-lm` option when using the **gcc** command. This option directs the linker to obtain the code for **pow** from the math library. Visual Studio performs the task of linking automatically.

```
/* sphere_calc.c: Uses a function that returns two values using pointers.
                  Program needs linking with math library. */
#include <stdio.h>
#include <math.h>                       /* Required by pow function */
#define PI 3.142

short sphere_area_vol(float f_radius, double *f_area, double *f_vol);

int main(void)
{
   float radius;
   double area, vol;

   printf("Enter radius of sphere: ");
   scanf("%f", &radius);
   if (sphere_area_vol(radius, &area, &vol)) {
      printf("Radius = %.4f, surface area = %.2f, volume = %.2f\n",
              radius, area, vol);
      return 0;
   }
   else {
      printf("Illegal radius: %.4f\n", radius);
      return 1;
   }
}

short sphere_area_vol(float f_radius, double *f_area, double *f_vol)
{
   if (f_radius <= 0)
      return 0;
   *f_area = 4 * PI * pow(f_radius, 2);          /* Include math.h for pow */
   *f_vol =  4.0 / 3 * PI * pow(f_radius, 3);
   return 1;
}
```

PROGRAM 12.6: **sphere_calc.c**

```
Enter radius of sphere: -4
Illegal radius: -4.0000
```

```
Enter radius of sphere: 2
Radius = 2.0000, surface area = 50.27, volume = 33.51
```

PROGRAM OUTPUT: **sphere_calc.c**

12.7.4 Returning a Pointer

A function can also return a pointer using the **return** statement. The usual caveat applies; the returned address can't point to a data object that is locally declared in the function. A pointer as a returned value finds wide application when handling arrays and strings, and specially when dynamically allocating memory. We'll come across one instance of returning a pointer later in the chapter and some more in Chapter 16.

12.8 POINTERS AND ARRAYS

Pointers have an inseparable relationship with arrays. Like variables, array elements can be accessed and updated with pointers using *pointer arithmetic*. Also, a pointer is implicitly used whenever we pass an array name as argument to a function. The following examples should help refresh your memory:

```
input2array(arr, SIZE2, 'y');                          (11.8.1)
print_array(crr, SIZE1 + SIZE2);                       (11.9)
bubble_sort(arr, size, ASCENDING, PRINT_YES);          (11.12)
```

We had passed as arguments to these functions, not individual array elements, but the name of the array itself (`arr` and `crr`). This was done to take advantage of a convenience provided by C:

> *The name of the array is a pointer that signifies the base address, i.e. address of the first element of the array.*

This means that `arr` is the same as `&arr[0]`, the address of the first element, so `arr` and `&arr[0]` can be used interchangeably. Adding 1 to `arr` or `&arr[0]` yields `&arr[1]`, the address of the next element. Observe pointer arithmetic at work from the following examples:

`&arr[2] + 2` — Same as `&arr[4]`
`&arr[6] - 3` — Same as `&arr[3]`
`arr[4] + 3` — No `&`; adds 3 to the value of `arr[4]`
`&arr[0] - 2` — Same as `&arr[-2]`; C allows negative subscript.

The address represented by `arr` is a constant; it is allocated when the array is declared and can't be changed subsequently. For this reason, the name of an array is often referred to as a *constant pointer*. This is in contrast to a *pointer to constant* which can change the address that it stores but not its dereferenced value.

12.8.1 Using a Pointer for Browsing an Array

To navigate an array using its name as a pointer, we need to use an expression that relates the address of an array element (`&arr[`*n*`]`) to its base address (`arr` or `&arr[0]`). Using basic pointer arithmetic, this expression can be developed in a type-independent way. Consider this array named `month`:

```
int month[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

While `month = &month[0]`, the expression `month + 1` increases `month` by one *storage unit* to represent the address of the next element, i.e. `&month[1]`. The fourth element of `month` can be represented in a similar manner:

```
&month[3] = month + 3                    Adds 3 storage units to base address
```

The size of the storage unit is the number of bytes used by the data type. We can generalize the previous relation in the following manner:

```
&month[n] = month + n
```

This is the central feature of pointer arithmetic. While this expression itself is type-independent, its evaluated value is not. For instance, if `arr` has the base address 1000, `arr + 1` would be 1001 for `char`, 1002 for `short` and 1004 for `int`. C automatically scales the addition of a pointer, so the expression `arr + 1` evaluates to the address of the next element *irrespective of its data type*. This has already been established by Program 12.3.

12.8.2 Dereferencing the Array Pointer

The address of an array element is dereferenced in the usual manner by prefixing the expression representing the address with *, the dereferencing operator. This means `*month` evaluates to `month[0]` and `*(month + 1)` evaluates to `month[1]`. So, in the general case,

`month[`*n*`] = *(month + `*n*`)`

Table 12.1 highlights the equivalence of standard array notation and pointer notation. Internally, C uses pointer notation, so on encountering `month[1]`, C immediately converts it to `*(month + 1)`. The dereferenced expression can represent either a variable or value depending on which side of the = it occurs:

`x = *(month + 1);`	`x` *assigned value of* `month[1]`
`*(month + 1) = x;`	`month[1]` *assigned value of* `x`
`*(month + 2) = 29;`	`month[2]` *set to 29*
`*(month + 1) = *(month + 3);`	`month[1]` *assigned value of* `month[3]`

The parentheses are necessary because the unary * has a higher precedence than the arithmetic operators. If we use `*month + 1`, `*month` would be evaluated before the addition, so the result would be 1 and not 31 (`month[0]` = 0). In the following examples, the first one results in a wrong assignment, while the second one is illegal:

`*(month + 1) = *month + 5;`	`month[1]` *assigned value of* `month[0] + 5`
`*month + 1 = *month + 5;`	*Bad lvalue; illegal usage*

Pointer notation was considered to be more efficient than array notation, but today's powerful CPUs and smart compilers have virtually made this comparison meaningless. Most people feel comfortable using array notation, but the equivalent pointer notation is also important because you'll see it almost everywhere.

TABLE 12.1 Array Access with Pointers

`(short month[6] = {0, 31, 28, 31, 30, 31};)`

Access		*Dereference*		
Array Notation	*Pointer Notation*	*Array Notation*	*Pointer Notation*	*Value*
`&month[0]`	`month`	`month[0]`	`*month`	0
`&month[1]`	`month + 1`	`month[1]`	`*(month + 1)`	31
`&month[2]`	`month + 2`	`month[2]`	`*(month + 2)`	28
`&month[3]`	`month + 3`	`month[3]`	`*(month + 3)`	31
`&month[`*n*`]`	`month + `*n*	`month[`*n*`]`	`*(month + `*n*`)`	-

12.8.3 Treating a Pointer as an Array

We used expressions like `month + 1`, but can we use `month++` to represent `&month[1]`? No, we can't change the value of `month` because it is a constant pointer. We need to use a pointer variable, so let's create one named `p` and point it to `month`:

```
short *p = month;                                   month or &month[0]
```

Now, both `p` and `month` signify `&month[0]`. Even if `month++` is illegal, you can use `p++` to move the pointer to the next element and `p--` to step back to the previous element. You can also update any element of `month`, say `month[2]`, using the following expression:

```
*(p + 2) = 29;                                      Updates month[2]
```

But then we have just seen that `*(month + 2) = 29` is also legal. The preceding assignment has another important consequence: `p` *can be treated as an array even though it is not one.* This means you can access `month[2]` as `p[2]`, so the following assignment is also valid:

```
p[2] = 29;
```

Currently, `p` points to `month[0]`, but you can make it point to any element of this array:

```
p = &month[6];                                      p points to 7th element
```

You can now access the previous months as `p[-1]`, `p[-2]`, and so on. C treats a negative index as legitimate, so with `p` pointing to `month[6]`, `p[-6]` is now the same as `month[0]`. However, the value of `p[-7]` is unpredictable because it violates the bounds of the original array. From this relationship between an array and a pointer, what emerges is this:

> *If you have access to a contiguous section of memory and know its starting address, then you can treat this memory segment as an array.*

Declaration of an array makes this space available for an external pointer. We'll soon consider a program to establish this near-equivalence of an array and pointer. Chapter 16 extensively uses a dynamically allocated memory block as an array without declaring it in the usual way.

12.8.4 Array vs Pointer Which Points to an Array

Even though a pointer, after being made to point to an array, can subsequently be used like one, a pointer isn't really an array for the following reasons:

- The address it stores is not a constant. It can point to one array this moment and another variable or array in the next.
- The expression **`sizeof p`** signifies the size of the pointer. But **`sizeof month`** evaluates to the total number of bytes allocated for the `month` array. You need to keep this distinction in mind when you pass the name of an array as an argument to a function.
- On declaration of `p`, memory is allocated for only the pointer variable. For `p` to behave like an array, it must use a block of memory allocated to a regular array, i.e. it must point to an existing array.

The mechanism of using a copy of a pointer to access and alter the object of interest is often used by functions that use array names as arguments. In Section 12.12.1, we'll make a detailed examination of this property.

Takeaway: If the address of an element of the array `arr` is copied to a pointer variable `p`, then `p` can be used with both pointer and array notation to access `arr`. Thus, if `p = arr`, then both `p[`*n*`]` and `p + n` access `arr[`*n*`]`, but if `p = &arr[2]`, then `p[-2]` and `p - 2` access `arr[0]`.

12.8.5 `pointer_array.c`: Treating an Array as a Pointer

Program 12.7 uses a mix of array and pointer notation to access and update the elements of a small array. Using a pointer variable as an array, we'll learn that the use of a negative subscript is not only legal but also sometimes necessary.

```
/* pointer_array.c: Uses pointer arithmetic and a pointer variable to step
                    through and update array elements. */
#include <stdio.h>
#define SIZE 4
int main(void)
{
   int i, arr[SIZE] = {10, 20, 25, 35}, *p;
   printf("Contents of array elements:\n");
   for (i = 0; i < SIZE; i++)
      printf("*(arr + %d) = %d   ", i, *(arr + i));
   /* Changing arr[2] to 30 and arr[3] to 50 */
   *(arr + 2) = 30;
   *(arr + 3) = *(arr + 1) + *(arr + 2);     /* 20 + 30 */
   p = arr;                                  /* p points to base address of arr */
   printf("\nSeeing changes with p:\n");
   for (i = 0; i < SIZE; i++)
      printf("p[%d] = %d   ", i, p[i]);
   p = &arr[3];              /* p[0] is now arr[3] */
   printf("\nSet p to &arr[3], using p with a negative index:\n");
   for (i = 0; i > -SIZE; i--)
      printf("p[%d] = %d   ", i, p[i]);
   return 0;
}
```

PROGRAM 12.7: **`pointer_array.c`**

```
Contents of array elements:
*(arr + 0) = 10   *(arr + 1) = 20   *(arr + 2) = 25   *(arr + 3) = 35
Seeing changes with p:
p[0] = 10   p[1] = 20   p[2] = 30   p[3] = 50
Set p to &arr[3], using p with a negative index:
p[0] = 50   p[-1] = 30   p[-2] = 20   p[-3] = 10
```

PROGRAM OUTPUT: **`pointer_array.c`**

The first loop prints the values of the first four elements using pointer notation. After arr[2] and arr[3] are reassigned, the second **for** loop displays the array elements using a pointer variable p with array notation. p is finally made to point to arr[3], the last element. The "array" p must now be used with a negative index to access the previous elements of arr.

12.9 OPERATIONS ON POINTERS

Apart from dereferencing with the *, there are a number of operations that can be performed on pointers. All of the operations described in the following sections apply to pointer variables, and some to pointers representing arrays (constant pointers). We consider here three types of operations—assignment, arithmetic and comparison, using a pointer p that points to int.

12.9.1 Assignment

A pointer can be assigned the address of a variable (p = &c;) or the name of an array (p = arr;). It can also be assigned the value of another pointer variable of the same type. For instance, if both p and q are of type int *, then p can be assigned to q, and vice versa:

```
int c = 5; int *p, *q;
p = &c;
q = p;
```

It is possible to assign an integer, suitably cast to the type of the pointer, to a pointer variable using, say, p = (int *) 128;. Unless this address points to an existing object, it makes no sense to make this assignment.

However, you can, and often will, assign the value 0 to a pointer. This value has a synonym in the symbolic constant, NULL. You can assign either 0 or NULL to a pointer of any type without using a cast:

```
int *p = 0;
int *q = NULL;
char *s = NULL;
```

After these assignments, p, q and s become *null pointers*. We'll discuss the significance of NULL and the null pointer soon.

12.9.2 Pointer Arithmetic Using + and -

The domain of pointer arithmetic is small and simple. Among the basic arithmetic operators, only the + and - can be used with pointers. An integer may be added to or subtracted from a pointer. The resultant increase or decrease occurs in terms of storage units, so p + 2 advances the pointer by two storage units *(12.8.1)*. For an array, this means skipping two elements. The following operations on &x and an array element are also permitted:

```
&x - 2                  Address reduced by 2 storage units
&arr[3] + 3             This is &arr[6]
```

Addition of two pointers doesn't make sense, but subtraction between two pointers is valid provided both pointers refer to elements of the same array. The result signifies how far apart the elements are. Thus, the following expression

```
&arr[6] - &arr[3]
```

evaluates to 3, the difference between the subscripts of the array. In the general case, `&arr[`*m*`] - &arr[`*n*`]` evaluates to *m - n* irrespective of the data type of the array.

12.9.3 Pointer Arithmetic Using ++ and --

A pointer variable can also be used with the increment and decrement operators. Because `++` and `--` have the side effect of changing their operand, they can't be used with the name of the array which is a constant pointer. The following expressions are not permitted for the array `arr`:

```
arr++;                    Not permitted
arr--;                    Ditto
```

But if `p` points to `arr`, the expressions `p++` or `--p` are legal. `p++` increases the pointer to point to the next array element and `p--` decrements it to point to the previous element. If `p` points to an `int` variable, each `++` or `--` operation changes the numerical value of the pointer by 4 bytes.

The expressions `p++` and `p--` are often combined with the `*` for dereferencing the pointer. Expressions like `*p++` and `++*p` are commonly used in programs to evaluate the value pointed to by `p`, before or after the incrementing operation. Using the precedence table (Table 12.2) for these operators as reference, make sure you are not changing the value pointed to when you mean to change the value of the pointer. Consider the following expressions involving `*` and the `++` operator:

`int arr[] = {10, 20, 25, 35}; int *p = arr; (i.e., *p is 10)`

Expression	*Significance*	*Value of* `*p`
`*++p`	Increments `p` before evaluating `*p`	20
`++*p`	Increments value of `*p`	21
`*p++`	Increments `p` before evaluating `*p`	25
`(*p)++`	Increments value of `*p`	26

When the expression in the first example is evaluated, `p` advances from `arr[0]` to point to `arr[1]` before indirection is applied. `p` advances again in the third example to point to `arr[2]`. In the second and fourth examples, it's the value of `*p` that is incremented and not the pointer. The last example needs parentheses to change the default order of evaluation. Keep in mind that the postfix operators bind more tightly than the prefix ones, which have the same precedence as the `*` and `&`.

Tip: If you need to step through an array `arr` using the `++` and `--` operators, simply assign the array to a pointer variable `p` (`p = arr;`), and then use `p++` or `p--`.

TABLE 12.2 Precedence and Associativity of Pointer and Arithmetic Operators

Operators	*Description*	*Associativity*
`++ --`	Postfix increment/decrement	L-R
`++ --`	Prefix increment/decrement	R-L
`* &`	Dereference/Address	R-L
`* / %`	Arithmetic	L-R
`+ -`	Arithmetic	L-R

12.9.4 Comparing Two Pointers

Finally, two pointers can be compared only if they point to members of the same array. There are two exceptions though. First, any pointer can be compared to the null pointer (`p == NULL`) that is often returned by a function when an error condition is encountered. Also, a pointer of any type can be compared to a *generic* or *void* pointer. This pointer type is discussed in Section 12.17.

12.10 `max_min.c`: USING `scanf` AND `printf` WITH POINTER NOTATION

How does one use pointer notation to populate an array `arr` with **`scanf`** and display each element with **`printf`**? One way is to use the expression `(arr + n)` to step through the array where the variable `n` creates the required offset. We can then dereference the same expression to obtain the value. The following code segment achieves this task:

```
short n = 0, arr[20];
while (scanf("%hd", arr + n) != EOF)
   n++;
for (n--; n >= 0; n--)
   printf("%hd ", *(arr + n));                    Prints in reverse order
```

It would be more convenient, however, to scan the array with the `++` and `--` operators. We can't obviously use `arr++`, so let's use a pointer variable `p`, make it point to `arr`, and then use `p++` or `p--` to access each element. Program 12.8 uses this technique to print the maximum and minimum number found in an array.

In this program, **`scanf`** uses the pointer `p` as its second argument, while **`printf`** displays the value of `*p`. The counter `n` keeps track of the number of items read, and the **`while`** loop uses this value to determine the number of items to be printed in reverse order. The standard technique of determining the maximum and minimum values is to first initialize `max` and `min` to the first or last element of the array. After every iteration, `max` and `min` are compared to `*p` and reassigned if required.

12.11 `NULL` AND THE NULL POINTER

Sometimes we need to make a pointer variable point to "nowhere," i.e., not to an existing object. For this purpose, C supports a value that is "guaranteed to compare unequal to a pointer to any object or function." The expression `&x` cannot be guaranteed to yield such a value, so C allows the value 0 or the symbolic constant `NULL` to be assigned to a pointer:

```
/* max_min.c: Uses scanf and printf with pointer notation.
                  Also prints maximum and minimum numbers. */
#include <stdio.h>
int main(void)
{
   short n = 0, arr[20], max = 0, min = 0;
   short *p = arr;
   printf("Key in some integers and press EOF: ");
   while (scanf("%hd", p) != EOF) {
      n++; p++;
   }                                        /* p moves beyond last element ... */
   p--; n--; max = min = *p;                /* ... so has to be brought back */
   while (n-- >= 0 ) {
      printf("%hd ", *p);                   /* Prints in reverse order */
      if (*p > max)
         max = *p;
      else if (*p < min)
         min = *p;
      p--;
   }
   printf("\nMax = %hd, Min = %hd\n", max, min);
   return 0;
}
```

PROGRAM 12.8: **max_min.c**

```
Key in some integers and press EOF: 33 99 55 22 66 44 11 0 -5 77
77 -5 0 11 44 66 22 55 99 33
Max = 99, Min = -5
```

PROGRAM OUTPUT: **max_min.c**

```
int *p = 0;
int *p = NULL;
```

C guarantees that no object can have the address 0 or NULL. This constant NULL is defined in many header files including stdio.h. Any pointer which is initialized to 0 or NULL is called a *null pointer*. An uninitialized pointer is not a null pointer because it points "somewhere," and, purely by chance, could have the address of an existing object.

Why do we need a null pointer? One reason is that many functions that return a pointer also need to indicate an error condition when it occurs. These functions can then return NULL to indicate failure. For instance, some of the functions of the standard library (like **malloc** and **strstr**) return NULL if the desired objective isn't achieved. This is like **getchar** returning EOF when it encounters an error.

By default, C doesn't initialize pointers to NULL, so you may have to make this check if necessary. Sometimes, this check is mandatory because dereferencing a null pointer will abort a program. Note that NULL or 0, when used in a pointer context, doesn't need a cast:

```
int *p = NULL;                          Initialize p to NULL
   ...                                  After processing ...
if (p != NULL)                          ... check whether p is still null
   printf("%d\n", *p);
```

Do we use 0 or NULL? It really doesn't matter because the preprocessor converts NULL to 0 anyway. Using NULL instead of 0 is only a "stylistic convention," a gentle way of reminding a programmer that a pointer is being used. However, this is not the case when a pointer is used as argument to a function that takes a variable number of arguments. A value of 0 could be interpreted as the integer 0, so for getting a pointer to int, use (int *) 0 instead of 0. Better still would be to use NULL.

Takeaway: The address 0 or NULL can never be used by an existing object. Also, a pointer can be assigned the value 0 without using a cast. However, a cast is required when passing the pointer value 0 as an argument to a function that takes a variable number of arguments.

12.12 POINTERS, ARRAYS AND FUNCTIONS REVISITED

Pointers, arrays and functions are closely intertwined. This relationship needs to be understood fully to guard against possible pitfalls. Chapter 13 features several examples where array names have been passed as function arguments and array notation used inside the function body. We'll now examine the use of pointer notation and understand whether we can have a mix of both notations.

12.12.1 Pointers in Lieu of Array as Function Argument

Before we consider replacing the array name by a pointer in a function argument, we need to understand the following:

- How the array name is interpreted by the function.
- Why the size of the array is also passed as an argument.
- How the [] in the function parameter is interpreted by the compiler.

Program 12.9 features the **update_array** function which updates each element of an array to twice its initialized value. When the function is invoked, its first argument, arr, is copied as usual to its corresponding parameter, f_arr[], inside the function. The function treats f_arr[] as an array and initializes all of its "elements." But is f_arr actually an array? The operations involving the use of the **sizeof** operator provide the answer.

When **update_array** is called, &arr[0] is copied to f_arr. Even though the function signature shows f_arr[], this does not imply that f_arr is an array. (In fact, it is not.) f_arr is merely a copy of the pointer signified by arr. Because f_arr = arr, f_arr *can be treated as an array.* Thus, updating f_arr with pointer notation actually updates arr.

```
/* array_update_func.c: Uses a mix of array and pointer notation to
                        update and print an array. */
#include <stdio.h>

void update_array(short f_arr[], short f_size);

int main(void)
{
   short arr[] = {11, 22, 33, 44, 55, 66};
   short i, size;

   printf("Size of arr = %d\n", sizeof arr);          /* Bytes used by arr[] */
   size = sizeof arr / sizeof(short);                 /* Number of elements */

   update_array(arr, size);
   for (i = 0; i < size; i++)
      printf("arr[%hd] = %hd ", i, arr[i]);
   printf("\n");

   return 0;
}

void update_array(short f_arr[], short f_size)
{
   short i;
   printf("Size of f_arr = %d\n", sizeof f_arr);      /* Size of pointer! */
   for (i = 0; i < f_size; i++)
      *(f_arr + i) *= 2;                              /* Doubles existing value */
   return;
}
```

PROGRAM 12.9: **array_update_func.c**

```
Size of arr = 12
Size of f_arr = 4
arr[0] = 22 arr[1] = 44 arr[2] = 66 arr[3] = 88 arr[4] = 110 arr[5] = 132
```

PROGRAM OUTPUT: **array_update_func.c**

Observe that **sizeof arr** evaluates differently to **sizeof f_arr**. Because arr is the "real" array, **sizeof arr** signifies the number of bytes used by the array. But f_arr is merely a pointer, so **sizeof f_arr** shows 4, the size of a pointer on this machine. For this reason, the size of the array needs to be specified as an argument to the function.

Why doesn't the compiler treat f_arr as an array in spite of seeing it declared as such in the function prototypes? The answer is that the [] is simply a convenience to make the *programmer* know that f_arr is *meant* to point to an array. The compiler simply interprets it as an alternative form of a pointer declaration. The following argument forms are thus equivalent *only* when they occur in function signatures:

```
void update_array(short *f_arr, short f_size);          Declaration
void update_array(short f_arr[], short f_size) {        Definition
```

The compiler is not confused by the inconsistency of form as long as the types match. However, it would be unwise to take advantage of this equivalence in this manner.

12.12.2 Using const to Protect an Array from a Function

Creating an array and initializing its elements involve overheads that can be significant for large arrays. C eliminates this overhead by copying the pointer and not the array, when the array name is used as an argument. In most cases, this property is in sync with our requirements because we usually access and update a single array. There's no point in copying an array of a thousand elements if all we need is to simply search it. However, this saving also comes at a price: the array is no longer secure.

We pass the value of a variable or its address depending on whether we want to protect the original variable or have it changed by the function. An array doesn't provide this option because by passing its name as argument, we make it *completely* accessible to the function. C has two solutions to this problem. The first one is known to you *(5.13.2)*; use the **const** qualifier to declare the array:

```
const short arr[SIZE] = {10, 10, 10, 10, 10, 10};
```

This array can never be changed—either directly or by using a function. Arrays meant only for lookup should be declared in this way. But if the intention is to protect the array from modification *only* by a function, then use **const** in the function signatures in the following manner:

```
void print_array(const short arr[], short size);          Array notation
void print_array(const short *arr, short size);           Pointer notation
```

print_array can't now change the array pointed to by arr even though the array may be otherwise changeable. Many of our previous programs could be made safer using **const**. For instance, the **merge_arrays** function (Program 11.6) should be declared in the following manner to protect the array, a, from being written by the function:

```
void merge_arrays(const short a[], short m, const short b[], short n, short c[]);
```

You must now develop the habit of using the **const** keyword in function signatures wherever it is relevant. Chapter 13, which deals with string manipulation, uses this keyword extensively. Section 12.18 examines the three possible ways of using the **const** qualifier in the declaration of a pointer.

Tip: If an array is not to be modified in any way, then declare it as constant using the **const** keyword. But if a specific function is to be prevented from changing the array, then use the **const** keyword in the function prototype and header of the definition.

12.12.3 Returning a Pointer to an Array

A function can return a pointer to an array, but where should you create this array? If you create it inside the function, then the array can't be declared in the regular manner (i.e., automatic) because the object would vanish after the function terminates. Program 12.10 offers a simple solution by creating a **static** array in the function body. This array is also visible in **main** because the function returns a pointer to it. Observe the * preceding the function name; **days_in_month** returns a pointer to short.

```
/* return_pointer.c: Creates static array inside function which is visible
                     in main even after termination of function. */
#include <stdio.h>
short *days_in_month(void);
int main(void)
{
   short i, *px;
   px = days_in_month();                                  /* px points to month[0] */
   ++px;                                                  /* px points to month[1] */
   for (i = 1; i < 6; i++)
      printf("Month %hd: %hd   ", i, *px++);
   return 0;
}
short *days_in_month(void)
{
   static short month[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
   return month;
}
```

PROGRAM 12.10: **return_pointer.c**

```
Month 1: 31   Month 2: 28   Month 3: 31   Month 4: 30   Month 5: 31
```

PROGRAM OUTPUT: **return_pointer.c**

A **for** loop prints five elements using *px++. The postfix ++ has a higher precedence than *, so px is incremented and not *px, even though incrementing is done after *px is evaluated. Even if the month array has block scope, by returning a pointer to it, the program has made it visible in **main**. This is why C needs pointers!

12.13 ARRAY OF POINTERS

The programs we used previously to sort an array modified the original array. To preserve the original, you can copy the array and sort the copy. But moving array elements across memory locations often entails heavy overheads. An elegant solution would be to store the pointers to these array elements in a separate array and then rearrange these pointers. We need to use an array of pointers.

Program 12.11 uses a 6-element initialized array, arr, of type short. A second array, p_arr, has the same size but its elements have the type pointer to short. The loop sets the elements of p_arr to point to their corresponding elements of arr (p_arr[n] = &arr[n];). The elements of arr are then printed by simply dereferencing the corresponding elements of p_arr.

12.14 sort_selection2.c: SORTED VIEW OF ARRAY USING ARRAY OF POINTERS

Program 12.12 modifies **sort_selection.c** (Program 10.7), to use an array of pointers, p_arr, to present a reverse-sorted view of the original array, arr. The program uses array notation to present

```
/* array_of_pointers.c: Prints array1 using array2 which contains
                        pointers to elements of array1. */
#include <stdio.h>
#define SIZE 6
int main(void)
{
   short n, arr[] = {11, 33, 55, 77, 99, 22};
   short *p_arr[SIZE];                          /* Array of pointers */
   for (n = 0; n < SIZE; n++) {
      p_arr[n] = &arr[n];
      printf("%hd ", *p_arr[n]);
   }
   return 0;
}
```

PROGRAM 12.11: **array_of_pointers.c**

```
11 33 55 77 99 22
```

PROGRAM OUTPUT: **array_of_pointers.c**

two views of `arr` using `p_arr` and one view that displays the elements directly. The entire exercise is handled by four simple **for** loops and one nested one.

The first loop populates the two arrays (`arr` and `p_arr`), while maintaining a count of the number of items input. The second loop dereferences the pointers to display the elements of `arr`. The nested loop uses the same logical framework used in Program 10.7, except that the current exercise rearranges pointers stored in `p_arr`. Note that the relational operator has changed from > to < because this program reverse-sorts the array. Finally, the last two loops present a sorted view of `arr` using pointers while also establishing that `arr` has not been changed in any way.

We used the expression `*p_arr[i]` to evaluate `arr[i]`, but theory tells us that `p_arr[i]` is the same as `*(p_arr + i)`. After all, `p_arr` represents `&p_arr[0]`, *which itself contains a pointer*. So `*p_arr[i]` is equivalent to `**(p_arr + i)`. Does ** make any sense in C? Replace `*p_arr[i]` with `**(p_arr + i)` in the second **for** loop and satisfy yourself that it does. Section 12.15 discusses the concept of pointing to a pointer.

12.15 POINTER TO A POINTER

As the last paragraph of Section 12.14 would make us believe, C supports a pointer that points to another pointer. This pointer in turn can point to another pointer, and so on. At the simplest level, a pointer to a pointer is declared in this manner:

```
int **q;
```

This pointer currently points nowhere. It can be *completely* dereferenced after (i) pointing `q` to a pointer variable, say, `p`, and (ii) pointing `p` to another object, say, the simple variable `x`. Thus, if `p = &x` and `q = &p`, then the value of `x` can be evaluated using double indirection:

```
**q                                    Same as x and *p
```

```
/* sort_selection2.c: Presents a reverse-sorted view of an array arr
   by sorting an array of pointers that point to elements of arr. */
#include <stdio.h>
int main(void)
{
   short i, j, n = 0, arr[20];
   short *p_arr[20], *p_tmp;
   printf("Key in some integers and press EOF: ");
   for (n = 0; scanf("%hd", &arr[n]) != EOF; n++)
      p_arr[n] = &arr[n];       /* Array of pointers also filled up */
   printf("Contents of arr viewed through p_arr...\n");
   for (i = 0; i < n; i++)
      printf("%hd ", *p_arr[i]);                /* Accessing arr[i] ... */
   for (i = 0; i < n - 1 ; i++)  {
      for (j = i + 1; j < n; j++)
         if (*p_arr[i] < *p_arr[j]) {           /* ... through its pointer */
            p_tmp = p_arr[i];                   /* Pointers rearranged here */
            p_arr[i] = p_arr[j];
            p_arr[j] = p_tmp;
         }
   }
   printf("\nReverse-sorted view of arr after "
          "rearranging elements of p_arr...\n");
   for (i = 0; i < n; i++)
      printf("%hd ", *p_arr[i]);
   printf("\nBut arr still contains...\n");
   for (i = 0; i < n; i++)
      printf("%hd ", arr[i]);
   putchar('\n');                               /* Alternative to printf("\n") */
   return 0;
}
```

PROGRAM 12.12: **sort_selection2.c**

```
Key in some integers and press EOF: 11 88 33 66 55 99 4 77 22 44[Ctrl-d]
Contents of arr viewed through p_arr...
11 88 33 66 55 99 4 77 22 44
Reverse-sorted view of arr after rearranging elements of p_arr...
99 88 77 66 55 44 33 22 11 4
But arr still contains...
11 88 33 66 55 99 4 77 22 44
```

PROGRAM OUTPUT: **sort_selection2.c**

This expression can be interpreted as *(*q), which means using the dereferencing operator twice. You need to use an extra * for every additional level of indirection. Thus, you can create a pointer to q, say, r, and then use ***r or **q or *p to evaluate x.

Program 12.13 features a simple variable x, a pointer variable p and a pointer-to-pointer variable q (which always points to p). This three-part program confirms that x can be accessed and updated using *p and **q.

```
/*pointer2pointer.c: Uses a pointer and pointer to a pointer
                     to access and update the same variable. */
#include <stdio.h>

int main(void)
{
   int x = 100;
   int *p = &x;
   int **q;                                /* A pointer to a pointer */
   q = &p;                                 /* q points to p which points to x */

   printf("Initial value: x = %d\n", x);
   printf("When p points to x,\n");
   printf("value of *p is %d,", *p);
   printf("value of **q is %d\n", **q);

   *p = 200 ;
   printf("\nAfter *p is changed to 200,\n");
   printf("value of **q is %d, ", *(*q));
   printf("value of x is %d\n", x);

   **q = 300;
   printf("\nAfter **q is changed to 300,\n");
   printf("value of x is %d, ", x);
   printf("value of *p is %d\n", *p);

   return 0;
}
```

PROGRAM 12.13: **pointer2pointer.c**

```
Initial value: x = 100
When p points to x,
value of *p is 100,value of **q is 100

After *p is changed to 200,
value of **q is 200, value of x is 200

After **q is changed to 300,
value of x is 300, value of *p is 300
```

PROGRAM OUTPUT: **pointer2pointer.c**

Initially, p points to x and q points to p. The second section changes x using *p but because q still points to p, x can still be evaluated using **q. The final section changes x to 300 by setting **q to 300. The link between x, p and q is maintained at all times provided you handle them correctly.

Why do we need pointer-to-pointers? Can a function that swaps two regular variables be used to swap two pointers? No, it can't because the function will swap the *copies* of these pointers. The solution obviously lies in the use of pointer-to-pointers as function arguments. While these double-indirection pointers are implicitly used in handling 2D arrays, they are also explicitly used in applications that need to manipulate pointer values inside a function.

Caution: The C standard imposes no limits on the maximum number of levels of indirection that you can have. However, if you need to use more than two levels of indirection, there is probably something wrong with the design of your program.

12.16 POINTERS AND TWO-DIMENSIONAL ARRAYS

Now that we know how to handle a pointer to a pointer, let's have a final look at the relationship between arrays and pointers. In this section, we consider a two-dimensional array in a pointer context based on the premise that a 2D array is actually an array of arrays. In Section 10.12.4, we accessed and updated each element of a 2D array using array notation, so let's now use pointer notation for the same task. Consider this 2D array named table:

```
int table[3][5];
```

This array of 3 rows and 5 columns can also be interpreted as an array of 3 elements (table[0], table[1] and table[2]), where each element is itself an array of 5 elements (Fig. 12.5). The 15 elements are laid out contiguously in memory, where the 5 elements of table[0] are followed by 5 elements of table[1] and so on. This implies that element table[0][4] is followed by table[1][0] in memory.

`int table[3][5];`

Sub-Array	*Element 0*	*Element 1*	*Element 2*	*Element 3*	*Element 4*
table[0]	table[0][0]	table[0][1]	table[0][2]	table[0][3]	table[0][4]
table[1]	table[1][0]	table[1][1]	table[1][2]	table[1][3]	table[1][4]
table[2]	table[2][0]	table[2][1]	table[2][2]	table[2][3]	table[2][4]

FIGURE 12.5 Layout of Elements of Two-Dimensional Array

The significance of the name table remains unchanged; it evaluates to the address of its first element. But here the first element is the "array" table[0], which itself points to its first "element", table[0][0]. We can thus represent these two relationships in the following manner:

```
table = &table[0]
table[0] = &table[0][0]
```

We now have two types of objects to deal with, table and table[0], both of which signify addresses and hence can be treated as pointers. table[0] is a pointer to an int, while table is a pointer to

a pointer to int. Thus, the expression *table is equivalent to &table[0][0] while the expression **table evaluates to the element table[0][0].

We'll now derive a general expression that points to the element table[i][j]. Note that table and table[0] have the same address *but of different objects*. Adding 1 to table creates a pointer to table[1], but adding 1 to table[0] creates a pointer to table[0][1]. Thus, in the general case, table[i] + j is equivalent to &table[i][j]. But then table[i] is the same as *(table + i), so &table[i][j] can be represented in pointer notation as

```
*(table + i) + j
```

When we dereference the preceding expression, the resultant expression evaluates to the value of the element table[i][j]:

```
*(*(table + i) +j)
```

Array notation is simpler to use but you should also feel comfortable with pointer notation used in the following program (Program 12.14). This program populates a two-dimensional array with **scanf** using the pointer expression that was just derived. **printf** simply dereferences the same expression to print all array elements.

The annotations provided in the first three lines of output adequately explain the significance of the sizes of three objects determined by **sizeof**. Further, observe that the expressions **(table + 1) and **(table + 2) correctly move the pointer by increments of one row. *The compiler must know the number of columns of an array for such arithmetic to be possible.* Finally, the last loop establishes that table is actually a one-dimensional array. Because the elements of each row are contiguously laid out, the expression p + i could sequentially access all 15 elements.

```
/* pointers_in_2darray.c: Uses scanf and printf with pointer notation
                          to initialize and print a 2D array. */
#define ROWS 3
#define COLUMNS 5
#include <stdio.h>

int main(void)
{
   int i, j;
   int table[ROWS][COLUMNS];
   int *p = &table[0][0];

   printf("size of table = %d\n", sizeof(table));
   printf("size of table[0] = %d\n", sizeof(table[0]));
   printf("size of table[0][0] = %d\n", sizeof(table[0][0]));

   printf("\nKey in %d integers: ", ROWS * COLUMNS);
   for (i = 0; i < ROWS ; i++ )
      for (j = 0; j < COLUMNS ; j++ )
         scanf("%d", *(table + i) + j);
```

```
   printf("Contents of table[0][0] = %3d\n", **table);
   printf("Contents of table[1][0] = %3d\n", **(table + 1));
   printf("Contents of table[2][0] = %3d\n", **(table + 2));

   printf("\nPrinting entire array ...\n");
   for (i = 0; i < ROWS ; putchar('\n'), i++ )
      for (j = 0; j < COLUMNS ; j++ )
         printf("%3d ", *(*(table + i) + j));

   printf("\nBut table is actually a one-dimensional array ...\n");
   for (i = 0; i < ROWS * COLUMNS ; i++ )
      printf("%3d ", *(p + i));

   return 0;
}
```

PROGRAM 12.14: **pointers_in_2darray.c**

```
Size of table = 60                                  Size of entire array
Size of table[0] = 20                               Size of 5 elements
Size of table[0][0] = 4                             Size of 1 element

Key in 15 integers: 1 3 5 7 9 11 33 55 77 99 111 333 555 777 999
Contents of table[0][0] =   1
Contents of table[1][0] =  11
Contents of table[2][0] = 111

Printing entire array ...
  1   3   5   7   9
 11  33  55  77  99
111 333 555 777 999

But table is actually a one-dimensional array ...
  1   3   5   7   9  11  33  55  77  99 111 333 555 777 999
```

PROGRAM OUTPUT: **pointers_in_2darray.c**

12.17 THE GENERIC OR VOID POINTER

If you thought that variables of three data types need three types of pointers to point to them, then you probably are not aware of the *generic pointer*. This special pointer can point to a variable of any type. A generic pointer is declared using the keyword `void`, and has the type `void *`. Consider the following declaration of a generic pointer variable named `void_p`:

```
void *void_p;
```

Even though `void_p` doesn't have a specific data type, it can be made to point to, say, an `int` variable `x`:

```
int x = 5;
void_p = &x;
```

void_p now knows where the data represented by x resides, but it doesn't yet know how to interpret this data. We can't thus dereference void_p to retrieve the value of x. To obtain type information of the data, we need to cast this pointer to int * before using * to fetch the value of x:

```
printf("x = %d\n", *(int *) void_p);                    Prints x = 5
```

Like any other dereferenced pointer expression, this one, *(int *) void_p, can also be used on the left-side of an assignment expression to update the variable pointed to:

```
*(int *) void_p = 100;                                  Changes x to 100
```

We can also unhook void_p from x and make it point to a variable of another data type. Program 12.15 uses a generic pointer to point, in turn, to three variables of type short, int and long. The program also uses the same generic pointer to update the variable it is pointing to.

A generic pointer can be compared to any pointer without using a cast (as in **if void_p == p**) and can also be assigned NULL. This pointer type is very useful for designing functions that need to work with multiple data types of its arguments. The functions that dynamically allocate memory, **malloc** and **calloc**, return a generic pointer to the allocated memory. We'll meet void pointers again in Chapter 16.

```
/* void_pointer.c: Uses a void pointer to access variables of 3 types.
                   Also uses the same pointer to update a variable. */
#include <stdio.h>

int main(void)
{
   short short_x = 10; int int_x = 200; long long_x = 4000L;
   void *void_x;                                  /* A pointer of type void */

   void_x = &short_x;
   printf("Value of short_x = %hd\n", *(short *) void_x);

   void_x = &int_x;
   printf("Value of int_x = %d\n", *(int *) void_x);

   void_x = &long_x;
   printf("Value of long_x = %ld\n", *(long *) void_x);

   *(long *) void_x = 8000L;       /* Updates long_x */
   printf("New value of long_x = %ld\n", long_x);

   return 0;
}
```

PROGRAM 12.15: **void_pointer.c**

```
Value of short_x = 10
Value of int_x = 200
Value of long_x = 4000
New value of long_x = 8000
```

PROGRAM OUTPUT: **void_pointer.c**

Takeaway: A generic or void pointer can point to a variable of any type, but it can't be dereferenced before it is cast to a regular pointer type. A cast is also required when updating the pointed-to variable using the generic pointer.

12.18 USING THE const QUALIFIER WITH POINTERS

We have used the **const** qualifier with regular variables to prevent inadvertent modification of their values. The same qualifier can be used with pointer variables, but with a difference. Because a pointer is associated with two values—the value it stores and the one pointed to, you may like to prevent modification of

- the value pointed to.
- the value stored in the pointer variable.
- both values.

For regular variables, **const** is the first word in the declaration (as in `const int x;`). When the same technique is applied to the pointer, the value pointed to (`x`) is protected from modification:

```
const int *p = &x;
```
`p` *can be changed but not* `x`

However, by changing the location of **const**, you can protect the value of the pointer:

```
int * const p = &x;
```
`x` *can be changed but not* `p`

You can combine the two assignments to protect both the pointer and the value pointed to. This involves using **const** twice:

```
const int * const p = &x;
```
Neither `p` *nor* `x` *can be changed*

Program 12.16 uses three pointers `p`, `q` and `r` to test the three situations by changing a variable that is meant to be a constant. The error messages output by the GCC compiler confirm the assertions just made. You should have no problems in locating the offending lines in the program because the four line numbers indicated in the diagnostic output are also the ones containing the annotations.

The **const** qualifier is often seen in the prototypes of many of the string-handling functions of the standard library, say, the **strcpy** function that copies a string:

```
char *strcpy(char *dest, const char *src);
```

Because the string to be copied doesn't change, it makes sense to use the **const** qualifier for the source string. When designing functions that use pointers as arguments, make sure you use **const** with an argument that the function is not permitted to modify.

```
/* const_in_pointers.c: Three ways of using const in pointer declaration. */
int main(void)
{
   int x, y, z;
   const int *p = &x;
   *p = 10;                          /* Can't change value pointed to */

   int * const q = &x;
   q = &y;                           /* Can't change value of pointer */

   const int * const r = &z;
   r = &y;                           /* Can't change either value of pointer ... */
   *r = 30;                          /* ... or value pointed to */

   return 0;
}
```

PROGRAM 12.16: **const_in_pointers.c**

```
const_in_pointers.c: In function main:
const_in_pointers.c:7: error: assignment of read-only location
const_in_pointers.c:10: error: assignment of read-only variable q
const_in_pointers.c:13: error: assignment of read-only variable r
const_in_pointers.c:14: error: assignment of read-only location
```

PROGRAM OUTPUT: **const_in_pointers.c**

Pointers are very powerful objects, the main reason behind the continuing popularity of C even after 40 years of its creation. We have done as much as we could to demystify them but we are not done with them yet. We have to apply these concepts to strings, structures and input/output operations on disk files. Finally, we need to know how to handle pointers to functions.

WHAT NEXT?

We have avoided a discussion on strings in this chapter even though a string itself is a pointer. The world of strings is vast and unique enough to deserve separate examination. All string-handling functions of the standard library use pointers as arguments, and some of them return pointers as well.

WHAT DID YOU LEARN?

A *pointer* is a derived data type containing a memory address. A pointer can access any legal memory location and retrieve or change the contents at that location.

A pointer is obtained by using the & (address) operator on a variable. The * (indirection) operator *dereferences* a pointer to obtain the value at the address pointed to.

A pointer has a fixed size and its type is directly related to the data type of the object pointed to.

A pointer to a variable used as a function argument enables the function to change the variable defined outside the function. A function can return multiple values using the pointers in its parameter list.

A function can also return a pointer provided the object pointed to exists after the function has completed execution.

The name of an array signifies a *constant pointer* having the address of the first element. The expression `&brr[`*n*`]` is the same as `brr +` *n* and `brr[`*n*`]` is the same as `*(brr +` *n*`)`. Adding 1 to `&brr[`*n*`]` advances the pointer by one *storage unit* to point to the next element.

A pointer can be treated as an array if the pointer is first made to point to an array. When an array name is used as a function argument, the function can choose to treat it as an array or a pointer.

A pointer `x` can point to another pointer `y` to form a *pointer-to-pointer*. The value at the address pointed to by `y` can be evaluated as `*y` or `**x`.

A *null* pointer has the value 0 or `NULL` and is guaranteed not to point to an existing object. A *void* or *generic* pointer can point to an object of any data type but it can be dereferenced only after it has been cast to the specific data type.

The **`const`** qualifier can be used to prevent modification of the pointer, its pointed-to value or both.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

12.1 A pointer need not always represent a memory address.

12.2 Adding 1 to a pointer value doesn't necessarily increment the value by 1.

12.3 The array `arr` and the string `"Hello"` also represent pointer values.

12.4 The size of a pointer is related to the size of the variable pointed to.

12.5 A function can return multiple values using pointers.

12.6 If pointer `p` points to the array `arr`, one can use both `p++` and `arr++` to navigate the array.

12.7 For the array `arr`, the expression `arr + 5` equates to `&arr[5]`.

12.8 If an array is not declared in **`main`** with the **`const`** qualifier, a function cannot be prevented from modifying it.

12.9 The function call **`f(**p, **q)`** can be used to swap two pointers.

A2. FILL IN THE BLANKS

12.1 The _____ operator represents the address of a variable, and the _____ operator retrieves the value stored at the address.

12.2 Any expression that evaluates to an _____ can be used as a pointer.

12.3 A _____ pointer can never point to an existing object.

12.4 A function can swap two pointers only if it is passed _____ to these pointers as arguments.

12.5 A function can return a pointer provided it doesn't point to a _____ variable or object of the function.

12.6 For a pointer to be interpreted as an array, the pointer must point to a _____ section of memory.

12.7 A generic pointer is declared using the keyword _____.

A3. MULTIPLE-CHOICE QUESTIONS

12.1 The * operator is used to (A) declare a pointer, (B) dereference the pointer, (C) multiply two numeric operands, (D) all of these.

12.2 If a pointer is passed as argument to a function, the latter (A) accesses only a copy of the pointer, (B) can use the argument to change the value pointed to, (C) A and B, (D) none of these.

12.3 A function that is passed the name of an array as argument, (A) doesn't know the number of elements of the array, (B) can determine the size of the array easily, (C) copies the entire array, (D) none of these.

12.4 If the variable x and the array `arr` have the same data type, which of the following statements is legal?

```
(i) x = *(arr + 5);
(ii) *(arr + 5) = x;
```

(A) (i), (B) (ii), (C) both, (D) neither.

12.5 If p and q are two pointers of the same type, which of the following operations are permitted?

(i) `p + q`
(ii) `p + 3`
(iii) `p - q` where p and q point to elements of the same array.
(iv) `p * q`

(A) (i) and (ii), (B) (ii) and (iii), (C) (iii), (D) (i) and (iii).

12.6 The statement `const int *p = &x;` (A) is the same as `int * const p = &x;`, (B) protects the pointer from modification, (C) protects the pointed-to value from modification, (D) B and C.

CONCEPT-BASED QUESTIONS

12.1 Name three expressions or objects that evaluate to a pointer.

12.2 If p, which is set to `&x` in **`main`**, is passed as an argument to a function, can the latter change the value of x?

12.3 Explain with a suitable example how a function can return multiple values using arguments and the **`return`** statement. In what circumstances could one still need the **`return`** statement?

12.4 Can we use the following statements?

```
int i = 50; int *p = &i; p[4] = 100;
```

12.5 The following code doesn't always fail. Why?

```
int *p;
printf("%d", *p);
```

12.6 If the pointer p points to the array arr, explain how pointer arithmetic works better with p than with arr.

12.7 How can two pointers be swapped using a function? Why can't the standard technique of using two pointers as arguments work here?

12.8 Explain how the use of **const** differs between the declaration of a variable in **main** and a parameter of a function.

PROGRAMMING & DEBUGGING SKILLS

12.1 Write a program that declares and initializes two variables x and y to 10 and 20, respectively. Use two pointers to sum these values and print the result.

12.2 Write a program that declares three pointers of type short *, int * and float * pointing to regular variables. Prove that the size of the pointer has no relation to the size of the variable pointed to.

12.3 Use pointer arithmetic to prove that the size of the char data type is 1 byte.

12.4 Use a pointer to print the lowercase letters of the English alphabet.

12.5 Write a program that declares an int array of 10 elements. Using only pointer notation, populate the array with **scanf** and print the elements with **printf**.

12.6 Write a program where an int pointer p points to a variable x which is initialized to 10. Using only the *, ++ and -- operators with p, (i) add 1 to the address of x, (ii) subtract 1 from x, (iii) add 1 to x.

12.7 Write a program that uses **scanf** to accept a word as string input. Using a pointer to a character array and without using array notation, compute the length of the string, i.e. until the NUL character is encountered.

12.8 Write a program to compute the sum of all elements of an int array using only pointers.

12.9 Correct the following program which issues a warning on compilation but fails at runtime. Does the **create_array** function return an array?

```
#include <stdio.h>
short * create_array(void);
int main(void)
{
   short i;
   short *brr;
   brr = create_array();
```

```
    for (i = 0; i < 6; i++)
       printf("%hd ", brr[i]);
    return 0;
}
short * create_array(void)
{
    short arr[] = {1, 3, 5, 7, 99, 88};
    return arr;
}
```

12.10 Create a function named **`download_time`** that accepts the download speed in Mbps and file size in MB. The function must return a pointer to a 3-element array containing the downloaded time in hours, minutes and seconds. Without returning a pointer, how can the same values be made available to the caller?

12.11 Write a program that uses the **`interchange`** function to swap the contents of two arrays using only pointers. However, the program may use array notation for initializing the two arrays. The size of the two arrays must be made available as a global variable. What is the major drawback of this program?

12.12 Write a program that initializes a 2D array, `arr[3][5]`, and uses a pointer to this array in a *single* loop to print all elements. What does the output indicate about the organization of a 2D array in memory?

12.13 Write a program that defines two `int` pointer variables `p` and `q` that point to variables `x` and `y`, respectively. Devise a function named **`swap_pointer`** that swaps the pointers so that `p` and `q` point to `y` and `x`, respectively.

13 Strings

WHAT TO LEARN

- Interpreting a string as an array and a sequence of double-quoted characters.
- Handling a line of text as a string with **`fgets`** and **`fputs`**.
- Reading and writing a string with **`sscanf`** and **`sprintf`**.
- Manipulating strings using their pointers.
- Rudimentary implementation of some string-handling functions of the standard library.
- How to use the string- and character-oriented functions of the library.
- Handling multiple strings using a two-dimensional array.
- Sorting a line of characters and an array of strings.
- How to conserve memory using an array of pointers to strings.
- Tweaking the **`main`** function for invoking a program with arguments.

13.1 STRING BASICS

Programming isn't only about crunching numbers. It also involves extensive string handling. Unlike Java, C doesn't support a primary data type for a string. The language approaches strings through the "backdoor"—by tweaking the `char` data type. A string belongs to the category of derived data types and is interpreted as an array of type `char` terminated by the NUL character. C offers no keywords or operators for manipulating strings but its standard library supports a limited set of string-handling functions.

Even though a string is a collection of characters, they are interpreted differently. A character constant is a single-byte integer whose equivalent symbol is enclosed in *single* quotes. Thus, `'A'` and its ASCII value, `65`, can be used interchangeably in programs. But a string constant is a set of one or more characters enclosed in *double* quotes. Thus, `"A"` signifies a string. `'A'` uses one byte but `"A"` needs two bytes which include the NUL character. Functions that operate on characters don't work with strings and vice versa.

The pointer introduces an additional dimension to a string. The string constant `"Micromax"` represents a pointer to the address of its first character, i.e., `'M'`. This string can thus be assigned to a pointer variable of type `char *`. Since `"Micromax"` can also be interpreted as an array of characters, the name of the array storing the string also represents the address of `'M'`. A string can be manipulated with both array and pointer notation even though they are not *entirely* equivalent.

Because strings don't have fixed sizes, every string-handling function must know where a string begins and where it ends. A function knows the beginning of a string from the pointer associated with it. The end is signified by the NUL character. Functions of the standard library that create or modify strings make sure that the NUL is always in place. However, when you create a string-handling function, it's your job to append the NUL at the end of the string. NUL has the decimal value 0 and is represented by the escape sequence `'\0'`.

Note: Because a string must contain NUL, even the empty string `""` takes up one byte. The length of the string is zero though.

13.2 DECLARING AND INITIALIZING A STRING

A string can be declared either as an array of type `char` or as a pointer of type `char *`. The two techniques of declaration are not fully equivalent. You must understand how they differ and when to use one technique in preference to the other.

13.2.1 Using an Array to Declare a String

We have previously declared and initialized an array with a set of values enclosed in curly braces. The same technique applies to strings as well. The following declaration (and definition) statements also include initialization with a set of single-quoted characters:

```
char stg1[20] = {'F', 'a', 'c', 'e', 'b', 'o', 'o', 'k', '\0'};
char stg2[] = {'T', 'w', 'i', 't', 't', 'e', 'r', '\0'};
```

The last element in each case is NUL and has to be explicitly inserted when you initialize a string like this. The first declaration wastes space because the string uses only nine elements. But the compiler automatically sets the size of `stg2` by counting eight items in the list.

Keying in single-quoted characters for initialization is a tedious job, so C also offers a convenient alternative. You can use a double-quoted string to assign an array at the time of declaration:

```
char stg3[] = "Whatsapp";                    Array has 9 elements and not 8
```

The NUL is automatically inserted when an array is assigned in this way. So, even though **`sizeof stg1`** evaluates to 20 (the declared size), **`sizeof stg2`** and **`sizeof stg3`** evaluate to 8 and 9, respectively.

13.2.2 When an Array is Declared But Not Initialized

It is possible to declare an array—meant for holding a string—without initializing it. In that case, declare the array in the same way you declare a variable:

```
char stg4[30];                               Also defines stg4
```

Once an array has been declared in this manner, you can't assign values to it by using either of the methods shown previously. This means that the following assignments are invalid:

```
stg4 = {'X', 'o', 'l', 'o', '\0'};          Wrong!
stg4 = "Xolo";                              Wrong!
```

You must now assign the elements individually, but don't forget to include the NUL:

```
stg4[0] = 'X'; stg4[1] = 'o'; stg4[2] = 'l'; stg4[3] = 'o'; stg4[4] = '\0';
```

Most of the space allocated to `stg4` are not assigned specific values. But make no mistake: *this is not a partially initialized array because values were not assigned at the time of declaration*.

Of the four arrays considered, it's only for `stg3` that NUL was automatically inserted by the compiler. We need to know why the compiler would have done the same for `stg1` even if it was initialized without including NUL.

13.2.3 Using a Pointer to Declare a String

C supports another way of creating a string. Declare a pointer to `char` and then assign a double-quoted string to the pointer variable:

```
char *p;                                    p currently pointing "nowhere"
p = "Redmi Note";                           p contains address of 'R'
```

This string comprises multiple words which **`scanf`** reads with two `%s` specifiers, but **`printf`** needs a single `%s` to print. Like with a regular variable, you can combine declaration and initialization in one statement:

```
char *p = "Redmi Note";                     NUL automatically appended
```

The previous declarations that used an array allocated memory only for the array. But the preceding one creates space in memory for two objects—for the pointer `p` and for the string constant, `"Redmi Note"`. It then makes `p` point to the first character of the string (Figure 13.1).

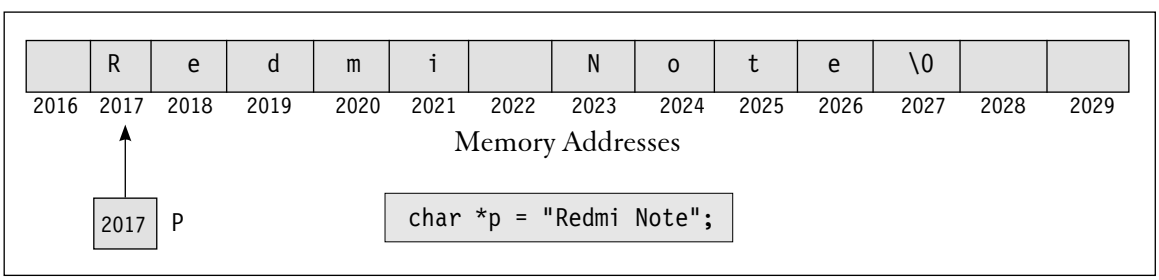


FIGURE 13.1 Memory Allocation For the Pointer and the String Pointed To

Unlike the array `stg4`, which signifies a *constant pointer*, `p` is a *pointer constant*. That doesn't mean `p` is a constant but that the object it points to is a constant. At any time, the pointer `p` can be unhooked from the current string and made to point somewhere else, which could be another string or an array:

```
p = "Samsung Galaxy";                    p now contains address of 'S'
p = stg4;                                p points to the string "Xolo"
```

Note that arrays have fixed addresses which can't be changed even though their contents can be. For instance, you can reassign any element, say, `stg4[3]` to `'C'`. But you can't change the value `stg4` evaluates to.

13.2.4 When an Array of Characters Is Not a String

An array of characters is not a string when it doesn't include the `'\0'` character as the terminator. Usually, we don't bother to include the NUL because the compiler does this job for us, but negligence on our part can prevent the compiler from doing so. Consider the following declarations:

```
char stg5[5] = {'f', 'l', 'o', 'a', 't'};        This isn't a string ...
char stg6[30] = {'f', 'l', 'o', 'a', 't'};       ... but this one is.
```

In this scenario, `stg5` has no space left to store the NUL character. When **`printf`** encounters this string, it continues printing beyond the `'t'` until it encounters a NUL somewhere on the way. This means looking at a region of memory not reserved for it. You may see extra characters printed at the end of the word `float` or the program could even crash.

The NUL character has not been *explicitly* inserted in `stg6`, so one would be tempted to conclude that `stg6` is not a string. However, `stg6` is a partially initialized array, which by definition, has the uninitialized elements set to NUL. Hence, `stg6` is a string.

Takeaway: The compiler automatically inserts NUL when (i) a double-quoted set of characters is used as the value, or (ii) the array is partially initialized. When assigning an array, there must be room for the NUL to be included.

13.3 `intro2strings.c`: DECLARING AND INITIALIZING STRINGS

Program 13.1 demonstrates the techniques of assigning string values to array and pointer variables. It also shows how **`sizeof`** computes the size of an array containing a string. Finally, the program runs **`scanf`** in a loop to repeatedly assign an array. One of the arrays is not a string, and the output shows the consequences of not using the NUL character when it should have been used.

The program declares and initializes four arrays in four different ways. It also assigns one of the arrays to a pointer. Three of the arrays (`stg1`, `stg2` and `stg4`) are valid strings because they are terminated—explicitly or implicitly—by NUL. However, `stg3` is not a valid string. Observe that **`sizeof`** doesn't return the string length but the number of bytes used by the array in memory.

Printing of `stg3` has created problems on this machine. **`printf`** failed to stop after printing `char`, but intruded into the territory used by `stg2` to print `double` as well. You may not get the same output, but the output would still be unpredictable because there's no guarantee that **`printf`** will find NUL at the end of the character sequence.

Finally, **`scanf`** runs in a loop to read multiple words and assign each word in turn to `stg1`. To read a line as a *single* string, **`scanf`** needs to use the *scan set (9.11.1)*, but the **`fgets`** function *(13.4.3)* also does the job. We'll use both techniques in this chapter.

```
/* intro2strings.c: Declares and initializes strings.
                    Shows consequence of not inserting NUL in stg3. */
#include <stdio.h>
int main(void)
{
   char stg1[30] = {'f', 'l', 'o', 'a', 't'};
   char stg2[] = {'d', 'o', 'u', 'b', 'l', 'e', '\0'};
   char stg3[4] = {'c', 'h', 'a', 'r'};
   char stg4[] = "long double";
   char *p = stg4;

   printf("Size of stg1 = %d  ", sizeof stg1);
   printf("Size of stg2 = %d  ", sizeof stg2);
   printf("Size of stg3 = %d  ", sizeof stg3);
   printf("Size of stg4 = %d\n", sizeof stg4);

   printf("%s\n", stg1); printf("%s\n", stg2);
   printf("%s\n", stg3); printf("%s\n", stg4);
   printf("%s\n\n", p);

   printf("Enter a string ([Ctrl-d] to terminate): ");
   while (scanf("%s", stg1) == 1)
      printf("Each word: %s\n", stg1);

   return 0;
}
```

PROGRAM 13.1: **intro2strings.c**

```
Size of stg1 = 30  Size of stg2 = 7  Size of stg3 = 4  Size of stg4 = 12
float
double
chardouble                                  printf accesses space of previous array
long double
long double                                 Dereferenced value of pointer p

Enter a string ([Ctrl-d] to terminate): Lenovo K6 Power
Each word: Lenovo
Each word: K6
Each word: Power
[Ctrl-d]
```

PROGRAM OUTPUT: **intro2strings.c**

Note: It's only when accessing a string that both **printf** and **scanf** use the same identifier name (the name of the array). **scanf** doesn't use the & prefix for this purpose.

13.4 HANDLING LINES AS STRINGS

We often deal with *text* files which are organized as a group of lines containing printable characters. Each line is separated from its adjacent one by the appropriate newline character specific to the operating system (CR-LF for MSDOS, LF for UNIX). We routinely sort lines, display those that contain a specific string and also modify them. C supports the following I/O functions that read and write lines:

- **gets/puts** (Discussed here but unsafe to use)
- **fgets/fputs**

These functions use a *buffer* (an array) as an argument to store a line of data. The two sets differ in handling of NUL and the newline character. We'll first take up the **gets/puts** duo, understand the danger involved in using **gets** and then discuss the safer functions, **fgets** and **fputs**. We'll also understand how the **fgets/sscanf** combination is often a better alternative to **scanf** for formatting input data.

13.4.1 Using gets/puts: The Unsafe Way

Prototype: `char *gets(char *s);`
Header file: `stdio.h`
Return value: (i) *s* on success, (ii) NULL on error or when EOF occurs before any characters are read.

Prototype: `int puts(const char *s);`
Header file: `stdio.h`
Return value: A non-negative number on success or EOF on error.

Both functions use a single argument, a pointer to `char`. For **gets**, this must be an implicit pointer to a buffer represented by an array of type `char`. **puts** simply writes the contents of the buffer to the terminal:

```
char stg[80];                    Can hold up to 80 characters including NUL
gets(stg);                       Reads a line from standard input into stg
puts(stg);                       Writes contents of stg to standard output
```

gets converts the trailing newline to NUL, while **puts** adds it back before writing to the terminal. `stg` must be large enough to hold an entire line of input even though **gets** doesn't check this size. Thus, if the line is longer than the size of `stg`, **gets** will simply overflow this buffer.

Note: The infamous Morris worm, which paralyzed (possibly) one-tenth of the Internet in 1988, originated from an overflow situation caused by **gets** in a self-replicating and self-propagating program.

13.4.2 gets_puts.c: A Program Using gets and puts

Program 13.2 changes the case of text from lower to upper in a line of text read by **gets**. **puts** writes the modified line to the standard output. A subsequent program will use the **toupper** library

```
/* gets_puts.c: Gets a line from standard input, changes lowercase
                to uppercase before writing line to standard output. */
#include <stdio.h>
#define SIZE 80

int main(void)
{
   char line[SIZE], diff = 'a' - 'A';
   int i;

   printf("Enter a line of text: \n");
   gets(line);                                  /* Replaces '\n' with '\0' */

   for (i = 0; line[i] != '\0'; i++)
      if (line[i] >= 'a' && line[i] <= 'z')
         line[i] -= diff;

   puts(line);                                  /* Replaces '\0' with '\n' */

   return 0;
}
```

PROGRAM 13.2: **gets_puts.c**

```
the gets function is dangerous and should not be used.
THE GETS FUNCTION IS DANGEROUS AND SHOULD NOT BE USED.
```

PROGRAM OUTPUT: **gets_puts.c**

function for doing the same job. However, the message put out here must be taken seriously: use **fgets** instead of **gets**. After this program, we won't use **gets** in this book.

Caution: Never use the **gets** function! If the data can't fit in the buffer, **gets** will write past its end and change other parts of the program. The **fgets** function eliminates this possibility because it takes the size of the buffer as a separate argument. C11 has removed **gets** from its specification.

13.4.3 Using fgets/fputs: The Safe Way

Prototype: `char *fgets(char *s, int size, FILE *stream);`
Header file: `stdio.h`
Return value: (i) *s* on success, (ii) NULL on error or when EOF occurs before any characters are read.

Prototype: `int fputs(const char *s, FILE *stream);`
Header file: `stdio.h`
Return value: A non-negative number on success, EOF on error.

Unlike **gets** and **puts**, **fgets** and **fputs** work with any file type, including disk files. The file type is specified as the last argument for both functions. For I/O operations with the terminal, this argument is `stdin` and `stdout` for **fgets** and **fputs**, respectively. **fgets** overcomes the buffer overflow problem by using the size of the buffer as the second argument:

```
char stg[80];
fgets(stg, 80, stdin);                    Reads up to 79 characters into stg
fputs(stg, stdout);
```

fgets reads a maximum of *size* - 1 characters from the standard input *stream* to the buffer *s*. Unlike **gets**, **fgets** reads the newline into the buffer and adds `'\0'` after the last character. **fputs** writes the contents of *s* to the standard output *stream*, but without the trailing `'\0'`.

13.4.4 fgets_fputs.c: A Program Using fgets and fputs

Program 13.3 modifies and enhances the previous program to reverse the case of characters using three library functions that need the file `ctype.h` to be included. The **islower** function returns true if its argument is a lowercase letter. The **toupper** and **tolower** functions convert the case of a letter. They either return the converted character or the existing character if conversion is not possible. This program prepares us for using character-oriented functions for string manipulation tasks. These functions are listed and discussed in Section 13.10.

```
/* fgets_fputs.c: Gets a line of input from standard input and reverses
                  the case before writing line to standard output. */
#include <stdio.h>
#include <ctype.h>                    /* For islower, tolower and toupper */
#define SIZE 80

int main(void)
{
   char line1[SIZE], line2[SIZE];
   short i;

   printf("Enter a line of text: ");
   fgets(line1, SIZE, stdin);                  /* Doesn't remove '\n' */

   for (i = 0; line1[i] != '\0'; i++)
      if (islower(line1[i]))
         line2[i] = toupper(line1[i]);
      else
         line2[i] = tolower(line1[i]);

   line2[i] = '\0';                            /* '\0' was not copied */
   fputs(line2, stdout);                       /* Doesn't add '\n' */

   return 0;
}
```

PROGRAM 13.3: **fgets_fputs.c**

```
Enter a line of text: fgets and sscanf form a useful combination.
FGETS AND SSCANF FORM A USEFUL COMBINATION.

Enter a line of text: FPUTS AND SPRINTF ARE OFTEN USED TOGETHER.
fputs and sprintf are often used together.
```

PROGRAM OUTPUT: **fgets_fputs.c**

13.5 THE sscanf AND sprintf FUNCTIONS

Sometimes, you would like to save keyboard input in a string before using a **scanf**-type function to read from the string. Because the string itself doesn't change, you can use multiple formatting techniques on this data. **scanf** can't do this job because it changes the buffer in every invocation, but the **sscanf** function meets our requirement.

In the same vein, you may like to use **printf**-like functionality to format a line of data and save it in a string. This gives you the chance to modify the string before or after printing. The **sprintf** function does whatever **printf** does except that it saves the formatted output in a string. Once that is done, you can use **fputs** to print the string.

13.5.1 sscanf: Formatted Input from a String

Consider a situation where you have the option of inputting a date in multiple formats (say, *dd-mm-yy*, *month_name-dd-yy* or *dd-month_name-yy*). You can't catch them all by repeatedly invoking **scanf** with different format specifiers because each invocation will fetch data from the buffer and overwrite the previous data. You need to use **sscanf** which has the following prototype:

```
int sscanf(char *str, cstring, &var1, &var2, ...);
```

Unlike **scanf** which reads standard input, **sscanf** reads the string *str* and uses the information provided in *cstring* (the control string) to format the data before saving them in variables. The function is often used in combination with **fgets**. To validate a date, read the input into a buffer with **fgets** first:

```
fgets(line, SIZE, stdin);
```

You can now use **sscanf** to match the input saved in `line` with the form *dd-mm-yyyy*:

```
sscanf(line, "%2d-%2d-%4d", &day, &month, &year);
```

If the match doesn't succeed, you can call **sscanf** multiple times, with a different format specifier in each invocation. The following code snippet accepts `12-05-16`, `May-12-16` and `12-May-16` as valid date formats:

```
if (sscanf(line, "%2hd-%2hd-%2hd", &day, &month, &year) == 3)
   date_ok = 1;
else if (sscanf(line, "%3s-%hd-%hd", month_arr, &day, &year) == 3)
   date_ok = 1;
```

```
else if (sscanf(line, "%hd-%3s-%hd", &day, month_arr, &year) == 3)
   date_ok = 1;
else
   date_ok = 0;
```

You couldn't have done this by repeatedly calling **scanf** because each call would change the input buffer. Here, `line` remains unchanged throughout the checking process. After discussing **sprintf**, we'll use both **sscanf** and **sprintf** in a program.

Tip: When a data item can occur in multiple formats, use **fgets** to read the input into a buffer. You can then use **sscanf** multiple times and with different format specifiers to match the data in the buffer.

13.5.2 sprintf: Formatted Output to a String

The dates that were matched with **sscanf** in the previous section can now be saved (along with additional information) in a string. This is possible with the **sprintf** function which uses an additional argument to specify the string:

```
int sprintf(char *str, cstring, var1, var2, ...);
```

sprintf uses the format specifiers in *cstring* to format the data represented by the variables. It then writes the formatted data to the string *str* instead of the display. Depending on whether you want the date `19/05/53` to be represented as `19-05-1953` or `May 19, 1953`, you can populate the array, `line`, in one of these two ways:

```
sprintf(line, "dd-mm-yyyy format: %02d-%02d-%04d\n", day, month, year);
sprintf(line, "month dd, yyyy format: %s %02d, %04d\n", month_arr, day, year);
```

sprintf appends NUL to `line`. You can save this line in a file using **fputs** but you can also print it using the same function:

```
fputs(line, stdout);
```

sprintf also returns the number of characters written to the string (including `'\0'`). You can use this return value to validate the length of the string. This feature is utilized in the next program.

13.5.3 validate_pan.c: Using sscanf and sprintf to Validate Data

Program 13.4 validates a code (PAN, the Permanent Account Number) keyed in by the user. The 10-character PAN code has the form *XXXXX9999X*, where *X* represents a letter. The program runs in a loop which is exited only after the user has keyed in a valid PAN. This program totally avoids the use of the **scanf**/**printf** heavyweights by dividing the work among four functions.

sscanf uses three scan sets to specify the size and type of the three components of the code. The side effect of **sprintf** writes the validated input to a string while its return value is used to check whether the code comprises 10 characters. The entire work is divided among two sets of functions. The logic is handled by **sscanf** and **sprintf**, and I/O operations are carried out using **fgets** and **fputs**.

```
/* validate_pan.c: Accepts and validates the user's Permanent Account Number.
                   PAN has the form XXXXX9999X (10 characters). */
#include <stdio.h>
#include <string.h>
#define SIZE 30

int main(void)
{
   char line1[SIZE], line2[SIZE];
   char stg1[6], stg2[5], stg3[2];
   short pan_length;

   while (fputs("Enter your PAN identifier: ", stdout)) {
      fgets(line1, SIZE, stdin);
      if (sscanf(line1, "%5[a-zA-Z]%4[0-9]%[a-zA-Z]", stg1, stg2, stg3) != 3)
         fputs("Invalid PAN\n", stdout);
      else {
         pan_length = sprintf(line2, "%s%s%s", stg1, stg2, stg3);
         if (pan_length != 10)
            fputs("PAN must be 10 characters.\n", stdout);
         else
            break;
      }
   }
   fputs(line2, stdout);

   return 0;
}
```

PROGRAM 13.4: **validate_pan.c**

```
Enter your PAN identifier: ADOYTT4598C
Invalid PAN
Enter your PAN identifier: AXEGB1097EE
PAN must be 10 characters.
Enter your PAN identifier: AXEGB1097E
AXEGB1097E                                                    Valid entry
```

PROGRAM OUTPUT: **validate_pan.c**

13.6 USING POINTERS FOR STRING MANIPULATION

String manipulation is an important programming activity, so we must fully understand the mechanism of navigating a string and accessing each character. Because a string can be treated as an array and a pointer, the previous discussions *(12.8.2)* related to the use of pointer and array notation in pointer arithmetic also apply here. Consider the following strings defined in two different ways:

```
char stg[] = "Nyasaland";
char *p = "Malawi";
```

Because stg is a constant, we can use *limited* pointer arithmetic (like stg + 1) to access and update each element, but we cannot use stg++ and stg--. The pointer p, being a variable, knows no such restrictions. In addition to using expressions like p + 1, we can also use p++ and p-- to change the current pointer position in the string.

There is one more difference between stg and p. While you can change the contents of stg ("Nyasaland"), you can't change the dereferenced value of p ("Malawi") because p is a pointer constant. p points to a region of memory that is marked read-only. However, if p is made to point to stg, then you can change any character in stg using p. Table 13.1 compares the validity of operations made on the two representations of a string.

TABLE 13.1 Permissible Operations on Pointer that Represents a String

```
char stg[] = "Facebook"; char *p = "Twitter";
```

Statement/Expression	*OK/Wrong*	*Statement/Expression*	*OK/Wrong*
`stg = "Skype";`	Wrong	`p = "Skype";`	OK
`stg + 1`	OK	`p + 1`	OK
`stg++;`	Wrong	`p++;`	OK
`stg[0] = 'C';`	OK	`p[0] = 'C';`	Wrong
`stg = p;`	Wrong	`p = stg;`	OK

13.7 COMMON STRING-HANDLING PROGRAMS

We'll now discuss three programs often encountered by beginning C programmers. Except for the use of **strlen**, none of the programs uses a function of the standard library for handling strings or characters. We'll later reinvent the wheel by developing our own code for some library functions before we eventually examine the important string-handling functions of the standard library.

A word about **scanf** and **fgets** could be helpful before we proceed. Most programs in this chapter handle a string input from the keyboard. In these programs, we have used either **fgets** or **scanf** (with the [^\n] scan set) to read a line of input and store it as a NUL-terminated string. Because **fgets** also saves the '\n', it may sometimes be necessary to overwrite it with NUL. However, in all cases, **scanf** with the scan set works without problems.

13.7.1 string_palindrome.c: Program to Check a Palindrome

Program 13.5 checks a string for a *palindrome*, which is a string of characters that read the same whether read forward or backward. The string to be checked is saved by **scanf** in the array, stg. The variable j represents the index of the last non-NUL character in stg. The **while** loop begins the equality check from the two ends of the string and completely scans the string. Ideally, the scan could have stopped midway and all elements would still have been compared.

```
/* string_palindrome.c: Ascertains whether a string is a palindrome */
#include<stdio.h>
#include <string.h>
```

```
int main(void)
{
   short i = 0, j;
   char stg[80];

   printf("Enter a string: ");
   scanf("%[^\n]", stg);              /* Only '\0' at end of stg, not '\n' */
   j = strlen(stg) - 1;               /* strlen is a library function */

   while (j) {
      if (stg[i++] != stg[j--]) {
         printf("%s is not a palindrome\n", stg);
         return 1;
      }
   }
   printf("%s is a palindrome\n", stg);

   return 0;
}
```

PROGRAM 13.5: **string_palindrome.c**

```
Enter a string: madam
madam is a palindrome
```

```
Enter a string: gentleman
gentleman is not a palindrome
```

PROGRAM OUTPUT: **string_palindrome.c**

13.7.2 count_words.c: Counting Number of Words in a String

Program 13.6 counts the number of words in a string, where a word is defined as a group of characters not containing whitespace (i.e., without spaces, tabs or newlines). The clever use of the variable flag ensures that multiple contiguous whitespace characters are treated as a single delimiter.

```
/* count_words.c: Counts number of words separated by any number
                  of spaces, tabs and newlines. */
#include <stdio.h>
#define SIZE 80

int main(void)
{
   char stg[SIZE];
   short i = 0, flag = 1, count = 0;

   fputs("Enter a string: ", stdout);
   fgets(stg, SIZE, stdin);
```

```
   while (stg[i] != '\0') {
      if (stg[i] == ' ' || stg[i] == '\t' || stg[i] == '\n')
         flag = 1;
      else if (flag == 1) {         /* If this character is not whitespace ... */
         flag = 0;                  /* ... increment word count when ... */
         count++;                   /* ... beginning of word detected. */
      }
      i++;                          /* Take up next character in string */
   }
   printf("Number of words = %hd\n", count);

   return 0;
}
```

PROGRAM 13.6: **count_words.c**

```
Enter a string: chad mauritania    niger    upper_volta mali
Number of words = 5
```

PROGRAM OUTPUT: **count_words.c**

13.7.3 sort_characters.c: Sorting Characters in a String

The selection sort algorithm was developed in Section 10.8 for ordering a set of integers. Program 13.7 applies the same algorithm on a set of one-byte characters (which are actually integers). The program output in both invocations includes all alphabetic letters preceded by spaces which have a lower ASCII value than the letters. To understand the working of this program, it could be helpful to recall the central features of the sorting algorithm.

Selection sort is based on scanning an array of n elements in multiple passes. After completion of each pass, the start element has the lowest number among the elements encountered in that pass. The start point progressively moves inward until n - 1 passes have been made. In this program, the outer **for** loop progressively changes the start point, while the inner loop compares the elements to set the start element to the lowest value encountered in that pass.

```
/* sort_characters.c: Sorts characters of a string using selection sort */
#include <stdio.h>
#include <string.h>

int main(void)
{
   short i, j, length;
   char stg[100], tmp;

   printf("Enter string: ");
   scanf("%[^\n]", stg);
   length = strlen(stg);
```

```
   for (i = 0; i < length - 1; i++)  {       /* Selection sort logic ... */
      for (j = i + 1; j < length; j++)       /* ... also used in Program 10.7*/
         if (stg[i] > stg[j]) {
            tmp = stg[i];
            stg[i] = stg[j];
            stg[j] = tmp;
         }
   }
   printf("%s\n", stg);

   return 0;
}
```

PROGRAM 13.7: **sort_characters.c**

```
Enter string: the quick brown fox jumps over the lazy dog
        abcdeeefghhijklmnoooopqrrsttuuvwxyz
```

```
Enter string: pack my box with five dozen liquor jugs
       abcdeefghiiijklmnooopqrstuuvwxyz
```

PROGRAM OUTPUT: **sort_characters.c**

13.8 DEVELOPING STRING-HANDLING FUNCTIONS

C handles strings using a small set of functions supported by the standard library. In the following sections, we'll develop five string-handling functions from scratch. We'll integrate four of them into a main program from where they will be invoked. None of these functions specifies the size of the string as an additional argument simply because each function checks for NUL. We have also protected the source string from being modified by using the **const** qualifier.

13.8.1 my_strlen: Function to Evaluate Length of a String

In two previous programs, we used **strlen**, the function supported by the standard library for evaluating the length of a string (without NUL). The following function definition of **my_strlen** implements the functionality of **strlen**. **my_strlen** returns an integer of type short, displays array notation in the parameter list but uses pointer notation in the body without causing conflict.

```
short my_strlen(const char arr[])
{
   short count = 0;
   while (*(arr + count) != '\0')
      count++;
   return count;
}
```

13.8.2 reverse_string: Function to Reverse a String

This function, which doesn't have a peer in the standard library, reverses a string. After reversal, the '\0' is appended to the string. The function returns a pointer to the destination string, but it is also available in the first parameter. You can thus invoke the function either directly:

```
reverse string(s2, s1);
printf("Reversed string is %s\n", s2);
```

or by using it as an expression:

```
printf("Reversed string is %s\n", reverse_string(s2, s1);
```

Most string-handling functions of the standard library that use two strings as arguments are designed to return a value in these two ways.

```
char *reverse_string(char *destination, const char *source)
{
   short i = 0, j;
   j = my_strlen(source) - 1;

   while (j >= 0)
      destination[i++] = source[j--];
   destination[i] = '\0';
   return destination;
}
```

13.8.3 my_strcpy: Function to Copy a String

The **my_strcpy** function, which copies a string, emulates the **strcpy** function of the standard library. This function also uses two arguments, and like **reverse_string**, returns the destination string in two ways.

```
char *my_strcpy(char *destination, const char *source)
{
   short i = 0, j;
   j = my_strlen(source) - 1;
   while (j-- >= 0) {
      destination[i] = source[i];
      i++;
   }
   destination[i] = '\0';
   return destination;
}
```

13.8.4 my_strcat: Function to Concatenate Two Strings

The standard library supports the **strcat** function for concatenating two strings, i.e. appending one string to another. The one that we have developed here (**my_strcat**) does the same work.

Because *destination* initially contains '\0' at the end, *source* is appended at that point by overwriting the NUL. A separate NUL is later added at the end of the concatenated string.

```
char *my_strcat(char *destination, const char *source)
{
   short i = 0, j;
   j = my_strlen(destination);
   for (i = 0; source[i] != '\0'; i++)
      destination[j++] = source[i];
   destination[j] = '\0';
   return destination;
}
```

13.8.5 `string_manipulation.c`: Invoking All Four Functions from `main`

Now that we have defined the four functions, we need to declare them in a central program (Program 13.8). The function definitions just discussed must be appended to the main code segment to form a complete standalone program. The program takes two strings as user input before invoking the four functions. Note that each call of **`fgets`** must be followed by an assignment operation that replaces '\n' with NUL.

```
/* string_manipulation.c: Develops four string-handling functions
   from scratch. "my_" functions also available in the standard library. */

#include <stdio.h>

short my_strlen(const char arr[]);
char *reverse_string(char *destination, const char *source);
char *my_strcpy(char *destination, const char *source);
char *my_strcat(char *destination, const char *source);

int main(void)
{
   char stg1[80], stg2[80], stg3[80], stg4[80];

   printf("Enter main string: ");
   fgets(stg1, 80, stdin);
   stg1[my_strlen(stg1) - 1] = '\0';      /* Replaces '\n' with '\0' */

   printf("Length of string = %hd\n", my_strlen(stg1));

   reverse_string(stg2, stg1);
   printf("Reversed string: %s\n", stg2);

   printf("Copied string: %s\n", my_strcpy(stg3, stg1));

   printf("Enter second string: ");
   fgets(stg4, 80, stdin);
   stg4[my_strlen(stg4) - 1] = '\0';      /* Replaces '\n' with '\0' */
```

```
   my_strcat(stg4, stg1);
   printf("Concatenated string: %s\n", stg4);

   return 0;
}
   ... Function definitions to be appended here ...
```

PROGRAM 13.8: **string_manipulation.c**

```
Enter main string: Botswana
Length of string = 8
Reversed string: anawstoB
Copied string: Botswana
Enter second string: Bechuanaland/
Concatenated string: Bechuanaland/Botswana
```

PROGRAM OUTPUT: **string_manipulation.c**

13.8.6 substr.c: Program Using a Function to Extract a Substring

Unlike most languages, C doesn't have a function to return a substring from a larger string. The **substr** function developed in Program 13.9 uses three arguments: the main string, index and length of the substring to be extracted. For valid inputs, the function returns a pointer to the substring which is declared as a **static** array inside the function. For invalid index or length, **substr** returns NULL.

```
/* substr.c: Uses a function to extract a substring from a string.
             Index and length of substring specified as arguments. */
#include <stdio.h>
#include <string.h>

char *substr(char *s, short start, short length);

int main(void)
{
   char *months = "JanFebMarAprMayJunJulAugSepOctNovDec";
   char *p;
   short index, len;

   fputs("Enter index and length of substring: ", stdout);
   scanf("%hd%hd", &index, &len);
   if ((p = substr(months, index, len)) != NULL)
      printf("Substring is %s\n", p);
   else
      fputs("Illegal arguments\n", stdout);

   return 0;
}
```

```
char *substr(char *s, short start, short length)
{
   static char stg[20];
   short i, j = 0;

   if (start + length > strlen(s))  /* If substring goes past array end */
      return NULL;
   else {
      for (i = start; i < start + length; i++)
         stg[j++] = s[i];
      stg[j] = '\0';
      return stg;
   }
}
```

PROGRAM 13.9: **substr.c**

```
Enter index and length of substring: 0 6
Substring is JanFeb
Enter index and length of substring: 33 3
Substring is Dec
Enter index and length of substring: 33 4
Illegal arguments
```

PROGRAM OUTPUT: **substr.c**

13.9 STANDARD STRING-HANDLING FUNCTIONS

Unlike C++ and Java, which have strong string-handling capabilities, C supports a limited number of string-handling functions. An enlarged list of functions supported by the standard library is available in Appendix A. Table 13.2 lists a subset of these functions which are discussed in the following sections. They are presented here with their prototypes and code snippets that highlight their usage. All of them need the **#include <string.h>** directive.

13.9.1 The strlen Function

Prototype: `size_t strlen(const char *s);`

```
printf("Enter string: ");
scanf("%[^\n]", stg1);                          If input is Burundi ...
printf("Length: %hd\n", strlen(stg1));          ... printf displays 7
```

The **strlen** function returns the length of the string without the NUL character. The value returned is of type `size_t`, which is an unsigned type guaranteed to return at least 16 bits.

13.9.2 The strcpy Function

Prototype: `char *strcpy(char *dest, const char *src);`

```
strcpy(stg2, stg1);                             Copies stg1 to stg2
printf("Copied string: %s\n", stg2);            Also prints Burundi
```

TABLE 13.2 String-Handling (`string.h`) Functions

Function	*Significance*
`size_t strlen(const char *s)`	Returns length of *s*.
`char *strcpy(char *dest, const char *src)`	Copies *src* to *dest*, returns pointer to *dest*.
`char *strcat(char *dest, const char *src)`	Appends *src* to *dest*, returns pointer to *dest*.
`int strcmp(const char *s1, const char *s2)`	Compares *s1* to *s2*, returns -ve, 0 or +ve integer depending on whether *s1* <, ==, or > than *s2*.
`char *strchr(const char *s, int c)`	Searches *c* in *s*, returns pointer to first occurrence of *c* in *s* or NULL if *c* not found in *s*.
`char *strrchr(const char *s, int c)`	As above, but returns pointer to last occurrence of *c* in *s* or NULL if *c* not found in *s*.
`char *strstr(const char *s1, const char *s2)`	Searches *s2* in *s1*, returns pointer to *s2* or NULL if *s2* not found in *s1*.

C doesn't allow a string *src* to be copied to *dest* using the assignment *dest* = *src* because that would mean having the same pointer value in both variables. A string is copied with the **`strcpy`** function. All characters of *src*, including the `'\0'`, are copied to *dest*, and **`strcpy`** returns a pointer to *dest*. Note that the destination string is always the first argument for functions that use two strings.

13.9.3 The `strcat` Function

Prototype: `char *strcat(char *dest, const char *src);`

```
printf("Enter second string: ");                    Assuming stg1 is Burundi ...
scanf("%[^\n]", stg3);                              ... if user enters Congo ...
strcat(stg3, stg1));                                ... stg3 is CongoBurundi
printf("Concatenated string: %s\n", stg3);          ... after concatenation
```

You can't use the + operator to concatenate two strings. The expression *dest* + *src* is invalid because that implies adding two pointers, which is not permitted in C. Strings are concatenated with the **`strcat`** function. The function overwrites the NUL at the end of *dest* with the string *src* and then writes a separate NUL at the end of the concatenated string. **`strcat`** returns a pointer to *dest*.

13.9.4 The `strcmp` Function

Prototype: `int strcmp(const char *s1, const char *s2);`

```
if (strcmp(s1, s2) == 0)
   printf("The two strings are equal\n");
else if (strcmp(s1, s2) > 0)
   printf("s1 is greater than s2\n");
else
   printf("s1 is less than s2\n");
```

It's not possible to use the comparison operators, ==, > and <, to compare two strings because strings evaluate to pointers and two pointers can only be compared when they refer to the same array. The **strcmp** function compares strings by comparing their individual characters and returns three values:

- 0 if both strings are identical.
- a positive integer if the first unmatched character of *s1* is greater than its counterpart in *s2*.
- a negative integer if the first unmatched character of *s1* is less than its counterpart in *s2*.

For determining the value of a comparison, the ASCII code list is used. Thus, `Angola` is less than `angola` because the ASCII value of `'A'` is 65 and that for `'a'` is 97. Similarly, `Barbados` is greater than `Bahamas` because `'r'` in `Barbados` (the first unmatched character) is greater than `'h'` in `Bahamas`. This function is extensively used for sort operations on strings.

13.9.5 The strchr and strrchr Functions

Prototype: `char *strchr(const char *`*s*`, int` *c*`);`

```
char stg[] = "Lesotho/Basutoland"; char *p;
if ((p = strchr(stg, c)) != NULL) {                    If c is 'B' ....
   printf("%c found at position %d\n", c, p - stg);    ... p - stg is 8 ...
   printf("Remaining string is %s\n", p);              ... p now points
}                                                      ... to Basutoland
```

The **strchr** function determines whether a character *c* can be found in a string *s*. The function returns a pointer to the *first* matched character, or NULL if the character is not found. The index of the found character can be easily computed by subtracting the pointer *s* from the returned pointer value. In the preceding example, if `c` is `'B'`, its index is `p - stg`, i.e. 8 (first index: 0).

The **strrchr** function is a variant that uses an identical prototype but returns the *last* occurrence of the character. To determine whether a character occurs only once in a string, the returned values of **strchr** and **strrchr** must be equal.

13.9.6 The strstr Function

Prototype: `char *strstr(const char *`*haystack*`, const char *`*needle*`);`

```
char *hstack = "JanFebMarAprMayJunJulAugSepOctNovDec"; char *p;
if ((p = strstr(hstack, ndl)) != NULL) {
   printf("%s found at position %d\n", ndl, p - hstack);
   printf("%s is month number %d\n",                    If ndl is May, the last ...
   ndl, (p - hstack) / 3 + 1);                          ... expression evaluates to 5
}
```

While **strchr** searches for a character, the **strstr** function searches for a substring. The prototype makes it clear that this function lets you search for *needle* in *haystack*. If the search is successful, **strstr** returns a pointer to *needle* that exists in *haystack*, or NULL otherwise.

13.10 THE CHARACTER-ORIENTED FUNCTIONS

For some tasks, you won't find an appropriate string-handling function in the standard library. If it's not worthwhile developing one, you have to inspect each character of the string to determine the action to be taken on it. The standard library doesn't have a function to change the case of a string or determine whether a string is fully alphabetic. C makes up for this deficiency—at least partially—by supporting a number of character-handling functions (Table 13.3).

Most of these functions test a specific attribute and simply return true or false. For instance, the **`isdigit`** function can be used in a loop to validate a string that must not have any digits:

```
while ((c = arr[i++]) != '\0') {
   if (isdigit(c)) {
      fputs("String is invalid because it contains a digit", stderr);
      break;
   }
}
```

Table 13.3 also lists two functions (**`tolower`** and **`toupper`**) that *change* the character. You have used both of them in Program 13.3, so you know how to use them. All character-oriented functions need the **`#include <ctype.h>`** directive.

TABLE 13.3 Character-handling (`ctype.h`) Functions

Function	*Returns True If* `c` *is*
`isalnum(c)`	An alphanumeric character
`isalpha(c)`	An alphabetic character
`isblank(c)`	A blank character (space or tab) *(C99)*
`iscntrl(c)`	A control character
`isdigit(c)`	A digit (0 to 9)
`isgraph(c)`	A printable character except space
`islower(c)`	A lowercase letter
`isprint(c)`	A printable character including space
`ispunct(c)`	A printable character excluding space and alphanumeric characters
`isspace(c)`	A whitespace character (space, `'\f'`, `'\n'`, `'\r'`, `'\t'` and `'\v'`).
`isupper(c)`	An uppercase letter
`isxdigit(c)`	A hexadecimal digit
Function	*Significance*
`toupper(c)`	Converts letter `c` to uppercase, if possible.
`tolower(c)`	Converts letter `c` to lowercase, if possible.

13.11 password_check.c: USING THE CHARACTER-HANDLING FUNCTIONS

Program 13.10 validates a user-input string meant to represent a password. The string must have a length of at least eight characters and can have only letters, digits and other printable characters. The outer **while** loop tests the string length while the inner loop tests the character attributes. Based on these tests, three flags are set inside the inner loop which is prematurely terminated when an

```
/* password_check.c: Validates a password string (Minimum: 8 characters,
                     with letters, digits and special characters only) */
#include<stdio.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
   char stg[80];
   short c, i, is_digit, is_alpha, is_punct;

   while (fputs("Enter password (8 characters minimum) : ", stdout)) {
      scanf("%[^\n]", stg);
      while (getchar() != '\n')
         ;
      if (strlen(stg) < 8) {
         fputs("Password less than 8 characters\n", stdout);
         continue;
      }

      i = is_digit = is_alpha = is_punct = 0;
      while ((c = stg[i++]) != '\0') {
         if (isdigit(c))
           is_digit = 1;
         else if (isalpha(c))
           is_alpha = 1;
         else if (ispunct(c))
           is_punct = 1;
         else
           break;                    /* String contains invalid character */
      }

      if (is_digit == 1 && is_alpha == 1 && is_punct == 1) {
         printf("Valid password\n");
         break;
      }
      else
         printf("Only letters, digits and special characters allowed\n");
   }
   return 0;
}
```

PROGRAM 13.10: **passwd_check.c**

```
Enter password (8 characters minimum) : flocci
Password less than 8 characters
Enter password (8 characters minimum) : flocci123
Only letters, digits and special characters allowed
Enter password (8 characters minimum) : flocc_123
Valid password
```

PROGRAM OUTPUT: **passwd_check.c**

invalid character is encountered. Efficient use of **break** and **continue** has kept the program lean and simple.

13.12 TWO-DIMENSIONAL ARRAY OF STRINGS

A string can be stored in an array of type `char`, so multiple strings can be stored in a two-dimensional array. The number of strings that can be stored is determined by the number of rows (first subscript), and the maximum length of each string is set by the number of columns (second subscript). Here's how we assign four strings to a 2D array of type `char`:

```
char types[4][12] = { {'c', 'h', 'a', 'r', '\0'},
                      {'s', 'h', 'o', 'r', 't', '\0'},
                      {'i', 'n', 't', '\0'},
                      {'l', 'o', 'n', 'g', '\0'} };
```

When using this technique, we need the inner braces because all four strings are shorter than the number of columns (12). This is not a convenient way of populating arrays with strings. A simpler method is to use a set of double-quoted strings enclosed in a single set of braces:

```
char types[4][12] = {"char", "short", "int", "long"};
```

For printing a string held in a 2D array, **printf** needs the array name along with the subscript that represents the row containing the string:

```
printf("%s\n", types[1]);          Prints short
printf("%s\n", types[4]);          Prints long
printf("%s\n", types);             Prints char, same as types[0]
```

Program 13.11 fills up a 2D array of type `char` using keyboard input. Each word of the input string is read into the variable `stg2d[i]`, which represents a row of the array `stg2d`. Each string is printed using the pointer notation, `*(stg2d + i)`, which is the same as `stg2d[i]`. Note that some of the space allocated to the array is wasted mainly because the strings are of unequal length, an issue that will be addressed soon.

13.13 ARRAY OF POINTERS TO STRINGS

The array, `stg2d[7][20]`, that was used in Program 13.11 has space for seven strings of 19 characters each. No space is wasted if all seven strings have this size, but that is often not the case. Some wastage is thus inevitable. However, when the strings are available as constants in a program, using an array of pointers to these strings is a better option. The following declaration,

```
char *stg[7];
```

```
/* input_to_2darray.c: Fills up a 2D array with single-word strings. */
#include<stdio.h>
int main(void)
{
   short i = 0;
   char stg2d[7][20];                /* A 2D array to hold multiple strings */
   fputs("Enter a multi-word string: ", stdout);
   while (scanf("%s", stg2d[i]) == 1) {                  /* Array notation */
      printf("stg2d[%hd] = %s\n", i, *(stg2d + i));      /* Pointer notation */
      i++;
   }
   return 0;
}
```

PROGRAM 13.11: **input_to_2darray.c**

```
Enter a multi-word string: Peru Brazil Bolivia Suriname Chile
stg2d[0] = Peru
stg2d[1] = Brazil
stg2d[2] = Bolivia
stg2d[3] = Suriname
stg2d[4] = Chile
[Ctrl-d]
```

PROGRAM OUTPUT: **input_to_2darray.c**

creates space in memory for storing seven pointers only. When declared without initialization, the elements have to be assigned individually:

```
stg[0] = "Sunday";                              stg[0] is a pointer
stg[1] = "Monday";
   ...
```

However, if the values pointed to are known beforehand, then you can combine declaration and initialization:

```
char *stg[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday" };
```

Each element of this array signifies a pointer that points to the first character of the respective string. For example, `stg[0]` contains the address of `'S'`, the first character of `"Sunday"`. The strings can be printed by running **printf** with the `%s` format specifier in a loop:

```
for (i = 0; i < 7; i++)
   printf("%s ", stg[i]);
```

In this arrangement, the pointers and pointed-to strings are not stored in one contiguous region of memory. The array of pointers are stored in one place, while the strings themselves are stored elsewhere. Figure 13.2 depicts the storage scheme for strings that have their pointers stored in an array of pointers. Even though the pointers require additional storage, no wastage is incurred in storing the strings.

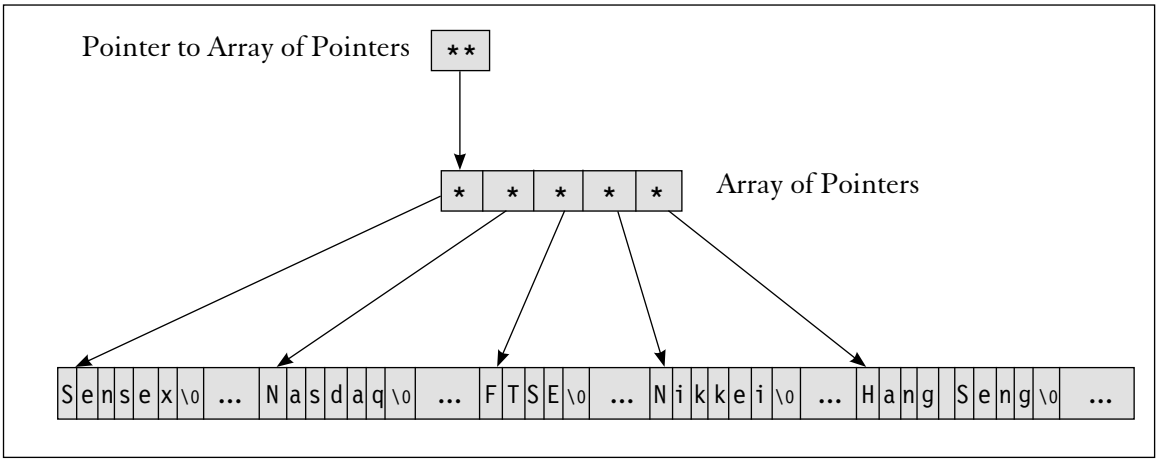


FIGURE 13.2 Storage Scheme for Array of Pointers to Strings

Note: Even though it's more efficient to store pointers to strings in the array `*stg[10]`, you can't use **`scanf("%s", stg[i])`** because `stg[i]` points to nowhere. You must provide the address of an existing array (like `temp[20]`) and then assign `temp` to `stg[i]`.

13.14 sort_strings.c: SORTING A 2D ARRAY OF STRINGS

Program 13.12 uses a 2D array to sort strings using the selection sort mechanism that was first discussed in Section 10.8 and reviewed in Section 13.7.3. The strings are input through the keyboard. The program makes use of the **`strcpy`** and **`strcmp`** functions to swap strings. The sort is case-sensitive and this is reflected in the first line of output.

```
/* sort_strings.c: Uses a 2D array to sort strings in ASCII sequence.
                   Algorithm used: selection sort */
#include<stdio.h>
#include <string.h>
#define ROWS 10
#define COLUMNS 20

int main(void)
{
   int i = 0, imax, j;
   char stg2d[ROWS][COLUMNS], temp[COLUMNS];

   fputs("Enter a multi-word string: ", stdout);
   while (scanf("%s", stg2d[i++]) == 1)
      ;

   imax = i - 1;
```

```
   for (i = 0; i < imax - 1; i++)                /* Selection sort algorithm ... */
      for (j = i + 1; j < imax; j++)             /* ... explained in Section 10.8 */
         if (strcmp(stg2d[i], stg2d[j]) > 0) {
            strcpy(temp, stg2d[i]);
            strcpy(stg2d[i], stg2d[j]);
            strcpy(stg2d[j], temp);
         }
   for(i = 0; i < imax; i++)
      printf("%s\n", stg2d[i]);
   return 0;
}
```

PROGRAM 13.12: **sort_strings.c**

```
Enter a multi-word string: sibelius mahler bach beethoven Mozart handel
[Ctrl-d]
Mozart                                  ASCII sequence lists uppercase before lowercase
bach
beethoven
handel
mahler
sibelius
```

PROGRAM OUTPUT: **sort_strings.c**

13.15 string_swap.c: SWAPPING TWO STRINGS

We swap two variables by passing their pointers as arguments to a swapping function *(12.7.1)*. How does one swap two strings that themselves are pointers? Simple, pass the addresses of these pointers to a function that accepts pointers to pointers as the two arguments. This task is achieved by Program 13.13.

The declaration of **p_swap** correctly shows the type of the parameters as `char **`. The function dereferences the two double pointers before swapping the (single) pointers. Note that the strings themselves remain lodged in their original locations in memory. We have encountered a similar situation before *(12.14)*.

```
/* string_swap.c: Swaps two strings by passing a pair of
                  pointers to a pointer to a swap function. */
#include <stdio.h>
void p_swap(char **fx, char **fy);
int main(void)
{
   char *stg1 = "Sunday"; char *stg2 = "Monday";
   printf("Before swap: stg1 = %s, stg2 = %s\n", stg1, stg2);
   p_swap(&stg1, &stg2);    /* Each argument is a pointer to a pointer */
   printf("After swap:  stg1 = %s, stg2 = %s\n", stg1, stg2);
   return 0;
}
```

```
void p_swap(char **fx, char **fy)
{
   char *temp = *fx;   *fx = *fy;   *fy = temp;
   return;
}
```

PROGRAM 13.13: **string_swap.c**

```
Before swap: stg1 = Sunday, stg2 = Monday
After swap:  stg1 = Monday, stg2 = Sunday
```

PROGRAM OUTPUT: **string_swap.c**

13.16 THE main FUNCTION REVISITED

So far, we have run our C programs without arguments (**a.out** or **AOUT**) instead of using, say, **a.out 3 7** (with two arguments). If programs of the operating system—like **COPY** and **MKDIR**—can be used with arguments, so why can't **a.out** be used in a similar manner? Yes, we can run C programs with command-line arguments, which means running **main** with arguments. For this to be possible, the definition of **main** must change to one of these two equivalent forms:

```
int main(int argc, char *argv[])
int main(int argc, char **argv)
```

The number of arguments is stored in `argc`. The arguments themselves are saved *as strings* in `argv`, which is an array of pointers to strings. Recall that `f_arr[]` is the same as `*f_arr` when used as function parameters *(12.12.1)*. By the same token, `char *argv[]` is synonymous with the pointer-to-pointer variable `**argv`. The parameter list for **main** thus allows the program to be invoked with a variable number of arguments.

Program 13.14 is meant to be invoked with three arguments for adding two numbers and displaying the last argument. The array of pointers, `argv`, stores, not three, but four strings. *The first element stores the pointer to the name of the program.* The remaining elements store pointers to the three arguments. Because the two numbers are input as strings, the **atoi** function is invoked to convert them to integers. After studying the output closely, you may not want all input to be read by **scanf** or **fgets** in your future programs.

```
/* main_with_args.c: A program that uses arguments. Definition of main
                     reflects number of arguments and pointers to them. */
#include <stdio.h>
#include <stdlib.h>                                        /* for atoi */

int main(int argc, char *argv[])
{
   int x = atoi(argv[1]);              /* First argument is the program name */
   int y = atoi(argv[2]);
```

```
    printf("Number of arguments = %d\n", argc);
    printf("First argument = %s\n", argv[0]);
    printf("Last argument = %s\n", argv[argc - 1]);
    printf("Sum of first two arguments = %d\n", x + y);

    return 0;
}
```

PROGRAM 13.14: **main_with_args.c**

```
$ a.out 333 555 "All arguments are strings"
Number of arguments = 4                          argc is 4 not 3
First argument = a.out                           Program name is first argument
Last argument = All arguments are strings
Sum of first two arguments = 888
```

PROGRAM OUTPUT: **main_with_args.c**

Note: It may seem strange that the program name is saved as the first argument, but there is an advantage for a program to know its own name. On UNIX systems (where C was originally developed), it is possible for a program to have multiple filenames. By using this (linking) feature, you can design a program to behave differently depending on the name by which it is invoked. This feature is not available in MSDOS/Windows.

WHAT NEXT?

We have examined the primary and derived data types supported by C. But can we create our own types, say, to store the three components of time in a single variable? C permits the creation of user-defined data types and we must know how to use this facility.

WHAT DID YOU LEARN?

A *string* is a sequence of characters terminated by `'\0'` or NUL. It can be interpreted as an array of type `char` or a double-quoted string. Both the name of the array and the quoted string are evaluated as a pointer to the first character of the string.

A string manipulated by a function of the standard library always has the NUL in place, but a programmer handling a string directly must make arrangements for appending the NUL to the string.

The **`fgets`** and **`fputs`** functions read and write a line of characters. Both functions can also perform I/O operations with disk files. For handling unformatted input and output, these functions are simpler to use than **`scanf`** and **`printf`**.

The **`sscanf`** and **`sprintf`** functions are used to read and write formatted data from and to a string, respectively. **`sscanf`** is often used with **`fgets`** to match input with multiple format specifiers.

The standard library includes functions for determining the length of a string (**strlen**), copy (**strcopy**), concatenate (**strcat**) and compare (**strcmp**) two strings. You can also search for a character (**strchr**) or a string (**strstr**) in another string.

The library also includes character-oriented functions to check a character attribute or convert the case of a character.

Multiple strings can be held in a two-dimensional char array, but if the strings are known at compile time, an array of pointers would be a better choice.

A program can be run with arguments by modifying the parameter list of **main**. The variable argc contains the number of arguments, and the array of pointers, argv, contains the arguments used during invocation of the program.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

13.1 The sequence 'abcd' signifies a pointer that contains the address of 'a'.

13.2 The empty string "" uses no memory.

13.3 stg in the declaration char stg[3] = "YES" ; is not a string.

13.4 The string used as the first argument to **sscanf** changes every time the function reads it.

13.5 If a char array is not initialized on declaration, each element must be assigned individually.

13.6 If p is declared as char *p; and arr is declared as char arr[10];, the statement arr = p; is legal.

13.7 If char *p = "Python";, the statement p[0] = 'p'; is illegal.

13.8 A string s1 cannot be copied to the string s2 by using s2 = s1.

13.9 The expression s1 + s2 concatenates two strings to evaluate to a larger string.

13.10 For storing strings of unequal length, an array of char pointers is a better option than a 2D char array.

13.11 The command-line arguments to a C program are read by the program as strings.

A2. FILL IN THE BLANKS

13.1 A function looks for the _____ character to determine the end of a string.

13.2 A string can be declared as a pointer of type _____ or as an array of type _____.

13.3 The _____ function must not be used because it can overflow the buffer.

13.4 The _____ function reads data from a string and the —— function writes data to a string.

13.5 For using the string-handling functions of the standard library, the file _____ must be included.

13.6 The _____ function checks for an alphabetic character, and the —— function checks for a digit.

13.7 For a program to accept command-line arguments, the signature of the _____ function must be modified.

A3. MULTIPLE-CHOICE QUESTIONS

13.1 For the declaration `char s[] = {'N', '0'};`, (A) s represents a string, (B) s doesn't represent a string, (C) length of s is 2, (D) length of s is 3, (E) B and C.

13.2 If `char *s = "Nokia";` and `char t[] = "Nokia";`, (A) s and t represent the address of `'N'`, (B) the pointer value of t can be changed, (C) pointer value of s can be changed, (D) A and B, (E) A and C.

13.3 If `char *p = "Redmi";` and `char arr[] = "Redmi";`, it is possible to use (A) `p++`, (B) `p + 1`, (C) `arr + 1`, (D) all of these, (E) A and C.

13.4 For the string s, the expression `*(s + strlen(s) - 1)` signifies (A) the last character, (B) the character prior to the last one, (C) implementation-dependent, (D) none of these.

13.5 If `char *s = "Samsung";`, the fourth character of the string can be accessed as (A) `s[4]`, (B) `s[3]`, (C) `*(s + 3)`, (D) A and C, (E) B and C.

13.6 The function call **`strcmp("somalia", "somalis");`** returns (A) 0, (B) 1, (C) a negative integer, (D) an integer greater than 1.

A4. MIX AND MATCH

13.1 Match the functions with their significance:

(A) **`strchr`**, (B) **`strstr`**, (C) **`strcat`**, (D) **`strcopy`**, (E) **`strcmp`**.

(1) Wrong function name, (2) comparison, (3) search for character, (4) concatenation, (5) search for string.

CONCEPT-BASED QUESTIONS

13.1 If a program crashes when **`printf`** is used with `%s`, what could the reason be?

13.2 Explain how **`scanf("%[^\n]", stg)`**, **`gets(stg)`** and **`fgets(stg, 80, stdin)`** treat the newline character. Can `stg` be treated as a string for all of them?

13.3 In B13.3, why must you declare `stg` as, say, `char stg[80];` instead of `char *stg;`?

13.4 Why is it sometimes advantageous to use **`sscanf`** to read a string instead of using **`scanf`** to read standard input?

13.5 Explain why the **`strcmp`** function may return one of three values.

13.6 Explain why the standard technique of passing pointers to a function doesn't work when swapping two strings.

PROGRAMMING & DEBUGGING SKILLS

13.1 Correct the errors in the following program:

```
#include <stdio.h>
int main(void)
{
   char *stg = "aBCD";
   printf("%s\n", stg);
   stg[0] = "A";
   printf("%s\n", stg);
   return 0;
}
```

13.2 Point out the errors in the following program which compiles correctly but fails at runtime:

```
#include <stdio.h>
int main(void)
{
   char *stg;
   printf("Enter a string: ");
   scanf("%s", &stg);
   printf("%s\n", stg);
   return 0;
}
```

13.3 Rewrite the following code to compile and execute correctly:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char stg[50];
   stg = "ice cream sandwich";
   printf("%s\n", toupper(stg));
   return 0;
}
```

13.4 If the `char` array `stg` is set to `"   Moonlight Sonata   "` in a program, reassign `stg` after removing all leading and trailing whitespace from the string.

13.5 Write a program that accepts an integer between 1 and 12 from the keyboard and prints the first three letters of the corresponding month. For instance, keying in `3` should print `Mar`. All month names must be extracted from a *single* string without using an **`if`** or **`switch`** statement.

13.6 Develop a function named **`char_replace(s, c1, c2)`** which replaces all occurrences of the character `c1` in the string `s` with `c2` and returns the number of characters replaced.

13.7 Write a program that creates a string containing the 26 letters of the English alphabet in both lower- and uppercase. The characters must be formed using one or more loops and the final string must look like `"AaBbCc...."`.

13.8 Write a program that accepts five lines from the keyboard and saves them in two 2D arrays depending on whether the line length is longer than 20 characters (excluding newline and NUL). The program must print the contents of both arrays.

13.9 Write a program containing the function **str_reverse(s1, s2)** that reverses the case of letters in the string s1 and saves the converted string in s2. The function must also return a pointer to s2.

13.10 Write a program that accepts a string from the user and inserts a **:** as the delimiter after every five characters before printing the delimited string.

13.11 Write a program containing the function **str_replace(s, s1, s2)** that replaces the *first* occurrence of substring s1 with s2 in the string s. The function must return a pointer to s if the replacement is successful, or NULL otherwise. (HINT: Use the **strstr** function.)

13.12 Write a program to determine the character that occurs most in a user-input string.

13.13 Write a program that uses the **str_insert(s, s1, s2**) function to locate a substring s1 in string s and insert a substring s2 before s1. The function must return a pointer to s if the insertion is successful, or NULL otherwise.

13.14 Correct the errors in the following program which is meant to concatenate a string stg with another one specified as a command-line argument:

```
#include <stdio.h>
#include <strings.h>
int main(int argc, char argv[])
{
   char stg[] = "ABCDEFGH";
   printf("Concatenated string = %s\n", strcat(stg, argv[0]));
   printf("Length of stg = %d\n", strlen(stg));
   return 0;
}
```

13.15 Write a program that splits a string on any of the delimiters stored in the string " |:;" (includes a space), and saves the words in a 2D char array. The program must accept a line of text from keyboard input and print each word on a separate line.

13.16 Write a program that sorts a set of five single-word strings input from the keyboard using a 2D char array and the bubble sort algorithm.

13.17 Develop a function named **str_compare(s1, s2)** which returns a positive integer if any character of string s2 occurs in s1. The function must return -1 on failure.

13.18 Correct the errors in the following program which is meant to accept an unspecified number of strings as command-line arguments and print them as a single concatenated string:

```
#include <stdio.h>
#include <strings.h>
int main(int argc, char argv[])
{
   int i;
   char stg[300];
   char *s = stg;
   for (i = 0; i < argc; i++)
      s = strcat(s, argv[i]);
   printf("Concatenated string = %s\n", s);
   return 0;
}
```

14 User-Defined Data Types

WHAT TO LEARN

- How *structures* extend the capabilities of arrays.
- The two-part sequence of declaring and defining structures.
- The restricted operations that can be performed on structures.
- How structures differ from arrays when used in functions.
- When a structure requires to be accessed by a pointer.
- How a *union* interprets a section of memory in multiple ways.
- The use of *bit fields* for handling data in units of bits rather than bytes.
- Enhance the readability of programs using *enumerators*.

14.1 STRUCTURE BASICS

C offers a basket of data types for handling different levels of complexity in the organization of data. The primary data types (like `int` and `float`) are meant for using simple and small amounts of data. The array is offered as a derived data type for handling a group of data items of the same type. Real-life situations often involve the use of heterogeneous data that need a mix of data types to be handled as a single unit. Structures and unions meet this requirement perfectly.

A group of data items are often logically related to one another. For instance, the attributes of an employee include the name, address, date of birth, salary, and so forth. Each of these attributes can be encapsulated as a *member* or *field* in a structure which can be manipulated as a *single* unit. Each member or field, however, retains its separate identity in the structure. That is not all; a member representing a date, for instance, can itself be a structure comprising the day, month and year as its members.

A *structure* is a data type that is designed by the user. However, unlike the other data types, a structure variable is based on a *template* that must be created first. Each member of the structure is accessed by the notation *structure*.*member*, and can be used in practically the same way as a variable. A structure

member can be a primary data type, an array or even another structure (or union). To accommodate multiple structures having the same type, C also supports an array of structures.

The advantage of using a structure is strongly felt when it is used as an argument to a function. It is more convenient to pass a structure containing all employee details as a single argument to a function rather than pass its members as separate arguments. A function can also return a structure, and if copying an entire structure bothers you, you can even pass a pointer to a structure as a function argument. Structures present the best of both worlds as you'll see soon.

14.2 DECLARING AND DEFINING A STRUCTURE

Because the components of a structure are determined by the user and not by C, a structure variable is created in two steps which may or may not be combined:

- Declare the structure to create a template which specifies the components of the structure. Declaration simply makes type information available to the compiler. However, the compiler doesn't allocate memory by seeing the declaration.
- Define a variable that conforms to the template. The compiler allocates memory for the variable and creates an *instance* of the template.

Declaration uses the **struct** keyword, optionally followed by the name of the structure and a set of specifications for each member or field. The following generalized syntax declares a structure named *struct_name*:

```
struct struct_name {
    data_type_1 member_name_1;          First member
    data_type_2 member_name_2;          Second member
        ...
    data_type_n member_name_n;          Last member
};
```

struct_name represents the *structure tag*. It is followed by the structure body enclosed in a set of curly braces terminated with a semicolon. The body contains the names of the members preceded by their data types. For instance, *member_name_1* has the data type *data_type_1*, which could be a primary, derived or user-defined type (including another structure). Each member specification is actually a declaration statement, the reason why it is terminated with a semicolon.

On seeing the declaration, the compiler determines how to organize the structure members in memory. It's only when you actually create structure variables (by definition), that memory is allocated by the compiler for the structure. The syntax of the definition also specifies the keyword **struct**, the structure tag (*struct_name*) and the variable name (*struct_var*):

```
struct struct_name struct_var;
```

Let's now declare and define a structure comprising three members that contain information of an audio CD sold in a store. This information is represented by the title, the quantity in stock and the price. The following declaration creates a structure named `music_cd`:

```
struct music_cd {                          music_cd is the structure tag
   char title[30];                         First member
   short qty;
   float price;
};                                         Declaration ends with ;
```

The structure tag (here, `music_cd`) is simply an identifier that is needed to create variables of this type. The names of the three members (`title`, `qty` and `price`) are preceded by their types and followed by the `;` terminator. Like `int` and `float`, `music_cd` is a data type, but a user-defined one.

You can now define one or more structure variables based on the created template. Just as a variable definition begins with the data type (`int x;`), the same is true for the definition of a structure:

```
struct music_cd disk1;                     Allocates memory for the members
struct music_cd disk1, disk2;              Can define multiple variables
```

The compiler allocates memory for the variables `disk1` and `disk2`, which have `struct music_cd` as their data type. These two words representing the type must precede every definition of this structure. However, C supports an abbreviation feature called **`typedef`** *(14.4.2)* that can create a simple synonym for the words `struct music_cd`.

Takeaway: Declaration of a structure creates a template but doesn't allocate memory for the members. Memory is allocated when variables based on this template are created by definition.

Note: The terms *declaration* and *definition* are often used interchangeably with structures. This book, however, follows the convention adopted by Kernighan and Ritchie to create a template by declaration and an instance (i.e., an actual object) by definition.

14.2.1 Accessing Members of a Structure

The variable `disk1` (or `disk2`) is a real-life object of type `struct music_cd`. The individual members are connected to their respective variables with a dot, the member operator. Thus, the members of `disk1` are accessed as `disk1.title`, `disk1.qty` and `disk1.price`. These members can be treated like any other variable, which means that you can assign values to them in the usual manner:

```
strcpy(disk1.title, "Mozart");
disk1.qty = 3;
disk1.price= 10.75;
```

Note that it is not possible to use `disk1.title = "Mozart";` because `title` is an array which signifies a constant pointer. You can also assign values to these members using **`scanf`**:

```
scanf("%s", disk1.title);
scanf("%hd", &disk1.qty);
```

The same dot notation must be used to print the values of these members. For instance, **`printf("%f", disk1.price);`** displays the value of the member named `price`.

Another structure in the program may also have a member named `title`, but the name `disk1.title` will be unique in the program. The naming conventions used for simple variables apply equally to structure members. This means that the name can't begin with a digit but can use the underscore.

14.2.2 Combining Declaration, Definition and Initialization

The preceding definitions created the variables `disk1` and `disk2` without initializing them. It is possible to combine declaration and definition by adding the variable name after the closing curly brace (Form 1). It is also possible to simultaneously initialize the variable (Form 2):

```
struct music_cd {
   char title[30];
   short qty;
   float price;
} disk1;
```

Form 1

```
struct music_cd {
   char title[30];
   short qty;
   float price;
} disk2 = {"Beethoven", 3, 12.5};
```

Form 2

For an initialized structure, the variable name is followed by an `=` and a list of comma-delimited values enclosed by curly braces. These initializers obviously must be provided in the right order. In Form 2, `"Beethoven"` is assigned to `title` and `3` to `qty`.

You can create multiple variables and initialize them at the same time, as shown in the following snippets which represent the last line of declaration:

```
} disk1, disk2, disk3;
} disk1 = {"Beethoven", 3, 12.5}, disk2 = {"Mahler", 4, 8.75};
```

Like with arrays, you can partially initialize a structure. If you initialize only `title`, the remaining members, `qty` and `price`, are automatically assigned zeroes.

By default, uninitialized structures have junk values unless they are global. A structure defined before **`main`** is a global variable, which means that the members are automatically initialized to zero or NULL.

14.2.3 Declaring without Structure Tag

If declaration and definition are combined, the structure tag can be omitted. This is usually done when no more structure variables of that type require to be defined later in the program. The following statement has the tag missing:

```
struct {                                                    No tag specified
   char title[30];
   short qty;
   float price;
} disk4 = {"Borodin", 5}, disk5 = {"Schubert"};
```

You can't subsequently create any more variables of this type. Here, both `disk4` and `disk5` are partially initialized, which means that `disk4.price`, `disk5.qty` and `disk5.price` are automatically initialized to zeroes.

Note: The structure tag is necessary when a structure is declared and defined at two separate places.

14.3 intro2structures.c: AN INTRODUCTORY PROGRAM

Program 14.1 declares a structure named `music_cd` containing three members. One of them is an array of type `char` meant to store a string. In this program, four variables are created and assigned values using different techniques:

- `disk1` Declared, defined and initialized simultaneously.
- `disk2` Defined separately but initialized fully.
- `disk3` Defined separately and initialized partially; two members are assigned values by **`scanf`**.
- `disk4` Defined separately but not initialized at all; its members are assigned individually.

```
/* intro2structures.c: Declares and initializes structures. */
#include <stdio.h>
#include <string.h>

int main(void)
{
   struct music_cd {                      /* Declares structure for music_cd */
      char title[27];                     /* Member can also be an array */
      short qty;
      float price;
   } disk1 = {"Bach", 3, 33.96};          /* Defines and initializes ...
                                             ... structure variable disk1 */
   printf("The size of disk1 is %d\n", sizeof disk1);

   /* Three more ways of defining structure variables */
   struct music_cd disk2 = {"Bruckner", 5, 6.72};
   struct music_cd disk3 = {"Handel"};    /* Partial initialization */
   struct music_cd disk4;

   /* Assigning values to members */
   strcpy(disk4.title, "Sibelius");
   disk4.qty = 7;
   disk4.price = 10.5;

   printf("The four titles are %s, %s, %s and %s\n",
           disk1.title, disk2.title, disk3.title, disk4.title);

   /* Using scanf */
   fputs("Enter the quantity and price for disk3: ", stdout);
   scanf("%hd%f", &disk3.qty, &disk3.price);
   printf("%s has %hd pieces left costing %.2f a piece.\n",
          disk3.title, disk3.qty, disk3.price);
   return 0;
}
```

PROGRAM 14.1: **`intro2structures.c`**

```
The size of disk1 is 36
The four titles are Bach, Bruckner, Handel and Sibelius
Enter the quantity and price for disk3: 3  7.75
Handel has 3 pieces left costing 7.75 a piece.
```

PROGRAM OUTPUT: **intro2structures.c**

Note that **sizeof** computes the total memory occupied by a structure. This is not necessarily the sum of the sizes of the members. For reasons of efficiency, the compiler tries to align one or more members on a word boundary. Thus, on this machine having a 4-byte word, `title` occupies 28 bytes (7 words) even though it uses 27 of them. `qty` and `price` individually occupy an entire word (4 bytes each), but `qty` uses two of these bytes. `disk1` thus needs 33 bytes (27 + 2 + 4) but it actually occupies 36 bytes.

Alignment issues related to structures lead to the creation of *slack bytes* or *holes* in the allocated memory segment (Fig. 14.1). If you reduce the size of the array to 26, no space is wasted and **sizeof disk1** evaluates to 32.

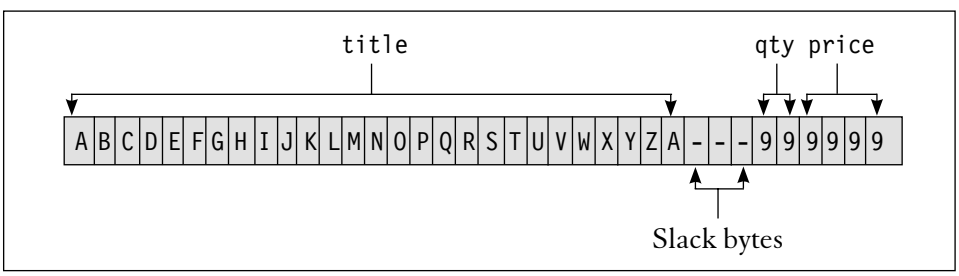


FIGURE 14.1 Memory Layout of Members of `music_cd`

14.4 IMPORTANT ATTRIBUTES OF STRUCTURES

Because a structure can include any data type as a member (including pointers, unions and other structures), this diversity leads to restrictions on the operations that can be performed on them. Be prepared for a surprise or two when you consider the following features of structures:

- *There are no name conflicts between structure templates, their instantiated variables and members.* It is thus possible to have the same names for all of them as shown in the following:

```
struct x {                          Name of template is the same ...
   char x[30];                      ... as a member and ...
   short y;
} x, y;                             ... a structure variable.
```

  Even though the compiler finds nothing wrong in this declaration and definition, a programmer would do well to avoid naming structures and members in this manner.

- *The =, dot, -> and & are the only operators that can be used on structures.* Two of them (= and &) retain their usual significance, while the dot and -> are exclusively meant for use with structures.

- *A structure can be copied to another structure provided both objects are based on the same template.* The first statement in the following code segment defines a variable `disk2` based on a template declared earlier. The second statement uses the assignment operator to copy all members of `disk2` to `disk3`:

```
struct music_cd disk2 = {"mozart", 20, 9.75};
struct music_cd disk3 = disk2;
```

  This feature is not available with arrays; all array elements have to be copied individually.

- *No arithmetic, relational or logical operations can be performed on structures even when the operation logically makes sense.* It is thus not possible to add two structures to form a third structure even if the members are compatible.

- *It is not possible to compare two structures using a single operator even though such comparison could be logically meaningful.* Each member has to be compared individually as shown by the following code:

```
if (strcmp(disk2.title, disk3.title) == 0
    && disk2.qty == disk3.qty
    && disk2.price == disk3.price)
       fputs("disk2 and disk3 are identical structures\n", stdout);
```

  If a structure contains 20 members, you need to use 20 relational expressions to test for equality. Unfortunately, C doesn't support a better option.

- *When a structure is passed by name as an argument to a function, the entire structure is copied inside the function.* This doesn't happen with arrays where only a pointer to the first element of the array is passed. However, copying can be prevented by passing a pointer to a structure as a function argument.

- *A member of a structure can contain a reference to another structure of the same type.* This property of *self-referential* structures is used for creating *linked lists*.

Barring the last attribute in the list, the other attributes will be examined in this chapter. Self-referential structures are discussed in Chapter 16.

Takeaway: An array and structure differ in three ways: 1) A structure variable can be assigned to another structure variable of the same type. 2) The name of a structure doesn't signify a pointer. 3) When the name of a structure is passed as an argument to a function, the entire structure is copied inside the function.

14.4.1 `structure_attributes.c`: Copying and Comparing Structures

Program 14.2 demonstrates the use of (i) the `=` operator to copy a four-member structure named `cricketer`, (ii) a set of four relational expressions to individually compare the members of two structure variables, `bat1` and `bat2`. `bat1` is declared, defined and initialized simultaneously and is then copied to `bat2` before they are compared.

```
/* structure_attributes.c: Copies a structure and tests whether two
                           structures are identical. */
#include <stdio.h>
#include <string.h>

int main(void)
{
   struct cricketer {
      char name[30];
      short runs;
      short tests;
      float average;
   } bat1 = {"Don Bradman", 6996, 52, 99.94};

   printf("bat1 values: %s, %hd, %hd, %.2f\n",
           bat1.name, bat1.runs, bat1.tests, bat1.average);

   struct cricketer bat2;
   bat2 = bat1;                        /* Copies bat1 to bat2 */
   printf("%s scored %hd runs in %hd tests at an average of %.2f.\n",
           bat2.name, bat2.runs, bat2.tests, bat2.average);

   /* Compares members of two structures */
   if (strcmp(bat2.name, bat1.name) == 0 && bat2.runs == bat1.runs
       && bat2.tests == bat1.tests && bat2.average == bat1.average)
      fputs("The two structures are identical\n", stdout);
   else
      fputs("The two structures are not identical\n", stdout);

   return 0;
}
```

PROGRAM 14.2: **structure_attributes.c**

```
bat1 values: Don Bradman, 6996, 52, 99.94
Don Bradman scored 6996 runs in 52 tests at an average of 99.94.
The two structures are identical
```

PROGRAM OUTPUT: **structure_attributes.c**

14.4.2 Abbreviating a Data Type: The typedef Feature

The **typedef** keyword is used to abbreviate the names of data types. Using a data type `LL` instead of `long long` involves less typing. Also, use of meaningful names makes programs easier to understand. The syntax of **typedef** is simple; follow **typedef** with the existing data type and its proposed synonym:

`typedef` *data_type new_name*`;`

You can now use *new_name* in place of *data_type*. Some of the declarations we have used previously can now be abbreviated using **typedef**:

```
typedef unsigned int UINT;          UINT is synonym for unsigned int
typedef unsigned long ULONG;        ULONG is synonym for unsigned long
```

These **typedef** statements are usually placed at the beginning of the program, but they must precede their usage. You can now declare variables of the UINT and ULONG types:

```
UINT int_var;                       int_var is of type unsigned int
ULONG long_var;                     long_var is of type unsigned long
```

Use of uppercase for synonyms is not mandatory but is recommended because you can then instantly spot the "typedef'd" data types. **typedef** is commonly used for creating synonyms to names of structures, their pointers and enumerators. The synonym can be formed when creating the template or after, as shown by the following forms shown side by side:

```
typedef struct student {
   char name[30];
   int dt_birth;
   short roll_no;
   short total_marks;
} EXAMINEE;
EXAMINEE stud1;
```

Form 1

```
struct student {
   char name[30];
   int dt_birth;
   short roll_no;
   short total_marks;
}
typedef struct student EXAMINEE;
EXAMINEE stud1, stud2;
```

Form 2

Form 1 combines the creation of the synonym named EXAMINEE with the declaration of student, which is preceded by the keyword **typedef**. In this form, the structure tag (student) is optional and can be dropped. But the second form creates the synonym *after* the declaration, so the tag is necessary. EXAMINEE is now a replacement for struct student, so you can create variables of type EXAMINEE.

Note: typedef was not designed simply to replace data types with short names. It was designed to make programs portable by choosing the right data types without majorly disturbing program contents. If you need 4-byte integers, you can use **typedef int INT32;** to create a synonym in a special file and then create variables of type INT32 in all programs. If the programs are moved to a machine where int uses 2 bytes, you can simply change the statement to **typedef long INT32;** without disturbing the contents of the programs. (The long data type uses a minimum of 4 bytes.)

14.4.3 structure_typedef.c: Simplifying Use of Structures

Program 14.3 demonstrates the use of **typedef** in abbreviating a primary data type and two structures. The structure named film is given the synonym MOVIE1 at the time of declaration. An identical but separate structure named cinema is typedef'd to MOVIE2 after declaration. Observe the dual use of **typedef** with cinema; the data type of year is typedef'd to USHORT before cinema is typedef'd to MOVIE2.

```
/* structure_typedef.c: Demonstrates convenience of abbreviating
                        data types with typedef. */
#include <stdio.h>

int main(void)
{
   typedef struct film {                  /* Declaration also creates synonym */
      char title[30];
      char director[30];
      unsigned short year;
   } MOVIE1;                              /* Synonym MOVIE1 created as a data type */

   MOVIE1 release1 = {"The Godfather", "Francis Ford Coppola", 1972};
   printf("release1 values: %s, %s, %hd\n",
           release1.title, release1.director, release1.year);

   typedef unsigned short USHORT;         /* Synonym USHORT created here ...  */
   struct cinema {
      char title[30];
      char director[30];
      USHORT year;                        /* ... and used here */
   };

   typedef struct cinema MOVIE2;          /* Synonym created after declaration */
   MOVIE2 release2 = {"Doctor Zhivago", "David Lean", 1965};
   printf("release2 values: %s, %s, %hd\n",
           release2.title, release2.director, release2.year);

   return 0;
}
```

PROGRAM 14.3: **structure_typedef.c**

```
release1 values: The Godfather, Francis Ford Coppola, 1972
release2 values: Doctor Zhivago, David Lean, 1965
```

PROGRAM OUTPUT: **structure_typedef.c**

Note: Even though `film` and `cinema` have identical members, they represent separate templates. Their variables are thus not compatible for copying operations. It's not possible to assign `release1` (of type `film`) to `release2` (of type `cinema`) or vice versa.

14.5 NESTED STRUCTURES

A structure member can have any data type—including another structure (but not of the same type though). The inner structure can contain another structure and so on. C supports *nested structures* and the following outer structure named `student` contains an inner structure named `dt_birth` as one of its four members:

```
struct student {
   char name[30];                          Member 1
   struct {                                Structure as member
      short day;
      short month;
      short year;
   } dt_birth;                             Member 2
   int roll_no;                            Member 3
   short total_marks;                      Member 4
};
struct student stud1;
```

Here, the declaration of the inner structure forms part of the declaration of the outer one. Note that `dt_birth` is actually a variable to which its members are connected. *However, you can't separately create a structure variable of this type.* This restriction, however, doesn't apply if the two structures are declared separately. Define the inner structure before the outer one:

```
struct dob {                               dob must be declared before ...
   short day;
   short month;
   short year;
};
struct student {
   char name[30];
   struct dob dt_birth;                    ... it is used in student.
   int roll_no;
   short total_marks;
};

struct student stud1;
struct dob dob1;
```

Encapsulating the three components of a date into a separate structure has two advantages over using separate "unstructured" members. First, you can pass this structure to a function as a single argument without losing the ability of individually accessing its members. Second, because `dob` can be used by multiple programs, its declaration could be stored in a separate file. A program that uses a date field as a three-member structure can simply include this file.

14.5.1 Initializing a Nested Structure

When it comes to initialization, nested structures resemble multi-dimensional arrays. For every level of nesting, a set of inner braces have to be used for members of the inner structure. This is how you initialize the structure of the preceding template:

```
struct stud1 = {"Oskar Barnack", {1, 11, 1879}, 3275, 555};
```

The three initializers inside the inner braces represent the values of `dob` members. The inner braces are optional but you must use them for two reasons. First, they provide clarity when associating initializers with their members. Second, they let you partially initialize a structure. For instance, you can drop the value for `month` or both `month` and `year` while initializing `stud1`.

14.5.2 Accessing Members

A member of an inner structure is connected by a dot to its immediately enclosing structure, which is connected by another dot to the outer structure. For a structure with a single level of nesting, the innermost member can be accessed as

outer_structure.*inner_structure*.*member*

In the present case, after `student` has been instantiated to form the variable `stud1`, `month` can be accessed as

`stud1.dt_birth.month`

`scanf` *needs the* `&` *prefix*

For every increment in the level of nesting, the number of dots increase by one. This dot-delimited notation is similar to the pathname of a file. On Windows and UNIX systems, the pathname `a/b/c` refers to a file named `c` having `b` as its parent directory, which in turn has `a` as its parent. For a structure member, the name `a.b.c` can be interpreted in a similar manner: `a` and `b` must be names of structures while `c` can never be one.

14.5.3 `structure_nested.c`: Program Using a Nested Structure

Program 14.4 uses a nested structure where the inner structure named `dob` is incorporated as a member of the outer structure named `student`. This data type is typedef'd to `EXAMINEE` and used to create two variables, `stud1` and `stud2`. `stud1` is initialized but `stud2` is populated by **`scanf`**. Note the use of the flag `0` in the **`printf`** format specifier (`%02hd`). This flag pads a zero to the `day` and `month` members of `dt_birth` to maintain the 2-digit width.

```
/* structure_nested.c: Shows how to access each member of a nested structure
                       using structure1.structure2.member notation. */
#include <stdio.h>
int main(void)
{
   struct dob {                          /* Must be declared before student */
      short day;
      short month;
      short year;
   };
   typedef struct student {
      char name[30];
      struct dob dt_birth;               /* Member is a nested structure */
      int roll_no;
      short total_marks;
   } EXAMINEE;                           /* Synonym for struct student */

   EXAMINEE stud1 = {"Oskar Barnack", {1, 11, 1879}, 3275, 555};
   printf("Name: %s, DOB: %02hd/%02hd/%hd, Roll No: %d, Marks: %hd\n",
           stud1.name, stud1.dt_birth.day, stud1.dt_birth.month,
           stud1.dt_birth.year, stud1.roll_no, stud1.total_marks);
```

```
    EXAMINEE stud2;
    fputs("Enter name: ", stdout);
    scanf("%[^\n]", stud2.name);            /* Possible to enter multiple words */

    while (getchar() != '\n')
       ;                                    /* Clears newline from buffer */

    fputs("Enter DOB (dd/mm/yyyy), roll no. and marks: ", stdout);
    scanf("%2hd/%2hd/%4hd %d %hd", &stud2.dt_birth.day, &stud2.dt_birth.month,
           &stud2.dt_birth.year, &stud2.roll_no, &stud2.total_marks);

    printf("Name: %s, DOB: %02hd/%02hd/%hd, Roll No: %d, Marks: %hd\n",
            stud2.name, stud2.dt_birth.day, stud2.dt_birth.month,
            stud2.dt_birth.year, stud2.roll_no, stud2.total_marks);
    return 0;
}
```

PROGRAM 14.4: **structure_nested.c**

```
Name: Oskar Barnack, DOB: 01/11/1879, Roll No: 3275, Marks: 555
Enter name: Carl Zeiss
Enter DOB (dd/mm/yyyy), roll no. and marks: 11/9/1816 5723 666
Name: Carl Zeiss, DOB: 11/09/1816, Roll No: 5723, Marks: 666
```

PROGRAM OUTPUT: **structure_nested.c**

14.6 ARRAYS AS STRUCTURE MEMBERS

An array can also be a member of a structure, which is evident from the way we used one of type `char` to represent the name or title in `music_cd`, `cricketer` and `student`. But structures in C support arrays of any type as shown in the following modified form of the `student` structure:

```
struct student {
   char name[30];
   int roll_no;
   short marks_pcb[3];                  /* Member is an array of 3 elements */
};
```

Like nested structures, a variable of this type is initialized using a separate set of braces for the values related to `marks_pcb`. You are aware that the inner braces are not mandatory but they provide clarity:

```
struct student stud1 = {"Oskar Barnack", 3275, {85, 97, 99}};
```

We can access the individual elements of `marks_pcb` as `stud1.marks_pcb[0]`, `stud1.marks_pcb[1]` and so forth. These elements are handled as simple variables:

```
stud1.marks_pcb[0] = 90;
printf("Physics Marks: %hd\n", stud1.marks_pcb[0]);
scanf("%hd", &stud1.marks_pcb[0]);
```

Could we not have used three short variables here? Yes, we could have, and added another two variables if we wanted to include five subjects. But would you like to use five variables with five statements or a single array in a loop to access all the marks?

Note: Besides providing clarity, the inner braces in the initializer segment allow the partial initialization of array elements and members of nested structures.

14.7 ARRAYS OF STRUCTURES

We used two structure variables, stud1 and stud2, to handle data of two students. This technique won't work with 500 students. C supports an array of structures, whose definition may or may not be combined with the declaration of the structure. The following statement defines an array of type struct student that can hold 50 sets (or records) of student data:

```
struct student {
   char name[30];
   int roll_no;
   short marks;
} stud[50];                                   Memory allocated for 50 elements
```

Alternatively, you can separate the declaration and definition. You can also use **typedef** to replace struct student with STUDENT:

```
typedef struct student {
   char name[30];
   int roll_no;
   short marks;
} STUDENT;
STUDENT stud[50];                             All elements now uninitialized
```

Because stud is an array, its elements are laid out contiguously in memory. The size of each array element is equal to the size of the structure in memory after accounting for slack bytes. Pointer arithmetic can easily be employed here to access each array element, and using a special notation (->), the members as well *(14.9)*.

This array can be partially or fully initialized by enclosing the initializers for each array element in a set of curly braces. Each set is separated from its adjacent one by a comma, while the entire set of initializers are enclosed by outermost braces:

```
STUDENT stud[50] = {
                     {"Benjamin Franklin", 1234, 90},
                     {"Max Planck", 4321, 80},
                        ...
                     {"Albert Einstein", 9876, 70}
                   };
```

Each member of each element of this array is accessed by using a dot to connect the member to its array element. For instance, the member name is accessed as stud[0].name, stud[1].name and so on. If you have not initialized the array, then you'll have to separately assign values in the usual manner:

```
strcpy(stud[5].name, "Emile Berliner");
stud[5].roll_no = 3333;
stud[5].marks = 75;
```

You can use **scanf** to input values to each member using pointers to these variables (like `&stud[i].marks`). A simple **for** loop using the array index as the key variable prints the entire list:

```
for (i = 0; i < 50; i++)
   printf("%s %d %hd\n", stud[i].name, stud[i].roll_no, stud[i].marks);
```

An array of structures is akin to *records* with *fields*. In the early days of commercial data processing, an employee's details were held as individual fields in a record. The fields of each record were processed before the next record was read. The same principle now applies to an array of structures and the two programs that are taken up next clearly demonstrate this.

Tip: An array of structures can take up a lot of memory, so make sure that you set the size of the array to a realistic value. On a Linux system, **sizeof stud** evaluated to 2000 bytes, i.e., 40 bytes for each element.

14.7.1 array_of_structures.c: Program for Displaying Batting Averages

Program 14.5 features an array of structures of type `cricketer`. Each element of the array variable, `bat`, stores the batting average and number of tests played by an individual. `bat` is partially initialized with the values of two players. The third array element is assigned separately while the fourth element is populated with **scanf**. The complete list is printed using **printf** in a loop.

The program also uses the **FLUSH_BUFFER** macro for ridding the buffer of character remnants that are left behind every time **scanf** reads a string. Note the use of the **-** symbol in the format specifier of the last **printf** statement. The **-** left-justifies the name and country—the way string data should be printed.

```
/* array_of_structures.c: Demonstrates techniques of defining, initializing
                          and printing an array of structures. */
#include <stdio.h>
#include <string.h>
#define FLUSH_BUFFER while (getchar() != '\n');

int main(void)
{
   short i, imax;
   struct cricketer {
      char name[20];
      char team[15];
      short tests;
      float average;
   } bat[50] = {
                 {"Don Bradman", "Australia", 52, 99.94},
                 {"Greame Pollock", "South Africa", 23, 60.97}
               };
   i = 2;
```

```
   strcpy(bat[i].name, "Wally Hammond");
   strcpy(bat[i].team, "England");
   bat[i].tests = 85; bat[i].average = 58.46f;

   i++;
   fputs("Enter name: ", stdout);
   scanf("%[^\n]", bat[i].name);

   FLUSH_BUFFER
   fputs("Enter team: ", stdout);
   scanf("%[^\n]", bat[i].team);

   fputs("Enter tests and average: ", stdout);
   scanf("%hd %f", &bat[i].tests, &bat[i].average);

   imax = i;
   for (i = 0; i <= imax; i++)
      printf("%-20s %-15s  %3hd  %.2f\n",
             bat[i].name, bat[i].team, bat[i].tests, bat[i].average);
   return 0;
}
```

PROGRAM 14.5: **array_of_structures.c**

```
Enter name: Sachin Tendulkar
Enter team: India
Enter tests and average: 200 53.79
Don Bradman          Australia         52  99.94
Greame Pollock       South Africa      23  60.97
Wally Hammond        England           85  58.46
Sachin Tendulkar     India            200  53.79
```

PROGRAM OUTPUT: **array_of_structures.c**

14.7.2 structure_sort.c: Sorting an Array of Structures

Program 14.6 fully initializes an array of five structures. The structures are then sorted in descending order on the marks field, using the selection sort algorithm that has been discussed previously *(10.8)*. The program prints the student details before and after sorting.

```
/* structure_sort.c: Sorts an array of structures on the marks field.
                     Algorithm used: selection sort */
#include <stdio.h>
#define COLUMNS 5

int main(void)
{
   short i, j, imax;
```

```
   struct student {
      char name[30];
      int roll_no;
      short marks;
   } temp, stud[COLUMNS] = {
                              {"Alexander the Great", 1234, 666},
                              {"Napolean Bonaparte", 4567, 555},
                              {"Otto von Bismark", 8910, 999},
                              {"Maria Teresa", 2345, 777},
                              {"Catherine The Great", 6789, 888}
                            };

   printf("Before sorting ...\n");
   for (i = 0; i < COLUMNS; i++)
      printf("%-20s %4d %4hd\n", stud[i].name, stud[i].roll_no, stud[i].marks);

   printf("\nAfter sorting ...\n");
   imax = i;
   for (i = 0; i <  imax - 1; i++)           /* Selection sort algorithm ... */
      for (j = i + 1; j < imax; j++)         /* ... explained in Section 10.8 */
         if (stud[j].marks > stud[i].marks) {
            temp = stud[i];
            stud[i] = stud[j];
            stud[j] = temp;
         }

   for (i = 0; i < COLUMNS; i++)
      printf("%-20s%5d%5hd\n", stud[i].name, stud[i].roll_no, stud[i].marks);

   return 0;
}
```

PROGRAM 14.6: **structure_sort.c**

```
Before sorting ...
Alexander the Great  1234  666
Napolean Bonaparte   4567  555
Otto von Bismark     8910  999
Maria Teresa         2345  777
Catherine The Great  6789  888

After sorting ...
Otto von Bismark     8910  999
Catherine The Great  6789  888
Maria Teresa         2345  777
Alexander the Great  1234  666
Napolean Bonaparte   4567  555
```

PROGRAM OUTPUT: **structure_sort.c**

It's impractical to provide large amounts of data in the program. In real-life, structure data are saved in a file, with each line (or record) representing the data of members stored in an array element. Chapter 15 examines the standard functions that read and write files and how they can be used in handling data of structures.

14.8 STRUCTURES IN FUNCTIONS

The importance of structures can be strongly felt when they are used in functions. Functions use structures in practically every conceivable way, and this versatility makes structures somewhat superior to arrays when used in functions. A structure-related data item can take on the following forms when used as a function argument:

- An element of a structure. The function copies this element which is represented in the form *structure.member*.
- The structure name. The function copies the entire structure. *It doesn't interpret the name of the structure as a pointer.* This is not the case when the name of an array is passed to a function.
- A pointer to a structure or a member. This technique is normally used for returning multiple values or avoiding the overhead of copying an entire structure inside a function.

A function can also return any of these three objects. Structures present the best of both worlds because a function can be told to either copy the entire structure or simply its pointer. The former technique is memory-intensive but safe while the latter could be unsafe if not handled with care.

We declare a function before **main** for its signature to be visible throughout the program. This means that the declaration of a structure used by a function must precede the declaration of the function, as shown in the following:

```
struct date {                                   Structure declared before main ...
   short day;
   short month;
   short year;
};
void display(struct date d);                    ... and before function declaration
```

The **display** function returns nothing but it could have an implementation similar to this:

```
void display(struct date d)                     Function definition
{
   printf("Date is %hd/%hd/%hd\n", d.day, d.month, d.year);
   return;
}
```

After the `date` template has been instantiated, you can invoke the function inside **main** using the structure variable as argument:

```
struct date today = {29, 2, 2017};
display(today);                                 Displays Date is 29/2/2017
```

Note that the **display** function copies the entire structure, `today`. However, this copy disappears after the function has returned. Can we use this copy to change the original structure? We'll soon learn that we can.

Takeaway: A function using a structure as argument copies the entire structure. If the structure contains an array, it too will be copied.

Caution: A program may run out of memory when using structures as arguments. A structure containing numerous members and large arrays can lead to stack overflow and cause the program to abort.

14.8.1 `structure_in_func.c`: An Introductory Program

Program 14.7 uses two functions that use a three-member structure named `time` as an argument. The **`display_structure`** function simply outputs the values of the three members. The **`time_to_seconds`** function returns the time after conversion to seconds. Note that we have now moved the structure declaration before **`main`** and before the function declarations. `time` is now a global template even though its variable `t` is not global.

```
/* structure_in_func.c: Use a structure as a function argument. */
#include <stdio.h>
struct time {                   /* Must be declared before main because ... */
   short hour;                  /* ... following function declarations ... */
   short min;                   /* ... use this structure. */
   short sec;
};
void display_structure(struct time f_time);
int time_to_seconds(struct time f_time);
int main(void)
{
   struct time t;
   fputs("Enter a time in hh:mm:ss format: ", stdout);
   scanf("%hd:%hd:%hd", &t.hour, &t.min, &t.sec);
   display_structure(t);
   int value = time_to_seconds(t);
   printf("Value of t in seconds: %d\n", value);
   return 0;
}
void display_structure(struct time f_time)
{
   printf("Hours: %hd, Minutes: %hd, Seconds: %hd\n",
           f_time.hour, f_time.min, f_time.sec);
   return;
}
int time_to_seconds(struct time f_time)
{
   return f_time.hour * 60 * 60 + f_time.min * 60 + f_time.sec;
}
```

PROGRAM 14.7: **`structure_in_func.c`**

```
Enter a time in hh:mm:ss format: 9:15:45
Hours: 9, Minutes: 15, Seconds: 45
Value of t in seconds: 33345
```

PROGRAM OUTPUT: **structure_in_func.c**

14.8.2 time_difference.c: Using a Function that Returns a Structure

A function can also return a structure which often has the same type that is passed to it. In the following declaration, the **time_diff** function accepts two arguments of type `struct time` and also returns a value of the same type:

```
struct time time_diff(struct time t1, struct time t2);
```

Program 14.8 computes the difference between two times that are input to **scanf** in the form *hh*:*mm*:*ss*. The values are saved in two structure variables, `t1` and `t2`, that are typedef'd to `TIME`. A third variable, `t3`, stores the returned value. Note that the `mins` operand is shared by the equal-priority operators, `--` and dot, but because of L-R associativity, no parentheses are needed.

```
/* time_difference.c: Uses a function to compute and return
                      the difference between two times as a structure. */
#include <stdio.h>

typedef struct time {
   short hours;
   short mins;
   short secs;
} TIME;

TIME time_diff(TIME t1, TIME t2);           /* Function returns a structure */

int main(void)
{
   TIME t1, t2, t3;

   fputs("Enter start time in hh:mm:ss format: ", stdout);
   scanf("%hd:%hd:%hd", &t1.hours, &t1.mins, &t1.secs);

   fputs("Enter stop time in hh:mm:ss format: ", stdout);
   scanf("%hd:%hd:%hd", &t2.hours, &t2.mins, &t2.secs);

   t3 = time_diff(t1, t2);
   printf("Difference: %hd hours, %hd minutes, %hd seconds\n",
           t3.hours, t3.mins, t3.secs);
   return 0;
}
```

```
TIME time_diff(TIME t1, TIME t2)
{
   TIME diff;                          /* TIME visible here because it is ... */
   if (t1.secs > t2.secs) {            /* ... declared before main */
      t2.mins--;
      t2.secs += 60;
   }
   if (t1.mins > t2.mins) {
      t2.hours--;
      t2.mins += 60;
   }
   diff.secs = t2.secs - t1.secs;
   diff.mins = t2.mins - t1.mins;
   diff.hours = t2.hours - t1.hours;

   return diff;
}
```

PROGRAM 14.8: **time_difference.c**

```
Enter start time in hh:mm:ss format: 6:45:50
Enter stop time in hh:mm:ss format: 8:35:55
Difference: 1 hours, 50 minutes, 5 seconds
```

PROGRAM OUTPUT: **time_difference.c**

Since t3 now has three new values, it can be said that **time_diff** has "returned" three values. This property of structures breaks the well-known maxim that a function can return a single value using the **return** statement. We'll use this property to provide an alternative solution to the swapping problem.

Note: The dot has the same precedence as the increment and decrement operators, but it has L-R associativity. Thus, in the expression t2.mins--, the dot operates on mins before the -- does, so we don't need to use (t2.mins)-- in the program.

14.8.3 swap_success2.c: Swapping Variables Revisited

The programs, **swap_failure.c** (Program 11.3) and **swap_success.c** (Program 12.5) taught us an important lesson: a function can interchange the values of two variables only by using their pointers as arguments. What if these two variables are members of a structure? Program 14.9 proves that it is possible to swap them without using pointers. The game-changer in this program is the statement, **s = swap(s);**, which merits some discussion.

The **swap** function here accepts and returns a two-member structure of type two_var. This function swaps the copies of s.x and s.y in the usual manner but it also returns a copy of the structure. The swapping operation succeeds here because this returned value is assigned to the same variable that was passed as argument (**s = swap(s);**). Using this technique, it is possible for a function to "return" multiple values without using pointers.

```
/* swap_success2.c: Successfully swaps two variables using a structure
                    and without using a pointer. */
#include <stdio.h>

struct two_var {
   short x;
   short y;
} s = { 1, 10};

struct two_var swap(struct two_var);

int main(void)
{
   printf("Before swap: s.x = %hd, s.y = %hd\n", s.x, s.y);
   s = swap(s);                          /* The game changer */
   printf("After swap: s.x = %hd, s.y = %hd\n", s.x, s.y);

   return 0;
}

struct two_var swap(struct two_var z)
{
   short temp;
   temp = z.x;   z.x = z.y;   z.y = temp;
   return z;
}
```

PROGRAM 14.9: **swap_success2.c**

```
Before swap: s.x = 1, s.y = 10
After swap: s.x = 10, s.y = 1
```

PROGRAM OUTPUT: **swap_success2.c**

Takeaway: The assignment property (=) of structures, because of which one structure variable can be assigned to another, allows a function to change the original values of structure members using their copies.

14.9 POINTERS TO STRUCTURES

Pointers represent the ultimate access mechanism of structure members. As with the other data types, a pointer to a structure must be defined before it is pointed to a structure variable. These steps are shown in the following statements:

```
struct rectangle {
   short length;
   short width;
} rect1 = {10, 20};

struct rectangle *p;                    Creates pointer p
p = &rect1;                             p points to rect1
```

The structure members can be accessed in three ways; two of them use a pointer with standard and special notation. For instance, `length` can be accessed as

- `rect1.length` The standard access mechanism that doesn't use a pointer.
- `(*p).length` Pointer dereferenced with * and then connected to the member.
- `p->length` Uses a special operator, `->`, that works only with structures.

Pointers are so often used with structures that C supports a special operator, `->`, to access a structure member in a convenient manner. This symbol is formed by combining the hyphen with the > symbol. Once a structure has been pointed to, you can access and manipulate a pointed-to member in the following ways:

Using **(*p).length**	*Using* **p->length**

`(*p).length = 100;`
`(*p).length++;`
`scanf("%hd", &(*p).length);`
	`p->length = 30;`
	`p->length++;`
	`scanf("%hd", &p->length);`

Form 1	Form 2

The technique shown in Form 1 encloses the *, the standard dereference symbol, in parentheses because the dot has a higher precedence than the *. In most cases, we'll use the `->` operator rather than the dot operator with the *. The `->` has the same priority and associativity (L-R) as the dot.

Arithmetic works in the usual manner for pointers to structures. Unlike an array, the name of a structure doesn't represent a pointer, so we can't use expressions like `rect1 + 1` or `rect1++`. However, if we point a compatible pointer `p` to an array of structures, `p++` advances the pointer to the next array element, i.e., the next structure. This property will be used in a project that follows soon.

14.9.1 `pointer_to_structure.c`: Accessing Structure Members

Program 14.10 uses a pointer `p` to access the members of a three-member structure named `employee`. After `p` is made to point to the structure variable `emp`, the members of `emp` are displayed using a mix of all the three access mechanisms discussed previously. Two of these mechanisms (using the * and `->`) are subsequently used to reset the values.

```
/* pointer2structure.c: Demonstrates multiple techniques of accessing
                        structure members using a pointer. */
#include <stdio.h>
#include <string.h>

int main(void)
{
   struct employee {
      short id;
      char name[30];
      int pay;
   } emp = {1024, "Steve Wozniak", 10000}, *p;
```

```
    p = &emp;
    printf("Emp-id: %hd, Name: %s, Pay: %d\n", emp.id, (*p).name, p->pay);
    (*p).id = 4201;                              /* Updates id */
    strcpy(p->name, "Steve Jobs");               /* Updates name */
    p->pay = 20000;                              /* Updates pay */
    printf("Emp-id: %hd, Name: %s, Pay: %d\n", p->id, p->name, p->pay);
    return 0;
}
```

PROGRAM 14.10: **pointer2structure.c**

```
Emp-id: 1024, Name: Steve Wozniak, Pay: 10000
Emp-id: 4201, Name: Steve Jobs, Pay: 20000
```

PROGRAM OUTPUT: **pointer2structure.c**

14.9.2 update_pay.c: Using a Pointer as a Function Argument

We passed one or more structures as function arguments to swap two structure members (Program 14.9) and compute the difference between two times (Program 14.8). This pass-by-value mechanism is safe; it protects the original structure from modification by the function. However, large structures can consume a lot of memory, so we may need to pass a pointer to a structure instead of the entire structure.

Program 14.11 uses the **update_pay** function which accepts a pointer to the structure variable `emp` and the revised value of `pay` as its two arguments. The function parameter, `f_emp`, is a copy of `&emp`, so `f_emp->pay` represents the value at the memory location of `emp.pay`. Updating a member's value using a pointer represents a memory-efficient technique because the function copies not the structure (36 bytes) but its pointer (4 bytes).

```
/* update_pay.c: Uses a pointer to a structure as a function argument
                 to update the structure. */
#include <stdio.h>
struct employee {
    short id;
    char name[30];
    int pay;
} emp = {1024, "Steve Wozniak", 10000};
void update_pay(struct employee *f_emp, int f_pay);
int main(void)
{
    printf("Old pay = %d\n", emp.pay);
    update_pay(&emp, 20000);
    printf("New pay = %d\n", emp.pay);
    return 0;
}
```

```
void update_pay(struct employee *f_emp, int f_pay)
{
   f_emp->pay = f_pay;                  /* Updates member pay of emp */
   return;
}
```

PROGRAM 14.11: **update_pay.c**

```
Old pay = 10000
New pay = 20000
```

PROGRAM OUTPUT: **update_pay.c**

14.9.3 time_addition.c: Using Pointers as Function Arguments

Program 14.12 features the **time_add** function which uses pointers to a structure of type time as its three arguments. The function adds the times represented by the dereferenced value of the first two arguments and saves the result in the third argument. The first two arguments have been declared with the **const** qualifier to protect them from inadvertent modification by the function.

```
/* time_addition.c: Uses pointers to a structure as function arguments
                    to compute the sum of two times. */
#include <stdio.h>

typedef struct time {
   short hours;
   short mins;
   short secs;
} TIME;

void time_add(const TIME *, const TIME *, TIME *);

int main(void)
{
   TIME t1, t2, t3;

   fputs("Enter the two times in hh:mm:ss format: ", stdout);
   scanf("%hd:%hd:%hd %hd:%hd:%hd",
         &t1.hours, &t1.mins, &t1.secs, &t2.hours, &t2.mins, &t2.secs);

   time_add(&t1, &t2, &t3);
   printf("Sum of the two times: %hd hours, %hd minutes, %hd seconds\n",
           t3.hours, t3.mins, t3.secs);

   return 0;
}
```

```
void time_add(const TIME *t1, const TIME *t2, TIME *t3)
{
   short temp, quot;
   temp = t1->secs + t2->secs; quot = temp / 60;
   t3->secs = temp % 60;
   temp = t1->mins + t2->mins + quot; quot = temp / 60;
   t3->mins = temp % 60;
   t3->hours = t1->hours + t2->hours + quot;
   return;
}
```

PROGRAM 14.12: **time_addition.c**

```
Enter the two times in hh:mm:ss format: 6:40:30 8:30:50
Sum of the two times: 15 hours, 11 minutes, 20 seconds
```

PROGRAM OUTPUT: **time_addition.c**

14.10 student_management.c: A PROJECT

Let's complete our study of structures by examining a small project involving a student database. Program 14.13 (shown here in two sections with 'a' and 'b' suffixes) uses three functions to add to, display and query an array of structures of type STUDENT. The array, which can hold data for up to 50 students, is partially populated by user input, followed by display of the data. Finally, the program queries the database for a specific roll number. The two program sections must be concatenated before use.

```
/* student_management.c: Manages student details using pointers
                         to structures as function arguments. */
#include <stdio.h>
#define COLUMNS 50
#define FLUSH_BUFFER while (getchar() != '\n') ;

typedef struct {
   char name[20];
   int roll_no;               /* Doesn't conflict with variable of same name */
   short marks[4];
} STUDENT;

void add_student(STUDENT f_stud[]);
float display_marks(STUDENT f_stud[]);
STUDENT *search_student(STUDENT f_stud[], int f_roll_no);

int main(void)
{
   int roll_no; float average;
   STUDENT stud[COLUMNS] = { {"", 0, {0, 0, 0}} };        /* Note the braces */
   STUDENT *q;
```

```
   add_student(stud);
   average = display_marks(stud);
   printf("Average marks of class = %.2f\n", average);
   fputs("\nEnter Roll No: ", stdout);
   scanf("%d", &roll_no);
   q = search_student(stud, roll_no);
   if (q != NULL)
      printf("%-20s %4d %5hd %5hd  %5hd %6hd\n", q->name, q->roll_no,
      q->marks[0], q->marks[1], q->marks[2], q->marks[3]);
   else
      fputs("Not found", stdout);
   return 0;
}
```

PROGRAM 14.13a: **student_management.c** (Pre-**main** and **main** sections)

```
void add_student(STUDENT p[])
{
   short i = 0;
   do {
      fputs("Name: ", stdout);
      if (scanf("%[^\n]", p->name) != 1)
         break;
      fputs("Roll No: ", stdout);
      scanf("%d", &p->roll_no);
      fputs("Marks in 3 subjects: ", stdout);
      scanf("%hd%hd%hd", &p->marks[0], &p->marks[1], &p->marks[2]);
            p->marks[3] = p->marks[0] + p->marks[1] + p->marks[2];
      FLUSH_BUFFER
      p++;                                  /* Moves pointer to next element */
  } while (i++ < COLUMNS - 1);              /* Won't accept more than 50 records */
   return;
}

float display_marks(STUDENT p[])
{
   short i; int total_marks = 0;
   fputs("Name              Roll_No  Phys  Chem  Maths  Total\n", stdout);
   for (i = 0; i < COLUMNS; i++, p++) {
      if (p->roll_no > 0) {
         printf("%-20s %4d %5hd %5hd  %5hd %6hd\n", p->name, p->roll_no,
         p->marks[0], p->marks[1], p->marks[2], p->marks[3]);
         total_marks += p->marks[3];
      }
      else
         break;
   }
   return (float) total_marks / i;          /* Returns average of total marks */
}
```

```
STUDENT *search_student(STUDENT p[], int f_roll_no)
{
   short i;
   for (i = 0; i < COLUMNS; i++, p++)
      if (p->roll_no == f_roll_no)
         return p;
   return NULL;
}
```

PROGRAM 14.13b: **student_management.c** (The function definitions)

```
Name: Sohini Gupta
Roll No: 1111
Marks in 3 subjects: 70 85 90
Name: Shiksha Agarwal
Roll No: 2222
Marks in 3 subjects: 55 75 60
Name: Juhi Parekh
Roll No: 3333
Marks in 3 subjects: 95 90 98
Name: [Enter]
Name              Roll_No  Phys  Chem  Maths  Total
Sohini Gupta         1111    70    85     90    245
Shiksha Agarwal      2222    55    75     60    190
Juhi Parekh          3333    95    90     98    283
Average marks of class = 239.33

Enter Roll No: 2222
Shiksha Agarwal      2222    55    75     60    190
```

PROGRAM OUTPUT: **student_management.c**

Pre-main and main Sections The array, stud, is first initialized by setting the first element to zero. (Setting only the first member of this element using {{""}}; would have been enough.) The program can now evaluate the number of utilized array elements by checking the roll_no field. The three functions (discussed next) are invoked sequentially.

Function Definitions All three functions use the parameter STUDENT p[] to represent the array of structures. However, p is actually a pointer to the beginning of the array, stud, defined in **main**. The functions use p for navigating the array; every invocation of p++ advances the pointer to the next array element.

The **add_student** function invokes **scanf** twice in a **do-while** loop to accept user data. Because **scanf** returns the number of items successfully read, pressing *[Enter]* without keying in data terminates this loop. The loop will otherwise terminate after 50 records have been entered. **add_student** also sets the fourth element of the marks array to the sum of the first three elements.

The **display_marks** function uses a **for** loop to display each record. Even though the loop can iterate 50 times, it terminates prematurely when roll_no is no longer a positive integer. The function also returns the average marks of the class. The **search_student** function searches for a roll number and returns a pointer to the matching array element if the number is found.

14.11 UNIONS

When searching a database using a person's id, you must have often wanted to search by name as well. That is possible if the two attributes are held in a *union*, a data type that resembles a structure in form. A union comprises two or more members that share the same memory, with the most recent assignment to a member being active at any instant. Consider the following declaration and definition of a union:

```
union uemp {                                    Declares union of type uemp
   short emp_id;
   char name[30];
} u;                                            Defines union variable u of type uemp
```

A union is declared, defined and accessed exactly like a structure. The previous statement declares a two-member union named `uemp` and simultaneously defines the union variable `u`. The members are accessed as `u.emp_id` and `u.name`. However, both members share a single memory region and you can *correctly* access only one member at a time—the one that was last assigned. For instance, you can assign and access `emp_id` in this way:

```
u.emp_id = 1234;
printf("%hd:\n", u.emp_id);                     Prints 1234
```

If you subsequently assign `u.name`, everything will work as expected:

```
strcpy(u.name, "George Martin");
printf("%s:\n", u.name);                        Prints George Martin
```

However, `u.name` overwrites the memory segment previously used by `u.emp_id`, so you can't access the latter now (even though the compiler won't complain):

```
printf("%hd:\n", u.emp_id);                     Output is undefined
```

Barring this essential difference, unions are handled like structures. You can separate the definition of a union from its declaration, and **`typedef`** works in identical manner on unions. You can create pointers to unions and also create arrays of unions. Because of these similarities, unions are most useful when they are used as members of structures or comprise structures as members.

Takeaway: The members of a union can't be accessed simultaneously because they share a single memory region. You can correctly access only the member that was last assigned.

14.11.1 Unique Attributes of Unions

Apart from the difference in the way structures and unions use computer memory, there are some unique features possessed by unions. The following features need to be kept in mind:

- *All members of a union can't be initialized simultaneously.* This is obvious because only one member is correctly accessible at any point of time. You can initialize only the first member:

```
union ucust u2 = {1234};                        Initializes only emp_id
```

- *The size of a union is determined by the size of the member having the largest footprint in memory.* For `uemp`, this would be 30, the size of the `name` field. As with structures, issues related to alignment on word boundaries *(14.3)* may increase this number.
- *The compiler doesn't output an error when you access a member that was not the last one to be assigned.* A programmer must keep track of the member that is active at any instant.
- *You can use a union to assign a value to a member and read it back using a different member.* For instance, you can store a string in a `char` array and then examine its bit pattern by reading the `int` member of the union.

A union can easily determine whether a machine is *little-endian* or *big-endian*. In a big-endian machine, the most significant bit is stored first while the reverse is true for little-endian machines. However, unions are mainly used as memory-saving objects.

Tip: It is often necessary to include a member that saves the type of information currently saved in the union. This value could be 0, 1, 2, etc., which could be used by the program to determine the currently active member.

14.11.2 `intro2unions`: An Introductory Program

Program 14.14 defines two union variables, `u1` and `u2`, based on the `ucust` template. `u1` is not initialized but `u2` is initialized with a single value which is automatically assigned to the first member (`aadhar_id`). The remaining two members are assigned later. Note that after `u2.name` has been assigned, `u2.aadhar_id` and `u2.cust_id` evaluate to incorrect values.

```
/* intro2unions.c: Demonstrates basic features of unions. */
#include <stdio.h>
#include <string.h>

int main(void)
{
   union ucust {                    /* Declares union of type ucust */
      int aadhar_id;                /* Only of these three members ... */
      short cust_id;                /* ... is active at any time. But ... */
      char name[30];                /* ... this one determines size of u1. */
   } u1;                            /* Defines union variable of type ucust */

   printf("Size of u1 is %d\n", sizeof(u1));

   union ucust u2 = {1234567890};           /* Initializes a second variable */
   printf("u2.aadhar_id = %d\n", u2.aadhar_id);

   u2.cust_id = 3456;
   printf("u2.cust_id = %hd\n", u2.cust_id);
   strcpy(u2.name, "Paul McCartney");
```

```
    printf("u2.name = %s\n", u2.name);
    printf("u2.cust_id = %hd\n", u2.cust_id);              /* Wrong */
    printf("u2.aadhar_id = %d\n", u2.aadhar_id);           /* Wrong */

    return 0;
}
```

PROGRAM 14.14: **intro2unions.c**

```
Size of u1 is 32
u2.aadhar_id = 1234567890
u2.cust_id = 3456
u2.name = Paul McCartney
u2.cust_id = 24912                                  Wrong
u2.aadhar_id = 1819631952                           Wrong
```

PROGRAM OUTPUT: **intro2unions.c**

14.12 BIT FIELDS

Have our programs wasted memory? Yes, in many cases they have. We used 2-byte `short` integers to store each component of a date when a few bits would have sufficed. We also wasted 15 bits of the 16 available in `short` to set a flag to 0 and 1. It's time to know that C supports a memory-saving feature—the bit field—which can be used with the right number of *bits*, rather than bytes, to store data.

C permits the division of one or more contiguous bytes of memory into multiple sets of contiguous bits. Each set of bits can be assigned to a *bit field*, which is represented as a member of a structure. One or two bits are adequate to represent the sex, while six or seven bits are good enough to represent the age. A bit field is normally of type `int` and the following structure shows two of them as members:

```
struct employee {
    unsigned int id : 10;                               A bit field
    unsigned int age : 6;                               Ditto
    char name[30];
} emp = {1001, 12, "John Lennon"};
```

The first two members, `id` and `age`, are bit fields. The declaration of a bit field specifies an `int` (signed or unsigned), followed by the member name, a colon and the number of allocated bits. Here, `id` and `age` occupy 10 and 6 bits, respectively. This means that the maximum values of `id` and `age` are restricted to 1023 and 63, respectively. **`sizeof`** evaluates the size of `emp` to 32. A similar structure using two `short` variables for `id` and `age` would occupy 34 bytes.

If a field can't be accommodated in the current word, the compiler moves the entire field to the next word. This leaves slack bytes in the previous word and this wastage can sometimes be reduced by proper design of the structure. However, you can modify the memory layout of the structure by making the following changes:

- Drop the name of a member to create an anonymous member. The bit size then indicates the number of slack bits to be formed:

```
unsigned int: 4;                    Creates 4 slack bits
```

- Explicitly specify a change of word by specifying 0 as the bit size:

```
unsigned int: 0;                    Moves remaining members to next word
```

  In this case, the remaining members will use the next word.

Bit fields like `emp.age` and `emp.name` behave like regular variables and can be assigned normally. However, **`scanf`** can't use a bit field because a pointer can't address a location that is not on a byte boundary:

```
emp.id = 900;                       OK
scanf("%d", &emp.age);              Wrong!
```

To input the age, use a temporary variable and then assign the input value to `emp.age`.

Bit fields were extensively used when memory was scarce and expensive. In those days, a saving of a few bits significantly impacted the design of programs. Today, 2GB RAM is a common feature in entry-level smartphones, so the importance of bit fields has been greatly reduced. However, there is a lot of legacy code that contain bit fields, so it would be a mistake to ignore this feature.

Takeaway: Bit fields can be used only as structure members. Properly sized bit fields prevent both data overflow and memory wastage.

14.13 THE ENUMERATED TYPE

We have used symbolic constants to represent integers using the **`#define`** feature of the preprocessor. C also supports the user-defined *enumeration* data type that assigns names to a set of integer values called *enumerators*. For instance, you can create a set of named constants using the **`enum`** keyword:

```
enum {YES, NO, CANT_SAY};           Enumeration without a name
```

This statement creates a set of enumerated integer values which, by default, begin with 0. `YES` evaluates to 0, while `NO` and `CANT_SAY` have the values 1 and 2, respectively. If this declaration is made before **`main`**, you can use `NO` to represent 1 anywhere in the program.

The previous declaration created an unnamed enumeration, but you can also create a named one. The following statement creates the enumerated data type named `eday`:

```
enum eday {SUN, MON, TUE, WED, THU, FRI, SAT};
```

Unlike a structure, every enumerated data type must be declared with a set of values which can't be changed subsequently. In the preceding declaration, the seven named constants represent the integers 0 to 6. You can now create one or more `enum` variables of type `eday`:

```
enum eday day1, day2;
```

Because day1 is of type eday, you can assign any of the seven symbolic constants to it:

```
day1 = WED;                                    day1 has the value 3
```

The **typedef** feature applies here, so you can create DAY as a synonym for enum eday and then create variables of this abbreviated data type:

```
typedef enum eday DAY;
DAY day1, day2;
```

If the default numbering sequence doesn't meet your requirements, you can explicitly assign a value to an enumerator using the = symbol. Thus, an enumerator acquires a value that is one more than the previous enumerator unless assigned explicitly. In the following example, THU and FRI acquire the values 4 and 5, respectively:

```
enum eday {MON = 1, WED = 3, THU, FRI, SUN = 7};
```

There are other features of enumerators that you can explore on your own. We'll now bring the curtains down on this chapter by examining a program that demonstrates the key features of enumerators. Program 14.15 considers an unnamed enumerator type and two named ones (eday and emonth). Observe how a variable of type eday is used as a subscript of an array to print the days of the week.

```
/* intro2enumerators.c: Demonstrates basic features of enumerators. */
#include <stdio.h>
int main(void)
{
   enum {YES, NO, CANT_SAY};                  /* Enumerator without a name */
   printf("YES = %d, NO = %d, CANT_SAY = %d\n", YES, NO, CANT_SAY);
   enum eday {SUN, MON, TUE, WED, THU, FRI, SAT};
   enum eday day = WED;                       /* Defines variable named day */
   printf("Numeric value of WED = %d\n", day);
   char *week_day[] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
   for (day = SUN; day <= SAT; day++)
      printf("%s ", week_day[day]);           /* Prints all 7 strings */
   enum emonth {JAN, MAR = 3, APR, JUL = 7, AUG, SEP, OCT, DEC = 12};
   printf("\nJAN = %d, MAR = %d, APR = %d, AUG = %d, SEP = %d\n",
           JAN, MAR, APR, AUG, SEP);
   return 0;
}
```

PROGRAM 14.15: **intro2enumerators.c**

```
YES = 0, NO = 1, CANT_SAY = 2
Numeric value of WED = 3
Sun Mon Tue Wed Thu Fri Sat
JAN = 0, MAR = 3, APR = 4, AUG = 8, SEP = 9
```

PROGRAM OUTPUT: **intro2enumerators.c**

Tip: For applications that use |, the bit-wise OR operator, it often helps to choose enumerators that are a power of 2. For instance, if you choose values like BOLD = 1, ITAL = 2, UNDERLINE = 4, then you can choose more than one of these values in an expression. The expression `BOLD | ITAL | UNDERLINE` evaluates to 7 and represents all three display attributes. Bit-wise operators are discussed in Chapter 17.

WHAT NEXT?

Structures and unions are used to handle large amounts of data but not in the ways discussed in this chapter. Structured data are commonly obtained, not from the keyboard, but from files. We must know how to read and write files.

WHAT DID YOU LEARN?

A *structure* is a user-defined data type meant to hold related data items having multiple data types. It is declared to create a *template* based on which variables are defined.

The **`typedef`** statement allows structure data types to be abbreviated to short and meaningful names.

A structure variable can be assigned to another of the same type. However, no arithmetic, relational and logical operations can be carried out on structures.

A structure can contain an array or another structure as members. C also supports arrays of structures.

When the name of a structure is passed as argument to a function, the entire structure is copied inside the function.

C supports a pointer to a structure whose members can be connected to the pointer with the -> symbol.

A *union* resembles a structure in form but its members share the same memory storage. A union allows the interpretation of data in multiple ways.

A structure member can be a *bit field* which has its size specified in bits. Properly sized bit fields can lead to savings in memory.

The *enumeration* data type allows named constants to be automatically assigned integer values beginning with 0. Enumerations improve the readability of programs.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

14.1 A member of a structure can point to another structure of the same type.

14.2 Like a variable, a structure is always simultaneously declared and defined.

14.3 It is not possible to declare a structure without a structure tag.

14.4 The assignment `struct movie s1 = s2;` is legal only if `s1` and `s2` are based on the same template.

14.5 An uninitialized structure has all of its members set to zeroes.

14.6 The size of a structure is not always equal to the sum of the sizes of its members.

14.7 It is possible to have the same name for a structure tag, structure variable and a member variable.

14.8 A function argument can be a structure but not its pointer.

14.9 It is not possible to create a separate variable of an inner structure.

14.10 It is possible for a function to copy an array when it is passed as a member of a structure.

14.11 The name of a structure represents a pointer to the first member.

14.12 It is not possible to simultaneously access all members of a union.

14.13 A union uses more memory that its corresponding structure.

14.14 A member of a structure cannot be a bit field.

14.15 It is not possible to change the value of an enumerator.

A2. FILL IN THE BLANKS

14.1 A structure is meant to store _____ data.

14.2 A structure is a _____ data type.

14.3 The `int` data type can be abbreviated to `INT` using _____.

14.4 A member of a _____ structure contains a reference to another identical structure.

14.5 The members of a _____ share the same region of memory.

14.6 Multiple instances of a structure are conveniently stored in an _____.

14.7 The size of a union is determined by the member having the _____ _____.

14.8 A member of a structure containing one or more bits is known as a _____ _____.

14.9 The _____ keyword is used to create a set of named constants.

A3. MULTIPLE-CHOICE QUESTIONS

14.1 Definition of a structure (A) allocates memory for a structure variable, (B) creates a template, (C) A and B, (D) none of these.

14.2 The following declaration is (A) valid, (B) valid but doesn't allow creation of a second variable, (C) valid but doesn't allocate memory, (D) not valid.

```
struct {
  char x;
} x;
```

14.3 For comparing two structures, the expression `s1 == s2` is (A) legal, (B) illegal, (C) legal only if `s1` and `s2` are based on the same template, (D) legal only if `s2` is a template.

14.4 A member of a structure can (A) contain a reference to another structure of any type, (B) contain a reference to another structure of the same type, (C) be another structure of any type, (D) A and B, (E) none of these.

14.5 If two structure variables `s1` and `s2` have identical members, `s1` and `s2` (A) may belong to different templates, (B) must be based on the same template, (C) can be assigned to each other as `s1 = s2;`, (D) A and C.

14.6 If pointer `p` points to a structure variable `s` containing members `m1` and `m2`, then `m1` can be accessed as (A) `(*p).m1`, (B) `p->m1`, (C) A and B, (D) none of these.

CONCEPT-BASED QUESTIONS

14.1 What is the difference between the dot and -> operators?

14.2 Explain with an example how *slack bytes* are created.

14.3 Explain how the use of **`typedef`** makes programs portable across machines that have different sizes for the same data type.

14.4 Explain with an example why one should use a nested structure wherever possible.

14.5 When passed as an argument to a function, what advantage does a structure have over an array?

14.6 Explain with an example how a function can return multiple values without using pointers as arguments.

14.7 Name two features of a union that are absent in a structure.

PROGRAMMING & DEBUGGING SKILLS

14.1 Point out the error in the following code:

```
struct s {
   float f;
   char c[10];
};
```

14.2 Correct the errors in the following code segment:

```
struct cd {
   char title[20];
   short qty;
}
cd.qty = 3;
strcpy(title, "Woodstock69");
```

14.3 Create a structure for a smartphone that contains its model name (20 characters), RAM size (in GB) and price (integer). Create two objects from this template, populate its members from the keyboard and print their values.

14.4 Modify the preceding program *(C14.3)* to initialize one variable at the time of its definition and populate the other by later assignment.

14.5 Modify a preceding program *(C14.3)* to accept all six values from command-line arguments. (HINT: Use the **`atoi`** function.)

14.6 Correct the errors in the following program:

```
#include<stdio.h>
int main(void)
{
   struct country {
      char name[30];
      long population;
      float area;
   } c = ("Burundi", 0, 12345)
   fputs(stderr, "Enter population: ");
   scanf("%ld", c->population);
   printf("Country: %s, Population: %d, Area: %f\n",
           c.name, c.population, c.area);
   return 0;
}
```

14.7 Write a program that creates two variables of type `country` (declared in C14.6) but without using a structure tag. Initialize one variable but populate the other with user input. What is the disadvantage of using this technique?

14.8 Using the `country` structure declared in C14.6, write a program that uses a function to populate a variable of this type. Does the location of the structure declaration need to be changed?

14.9 Why does the following code segment generate a compilation error?

```
struct auto {
   char model[30];
   short price;
} a1 = {"Maruti", 410000}, a2 = {"Hyundai", 350000};
```

14.10 Point out the errors in the following code:

```
typedef struct camera {
   char model[30];
   unsigned short price;
} C;
C.model = "Canon";
C.price = 32000;
```

14.11 Correct the following code segment first:

```
struct phone {
   char model[30];
   char memory;
```

```
   struct resolution {
      short horizontal;
      short vertical;
   } res;
} p;
```

Next, write a program that assigns the values "Galaxy Note 9" and 8 for the first two members, and 1920 and 1080 for the inner structure. Print all values.

14.12 Modify the preceding program *(C14.11)* to include a pointer q that points to p and then print the same values using q with both * and -> notations.

14.13 Modify the function devised in C12.10 *(Chapter 12)* to return the downloaded time as a 3-member structure. How does this solution differ from the array-based technique?

14.14 Write a program that accepts three parameters of a hard disk (heads, cylinders and sectors/track) from user input and passes them as a structure to a function. The function must return the capacity as a 3-member structure (GB, MB and KB). Print the value as a single floating point number.

14.15 Write a program that accepts the details of five mutual fund schemes into an array of structures. Each structure comprises the fund name (30 characters), fund code (short) and the current Net Asset Value (NAV—a float). Using keyboard input, update the NAV of a specific fund using a function that accepts three arguments (fund code, pointer to the array and latest NAV).

14.16 Write a program that creates a union of a person's name and age. Fill up both members using user input and then print the values. Does it matter if the input sequence is reversed?

14.17 Using a union, write a program that extracts each byte from an unsigned int and prints them as separate characters.

14.18 Point out the error in the following code:

```
struct any_struct {
   short s : 20;
   int i : 4;
}
```

14.19 Locate the errors in the following program so it prints X.x correctly. For X.x to have the value 300, what changes are required to be made in the program?

```
#include <stdio.h>
int main(void)
{
   struct x {
     short x : 4;
   } X;
   X.x = 8;
   printf("sizeof x = %hd\n", sizeof(x));
   printf("x.x = %hd\n", X.x);
   return 0;
}
```

14.20 Write a program to store a date in an *optimized* structure that includes *properly-sized* bit fields. The program accepts a date as three integers in the form *dd mm yyyy* and prints the values using the structure.

14.21 Correct the following code so as to print 5 for `MAY` and 12 for `DEC`:

```
enum E {JAN, FEB, MAR, APR, MAY, JUN, DEC};
E month;
month = MAY;
printf("%d\n", month);
month = DEC;
printf("%d\n", month);
```

14.22 Write a program that assigns the values 1, 2 and 4, representing read, write and execute permissions, to an enumeration. With valid user input, the program must print a message, say, `"File has read permission."`

15 File Handling

WHAT TO LEARN

- Significance of the *file pointer* and *buffer* in file-I/O operations.
- Opening a file in read, write and append modes (**`fopen`**).
- Handling I/O errors (**`perror`**) and detecting EOF (**`feof`**).
- How functions use the *file position indicator* in I/O operations.
- Handling data in units of characters (**`fgetc`**/**`fputc`**) and lines (**`fgets`**/**`fputs`**).
- Formatting data with **`fscanf`** and **`fprintf`**.
- Using command-line arguments to specify filenames.
- Read/write operations on arrays and structures (**`fread`**/**`fwrite`**).
- Random file access using the file position indicator (**`fseek`**/**`ftell`**).
- Deleting and renaming files (**`remove`** and **`rename`**).

15.1 A PROGRAMMER'S VIEW OF THE FILE

So far, our programs used data that was either provided in the program itself or input from the keyboard. The output always appeared on the terminal. Unsaved input and output are totally unsuitable for handling voluminous data that we often encounter in real life. You can't obviously key in the details of every employee for preparing the payroll, and you certainly can't do it every month. The most convenient mechanism for handling bulk data is to store them in files and use programs to read from and write to these files.

A file is a *named* container of information that is stored in multiple *blocks* in a file system. This file system typically resides in the hard disk but a CD-ROM, DVD-ROM or flash drive also support a file system. All files are administered by the operating system which allocates and deallocates disk blocks as and when needed. The OS also controls the transfer of data between programs and the files they access. Thus, a file-oriented program has to make a call to the OS for a service that it can't handle on its own.

When we invoked **a.out < foo1 > foo2** from the shell *(9.6)*, the OS opened the two files, gathered their scattered blocks and connected their data to the **a.out** program. This mechanism has some drawbacks. First, a program can use more than two files. Second, read/write operations don't always start from the beginning of the file. Finally, the same file could be both read and written. These situations can only be handled by the I/O functions offered by the standard library.

A file comprises a set of byte-sized characters, where each character can have any value between 0 and 255. If we needed to have only a character view of a file, only one read and one write function would have been adequate for read/write operations. However, programs need to know whether a group of characters represents a string or a number, so multiple functions are needed for making this distinction. Partly for this reason, we often classify files as *binary* or *text*, even though this distinction has never been fully convincing.

Note: The file-I/O functions have no knowledge of the organization of a file's data on disk. Neither are they aware of the mechanism employed by the OS to transfer data between the program and a file.

15.2 FILE-HANDLING BASICS

A file has to be *opened* before data can be read from or written to it. Typically, data is read from a file and then manipulated by program logic. The output is often written to another file. Because different functions can read the same file, a common *buffer* and a *file position indicator* are maintained in memory for a function to know how much of the file has already been read. C also supports functions that handle issues like EOF and file-related errors.

Only the **fopen** function needs to know the name and location of a file. **fopen** returns information on the opened file in the form of a *file pointer* which points to a structure of type FILE. The other functions use this file pointer to access the information stored in the structure. This information includes the file position indicator and the location of the buffer that are associated with each opened file.

The buffer is used as temporary storage by the I/O-bound functions like **fgetc** and **fputc**. Even though **fgetc** reads one character at a time, it would be inefficient to access the disk every time a character is read. The OS reduces this overhead by reading a block of data into the buffer from where **fgetc** fetches a character at a time. A similar buffer is set up when a file is opened for writing. This buffer is written to disk when it is full or when specifically instructed to do so.

Even though files simply contain characters, it is often necessary to group these characters into words, lines, arrays and structures. The **fgets** function fetches a line while **fread** reads a structure or even an array of them in one call. *Text* files contain only printable characters while *binary* files may contain any character. It is sometimes necessary for a function to know the type of file it is dealing with, so the type has to be specified as text or binary when the file is opened.

C was born in a UNIX environment which uses the LF character (ASCII value: 10) as newline. However, MSDOS/Windows uses CR-LF (two characters) as newline, while Mac uses only CR. For this reason, some of the library functions have to carry out the necessary replacement when newline is read or written on these systems.

Note: All file-handling functions discussed in this chapter use the file `stdio.h`, which must be included in every program containing one of these functions.

15.3 OPENING AND CLOSING FILES

File handling begins with the opening of a file and ends with its closing. A file can be opened in several modes, and we must know the exact significance of each mode. Opening a file in the wrong mode can lead to errors—or even disaster. File closing is not mandatory for most situations, even though it is a good idea to explicitly close a file when you no longer need it.

When a file is opened, the OS associates a *stream* with the file. Because C treats all devices as files and device characteristics vary, it is logical that read/write operations are performed on the stream rather than to the device directly. The OS connects one end of the stream to the physical file and the other end to the program. This means that any read/write operation on the stream actually accesses the file on disk. The stream is disconnected when the file is closed.

15.3.1 fopen: Opening a File

Prototype: `FILE *fopen(`*pathname*`,` *mode*`);`
Return value: A pointer to `FILE`, or NULL on error.

The **`fopen`** function needs two strings as arguments—the pathname of the file and its mode of opening. The *pathname* can be an absolute one (like `"/data/foo.txt"`) or a simple one (`"foo.txt"`). The *mode* is usually `"r"`, `"w"` or `"a"` for simple read, write and append operations, but we'll also learn to use the `+` suffix for read/write operations. The following code shows the simplest way of using **`fopen`**:

```
FILE *fp;                                  Defines file pointer
fp = fopen("foo.txt", "r");                Only reading permitted
```

If the call is successful, **`fopen`** returns a pointer to a structure typedef'd to `FILE`, so the pointer needs to be defined first. The file is opened for reading (`"r"`); you can't write to this file. The pointer variable, `fp`, assigned by **`fopen`** now acts as a file handle which will be used subsequently by all functions that access the file. Once a file has been opened, its pathname is not required any longer.

15.3.2 File Opening Modes

We opened the file `foo.txt` for reading, but **`fopen`** supports a number of options to represent the mode. These options enable you to write, append and also update a file (Table 15.1). For the time being, you need to keep in mind the following modes:

- `r` Reading a file.
- `w` Writing a file.
- `a` Appending to the end of an existing file.

`fopen` used with `"r"` as the mode expects the file to exist, but when used with `"w"`, it overwrites an existing file or creates one if the file doesn't exist. When used with `"a"`, **`fopen`** also creates a file if

it doesn't find one. **fopen** will fail if the file doesn't have the necessary permissions (read for "r" and write for "w" and "a").

Database applications often need both read and write access for the same file, in which case, you must consider using the "r+", "w+" and "a+" modes. However, using "w+" will overwrite a file, so to preserve the contents of a file, use either "r+" or "a+", depending on whether you need write access for the entire file or only at its end.

All of the modes that we have discussed apply to text files which are organized as lines terminated by the newline character used by the operating system. We'll postpone discussions on the "b" suffix to Section 15.13 which deals with binary files.

Takeaway: The filename is used only by **fopen**, but other functions use the FILE pointer returned by **fopen**.

Tip: The "w" mode always overwrites an existing file, so if you need to use an existing file for read/write access, use r+ as the mode. If you use a+, you can read the entire file all right, but you can't change the existing contents.

TABLE 15.1 File Modes Used by **fopen**

Mode	*File Opened for*
r	Reading only. Error if file doesn't exist.
r+	Both reading and writing. Error if file doesn't exist.
w	Writing only. If file exists, then it is overwritten, else it is created.
w+	Both reading and writing. If file exists, then it is overwritten, else it is created.
a	Appending only. Creates file if it doesn't exist.
a+	Reading entire file but only appending permitted. Creates file if it doesn't exist.
b (*prefix*)	Same as their non-prefixed counterparts, except that binary mode is used.

15.3.3 The Filename

The *pathname* shown in the prototype can either be a simple filename like "foo.txt", or a complete pathname like "/project2/data/foo.txt". The former is used when the file exists in the same directory from where the program is run. But that is often not the case, in which case the absolute pathname must be used.

The preceding pathname is used by UNIX systems where C first made its appearance, but MSDOS/Windows systems use the \ for delimiting the components of the pathname. Because C uses the \ for escape sequences and Windows also uses a drive name, you would have to use a pathname like "C:\\project2\\data\\foo.txt" for **fopen** to locate a file on Windows. So, C takes a hit here as far as portability goes.

Instead of hard-coding the filename in the program, we often need to take the name from user input using code similar to the following:

```
FILE *fp1;
char fname1[40];                                    For storing the filename
fputs("Enter name of file for reading: ", stderr);
scanf("%s", fname1);
fp1 = fopen(fname1, "r");                           File must exist
```

Another frequently used technique is to accept the input filename from the keyboard and then use a suffix, say, ".out", for the output filename. The **strcpy** and **strcat** functions easily achieve this task:

```
strcpy(fname2, fname1);
strcat(fname2, ".out");
fp2 = fopen(fname2, "w");                           Existing file overwritten
```

Thus, if the input filename (fname1) is foo, the output filename (fname2) is set to foo.out. Filenames are also provided by command-line arguments, and in this chapter, you'll encounter some programs that employ this third technique.

Takeaway: In a program, a filename can be (i) hard-coded in the program, (ii) obtained from user input, (iii) derived from another filename, (iv) provided as a command-line argument.

15.3.4 Error Handling

The preceding examples used **fopen** to assign a file pointer variable without performing any error-checking. But an attempt to open a file can fail for a number of reasons including the following:

- The file may not exist.
- The file not be readable when the "r" mode is used.
- The file may not be writable when the "w" or "a" modes are used.
- The number of open files has reached the maximum permitted value.

There's no point in continuing with program execution if a key file can't be opened. **fopen** must always be used with error-checking code:

```
fp = fopen("C:\\data\\foo.txt", "r");
if (fp == NULL) {
   fputs("Error opening file for reading.\n", stderr);
   exit(1);
}
```

As we have previously done with **getchar** and **scanf**, the assignment to the file pointer and checking for NULL can be combined:

```
if ((fp = fopen("C:\\data\\foo.txt", "r")) == NULL)
   fputs("Error opening file for reading.\n", stderr);
```

How do you quit a program if **fopen** is used inside a function? The **return** statement won't help because it will transfer control to the caller (often **main**), so we need to use the **exit** function which terminates a program instantly after closing all open files. This function, which uses stdlib.h, is used in all programs featured in this chapter.

15.3.5 fclose: Closing a File

Prototype: `int fclose(FILE *fp);`
Return value: 0 on success, EOF on failure.

Operating systems have a limit on the number of files that can be opened by a program. Files must, therefore, be closed with the **fclose** function once they are done with. **fclose** uses the file pointer as a single argument:

```
fclose(fp);                                    fp previously declared as FILE *fp;
```

Closing a file frees the file pointer and associated buffers. Before a file is closed, any data remaining in the output buffer is automatically written (flushed) to disk. In most cases, using **fclose** is not mandatory because files are automatically closed when the program terminates. However, it's good programming practice to "fclose" all files when you no longer need them.

Using **fclose** followed by an immediate **fopen** of the same file is a convenient way of moving the file position indicator to the beginning of the file. This repositioning is better achieved with the **rewind** function. However, if you have opened a file using "r" as the mode, and you now want to write it using "r+", you *must* close and reopen the file.

15.3.6 fopen_fclose.c: An Introductory Program

Program 15.1 opens and closes the file foo twice—first for writing, and later for reading. A string input from the keyboard is written to foo and then read back for display using **fgets** and **fputs**. You had used the same functions for performing I/O with the terminal *(13.4.3)*. This program demonstrates that stdin and stdout are streams that are connected to different devices. **fgets** and **fputs** are revisited in Section 15.6.

```
/* fopen_fclose.c: Uses fgets and fputs for performing I/O with both
                   terminal and file. */
#include <stdio.h>
#include <stdlib.h>                  /* For exit */

int main(void)
{
   FILE *fp;                         /* Declares file pointer */
   char buf1[80], buf2[80];

   fputs("Enter a line of text:\n", stdout);
   fgets(buf1, 79, stdin);           /* Space left for '\0' */

   fp = fopen("foo", "w");           /* Creates and opens file for writing */
   fputs(buf1, fp);                  /* Writes input string to foo */
   fclose(fp);
```

```
    fp = fopen("foo", "r");           /* Opens foo for reading */
    fgets(buf2, 79, fp);              /* Reads a line from foo */
    fputs(buf2, stdout);              /* Writes line to display */
    fclose(fp);

    exit(0);
}
```

PROGRAM 15.1: **fopen_fclose.c**

```
Enter a line of text:
Characters, words, lines and structures
Characters, words, lines and structures                    Output by fputs
```

PROGRAM OUTPUT: **fopen_fclose.c**

15.4 THE FILE POINTER AND FILE BUFFER

Opening a file with **fopen** has two important consequences—creation of a file pointer and file buffer. The file pointer (like `fp` in previous programs) is a variable that points to a structure typedef'd to `FILE`. This structure, which is instantiated when a file is opened, contains the following information related to the opened file:

- The mode of opening.
- The location of the file buffer.
- The file position indicator.
- Whether EOF or any errors have been encountered on reading or writing.

The *file position indicator*, also called *file offset pointer*, has an important role to play in input/output operations. For instance, a read operation updates this field in the `FILE` structure with the position of the buffer up to which all data have been read. The next read operation (by the same or another function) looks up this field to determine the position from where to resume reading. The same logic applies to write operations.

File opening also sets up a buffer which acts as a go-between between the program and the file. When a file is read with, say, **fgetc**, the data is first sent from disk to the buffer from where it is sent to the program. So, even if **fgetc** reads one character from the buffer in each call, the buffer itself is populated in one disk-read operation. This speeds up the eventual data transfer rate from disk to the program.

A similar buffer is set up when a file is opened for writing. In this case, all write operations are made to the buffer, which itself is flushed (i.e., written) to disk when full. A file used for read/write access thus needs two buffers. Figure 15.1 replicates Figure 9.1 which demonstrated the role of these buffers in terminal I/O).

Takeaway: All read/write operations on files actually take place through their buffers. A buffer is connected to a disk file by the file pointer which is generated when a file is opened with **fopen**.

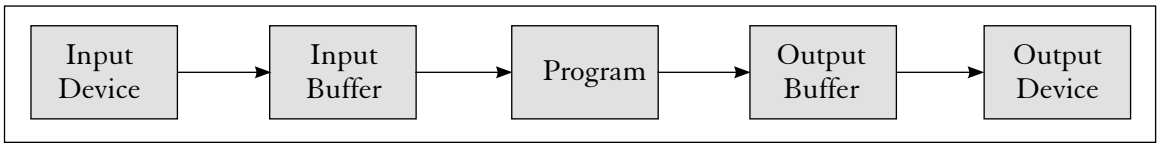


FIGURE 15.1 The File Buffers

Note: The standard streams, `stdin`, `stdout` and `stderr`, which can be used as arguments to file-handling functions like **`fgets`** and **`fputs`**, are actually pointers of type `FILE`. However, every program always finds these files open even though it can explicitly close them.

15.5 THE FILE READ/WRITE FUNCTIONS

The standard library offers a number of functions for performing read/write operations on files. All of these functions use the file pointer as argument and can thus work with any file including `stdin`, `stdout` and `stderr`. In this chapter, we'll discuss the following functions:

- Character-oriented functions (**`fgetc`** and **`fputc`**)
- Line-oriented functions (**`fgets`** and **`fputs`**)
- Formatted functions (**`fscanf`** and **`fprintf`**)
- Array- and structure-oriented functions (**`fread`** and **`fwrite`**)

The functions belonging to the first two categories have been used with the standard streams in previous chapters. The functions of the third category (**`fscanf`** and **`fprintf`**) behave exactly like their terminal-based counterparts (**`scanf`** and **`printf`**). The last category needs detailed examination because **`fread`** and **`fwrite`** work with binary files.

All of these functions use the file `stdio.h`. They also read or write a buffer *(15.2)* that is created when the file is opened with **`fopen`**. Calls to these functions can be mixed with one another without causing conflict. Each call moves the file position indicator in the buffer so the next call resumes from where the previous call left off. This assertion needs vindication by a suitable program.

15.5.1 `mixing_functions.c`: Manipulating the File Offset Pointer

Program 15.2 uses **`fputs`** to write a string stored in the `char` array, `buf`, to the file `foo`. The contents of `foo` are then retrieved using three separate functions (**`fgetc`**, **`fscanf`** and **`fgets`**) and written to the standard output. Each write is done by the function corresponding to its read counterpart. The program uses the **`rewind`** function to set the file offset pointer to the beginning of the buffer. This function will be examined in Section 15.14.2.

```
/* mixing_functions.c: Uses 3 sets of I/O functions to demonstrate the
                       significance of the file indication indicator. */
#include <stdio.h>
#include <stdlib.h>                          /* For exit */
```

```
int main(void)
{
   FILE *fp;
   int c, x;
   char buf[80] = "A1234 abcd";
   char stg[80];

   fp = fopen("foo", "w+");                  /* File will also be read */
   fputs(buf, fp);                           /* Writes buf to foo */

   rewind(fp);                               /* At position 0 in foo */
   c = fgetc(fp);
   fputc(c, stdout);                         /* Prints "A" */

   fscanf(fp, "%d", &x);
   fprintf(stdout, "%d", x);                 /* Prints "1234" */

   fgets(stg, 79, fp);
   fputs(stg, stdout);                       /* Prints " abcd" */

   exit(0);
}
```

PROGRAM 15.2: **mixing_functions.c**

```
A1234 abcd
```

PROGRAM OUTPUT: **mixing_functions.c**

15.5.2 The fgetc and fputc Functions Revisited

Prototype: `int fgetc(FILE *`*stream*`);`
Return value: The character read on success, EOF on error or end-of-file.

Prototype: `int fputc(int` *c*`, FILE *`*stream*`);`
Return value: The character written on success, EOF on error.

The **fgetc** and **fputc** functions have been used in Section 9.7.1 to interact with the terminal using `stdin` and `stdout`, respectively, as the file pointers. **fgetc** accepts the file pointer *stream* as a single argument. It fetches and returns a character from the buffer associated with *stream*. This is how the function is typically used:

```
int c;
c = fgetc(fp);                          fp returned by a prior fopen
```

Here, **fgetc** returns a character from the buffer associated with `fp`. To read an entire file, **fgetc** must be used in a loop which terminates when **fgetc** detects EOF:

```
while ((c = fgetc(fp)) != EOF)          EOF flag set in FILE structure
```

The **fputc** function uses an additional argument representing the numeric value of the character written to *stream*. The following statement writes the character c to the file associated with fp2:

```
fputc(c, fp2);
```

Even though **fputc** returns the character written, we normally use the function only for its side effect. However, error-checking in **fputc** is made by comparing the return value with its first argument.

Note: The macros **getc** and **putc** are equivalent to **fgetc** and **fputc**, respectively, and use the same syntax. But **getchar** and **putchar** can be used only with stdin and stdout, respectively. In fact, **getchar()** is equivalent to **getc(stdin)** and **putchar(c)** is equivalent to **putc(c, stdout)**.

15.5.3 file_copy.c: A File Copying Program

Program 15.3 copies a file whose name is input by the user. The output filename is generated by appending the ".out" extension to the input filename. After successful validation of file opening, the file foo is copied to foo.out by using **fgetc** and **fputc** in a loop. Each character fetched by **fgetc** from the file associated with the file pointer fp1 is written by **fputc** to the file associated with fp2.

```
/* file_copy.c: Copies and displays file contents using fgetc and fputc.
                        Also uses the exit function instead of return. */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
   FILE *fp1, *fp2;
   char flname1[80], flname2[80];
   int c;

   fputs("Enter filename to be copied: ", stderr);
   scanf("%[^\n]", flname1);
   strcpy(flname2, flname1);
   strcat(flname2, ".out");

   if ((fp1 = fopen(flname1, "r")) == NULL) {
      fputs("Error opening file for reading.\n", stderr);
      exit(2);                    /* Returns 2 to the operating system */
   }
   if ((fp2 = fopen(flname2, "w")) == NULL) {
      fputs("Error opening file for writing.\n", stderr);
      exit(3);                    /* Returns 3 to the operating system */
   }

   while ((c = fgetc(fp1)) != EOF)
      fputc(c, fp2);
```

```
   fclose(fp1);
   fclose(fp2);

   exit(0);
}
```

PROGRAM 15.3: **file_copy.c**

```
Enter filename to be copied: foo                    foo copied to foo.out
```

PROGRAM OUTPUT: **file_copy.c**

Note: It makes sense to send diagnostic messages to the *stderr* stream rather than standard output. This has been done in the previous program. Systems like UNIX and MSDOS support redirection of the standard output stream using > and >>, and these diagnostic messages must not be included with the useful output.

15.5.4 file_append.c: A File Appending Program

Program 15.4 appends the file `bar` to `foo` after obtaining both filenames from the keyboard. The file appended to must be opened in the `"a"` mode. Validation of file opening is combined (using ||), so an error will only indicate that one of the files created a problem, but not which one. We also have a feel of the **perror** function which specifies the exact cause of failure. The error occurred in the first invocation, and **perror** tells us that one of the files doesn't exist. This function is discussed in Section 15.10.

```
/* file_append.c: Appends contents of one file to another.
                  Introduces the perror function to specify the error. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *fp1, *fp2;
   char flname1[80], flname2[80];
   int c;

   fputs("Enter source and destination filenames: ", stderr);
   scanf("%s %s", flname1, flname2);

   fp1 = fopen(flname1, "r");
   fp2 = fopen(flname2, "a");          /* File to be appended to */

   if (fp1 == NULL || fp2 == NULL) {
      fputs("Error in opening at least one file.\n", stderr);
      perror("");                      /* Specifies exact cause of failure */
      exit(2);
   }
```

```
   while ((c = fgetc(fp1)) != EOF)
     if (fputc(c, fp2) == EOF) {
        fputs("Error in writing file.\n", stderr);
        exit(3);
     }
   fclose(fp1); fclose(fp2);
   exit(0);
}
```

PROGRAM 15.4: **file_append.c**

```
Enter source and destination filenames: bar foo
Error in opening at least one file.
No such file or directory                          Message from perror

Enter source and destination filenames: foo bar
```

PROGRAM OUTPUT: **file_append.c**

15.6 THE fgets AND fputs FUNCTIONS REVISITED

Prototype: `char *fgets(char *s, int size, FILE *stream);`
Return value: *s* on success, NULL on error or when EOF is encountered before reading.

Prototype: `int fputs(const char *s, FILE *stream);`
Return value: A non-negative number on success, EOF on error.

The line-oriented **fgets** and **fputs** functions. which were examined in Section 13.4.3, also work with disk files. **fgets** reads at most *size* - 1 characters from *stream*, saves the characters in the buffer *s* and appends `'\0'` to the buffer. If a newline is encountered before the buffer size limit is encountered, **fgets** stops reading and stores the newline in the buffer. This is how the function is typically used:

```
char buf[80];
fgets(buf, 80, fp) != NULL);                    fp returned by fopen
```

Because of the presence of the terminating `'\0'`, `buf` is a string. If the string is shorter than the size of `buf`, the latter will also contain the newline character. This character has to be removed from `buf` for string handling operations to be possible.

The **fputs** function writes the buffer *s* to the file associated with stream. The following statement writes the contents of `buf` to the file associated with `fp`:

```
fputs(buf, fp);
```

fputs strips off the NUL character but doesn't add a newline character to the output. This appears reasonable because **fputs** often outputs a line fetched by **fgets** which generally contains a newline.

Note: Unlike **fputs**, the **puts** function appends a trailing newline to the output. **puts** is normally not used with **fgets** because it creates double-spaced lines and it works only with stdout.

15.6.1 fgets_fputs2.c: Using fgets and fputs with a Disk File

Program 15.5 uses **fgets** to fetch a few lines from keyboard input. **fputs** writes these lines to the file foo. After **rewind** resets the file position indicator for foo to the beginning, **fgets** reads this file. The **rewind** function, which often eliminates the need to close and reopen a file, is discussed in Section 15.14.2.

```
/* fgets_fputs2.c: Saves and reads back user input to and from a file. */

#include <stdio.h>
#include <stdlib.h>
#define BUFSIZE 80

int main(void)
{
   FILE *fp;
   char buf[BUFSIZE];

   if ((fp = fopen("foo", "w+")) == NULL) {     /* File will also be read */
      perror("");
      exit(2);
   }

   fputs("Enter a few records, [Ctrl-d] to exit\n", stderr);
   while (fgets(buf, BUFSIZE, stdin) != NULL)
      fputs(buf, fp);

   rewind(fp);                          /* Positions indicator at beginning */

   fputs("\nReading from foo...\n", stderr);

   while (fgets(buf, BUFSIZE, fp) != NULL)
      fputs(buf, stdout);

   exit(0);
}
```

PROGRAM 15.5: **fgets_fputs2.c**

```
Enter a few records, [Ctrl-d] to exit
fgets must be used in place of gets.
fputs should be used as the matching counterpart of fgets.

Reading from foo...
fgets must be used in place of gets.
fputs should be used as the matching counterpart of fgets.
```

PROGRAM OUTPUT: **fgets_fputs2.c**

15.6.2 save_student_data.c: Writing Data Stored in Array to a File

Program 15.6 uses **fputs** in a loop to save five strings to a file. The pointers to the strings are originally stored in an array of pointers, and each string comprises three colon-delimited fields. After the file is "rewound," each line is fetched by **fgets**, parsed into three separate fields by **sscanf**, and then finally displayed in formatted form by **printf**. It would be helpful to recall the power of the **fgets-sscanf** combination in splitting a string into individual fields *(13.5.1)*.

```
/* save_student_data.c: (i) Saves records stored in an array of pointers.
                        (ii) Splits each record into fields when reading file. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 80

int main(void)
{
   FILE *fp;
   short i, size, roll_no, marks;
   char buf[BUFSIZE], name[20];

   char *student[] = { "Juhi Agarwal:11:97", "Paridhi Jalan:22:80",
                       "Shiksha Daga:33:92", "Paul Oomen:44:100",
                       "Abhijith:55:96" };    /* Each record has 3 fields */

   size = sizeof student / sizeof (char *);   /* Number of records */

   if ((fp = fopen("foo", "w+")) == NULL) {
      perror("");
      exit(2);
   }

   for (i = 0; i < size; i++) {
      fputs(student[i], fp);        /* fputs doesn't insert \n by default */
      fputs("\n", fp);
   }

   rewind(fp);

   while (fgets(buf, BUFSIZE, fp) != NULL) {
      sscanf(buf, "%[^:]:%hd:%hd", name, &roll_no, &marks);
      printf("%-15s %2hd %3hd\n", name, roll_no, marks);
   }
   exit(0);
}
```

PROGRAM 15.6: **save_student_data.c**

```
Juhi Agarwal    11  97
Paridhi Jalan   22  80
Shiksha Daga    33  92
Paul Oomen      44 100
Abhijith        55  96
```

PROGRAM OUTPUT: **save_student_data.c**

15.7 fscanf AND fprintf: THE FORMATTED FUNCTIONS

Prototype: `int fscanf(FILE *stream, cstring, &var1, &var2, ...);`
Return value: Number of items matched, EOF on error or if end of input encountered before conversion of first item.

Prototype: `int fprintf(FILE *stream, cstring, var1, var2, ...);`
Return value: Number of characters printed on success, a negative value on error.

The **fscanf** and **fprintf** functions behave exactly like their terminal-oriented counterparts, **scanf** and **printf**, except that both use the file pointer as an additional argument. This is how **fscanf** reads a record containing three whitespace-delimited fields from a file and **fprintf** writes them along with a computed field to another file:

```
fscanf(fp1, "%s%hd%hd", name, &marks1, &marks1);
fprintf(fp2, "%s|%hd|%hd|%hd", name, marks1, marks2, marks1 + marks2);
```

Unlike the other I/O functions, the file pointer for both functions is the first, and not the last, argument. The functions had to be designed that way because they accept a variable number of arguments. The next argument is a control string (*cstring*) containing one or more format specifiers. This string is followed by a list of pointers for **fscanf**, and variables for **fprintf**.

Note that the preceding **fprintf** function writes "|"-delimited fields. Using a delimiter dispenses with the need to have lines of equal size, which helps conserve disk space. As you'll soon learn, the presence of a delimiter also makes it convenient to extract the fields from a line (or record). It's obvious that the delimiter must not be present in the data.

15.8 fscanf_fprintf.c: WRITING AND READING LINES CONTAINING FIELDS

Program 15.7 accepts a student's name and marks in two subjects from the keyboard and appends them to a file. This sequence is performed in a **do-while** loop. This loop terminates when the user presses *[Enter]* when prompted for the name. After rewinding, **fscanf** reads the file and **fprintf** displays the formatted contents of the *entire* file (including data saved in previous invocations of the program). This is possible only when **fopen** is used with the `"a+"` mode. Observe that all I/O operations have been carried out using only **fscanf** and **fprintf**.

```
/* fscanf_fprintf.c: Uses (i) fprintf to append user input to a file,
                     (ii) fscanf to read the entire file. */
#include <stdio.h>
#include <stdlib.h>
#define FLUSH_BUFFER while (getchar() != '\n') ;

int main(void)
{
   FILE *fp;
   char name[20];
   short marks1, marks2, total;

   if ((fp = fopen("foo", "a+")) == NULL) {      /* File will also be read */
      perror("");
      exit(2);
   }

   do {
      fprintf(stderr, "Student name: ");
      if (fscanf(stdin, "%[^\n]", name) != 1)
         break;                  /* Quits loop when only [Enter] is pressed */

      fprintf(stderr, "Marks in 2 subjects: ");
      fscanf(stdin, "%hd%hd", &marks1, &marks2);

      FLUSH_BUFFER

      fprintf(fp, "%s:%hd:%hd:%hd\n",             /* Writes a line to file */
                  name, marks1, marks2, marks1 + marks2);
   } while (1);

   rewind(fp);

   while (fscanf(fp, "%[^:]:%hd:%hd:%hd",         /* Reads each line */
                 name, &marks1, &marks2, &total) == 4)
      fprintf(stdout, "%-20s %3hd %3hd %3hd", name, marks1, marks2, total);

   exit(0);
}
```

PROGRAM 15.7: **fscanf_fprintf.c**

```
Student name: Larry Ellison
Marks in 2 subjects: 65 75
Student name: Jeff Bezos
Marks in 2 subjects: 75 85
Student name: Elon Musk
Marks in 2 subjects: 90 95
Student name: [Enter]
Larry Ellison         65  75 140
Jeff Bezos            75  85 160
Elon Musk             90  95 185
```

PROGRAM OUTPUT: **fscanf_fprintf.c**

Have a look at the file foo and you'll find that the marks are represented as text characters that look like numbers. That's how the data were actually keyed in, but they were converted by **fscanf** to binary. **fprintf** reconverted the binary numbers back to text before writing them to foo. When heavy number crunching is involved, numeric data must be stored and retrieved in binary form. This what the **fread** and **fwrite** functions are meant to be used for.

The previous program worked fine simply because each record of the file foo contained four fields. When a database is created from multiple and diverse sources, one often encounters lines having a wrong field count. If foo contained a line with three fields, the program would have simply gone haywire because of the usual buffer problems associated with the functions of the "scanf" family. We'll improve this program using **fgets** and **sscanf**, and eventually provide a near-perfect solution with **fread** and **fwrite**.

15.9 FILENAMES FROM COMMAND-LINE ARGUMENTS

In real life, we don't hard-code filenames in programs. Neither do we ask the user to key it in response to a prompt. If a program has to pause for taking user input, then the user needs to be present when the program is running. It would then be impossible to automate or schedule programs. The solution lies in using command-line arguments with file-handling programs.

When copying a file using the MSDOS **COPY** or UNIX **cp** command, we provide the two filenames as command-line arguments to the program (like **COPY foo foo.bak**). To briefly recall *(13.16)*, for a program to be invoked with arguments, its **main** section must look like this:

```
int main(int argc, char *argv[])
```

The first argument (argc) is set to the number of arguments which includes the program name. The second argument represents an array of pointers of type char *. A program to copy a file needs two filenames as arguments and is invoked in the following manner:

```
AOUT foo1 foo2
```

When you run the **AOUT** (or **a.out**) program from the MSDOS (or UNIX) shell, the program loader invokes **main** with its arguments set in the following manner:

```
argc = 3
argv[0] = AOUT.EXE or a.out          The name of the program
argv[1] = "foo1"                     The pointer to "foo1"
argv[2] = "foo2"                     The pointer to "foo2"
```

The reason why argv[0] stores the program name has been explained in Section 13.16, and is taken advantage of by UNIX programmers.

15.9.1 file_copy2.c: File Copying Using Command-Line Arguments

Program 15.8 improves a previous one (Program 15.3) by using command-line arguments to copy a file. The program first checks whether the right number of arguments has been entered. It then checks the access rights of the two files, which are represented as argv[1] and argv[2]. All error messages are written to the standard error stream using **fprintf**. Note that argv[0] represents the name of the program, which has been used here to display the first error message.

```
/* file_copy2.c: Copies a file using command-line arguments. */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   FILE *fp1, *fp2;
   int c;

   if (argc != 3) {               /* One argument is the program name */
      fprintf(stderr, "%s: Wrong number of arguments.\n", argv[0]);
      exit(1);
   } else if ((fp1 = fopen(argv[1], "r")) == NULL) {
      fprintf(stderr, "%s: Error in opening file for reading.\n", argv[1]);
      exit(2);
   } else if ((fp2 = fopen(argv[2], "w")) == NULL) {
      fprintf(stderr, "%s: Error in opening file for writing.\n", argv[2]);
      exit(3);
   }
   while ((c = fgetc(fp1)) != EOF)
      fputc(c, fp2);
   fclose(fp1); fclose(fp2);
   exit(0);
}
```

PROGRAM 15.8: **file_copy2.c**

```
$ a.out
a.out: Wrong number of arguments.
$ a.out foo
a.out: Wrong number of arguments.
$ a.out foo2 foo3
foo2: Error in opening file for reading.
$ a.out foo foo2
$ _                                                    Prompt returns; job done
```

PROGRAM OUTPUT: **file_copy2.c**

15.9.2 validate_records.c: Detecting Lines Having Wrong Number of Fields

Program 15.9 examines a file containing student data. A valid record contains four colon-delimited fields, but some records have a lesser number of fields. The original file contains seven records:

```
Larry Ellison:65:75:140
Jeff Bezos:75:160
Elon Musk:90:95:185
Sergei Brin:10:20:30
Bill Gates:99:99
Steve Jobs:99:98:97
Dennis Ritchie:22:33
```

Three records of this file contain three fields, and the program uses the **fgets-sscanf** duo to locate them. The valid and invalid lines are saved in two separate files. All filenames are supplied as command-line arguments. After the program has been invoked correctly, the contents of the input file foo are segregated and saved in the files foo_valid and foo_invalid.

```
/* validate_records.c: Validates records of a file using fgets and sscanf.
                       Filenames supplied as command line arguments. */
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv)          /* **argv is same as char *argv[] */
{
   FILE *fp1, *fp2, *fp3;
   short m1, m2, m3, items;
   char buf[80], name[20];
   if (argc != 4) {
      fprintf(stderr, "Usage: %s source validfile invalidfile\n", argv[0]);
      exit(1);
   } else if ((fp1 = fopen(argv[1], "r")) == NULL ||
              (fp2 = fopen(argv[2], "w")) == NULL ||
              (fp3 = fopen(argv[3], "w")) == NULL) {
      perror("");
      exit(2);
   }
   while (fgets(buf, sizeof(buf), fp1) != NULL) {
      items = sscanf(buf, "%[^:]:%hd:%hd:%hd", name, &m1, &m2, &m3);
      if (items != 4)
         fputs(buf, fp3);                        /* Invalid records */
      else                                       /* For valid records */
         fprintf(fp2, "%s:%hd:%hd:%hd\n", name, m1, m2, m3);
   }
   exit(0);
}
```

PROGRAM 15.9: **validate_records.c**

```
$ a.out
Usage: a.out source validfile invalidfile
$ a.out foo5 foo_valid foo_invalid
No such file or directory
$ a.out foo foo_valid foo_invalid
$_
```

Valid input; job done

PROGRAM OUTPUT: **validate_records.c**

```
Larry Ellison:65:75:140
Elon Musk:90:95:185
Sergei Brin:10:20:30
Steve Jobs:99:98:97
```

FILE: foo_valid

```
Jeff Bezos:75:160
Bill Gates:99:99
Dennis Ritchie:22:33
```

FILE: foo_invalid

sscanf can easily parse a record with delimited fields and isolate them. However, if the record length is not fixed (as is the case here), it is impossible to locate and update a specific record using the file offset pointer. The updated record could have a longer name and this would require the record to intrude into the space occupied by the next record. But if the records are saved as an array of structures (of equal length), then we can use the **fseek** function to move to a specific record and use the **fread** and **fwrite** functions to update the record. We'll soon be doing that, so stay tuned.

15.10 perror AND errno: HANDLING FUNCTION ERRORS

Program errors can occur for a host of reasons: a resource not available, the receipt of a signal, I/O operational failures or invalid call arguments. File handling is associated with its own set of problems:

- The file doesn't exist.
- The `"r"` mode is used but the file doesn't have read permission.
- The `"w"` mode is used but the file doesn't have write permission.
- The directory doesn't have write permission for creating a file.
- The disk may be full, so no further writes are possible.

Library functions indicate an error condition by returning a unique value. The file-handling ones either return NULL or EOF, so to create robust code, we should always check for this condition (unless we are certain that this check isn't necessary).

When a library function returns an error, the operating system sets the static (global) variable, `errno`, to a positive integer. This integer, represented by a symbolic constant, is associated with an error message. For instance, the integer `2` or `ENOENT` signifies "`No such file or directory`." There are two things we can do when an error occurs:

- Use **perror** to print the error message associated with `errno`.
- Determine the cause of the error by checking `errno`.

We use **perror** only when a function returns an error. The function follows this syntax:

```
void perror(const char *s);
```

The function takes a string *s* as argument and prints the string, a colon and a space, followed by the message associated with `errno`. So far, we have used **perror** using the null string (`""`) but that's not how it should be used. The use of `errno` and **perror** are demonstrated in Program 15.10, which tries to open a file that either doesn't exist or is not readable. Don't forget to include the file `errno.h`, which defines `errno`.

Tip: Program flow is often dependent on the cause of the error, so you should check the value of `errno` to determine the future course of action to take.

Caution: You must check the value of `errno` *immediately* after a function returns an error, and before you do anything else. The behavior of `errno` in the event of a successful call is undefined. Some systems leave it unchanged, but some don't. If necessary, save the value of `errno` in a separate variable if you need this value later in your program.

```
/* perror_errno.c: Displays function errors with perror  */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>                          /* For errno */

int main (int argc, char **argv) {
   if (fopen(argv[1], "r") == NULL) {
      fprintf(stderr, "errno = %d\n", errno);
      perror("open");
      exit(1);                              /* Indicates error condition */
   }
   exit(0);                                 /* Normal exit without errors */
}
```

PROGRAM 15.10: **perror_errno.c**

```
$ a.out foofoo
errno = 2
open: No such file or directory
$ a.out /etc/shadow                    An unreadable file on UNIX/Linux
errno = 13
open: Permission denied
```

PROGRAM OUTPUT: **perror_errno.c**

15.11 feof, ferror AND clearerr: EOF AND ERROR HANDLING

Prototype: `int feof(FILE *stream);`
Return value: Non-zero if EOF indicator is set, zero otherwise.

Prototype: `int ferror(FILE *stream);`
Return value: Non-zero if error indicator is set, zero otherwise.

Some input functions like **fgetc** and **fgets** return EOF and NULL, respectively, when end-of-file is encountered. But these functions return the same values on error as well. So how does one know whether a loop terminated on account of EOF or an error?

The **feof** and **ferror** functions provide the answer. Both use a file pointer as a single argument to check the indicators for EOF and errors in the FILE structure. Both return true when their respective indicators are set. Thus, after a loop that uses **fgetc** or **fgets** terminates, you may like to determine the actual cause of termination by using code similar to this:

```
while ((c = fgetc(fp)) != EOF)
   putc(c);

if (feof(fp))
   fputs("Normal termination on encountering EOF", stderr);
else if (ferror(fp))
   fputs("Termination on encountering error", stderr);
```

Before you use **feof**, be aware that a function doesn't encounter end-of-file when the file position indicator has reached the end of the stream. It's only when the function *tries* to read beyond the end that it knows that EOF has occurred and then returns accordingly. Ignorance of this concept has led to the proliferation of erroneous code that use **feof** as a control expression in a loop. Figure 15.2 shows alternative forms of usage of **feof**. The first is totally wrong while the second one is acceptable if we assume that no errors would occur.

Declarations: `FILE *fp = fopen("foo", "r");  char buf[80];`

```
while (!feof(fp)) {
   fgets(buf, 80, fp);
   fputs(buf, stdout);
}
```

Incorrect Usage

```
while (1) {
   fgets(buf, 80, fp);
   if (feof(fp))
      break;
   fputs(buf, stdout);
}
```

Acceptable Usage

FIGURE 15.2 Using `feof` in a Loop

To understand why the code on the left is erroneous, consider that the last line of `foo` has been read and printed. At this stage, **fgets** has not encountered EOF yet because it doesn't know whether further data is forthcoming. When the control expression is tested, `!feof(fp)` is still true, so the body of the loop is executed. This time, **fgets** encounters EOF (which would impact the next iteration), but **fputs** still goes ahead and prints the previous line. This means that the last line is printed twice.

The EOF and error indicators remain set until a third function—**clearerr**—clears both indicators. This function also uses the file pointer as argument but it returns nothing (`void`). None of the three functions sets the variable `errno`, so you can't use `errno` and **perror** to handle EOF and file I/O errors.

15.12 TEXT AND BINARY FILES

So far, we have been reading and writing *text* files as opposed to *binary* ones. The popular distinction that is made between them is that text files contain data in character form while binary files contain binary data. This view is unacceptable because all data are eventually read and written in binary form. Though no formal definition of a text file exists to this day, we must adopt one that is consistent with its usage. A text file generally has these features:

- It contains *readable* and *printable* data that are organized as a set of *lines* delimited by the newline character appropriate to the machine.
- Some characters like newline and EOF (like *[Ctrl-z]*) are treated differently by a program. You are aware of the newline conversions that take place when these files experience a change of environment.
- Text characters are drawn from the ones available on the keyboard. They have the ASCII values between 0 and 127 (7-bit ASCII), though all characters in this range are not printable.

- A number is represented as a string comprising digit characters and a decimal point. For instance, the number 123 is saved as the characters '1', '2' and '3' in a text file.

None of the preceding observations apply to binary files. A binary file can contain any character in the entire ASCII range (0 to 255). The newline or EOF character has no special significance in a binary file. Even though strings are accorded the same treatment by both file types, a binary file saves numbers in binary form. The number 123 can be stored in one byte, but a program may specify a `short` or `int`, in which case two or four bytes may be allocated.

Number crunching becomes very efficient when numbers are handled directly in binary form. The overheads of conversion from text to binary and vice versa can be quite significant, the reason why databases always store numeric data in binary form.

This review of the two file types will help us handle the **`fread`** and **`fwrite`** functions that are meant to be used with binary files.

15.13 `fread` AND `fwrite`: READING AND WRITING BINARY FILES

The previous file I/O functions handled data in units of characters (**`fgetc`**/**`fputc`**), lines (**`fgets`**/**`fputs`**) or matched items (**`fscanf`**/**`fprintf`**). The fourth and final category comprising the **`fread`** and **`fwrite`** functions handle data in more versatile ways:

- Numeric data are read and written in binary form.
- The amount of data to be transferred can be specified in larger units—an array, structure or an array of structures.

Each read or write operation transfers one or more records of equal length. Because of this equality, binary files can be updated by "seeking" to a specific record using the **`fseek`** function. We haven't updated a record so far, but we'll do it this time using **`fread`** and **`fwrite`**.

15.13.1 The `fread` Function

Prototype: `size_t fread(void *`*p*`, size_t` *size*`, size_t` *num*`, FILE *`*stream*`);`
Return value: Number of items successfully read even when EOF or error is encountered.

The **`fread`** function takes four arguments. It reads *num* units of data of *size* characters each from *stream*, and saves the data in the variable *p*, whose pointer is specified as the first argument. Because *p* can be a variable of any type, a generic (`void`) pointer is specified. `size_t` represents an integer type which is typedef'd to one of the primitive data types during compilation.

`fread` is meant for handling user-defined and derived data types, but it's a good idea to get started with a primitive data type. For reading a single integer, *num* must be specified as 1 and *size* as the size of an `int`:

```
int i;
fread(&i, sizeof(int), 1, fp);                    fp returned by fopen
```

This call reads one unit of four bytes *in binary form* and saves it in the variable `i`. A string is read in the following manner:

```
char stg[10];
fread(stg, sizeof(char), 10, fp);
```

fread reads 10 characters from the stream, or rather, 10 units of one `char` where the unit size is specified as the second argument. What makes **fread** different from its peers is that a single call can read a set of integers into an array:

```
short arr[5];
fread(arr, sizeof(short), 5, fp);                    Reads five numbers to array arr
```

Like the members of the "scanf" family, **fread** returns the number of items successfully read even if the function terminates with an error. **fread** can also read multiple data items in one call provided they form members of a structure. We'll be demonstrating this feature shortly.

15.13.2 The fwrite Function

Prototype: `size_t fwrite(void *`*p*`, size_t` *size*`, size_t` *num*`, FILE *`*stream*`);`
Return value: Number of items successfully written even when error is encountered.

fread and **fwrite** have identical prototypes; they differ only in the direction of data transfer. **fwrite** writes *num* units of data of *size* characters each from the variable *p* to the file associated with *stream*. This is how you save an entire array to a file:

```
short arr[5] = {100, 200, 300, 400, 500};
fwrite(arr, sizeof(short), 5, fp);
```

This is where using the binary mode can result in significant savings. The preceding **fwrite** function writes 10 bytes to disk. If these five numbers were saved in a text file, they would have taken up 15 bytes of disk space (plus one or two for the newline character).

15.13.3 fread_fwrite.c: Using the Primitive and Derived Data Types

Program 15.11 uses **fwrite** to write three data items—an integer, string and array of type `short`—to the file `foobar`, which is opened in binary mode (`"w+b"`). The items are originally stored in variables having the "out" suffix. They are read back using **fread** into a separate set of variables having the "in" suffix. Note that the contents of the five-element arrays, `s_in` and `s_out`, are transferred using a *single* invocation each of **fwrite** and **fread**.

```
/* fread_fwrite.c: Uses fwrite to write three types of data items
                   to a file and then reads them back with fread. */
#include <stdio.h>
#include <stdlib.h>
#define SIZE 15
```

```
int main(void)
{
   FILE *fp;
   int i_out = 1234567, i_in, i;
   char arr_out[SIZE] = "Zuckerberg", arr_in[SIZE];
   short s_out[5] = {10, 20, 30, 40, 50}, s_in[5];

   if ((fp = fopen("foobar", "w+b")) == NULL) {
      perror("foobar");
      exit(1);
   }

   fwrite(&i_out, sizeof(int), 1, fp);
   fwrite(arr_out, sizeof(char), SIZE, fp);
   fwrite(s_out, sizeof(short), 5, fp);


   rewind(fp);

   fread(&i_in, sizeof(int), 1, fp);
   fread(arr_in, sizeof(char), SIZE, fp);
   fread(s_in, sizeof(short), 5, fp);

   printf("%d %s ", i_in, arr_in);
   for (i = 0; i < 5; i++)
      printf("%hd ", s_in[i]);

   exit(0);
}
```

PROGRAM 15.11: **fread_fwrite.c**

```
1234567 Zuckerberg 10 20 30 40 50
```

PROGRAM OUTPUT: **fread_fwrite.c**

Note: Because `foobar` is a binary file, you can't view its contents using a text editor like Notepad or **`vim`**. Except for the string `Zuckerberg`, the other characters saved in `foobar` are unreadable.

15.13.4 save_structure.c: Saving and Retrieving a Structure

Program 15.12 demonstrates the use of **`fread`** and **`fwrite`** in handling an array of structures. The five-element array variable, `stud`, stores a three-member structure in each element. After **`sizeof`** has determined the total size of the array, a single invocation of **`fwrite`** writes the entire array to the file `student.dat`. After the file is rewound, a **`for`** loop retrieves each structure from the file to a separate variable (`stud2`) before displaying the value of the structure members on the terminal.

```
/* save_structure.c: Writes an array of structures to a file and
                     reads the data back. */
#include <stdio.h>
#include <stdlib.h>
#define COLUMNS 5

int main(void)
{
   FILE *fp;
   short i;
   struct student {
      char name[30];
      int roll_no;
      short marks;
   }  stud[COLUMNS] = {
                        {"Narciso Yepes", 1234, 666},
                        {"Andres Segovia", 4567, 555},
                        {"Sergio Abreu", 8910, 999},
                        {"Celedonio Romero", 2345, 777},
                        {"Leo Brouwer", 6789, 888}
                      }, stud2;              /* A second variable defined */

   int size = sizeof(struct student);
   printf("Size of structure: %d bytes\n\n", size);

   if ((fp = fopen("student.dat", "w+b")) == NULL) {
      perror("foobar");
      exit(1);
   }

   fwrite(&stud[0], size, COLUMNS, fp);    /* Saves entire array */
   rewind(fp);
   for (i = 0; i < COLUMNS; i++) {
      fread(&stud2, size, 1, fp);          /* Reads one element at a time */
      printf("%s %d %hd\n", stud2.name, stud2.roll_no, stud2.marks);
   }
   exit(0);
}
```

PROGRAM 15.12: **save_structure.c**

```
Size of structure: 40 bytes

Narciso Yepes 1234 666
Andres Segovia 4567 555
Sergio Abreu 8910 999
Celedonio Romero 2345 777
Leo Brouwer 6789 888
```

PROGRAM OUTPUT: **save_structure.c**

15.14 MANIPULATING THE FILE POSITION INDICATOR

Normally, files are read and written sequentially, with the file position indicator initially set to 0, the beginning of the stream. Sequential access can be terribly inefficient for large files, specially when the line to be accessed occurs near the end of the file. C supports a set of functions—**fseek**, **ftell** and **rewind**—that can *directly* take the file offset pointer to any point in the stream and report the current position of the offset pointer. These functions allow a file to be accessed in a *random* manner.

15.14.1 fseek: Positioning the Offset Pointer

Prototype: `int fseek(FILE *`*stream*`, long` *offset*`, int` *whence*`);`
Return value: 0 on success, -1 on failure.

The **fseek** function moves the file offset pointer to a specified point. **fseek** doesn't do any physical I/O, but it determines the position in the file where the next I/O operation will take place. The *offset* and *whence* arguments together control the location of the file's offset pointer. *offset* signifies the position (positive or negative) of this pointer relative to *whence*, which can take one of three values:

`SEEK_SET`	Offset pointer set to beginning of file.
`SEEK_END`	Offset pointer set to end of file.
`SEEK_CUR`	Offset pointer remains at current location.

These constants have the values `0`, `1` and `2`. With some restrictions, *offset* can be a positive or negative integer, and a constant is generally used with the `L` suffix. For instance,

```
fseek(fp, 10L, SEEK_CUR);
```

moves the pointer forward by 10 characters from its current position, and

```
fseek(fp, -10L, SEEK_END);                    Negative offset
```

sets the pointer 10 characters before EOF. You can also set the offset relative to the beginning of the file, so the following call,

```
fseek(fp, 100L, SEEK_SET);
```

positions the offset pointer to 100 bytes from the beginning of the file. You can't have a negative *offset* with *whence* set to `SEEK_SET`, but strangely enough, you can have a positive *offset* with *whence* at `SEEK_END`. In this case, the pointer moves beyond EOF, thus creating a *sparse* file—also called a file with a "hole". Sparse files find use in database applications.

15.14.2 The ftell and rewind Functions

Prototype: `long ftell(FILE *`*stream*`);`
Return value: Current value of the offset pointer on success, -1 on failure.

The **ftell** function takes the file pointer as argument and returns the current position of the file position indicator, i.e., the offset pointer. The function is always used in conjunction with **fseek**, for instance, to determine the size of a file:

```
long size;
fseek(fp, 0 SEEK_END);                                  Offset pointer at EOF
size = ftell(fp);
```

The **rewind** function simply moves the file position indicator to the beginning. Because the function doesn't return a value, we have always been using it in this manner:

```
rewind(fp);
```

rewind is often a better alternative to closing a file before reopening it (unless the reopening is done in a different mode).

Note: fseek, **ftell** and **rewind** work only with those files that are capable of seeking. They don't work with the standard files associated with the terminal and keyboard.

15.14.3 reverse_read.c: Reading a File in Reverse

Program 15.13 makes use of **fseek** and **ftell** to determine the size of a file before reading it in reverse. For that to be possible, we must first position the offset pointer to one byte beyond EOF. Each iteration of the **while** loop moves the indicator back by two bytes before a character is read. The output begins with a blank line simply because newline is the last character of `foo`. Before running the program, have a look at the contents of this file.

```
/* reverse_read.c: Reads a file in reverse using fseek.
                   Also uses ftell to determine file size. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *fp;
    int c;

    if (argc != 2) {
       fputs("Not enough arguments\n", stderr);
       exit(1);
    } else if ((fp = fopen(argv[1], "r")) == NULL) {
       perror(argv[1]);
       exit(2);
    }

    fseek(fp, 1L, SEEK_END);                /* Pointer taken to EOF + 1 ... */
    fprintf(stderr, "Size of file = %ld\n", ftell(fp) - 1);

    while (fseek(fp, -2L, SEEK_CUR) == 0)   /* ... and then back by 2 bytes */
       if ((c = fgetc(fp)) != EOF)
          putchar(c);

    fclose(fp);
    exit(0);
}
```

PROGRAM 15.13: **reverse_read.c**

```
abcdefghijkl
mnopqrst
uvwxyz
```

PROGRAM INPUT: Contents of `foo`

```
$ a.out
Not enough arguments
$ a.out foo
Size of file = 29
                                        Effect of \n at end of foo
zyxwvu
tsrqponm
lkjihgfedcba                            The first line in reverse
```

PROGRAM OUTPUT: **`reverse_read.c`**

Tip: As with some of the previous programs, this program can be redirected (using, say, **`a.out > bar`**) when run from the OS shell. The reversed output of `foo` would then be saved in `bar`. But the messages would still appear on the terminal because they write to `stderr`, which is unaffected by redirection. Wherever possible, programs should be designed to behave in a flexible manner.

15.15 `update_structure.c`: UPDATING A STRUCTURE STORED ON DISK

Program 15.14 highlights the advantages of using **`fread`** and **`fwrite`** with structures. It accesses the five structures saved in the file `student.dat` in Program 15.12. The current program searches this database for a specific record and updates it with user-input data. The updated contents of the binary database are finally displayed in text form. Like in a previous program, all messages barring one are directed to the standard error. Thus, this program can also be used with redirection to save the output—but only to a *text* file.

```
/* update_structure.c: Uses fseek, ftell, fread and fwrite
                       to update a structure saved in a file. */
#include <stdio.h>
#include <stdlib.h>
#define COLUMNS 5

int main(void)
{
   FILE *fp;
   struct student {
      char name[30];
      int roll_no;
      short marks;
   } stud;

   short i, s_marks, found = 0, records;
   int size, s_roll_no;
```

```
   if ((fp = fopen("student.dat", "rb+")) == NULL) {   /* wb+ would ... */
      perror("student.dat");                           /* ... overwrite file */
      exit(1);
   }
   size = sizeof(struct student);                 /* Size of each record */
   fseek(fp, 0, SEEK_END);
   records = ftell(fp) / size;
   fprintf(stderr, "Number of records available = %hd\n", records);
   fputs("Enter roll no and new marks: ", stderr);
   scanf("%d%hd", &s_roll_no, &s_marks);
   rewind(fp);
   for (i = 0; i < records; i++) {                /* Start search */
      fread(&stud, size, 1, fp);
      if (stud.roll_no == s_roll_no) {            /* If record located ... */
         found = 1;
         stud.marks = s_marks;                    /* ... update the member. */
         fseek(fp, -size, SEEK_CUR);              /* Go back to that record ... */
         fwrite(&stud, size, 1, fp);              /* ... and update file. */
         break;                                   /* No need to search further */
      }
   }
   if (found == 1) {
      rewind(fp);
      fputs("Record updated. Displaying all records ...\n\n", stderr);
      for (i = 0; i < records; i++) {
         fread(&stud, size, 1, fp);
         printf("%s %d %hd\n", stud.name, stud.roll_no, stud.marks);
      }
   }
   else
      fputs("Record not found\n", stderr);
   fclose(fp);
   exit(0);
}
```

PROGRAM 15.14: **update_structure.c**

```
Number of records available = 5
Enter roll no and new marks: 2345 5432
Record updated. Displaying all records ...

Narciso Yepes 1234 666
Andres Segovia 4567 555
Sergio Abreu 8910 999
Celedonio Romero 2345 5432                    Record updated on disk
Leo Brouwer 6789 888
```

PROGRAM OUTPUT: **update_structure.c**

It's necessary to understand the technique used in updating a portion of a file. After the record has been located, the file position indicator must be moved to the beginning of the next record and then brought back before **fwrite** updates the selected record on disk. This has been done in the first **for** loop.

15.16 THE OTHER FILE-HANDLING FUNCTIONS

Before we complete our discussions on the file-related functions, let's briefly examine three functions that don't involve any I/O with the file. Two of them—**remove** and **rename**—perform tasks that can also be carried out from the operating system. It is often necessary to carry out these tasks from a program as well. All three functions set the `errno` variable when they encounter an error, so you can determine the exact cause of failure.

15.16.1 The remove Function

The **remove** function deletes a file that is not open. The function uses the pathname as a single argument and returns 0 if the deletion succeeds, and -1 otherwise. The function uses the following syntax:

```
int remove(char *pathname);
```

It is necessary to check the return value of **remove** to determine whether deletion has actually taken place. Here's how the function should be used:

```
int retval = remove("foo");
if (retval == 0)
   fprintf(stderr, "File removal successful\n");
else
   fprintf(stderr, "File not deleted\n");
```

On UNIX/Linux systems, **remove** can also remove an empty directory. But the C standard doesn't address directories, so the function may or may not work with directories on non-UNIX systems.

15.16.2 The rename Function

The **rename** function renames a file (even if it is open). As seen in the following prototype, the function uses two arguments and returns 0 if successful, and -1 otherwise:

```
int rename(char *old, char *new);
```

The function renames *old* to *new*. If *new* exists, it will be replaced. Because both **remove** and **rename** return the same set of values, the latter is also used in a similar manner:

```
int retval = rename("foo", "foo.bak");
if (retval == 0)
   fputs("foo renamed to foo.bak\n", stderr);
else
   fputs("foo not renamed\n", stderr);
```

On UNIX/Linux systems, **rename** can also change the name of a directory provided that *new* either doesn't exist or is empty. However, this is not mandated by ANSI.

Note: The **rm** and **mv** commands in UNIX/Linux and **DEL** and **REN** commands in MSDOS also delete and rename files, respectively, when invoked from the shell.

15.16.3 The tmpfile Function

The **tmpfile** function opens a temporary file in the "wb+" mode, which enables read/write operations on it. As the prototype suggests, the function returns a FILE pointer but uses no argument:

```
FILE *tmpfile(void);
```

tmpfile returns a null pointer if the file can't be opened. The following statement opens a temporary file:

```
FILE *fp = tmpfile();
```

The file associated with fp is deleted when it is closed or when the program terminates.

WHAT NEXT?

The file-handling tools provide the framework for the storage and retrieval of large amounts of data. However, we are yet to address a fundamental storage problem. How does one handle data that exceed the defined storage? We need to create memory as and when a program needs it and create data structures that can be dynamically expanded or shrunk.

WHAT DID YOU LEARN?

A *file* is a named container that holds text or binary data. It is opened (**fopen**) in one of three basic modes ("r", "w" and "a"). A file is closed explicitly (**fclose**) or implicitly (program termination).

The standard files (stdin, stdout and stderr) are always open and connected to every running program.

On file opening, a *file pointer* and one or two data buffers are associated with the file. All I/O functions use the file pointer to access the file. The buffer acts as a go-between for data transfer.

I/O functions obtain information on the current state of a file from a structure of type FILE. They look up the *file position indicator* or *file offset pointer* to know how much of the file has been accessed. FILE also contains the EOF and error flags.

The **fgetc** and **fputc** functions read and write one character at a time. The macros, **getc** and **putc**, perform the same functions.

The **fgets** and **fputs** functions handle a line of text in each call. **fgets** adds a newline (if possible) and the NUL character, while **fputs** removes the NUL. The **fscanf** and **fprintf** functions handle formatted data.

Filenames are often specified as command-line arguments. The redirection feature enables messages sent to `stdout` and `stderr` to be segregated.

I/O errors set the global variable, `errno`. The associated message is displayed with **perror**. The **feof** function determines whether EOF has occurred.

Binary files store numeric data in binary. The **fread** and **fwrite** functions are the only functions that can read and write arrays and structures to disk.

A file can be accessed in a random manner with **fseek** and **ftell**. The file position indicator can be reset to the beginning with **rewind**.

A file can be deleted (**remove**) and renamed (**rename**). **tmpfile** opens a temporary file which is removed automatically.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

15.1 It is possible to read a file without opening it first.

15.2 No conversion of newline takes place when a file is opened in the binary mode.

15.3 The only way to reposition the file position indicator is by using the **fclose** and **fopen** functions.

15.4 The standard streams, *stdin*, *stdout* and *stdout*, are not associated with the `FILE` structure.

15.5 The **fgets** and **fputs** functions are meant to be used with text files.

15.6 The **fscanf** and **fprintf** functions can read and write the same file.

15.7 The file position indicator can be moved forward but not back.

15.8 The size of a binary file remains unchanged when moved across operating systems.

15.9 It is not possible to write data beyond EOF.

A2. FILL IN THE BLANKS

15.1 The _____ _____ allocates the disk blocks used by a file.

15.2 A _____ file is organized as lines terminated by the _____ character.

15.3 All file-handling functions need the include file _____.

15.4 The _____ mode of **fopen** enables a file to be read and overwritten.

15.5 The **fopen** function returns a pointer to a structure of type _____.

15.6 The _____ function reads formatted input from a file.

15.7 When an error occurs, the variable _____ is set, and the _____ function prints the text associated with that variable.

15.8 The _____ function determines whether EOF was encountered by the previous function.

15.9 The _____ function is used to write an array or structure to disk.

A3. MULTIPLE-CHOICE QUESTIONS

15.1 The filename is used as an argument by (A) **`fread`**, (B) **`fclose`**, (C) **`rewind`**, (D) none of these.

15.2 For reading and updating an existing file, **`fopen`** uses the mode (A), `w`, (B) `w+`, (C) `r+`, (D) `a+`.

15.3 To read a character from a disk file, the function to use is (A) **`getc`**, (B) **`fgets`**, (C) **`getchar`**, (D) none of these.

15.4 For sending diagnostic messages to the standard error, the function to use is (A) **`fputs`**, (B) **`printf`**, (C) **`fprintf`**, (D) all of these, (E) A and C.

15.5 The data types handled by the **`fread`** and **`fwrite`** functions are (A) all types, (B) primary, (C) derived, (D) user-defined, (E) C and D.

15.6 The size of a file can be determined by (A) **`sizeof`**, (B) **`fseek`**, (C) **`ftell`**, (D) B and C, (E) none of these.

15.7 Pick the odd item out: (A) **`fgets`**, (B) **`getchar`**, (C) **`fseek`**, (D) **`rewind`**.

A4. MIX AND MATCH

15.1 Associate the following C terms with their areas of application:

(A) `rb`, (B) `SEEK_SET`, (C) `FILE`, (D) `errorno`.

(1) **`rewind`**, (2) **`fopen`**, (3) wrong item, (4) structure.

CONCEPT-BASED QUESTIONS

15.1 When will you use the library functions to handle files in preference to the < and > redirection features?

15.2 Discuss the significance of the file offset pointer when a file is read by different file-handling functions. How can this pointer be reset to the beginning of the file?

15.3 Explain the concept of *streams* and their significance in I/O operations.

15.4 Specify the circumstances when the `fopen` function can fail.

15.5 How do **`fgets`** and the scan set of **`scanf`** treat the newline character?

15.6 Discuss the circumstances that determine whether to save data with or without delimiters separating the fields.

PROGRAMMING & DEBUGGING SKILLS

15.1 Write a program that writes the lowercase letters of the English alphabet, one character at a time, to a file. Read back the file to display each letter but in uppercase.

15.2 Write a program that accepts three lines of text from user input and saves them to a file `foo`. Reopen the file to add three more lines and finally display all six lines.

15.3 Write a program that double-spaces a file whose name is input by the user. The output filename will have the "`.new`" suffix affixed to the input filename.

15.4 Write a program that removes all blank lines from a file, whose name is input by the user. The output filename has the "`new_`" prefix to the input filename. (A blank line comprises *only* zero or more whitespace characters.)

15.5 Write a program that first creates a string (with the NUL) by appending each lowercase letter to a `char` array in a loop. The string must next be written to a file and read back for display.

15.6 Write a program that accepts a search string from the user and then searches a file `foo` (using a library function) for displaying all lines containing the string. If the string is not found, an appropriate message must be displayed.

15.7 Write a program that concatenates two files whose names are input as command-line arguments.

15.8 Write a program that reads files `foo1` and `foo2` and writes `foo3` with a line of `foo1` followed by a line of `foo2` and so on. The sequence is terminated when EOF is encountered in one of the files. The program must display the contents of `foo3`.

15.9 Assuming that a text file contains characters in the ASCII range 0 to 127 and contains at least one newline character, create a function that uses the filename as argument to return 1 if the file contains only text, or -1 otherwise.

15.10 Write a program that compares two files whose names are provided as user input. The program must quit with a suitable message that indicates the location in the file where the first mismatch was encountered.

15.11 Write a program that accepts an integer *n* and a filename as command-line arguments to print the first *n* lines of the file.

15.12 Assuming that the line length is limited to 256 characters, write a program that reads all lines of a text file into a 2D array. Display the lines saved in the array.

15.13 Write a program that reads a file `foo`, checks the line length and saves the lines in `foo1` or `foo2` depending on whether the line length exceeds 80 or not.

15.14 Write a program that accepts an integer *n* and a filename as command-line arguments to print the last *n* lines of a file (like the UNIX **`tail`** command). (HINT: First count the number of lines in the file.)

15.15 A file `foo` contains the following lines:

```
Bosch 1861 Sparkplug
Berliner 1851 Gramophone
Daimler 1834 Engine
```

Write a program that uses **fscanf** to read and print only the third field. Repeat the exercise using **fgets** and **sscanf** and without closing `foo`.

15.16 A file `foo` contains the following lines:

```
Gagarin 23-12-1927
Glenn Dec-31-17
Armstrong 20-Jun-29
```

Write a program that scans each line with **sscanf** and writes the lines to a file `bar` using the format *dd-mm-yy* for the date field.

15.17 Correct the following program and explain why it still prints a line of `foo` after invoking the **remove** function.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char buf[80];
   FILE *fp = fopen("foo", 'r');
   remove("foo");
   fgets(fp, buf, 80);
   fprintf("%s\n", buf, stdout);
   return 0;
}
```

15.18 Write a program that deletes all files whose names are provided as command-line arguments.

16 Dynamic Memory Allocation and Linked Lists

WHAT TO LEARN

- Concepts of *dynamic memory allocation* and the returned generic pointer.
- Allocating memory with the **`malloc`** and **`calloc`** functions.
- Freeing the allocated block with the **`free`** function.
- Resizing an allocated memory block with **`realloc`**.
- How negligence leads to *memory leaks* and *dangling pointers*.
- How 2D arrays can be simulated on-the-fly while a program is running.
- Attributes of a *linked list* and how they overcome the limitations of an array.
- Manipulating a linked list with user-defined functions.
- Features of the abstract data types—*stacks*, *queues* and *trees*.

16.1 MEMORY ALLOCATION BASICS

Up until now, memory for variables, arrays and structures have been allocated at the time of their *definition*. For variables and arrays, definition is synonymous with *declaration*. The amount of memory thus allocated is fixed at compile time and can't be modified at runtime. This inflexibility becomes a serious limitation when working with arrays. In our quest to ensure that arrays are large enough to accommodate our needs, we often tend to oversize them. Because of the wastage resulting from this *static* allocation of memory, our programs are often not as efficient as they could otherwise have been.

The solution to this problem lies in *dynamically* allocating memory at runtime, i.e., as and when the program needs it. Access to the allocated block is provided by a generic pointer that is automatically aligned to reflect the correct type of data pointed to. Because a pointer to a contiguous memory block can also be treated as an array, we can easily manipulate this array. In the course of program execution, if this memory chunk is found to be under- or over-sized, it can be resized using a special function (**`realloc`**) of the standard library.

This flexibility comes at a price—the responsibility to free up the allocated memory when it is no longer required. So far, memory deallocation has not concerned us because the system has a well-defined mechanism for deallocating memory for variables defined inside and outside a function. A function on return frees memory used by its local variables, while global and static variables are automatically deallocated on program termination. For memory allocated dynamically, we must do this job ourselves. If we fail to behave responsibly, there can be serious runtime problems resulting from *memory leaks*.

When memory is allocated at runtime, its size is usually determined either by user input or an expression evaluated by the running program. This feature is exploited by a special data structure called a *linked list*. Unlike an array, a linked list cannot be defined with a predetermined size—either at compile time or runtime. When an item is added to the list, memory is created for that item *at that moment* and a link (in the form of a pointer) is established between this new item and its immediate predecessor in the list.

In this chapter, we'll first examine the library functions that dynamically allocate memory and one function that frees it. Using these functions, we'll develop the techniques needed to create, modify and query a linked list that can be expanded and contracted at will.

16.2 THE FUNCTIONS FOR DYNAMIC MEMORY ALLOCATION

When a program needs memory to be allocated, it makes a request to the operating system. If the allocated memory is later found to be inadequate or excessive, the program can also ask for a reallocation with a revised size. The C library supports a set of four functions for dynamically allocating and deallocating memory. The file `stdlib.h` must be included for the following functions to work properly:

- **`malloc`** This function uses a single argument to specify the memory requirement in bytes.
- **`calloc`** For allocating memory for arrays, **`calloc`** is more convenient to use than **`malloc`**. **`calloc`** uses two arguments—the number of array elements and the size of each element.
- **`realloc`** This is the function to use for resizing a memory block allocated by a prior call to **`malloc`** or **`calloc`**.
- **`free`** Memory allocated by any of the previous functions is deallocated by this function.

How does one access the allocated or reallocated memory? Unlike variables, arrays and structures, a dynamically allocated memory block has no name. For providing access to this memory block, all of these functions (barring **`free`**) return a *generic* pointer that points to the base address of the block. All read-write operations on the block are subsequently carried out using this pointer. The **`free`** function also uses this pointer to deallocate memory.

Because memory allocation can sometimes fail (for reasons that are discussed soon), the three allocation functions must always be tested for NULL on invocation (even though we won't always practice what we have just preached).

16.2.1 The Generic Pointer

The pointer returned by the memory allocation functions is generic because a memory block can be used to store any data type—fundamental, derived or user-defined. Since C89, a `void` pointer can be assigned to a pointer of any type with or without using an explicit cast. For instance, the following statements using the **`malloc`** function assign the base address of a block of 12 bytes to a pointer variable `p` of type `int *`:

```
int *p = (int *) malloc(12);          Not recommended
int *p = malloc(12);                  Recommended
```

Many programmers and textbooks use the former form, which made sense in the pre-C89 days when the explicit cast was necessary. However, C89, which introduced the `void` pointer, dispenses with the need to use a cast. We'll, therefore, use **`malloc`**, **`calloc`** and **`realloc`** without the cast (second form) to avoid cluttering the code, which also makes it easier to maintain.

16.2.2 Error Handling

All of these functions can fail if memory can't be allocated (or reallocated). Even though a computer may have a lot of "free" memory, it is not entirely available for dynamic memory allocation. While memory is needed by the code and data of a program, the latter needs additional memory while it is running. This additional memory is of two types—the stack and heap. Figure 16.1 replicates Figure 11.3 which shows the memory layout of these segments in a typical UNIX system; other systems are broadly similar.

The *stack* is consumed by parameters and local variables used by a function. The stack grows continuously as one function calls another, and shrinks when the functions return. *Stack overflow* commonly occurs when too many recursive function calls are made without encountering the terminating condition *(11.3)*.

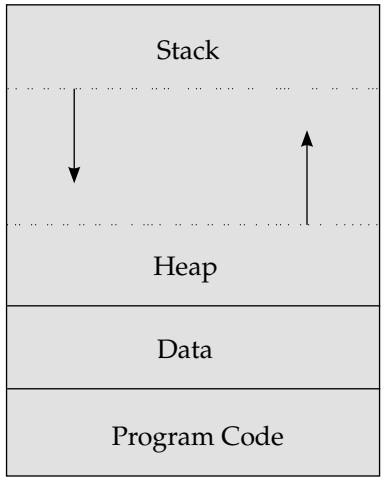


FIGURE 16.1 Organization of the Stack and Heap in Memory

The *heap* is used for dynamically allocating memory. It grows at the expense of the stack and vice versa. A block of memory that is freed using the **free** function is returned to the heap. Failure to free memory that is no longer required can result in severe heap depletion that may make further allocation impossible. The memory allocation functions return NULL if they fail to find the specified amount of memory *as a contiguous chunk* in the heap. Every time we use these functions, we must thus check for NULL.

Note: Dynamic memory allocation creates a chunk of contiguous memory. A pointer to this chunk can be treated as an array except that the latter is a constant pointer unlike a regular pointer which can be made to point to this block.

16.3 malloc: SPECIFYING MEMORY REQUIREMENT IN BYTES

Prototype: `void *malloc(size_t size);`
Header File: `stdlib.h`
Return Type: A generic pointer to the allocated memory on success, NULL on failure.

We begin our discussions on the individual functions with **malloc** which is most commonly used for dynamic memory allocation. **malloc** accepts the size of the requested memory in bytes as an argument and returns a generic pointer to the base address of the allocated block. As shown in the following, this generic pointer is suitably aligned to a pointer variable of a specific data type before the memory chunk can be accessed:

```
int *p;
p = malloc(4);                          No cast needed
```

malloc allocates a chunk of four bytes whose pointer is implicitly converted to `int *` on assignment. The allocated space can now be used to store a four-byte `int`, but the portable and correct technique would be to use **sizeof(int)** as the argument to **malloc**:

```
p = malloc(sizeof(int));
```

malloc doesn't initialize the allocated block, so each byte has a random value. (This is not the case with **calloc**.) However, we can dereference the pointer `p` to assign a value to this block:

```
*p = 5;                                 Assigns value to unnamed memory segment
```

Observe that the allocated block can't be treated as a variable even though it can be manipulated by its pointer. You must not lose this pointer because it represents the only mechanism that provides access to the block.. Further, this pointer is also used by the **free** function to deallocate the block:

```
free(p);                                Deallocates memory allocated by malloc
```

The pointer returned by **malloc** is also used by the **realloc** function for resizing the allocated block, if considered necessary. Both **realloc** and **free** are examined soon.

16.3.1 malloc.c: An Introductory Program

Before discussing the other features related to **malloc**, let's consider Program 16.1 which invokes **malloc** three times to return pointers of three different data types. The program doesn't use the **free** function to deallocate the blocks because it is not required here. The annotations make further discussions on this program unnecessary.

```
/* malloc.c: Uses malloc without error checking. */

#include <stdio.h>
#include <stdlib.h>           /* Needed by malloc, calloc, realloc and free */

int main(void)
{
   short *p1;
   int *p2;
   float *p3;

   p1 = malloc(2);                          /* Non-portable; not recommended */
   p2 = malloc(sizeof(int));                /* The right way */
   p3 = malloc(sizeof(float));              /* Ditto */

   *p1 = 256; *p3 = 123.456;
   printf("*p1 = %hd\n", *p1);
   printf("*p2 = %d\n",  *p2);              /* Prints uninitialized value */
   printf("*p3 = %f\n",  *p3);

   exit(0);                                 /* Frees all allocated blocks */
}
```

PROGRAM 16.1: **malloc.c**

```
*p1 = 256
*p2 = 0                                     Uninitialized value; may not be 0
*p3 = 123.456001
```

PROGRAM OUTPUT: **malloc.c**

16.3.2 Error-Checking in malloc

The preceding program assumed that **malloc** always succeeds in allocating memory, which may not be the case. The function fails when there is not enough contiguous memory available for allocation in the heap. We must, therefore, check the pointer returned by **malloc** for NULL before using it. If **malloc** fails, it may not make sense to continue with program execution:

```
long *p = malloc(sizeof(long));
if (p == NULL) {
   fputs("Memory allocation failed\n", stderr);
   exit(1);
}
```

When the total amount of dynamic memory required by a program exceeds the amount available on the heap, a program must recycle memory. At any instant, adequate memory *must* be available for the next allocation. For small programs that eventually terminate, this is not an issue as the allocated memory is automatically returned to the heap on program termination. However, memory recycling is essential for server programs that run continuously without ever terminating.

16.3.3 Using `malloc` to Store an Array

Because array elements are stored contiguously in memory and **`malloc`** creates a contiguous block, it is often used to create space for an array. Thus, the following statement creates a memory block for storing 10 `short` values:

```
short *p;
p = malloc(10 * sizeof(short));
```

Since the name of an array represents a pointer to the first array element, `p` can be treated as an array for navigating and performing pointer arithmetic. We can assign values to the "elements" of this dynamically created "array" using either pointer or array notation:

Pointer Notation	*Array Notation*
`*p = 5;`	`p[0] = 5;`
`*(p + 1) = 10;`	`p[1] = 10;`
`*(p + 2) = 15;`	`p[2] = 15;`

Even though the **`calloc`** function has been specifically designed for allocating memory for arrays and structures, simple arrays are easily handled with **`malloc`**. As the next program demonstrates, the size of the array can be determined at runtime.

Takeaway: The **`malloc`** function makes available a contiguous block of memory that can be used to store any data type including arrays and structures but without automatic initialization.

16.3.4 `malloc_array.c`: An Array-Handling Program Using `malloc`

Program 16.2 uses **`malloc`** to create space for an array, where the number of elements is determined by user input. Unlike Program 16.1, this one validates the return value (`p`) of **`malloc`**. Because `p` is used to browse the array and finally free the allocated block, it is necessary to keep its value unchanged. For this reason, `p` is copied to `q`, which is changed in a loop to assign values to the array elements.

```
/* malloc_array.c: Uses memory allocated by malloc as an array after
                   saving the original pointer returned. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int *p, *q;
   short size, i;
```

```
    fputs("Enter number of array elements: ", stderr);
    scanf("%hd", &size);

    p = malloc(size * sizeof(int));
    if (p == NULL) {
       fprintf(stderr, "Memory allocation failed.\n");
       exit(1);
    }

    q = p;                          /* Saves p for later use */

    /* Assigning values to the "array" q ... */
    for (i = 0; i < size; i++)
       *q++ = i * 10;
                        /* ... and printing them using p */
    for (i = 0; i < size; i++)
       printf("p[%hd] = %6d, Address = %p\n", i, p[i], (p + i));

    free(p);            /* Deallocates block, but not necessary here */
    exit(0);
}
```

PROGRAM 16.2: **malloc_array.c**

```
Enter number of array elements: 5
p[0] =      0, Address = 0x804b008
p[1] =     10, Address = 0x804b00c          4 bytes more than previous address
p[2] =     20, Address = 0x804b010          ... Ditto ...
p[3] =     30, Address = 0x804b014          ... Ditto ...
p[4] =     40, Address = 0x804b018          ... Ditto ...
```

PROGRAM OUTPUT: **malloc_array.c**

Takeaway: `malloc` doesn't determine the data type of the dynamically created block. The type is determined by the user when declaring the pointer to this block.

Caution: It is a common programming error to use the pointer variable assigned by **`malloc`** in a subsequent call to the same function without freeing the previous block.

16.4 free: FREEING MEMORY ALLOCATED BY malloc

Memory allocated dynamically must be freed when it is no longer required. This will happen automatically when the program terminates, but we often can't afford to wait for that to happen. Memory could be in short supply, the reason why we may need to explicitly free it with the **`free`** function.

`free` takes the pointer returned by **`malloc`**, **`calloc`** and **`realloc`** as its only argument and frees the block associated with the pointer. The function uses the following prototype:

```
void free(void *ptr);
```

free returns nothing and *ptr* must have been previously returned by any of the memory allocation functions. Thus, the memory allocated by

```
int *p = malloc(sizeof(int) * 10);
```

is returned to the heap by a call to **free**, preferably followed by setting the pointer to NULL:

```
free(p);                    Pointer lost
p = NULL;                   Can't access the freed memory
```

free doesn't need to know the size of the block because it is known to the OS which is ultimately responsible for memory management. Because of its subsequent assignment to NULL, the original value of the pointer is now lost, so the freed block can no longer be accessed directly.

For programs performing simple tasks that consume insignificant amounts of memory, we need not use **free** because the allocated memory will be automatically deallocated on program termination. But when heap memory is scarce, a program that calls **malloc** 100 times (possibly in a loop) must also call **free** 100 times. This ensures that, at any instant of time, adequate memory is available on the heap for allocation.

Takeaway: The pointer returned by **malloc** is used for three purposes: (i) for performing read/write operations on the allocated block, (ii) for resizing the block with **realloc**, (iii) for deallocating the block with **free**.

Note: **free** can't release *named* memory blocks used by variables, arrays and structures. The argument used by **free** must point to the *beginning* of a block previously allocated dynamically.

16.5 MEMORY MISMANAGEMENT

Dynamic memory allocation is associated with a couple of problems that we have not encountered previously because all memory management was hitherto handled automatically by the program. We are now in a different world where we need to address the following issues:

- Even though the memory block is freed, it is still accessible because the pointer to it is not lost (dangling pointer).
- The memory block is not freed at all. The problem becomes acute when **malloc** is repeatedly used in a loop (memory leak).

Failure to handle a dangling pointer or a memory leak may not (unluckily) affect some of your programs. If your program misbehaves soon, consider yourself lucky because you can rectify the mistake before finalizing the code. Let's now examine these two issues.

16.5.1 The Dangling Pointer

Freeing a memory block with **free(p)** doesn't necessarily make it inaccessible because p continues to point to the block. p is now a *dangling pointer*. As a safety measure, it should be set to NULL immediately after invoking **free**:

```
free(p);                    p becomes a dangling pointer
p = NULL;                   p is now a null pointer
```

Because this memory—partially or totally—may have already been allocated, a subsequent attempt to access it using p could produce erroneous results or even cause the program to crash. Setting p to NULL solves the problem easily.

16.5.2 Memory Leaks

A different problem arises when p is made to point to another block without freeing the existing one. The previous block can't then be freed (except by program termination) because its pointer is lost. A block without its pointer creates a *memory leak* which can turn serious when **malloc** is repeatedly used in a loop.

What happens when **malloc** is used in a function where the pointer to the block is declared as a local variable? Repeated invocation of the function will progressively deplete the heap unless arrangements are made to free the blocks outside the function. Figure 16.2 shows the improved version of the **my_strcpy** function *(13.8.3)* shown with the **main** section.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *my_strcpy2(const char *src);

int main(void){
   char stg[80], *p;

   printf("Enter string: ");
   scanf("%[^\n]", stg);

   p = my_strcpy2(stg);
   printf("Copied string: %s\n", p);

   free(p);                          /* Frees memory created in the function */
   return 0;
}

char *my_strcpy2(const char *src)
{
   char *dest;
   short length;

   length = strlen(src) + 1;         /* One extra slot for NUL character */
   dest = malloc(length);

   strcpy(dest, src);
   return dest;                      /* Pointer to block now available in main */
}
```

FIGURE 16.2 Freeing Memory Created in a Function

Unlike **my_strcpy**, the **my_strcpy2** function uses a single argument—a pointer to the source string. The function uses **malloc** to create a correctly-sized memory block by determining the length of the string first. The copied string is then written to this block and its pointer is returned by the function. Note that termination of the function doesn't free the block which is eventually freed in **main**. A memory leak is thus prevented *because the pointer to the block could be transmitted to the caller*.

Takeaway: A memory block created inside a function remains in existence after the function terminates. If this block has a local variable pointing to it, this pointer variable must be returned by the function so it can be used in the caller to free the block.

16.6 malloc_2Darray.c: SIMULATING A 2D ARRAY

How does one use **malloc** to create space for a 2D array? To briefly recall, a 2D array is an array where each element represents a separate array *(10.12)*. Also, for the array `arr[i][j]`, the non-existent "element", `arr[i]`, represents a pointer to the *i*th row, which contains *j* columns. Figure 16.3 depicts the simulation of a 2D array where each element of the array of pointers points to a separate array (shown on the right). The left-most pointer must, therefore, be defined as a pointer to a pointer.

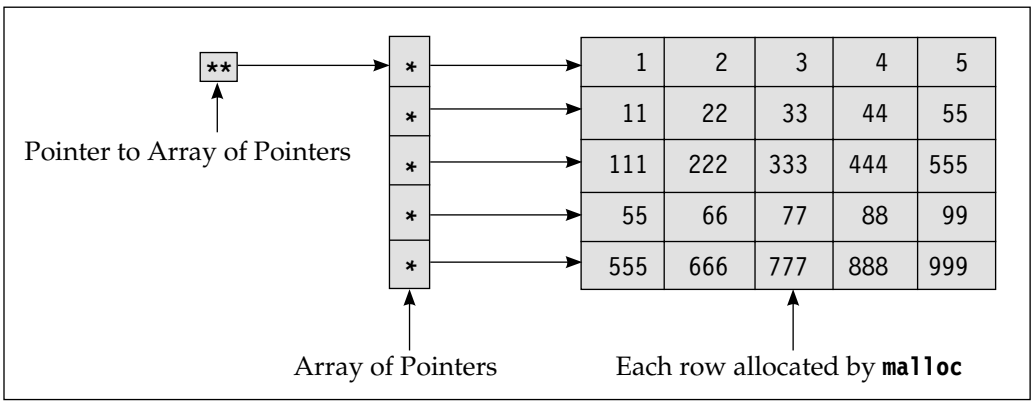


FIGURE 16.3 Simulating a 2D Array with an Array of Pointers

Program 16.3 uses **malloc** in two stages for creating space that can be accessed with 2D array notation: (i) for storing pointers to the rows, (ii) for storing the columns of each row. The program populates the array with user input, displays them and finally frees the allocated spaces. Observe that the block used by the array of pointers is freed *after* the blocks used by the columns are freed inside a loop.

```
/* malloc_2Darray.c: Uses malloc to (i) create an array of pointers,
                     (ii) an array of short integers for each pointer.
                     Also populates and prints array. */
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
{
   short rows, columns, i, j;
   short **p;                    /* Signifies a 2D array p[][] */

   fputs("Number of rows and columns: ", stderr);
   scanf("%hd %hd", &rows, &columns);

   /* Allocate memory for array of pointers */
   p = malloc(sizeof(short) * rows);
   if (p == NULL) {
      fputs("Cannot allocate memory for row pointers: ", stderr);
      exit(1);
   }

   /* Allocate memory for columns of each row */
   for (i = 0; i < rows; i++) {
      if ((p[i] = malloc(sizeof(short) * columns)) == NULL) {
         fputs("Cannot allocate memory for columns: ", stderr);
         exit(1);
      }
      /* Populate the array ... */
      fprintf(stderr, "Key in %hd integers for row %hd: ", columns, i);
      for (j= 0; j < columns; j++)
         scanf("%hd", &p[i][j]);
   }
   /* ... and print the array */
   for (i = 0; i < rows; i++) {
      for (j= 0; j < columns; j++)
         printf("%5hd ", p[i][j]);
      printf("\n");
      free(p[i]);              /* Frees each pointer allocated for columns */
   }
   free(p);                    /* Frees pointer for rows */
   exit(0);
}
```

PROGRAM 16.3: **malloc_2darray.c**

```
Number of rows and columns: 3 5
Key in 5 integers for row 0: 1 2 3 4 5
Key in 5 integers for row 1: 11 22 33 44 55
Key in 5 integers for row 2: 111 222 333 444 555
    1     2     3     4     5
   11    22    33    44    55
  111   222   333   444   555
```

PROGRAM OUTPUT: **malloc_2darray.c**

16.7 malloc_strings.c: STORING MULTIPLE STRINGS

The standard technique of reading a string from the keyboard is to use **scanf** with a char array. For reading a set of strings, however, we can use a 2D char array *(13.12)*. This technique wastes space if the strings are of unequal size but we didn't have a better option then—or so it seemed. But we do have a better option now, one that uses **malloc**.

A set of strings can be saved in an array of char pointers *(13.13)*, but this technique doesn't work with strings input from the keyboard. You can't use **scanf** with a char pointer that doesn't signify or point to an array. Program 16.4 achieves the best of both worlds by using **malloc** to dynamically allocate memory both for the strings and the array of pointers. No space is wasted for storing the strings which are handled in the following manner:

- Each string is temporarily read into an array (temp) of a fixed size.
- The length of the string is computed with **strlen**.
- A block of memory is dynamically created with **malloc** for the string.
- The string is finally copied from temp to the block.

The program first uses **malloc** to create the array of pointers named p_names after knowing the number of strings that it has to hold. In the first **for** loop, each string is input, its size ascertained and space for it created with **malloc**. In the second **for** loop, the string is printed and its storage deallocated. Finally, the space used by the array of pointers is freed.

```
/* malloc_strings.c: Program to create correctly-sized memory blocks for
                     storing multiple strings. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define FLUSH_BUFFER while (getchar() != '\n') ;

int main(void) {
   int i, length, num;
   char **p_names;           /* Array of pointers for multiple strings */
   char temp[80];

   fputs("Number of strings to enter: ", stderr);
   scanf("%d",&num);
   p_names = malloc(num * sizeof(char *)); /* (A) Allocates memory for ... */
                                           /* ... array of char pointers */
   for (i = 0; i < num; i++) {
      FLUSH_BUFFER
      fputs("Name: ", stderr);
      scanf("%[^\n]", temp);
      length = strlen(temp);

      /* Now allocate memory for every string read */
      p_names[i] = malloc(length + 1);      /* (B) Extra byte for NUL */
```

```
      if (p_names[i] == NULL) {
         fputs("Memory allocation failed\n", stderr);
         exit(1);
      }
      strcpy(p_names[i], temp);      /* Copies string to allocated memory */
   }

   for (i = 0; i < num; i++) {
      printf("%s\n", p_names[i]);        /* Prints each string */
      free(p_names[i]);                  /* Frees memory allocated in (B) */
   }
   free(p_names);                        /* Frees memory allocated in (A) */
   exit(0);
}
```

PROGRAM 16.4: **malloc_strings.c**

```
Number of strings to enter: 5
Name: Michelangelo Buonarroti
Name: Leonardo da Vinci
Name: Vincent van Gogh
Name: Rembrandt
Name: Pablo Picasso
Michelangelo Buonarroti
Leonardo da Vinci
Vincent van Gogh
Rembrandt
Pablo Picasso
```

PROGRAM OUTPUT: **malloc_strings.c**

Using **scanf** with the scan set *(9.11.1)* in a loop, you can key in multi-word strings. The %s specifier won't work here even though it works without problems in **printf**. The program is conceptually similar to Program 13.11, except that we are now working with a *ragged* array (containing an unequal number of columns).

16.8 calloc: ALLOCATING MEMORY FOR ARRAYS AND STRUCTURES

Prototype: `void *calloc(size_t num, size_t size);`
Header File: `stdlib.h`
Return Type: A generic pointer to the allocated memory on success, NULL on failure.

Even though **malloc** can create space for storing arrays and structures, C supports a separate function—**calloc**—for dynamically allocating memory for arrays of any type. Unlike **malloc**, **calloc** needs two arguments:

- The number of array elements (*num*).
- The size in bytes of each element (*size*).

calloc creates a block of *num * size* bytes and returns a generic pointer to the beginning of the block. Unlike **malloc**, **calloc** initializes the entire block with zeroes, sparing a programmer the trouble of using a loop for initialization.

Both of the following allocation statements create a block of memory that can hold an `int` array of 10 elements:

```
int *p;
p = malloc(10 * sizeof(int));
p = calloc(10, sizeof(int));                    Pointer to block created by malloc lost!
```

In either case, the size of the block is typically 40 bytes, but all bytes of the block created by **calloc** are initialized to zeroes. If initialization is not required, then you may use either of the two functions, even though the syntax of **calloc** is more intuitive when used for storing arrays.

Takeaway: Both **malloc** and **calloc** can create space for arrays except that (i) the arguments of **calloc** are array-oriented, (ii) **calloc** initializes the array elements to zero.

Program 16.5 uses **calloc** to create space for an array of structures containing the performance parameters of sportsmen. The size of this array is determined at runtime by user input. After the data is keyed in, the details of the sportsmen are displayed.

```
/* calloc.c: Allocates memory for an array of structures.
                  Size of array is determined at runtime. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   short i = 0, num;
   struct cricketer {
      char name[30];
      short runs;
      float average;
   } *p;                          /* Defines pointer to a structure */

   fputs("Number of structures? ", stderr);
   scanf("%hd", &num);

   p = calloc(num, sizeof(struct cricketer));
   if (p == NULL) {
      fputs("Cannot allocate memory, quitting ...\n", stderr);
      exit(1);
   }

   while (i < num) {
      fprintf(stderr, "Enter runs, average, name for element %hd: ", i);
      scanf("%hd %f %[^\n]", &p[i].runs, &p[i].average, p[i].name);
      i++;
   }
```

```
 fputs("\nPrinting each structure ...\n", stderr);
   for (i = 0; i < num; i++)
      printf("%20s %6hd %5.2f\n", p[i].name, p[i].runs, p[i].average);

   free(p);
   exit(0);
}
```

PROGRAM 16.5: **calloc.c**

```
Number of structures? 4
Enter runs, average, name for element 0: 6996 99.94 Don Bradman
Enter runs, average, name for element 1: 7249 58.45 Walter Hammond
Enter runs, average, name for element 2: 8032 57.78 Garfield Sobers
Enter runs, average, name for element 3: 15921 53.78 Sachin Tendulkar

Printing each structure ...
         Don Bradman   6996 99.94
      Walter Hammond   7249 58.45
      Garfield Sobers   8032 57.78
    Sachin Tendulkar  15921 53.78
```

PROGRAM OUTPUT: **calloc.c**

Note: Using **malloc** in tandem with the **memset** function, you can simulate the effect of **calloc**. **memset** uses three arguments: (i) a pointer to the block, (ii) the initial value, (iii) the number of bytes to be initialized. The functions have to be used in the following manner to initialize an array of 10 integers to zero:

```
p = malloc(sizeof(int) * 10);
memset(p, 0, sizeof(int) * 10);                    Must include <string.h>
```

memset assigns the same value to every *byte* of the block; you can't use it to set, say, each `int` array "element" to 1. That would imply setting every fourth byte to 1. Thus, zero is the only value we can use here.

16.9 realloc: CHANGING SIZE OF ALLOCATED MEMORY BLOCK

Prototype: `void *realloc(void *`*ptr*`, size_t` *size*`);`
Header File: `stdlib.h`
Return Type: A generic pointer to the reallocated memory on success, NULL on failure.

In spite of allocating memory at runtime, it is still possible for the allocation to be either excessive or inadequate. You would then need to resize the existing memory block with the **realloc** function. This function takes the old pointer (*ptr*) returned by a previous invocation of **malloc** or **calloc** as the first argument and the desired size (*size*) as the other argument.

realloc returns a pointer to the beginning of the reallocated block, or NULL on error. The following sequence that also uses **malloc** enlarges an existing block of 20 bytes to 100:

```
int *p, *q;
p = malloc(20);                                  Allocates 20 bytes
  ...
q = realloc(p, 100)                              Reallocates 100 bytes
```

If the requested size (100) is greater than the existing one (20), one of the following things will happen:

- **realloc** will try to enlarge the existing block to 100 bytes of contiguous memory and return the existing pointer (`p = q`) if successful.
- Otherwise, **realloc** will attempt to create a new block elsewhere and move the existing data to the new block. It will return a pointer (`q`) to this newly created block, free the old block and set `p` to NULL. C guarantees that there will be no loss or alteration of data during migration.
- If neither action succeeds, **realloc** will return NULL and leave the existing data intact.

In case `q` is different from `p`, you may still want to reassign `q` to `p` and continue to use the original pointer:

```
p = q;
```

In case you have overestimated the memory requirements, you can use **realloc** to shrink the existing block. The previous reallocation can thus be followed with this call to **realloc**:

```
r = realloc(q, 80)                               Shrinks size to 80 bytes
```

Program 16.6 uses **scanf** to accept any number of integers from the keyboard and save them in a memory block. The space for the first integer is created with **malloc**. This space is subsequently extended with **realloc** to store the remaining integers. The program progressively shows the number of bytes allocated along with the pointer to each of these blocks. Even though the pointer has not changed in this sample run, it need not necessarily be so, specially if the block becomes large.

```
/* realloc.c: Uses realloc to increase the size of allocated memory
                  every time another integer is read by scanf. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int *p, temp;
   short i = 0, num, newsize;

   fputs("Key in some integers, press [Enter] and [Ctrl-d]: ", stderr);
   while (scanf("%d", &temp) == 1) {
      i++;

      if (i == 1) {
         /* Allocate memory for 1st element of array */
         if ((p = malloc(sizeof(int))) == NULL) {
            fputs("Cannot allocate memory, quitting ...\n", stderr);
            exit(1);
         }
         printf("Allocated memory = %2d bytes, Pointer = %p\n", sizeof(int), p);
         *p = temp;               /* First integer allocated to memory block */
      }
```

```
      else if (i > 1) {                    /* For subsequent input ... */
         newsize = sizeof(int) * i;
         p = realloc(p, newsize);          /* ... memory reallocated */
         if (p == NULL) {
            fputs("Cannot reallocate memory, quitting ...\n", stderr);
            exit(1);
         }
         *(p + i - 1) = temp;              /* Populates reallocated block */
         printf("Reallocated memory = %2d bytes, Pointer = %p\n", newsize, p);
      }
   }  /* Matching brace for while */

   num = i;
   if (i > 0) {
      fputs("Printing array elements ...\n", stderr);
      for (i = 0; i < num; i++)
         printf("%d ", p[i]);
   }
   exit(0);
}
```

PROGRAM 16.6: **realloc.c**

```
Key in some integers, press [Enter] and [Ctrl-d]: 11 222 33 444 55 666
Allocated memory =  4 bytes, Pointer = 0x804b008
Reallocated memory =  8 bytes, Pointer = 0x804b008
Reallocated memory = 12 bytes, Pointer = 0x804b008
Reallocated memory = 16 bytes, Pointer = 0x804b008
Reallocated memory = 20 bytes, Pointer = 0x804b008
Reallocated memory = 24 bytes, Pointer = 0x804b008
[Ctrl-d]
Printing array elements ...
11 222 33 444 55 666
```

PROGRAM OUTPUT: **realloc.c**

This program provides the foundation for the remaining discussions in this chapter. The memory for each input integer is provided "just-in-time," i.e., when it is needed. Thus, to input *n* integers, you need to invoke **malloc** once and **realloc** *n* - 1 times. The program is inefficient because of multiple allocations, but the "just-in-time" technique illustrates how memory allocation is made for a linked list which we will now discuss.

16.10 THE LINKED LIST

Program data are often held in one or more *lists*. A list is simply a sequence of data items held together by a common access mechanism. An array is a linear list of data items where each item is identified by a unique array subscript. However, an array has some disadvantages that you are well aware of:

- The size of the array is fixed at compile time and can't be altered while the program is running. An undersized array breaks the program while an oversized one also wastes memory.
- It is inconvenient and inefficient to insert or delete data in an array. In the general case, existing data have to be moved to the "right" *before* insertion can begin, and moved "left" *after* deletion has been performed.

Both of these drawbacks are overcome by the *linked list*, which is a linear sequence of items or *nodes* connected by pointers. A node in a simple linked list is a structure that comprises the following members:

- One or more members containing the data.
- One member that represents a pointer to the next node.

A linked list exploits the self-referential property of a structure *(14.4)*. While a structure member can't represent another structure of the same type, a member can be a pointer to it. For instance, consider the following declaration that creates a structure named `node` containing two members:

```
struct node {
   short data;
   struct node *next;                    Pointer to structure of same type
};
```

The member named `data` contains the data, which here is a short integer. The second member named `next` is simply a pointer to its own structure. Figure 16.4 depicts a rudimentary linked list comprising three nodes where each node is connected to its next neighbor by the pointer representing the `next` member. As you'll soon discover, it must be possible to manipulate a linked list using *only* the information stored in the first node.

Is this scheme of linked structures superior to an array? In some cases, yes, because the limitation of fixed size no longer applies here. The memory for each node is dynamically allocated at the time of its creation and freed on its deletion. These operations also require resetting the pointers of the predecessor and successor nodes. A linked list can thus grow and shrink freely. You can't set its size, but then you don't need to.

However, a linked list has two drawbacks compared to an array. First, it takes up more space because of the presence of an additional member that links one node to the next one. Second, list access is sequential and not random; access must begin from the first node. For locating nodes toward the end of a large list, the access time can be quite significant.

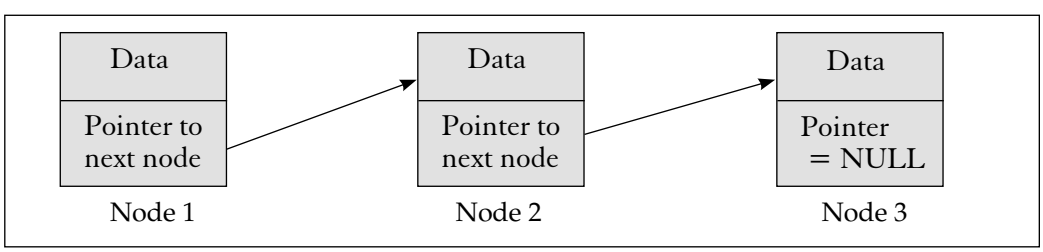


FIGURE 16.4 A Linked List of Three Nodes

16.10.1 Creating a Linked List with Variables

Using a separate variable to represent each node is certainly not the recommended way of creating a linked list. However, for an initial orientation, this elementary technique is conceptually helpful. Let's create this list by first defining three structure variables of type `struct node` that has been declared previously:

```
struct node n1, n2, n3;
```

Next, we assign the values 10, 20 and 30 to these three nodes using the familiar dot notation:

```
n1.data = 10;
n2.data = 20;
n3.data = 30;
```

The final step is to link these three nodes using the `next` member of each of these variables. This member in `n1` is assigned the address of `n2`, while `n2.next` is assigned the address of `n3` (Fig. 16.5). Like in a string, the last member of the list must be assigned the NULL value:

```
n1.next = &n2;
n2.next = &n3;
n3.next = NULL;
```

From these assignments, we can make a few observations straightaway. First, a linked list needs more memory compared to an array storing the same data. Second, a list can be completely examined if the address of the first node is known and the pointer of the last node is set to NULL. Finally, a program must be able to handle a linked list without knowing its size.

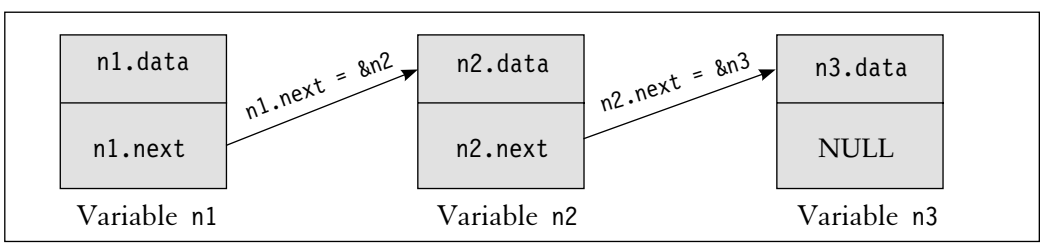


FIGURE 16.5 A Linked List of Three Nodes Represented by Variables

Note: Henceforth, we'll often refer to the `next` member of a node as the *.next* pointer because it represents the address of the next node.

Because these nodes are linked to one another, it is possible to print the value of a node by using the pointer of the previous node. The following statement prints the values 10, 20 and 30:

```
printf("%hd, %hd, %hd\n", n1.data, n1.next->data, n2.next->data);
```

The second node value is printed using the *.next* pointer of the first node and the third node value is accessed using the *.next* pointer of the second node. However, there's nothing in this technique

that can be considered significant. Instead, what is significant is that it is possible to print the same data using only the *.next* pointer stored in the first node (`n1.next`):

```
printf("%hd %hd, %hd\n", n1.data, n1.next->data, n1.next->next->data);
```

However, a linked list is not meant to be used with separate node identifiers (like `n1`, `n2`, etc.). A list of 100 nodes would then need 100 variables. List access must be general and should be achieved by following a *single* pointer. Also, the nodes themselves must be created on-the-fly while the program is running. Using variables to represent nodes won't do; we need **`malloc`**.

16.10.2 `create_list.c`: Creating a Linked List Using `malloc`

In real life, a list node is created by dynamic memory allocation. The pointer returned by **`malloc`** after allocation of a node is stored in the previous node, while the last node points to NULL. However, the address of the first node must be stored in a separate *head* pointer having the same data type as the node. To address the previous problem with **`malloc`**, we must first set the head pointer to the address of the first node and then set its data field using the head pointer:

```
struct node *head;
head = malloc(sizeof(struct node));          No cast needed
head->data = 10;                             Data of first node set
```

The `next` member of the first node can be assigned only after the second node is created. This process is repeated for all subsequent nodes except the last one (which is set to NULL). Figure 16.6 shows the schematic layout of a linked list that uses a head pointer but doesn't use variables to represent the nodes.

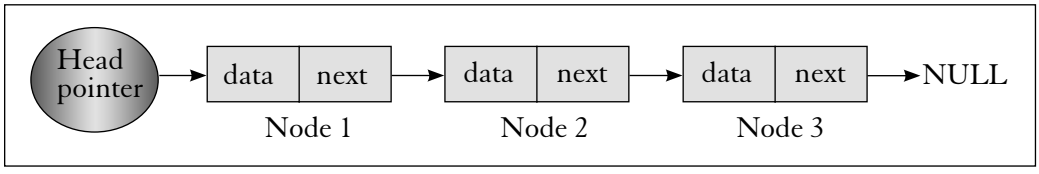


FIGURE 16.6 A Linked List of Three Nodes Preceded by Head Pointer

Program 16.7 shows a semi-real technique of creating a linked list of three nodes using a head pointer and a temporary pointer `p` for storing intermediate values. The contents of the list are printed using a **`while`** loop that doesn't use `p`.

Because the head pointer is key to accessing the entire list, we need to copy it to `current` and then use the latter in a loop to print the contents of each node. Every time the statement `current = current->next;` is executed, this pointer is incremented to point to the next node in the list. This technique allows a loop to access the entire list without knowing its size. However, we must also learn to use a loop to input data to the list and without using `p`.

Takeaway: Because of dynamic memory allocation, a linked list can grow freely without wasting memory. However, compared to an array, a list takes up more space because of the link field, i.e., the *.next* pointer present in each node.

```
/* create_list.c: Creates a linked list of 3 nodes or items.
                  Also prints their values and the node count. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   struct node {                  /* Declares structure for list element */
      short data;
      struct node *next;          /* The link field -- a pointer */
   };

   struct node *head, *p;
   struct node *current;
   short count = 0;

   head = malloc(sizeof(struct node));       /* Head pointer points to item 1 */
   head->data = 10;                          /* Item 1 populated */

   p = malloc(sizeof(struct node));          /* Memory allocated for item 2 */
   head->next = p;                           /* Item 1 linked to item 2 */
   p->data = 20;                             /* Item 2 populated */

   p->next = malloc(sizeof(struct node));    /* Memory for item 3 */
   p->next->data = 30;                       /* Item 3 populated */
   p->next->next = NULL;                     /* List ends here */

   current = head;                           /* Preserves value of head pointer */
   while (current != NULL) {
       printf("Value = %hd\n", current->data);
       current = current->next;
       count++;
   }
   printf("Number of items in linked list = %hd\n", count);
   exit(0);
}
```

PROGRAM 16.7: **create_list.c**

```
Value = 10
Value = 20
Value = 30
Number of items in linked list = 3
```

PROGRAM OUTPUT: **create_list.c**

16.10.3 Operations on Linked Lists

Most of the operations performed on arrays also apply to linked lists. Some operations are simpler while others are not. All but one of the following list operations are examined in this chapter:

- Adding a node at any position in the list.
- Deleting a node at any position in the list.
- Displaying the entire list.
- Computing the number of nodes.
- Searching a list for a specific value, and optionally adding or deleting a node in its immediate neighborhood.
- Sorting a list and inserting and deleting nodes in a sorted list.

We have performed most of these functions on an array, but only by moving the data around. With a linked list, however, we mainly need to realign the node pointers; the data remain at their original locations.

Irrespective of the operation performed, list access must begin from the first node using the head pointer. Unlike in an array, where you can randomly locate `arr[200]`, access to a node is sequential. To access the 200th node, you must begin from the head pointer and access all intermediate nodes before reaching the destination. Whether this overhead outweighs the advantages of flexible memory allocation is for the programmer to assess.

Note: It is standard programming practice to check for an empty (or null) linked list before performing operations on it. This check is made on the head pointer. However, the programs in this chapter have ignored this check.

16.11 ADDING A NODE

We'll now discuss the techniques used to perform the common operations on linked lists. We begin with the addition of a single node at any list location. A node is added by first creating memory for it and then assigning values to its data members. The linking mechanism is maintained by resetting the pointers in the immediate neighborhood of the point of insertion. A node can be added at the following locations:

- The beginning of the list. This requires the new node to point to where the head pointer was earlier pointing to. Also, the head pointer needs to be modified to point to the new node.
- The end of the list. This requires the predecessor node to point to the new node while the new node must point to NULL.
- Any other list position. This is generally done by searching for a value in the list and then adding a node either before or after the located node.

We'll examine the first two operations before developing a function for the third operation. After examining deletion at the terminal locations, we'll develop a simple program for demonstrating the basic list attributes.

16.11.1 Adding a Node at Beginning

Adding a node at the beginning of a list requires the new node to point to the one that was previously the first node, and the head pointer to point to the new node (Fig. 16.7). This operation involves the following steps:

- Allocate memory for the new node and set its data field if it is known at the time of allocation:

```
struct node *p;
p = malloc(sizeof(struct node));
p->data = 111;
```

- Set the *.next* pointer of the new node to point to the original first node. This latter node is accessed by the head pointer:

```
p->next = head;                    This assignment has to be made ...
```

- Reassign the head pointer to point to the new node:

```
head = p;                          ... before this one.
```

This is a simple and fast technique compared to the one used for insertion of an array element *(10.5.1)*. Simply create memory for the new node, assign values to one or more data fields and realign two pointers. Nothing can be simpler than that!

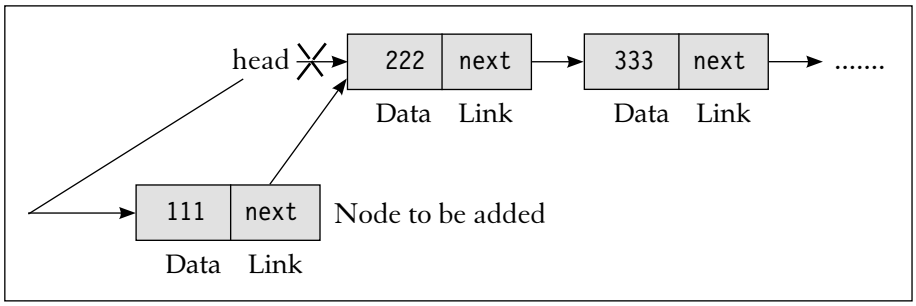


FIGURE 16.7 Adding a Node at Beginning of List

16.11.2 Adding a Node at End

For adding a node at the end of the list, first traverse the entire list to reach its end. The process may be iterative or recursive, and requires the following steps to be taken:

- Allocate memory in the usual manner for the new node and assign its data field if necessary. Also, set its *.next* pointer to NULL.
- For an iterative traversal of the list to reach its end, use a copy of the head pointer as a key variable:

```
current = head;
while (current->next != NULL)          Last node has .next set to NULL
   current = current->next;
```

- Set the *.next* pointer of the original last node (which was previously set to NULL) to point to the new node.

The assignment, `current = current->next;`, needs to be examined closely since it is used in many list operations. This statement changes the address of the current node in every loop iteration. It thus lets every node in the list to be accessed. However, since the traversal is sequential, it is slow compared to random access of an array element.

Tip: Don't lose the head pointer because it is the only tool you have to access a list node. For traversing a list, copy this pointer and use the copy as a key variable in a loop.

16.12 DELETING A NODE AT BEGINNING AND END

Like with addition, a node can be deleted at any point in the list. Deleting a node is a simple operation compared to insertion and it gets even simpler at the terminal points. Also, deletion at the non-terminal locations is normally preceded by searching for a value. Good discipline requires the allocated memory to be freed on deletion.

For deleting a node at the beginning of the list, the head pointer must point to the next node, but not before its original value has been saved:

```
p = head;
head = p->next;
```

Once that is done, the space used by the original first node can be freed:

```
free(p);
```

For deletion at the end of the list, traverse the list in the usual manner, but *you must also locate and save the address of the previous node*:

```
current = head;
while (current->next != NULL) {
   previous = current;
   current = current->next;
}
```

After loop iteration is complete, `current` points to the last node whose memory can now be freed. The pointer `previous` now points to the new last node, so `previous->next` should be set to NULL. Node deletion at the end is depicted in Figure 16.8.

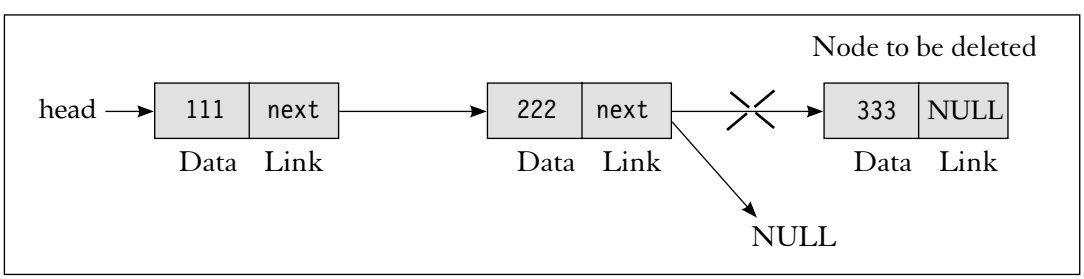


FIGURE 16.8 Deleting a Node at End of List

16.13 `head_tail_operations.c`: ADDING AND DELETING A SINGLE NODE

Program 16.8 demonstrates the addition and deletion operations at the beginning and end of a linked list. The program first creates a single-node list and then adds two nodes at the

beginning and end before removing them. At each stage, the contents of the list are printed using the **print_list** function which has been developed for this and the next program. Both program and output are well-annotated.

```
/* head_tail_operations.c: Inserts and deletes nodes at beginning and end
                           of a linked list. */
#include <stdio.h>
#include <stdlib.h>

struct node {                        /* node declared before main ... */
   short data;                       /* ... available throughout program. */
   struct node *next;
};

void print_list(struct node *head);

int main(void)
{
   struct node *head, *p, *current, *previous;

   /* Create head pointer */
   head = malloc(sizeof(struct node));
   head->data = 20; head->next = NULL;
   print_list(head);                            /* Prints 20 */

   /* Insert node at beginning of list */
   p = malloc(sizeof(struct node));
   p->data = 10;
   p->next = head;
   head = p;                                    /* Reset head pointer */
   print_list(head);                            /* Prints 10 20 */

   /* Insert node at end after ... */
   p = malloc(sizeof(struct node));
   p->data = 30;
   p->next = NULL;

   current = head;
   while (current->next != NULL)                /* ... locating end of list */
      current = current->next;
   current->next = p;
   print_list(head);                            /* Prints 10 20 30 */

   /* Delete node at beginning */
   p = head;
   head = p->next;
   free(p);
   print_list(head);                            /* Prints 20 30 */
```

```
  /* Delete node at end after ... */
   current = head;
   while (current->next != NULL) {            /* ... locating end of list */
      previous = current;
      current = current->next;
   }
   previous->next = NULL;                     /* previous is now last node */
   free(current);
   print_list(head);                          /* Prints 20 */

   exit(0);
}

void print_list(struct node *head)
{
   while (head != NULL) {                     /* head is local variable here */
      printf("%hd ", head->data);
      head = head->next;                      /* Doesn't change head in main */
   }
   printf("\n");
   return;
}
```

PROGRAM 16.8: **head_tail_operations.c**

```
20                                  First node created
10 20                               Node inserted at beginning
10 20 30                            Node inserted at end
20 30                               Node deleted at beginning
20                                  Node deleted at end
```

PROGRAM OUTPUT: **head_tail_operations.c**

The **print_list** function uses the head pointer as argument to print the value of the data member of all list nodes. The parameter head is, however, a local variable of the function. So, even though the value of head changes in every loop iteration, the changed value is not available outside the function. To print this list without using a function, copy head to a separate variable and use the copy as a key variable in the loop.

Note: The addition and deletion operations performed at the beginning of the list in this program represent the *push* and *pop* operations associated with a stack. This data structure is examined in Section 16.16.1.

16.14 list_manipulation.c: A LIST HANDLING PROGRAM

We will now explore the other list-handling operations, but this time, we'll develop a function for each one of them. Program 16.9 employs a number of these user-defined functions for handling

query and edit operations at intermediate locations in the list. Some of these functions are called directly from **main** and the others are invoked from these functions. This program performs the following tasks:

- Creates a list from user input.
- Traverses the list for counting and printing nodes.
- Searches a node for a user-specified value.
- Adds and deletes a node as a consequence of the search.

The program lists the function prototypes and the **main** function. The function definitions are shown in the following sub-sections where they are discussed. These discussions don't include the **print_list** function which has already been explained in Section 16.13.

The program first creates the linked list using **scanf**, **malloc** and the **add_node** function. Because all the user-defined functions use a pointer to `struct node` as argument, the structure itself must be declared before **main**. This makes the structure and its pointer global, i.e., visible in the entire file.

```
/* list_manipulation.c: A comprehensive program that uses 6 functions to
                        create, query, add and delete nodes of a linked list. */
#include <stdio.h>
#include <stdlib.h>

struct node {                                   /* Declaration before main */
   short data;
   struct node *next;
};

/* The function prototypes */
void add_node(struct node *p, struct node *q);
void print_list(struct node *head);
short count_nodes(struct node *head);
struct node *find_node(struct node *head, short val);
void insert_after(struct node *head, short val1, short val2);
void delete_node(struct node **head, short val);

int main(void)
{
   struct node *head, *p, *q;
   short count = 0, temp, val1, val2;

   /* Create the list first */
   fputs("Key in some short integers and press [Ctrl-d]:\n", stderr);
   while (scanf("%hd", &temp) == 1) {
      count++;
      q = malloc(sizeof(struct node));
```

```
      q->data = temp;
      if (count == 1)
         head = p = q;              /* p represents previous node */
      else {
         add_node(p, q);            /* Establishes links between two nodes */
         p = q;                     /* Assigns current node to previous node */
      }
   }

   q->next = NULL;                  /* Marks end of linked list */

   print_list(head);                /* Prints the list */
   printf("Number of nodes in list = %hd\n", count_nodes(head));

   /* Search for a value and insert a node after located node */
   fputs("Values to search and insert: ", stderr);
   scanf("%hd%hd", &val1, &val2);
   insert_after(head, val1, val2);
   print_list(head);

   /* Search for a value and delete the located node */
   fputs("Value to delete: ", stderr);
   scanf("%hd", &val1);
   delete_node(&head, val1);     /* Allows head pointer to be modified */
   print_list(head);

   exit(0);
}
   ... Functions definitions ...
```

PROGRAM 16.9: **list_manipulation.c**

```
Key in some short integers and press [Ctrl-d]:
11 22 33 44 66 77 88 99
11 22 33 44 66 77 88 99
Number of nodes in list = 8
Values to search and insert: 44 55
11 22 33 44 55 66 77 88 99
Value to delete: 66
11 22 33 44 55 77 88 99
```

PROGRAM OUTPUT: **list_manipulation.c**

16.14.1 The add_node Function

```
void add_node(struct node *p, struct node *q)
{
   q->next = p->next;                    Statement redundant for creating list
   p->next = q;
   return;
}
```

This function, which accepts two list pointers as arguments, adds the node q after node p. In the program, **add_node** is first called directly from **main** for creating the list. It is invoked again by the **insert_after** function for adding a node as a consequence of a search operation.

For list creation, **add_node** simply sets the *.next* pointer of the previous node to the address of the new node. However, for inserting a node between two existing ones, this function ensures that the new node points to where the preceding node was earlier pointing to (`q->next = p->next;`). The *.next* pointer in p is then made to point to q.

16.14.2 The count_nodes Function

```
short count_nodes(struct node *head)
{
   struct node *current;
   short count = 0;

   for (current = head; current != NULL; current = current->next)
      count++;
   return count;
}
```

This function accepts the head pointer as the sole argument and uses a **for** loop to count the number of nodes. Because the head parameter is a local variable, its assignment to current is not necessary. The following **for** statement would have worked just as well:

```
for (; head != NULL; head = head->next)
```

Using a copy of the head pointer arguably provides clarity. However, if you don't like the idea of using an extra local variable, use the revised **for** structure just shown.

16.14.3 The find_node Function

```
struct node *find_node(struct node *head, short val)
{
   struct node *current = head;

   while (current != NULL)
      if (current->data == val)
         return current;
      else
         current = current->next;
   return NULL;                        /* When search value not found */
}
```

A list is often searched for a specific value to determine the point of insertion or deletion of a node. In this program, the **find_node** function performs this task and returns a pointer to the node containing the value, or NULL otherwise. Since both the **insert_after** and **delete_node** functions use **find_node**, it makes sense to maintain the search mechanism as a separate function.

16.14.4 The insert_after Function

```
void insert_after(struct node *head, short val1, short val2)
{
   struct node *q, *found_node;
   found_node = find_node(head, val1);

   if (found_node == NULL)
      printf("Search value not found\n");
   else {
      q = malloc(sizeof(struct node));
      q->data = val2;
      add_node(found_node, q);
   }
   return;
}
```

The **insert_after** function adds a node, containing val2 in its data field, after the node containing val1 in its data field. The function first invokes **find_node** to locate the node containing val1 and creates a new node containing val2. Finally, **insert_after** calls **add_node** to link the pointers of the two nodes.

16.14.5 The delete_node Function

```
void delete_node(struct node **head, short val)
{
   struct node *found_node, *current;

   found_node = find_node(*head, val);
   if (found_node == NULL)
      printf("Search value not found\n");
   else {
      current = *head;
      while (current->next != found_node)        /* Locates predecessor node */
         current = current->next;

      if (found_node == *head)
         *head = found_node->next;
      else
         current->next = found_node->next;
      free(found_node);
   }
   return;
}
```

This function deletes the node containing val in its data field. It first invokes **find_node** to locate the node to be deleted. For deleting a node, it is also necessary to know its predecessor and link it with the node following the deleted one. Depending on whether the deletion occurs at the beginning of the list or later, the *.next* pointer of **found_node** is assigned to either *head or current->next. The memory allocated for the deleted node is finally freed and returned to the heap.

Since the deleted node can be the first one in the list, **delete_node** must be able to change the head pointer defined in **main**. For this reason, the function uses a *pointer to the head pointer* as an argument (**head instead of *head).

Space constraints don't permit us to include the other-list handling functions in this chapter. Since you have hopefully understood the concepts related to node manipulation, you should be able to develop these functions on your own.

We have not discussed the **print_list** function here because it has been examined before and used in Program 16.8. However, the definition for this function must be included in this program. (The prototype has been included.) Is it a good idea to include the code for a function in every program that uses it? Certainly not. The preprocessor discussed in Chapter 17 solves this problem.

16.15 TYPES OF LINKED LISTS

The linked list we have worked with so far uses a head pointer and a *.next* field in every node. For the last node, this field is set to NULL. Depending on the requirements, this model is often tweaked to support the following variations:

- *Circular linked list* This list doesn't have the *.next* field of the last node set to NULL. Instead, this field points back to the first node, thus creating a circular list. Access to this list no longer requires the head pointer; a pointer to any node is good enough.
- *List with a tail pointer* In addition to the head pointer, this list also has a tail pointer which points to the last node. Having this pointer facilitates operations at the end of the list. You don't need to traverse the entire list to reach the end; simply approach the list from the other end. However, insertion and deletion at the end require the tail pointer to be updated.
- *Doubly-linked list* Every node in this list has an additional pointer that points to the previous node. This simplifies operations that need to know the predecessor node—like sorting, adding or deleting a node *before* a specified node. However, these operations also result in increased overheads because two pointers for each node have to be modified.

Instead of using a linked list, an array of contiguous elements allocated on the heap can often provide a better solution. For instance, if the initial allocation of, say, 100 elements, is later found to be inadequate, a block of another 10 elements could be allocated using **realloc**, so that the array is always contiguous. Note that contiguous memory is accessed faster than non-contiguous memory, which in a linked list is done by following its *.next* pointers.

16.16 ABSTRACT DATA TYPES (ADTs)

C offers the primitive (like `int`) and derived data types (the array) for handling most of our needs. It also allows us to create our own types (`struct`) and build lists using them (linked list). These data types would be totally useless if the language didn't also support the "tools" to access and manipulate them. For instance, we use an `int` or `float` only because they work with the arithmetic and relational operators. Consider it this way: the `+` operator is simply a *method* to perform an operation on an `int`.

Computer science offers the *abstract data types (ADTs)* like stacks, queues and trees. We call them "abstract" because they have an open implementation and are not native to a language. For instance, a data structure like a stack has an open implementation; both an array and linked list can represent a stack. Just as an `int` is operated upon by the `+`, abstract data types are also associated with a set of *methods* that can perform operations on the data they hold.

Abstract data types have to contend with overflow and underflow situations. An attempt to insert an element to a stack or queue that is full leads to an *overflow* situation. Similarly, an attempt to delete an element from a null or empty list results in an *underflow* situation. These checks are made right at the beginning of the operation. We'll briefly examine the well-known ADTs and the essential methods used with them.

16.16.1 The Stack

You are already familiar with the *stack* as a container used by functions to store their parameters and local variables. This data type operates on the last-in-first-out (LIFO) principle, i.e., data that is last *pushed* onto the stack is the first to be *popped* out. Operations on a stack are carried out only at its beginning (head pointer for a linked list). The following methods (or functions) are commonly associated with a stack:

- *Push* This operation inserts a data element at the top of the stack (if the stack is not full). We have already performed this operation in Program 16.8 by inserting a node at the beginning of the list.
- *Pop* This operation removes the element that was last inserted into the stack (if the stack is not empty). This too has been demonstrated in Program 16.8 by deleting the first node.
- *Peek* This operation evaluates the value at the top of the stack but it doesn't delete the value.

Other operations include counting the number of elements and checking whether the stack is empty or full. A stack is often compared to a pile of plates that are added and removed only at the top. Office software (like Word and Excel) use a stack for performing undo/redo operations. The Web browser uses a stack for implementing the forward/back feature that allows a user to repeatedly step back and forth between previously visited Web pages.

16.16.2 The Queue

The *queue* represents exactly what we see in real life—queuing up for entry into, say, a sports field, bus or for casting votes. This abstract data type operates on the first-in-first-out (FIFO) principle; what goes in first also comes out first. Like a stack, a queue can be implemented by an array or a linked list.

An element is inserted into a queue from its rear or tail end, while deletion takes place from the front or head end. For a linked list, two pointers are used to locate the first and last item of the queue. A queue is associated with the following functions:

- *Enqueue* This function adds an element to the tail end of the queue (if the queue is not full).
- *Dequeue* This function deletes an element from the front end of the queue (if the queue is not empty).

- *Peek* or *Front* Like in the stack, this function returns the value of the first element without deleting it.

Other supported functions include returning the size of the queue and checking whether the list is empty or full. The operating system of a computer uses queues for handling print jobs and servicing interrupts. The job that is scheduled first for printing is also printed first. An interrupt that is received first is also serviced first.

Takeaway: The *pop* operation in a stack deletes the last element that was pushed. In contrast, the *dequeue* operation in a queue deletes the *first* element that was queued.

Note: All programs that insert or delete elements in a stack or queue must first check whether the data structure is full or empty.

16.16.3 The Tree

Unlike a stack or queue which are linear data structures, a *tree* is a non-linear one. Elements in a tree are organized in a hierarchical manner, much like the file system used by MSDOS/Windows and UNIX/Linux systems. Thus, every tree has a single *root* node at the top from where multiple nodes emanate. Each of these nodes in turn generate further nodes and so on until the bottom of the tree is reached (Fig. 16.9). Nodes in trees have *levels*; the root node is at level 0.

Like with the file system, nodes have a *parent-child* relationship between them. However, the root node has no parent and the terminating nodes—also called *leaves*—have no children. All children having a common parent are known as *siblings* who are connected to their parent by pointers in the parent node.

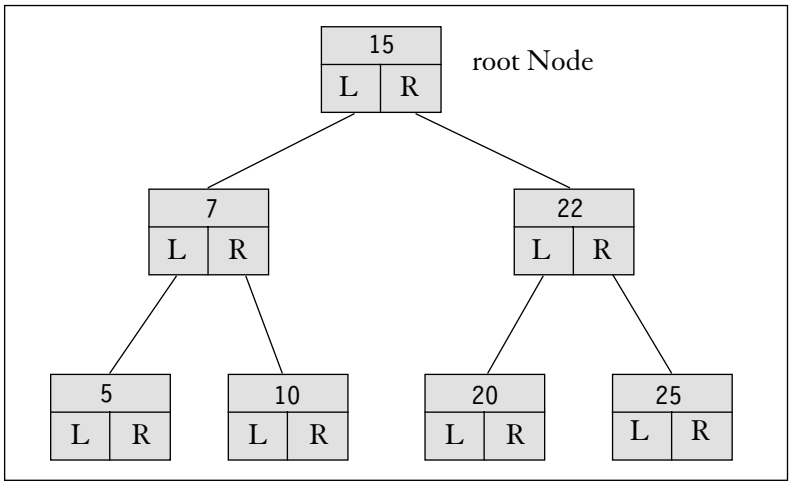


FIGURE 16.9 An Ordered Binary Tree

For a *binary* tree, each parent has zero to two children. This is represented in C by a structure containing a data field, a left and a right pointer. The complete tree can be traversed by following these pointers. For leaves, these pointers are set to NULL.

In an ordered binary tree (also called *binary search tree—BST*), the left sibling has a lower value than its parent, while the right sibling has a higher value. Searching begins from the root node and each descent to the next level halves the search domain. This makes search operations on an ordered tree faster than on a sorted linked list. Recall that binary search on an array also uses a similar algorithm *(10.10)*. A tree or sorted array containing 4 billion elements can be searched completely in only 32 attempts!

There are well-defined algorithms for traversing the entire tree, but it is beyond the scope of this text to discuss them. Trees are the preferred data type for hierarchical data. Sorted data lists can be conveniently manipulated when they are maintained in trees. Also, the well-known operating systems maintain the file system as a tree where directories act as tree nodes.

WHAT NEXT?

What does one do when the same function or symbolic constant is used by multiple programs? The solution lies in storing them at a single location and using *preprocessor directives* to make them available to every program that needs them. We must also be able to make programs portable by controlling the compilation process depending on the machine a program is meant to be run on.

WHAT DID YOU LEARN?

Memory can be allocated *dynamically* while a program is running. The allocated contiguous block can be treated like an array. Unlike a static array, the size of a dynamically created array can be changed at runtime.

The size of the allocated block can be specified in bytes (**`malloc`**) or the number of array elements (**`calloc`**). An allocated block can also be resized (**`realloc`**) or freed when it is no longer required (**`free`**).

The memory allocation functions return a generic pointer to the beginning of the allocated block. This pointer is automatically aligned to the right data type without using a cast.

A *memory leak* is caused by making a pointer point to a new block without freeing the old block that it was earlier pointing to. A *dangling pointer* to a freed block must be set to NULL or pointed to a different block.

A *linked list* is a linear collection of structures represented as *nodes*. Nodes are created by dynamic memory allocation and they are connected by pointers. Unlike an array, a linked list can be expanded or contracted.

Adding and deleting nodes require the realignment of node pointers in the immediate vicinity of the action. However, node access is sequential and is thus slower than random access of an array.

Abstract data types (ADTs) have an open implementation (like a stack). They are associated with a set of *methods* that access and manipulate the data.

A *stack* operates on the LIFO principle while a *queue* is based on FIFO. For a stack, the first element to be *pushed* in is also the last to be *popped* out. For a queue, the first element to be *enqueued* is also the first one to be *dequeued*. A *tree* is used for handling hierarchical data.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

16.1 A block of memory allocated dynamically can be assigned a name.

16.2 Once memory has been allocated dynamically, its size can be increased but not decreased.

16.3 The memory allocation functions return -1 on error.

16.4 It is possible for **`malloc`** and **`calloc`** to fail even if the system has adequate memory.

16.5 The block of memory created by **`malloc`** and **`calloc`** need not be contiguous.

16.6 It is possible to create a structure in a memory block allocated by **`malloc`**.

16.7 The **`free`** function can deallocate memory used by program variables.

16.8 A memory block allocated inside a function is automatically deallocated when the function terminates.

16.9 If the pointer to a dynamically allocated block is later made to point elsewhere, the previous block is automatically freed.

16.10 Unlike **`malloc`**, **`calloc`** initializes all bytes in the allocated memory block to zeroes.

16.11 If **`realloc`** cannot extend the existing memory block, it returns NULL.

16.12 A linked list can grow in size but not shrink.

16.13 A linked list takes up more space than an array containing the same data.

16.14 To access the entire data in a linked list, it is not necessary to know its size.

16.15 It is possible to access the entire list if the address of one of the nodes is known.

A2. FILL IN THE BLANKS

16.1 If a program needs memory while it is running, it has to invoke _____ or _____ .

16.2 The memory allocation functions return a _____ pointer on success.

16.3 The pointer returned by **`malloc`** and **`calloc`** can be aligned to any data type without using a _____ _____ .

16.4 The _____ function deallocates memory.

16.5 Freeing a memory block without setting its pointer to NULL creates a _____ _____ .

16.6 **realloc** uses the pointer returned by _____ or _____ as an argument.

16.7 For automatic initialization of the bytes of a memory block to zeroes, the block must be created by _____ .

16.8 While an array comprises elements, a linked list comprises _____ .

16.9 The *.next* member of the last node of a linked list has the value _____ .

16.10 Stacks and queues are implementations of the _____ data type.

A3. MULTIPLE-CHOICE QUESTIONS

16.1 Dynamic memory is allocated on the (A) stack, (B) heap, (C) region storing program code, (D) A and B.

16.2 If **malloc** is called repeatedly and its returned value is assigned to the same pointer variable, (A) a single memory block is allocated, (B) multiple blocks are allocated, (C) heap may become full, (D) B and C.

16.3 If a memory block is not freed, (A) a memory leak occurs, (B) a dangling pointer is created, (C) it is automatically cleared on program termination, (D) B and C, (E) A and C.

16.4 **realloc** returns a pointer (A) to the existing allocated block, (B) that is different from the existing block, (C) that may or may not be different from the existing block, (D) that may be NULL, (E) C and D.

16.5 Access to a linked list (A) is random, (B) is sequential, (C) begins from the head pointer, (D) B and C, (E) A and C.

16.6 The *.next* member of a linked list is assigned a value (A) after the next node is created, (B) before the next node is created, (C) NULL if no further node is created, (D) A and C, (E) B and C.

A4. MIX AND MATCH

16.1 Match the data structure with its attribute:

(A) Stack, (B) queue, (C) tree

(1) FIFO, (2) hierarchy, (3) LIFO

16.2 Match the linked list with its attribute:

(A) Circular list, (B) list with tail pointer, (C) doubly-linked list.

(1) Pointer to previous node, (2) last node not set to NULL, (3) access from end of list.

CONCEPT-BASED QUESTIONS

16.1 Explain the possible consequences of dynamically creating memory inside a user-defined function.

16.2 Consider the following statements that create space for an array of eight ints:

```
p = malloc(8 * sizeof(int));
p = calloc(8, sizeof(int));
```

Assuming that int uses four bytes, will the same statements work for storing an array of 16 two-byte short integers or a string of 32 characters? Explain with reasons.

16.3 Why should you never write code in the following way?

```
int *p = malloc(20);
*p = 10;
printf("*p = %d\n", *p);
p = calloc(10, sizeof(short);
```

16.4 List the steps involved in adding and deleting a node in a linked list at any location other than the beginning or end.

16.5 Compare the attributes of the array and linked list and explain the circumstances where one is preferred to the other.

PROGRAMMING & DEBUGGING SKILLS

Use of array notation is not permitted for the exercises in this section.

16.1 Point out the errors in the following program:

```
#include <stdio.h>
#include <stdlb.h>
int main(void)
{
   int *p;
   p = malloc(1, sizeof int);
   *p = 200;
   printf("*p = %d\n", *p);
   return 0;
}
```

16.2 Write a program to accept two integers from the keyboard. Using **malloc**, save them in a single memory block. Print the numbers and their average without using a temporary variable.

16.3 Modify Program C16.2 one to accept the integers as command-line arguments. Include error checking to ensure that two arguments are provided by the user.

16.4 Write a program that uses **malloc** to create space for storing 10 integers. Using a loop and the **pow** function of the math library, fill up the space with values of 2 raised to the power *n* where *n* varies from 1 to 10. Print the values.

16.5 Write a program that uses **calloc** to create a memory block for storing 10 integers. Save the values of two integers *at the end of the block* with user input and print the contents of the entire block using a loop. Could the output differ if **malloc** is used instead of **calloc**?

16.6 Write a program that uses **calloc** to create space for an array of 10 pointers to strings. Key in a string (maximum length = 80), save it using **malloc** and store its pointer in the fifth array element.

16.7 Write a program that uses **calloc** for storing an array of two integers and initializes the elements to 1234 and 4321. Overwrite the same space to store four short integers that are input from the keyboard. Print both sets of values.

16.8 What is the following code meant to do and why doesn't it work? What does **printf** otherwise display?

```
short *p = malloc(8);
printf("Size of block = %d\n", sizeof(p));
*p = 10; *(p + 1) = 20;
free(p + 2);
```

16.9 Write a program that creates an *optimized* structure comprising the name of the region (maximum = 100 characters), carpet area (maximum = 10,000 sqft) and rate/sqft (maximum = Rs 20,000). Enter the details of two apartments, store the data of one using **calloc** and the other using **realloc**.

16.10 Write a program to create space for an array of 10 int elements using **calloc**. Assign a *few* elements with non-zero values input from the keyboard using EOF to terminate input. Compute the count of values *by examining the memory block*. Print the count and the entered values before resizing this memory block with **realloc** to free unused space. Will this program work with **malloc** instead of **calloc**?

16.11 Write a program to create a linked list of two nodes containing a data field of type short and the *.next* field. Populate the two nodes with keyboard input and **malloc**, and print their contents.

16.12 Modify Program C16.11to repeat the exercise for *n* nodes where *n* is keyed in by the user. Print the contents of the list in reverse order.

17 The Preprocessor and Other Features

WHAT TO LEARN

- How the preprocessor enhances the productivity of a programmer.
- Creating *named constants* and *macros* with **#define**.
- Using **#include** to make common declarations available to multiple programs.
- Conditionally executing program segments with **#ifdef** and **#if**.
- Setting and clearing individual bits in an integer with the *bitwise* operators.
- Executing a function by passing its pointer as an argument to another function.
- Designing functions that accept a variable number of arguments.
- Organizing a program across multiple files.

17.1 THE PREPROCESSOR

The preprocessor is a separate program that is invoked at the beginning of the compilation process. Using special instructions, called *directives*, the preprocessor modifies a source program before it is seen by the compiler. A directive begins with the # symbol followed by a keyword and other text. For instance, the preprocessor acts on the directive, `#define SIZE 20`, by replacing `SIZE` with `20` wherever it occurs in the program. All directives are eventually removed from the source code so the compiler never gets to see them.

The preprocessor helps create code that is easy to read, maintain and port to multiple machines. Initially, it was not considered to be a part of the C language (even though the directives are included in a C program) until ANSI included it in the C standard (C89 onwards). Preprocessor directives can be categorized into three types (keywords shown in parentheses):

- *Macros* (**#define**) with and without function-like parameters.
- *File inclusion* (**#include**) that enables the contents of a file to be included in a program.
- *Conditional compilation* (**#if**, **#else**, **#endif**, etc.) that enables or disables compilation of certain sections of program code.

The **#define** or macro feature enables the use of an alias to replace a constant or a program segment that is used repeatedly. **#define** can also be used to replace simple functions. We haven't used the latter feature yet, but we have used the other feature in this text:

```
#define SIZE 40                                          Replaces a constant
#define FLUSH_BUFFER while (getchar() != '\n');          Replaces a statement
```

The **#include** feature places the contents of an external file at the point where the directive occurs. Every program in this text contains the following directive that places the contents of the file `stdio.h` in the program:

```
#include <stdio.h>
```

The conditional compilation feature uses a set of directives that begin with **#if** or **#ifdef** and end with **#endif**. Their "control expression" determines whether or not to include a program segment in the compilation process.

The preprocessor is usually a separate program which is automatically invoked by the compiling software. In Linux systems, this program is represented by the program **cpp** (in `/usr/bin`). In Visual Studio, the executable **cpp.exe** performs the same job. This is how you can generate *only* the preprocessed output without invoking the compiler:

```
gcc -E foo.c                                     Displays preprocessed output on screen
cpp foo.c foo.out                                Saves preprocessed output in foo.out
```

The preprocessor looks for syntax errors in *directives*, i.e., lines that begin with a #. The compiler checks *statements* that end with a semicolon. Note the following attributes of preprocessor directives:

- The # prefix may or may not be followed by whitespace. For instance, both **#define** and **# define** are valid directives. Also, since C89, it is not necessary for the # to be located at column 1.
- Unlike C statements, preprocessor directives don't end with a semicolon. If the preprocessor finds one, it either ignores it or treats it as part of the replaced text (for a **#define**). For the latter case, the compiler may output an error.
- Comments can be used in preprocessor directives in the same way (with /* and */) they are used in program statements. Like the compiler, the preprocessor simply removes them.
- Most directives use a single line of text. In case a directive needs multiple lines, precede the *[Enter]* key for every line (except the last) by a \. For instance, the following **#define** directive prints the entire string on a single line:

```
#define STRING "This is a multi-line \
                text."                           Spaces before text preserved
```

So far, we have used the bare essentials of **#define** and **#include**. After these features are examined in detail, you should be able to use a **#define** to replace a function with a macro, or place **#define** and function declarations in a separate file and "include" them in a program. You should also be able to exclude certain parts of a program from the compilation process.

Takeaway: The preprocessor performs two tasks. First, it modifies the source code in accordance with the directives provided. Second, it removes all directives from the source code before it is passed on to the compiler.

17.2 #define: MACROS WITHOUT ARGUMENTS

The **#define** directive defines an alias for text that may be repeated in one or more programs. An alias is commonly known as a *macro* and in this section, we'll discuss the macro in its simplest form—without arguments. The directive conforms to the following syntax:

```
#define identifier replaced_text
```

When the preprocessor encounters a **#define** directive in a program, it makes a global replacement of *identifier* with *replaced_text*. Consider the following example that you have encountered before:

```
#define PI 3.142
```

This directive replaces all instances of `PI` in the program with `3.142`. The compiler thus sees only `3.142` and not `PI`. The constant `3.142` now has a name called `PI` and this macro is also called a *named constant*, *symbolic constant* or *defined constant*. We'll mostly use these names to refer to a macro without arguments.

`PI` resembles a variable and uses the same naming convention (comprising letters, digits and underscores). For instance, `YOUR_ANSWER` and `__TIME__` are valid names for symbolic constants. Although not mandatory, symbolic constants are usually defined in uppercase. It's thus easy to distinguish them from variables which are normally defined in lowercase.

However, `PI` is not a variable and it is not assigned a value using the `=` operator. Furthermore, `PI` cannot be reassigned (say, to `3.142857` for greater precision) without first undefining it with **#undef**. (GCC simply issues a warning.) Here's a code fragment that assigns `PI` twice in a program:

```
#define PI 3.14                          No ; terminator
int main(void)
{
   printf("%f\n", PI);                   Prints 3.140000
   #undef PI                             Must undefine PI ...
   #define PI 3.142857                   ... before redefining it
   printf("%f\n", PI);                   Prints 3.142857
   ...
}
```

Note that preprocessor directives can be placed anywhere in a program even though they are normally bunched together at the beginning. Unlike with a variable, the scope of a named constant extends from the point of its definition to the end of the file. Thus, a named constant defined in a function is also visible in all subsequent functions; there are no "local" constants.

It's easier to program using named constants than with the numbers and text they represent. For instance, the constant `INR_USD` is easier to remember than the number `64.65`. If this constant

is used several times in a program and the dollar-rupee rate changes, you need to change only the **#define** statement.

Tip: Always use a named constant to specify the number of elements in an array. This constant can then be subsequently used to populate and print the array:

```
int i, arr[SIZE];
for (i = 0; i < SIZE; i++)
   scanf("%d", &arr[i]);
for (i = 0; i < SIZE; i++)
   printf("%d ", arr[i]);
```

The advantage of using `SIZE` instead of an integer is that the size of the array can be changed without disturbing this code.

Note: The preprocessor makes a literal replacement of a named constant with its defined value globally across the program. The data type of the constant is thus set by the compiler using its own rules. For instance, the compiler doesn't see `PI`, but it sees `3.142`, which it treats as a `double`.

17.2.1 Using Numbers and Expressions

The replaced text can be anything as long as the statement where it is literally substituted is syntactically correct *from the compiler's point of view*. This text can be a number, expression or a string comprising one or more *tokens*. (A token is a sequence of one or more non-whitespace characters.) As the following examples illustrate, tokens can also include another symbolic constant:

```
#define AREA 10 * 20                    Computation made by compiler
#define ROWS 3
#define COLUMNS 5
#define SIZE (ROWS * COLUMNS)           Replaced text contains named constants
```

The first directive literally substitutes the expression `10 * 20` into the program, but if this expression forms a component of a bigger expression (say, `400 / AREA`), the result would be 800 and not 2 (400 / 10 = 40, which is then multiplied by 20). To override the default precedence rules, you must enclose the expression in parentheses:

```
#define AREA (10 * 20)
```

The fourth example uses two previously defined symbolic constants (`ROWS` and `COLUMNS`) as components of the replaced text. Note that it is not necessary for `ROWS` and `COLUMNS` to be defined before they are used in `SIZE`. The following sequence is also valid:

```
#define SIZE (ROWS * COLUMNS)           SIZE defined ...
#define ROWS 3                          ... before ROWS ...
#define COLUMNS 5                       ... and COLUMNS
```

The guiding principle here is that a symbolic constant must be completely expanded before it is *used*. Thus, it doesn't matter whether `ROWS` and `COLUMNS` are defined before or after `SIZE`.

However, a symbolic constant is not expanded inside a string. For instance, SIZE in the control string of the following **printf** statement will not be expanded:

```
printf("SIZE = %d\n", SIZE);                    Prints SIZE = 15
```

The problem of expanding a named constant inside a string is handled with the # and ## operators. These symbols work only with function-like macros and are discussed in Section 17.4.

Note: Macro expansion is disabled when the identifier to be expanded is embedded in a string.

17.2.2 Why Not Use a Variable?

In most cases, a variable can serve the same purpose as a named constant. We can use a variable named size instead of the symbolic constant, SIZE, and still ensure that the value of size is set at a single point in the program. If we additionally use the **const** qualifier, we can also ensure that this value is not accidentally changed by the programmer.

While a variable is stored in memory, a named constant is stored in the program executable. Every time a variable is accessed, the program has to access its location in memory. Moreover, there are certain things you can do with a symbolic constant that you can't do with a variable. For instance, a symbolic constant can abbreviate a C statement and can even be used as a function. Named constants are thus more efficient, so use them wherever you can.

17.2.3 Abbreviating Text

Even though we commonly use symbolic constants to represent numbers, they can also be used to abbreviate a sequence of characters (tokens). We often use **#define** to abbreviate a **printf** statement that is repeated in the program. Consider the following directive:

```
#define ERROR_STRING  printf("Not a positive integer\n")        No semicolon
```

After **ERROR_STRING** is defined, you can "execute" it in the following manner:

```
if (x <= 0)
   ERROR_STRING;                                    Semicolon used here
```

Note that the semicolon is missing in the definition but it is provided in the invocation. Conversely, we could use a semicolon in the definition and drop it in the invocation. You'll recall that we had used the **FLUSH_BUFFER** macro *(9.10.4)* like this:

```
#define FLUSH_BUFFER while (getchar() != '\n');
FLUSH_BUFFER                                        No semicolon
```

However, in the present case, even if you invoke **FLUSH_BUFFER;** (with the **;**), the compiler won't complain. This is because the text while (getchar() != '\n');; generated after macro expansion is still a valid C statement (containing two NULL statements).

It is also not necessary for the expanded text to be a valid keyword or statement. Virtually any string can be part of the expanded text. Some programmers make the mistake of using = when == should

have been used. Others find AND and OR easier to remember than && and ||. You can define aliases for them,

```
#define EQUALS ==
#define AND &&
#define OR ||
```

and then use an **if** statement in the following way:

```
if (x EQUALS 10 AND y EQUALS 4)
```

The **#define** feature lets you use BEGIN and END to mark the two ends of a program. Simply use the following named constants for **main** and the final closing brace:

```
#define BEGIN int main(void) \          Continues to next line
       {                                Beginning of main
#define END }                           End of main
```

Over-enthusiastic programmers sometimes go overboard in abbreviating code. Beyond a point, however, this exercise becomes counter-productive. Keep in mind that you are not programming in Pascal, but in C, and you should use its keywords and operators as much as possible.

Tip: Don't abbreviate C keywords and symbols because they tend to affect your understanding of code. Eventually, you'll have to revert to the original code for comfort.

17.3 #define: MACROS WITH ARGUMENTS

The **#define** directive can create function-like macros which use one or more parameters but don't formally return a value. The "body" of this type of macro is evaluated as an expression and the resultant value is made available to the program. Consider the following definition of a macro named **CUBE** which raises a number to the power 3:

```
#define CUBE(y) y * y * y
```

Make sure that there is no space between CUBE and (. Otherwise, **CUBE** would be treated as a named constant and (would represent the beginning of the expanded text. Once properly defined, this macro can be invoked in the following manner:

```
int y = 4;
printf("CUBE(y) = %d\n", CUBE(y));          Prints CUBE(y) = 64
```

The preprocessor leaves the CUBE(y) in the string unaltered, but it replaces the second CUBE(y) with y * y * y. The actual computation is, however, performed only at runtime. The compiler thus sees the following statement:

```
printf("CUBE(y) = %d\n", y * y * y);
```

CUBE(y) could have been defined as a function, but before you decide to use a macro in preference to a function, note the following differences between them:

- A macro is only defined and not declared.
- The parameters of a macro have no specific data types.
- The arguments used to invoke a macro are not copied; they are simply substituted into the text.
- A macro doesn't return a value; the value of the expression is interpreted as the return value.

Thus, **CUBE(y)** can work without modification if y is a float or char. Because a macro doesn't use the stack, for simple computations, use a macro instead of a function.

17.3.1 When Parentheses Are Required

There is a problem with the **CUBE** macro. With y set to 4, what happens if you invoke **CUBE(y + 1)**? Because the substitution occurs literally, **printf** sees the following expression as its matching variable:

```
y + 1 * y + 1 * y + 1
```

This is $4 + 1 \times 4 + 1 \times 4 + 1 = 13$

The result is 13 and not 125 because the intention has been overridden by the normal precedence rules. The problem is easily fixed by enclosing each variable in parentheses:

```
#define CUBE(y) (y) * (y) * (y)
```

On invoking **CUBE(y + 1)** with this revised definition, the expanded text now becomes

```
(y + 1) * (y + 1) * (y + 1)
```

This is $5 \times 5 \times 5 = 125$

The evaluation is correct this time, but then there can be another problem if **CUBE(y + 1)** forms the denominator of an expression. For instance, the following statement,

```
printf("Value = %d\n", 500 / CUBE(y + 1));
```

wrongly prints 2500 instead of 4 (500 / 5 * 5 * 5 = 2500). The solution here is to enclose the entire expression within parentheses:

```
#define CUBE(y) ((y) * (y) * (y))
```

This definition correctly evaluates 500 / CUBE(y + 1) to 4. When using a macro for evaluating an expression, make sure that you enclose in parentheses, not only every numeric operand, but also the entire expression. A function, however, doesn't suffer from this restriction.

17.3.2 Useful Macros

We use a function to abbreviate a code segment that is repeatedly used in a program. In many cases, the same task can be performed by a macro. For instance, a macro can easily convert a temperature from one unit to another. The following macro definition is meant to convert a temperature from Fahrenheit to Celsius:

```
#define F2C(x) ((x - 32.0) * 5 / 9)
printf("Temp = %.2f\n", F2C(104));
```

Prints Temp = 40.00

To consider another example, we can design a macro to determine the maximum number in a pair of two integers:

```
#define MAX(x, y) ((x) > (y) ? x : y)
printf("Max = %d\n", MAX(1, 5));                    Prints Max = 5
```

Because macros can be nested, we can replace one of the two arguments with the same macro to handle three numbers:

```
printf("Max = %d\n", MAX(15, MAX(5, 25)));          Prints Max = 25
```

Some functions of the standard library are implemented as macros. The **getc**/**putc** and **getchar**/**putchar** functions are offered as macros even though function versions are also available. We can design our own case-conversion macro, **TOLOWER**, as an alternative to the **tolower** C function:

```
#define TOLOWER(x) ((x) >= 'A' AND (x) <= 'Z' ? (x) + 'a' - 'A' : (x))
#define AND &&
```

Note that we replaced && with the symbolic constant AND in the definition of **TOLOWER**. The sequence of these definitions doesn't matter as long as AND has been expanded before **TOLOWER** is used in the program. In the following code segment, this macro displays the contents of a string in lowercase:

```
char *stg = "REDMI NOTE 5";
for (i = 0; stg[i] != '\0'; i++)
   printf("%c", TOLOWER(stg[i]));                   Prints redmi note 5
```

Note: The C standard allows a function to be implemented as a macro but it also requires the function version to be supported. Thus, **tolower** and **toupper** exist both as macros and functions in every C distribution. When a library function exists in both forms, the macro version is used by default. To use the function version, disable the macro with **#undef**.

17.3.3 swap_with_macro.c: Swapping Two Numbers Using a Macro

Can we swap two variables using a macro? We couldn't do it using a function except by passing pointers to these variables as arguments. Unlike a function, a macro doesn't copy its arguments, so the **SWAP** macro in Program 17.1 easily performs this task. The program works with both integer and floating point variables but may not work in Visual Studio.

```
/* swap_with_macro.c: Uses a #define macro two swap two numbers. Same macro
                      definition works for int and float data types */
#include <stdio.h>
#define SWAP(x, y, type, temp) type temp = x; x = y; y = temp

int main(void)
{
   int   i1 = 5,   i2 = 10;
   float f1 = 5.5, f2 = 10.5;

   SWAP(i1, i2, int, t1);
   printf("i1 = %d, i2 = %d\n", i1, i2);     /* Numbers swapped without ... */
   SWAP(f1, f2, float, t2);
```

```
    printf("f1 = %.2f, f2 = %.2f\n", f1, f2); /* ... calling a function. */
    printf("t1 = %d, t2 = %.2f\n", t1, t2);

    return 0;
}
```

PROGRAM 17.1: **swap_with_macro.c**

```
i1 = 10, i2 = 5
f1 = 10.50, f2 = 5.50
t1 = 5, t2 = 5.50
```

PROGRAM OUTPUT: **swap_with_macro.c**

17.3.4 Functions vs Macros

Functions can do everything that macros can do, but the reverse is not true. Does that mean we use only functions and avoid macros? No, certainly not. Memory and speed considerations often determine the choice of the construct. Consider the following comparison of a macro and function:

- *A function needs to know the data type of its parameters.* On the other hand, the parameters of a macro have no data type. For this reason, the same macro often works with different data types. The **CUBE** macro works with both `int` and `float` but if **CUBE** is defined as a function, two separate versions will be needed.
- *A function can return a value using the* **return** *statement.* A macro doesn't support a **return** statement, but it can still "return" a value by evaluating the expression that forms the body of its definition. The **CUBE** macro "returns" this value by evaluating its replacement text, i.e., the expression `((y) * (y) * (y))`.
- *A function uses a stack but not a macro.* Pushing and popping data in and out of a stack involve overheads that can slow down operations. If a program calls a function repeatedly (say, in a loop), these overheads can be significant. Use of a macro in this situation could be a better option.
- *Macro expansion results in the substitution of the replaced text at the point of occurrence of the macro.* If the macro occurs at multiple places in a program, the program size would increase and this could impact performance.

Simple situations that are data type-independent and don't involve returning a value are conveniently handled by a macro. Also, if a recursive problem can be handled by a macro, it's better to use it rather than a function.

Takeaway: If a function is invoked repeatedly, performance may take a hit even though the size of the program is not impacted. In a similar situation, a macro would execute faster but the size of the executable would be increased.

17.4 MACROS AND STRINGS

As demonstrated previously, macro substitution doesn't occur inside strings. Consider the following macro where the parameter, `var`, occurs both inside and outside the control string of **`printf`**:

```
#define PRINT_VAR(var)  printf("var = %d\n", var)
```

When this macro is used in the following manner,

```
int i = 100;
PRINT_VAR(i);                                          Prints var = 100
```

the variable `var` is replaced with 100 in the macro body, but the first `var` embedded in the control string is left unchanged. But what if we wanted this `var` to be expanded as well? C89 addresses this problem and other string-related issues with the # and ## operators. We'll now use the # to solve this problem.

17.4.1 The # Operator

The #, known as the *stringizing* operator, is used as a prefix to a macro parameter, but only in the macro body. The preprocessor converts this token to a string. Consider the following macro definition of **`STG`**:

```
#define STG(text)  #text
```

Because of the # prefix, the preprocessor converts `text` in the macro body to a double-quoted string. This means that

```
STG(Elementary penguin)
```

is replaced with the string `"Elementary penguin"` wherever the macro **`STG`** occurs in the program. Thus, when **`printf`** is used with this macro as its sole argument,

```
printf(STG(Elementary penguin));
```

the compiler sees the following **`printf`** statement:

```
printf("Elementary penguin");
```

The stringizing feature is useful in framing macros for printing common error messages. The following macro named **`PRINT_ERROR`** uses **`fprintf`** to write to the standard error:

```
#define PRINT_ERROR(stg)  fprintf(stderr, #stg)
```

Thus, **`PRINT_ERROR(Not an integer\n);`** displays the message `Not an Integer` on the terminal. The `\n` is preserved in the **`printf`** argument in the usual manner, so the cursor moves to the next line after the string is printed.

Let's now use the # operator to revise the definition of the **`PRINT_VAR`** macro. For `var` in the control string to be replaced, simply take it out from there, apply the # operator on it and let **`printf`** concatenate the two strings. The following macro definition makes it possible:

```
#define PRINT_VAR(var)  printf(#var " = %d\n", var)
```

You can now use **PRINT_VAR** in the following manner:

```
int qty = 30;
PRINT_VAR(qty);                                    Prints qty = 30
```

The preprocessor converts #var to "qty" and **printf** concatenates this string with its adjacent one to print qty = 30.

Note: If the operand to # contains " or \, the preprocessor preserves them. Thus, **printf(STG(Hello\n));** sees "Hello\n" as its sole argument, prints Hello and moves the cursor to the next line. However, **printf(STG("Hello\n"));** actually prints "Hello\n" without printing a newline.

17.4.2 The ## Operator

The ##, also known as the *token pasting* operator, sits between two tokens and merges them to form a single one. For instance, the following sequence used in the replacement string of a macro,

```
x##y
```

converts it to xy. This feature is useful for printing values of variables that have a common prefix. Using this operator, you can improve the previous macro, **PRINT_VAR**, to print the values of variables x1, x2 and x3:

```
#define PRINT_VAR2(var, n) printf(#var#n " = %d\n", var##n)
```

The sequence var##n represents multiple variables having var as a common prefix. Furthermore, #var#n evaluates to a concatenated string after expansion of #var and #n. You can use this macro to print the values of the variables x1 and x2 in the form *variable* = *value*:

```
int x1 = 10, x2 = 20;
PRINT_VAR2(x, 1);                                  var##n becomes x1 here ...
PRINT_VAR2(x, 2);                                  ... and x2 here.
```

Because of ##, **printf** evaluates the newly created tokens, x1 and x2, as variables and prints 10 and 20 on the terminal.

Takeaway: The # transforms a token to a string, but the ## doesn't create a string. It simply creates a larger token from two existing ones.

17.4.3 tokens.c: Using the # and ## Operators

Program 17.2 uses the stringizing (#) and token pasting operators (##) with three variables. Input to these variables and their display are handled by two separate macros. Both macros merge the tokens x and 1 to form x1, x and 2 to form x2, and so on. Note the use of the \ in splitting the macro definition to occupy two physical lines. A surprising feature that is discussed in Section 17.6 is the use of "stdio.h" instead of <stdio.h>.

```
/* tokens.c: Uses macros and the # and ## operators to input and output
                  numbers using scanf and printf. */
#include "stdio.h"                /* Double quotes instead of <> ! */
#define PROMPT(stg) fprintf(stderr, #stg)
#define INPUT_NUM(var, n) scanf("%d", &var##n)
#define PRINT_VAR2(var, n)  \
        printf(#var#n " = %d\n", var##n)        /* Line split on \ */

int main(void)
{
  int x1, x2, x3;

  PROMPT(Key in three integers: );
  INPUT_NUM(x, 1);                    /* &var##n replaced with &x1 */
  INPUT_NUM(x, 2);
  INPUT_NUM(x, 3);

  PRINT_VAR2(x, 1);                   /* var##n replaced with x1 */
  PRINT_VAR2(x, 2);
  PRINT_VAR2(x, 3);

  return 0;
}
```

PROGRAM 17.2: **tokens.c**

```
Key in three integers:90 80 60
x1 = 90
x2 = 80
x3 = 60
```

PROGRAM OUTPUT: **tokens.c**

17.5 THE #undef DIRECTIVE

A defined constant or macro is undefined with the **#undef** directive which uses the name of the identifier as an argument. The following **#undef** directives,

```
#undef ROWS
#undef PRINT_VAR
```

reverse the definitions of the symbolic constant ROWS and the macro **PRINT_VAR**. These identifiers cannot be subsequently accessed by the program. While you may wonder why one would need to remove a definition, the following situations require the **#undef** directive to be invoked:

- You may need to redefine a symbolic constant to a different value. This cannot be done (GCC excepted) unless the constant is first undefined.
- You have included a file with **#include**, but you are unable to recall whether the file contains a specific named constant and, if so, what its value is. In such an event, simply redefine the constant after using **#undef**.

- C supports the facility of conditionally compiling a program, which would be difficult to implement without the undefining feature.

It is not an error to undefine a constant or macro that has already been undefined. Nor is it an error to use **#undef x** even if x has not been defined. We'll see more of **#undef** when we examine the **#ifdef**, **#ifndef** and **#if** directives in Section 17.7.

Note: Both GCC and Visual Studio allow the redefinition of a macro without first undefining it; they merely issue a warning.

17.6 THE #include DIRECTIVE

Multiple programs often use a common global variable, function or macro. Rather than key in their specifications in every program that requires them, C encourages you to store them in separate files and "include" them in a program with the **#include** directive. By convention, these *include* files have the .h extension (not mandatory though). Every program of this book uses **#include <stdio.h>**, but you have also used other include files like string.h and stdlib.h.

While **#define** enables the single-point modification of all symbolic constants and macros, **#include** extends the concept to include virtually anything that can be shared by programs. We commonly use a .h file (i.e., an include file) to store **#define** directives, function prototypes and global variables. However, function definitions are not stored in .h files but in .c (program) files, which are compiled separately and linked with the main program.

There are two ways of including a .h file in a program. The first is to use the angular brackets to enclose the filename:

```
#include <stdio.h>
```

On seeing the < and >, the preprocessor knows where the file stdio.h can be found. Every C distribution has an include directory that houses the compiler's .h files. For a UNIX/Linux system, this directory is /usr/include. For Visual Studio, a typical pathname could be C:\Program Files\Visual Studio\Include. The other .h files like string.h, ctype.h, etc. are often located in the same directory.

When developing applications, you will create include files. Make sure that you keep them at a separate location. This location is often the same directory containing the programs that use them (or somewhere close to it). In this situation, the filename must be enclosed within double quotes:

```
#include "arg_check.h"
```

The preprocessor would first look for arg_check.h in the current directory, failing which it would look in the standard include directories. If you have kept the .h files elsewhere, then use an absolute pathname that specifies the file's location relative to root:

```
#include "/home/sumit/progs/include/arg_check.h"
```

This technique may affect portability if the application is moved to a different machine. In case the new machine doesn't have the same directory structure, the **#include** directive has to be modified in all affected files.

What will you place in an include file? Practically anything that is required by multiple programs. The following sample file (say, my_include.h) shows that the contents are not restricted to preprocessor directives; declarations for structures and functions can also be included:

```
#include <stdio.h>                                  include files ...
#include <stdlib.h>

#define MAX(x, y) ((x) > (y) ? x : y)               ... macros ...
#define SIZE 10                                     ... named constants ...

short BUFSIZE = 512;                                ... variables ...

void arg_check (int, int, char *, int);             ... function prototypes ...

struct film {                                       ... structure declarations
   char title[30];
   char director[30];
   short year;
};
```

One include file can include another include file, which in turn can include another one, and this nesting can descend several levels. Furthermore, an identifier in file *h2* can be referenced in file *h1* only after *h2* is included by *h1*. Unless you are careful about sequencing these include operations, nesting and cross-referencing can create conflicts which will lead to compilation errors.

For instance, even if two .h files can define a named constant twice (say, **#define SIZE 10**) without problems, a variable or structure can't be declared twice. A function declaration may occur twice but not its definition. This delicate situation is addressed by the conditional compilation feature.

Caution: When an include file *h1* also includes *h2*, make sure that your program doesn't explicitly include them. The compiler complains if both *h1* and *h2* contain *declarations* for the same variable or structure, or *definitions* for the same function (declarations not affected).

17.7 CONDITIONAL COMPILATION

A program developed for one customer may often work for another customer but only after some changes have been made. If the changes are minor, then it may not make sense to have two separate versions of the program. Minor changes are also required to be made when the hardware and software environment undergo changes. Consider the following situations:

- A program is ported to a different hardware architecture having different sizes for the word and primary data types.
- A program may have to run under an operating system that treats filenames differently (say, one that uses \ instead of / as the pathname delimiter).

- During the testing period, it may be necessary to induct diagnostic print statements for debugging purposes. These statements must eventually be made inactive after the program code has been finalized.

Without adequate knowledge of the preprocessor, you probably will handle these situations by commenting and uncommenting code. When the modified program fails at runtime, you'll also debug the code by printing the value of a variable at key program locations using **printf**. As the changes become numerous, the entire exercise becomes unmanageable and eventually turns into a nightmare. The solution to this problem lies in the *conditional compilation* feature supported by the preprocessor.

The objective of conditional compilation is to develop a *single* program where certain sections are conditionally compiled. Using keywords like **#ifdef**, **#endif**, **#else**, etc., the preprocessor can test whether a macro or symbolic constant is defined or not. The outcome of the test can then be used to determine whether a specific code fragment will be included in the compilation process. These keywords can be grouped into three categories:

- The **#ifdef** keyword that tests whether a macro is defined or not.
- The **#ifndef** keyword that negates **#ifdef**, i.e. it tests whether a macro is *not* defined.
- The **#if** keyword which evaluates a constant expression for truth or falsehood.

These directives resemble the **if** statement of C except that they don't use curly braces to define the associated block. All three constructs terminate with **#endif** which acts as the closing brace. However, they all optionally use the **#else** keyword having its usual meaning.

17.7.1 The #ifdef and #ifndef Directives

Unlike the **if** statement of C, the **#ifdef** and **#ifndef** directives do not evaluate a control expression. Instead, they simply check whether a named constant or macro is defined or not. The following code determines whether SIZE is defined:

```
#ifdef SIZE
   printf("SIZE = %d\n", SIZE);
#endif
```

If SIZE is defined, all directives and statements between **#ifdef** and **#endif** are executed. Because the action here specifies a C statement, this code will work only in the body of a function (including **main**).

The **#ifndef** directive simply reverses the significance of **#ifdef**, i.e., it determines whether a named constant or macro is *not* defined. You can't use either **#ifdef** or **#ifndef** with a relational expression to determine whether SIZE has a specific value. That can be done with **#if**, which we'll examine soon.

Both keywords optionally use the **#else** keyword having its usual meaning. The following code snippet defines SIZE if it is not defined and undefines and redefines it otherwise:

```
#ifndef SIZE                          All 6 lines are preprocessor directives ...
   #define SIZE 20                    ... Can be placed before main
#else
   #undef SIZE                        Needed if SIZE is defined ...
```

```
   #define SIZE 20                          ... but its value has to be changed
#endif
```

This situation can arise when including a `.h` file (say, `foo.h`) without knowing whether `SIZE` has been defined there. The **`#ifndef`** section ensures `SIZE` is set to 20, but do we really need the **`#else`** segment? Yes, we do if we need to reset `SIZE` to a different value *(17.2)*. We may not remember the existing value of `SIZE` (if defined at all), so the **`#undef`**-**`#define`** sequence ensures that `SIZE` is properly set.

The **`#else`** option is needed when designing code for two operating systems. The following form lets you select the code to be executed depending on whether `UNIX` is defined or not:

```
#ifdef UNIX
  ... code ...
#else
  ... code ...
#endif
```

How do you extend this logic to work with more than two operating systems? You simply can't except by using multiple **`#ifdef`**-**`#else`**-**`#endif`** constructs. A better method would be to use the **`#if`**-**`#elif`**-**`#else`**-**`#endif`** feature introduced by C89.

Note: **`#ifdef`** and **`#ifndef`** work only with named constants and macros. They can't be used to check whether a *variable* is defined or not. The **`if`** statement in C can't perform this check either.

17.7.2 The `#if` and `#elif` Directives

The **`#if`**-**`#elif`** duo behaves much like the **`if`**-**`elif`** statement of C in two ways. First, the construct permits multi-way branching, unlike **`#ifdef`** which restricts the number of options to two. Second, **`#if`** evaluates a constant control expression using the following syntax:

```
#if expression1 is true
   directives1 or statements1
#elif expression2 is true
   directives2 or statements2
  ...
#else
   directives3 or statements3
#endif
```

Use of parentheses to enclose the expression is optional. Like the **`if`** statement, **`#if`** also uses the relational and logical operators. This is how you use the preprocessor to test a symbolic constant for different values:

```
#if OS == 1
   #define UNIX
#elif (OS == 2)                             Parentheses OK
   #define WINDOWS
#else
   #define MAC_OS_X
#endif
```

This task can't be performed by the **if** statement because OS would be expanded before it can be tested. **#if** can also be used with the logical operators (|| and &&) to test complex conditions. Furthermore, using the **defined** operator, **#if** can also check whether a symbolic constant or macro is defined or not:

```
#if (OS == 1 || OS == 2)
#if defined UNIX
#if !defined UNIX                                   Negates previous test
```

The two preceding examples replicate the behavior of the **#ifdef** and **#ifndef** directives. Even though **#if** and **if** have a lot in common, some features of **if** are not available in **#if**. For instance, **sizeof** and casts can't be used with **#if**.

17.8 USING #ifdef FOR DEBUGGING PROGRAMS

If a separate debugger program is not used, the standard technique of debugging programs is to display variables at key program locations. These print statements help determine whether a code section is working properly. After the debugging exercise is over, these lines are either deactivated by commenting them or removed altogether. A better solution would be to use the conditional compilation feature that makes debugging convenient.

We can use the **#ifdef** feature to print a diagnostic statement depending on whether a named constant is defined or not:

```
#ifdef DEBUG
   fprintf(stderr, "Value of x = %hd\n", x);
#endif
```

In the testing phase, we can insert either of the following statements at the beginning of the preprocessor section to enable execution of DEBUG-dependent statements:

```
#define DEBUG 1
#define DEBUG                                   DEBUG is considered defined
```

Once the code has been finalized, you need not remove this directive; simply add the following directive after the previous one:

```
#undef DEBUG
```

Can't this task be achieved using a variable and checking its value with the **if** statement? Sure it can, but the solution provided by the preprocessor is more efficient because DEBUG-related code will be ignored by the compiler if DEBUG is not defined. The size of the executable is thus trimmed, which won't happen when a variable is used instead.

Most C compilers also provide a separate option to set a named constant at the time of compilation. When using the GCC compiler, simply define DEBUG in the command line using the -D option:

```
gcc -D DEBUG foo.c
```

With Visual Studio, use the /D switch:

```
cl /DDEBUG foo.c
```

The preprocessor, which now considers DEBUG as defined, activates all directives and statements that require DEBUG to be set. *When using this technique, you need not modify the program at all.* Also, if program execution depends on the value of DEBUG, you can set a value in the command line as well:

```
gcc -D DEBUG=2 foo.c
cl /DDEBUG=2 foo.c
```

In this manner, you can control the compilation environment of a program without modifying it. Debugging is a critical phase of program development, and the tools supported by the preprocessor for this activity make life easier for a programmer. If this debugging feature doesn't help in some situations, then you have to use the debugger program of the compiling software (not discussed in this edition).

Takeaway: Without modifying a program, conditional compilation can be implemented by using the -D option or /D switch of the compiler.

17.9 THE BITWISE OPERATORS

C differs from other high-level languages in its support of some low-level features. You have used pointers to access and manipulate every byte of memory. In this section, we'll examine the C *bitwise* operators which work with every bit of every byte of memory.

Bitwise operations can speed up some operations that are otherwise carried out using arithmetic operators. Because of these operators, it is possible to use one byte as eight variables. Older UNIX systems handle 32 different signals using bitwise operators on a *single* int variable.

C supports a set of six operators for handling individual bit operations on an integer. Five of them are binary operators while one is unary. They can be grouped into the fallowing categories:

- The bitwise logical operators (&, | and ^)
- The bitwise one's complement operator (~)
- The bitwise shift operators (<< and >>)

The bitwise logical operators, & and |, represent AND and OR operations performed on an *integer*. Functionally, they are similar to the logical operators, && and ||, which evaluate the truth value of an *expression*. The ^ represents the exclusive-OR (XOR) operation and the ~ is akin to the ! (NOT) operator which inverts the logical value of an expression. Table 17.1 depicts the truth table of the binary operators that were discussed in Chapter 3.

As previously mentioned, all bitwise operators work on one or more integer operands, including a char, which is essentially an integer. They don't apply to floating point numbers. ANSI C specifies their behavior when used with unsigned integers only. For signed integers, their behavior is implementation-dependent and won't be examined here. Bits are numbered from right to left (Fig. 17.1), beginning with zero for the LSB (least significant bit).

Bitwise operators have a low precedence compared to the arithmetic and logical operators (Appendix A). For this reason, sub-expressions using bitwise operations (like i & j) are frequently

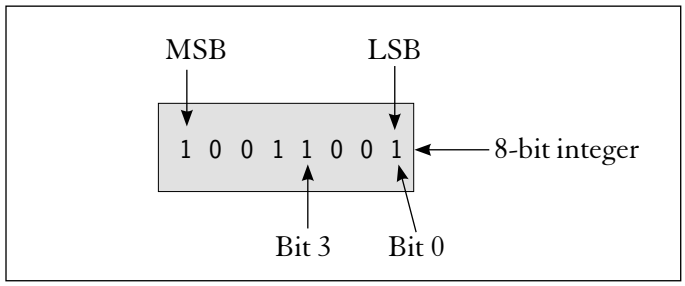


FIGURE 17.1 Bit Numbering in an Integer

enclosed by parentheses. Except for the shorthand assignment operators (like &=, >>=, etc.), all bitwise operators have left-right associativity.

TABLE 17.1 Truth Table for the AND, OR and ex-OR Operators

A	*B*	*A & B*	*A | B*	*A ^ B*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

17.9.1 The & Operator: Bitwise AND

The & operator performs the AND operation on two integer operands. Like the other binary bitwise operators, this operator uses a simple syntax:

operand1 & *operand2*

Each bit of *operand1* is AND'ed with its corresponding bit of *operand2*. This operation produces the result 1 if both bits are 1, and 0 if either bit is 0. This logic is demonstrated by the following example which shows the bit pattern of the integers 178 and 218 along with the result formed by AND'ing them:

```
1 0 1 1 0 0 1 0 (A)        Operand 1—Decimal number 178
1 1 0 1 1 0 1 0 (B)        Operand 2—Decimal number 218
1 0 0 1 0 0 1 0 (A & B)    Result—Decimal number 146
```

The expression `178 & 218` thus evaluates to `146`. For easy readability, we have ignored the high-order bytes which are obviously padded with zeroes. The result can be assigned to a variable:

```
int i1 = 178, i2 = 218;
int i3 = i1 & i2;
```

Every bitwise operator (except the unary ~) can be combined with the = to form its corresponding assignment operator. Thus, we can reassign `i1` using the `&=` operator:

```
i1 &= i2;                    Same as i1 = i1 & i2;
```

It's not easy to see what `178 & 218` evaluates to without comparing their bit patterns, but when the base of the operands is a power of 2, the result can often be seen by visual inspection. For instance, the expression `63 & 4095` can be specified in octal in the following manner:

```
077 & 07777                                   Bits 0 to 5 are 1 in both
```

All octal integers are known to C by their `0` prefix. The result is `63` in decimal or `77` in octal, which means that bits 0 to 5 are set (to 1) in both integers. You can also use hexadecimal numbers using the `0x` prefix.

17.9.2 Using a Mask with &

The second operand of the bitwise logical operators is often used as a *mask*. This is simply an integer containing a bit pattern that is used to manipulate the bit pattern of the first operand (the *target integer*). The result of this operation depends on the value of the mask and the operator used. A mask is commonly used to perform the following operations:

- Set one or more bits (to 1) in the target integer.
- Clear one or more bits (to 0) in the target integer.
- Test whether the target integer has a specific bit set or cleared.

Clearing Bits When a mask is used with the `&`, for all bits in the mask that are assigned 0, their corresponding bits in the target integer are cleared. The remaining bits retain their status. For instance, when the target integer `140` (`10001100`) is masked with `247` (`11110111`) using the `&` operator,

```
140 & 247                                     AND'ing 10001100 with 11110111
```

the expression evaluates to `132` (`10000100`), which differs from the original number in having a masked out (cleared) bit 3. Program 17.3 uses these integers to demonstrate this operation.

The preceding expression can be assigned to the same or a different variable. The following sequence changes the target itself:

```
short i = 140, mask = 247;
i = i & mask;                                 i changes from 140 to 132
```

This masking operation can also be represented as `i &= mask;` or printed simultaneously as **`printf("%hd\n", i &= mask);`**.

Testing for Set Bits We can use a mask having the specified bits set (to 1) to determine whether the same locations in the target are set as well. This is done by comparing the bitwise expression (*target* `&` *mask*) with the mask itself. The following snippet checks whether bits 0, 1 and 2 in the target are set:

```
short i = 15, mask = 7;
if ((i & mask) == mask)
   fputs("Bits 0 to 2 are set\n", stderr);
```

The decimal number `7` translates to `111` in binary, so `i & 7` evaluates to `7` if the same bits are set in the target `i`.

Takeaway: The expression `i = i & mask`, where `mask` has specific bits set to 0, *turns off* the bits in `i` (the target) at the same bit locations as `mask`. When `mask` has specific bits set to 1, the same expression can be compared to `mask` to *determine* whether the bits at the same locations in `i` are set.

17.9.3 The | Operator: Bitwise OR

The | operator performs the OR operation on two operands using the following syntax:

operand1 | *operand2*

Each bit of *operand1* is OR'd with its corresponding bit of *operand2*. This operation produces a result of 1 only if either of the bits is 1. For demonstrating this operation, consider the same numbers, 178 and 218, that were used with the & operator in Section *17.9.1*:

10110010 (A)	Operand 1—Decimal number 178
11011010 (B)	Operand 2—Decimal number 218
11111010 (A | B)	Result—Decimal number 250

The expression `178 | 218` evaluates to `250`. The | can be used with a bit mask to set specified bits of a number to 1 while keeping the remaining bits unchanged. For instance, the expression

```
192 | 8
```

uses the mask 8 (00001000) where bit 3 is set. The target integer, 192 (11000000) has bits 6 and 7 set, so OR'ing 192 and 8 sets bit 3 in 192 as well. The result is thus 200 (11001000). This operation is demonstrated in Program 17.3 which uses the same numbers.

17.9.4 The ^ Operator: Bitwise Exclusive OR (XOR)

The XOR operator is represented by the ^ symbol. It differs from the OR operator in only one respect: the result is 0 when both operands are 1. Thus, the result is 1 only when one of the operands (but not both) is 1. The XOR operation is demonstrated in the following example where operand A is encrypted by operand B (used as a key) and decrypted to get back A using the same key:

10110010 (A)	Operand 1—Decimal number 178
11011010 (B)	Operand 2 (key)—Decimal number 218
01101000 (A ^ B)	Result of XOR (encrypted)—Decimal number 104
11011010 (B)	Same operand 2
10110010 (A)	Result of XOR (decrypted) —Original number 178

The preceding example demonstrates the basic principles of *symmetric cryptography*. In this system, a number A is transformed (encrypted) to another number using number B as a key, and decrypted with the same key to get back the original number A. (Asymmetric cryptography uses different keys for encryption and decryption.)

As demonstrated later in Program 17.3, the XOR operator can be used to swap two numbers without using a temporary variable. Also, while the expression *target* `&` *mask* can determine whether specific bits in *target* are set, *target* `^` *mask* can check for cleared bits.

17.9.5 The ~ Operator: One's Complement (NOT)

The `~`, the only unary bitwise operator, known as the NOT or one's complement operator, follows the truth table of the NOT gate *(3.9.2)*. This operator flips 0 to 1 and 1 to 0, as shown in the following example that involves a `short` integer:

```
00000000 10111110 (A)          Decimal number 190
11111111 01000001 (~A)         Decimal number 65345
```

Unlike with the other bitwise operators, the result of the NOT operation depends on the size of the integer. Use the smallest integer type (the `char`) with the same integer, and the result becomes obvious:

```
char c = 190;
int i = ~c;
printf("%d, ASCII '%c'\n", i, i);          Prints 65, ASCII 'A'
```

The inverted value is now `65` (the ASCII value of `'A'`), and not `65345`, the value obtained when `190` was treated as a `short` integer.

17.9.6 The << Operator: Bitwise Left-Shift

Apart from changing bits selectively using logical operations, C also supports two operators (`<<` and `>>`) for shifting the entire set of bits of an integer in both directions. The `<<` shifts bits to the left using the following syntax:

operand `<<` *num*

The `<<` operates from the LSB (the low-order bit on the right) and pushes the bits of *operand* to the left as many times as specified by *num*. Every vacated bit on the right is padded with a zero, while the bit emerging from the MSB end is lost. For instance, the expression `3 << 1` moves the bits in the integer `3` by one position to the left, as shown in the following:

```
00000011                       Decimal 3
00000110                       Decimal 6; result of 3 << 1
```

The new bit pattern formed after shifting the bits of `3` evaluates to 6. Now, let's move the bits by another 2 positions:

```
00000110                       Decimal 6
00011000                       Decimal 24; result of 3 << 2
```

It's easy to conclude that every shift of the bit pattern by one position to the left has the effect of multiplying the number by 2. This is actually quite obvious because a similar operation performed on the decimal number 3 multiples the number by 10 to produce 30. Thus, instead of using the function **`pow(2, 10)`** to compute the 10th power of 2 (i.e., 1024), you can use `2 << 9` to obtain 1024.

Like with the other bitwise operators, you can assign the result of the left-shift operation to a variable in the usual ways:

```
mask = mask << 1;
mask <<= 1;                                   Same as mask = mask << 1;
```

This expression, when used in tandem with the expression *target* & *mask*, in a loop can perform an important operation. It can determine the specific bits that are set in the target. This has been set as a programming exercise at the end of this chapter.

17.9.7 The >> Operator: Bitwise Right-Shift

The >> right-shift operator performs the same function as <<, but from the MSB end, i.e., left end of the number. The syntax is also similar:

operand >> *num*

Every bit in *operand* is shifted right by as many positions as specified by *num*. Thus, `129 >> 1` moves the bits in `129` by one position to the right. This operation throws out the bit at the LSB end and pads the MSB end with a zero as shown in the following:

```
1000001                                       Decimal 129
0100000                                       Decimal 64; result of 129 >> 1
```

While left-shifting multiplies the number by 2, right-shifting divides the number by 2. Note that this integral division truncates the fractional component. You can assign the result of the operation to a variable:

```
y = x >> 4;
y >>= 4;                                      Same as y = y >> 4
```

A caveat applies when using the >> operator with signed numbers where the sign bit itself is shifted right. The ANSI standard is silent on the way the left end is padded, so it's not safe to assume that zero is used for padding. The result is actually implementation-dependent.

The >> operator is useful for extracting bits from the target number by shifting bits in the mask. Program 17.3 uses this feature to convert a decimal number to binary.

Takeaway: In every shift of one bit position, the << operator multiplies the number by 2 while >> divides the number by 2. Integer division truncates the fractional part of the number.

Note: In general, multiplying and dividing a number using the bitwise shift operators is faster than those performed with their corresponding arithmetic operators.

17.10 `bitwise_operations.c`: USING THE BITWISE OPERATORS

Program 17.3 summarizes the attributes of all the bitwise operators except one (the ~). The first two sections of this four-part program use the AND and OR operators with a mask to clear and set a bit in a `short` integer. The third section uses XOR to swap two numbers without using a temporary variable. The fourth section uses the shift operators to multiply and divide a number by a multiple of 2. For ease of understanding, numbers have often been displayed in binary using the user-defined **`dec2bin`** function.

```
#include <stdio.h>
char *dec2bin(unsigned short num);
int main(void)
{
  unsigned short i, j, mask;
  i = 140; mask = 247;
  fputs("Clearing Bit 3 ...\n", stderr);
  printf("   i = %s (Decimal: %hd)\n", dec2bin(i), i);
  printf("mask = %s (Decimal: %hd)\n", dec2bin(mask), mask);
  j = i & mask;
  printf("   j = %s (Decimal: %hd)\n\n", dec2bin(j), j);
  i = 192; mask = 8;
  fputs("Setting Bit 3 ...\n", stderr);
  printf("   i = %s (Decimal: %hd)\n", dec2bin(i), i);
  printf("mask = %s (Decimal: %hd)\n", dec2bin(mask), mask);
  j = i | mask;
  printf("   j = %s (Decimal: %hd)\n\n", dec2bin(j), j);
  j = 50;
  printf("Before swapping, i = %hd, j = %hd\n", i, j);
  i ^= j;
  j ^= i;
  i ^= j;
  printf("After swapping,  i = %hd, j = %hd\n\n", i, j);
  i = 25;
  printf("   i = %s (Decimal: %hd)\n", dec2bin(i), i);
  j = i << 2;
  fputs("Shifting left using i << 2 ...\n", stderr);
  printf("   j = %s (Decimal: %hd)\n", dec2bin(j), j);
  fputs("Shifting right using j >> 2 ...\n", stderr);
  printf("j >> 2: %s (Decimal: %hd)\n", dec2bin(j >> 2), j >> 2);
  return 0;
}
char *dec2bin(unsigned short num)
{
    unsigned short mask = 32768;  /* MSB = 1; other 15 bits = 0 */
    short i = 0;
    static char stg[17];          /* One slot reserved for NULL */
    while (mask > 0) {
       stg[i++] = (num & mask) == 0 ? '0' : '1';
       if (i == 8)
          stg[i++] = ' ';         /* A space after every 8 bits */
       mask = mask >> 1 ;         /* Shifts the 1 in MSB to the right */
    }
    stg[i] = '\0';
    return stg;
}
```

PROGRAM 17.3: **bitwise_operations.c**

```
Clearing Bit 3 ...
   i = 00000000 10001100 (Decimal: 140)
mask = 00000000 11110111 (Decimal: 247)
   j = 00000000 10000100 (Decimal: 132)

Setting Bit 3 ...
   i = 00000000 11000000 (Decimal: 192)
mask = 00000000 00001000 (Decimal: 8)
   j = 00000000 11001000 (Decimal: 200)

Before swapping, i = 192, j = 50
After swapping,  i = 50, j = 192

   i = 00000000 00011001 (Decimal: 25)
Shifting left using i << 2 ...
   j = 00000000 01100100 (Decimal: 100)
Shifting right using j >> 2 ...
j >> 2: 00000000 00011001 (Decimal: 25)
```

PROGRAM OUTPUT: **bitwise_operations.c**

The first two sections clear and set bit 3 (fourth bit from right) in the integer 140. Note that the selected mask (247) used for AND'ing with 140 has 0 in this position (11110111). The other mask (8) has bit 3 set (00001000). The third section swaps two integers using three XOR operations that reassign i and j. The final section shifts the bits twice, once on either side, which has the effect of multiplying and dividing the original number by 4.

The **dec2bin** function is used several times in the program. The function returns a string (i.e., its pointer) representing the binary equivalent of the decimal number passed as an argument. Each of the 16 loop iterations shifts the set bit in mask to the right, thus progressively reducing its value from 32768 (bit 15 set) to 1. In the process, the num & mask operation extracts every bit from num and this bit is saved in the array stg as a character ('0' and '1' instead of 0 and 1). You'll find this simple function quite useful when working with the bitwise operators.

17.11 POINTERS TO FUNCTIONS

A *pointer to a function* is a powerful and useful tool that enables a function *f2* or *f3* to be passed as an *argument* to a function *f1*. Because the decision to select *f2* or *f3* can be made at runtime, the entire process of declaring, assigning and invoking functions through their pointers can be somewhat tricky. However, it need not be so as you'll soon find out.

Just as a variable has an address in memory, so does a function. You can set a pointer to point to the base address of the function code and its variables in memory. The variable representing a pointer to a function has to be declared, a function assigned to it, and then dereferenced to execute the pointed-to function.

The following statement declares a function pointer named fp1 which is meant to point to a function that takes no arguments and returns nothing:

```
void (*fp1)();                          fp1 doesn't point to any function yet.
```

Here, we have a pointer variable `fp1` which *can* point to a function that takes no arguments (the `()` on the right) and returns nothing (`void`). The parentheses enclosing `*fp1` are needed simply to override the default precedence that `()` has over the asterisk. Had we used `void *fp1();` instead, we would have defined a function `fp1` that returns a `void` pointer.

The pointer `fp1` is now ready to point to a function, but only one that takes no arguments and returns `void`. If a function named **`print_something`** is defined in the following manner,

```
void print_something(void)
{
   printf("Hello World\n");
   return;
}
```

then we can assign the name of this function to `fp1` in either of the following ways:

```
fp1 = print_something;
fp1 = &print_something;                 & is optional
```

Like an array, the name of a function signifies a pointer to its base address. Because `fp1` is a pointer variable, it can be unhooked from **`print_something`** and made to point to another function having an identical signature.

We can now execute **`print_something`** using its pointer. Here again, we have two options. Since `fp1` contains the address of a function, we can dereference this address and "evaluate" the contents at the address. In this case, evaluation means execution of the function itself:

```
(*fp1)();                               Executes print_something function
```

But C is a smart language. It knows that you are trying to execute a function, so it also allows you to execute `fp1` directly without its accompanying baggage:

```
fp1();                                  Also executes print_something function
```

Before we explore the mechanism of passing `fp1` as an argument to another function, let's consider another type of function pointer. This is taken up next.

17.11.1 Function Pointer for Functions Using Arguments

Most functions use arguments and return a value, and we must know how to use a pointer to a function of this type. The following statement declares a function pointer `fp2` where the pointed-to function accepts two `float` arguments and returns a `float`:

```
float (*fp2)(float, float);
```

We can add variable names to the two arguments but the compiler would simply ignore them. Providing names don't make much sense here because `fp2` can point to multiple functions where the arguments will have different names. We now need to define a function, say, **`add`**, that matches the signature of `fp2`:

```
float add(float x, float y)
{
   return x + y;
}
```

Because the **add** function takes two float arguments and returns a float, the name of this function can now be assigned to fp2 in this manner:

```
fp2 = add;
```

Note that even though **add** uses two arguments, we don't specify them in this assignment. These arguments are provided when the pointer is dereferenced. Whether you would like to use fp2 or (*fp2) for executing the assigned function is purely a matter of choice:

```
float result = (*fp2)(4.5, 5.5);          result is 10.0
float result = fp2(4.5, 5.5);             Ditto
```

Because fp2 has the return type float (same as **add**), the value returned by **add** has to be saved in a separate variable. For the same reason, fp2 can also be used as an expression, and this will be done in the next program.

The discussions involving fp1 and fp2 are merely conceptual. It's unlikely that you'll actually invoke functions in the ways just discussed. We have to learn to use function pointers as function arguments, but we'll do that only after confirming what we have learned so far.

17.11.2 func_pointer.c: Using Function Pointers

The attributes of function pointers are demonstrated in Program 17.4, which makes two of them—fp1 and fp2—point to the functions **print_something** and **add**, respectively. The necessary explanations are adequately provided in the annotations.

```
/* func_pointer.c: Uses 2 function pointers to execute functions
                   with and without arguments and return value. */
#include <stdio.h>

void print_something(void);
float add(float x, float y);

int main(void)
{
   float result;

   /* Pointer for function using no arguments and returning nothing */
   void (*fp1)();             /* Declares function pointer fp1 */
   fp1 = print_something;     /* Assigns function to function pointer */
   (*fp1)();                  /* Invokes dereferenced function pointer */
   fp1();                     /* Invokes function pointer directly */
```

```
    /* Pointer for function using two arguments and returning a value */
    float (*fp2)(float, float);        /* Declares function pointer fp2 */
    fp2 = add;                         /* Assigns function to function pointer */
    result = (*fp2)(4.5, 5.5);
    printf("Result = %.2f\n", result);
    printf("Result = %.2f\n", fp2(2.5, 3.5));
    return 0;
}

void print_something(void)
{
    printf("Can execute function pointer with or without dereferencing.\n");
    return;
}
float add(float x, float y)
{
    return x + y;
}
```

PROGRAM 17.4: **func_pointer.c**

```
Can execute function pointer with or without dereferencing.
Can execute function pointer with or without dereferencing.
Result = 10.00
Result = 6.00
```

PROGRAM OUTPUT: **func_pointer.c**

17.11.3 Callback Functions

Function pointers are most useful when used as *callback functions*. A callback function is one that is passed as an argument to another function. There will generally be multiple callback functions to choose from, and the function that is actually called can be determined at runtime. In this section, we'll invoke the same functions, **print_something** and **add**, by passing them as arguments to other functions.

How does one pass the **print_something** function as an argument to another function named, say, **print**? Let's first define **print** to accept a function having the same signature as **print_something**:

```
void print(void (*fp)(void))
{
    fp();                                  Can also use (*fp)();
}
```

The argument to **print**, shown as `void (*fp)(void)`, tells **print** that its only parameter is a function that takes no arguments and returns `void`. This parameter matches the prototype of the **print_something** function. This means that we can pass a pointer to **print_something** as an argument to **print**. Like an array, the name of a function represents a pointer, so we can use **print** like this:

```
void (*fp1)();                          Declaration of fp1
fp1 = print_something;
print(fp1);                             Executes print_something
```

We have been able to pass the simplest function pointer as an argument to another function. How does one pass the **add** function with its two arguments to another function, say, **compute**? It is here that function pointers can get a little confusing, but it need not be so if you have a re-look at the declaration of the **print** function. The following declaration for the **compute** function uses three arguments, one for the **add** function itself and two for its arguments:

```
float compute(float (*fp)(float, float), float x, float y);
```

The declaration tells the compiler that the first parameter is a pointer to a function that accepts two float arguments and returns a float. This matches the prototype of the **add** function. The remaining two parameters represent the actual arguments of the function whose pointer is passed as the first argument. We can now call either **add** or fp2 from the **compute** function:

```
float (*fp2)(float, float);
fp2 = add;
printf("Result = %.2f\n", compute(fp2, 100, 200));
```

This prints the sum of 100 and 200, i.e. 300.00. Observe that even though we could run fp2 directly in the previous program as

```
result = (*fp2)(4.5, 5.5);
```

we can't pass this function call as a *single* argument to **compute**. The arguments used with fp2 have to be passed separately. If you appreciate and remember this feature, then you'll have no problem in handling callback functions. The observations we have just made are demonstrated in Program 17.5.

Takeaway: If function *f2* having n arguments is passed as an argument to function *f1*, then *f1* must be declared with at least $n + 1$ arguments. The first argument of *f1* represents *f2*, and the remaining ones represent the arguments of f2. The words "at least" are used because *f1* may have other arguments that are not related to *f2*.

```
/* callback_func.c: Uses callback functions with or without
                    arguments and return value. */
#include <stdio.h>

void  (*fp1)();                     /* Declaration of function pointer fp1 */
void print_something();             /* Callback function */
void print(void (*fp)());           /* Meant to call a callback function */

float (*fp2)(float, float);         /* Declaration of function pointer fp2 */
float add(float x, float y);        /* Callback function */

/* Following function is meant to call a callback function. */
float compute(float (*fp)(float, float), float x, float y);
```

```
int main(void)
{
   fp1 = print_something;
   print(fp1);

   fp2 = add;
   printf("Calling add from fp2 argument in compute, result = %.2f\n",
           compute(fp2, 100, 200));
   printf("Calling add from add argument in compute, "
          "result = %.2f\n", compute(add, 3500, 1000));

   return 0;
}
void print_something(void)
{
   printf("Used here as a callback function.\n");
   return;
}
void print(void (*fp)(void))
{
   printf("Calling a function passed as argument.\n");
   fp();
   return;
}
float add(float x, float y)
{
   return x + y;
}
float compute(float (*fp)(float, float), float x, float y)
{
   return fp(x, y);
}
```

PROGRAM 17.5: **callback_func.c**

```
Calling a function passed as argument.
Used here as a callback function.
Calling add from fp2 argument in compute, result = 300.00
Calling add from add argument in compute, result = 4500.00
```

PROGRAM OUTPUT: **callback_func.c**

17.12 FUNCTIONS WITH VARIABLE ARGUMENTS

Ever wondered how **printf** and **scanf** work with a variable number of arguments? C obviously supports functions of this type, so you can devise one, say, to sum a variable number of integers. However, C doesn't have the means to determine the number of arguments passed to a function. So, it's the programmer's job to convey that number to the function. This task can be achieved in the following ways:

- The first argument of the function could represent the number of remaining arguments.
- The first argument could provide some *indication* about the number of arguments to follow. For instance, **printf** counts the number of format specifiers in the control string to determine the number of remaining arguments.
- The last argument could be 0 or NULL to signal the end of the argument list. This technique is used by the "exec" family of functions available in UNIX-C but not in ANSI C.

In this section, we'll examine the first technique. For this purpose, we need to use one special data type and three macros defined in `stdarg.h`. The prototype of a function of this type needs the ellipsis (three dots represented as `...`) as the last argument. Here's one that returns an `int`:

```
int func(int, ...);
```

The return type of the function **func** need not compulsorily be an `int`; its type is determined by the requirements of the program. However, the second `int` is mandatory since it represents the number of arguments that follow the first argument. For defining this function, we need to do the following:

- Define a variable, say, `arglist`, of type `va_list`.
- Invoke the macro **va_start** to initialize the list (`arglist`). The arguments represented by the ellipsis (`...`) are now held in `arglist`.
- Invoke the **va_arg** macro to fetch each argument of the list. This exercise is often carried out in a loop.
- Invoke the **va_end** macro to clean up the list.

Let's now understand the functioning of the three macros, defined in `stdarg.h`, by examining their prototypes. The variable names, `arglist` and `argc`, are actually used in the program that follows:

```
void va_start(va_list arglist, argc);
type va_arg(va_list arglist, type);
void va_end(va_list arglist);
```

The **va_start** macro initializes `arglist` of type `va_list`. After initialization, `arglist` contains the argument list. Since **va_start** doesn't know the size of this list, it's necessary to pass this number, `argc`, as the second argument.

The elements of the list are now fetched by the **va_arg** macro. Note that the second argument of **va_arg** is the type of the item that is fetched from the list. It may seem strange but the return type of **va_arg** is determined by the type of its second argument!

After all elements of the list are fetched, the list must be cleaned with **va_end**. The list cannot be reused unless **va_start** is invoked again to initialize the list.

Program 17.6 uses a function named **sum** which uses a variable number of integer arguments to sum all but one of them. The first argument of **sum** (`argc`) represents the number of remaining arguments. The program is clearly documented, but note that the type of the arguments (here, `int`) is itself specified as a separate argument! **va_arg** can't be function; it's got to be a macro.

```
/* variable_arguments.c: Uses a variable of type va_list and 3 macros. */

#include <stdio.h>
#include <stdarg.h>       /* Defines 3 macros -- va_start, va_args, va_end */
int sum(int argc, ...);      /* ... indicates variable number of arguments */
int main(void) {
   printf("10 + 20 + 40 = %d\n", sum(3, 10, 20, 40));
   printf("1 + 2 + 3 + 5 + 10 = %d\n", sum(5, 1, 2, 3, 5, 10));

   return 0;
}
int sum(int argc, ...)           /* Function definition */
{
   int i, total = 0;

   /* First define a variable arglist of type va_list */
   va_list arglist;
   /* Initialize arglist */
   va_start(arglist, argc);
   /* Retrieve all arguments from arglist */
   for (i = 0; i < argc; i++)
      total += va_arg(arglist, int);

   /* Clean up arglist */
   va_end(arglist);
   return total;
}
```

PROGRAM 17.6: **variable_arguments.c**

```
10 + 20 + 40 = 70
1 + 2 + 3 + 5 + 10 = 21
```

PROGRAM OUTPUT: **variable_arguments.c**

17.13 MULTI-SOURCE PROGRAM FILES

The most trivial application uses a single source file. A typical commercial application is spread across multiple `.c` (source), `.h` (include) and `.o` (object) files, along with one or more executables. The standard technique of using **`gcc foo.c`** or **`cl foo.c`** doesn't work in this situation. When code is distributed across multiple files, all `.c` files must be included for compilation. This is how GCC and Visual Studio compile multiple-file programs:

```
gcc a.c b.c c.c          Creates a.out; no .o files
cl a.c b.c c.c           Creates a.exe and three .obj files
```

On a Linux system, **gcc** calls the assembler *(4.2.3)* to create the .o files before it invokes the linker to create a single executable. Using the -c option, you can create only the object files and without creating **a.out**:

```
gcc -c a.c b.c c.c                                    Creates only a.o, b.o and c.o
```

For single-source programs (where functions are stored in the same file as the main program itself), we normally don't need the intermediate .o file. However, in a real-life scenario, we are often compelled to do a *partial* compilation and maintain these .o files. To understand why, let's examine a multi-source application.

17.13.1 A Multi-Source Application

Our sample application computes the interest on a recurring deposit. The main program, **rec_deposit.c** (Program 17.7a), accepts three arguments representing the principal, interest and term. The maturity amount is written to the standard output. The main program invokes three functions:

- **arg_check** This function checks whether the right number of arguments has been keyed in. It is defined in **arg_check.c** which includes arg_check.h (Program 17.7b).
- **quit** This function prints a message and terminates the program. It is defined in **quit.c** and includes quit.h (Program 17.7c).
- **compute** It computes the interest from the three arguments provided. This function is not placed in a separate file.

arg_check and **quit** are useful reusable functions. **compute** is maintained in the main file only to understand why this too should be moved to its own source file. The two header files contain—apart from the usual include statements— the prototypes of their respective functions. These functions are used in **main**, so their header files must be included in **rec_deposit.c** also.

```
#include <math.h>
#include "quit.h"
#include "arg_check.h"
float compute(float, float, float);
int main(int argc, char **argv) {
   float principal, interest, term, sum ;
   char *mesg = "Three arguments required\n" ;
   char *mesg2 = "All arguments must be positive\n" ;
   arg_check(4, argc, mesg, 1);          /* Checks for three arguments */
   sscanf(argv[1], "%f", &principal);    /* Converting strings to float */
   sscanf(argv[2], "%f", &interest);
   sscanf(argv[3], "%f", &term);

   if (principal <= 0 || interest <= 0 || term <= 0)
      quit(mesg2, 2);                  /* Quits with 2 as $? on error */
```

```
   sum = compute(principal, interest, term);
   printf("Maturity amount: %f\n", sum );
   exit(0);
}

float compute(float principal, float interest, float term) {
   int i;
   float maturity = 0;
   interest = 1 + interest / 100 ;
   for (i = 1 ; i <= term ; i++)
      maturity += principal * pow(interest, i) ;
   return maturity;
}
```

PROGRAM 17.7a: **rec_deposit.c**

```
#include "arg_check.h"
void arg_check (int args, int argc, char *message, int exit_status) {
   if (argc != args) {
       fprintf(stderr, message);
       exit(exit_status);
   }
}
```

PROGRAM 17.7b: **arg_check.c**

```
#include <stdio.h>
#include <stdlib.h>
void arg_check (int, int, char *, int);
```

FILE: **arg_check.h**

```
#include "quit.h"
void quit (char *message, int exit_status) {
   fprintf(stderr, message);
   exit(exit_status);
}
```

PROGRAM 17.7c: **quit.c**

```
#include <stdio.h>
#include <stdlib.h>
void quit (char *, int);
```

FILE: **cat quit.h**

The main program, **rec_deposit.c**, invokes **arg_check** to determine whether three arguments have been input. It invokes **sscanf** three times to convert the argument strings to floating point numbers. If any of the arguments is not positive, **quit** prints a user-specified message to the standard error and terminates the program. If validation succeeds, **main** invokes **compute** to make the actual computation.

17.13.2 Compiling and Linking the Application

The functions **arg_check** and **quit** are general enough to be used by other programs also, so it makes sense to preserve their object files. The technique of generating these files differ between GCC and Visual Studio. For GCC, the following invocation of **gcc -c** creates three .o files:

```
$ rm *.o                                                    First remove all object files
$ gcc -c rec_deposit.c arg_check.c quit.c
$ ls *.o
arg_check.o      quit.o      rec_deposit.o                  No .o file for compute
```

These object files have to be linked now to create the executable. However, this time the linker must be *explicitly* instructed to obtain the code of the **pow** function from the math library. This is done with the -l option which assumes that the library name has the "lib" prefix and ".a" suffix, and requires only the remaining portion of the filename to be specified.

Thus, to link libm.a, which contains the code for **pow**, you need to use the -lm option. Observe that this option must be placed at the end of the command line:

```
$ gcc -o rec_deposit rec_deposit.o arg_check.o quit.o -lm
$ _                                                         Compilation successful
```

We have also used the -o option to specify our own executable filename (**rec_deposit** instead of **a.out**). Run the program a number of times to test the user-defined functions:

```
$ rec_deposit
Three arguments required                                    arg_check working
$ rec_deposit 100 5 0
All arguments must be positive                              quit working
$ rec_deposit 100 5 2
Maturity amount: 215.249985                                 compute working
```

The output shows 215.25 as the maturity value of 100 units invested every year for two years at 5% interest. Since the program is working fine, do you need to hold on to the .o files? Yes, because if you later decide to modify, say, **quit.c**, you have to recreate quit.o before relinking the three .o files. If these files are not available, you have to regenerate them with **gcc -c**—a job that could have been avoided had you preserved the .o files.

Note: The procedure followed by Visual Studio differs sharply from the one followed by GCC. For Visual Studio, you don't need to issue a specific instruction for linking with the math library. The command **cl rec_deposit.c quit.c arg_check.c** does everything in one shot including creating the .obj files.

Note: This application doesn't have a separate object file for **compute**; its code is embedded in `rec_deposit.o`. This is not the right thing to do because reusable functions should be placed in separate files.

WHAT NEXT?

We have reached the end of the book and our tour of C—for the time being at least. C is a powerful but tricky language which can both excite and frustrate. With this text by your side, make sure that you simultaneously explore the wealth of resources available inside and outside the Web (Appendix E).

WHAT DID YOU LEARN?

The C preprocessor modifies a program containing #-prefixed *directives* before compilation. The directives help minimize the number of possible editing locations, which makes a program easier to modify.

The **#define** directive creates *named constants* and *macros* that perform substitution globally across the program. Macros accept type-independent parameters and can often replace functions.

The **#include** directive places the contents of a file in the program. This file usually contains **#define** directives along with declarations of global variables, structures and function prototypes.

The preprocessor uses the *conditional compilation* feature to enable parts of a program to be included or excluded from the compilation process.

The *bitwise* operators enable specific bits of an integer to be accessed and manipulated. The logical operators (&, |, ^ and ~) enable specific bits to be set or cleared. All bits of a number can be shifted left or right with the shift operators (<< and >>).

A function can be accessed by its pointer, which can be passed as an argument to another function.

A function can be designed to accept multiple arguments using three macros (**va_start**, **va_arg** and **va_end**) that operate on a list of type `va_list`.

A program can be organized across multiple files that are compiled separately to generate individual object files. Function declarations, **#define** and **#include** directives can be stored in separate files and merged during compilation.

OBJECTIVE QUESTIONS

A1. TRUE/FALSE STATEMENTS

17.1 The preprocessor outputs an error if a directive ends with a semicolon.

17.2 It is possible to redefine a named constant to a different value without first undefining it.

17.3 You cannot place comments in a line containing a preprocessor directive.

17.4 Like a variable, a defined constant can also be local.

17.5 Preprocessor directives can be placed anywhere in a program.

17.6 A defined constant embedded in a string of the replacement text of a macro is not expanded.

17.7 The **`#ifdef`** and **`#undef`** directives also work with variables.

17.8 It is an error to undefine a variable that has not been defined.

17.9 It is not an error to undefine a variable that has already been undefined.

17.10 **`#ifdef`** cannot be used with relational expressions.

17.11 Both the operators, `&` and `&&`, evaluate the logical value of an expression.

17.12 For the `^` operator, the result is 1 when both operands are 1.

17.13 Using bitwise operators, it is possible to multiply and divide numbers by 2.

17.14 The status of the LSB of an integer determines whether a number is even or odd.

A2. FILL IN THE BLANKS

17.1 The preprocessor acts only on lines that begin with a _____ .

17.2 The preprocessor doesn't check for _____ errors.

17.3 Expansion of a macro inside a string is handled by the _____ and _____ operators.

17.4 A directive can be continued to the next line by using a _____ at the end of the first line.

17.5 To insert a file from the current directory into a program, the filename must be enclosed by _____ _____ in the _____ directive.

17.6 The directive **`#ifdef DISTANCE`** and **`#ifndef DISTANCE`** are the same as **`#if`**_____ **`DISTANCE`** and **`if`**_____ **`DISTANCE`**, respectively.

17.7 The `&` and `|` operators act on the _____ of an integer.

17.8 The `&` and `|` are often used with a _____ to set or clear bits in an integer.

17.9 In the assignment `a = a & b`, the bits that are _____ in `a` have the corresponding bits _____ in `b`.

17.10 The value of the expression `64 >> 3` is _____ .

17.11 The statement `void (*f)();` represents the declaration of a _____ _____ .

17.12 A _____ function can be used as an argument to another function.

17.13 The `...` seen in a function prototype indicates that the function uses a _____ number of arguments.

A3. MULTIPLE-CHOICE QUESTIONS

17.1 The preprocessor (A) edits the source code, (B) expands all macros, (C) inserts contents of other files, (D) all of them.

17.2 In the directive **#define TEMP 12.5**, the type of 12.5 is (A) double, (B) float, (C) implementation-dependent, (D) none of them.

17.3 For the directive **#define VOLUME = X * X * X**, where X is also a defined constant, the preprocessor (A) replaces X with its defined value, (B) replaces VOLUME with its defined value, (C) A and B, (D) none of them.

17.4 If macro A uses macro B in its replacement text, then (A) A must be defined before B, (B) B must be defined before A, (C) the sequence is irrelevant, (D) the sequence is implementation-dependent.

17.5 An include file can contain (A) anything that makes sense to the compiler, (B) only preprocessor directives, (C) B plus function prototypes only, (D) C plus variables of primary data types only.

17.6 The directive **#ifdef (x < 0)**, where x is an int variable, doesn't work because (A) parentheses have been used, (B) **#ifdef** doesn't work with a variable, (C) a relational expression has been used, (D) B and C, (E) none of them.

17.7 Pick the odd operator out: (A) |, (B) &&, (C) ^, (D) <<, (E) >>.

17.8 Pick the odd directive out: (A) **#ifdef**, (B) **#else**, (C) **#elif**, (D) **#endif**.

17.9 The relational expression (a & b) == b tests whether the bits that are (A) set in b are also set in a, (b) cleared in b are set in a, (c) cleared in b are also cleared in a, (D) none of these.

17.10 The expression 4 << 3 evaluates to (A) 12, (B) 32, (C) 64, (D) 81.

17.11 The arguments to a function using a variable number of arguments is stored in the list (A) valist, (B) va_list, (C) va_args, (D) none of these.

A4. MIX AND MATCH

17.1 Match the operators with their functions:

(A) !, (B) &&, (C) <<, (D) |, (E) ^.

(1) Bitwise shift, (2) Other, (3) XOR, (4) NOT, (5) OR.

CONCEPT-BASED QUESTIONS

17.1 Compare the features of named constants and variables and list the circumstances when you would use one in preference to the other.

17.2 Compare functions with macros that use arguments and list their relative merits and demerits.

17.3 Explain with an example why the macro replacement text represented by an arithmetic expression must have both the macro parameters and the entire expression enclosed by parentheses.

17.4 For the following macro definition,

```
#define PRINTX(x) printf("x = %d\n", x);
```

explain why the macro works partially. How will you ensure that x is expanded in the **printf** control string also?

17.5 Explain with examples the circumstances that lead to compilation errors when including two .h files or one .h file that includes another .h file.

17.6 Explain with a suitable example how you can control conditional compilation from the command line of the OS and without modifying the program.

17.7 What will be the output of the following program? Are there any advantages in OR'ing the values 1, 2, 4, 8 etc.?

```
#include <stdio.h>
#define BOLD 1
#define ITAL 2
#define UNDERLINE 4
int main(void)
{
   printf("%d\n", BOLD | ITAL);
   printf("%d\n", BOLD | ITAL | UNDERLINE);
   return 0;
}
```

17.8 Explain the role of the three macros and the va_list data type that feature in the definition of a function using a variable number of arguments.

17.9 Instead of designing a function that accepts a variable number of integers as arguments, can we not use an array and pass its name and size to a function?

PROGRAMMING & DEBUGGING SKILLS

17.1 Write a program that uses a defined constant to (i) declare an int array, (ii) populate the array with **scanf**, (iii) print the last two array elements. The program must work with any array size by changing *only* the value of the defined constant.

17.2 Point out the errors in the following code:

```
#define SIZE3 = SIZE1 + SIZE2
#define SIZE1 10
#define SIZE2 20
printf("SIZE3 = %d\n", SIZE3);
  ...
int main(void)
  ...
```

17.3 Why doesn't the following macro work?

```
#define SQUARE (x) (((x) * (x)))
```

17.4 Define a macro named **VOLUME_SPHERE(radius)** to represent the volume of a sphere as a float using the formula vol = $4 \times 3.142 \times r^3 / 3$). The macro must represent 3.142 as a symbolic constant.

17.5 Define a macro named **IS_LEAP_YEAR(year)** that determines whether year is a leap year or not. Use it in a program that accepts the year from the keyboard and outputs an appropriate message using a **switch** statement.

17.6 Write a program containing the **DISPLAY_VAR** macro which displays, say, `area3 = 20` when invoked as **DISPLAY_VAR(area, 3)**. Accept three values from the keyboard into the variables `area1`, `area2` and `area3` and use the macro to display their values.

17.7 Write the relevant code for opening a file `foo` which resides on `/home/henry/progs` on Linux and `F:\henry\progs` on a Windows system. Use the symbolic constant `OS` to determine the operating system.

17.8 Point out the errors in the following two `.h` files, placed side by side, where `a.h` includes `b.h`.

File: `a.h`

```
#include <stdio.h>
#include "b.h"

#define ROWS 10
#define CUBE(y) ((y) * (y) * (y))

void quit (char *, int);

struct film {
   char title[30];
   short year;
};
```

File: `b.h`

```
#include <stdio.h>


#define ROWS 10
#define CUBE(y) ((y) * (y) * (y))

void quit (char *, int);

struct film {
   char title[30];
   short year;
};
```

17.9 Point out the errors in the following code:

```
#ifdef SIZE == 20
   #undefine SIZE
#end if
```

17.10 Write a program to populate a 10-element array with a mix of even and odd integers. Using a bitwise operator, print the even integers found in the array.

17.11 Write a program to check whether the least significant bit (LSB) of an integer is set.

17.12 Write a program that sets the bits 0, 1 and 3 of the integer 20. Save the result and then clear the same bits from the result.

17.13 Write a program to determine whether the *n*th bit of an integer is set, where *n* is obtained from the user.

17.14 Write a program that establishes that the result of the NOT (~) operation on an integer is determined by its data type.

17.15 Write a program that displays the set bits of a user-input integer in the form "`Bit` *n* `is set.`"

17.16 Write a program the uses the bitwise operators to perform the following operations: (i) compute the 16th power of 2, (ii) convert the decimal number 1532 to binary.

17.17 Write a program that determines whether the most significant bit (MSB) of a user-input integer is set.

17.18 Point out the errors in the following program containing a function that takes a variable number of arguments:

```
#include <stdio.h>
#include <std_arg.h>

int sum(int argc, ....);

int main(void) {
   printf("Sum of 3 integers = %d\n", sum(11, 22, 33));
   return 0;
}

int sum(int argc, ...)
{
   int i, total = 0;
   va_list arglist;

   va_start(arglist, argc);

   for (i = 0; i < argc; i++)
      total += va_arg(arglist);
   va_end(arglist);

   return total;
 }
```

17.19 Consider a program named **`array_handle.c`** that populates a 5-element `int` array with **`scanf`** and invokes the **`print_array`** function to print the array. This function is to be saved in **`print_array.c`**. The `.c` files can have only one preprocessor directive for including their respective `.h` files. Create the code for all four files.

APPENDIX

A Selective C Reference

A. Escape Sequences

Escape Sequence	*Significance*
`\a`	Bell
`\b`	Backspace
`\c`	No newline (cursor in same line)
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\v`	Vertical tab
`\\`	Backslash
`\`*n*	ASCII character represented by the octal value *n*, where *n* can't exceed 0377 (decimal value 255)
`\x`*n*	ASCII character represented by the hex value *n*, where *n* can't exceed 0xFF (decimal value 255)

B. Operator Precedence and Associativity

Operator	*Significance*	*Associativity*
( )	Function call	
[ ]	Array subscript	
., ->	Member selection (for structures)	
++, --	Postfix increment/decrement	L-R
++, --	Prefix increment/decrement	
+, -	Unary	
!	Logical NOT	
sizeof	Size of data or type	
(*data type*)	Cast	
*, &	Dereference/address (for pointers)	R-L
*, /, %	Arithmetic	L-R
+, -	Arithmetic (binary)	L-R
<<, >>	Bitwise shift left/right	L-R
<, <=, >, >=	Relational	L-R
==, !=	Relational	L-R
&	Bitwise AND	L-R
^	Bitwise XOR	L-R
|	Bitwise OR	L-R
&&	Logical AND	L-R
||	Logical OR	L-R
? :	Conditional (ternary)	R-L
=, +=, -=, *=, /=, %=	Assignment (arithmetic)	R-L
&=, ^=, |=, <<=, >>=	Assignment (bitwise)	R-L
,	Comma (expression delimiter)	L-R

C. Character-handling Functions (ctype.h)

isalnum(c)	Returns true if c is an alphanumeric character.
isalpha(c)	Returns true if c is an alphabetic character .
isascii(c)	Returns true if c is a 7-bit ASCII character within the range 0 to 127.
isblank(c)	Returns true if c is a blank character (space or tab) (C99).

(*Continued*)

`iscntrl(c)`	Returns true if `c` is a control character.
`isdigit(c)`	Returns true if `c` is a digit (0 to 9).
`isgraph(c)`	Returns true if `c` is a printable character except space.
`islower(c)`	Returns true if `c` is a lowercase letter.
`isprint(c)`	Returns true if `c` is a printable character including space.
`ispunct(c)`	Returns true if `c` is a printable character excluding space and alphanumeric characters.
`isspace(c)`	Returns true if `c` is a whitespace character (space, `'\f'`, `'\n'`, `'\r'`, `'\t'` and `'\v'`).
`isupper(c)`	Returns true if `c` is an uppercase letter.
`isxdigit(c)`	Returns true if `c` is a hexadecimal digit.
`toupper(c)`	Converts letter `c` to uppercase if possible. Returns value of converted letter if possible, otherwise returns `c`.
`tolower(c)`	Converts letter `c` to lowercase if possible. Returns value of converted letter if possible, otherwise returns `c`.

For Sections D, E, F and I, the parameter `FILE *`*stream* signifies a pointer associated with an opened file. This pointer is normally returned by the **`fopen`** function, but it can also represent `stdin`, `stdout` or `stderr`.

D. Character-oriented I/O Functions (`stdio.h`)

`fgetc(FILE *`*stream*`)` Returns the next character from *stream* on success, EOF on end-of-file or error.
`getc(FILE *`*stream*`)` As in `fgetc` except that `getc` may be implemented as a macro.
`getchar()` Returns the next character from standard input on success, EOF on end-of-file or error. Equivalent to `getc(stdin)` and `fgetc(stdin)`.
`ungetc(int` *c*`, FILE *`*stream*`)` Writes back the character *c* to *stream* to be read back by the next read operation. This operation must be performed before `ungetc` can be re-invoked. Returns *c* on success, EOF on error.
`fputc(int` *c*`, FILE *`*stream*`)` Writes the character *c* to *stream*. Returns *c* on success, EOF on error.
`putc(int` *c*`, FILE *`*stream*`)` As in `fputc` except that `putc` may be implemented as a macro.
`putchar(int` *c*`)` Writes the character *c* to standard output. Returns *c* on success, EOF on error.

For Sections E, F and G, the parameter `char *`*s* represents an array of type `char`, not a regular pointer to `char`.

E. Line-oriented I/O Functions (stdio.h)

`gets(char *s)` Reads a line from standard input into the buffer *s* until newline or EOF is encountered. Newline is replaced with `'\0'` and stored in the buffer. Returns *s* on success, NULL on error. Function doesn't check for buffer overrun, so its use has been deprecated and the function has been removed from the C11 specification.

`fgets(char *s, int size, FILE *stream)` Reads a maximum of *size* - 1 characters from *stream* into the buffer *s*. Reading stops when *size* - 1 characters have been read or a newline is encountered, whichever occurs first. If a newline is read, it is stored in the buffer, followed by `'\0'`. Returns *s* on success, NULL on error or when EOF occurs before any characters are read.

`puts(const char *s)` Writes the contents of the buffer *s* along with a trailing newline to the standard output. Writing stops if `'\0'` is encountered. Returns a non-negative number on success, EOF on error.

`fputs(const char *s, FILE *stream)` Writes the contents of the buffer *s* to *stream* but without the trailing `'\0'`. The newline character is not automatically written. Returns a non-negative number on success, EOF on error.

F. Formatted I/O Functions (stdio.h)

`printf(cstring, var1, var2, ...)` Writes *var1*, *var2*, ... to standard output using the formats specified in *cstring*. Returns number of characters printed on success, a negative value on error.

`fprintf(FILE * stream, cstring, var1, var2, ...)` As in `printf` except that the output is written to *stream* which can be a disk file.

`sprintf(char *s, cstring, var1, var2, ...)` See "String-handling functions."

`scanf(cstring, &var1, &var2, ...)` Reads data from standard input using the formats specified in *cstring* and saves the items in variables *var1*, *var2*, ..., whose addresses are specified as arguments. Returns number of items successfully matched or EOF if end of input is encountered before conversion of first item.

`fscanf(FILE *stream, cstring, &var1, &var2, ...)` As in `scanf` except that input is taken from *stream* which can be a disk file.

`sscanf(char *s, cstring, &var1, &var2, ...)` See "String-handling functions."

G. String-handling Functions (string.h)

`strlen(const char *s)` Returns length of *s*.

`strcpy(char *dest, const char *src)` Copies *src* to *dest*. Returns pointer to *dest*.

`strncpy(char *dest, const char *src, size_t n)` Similar to `strcpy` except that at most *n* bytes will be copied.

(*Continued*)

`strcat(char *dest, const char *src)` Appends *src* to *dest*. Returns pointer to *dest*.

`strncat(char *dest, const char *src, size_t n)` Similar to `strcat` except that at most *n* characters of *src* will be appended.

`strcmp(const char *s1, const char *s2)` Compares *s1* to *s2*. Returns negative, zero or positive integer depending on whether *s1* <, ==, or > than *s2*.

`strncmp(const char *s1, const char *s2, size_t n)` Similar to `strcmp` except that at most *n* characters of *s1* and *s2* are compared.

`strchr(const char *s, int c)` Searches character *c* in string *s*. Returns pointer to first occurrence of *c* in *s*, or NULL if *c* is not found in *s*.

`strrchr(const char *s, int c)` Similar to `strchr` except that the function returns a pointer to last occurrence of *c* in *s*, or NULL if *c* not found in *s*.

`strstr(const char *s1, const char *s2)` Searches *s2* in *s1*. Returns pointer to *s2* or NULL if *s2* not found in *s1*.

`sprintf(char *s, cstring, var1, var2, ...)` Writes *var1*, *var2*, ... to string *s* using the formats specified in *cstring*. Returns number of characters printed on success, a negative value on error.

`sscanf(char *s, cstring, &var1, &var2, ...)` Reads data from string *s* using the formats specified in *cstring* and saves the items in variables *var1*, *var2*, ..., whose addresses are specified as arguments. Returns number of items successfully matched or EOF if end of input is encountered before conversion of first item.

H. Number-String Conversion Functions (`stdlib.h`)

`atoi(const char *ptr)`	Converts string *ptr* to `int`, which is returned.
`atol(const char *ptr)`	Converts string *ptr* to `long`, which is returned.
`atoll(const char *ptr)`	Converts string *ptr* to `long long`, which is returned *(C99)*.
`atof(const char *ptr)`	Converts string *ptr* to `double`, which is returned.

I. File-handling Functions (`stdio.h`)

`fopen(pathname, mode)` Opens file whose name is represented by *pathname*. File can be opened in read (`"r"`), write (`"w"`) or append (`"a"`) *mode*. Using the + suffix, file can be opened in the following modes: read-write (`"r+"`), read-write with truncation (`"w+"`), read-append (`"a+"`). Returns pointer to `FILE`, NULL on error.

`fclose(FILE *stream)` Closes file associated with *stream* after flushing the buffer. Returns 0 on success, EOF on error.

(*Continued*)

`fread(void *p, size_t size, size_t num, FILE *stream)` Reads *num* items of data of *size* bytes in length from the file associated with *stream* and saves the data in the buffer *p*. Returns the number of items successfully read even when EOF or error is encountered. Does not distinguish between end-of-file or error.

`fwrite(void *p, size_t size, size_t num, FILE *stream)` Writes *num* items of data of *size* bytes in length from the buffer *p* to the file associated with *stream*. Returns the number of items successfully written even when error is encountered.

`fseek(FILE *stream, long offset, int whence)` Repositions the file position indicator of file associated with *stream*. The new position is computed by adding *offset* bytes to the position determined by *whence*. If *whence* is set to `SEEK_SET`, the offset is relative to the beginning of the file. If *whence* is set to `SEEK_CUR`, the offset is relative to the current position in the file. If *whence* is set `SEEK_END`, the offset is relative to the end of the file. Returns 0 on success, -1 on failure.

`ftell(FILE *stream)` Returns the current value of the file position indicator on success, -1 on failure. With the position indicator set to the beginning of the file, `ftell` returns the size of the file.

`rewind(FILE *stream)` Resets the file position indicator of the file associated with *stream* to the beginning of the file. Returns nothing.

`feof(FILE *stream)` Tests the EOF indicator for file associated with *stream*. Returns non-zero if EOF indicator is set, zero otherwise.

`ferror(FILE *stream)` Tests error indicator for file associated with *stream*. Returns non-zero if error indicator is set, zero otherwise.

`clearerr(FILE *stream)` Clears end-of-file and error indicators for file associated with *stream*. Returns nothing.

`remove(char *pathname)` Removes file associated with *pathname*. Returns 0 on success, -1 on failure.

`rename(char *old, char *new)` Renames file *old* to *new*, where *old* and *new* represent pathnames. Returns 0 on success, -1 on failure.

`tmpfile()` Creates and opens a temporary binary file in read-write (`w+b`) mode and deletes it on termination of the program. Returns pointer to `FILE` on success, NULL if a unique filename cannot be generated or the file cannot be opened.

J. Dynamic Memory Allocation Functions (`stdlib.h`)

`malloc(size_t size)` Allocates *size* bytes of memory. Returns pointer to allocated block, NULL on error.

`calloc(size_t num, size_t size)` Allocates memory for array of *num* elements of *size* bytes each. Returns pointer to allocated block, NULL on error.

`realloc(void *ptr, size_t size)` Changes size of memory block pointed to by *ptr* to *size* bytes. Returns pointer to the new memory block or *ptr* if the reallocation fails.

`free(void *ptr)` Frees memory block pointed to by *ptr*.

K. Memory-handling Functions (`string.h`)

`memcpy(void *dest, const void *src, size_t n)` Copies *n* bytes from memory block *src* to memory block *dest* if objects don't overlap. Returns pointer to *dest*.

`memmove(void *dest, const void *src, size_t n)` Similar to `memcpy` except that objects may overlap.

`memcmp(const void *s1, const void *s2, size_t n)` Compares first *n* bytes of memory block *s1* to *s2*, returns negative, zero or positive integer depending on whether *s1* <, ==, or > than *s2*.

`memchr(const void *s, int c, size_t n)` Returns pointer to first occurrence of *c* in first *n* characters of memory block *s*, NULL if *c* not found.

`memset(void *s, int c, size_t n)` Replaces each of the first *n* bytes of memory block *s* with character *c*. Returns pointer to *s*.

L. Mathematical Functions (`math.h`)

`abs(int x)` Returns the absolute value of the integer *x* (`stdlib.h`).

`fabs(double x)` Returns the absolute value of the floating-point number *x*.

`ceil(double x)` Returns the smallest integer that is greater than or equal to *x*.

`floor(double x)` Returns the largest integer that is less than or equal to *x*. C99 features the functions `floorf` and `floorl` with `float` and `long double` arguments, respectively.

`pow(double x, double y)` Returns the value of *x* raised to the power *y* as a `double`.

`sqrt(double x)` Returns the square root of *x*.

M. Miscellaneous Functions

`perror(const char *s)` Writes string *s* to the standard error. The string describes the nature of the error that was encountered by the last library function. Sets the `errno` variable and returns nothing (`stdio.h`, `errno.h`).

`exit(int status)` Terminates a program normally after closing all files and removing those created by the `tmpfile` function. `exit` passes the value of *status* to the operating system to signify the success or failure of the program. The symbolic constants, `EXIT_SUCCESS` and `EXIT_FAILURE`, may be used for this purpose (`stdlib.h`).

APPENDIX

B The ASCII Character Set

Character	*Decimal*	*Hex*	*Octal*	*Remarks*
NUL	0	00	000	Null
[Ctrl-a]	1	01	001	
[Ctrl-b]	2	02	002	
[Ctrl-c]	3	03	003	
[Ctrl-d]	4	04	004	
[Ctrl-e]	5	05	005	
[Ctrl-f]	6	06	006	
[Ctrl-g]	7	07	007	Bell (`\a`)
[Ctrl-h]	8	08	010	Backspace (`\b`)
[Ctrl-i]	9	09	011	Tab (`\t`)
[Ctrl-j]	10	0A	012	Newline (`\n`) (LF)
[Ctrl-k]	11	0B	013	Vertical tab (`\v`)
[Ctrl-l]	12	0C	014	Formfeed (`\f`) (FF)
[Ctrl-m]	13	0D	015	Carriage return (`\r`) (CR)
[Ctrl-n]	14	0E	016	
[Ctrl-o]	15	0F	017	
[Ctrl-p]	16	10	020	
[Ctrl-q]	17	11	021	
[Ctrl-r]	18	12	022	
[Ctrl-s]	19	13	023	
[Ctrl-t]	20	14	024	
[Ctrl-u]	21	15	025	
[Ctrl-v]	22	16	026	
[Ctrl-w]	23	17	027	

(*Continued*)

Character	*Decimal*	*Hex*	*Octal*	*Remarks*
[Ctrl-x]	24	18	030	
[Ctrl-y]	25	19	031	
[Ctrl-z]	26	1A	032	
[Ctrl-[ ]	27	1B	033	Escape
[Ctrl-\]	29	1C	034	
[Ctrl-]]	29	1D	035	
[Ctrl-^]	30	1E	036	
[Ctrl-_]	31	1F	037	
(space)	32	20	040	Space
!	33	21	041	Exclamation mark or bang
"	34	22	042	Double quote
#	35	23	043	Pound sign
$	36	24	044	Dollar sign
%	37	25	045	Percent
&	38	26	046	Ampersand
'	39	27	047	Single quote
(	40	28	050	Left parenthesis
)	41	29	051	Right parenthesis
*	42	2A	052	Asterisk
+	43	2B	053	Plus sign
,	44	2C	054	Comma
-	45	2D	055	Hyphen
.	46	2E	056	Period
/	47	2F	057	Slash
0	48	30	060	
1	49	31	061	
2	50	32	062	
3	51	33	063	
4	52	34	064	
5	53	35	065	
6	54	36	066	
7	55	37	067	
8	56	38	070	
9	57	39	071	
:	58	3A	072	Colon
;	59	3B	073	Semicolon

(*Continued*)

Character	*Decimal*	*Hex*	*Octal*	*Remarks*
<	60	3C	074	Lef chevron
=	61	3D	075	Equal sign
>	62	3E	076	Right chevron
?	63	3F	077	Question mark
@	64	40	100	At sign
A	65	41	101	
B	66	42	102	
C	67	43	103	
D	68	44	104	
E	69	45	105	
F	70	46	106	
G	71	47	107	
H	72	48	110	
I	73	49	111	
J	74	4A	112	
K	75	4B	113	
L	76	4C	114	
M	77	4D	115	
N	78	4E	116	
O	79	4F	117	
P	80	50	120	
Q	81	51	121	
R	82	52	122	
S	83	53	123	
T	84	54	124	
U	85	55	125	
V	86	56	126	
W	87	57	127	
X	88	58	130	
Y	89	59	131	
Z	90	5A	132	
[	91	5B	133	Left square bracket
\	92	5C	134	Backslash
]	93	5D	135	Right square bracket

(*Continued*)

Character	*Decimal*	*Hex*	*Octal*	*Remarks*
^	94	5E	136	Caret or Hat
_	95	5F	137	Underscore
`	96	60	140	Backquote or Backtick
a	97	61	141	
b	98	62	142	
c	99	63	143	
d	100	64	144	
e	101	65	145	
f	102	66	146	
g	103	67	147	
h	104	68	150	
i	105	69	151	
j	106	6A	152	
k	107	6B	153	
l	108	6C	154	
m	109	6D	155	
n	110	6E	156	
o	111	6F	157	
p	112	70	160	
q	113	71	161	
r	114	72	162	
s	115	73	163	
t	116	74	164	
u	117	75	165	
v	118	76	166	
w	119	77	167	
x	120	78	170	
y	121	79	171	
z	122	7A	172	
{	123	7B	173	Left curly brace
|	124	7C	174	Vertical bar or Pipe
}	125	7D	175	Right curly brace
~	126	7E	176	Tilde
	127	7F	177	Delete or Rubout

APPENDIX

C Glossary

abstract data type (ADT) A logical description of the data along with a set of permitted methods that can access and manipulate the data. A **stack** is an ADT that can be implemented by a **linked list** or an **array**.

algorithm A language-independent sequence of unambiguous steps required to solve a computing problem. Any computing problem can be represented by a program if the problem can also be represented as an algorithm.

ANSI The American National Standards Institute that has developed the universally accepted standard for C. Versions of ANSI C are named C89, C99 and C11.

application software Software related to a specific application. Microsoft Word, SAP and Whatsapp belong to this category.

archive Term used to describe a collection of the object codes of multiple functions. It is usually a file that is linked by the **linker** during the compilation phase. Same as **library**.

argument A number or a string that is supplied additionally when invoking a C function or a command of the operating system.

Arithmetic and Logic Unit (ALU) The calculator of the CPU. Also carries out relational tests like comparing two numbers.

arithmetic operator An operator used for computation and represented by the symbols `+`, `-`, `*`, `/` and `%`.

array A collection of data of the same type that are laid out contiguously in memory. Each piece of data is accessed by a common name and a varying index.

array element A single data item stored in an array. The name `arr[10]` represents the 11th element of the array `arr`.

array index An integer that refers to the position of an array element in the array. The element `arr[5]` is accessed by the index `5`. Same as **array subscript**.

array subscript Same as **array index**.

ASCII sequence A numeric sequence specified by ASCII (American Standard Code for Information Interchange) for coding the character set of a computer. Control characters occupy the top slots, followed by numerals, uppercase letters, and then lowercase. Sequence used for sort operations.

assembler A component of the compiling software that converts a program to assembly language.

assembly language A second-generation programming language that uses mnemonic names to represent program instructions. One assembly language instruction represents one CPU instruction.

assignment operator A binary operator represented by the = symbol. The value on its right is assigned to the variable on its left. Also used with the arithmetic and bitwise operators (like += and |=) to combine two operations.

associativity An attribute of an operator that determines whether it is associated with the operand on its left or right. Called into play only when two operators having the same precedence share an operand. See also **precedence**.

automatic variable A variable that comes into existence when a function is called and disappears when the function terminates. By default, a variable is automatic but it can also be explicitly specified with the keyword `auto`.

Basic Input Output System (BIOS) A small program that resides in an EEPROM on the motherboard. Conducts system checks and loads a minimum set of **device drivers**. Eventually transfers control to the **boot loader** for loading the operating system.

Binary Coded Decimal (BCD) A form of representation of a decimal number in binary where each decimal digit is stored in binary form.

binary file A file that contains any character of the 8-bit ASCII set. Includes executable code and library archives. The **`fopen`** function can open a file in the binary mode. See also **text file**.

binary numbering system A numbering system that uses two digit symbols (0 and 1) to represent a numeral. Also known as base-2 system.

binary operator An operator that works with two **operands**. Most C operators belong to this type.

binary search A search mechanism that works on sorted data in multiple passes. It is more efficient that **sequential search** because each pass halves the search domain.

bit A digit used by the binary numbering system (base-2). A bit has the value 0 or 1. Abbreviated from **bi**nary digi**t**.

bit field A structure member that has a size of one or more bits instead of one or more bytes. A one-byte integer can be divided into as many as eight bit fields.

bitwise operator An operator that works on individual bits of an integer. Often used with a **mask** to determine whether a bit in an integer is set or cleared. Also used to shift the entire set of bits to the left or right.

bitwise shift Refers to the shifting of all bits of an integer using the `<<` and `>>` operators. Each shift to the left or right multiplies or divides the integer by two, respectively.

boot loader A small program that resides in the first sector of a bootable device. The BIOS eventually transfers control to the boot loader which loads the operating system.

bottom-up programming A programming methodology that advocates the design of the individual components of a system before the entire system has been visualized. See **top-down programming**.

bounds checking A check made on an array to ensure that it is not used with an illegal subscript (like `a[-5]` or `a[25]` for a 10-element array).

browser A program used to view HTML pages of the World Wide Web. Common Web browsers include Mozilla Firefox and Google Chrome.

bubble sort A technique used to sort data in multiple passes by comparing adjacent data items and swapping them depending on whether one is greater or less than the other. See also **selection sort**.

buffer A block of memory used by many I/O functions to temporarily store data before it reaches the final destination. Keyboard input, terminal display and read-write operations on disk files take place through their buffers.

buffer flushing Term used to clear the buffer by setting all its bytes to zero or NULL or to forcibly send the data to its final destination.

bus topology A type of networking where all nodes are connected to a single cable acting as a bus.

byte A unit of memory size representing a group of eight bits. A byte in C is just wide enough to hold the entire character set of the machine.

C89, C99, C11 Terms used to refer to the ANSI C specifications that were issued in 1989, 1999 and 2011, respectively.

cache memory A block of memory that stores the most frequently requested data in order to speed up data transfer. Depending on their speed, hardware caches are classified as L1, L2, L3 etc.

callback function A function that can be used as an argument to another function.

cast An operator that forces a variable, constant or expression to have a compatible data type. It is specified by enclosing the desired type in parentheses and placing it before the variable, constant or expression.

cast operator A set of parentheses, `()`, that encloses a data type to force the data item on its right to have that type. The expression `(int) x` forces `x` to be an `int`, but only in that expression.

Central Processing Unit (CPU) The brain of the computer that is usually contained in a single microprocessor chip. The CPU processes program instructions and interacts with other parts of the computer including memory. Most of the work of the CPU is carried out by the **Arithmetic and Logic Unit (ALU)**.

character The smallest unit of information. The press of a key generates a single character, while 7-bit and 8-bit ASCII have a set of 128 and 256 characters, respectively.

Character User Interface (CUI) The text-based interface used by the operating system to interact with a user. A CUI uses a shell which accepts user input and interprets them as commands to be executed. See also **Graphical User Interface (GUI)**.

command Normally the first word entered at the shell prompt. It is usually an executable file, but can also be built into the shell of the operating system (like **`DIR`** in MSDOS and **`cd`** in Linux).

command line A complete sequence of a command or program, its options, filenames and other arguments that are specified at the prompt of the shell.

command-line argument All words in a **command line** except the name of the command.

comma operator A binary operator that evaluates two or more expressions from left to right and returns the value of the right-most expression.

compiler A program that translates source code to object code before the latter is converted into executable code by the **linker**.

compound statement A set of statements enclosed by curly braces and addressed as a group. Associated with a function, the **`if`**, **`while`**, **`do-while`** and **`for`** constructs. Same as **control block**.

computer generations A hardware-based categorization of computers. A computer of a generation is faster, smaller and more powerful than its counterpart of the preceding generation.

concatenation The combination of two or more entities. Term used in connection with the **`cat`** (Linux) and **`COPY`** (MSDOS) commands of the operating system, and the **`strcat`** function.

conditional compilation Term used to determine whether a section of program code will be included in the compilation process. The affected code section is identified by the **`#ifdef`**, **`#ifndef`** and **`#if`** directives of the preprocessor.

conditional expression An expression formed with the **conditional operator** comprising the **`?`** and **`:`** symbols. Its value is determined by its three sub-expressions.

conditional operator A **ternary operator** comprising the **`?`** and **`:`** symbols and used with three operands. See also **conditional expression**.

constant An unchangeable value not usually associated with a variable. A variable or pointer declaration with the **`const`** qualifier is also treated as a constant.

constant pointer Term used to refer to a pointer whose value cannot be changed. The name of an array or function represents a constant pointer. See also **pointer constant**.

control block Same as **compound statement**.

control expression An expression used with the **`if`**, **`while`**, **`do-while`** and **`for`** statements to determine whether a loop will be entered or perform the next **iteration**.

control structure Term used for constructs that disturb the sequential flow of a program. Refers to the conditional and repetitive constructs of a language.

dangling pointer A pointer that remains in existence even after its pointed-to block has been freed. A dangling pointer must be set to NULL to prevent its accidental use.

declaration Term used to make type information of a variable or function available to the compiler without creating memory. For variables of the primary data type, declaration and **definition** are combined.

decrement operator Represented by `--`, which can be prefixed or postfixed to a variable to decrease its value by one. The statement `printf("%d", x--);` prints the value of `x` *before* it is decremented, while `printf("%d", --x);` prints `x` *after* it is decremented.

defined constant Same as **symbolic constant**.

definition Term used to create memory for a variable or function after it has been declared. Represents a separate activity for a structure and function. See also **declaration**.

dequeue A method associated with a **queue** that deletes a data element from its front end.

dereference The property of a pointer to access the value at the location stored by it. Dereferencing is done by applying the unary * operator on the pointer variable or expression.

derived data type A non-primary data type derived from a primary data type. Term used to refer to an array and pointer.

device driver A type of system software invoked by the operating system to access a hardware device. Usually developed by the manufacturer of the device.

directory A folder that forms a component of a file system . A directory houses other files and subdirectories. MSDOS and Linux support separate commands for handling directories.

Domain Name System (DNS) A support service that makes the Internet work. One or more DNS servers acting in tandem translate a **hostname** (like *www.google.com*) to its **IP address**.

dynamic memory allocation The allocation of a contiguous block of memory by a program while it is running. The outcome or side effect of the `malloc`, `calloc` and `realloc` functions.

editor A program that edits a text file. `vim` and Notepad are well-known editors in Linux and Windows.

enqueue A method associated with a **queue** that adds a data element to its tail end.

entry-controlled loop A repetitive construct that is entered only if its control expression evaluates to true. Includes the `while` and `for` constructs.

enumeration A user-defined data type that assigns names to a set of integer values known as **enumerators**. By default, the values begin with 0, but can be explicitly assigned to any other value with the = operator.

enumerator The integer values represented by names that form an **enumeration**.

end-of-file (EOF) A condition that signifies the end of data. Also represented as a symbolic constant (usually having the value -1).

escape sequence Usually a two-character sequence containing the \ as a prefix. Used in strings (like \t and \n) that are interpreted by **printf**. Also used with an ASCII octal or hexadecimal value to represent *any* character.

exit-controlled loop A repetitive construct like the **do-while** loop that is executed at least once and terminated by its control expression. See also **entry-controlled loop**.

exit status The argument of the **exit** function that is passed to the operating system on termination of a program. A value 0 indicates successful (true) termination, while any other value indicates unsuccessful (false) termination.

expression A combination of operators and operands that include variables and constants. An expression evaluates to a single numeric or logical value.

external command A program of the operating system that exists as a separate executable file. See also **internal command**.

external variable A variable defined outside a function. If declared before **main**, an external variable is visible in every function of the program.

Fibonacci sequence A sequence of integers beginning with 0 and 1 and where subsequent terms are formed by adding the two preceding terms.

file A container for storing binary or text data. Operating system commands and the **fopen** function accesses a file by its name.

file offset pointer See **file position indicator**.

file opening mode The second argument used by the **fopen** function to determine whether a file can be read, written or appended to. Uses the strings "r", "w" and "a", and the + as an optional suffix.

file permission A protection mechanism used in Linux that determines the read, write and execute permissions of a file. The permissions can be altered with the chmod command.

file pointer The pointer returned by the **fopen** function on opening a file. The pointer refers to a FILE structure which stores the attributes of the opened file. The file pointer is subsequently used by other file-handling functions.

file position indicator Represents the location in a file where the next read or write operation will take place. This location is maintained in the FILE structure and is updated every time a read or write operation on the file takes place. Also called **file offset pointer**.

file seek Refers to the movement of the **file offset pointer** to a specific location in a file. Represented by the **fseek** function which uses the symbolic constants SEEK_SET, SEEK_END and SEEK_CUR to move the indicator to the terminal ends of a file.

file system A hierarchical structure of files and directories having a root directory at the top. A hard disk, CD-ROM, DVD-ROM and flash memory device has a file system.

File Transfer Protocol (FTP) A TCP/IP application protocol that transfers files between two remote machines. Also represented by the **ftp** command available in Linux and MSDOS.

flag A single character used with a function argument to alter its default attribute. The **`printf`** and **`scanf`** functions use several flags.

flash memory A type of secondary storage based on the EEPROM. The USB-based pen drive, SD card and solid state disk (SSD) belong to this category.

floating-point number A number with a decimal point. Also known as **real number**.

flowchart A pictorial representation of an algorithm that clearly depicts the steps to be taken to solve a computing problem. Uses standard symbols for representing I/O devices, decision-making and data flow.

format specifier A placeholder embedded in the control string of the **`printf`** and **`scanf`** functions. Often represents a signed (like `%d`) or unsigned (like `%u`) number or a string (like `%s`).

fourth-generation language (4GL) A powerful language that has practically no dependence on the hardware. One 4GL statement can replace an entire 3GL program. The Structured Query Language (SQL) belongs to this type.

function Name given to a set of program statements that are invoked as a bunch by that name. A function may or may not accept arguments and may or may not return a value. A user can define their own functions.

function argument Additional information passed to a function on its invocation. Declaration of a function specifies the type and number of its arguments and the return value. For the function call **`scanf("%d", &x);`**, `"%d"` and `&x` are two function arguments.

function declaration A statement that specifies, for a function, the number and type of its arguments and return value, if any. It must precede the **function definition**. Also known as **function signature** and **function prototype**.

function definition The code associated with a C function that will be executed when the function is called. The compiler checks the number and type of arguments and the type of return value in both declaration and definition.

function parameter Term used to describe a formal argument in the function definition. During invocation of a function, data is copied from the actual argument to its matching parameter used in the function body.

function prototype Same as **function declaration**.

function signature Same as **function declaration**.

fundamental data type The primary data type supported by C. Includes integers, floating point numbers and characters.

generic pointer A pointer that has no specific data type but can be pointed to any data type without using a cast (since C89). Its data type is represented as `void *`. Same as **void pointer**.

global variable Same as **external variable**.

Graphical User Interface (GUI) Interface used by the operating system for user interaction without presenting a shell. A user invokes a program in a GUI by using mouse clicks on icons that represent programs. See also **Character User Interface (CUI)**.

hard disk Secondary storage device where data are organized in multiple platters, tracks and sectors. A read/write head moves radially to the center of the disk to access data. Can be fixed inside the computer housing or exist in portable form for use with the USB port.

header file Same as **include file**.

head pointer A pointer variable that points to the first node of a **linked list**. All access to the list is made by following the head pointer. See also **tail pointer**.

heap A data structure in memory used by the **`malloc`**, **`calloc`** and **`realloc`** functions to dynamically allocate memory. Shares the memory region with the **stack**. Heap may become full if the allocated memory is not freed with the `free` function.

hexadecimal numbering system A numbering system that uses 16 digit symbols (0 to 9 and A to F) to represent a numeral. C uses `0x` prefix and `%x` format specifier for a hexadecimal number. Also known as base-16 system.

High Definition Multimedia Interface (HDMI) An industry standard for moving high-quality audio and video data through a single cable. HDMI ports are found on computers, laptops, HDTVs, projectors and home theaters.

high-level language A language that belongs to a generation higher than assembly language. C, C++ and Java belong to this category.

hostname The name of a host or node that is unique in the network. Used with multiple dot-delimited strings to create a **fully qualified domain name (FQDN)** recognized by the Internet.

hub A central device that is connected to multiple devices in a network. Unlike a **switch**, a hub broadcasts data it receives from one node to *all* nodes in the network.

Hyper Text Markup Language (HTML) The universal language used for coding Web documents. Characterized by the presence of tags which can highlight text, handle multimedia and transfer control to a resource in another machine.

Hyper Text Transfer Protocol (HTTP) The application protocol used to access HTML documents hosted on Web sites.

implicit promotion The automatic type conversion of variables and constants used in a expression. As far as possible, the compiler promotes a lower type to a higher type.

include file A text file (like `stdio.h`) that is included—normally at the beginning of a program—with the **`#include`** directive of the preprocessor. This file contains symbolic constants and declarations for functions and structures used by multiple programs. Also called **header file**.

increment operator Represented by `++`, which can be prefixed or postfixed to a variable to increase its value by 1. The statement **`printf("%d", x++);`** prints the value of `x` *before* it is incremented by one, while **`printf("%d", ++x);`** displays `x` *after* the increment operation.

indirection Refers to the indirect use of a pointer to access a value instead of accessing the value directly. The indirection operator, *, is used to fetch the value from a memory location to which the pointer points. Concept used in **dereference**.

infinite loop A **while**, **do-while** or **for** loop that never terminates. The **break** statement transfers control out of the loop while **continue** starts the next iteration.

initialization The assigning of a value to a variable at the time of its declaration. The statement `int x = 0;` initializes `x`, but `x = 0;` assigns `x`.

instance The creation of an object in memory based on a template. Term used to refer to a structure whose template is created by declaration and instantiated by definition.

internal command A category of operating system commands that don't exist as separate executable files. The **DIR** and **MKDIR** commands of MSDOS and the **cd** command of Linux are internal commands of the shell.

Internet The super network of networks connected by the TCP/IP protocol. See also **World Wide Web**.

internet Two or more networks connected by the TCP/IP protocol. All Internet services can run on an internet. Same as **intranet**.

interpreted language A language that is not designed for conversion of source code to an executable. Each statement of an interpreted program is converted and executed on the CPU before the next statement is taken up. A language may be interpreted or compiled depending on the way it is implemented.

intranet Same as **internet**.

IP address A set of four dot-delimited numbers that represents the location of a node in the Internet. A **Fully Qualified Domain name (FQDN)** has to be converted to its IP address for data to reach its destination.

iteration Term used to refer to the execution of a set of statements associated with the **while**, **do-while** and **for** loops. An iteration is repeated if the **control expression** evaluates to true.

kernel The core of the UNIX/Linux operating systems which is accessed by the user through the shell. Programs access the hardware by making calls to the kernel.

key variable A variable used in the **control expression** of a loop to determine its continuity.

K&R C The original specification of the C language as proposed by Kernighan and Ritchie in the first edition of their book. Has since been replaced with ANSI standards C89, C99 and C11.

ladder Term used to refer to the **if-else-if-else**... construct.

language generations A categorization of computers from the software point of view. A higher generation language has greater power, convenience of programming and reduced proximity to the hardware.

library Same as **archive**.

line A sequence of characters terminated by the newline character. Text files are interpreted as a group of lines.

linked list A data structure represented by a linear set of nodes having the `struct` data type. Each node is created by dynamic memory allocation and has a field that points to the next node except that the last node has this field set to NULL.

linker A program that converts the object code created by the compiler to a standalone executable. The linker handles **unresolved references** left by the compiler to include the code of functions used in the program.

loader A component of the operating system that loads a program from disk to memory. The loader initiates program execution by transferring control to the **`main`** function.

local variable A variable declared inside a function that ceases to exist after termination of the function. The scope of a local variable is limited to the function in which it is declared. See also **global variable**.

logical operator An operator in an expression that is evaluated to one of two logical values (0 or 1). Logical operators include `&&`, `||` and `!` that are used in logical expressions, and `&`, `|` and `~` used in bitwise expressions.

logic gate Refers to one of the basic building blocks of digital systems. Most gates use two input lines and one output line, where the output is logically determined by the **truth table** of the gate.

loop A repetitive construct that repeats a body of statements as many times as its control expression permits. Term used to refer to the **`while`**, **`do-while`** and **`for`** constructs.

machine language A meaningful sequence of 0s and 1s that are executed on the CPU. All program instructions and data must be converted to machine language before execution.

macro An abbreviative feature of the preprocessor that uses the **`#define`** directive to create an alias for text. A macro without arguments creates a **symbolic constant** while one with arguments is invoked like a function.

`main` function The central function of a program that begins execution immediately after the program is loaded into memory. Every executable program must have this function which can be invoked with or without arguments.

malware Name given to a category of malicious software. Includes viruses, spyware (that steal confidential information) and trojan horses (that appear to look innocent).

manifest constant Same as **symbolic constant**.

mask An integer that is used as the right operand of a bitwise expression to determine or change the status of bits in the target integer.

matrix A rectangular array of numbers comprising rows and columns where each item is accessed with two comma-separated subscripts. Two compatible matrices can be added, subtracted and multiplied. A matrix is implemented in C by a two-dimensional array.

memory Hardware storage available in the computer and peripherals to store programs and data. Primary memory is volatile while secondary memory is non-volatile.

memory leak A situation caused by the failure to free dynamically allocated memory after use. A major cause of heap depletion.

mesh topology A type of networking where all nodes are connected to one another, thus offering multiple routes for data to travel.

microprocessor An electronic chip that contains the CPU and its associated memory. Used in computers, smartphones and embedded computers like washing machines and microwave ovens.

modem A device (**mo**dulator-**dem**odulator) that converts analog signals to digital, and vice versa. Used for connecting to the Internet or any TCP/IP network through a telephone line.

modulus operator Represented by the `%` symbol, this binary operator evaluates the remainder of an integer division.

multi-tasking operating system An operating system where multiple programs reside in memory and the OS runs them concurrently using the concept of **time sharing**.

multi-user operating system An operating system (like UNIX/Linux) where multiple users share the computer's CPU and memory.

named constant Same as **symbolic constant**.

nesting Term used to refer to the inclusion of an entity having the same type as the entity enclosing it. Often used to describe the **`if`**-**`if`**-**`if`**... construct, a structure containing another structure as a member, or a symbolic constant that contains another symbolic constant in its defined value.

newline One or more characters that signify the end of a line in a text file. UNIX and Mac OS use a single character while MSDOS uses two characters to represent a newline.

node Term used in hardware to refer to a device in a network. In C, term refers to a structure that is a component of a **linked list**.

NUL character The first character in the ASCII list having the value zero. Used to terminate a string.

NULL A symbolic constant representing a pointer value that is returned by some functions on failure. Normally has the value zero but is guaranteed not to point to any existing object. See also **null pointer**.

null pointer A pointer that has the value `NULL`. See also **NULL**.

null statement A line containing a semicolon which does nothing. Often used as a dummy statement in an **`if`** or **`while`** construct.

object code The code created by the compiler without including the code of functions. This code is further processed by the **linker** to create an executable.

Object-Oriented Programming (OOP) A programming paradigm that considers an application as a collection of objects where the data and methods to access it are encapsulated. Java and C++ are OOP languages.

octal numbering system A numbering system that uses eight digit symbols (0 to 7) to represent a numeral. C uses `0` prefix and `%o` format specifier for an octal number. Also known as base-8 system.

one's complement A form of representation of a negative number obtained by reversing the bits of its positive counterpart. Features positive and negative zeroes and used today as an assistant to the **two's complement** system.

operand An item or quantity that is acted upon by an **operator**. In the expression, `a + b`, `a` and `b` are operands of the `+` operator.

operating system (OS) A special software that handles the resources of a computer. Loaded at boot time, the OS manages memory, processes and files. Programs communicate with the OS for accessing a hardware device.

operator A token comprising one or more symbols used for performing a simple operation like addition (`+` and `++`), comparison (`==`) or bit shifting (`<<`). An operator may be unary (like `!`), binary or ternary (`?:`).

option A token normally beginning with a `-`, which changes the default behavior of a Linux command. The `ls -l` command has `-l` as its option. Counterpart in MSDOS is the **switch** which uses a / instead of `-`.

parameter passing Term used to refer to the transfer of values from function arguments to parameters. Also see **pass-by-value**.

pass-by-reference Term used in C++ to refer to the passing of a reference of an object to its parameter, but without copying the object. A change made to the parameter actually changes the object but without using a pointer.

pass-by-value The copying of *values* of the arguments of a function to its parameters. Because of pass-by-value, a function can't swap the values of its two arguments unless pointers are used.

`PATH` A shell variable in Linux and an internal command in MSDOS that contains a list of directories that the shell will look in for locating a command.

pathname A sequence of one or more filenames delimited by a / (in Linux) or \ (in MSDOS). All except the last filename must be directories.

peek A method associated with a **stack** or **queue** to retrieve its first element without deleting it.

pointer A derived data type that stores the address of a variable of any type. An `&`-prefixed variable and the name of an array evaluate to a pointer. A pointer is dereferenced with the `*` operator.

pointer constant A pointer whose pointed-to-value doesn't change. In the assignment `char *p = "Elon Musk";`, `p` is a pointer constant because the value of the string can't be changed. See also **constant pointer**.

pop A method associated with a **stack** for removing the last element that was inserted (pushed). See also **push**.

port A docking point for a peripheral device. Includes USB, HDMI, VGA, RJ45, serial and parallel ports.

Power-On-Self-Test (POST) A test carried out by the BIOS during boot time. Checks RAM and connectivity of devices and ports. Obtains the device information from a battery-powered CMOS chip.

precedence The priority accorded to an operator in an expression. When an operand is shared by two operators, the operator with a higher precedence gets priority during evaluation. See also **associativity**.

precision For a floating point data type, term refers to the number of significant digits used in its representation. For **`printf`**, term refers to the number of decimal places to print (floating point numbers), minimum field width (integers) and number of characters to print (character strings).

preprocessor A program that acts on **directives** beginning with a # in the source code before the compiler acts on the edited code. A preprocessor supports macros (**`#define`**), file inclusion (**`#include`**) and conditional compilation (**`#ifdef`**, **`#ifndef`** and **`#if`**).

preprocessor directive A #-prefixed instruction executed by the preprocessor. Directives usually occur at the beginning of a program and don't end with a semicolon.

primary memory The memory that resides on the computer motherboard and directly accessed by the CPU. A program is first loaded to primary memory before its instructions are executed. Includes RAM, ROM, cache and registers.

program counter (PC) A CPU register that stores the address of the next instruction to be executed by the CPU.

prompt A string that shows the position of the cursor. The appearance of a prompt generally indicates that the previous command has completed its run.

prototype Same as **function declaration**.

push A method associated with a **stack** for inserting an element at its top. See also **pop**.

queue An **abstract data type (ADT)** that operates on the first-in-first-out (FIFO) principle. An element in a queue is inserted from the tail end and deleted at the head end. Can be implemented by an array or linked list.

ragged array An array whose elements are arrays of different sizes. The declaration `char **arr;` is used to create an array of pointers where the elements can be made to point to strings of unequal length.

Random Access Memory (RAM) Volatile primary memory that stores the instructions and data of all programs currently in execution. Faster than secondary memory but slower than **cache** and **registers**. Exists today as static RAM (SRAM) and dynamic RAM (DRAM).

Read Only Memory (ROM) Primary memory that in its most basic form can be read but not written. Stores special programs (like the BIOS) or information that don't change. Erasable and programmable sub-categories available as PROM, EPROM and EEPROM.

real number Same as **floating-point number**.

real-time operating system An operating system that provides a guaranteed response time. Used in situations where the response needs to be instantaneous (like in driverless cars and process control).

recursion A form of cloning that uses instructions to specify themselves. Concept used in a C function that repeatedly calls itself.

recursive function A function that repeatedly calls itself until it encounters the terminating condition which halts further recursion. Too many recursive calls can lead to **stack overflow**.

redirection A feature of the shell for reassigning the standard input and standard output of a program. Using the < and > symbols, redirection can direct a data stream to a disk file or originate from it.

register In hardware, term refers to high-speed memory that is directly connected to the CPU. In C, term refers to the storage class of a variable, which involves these high-speed registers.

relational expression An expression containing a relational operator (like < or ==) which evaluates to a true or false value.

relational operator A binary operator that tests the relationship between two operands. Represented by symbols like >, <, ==, and !=.

ring topology A type of networking where all nodes are connected in a loop without using a hub. Failure of one node brings down the network.

root The top-most directory in a hierarchical file system which has no parent. Is indicated by the / (Linux) and \ (MSDOS) symbols. Also refers to the top-most node in a **tree** data structure.

router A special device that routes packets from one network to another.

runtime Term used to refer to a program in execution. A program that clears compilation can still fail at runtime.

scan set A format specifier of the **`scanf`** function that uses a set of characters enclosed by [and] for matching characters that either belong or don't belong to the set. The format specifier `%[^\n]` represents a scan set that matches an entire line not containing the newline character.

secondary memory Non-volatile memory that is not directly connected to the CPU. Used for offline storage and includes the hard disk, magnetic tape, CD-ROM and DVD-ROM. See also **primary memory**.

secure shell A set of applications used in a network that encrypts an entire session.

selection sort A technique used to sort data in multiple passes by comparing each data element with the start element and swapping them depending on whether one is greater or less than the other. See also **bubble sort**.

semantic error An error in program logic that is syntactically correct but fails at runtime. See also **syntax error**.

sequential search A technique of scanning a list for a specific data item in a linear manner from the beginning or end of a list.

shell The command-line interface of the Linux and MSDOS operating systems. Users type in a command at the shell, and the operating system interprets and executes it as a command.

side effect The action performed by a function apart from returning a value. A function may be invoked for its return value, side effect or both. The **`scanf`** function returns the number of characters successfully read, but it also has the side effect of assigning data to variables.

signature See **function declaration**.

signed number A number where the most significant bit (MSB) is reserved for the sign. The range of a signed number is half that of an unsigned one.

sorting Term used to refer to the ordering of data. Data is easily sorted in the ASCII sequence.

source code A program written in a language that can be compiled, assembled or interpreted.

stack A linear data structure in memory used by a function on its invocation. A stack can overflow if several function calls are made without returning (like in recursive functions). Also refers to an **abstract data type (ADT)** that operates on the last-in-first-out (LIFO) principle. An element in a stack is pushed in from its head end and popped out from the same end. Can be implemented by an array or linked list.

stack overflow A condition that is created when a program tries to use more space on the stack than it can actually accommodate. Stack overflow often occurs with recursive functions that don't encounter a terminating condition.

standard error The destination used by the diagnostic output stream to write its output. Includes all error messages generated by operating system commands. The default destination of this stream is the terminal but can be redirected to any file.

standard input The source opened by the shell to accept information as a stream of characters. By default, the keyboard is assigned this source, but it can also come from a file using redirection (<).

standard library A collection of the code of ANSI C functions and their supporting files archived into one or more files. All the commonly used functions like **`printf`** and **`scanf`** have their code and related information in this library.

standard output The destination used by commands to send output to. Used by all commands and programs that send output to the terminal. The default destination can also be reassigned to a disk file using redirection (>).

star topology A type of networking where all nodes are connected to a central hub. The network fails when the hub fails.

statement Term used to refer to an instruction in C. Unlike a preprocessor directive, a statement ends with a semicolon.

static function A function that is inaccessible in other program files. The keyword **`static`** is used in the declaration of a static function.

static variable A variable that is created and initialized only when the function is first called. Its value is preserved between successive calls. The keyword **`static`** is used in the declaration of a static variable.

storage class A variable attribute that determines its scope, lifetime and initial value. A variable must belong to one of four storage classes (automatic, static, external and register).

string An array of characters terminated by the NUL character (`'\0'`). Characters enclosed within double quotes also evaluate to a string. **`scanf`** and **`printf`** use the `%s` format specifier to match a word and complete string, respectively.

stringizing operator The # operator that is used with the **`#define`** directive of the preprocessor to convert a macro argument to a string constant. Solves the problem of replacing parameters inside strings. See also **token pasting operator**.

structure A user-defined data type meant to store heterogeneous data in its members or fields. Each member is accessed using dot and `->` pointer notation. Unlike an array, a structure passed as an argument to a function is copied inside the function.

structured programming A programming paradigm that advocates the use of independent reusable modules for faster development and easier maintainability. Based on the premise that any computing problem can be solved using only sequencing, decision making and repetition constructs.

structure tag A name that represents the data type of the structure. A structure declared without a tag doesn't allow further creation of variables of its type.

switch (hardware) A device that is connected to multiple devices in a network. Unlike a **hub**, a switch is intelligent and broadcasts the data it receives from one node *only* to the one the data is meant for.

switch (software) A token normally beginning with a /, which changes the default behavior of an MSDOS command. The **`DIR /P`** command has `/P` as its switch. Counterpart in Linux is the **option** which uses a `-` instead of /.

symbolic constant A named constant defined and undefined with the **`#define`** and **`#undef`** directives of the preprocessor. Also known as **named constant**, **defined constant** and **manifest constant**.

syntax error An error that occurs when a statement fails to conform to the rules of the language. The compiler doesn't create the executable if it detects a syntax error.

system software A category of software that manages the computer hardware and its peripherals. User programs access the hardware using system software. Includes device drivers and the operating system. See also **application software**.

tab A single character which simulates a contiguous set of spaces. Is generated by hitting a specific key or *[Ctrl-i]*, and forms one of the characters of **whitespace**. Useful for aligning columns.

tail pointer A pointer that points to the last node of a **linked list**. Facilitates list access from its end and is updated when a node is added or deleted at the end.

TCP/IP Expands to Transmission Control Protocol/Internet Protocol—a collection of protocols used for networking computers. Ensures reliable transmission and is used on the Internet.

ternary operator An operator that uses the `?` and `:` symbols along with three expressions *exp1*, *exp2* and *exp3* as operands. The value of the compound expression is either *exp2* or *exp3* depending on the logical value of *exp1*. Often replaces simple **`if-else`** statements.

text file A file that contains printable characters organized as lines and terminated by an OS-dependent newline character. The **`fopen`** function needs to know whether a file is to be opened in binary or text mode. See also **binary file**.

time sharing Term used to refer to the allotment of a small slice of time to each program, which is taken out of operation after it has used the slice. Concept used by multi-user and multi-tasking operating systems.

token A contiguous sequence of non-whitespace characters.

token pasting operator The ## operator that is used by the preprocessor to merge the two tokens on either side of the ## to a single token. See also **stringizing operator**.

top-down programming A programming methodology that advocates understanding the entire system before breaking it up into its individual components. See **bottom-up programming**.

tree A non-linear data structure whose elements are organized as a hierarchy of nodes, where child nodes are connected to their parents using pointers. An ordered binary tree speeds up search operations by halving the search domain in each descent to the next level.

truth table A table that lists the output of a logic gate for all possible combinations of the input. Also used with the same meaning for bitwise operations.

two's complement A widely used form of representation of a negative number, obtained by reversing the bits of its positive counterpart (one's complement) and adding 1 to it. Uses a single zero.

unary operator An operator that has a single operand. Includes the `++`, `--`, `!` and `~` operators.

union A structure-like data type where multiple members share the same memory region. The member that is updated last is also the one that is currently active, but its value can also be read back using another member.

Universal Serial Bus (USB) A port used universally to connect devices like mouse, printer, scanner and smartphone. Has both power and data lines.

unresolved reference An entry made to a table by the compiler when it encounters a function. The **linker** uses this entry to extract the code for the function from a library.

unsigned number A number where the most significant bit (MSB) is not used for the sign. The range of an unsigned number is double that of a signed one.

user-defined data type A data type that is not native to the language but defined by the user. Structures and unions belong to this category.

variable A memory location containing a value which is accessed by name. A variable also has a type that determines how the value is to be interpreted.

variable hiding A situation that arises when an inner block re-declares a variable. The new value then hides the old one which becomes visible when the block is exited.

variable-length array (VLA) A feature introduced by C99 to determine the length of an array at runtime. A VLA can be declared with a variable as its index.

variable lifetime Refers to the times of birth and death of a variable. A variable declared inside a function lives as long as the function is running.

variable scope Refers to the region of the program where a variable is visible and can be accessed. A variable defined inside a function has function scope.

variable visibility Same as **variable scope**.

Video Graphics Adapter (VGA) A port used to connect a display terminal. Supports a resolution of 640×480 pixels.

virus A software designed with malicious intent to cause damage to the computer. Belongs to a category called **malware**.

void pointer Same as **generic pointer**.

whitespace A contiguous sequence of spaces, tabs or newlines used to separate words in a line. The `%s` format specifier used in **`scanf`** matches a string that doesn't include whitespace.

wild card A special character used by the shell to match a group of filenames with a single expression. The wild card, `*`, used in the pattern `*.c` matches all C programs.

word A contiguous string of characters not containing whitespace.

World Wide Web A service on the Internet featuring a collection of linked documents and images that are fetched by a browser using the HTTP protocol.

APPENDIX

D Answers to Objective Questions

CHAPTER 1

A1. True/False Statements

1.1 False **1.2** True **1.3** True **1.4** False **1.5** False **1.6** True **1.7** True
1.8 False **1.9** False **1.10** True **1.11** False **1.12** True **1.13** False **1.14** True
1.15 False **1.16** False **1.17** True **1.18** True **1.19** False

A2. Fill in the Blanks

1.1 Instructions, data **1.2** Program, data **1.3** Volatile **1.4** First **1.5** Terabytes
1.6 Assembly language **1.7** Operating system **1.8** Speed **1.9** Fifth **1.10** Workstation
1.11 Addresses, files **1.12** Cache **1.13** Pits and lands
1.14 Optical character recognition (OCR) **1.15** Laser **1.16** Star **1.17** Internet

A3. Multiple-Choice Questions

1.1 B **1.2** B **1.3** D **1.4** C **1.5** A **1.6** D **1.7** C
1.8 D **1.9** B **1.10** C **1.11** B **1.12** C

A4. Mix and Match

1.1 A3, B5, C2, D1, E4 **1.2** A3, B4, C5, D6, E1, F2
1.3 A4, B3, C1, D2 **1.4** A4, B5, C1, D6, E3, F2
1.5 A5, B6, C2, D1, E3 F4 **1.6** A3, B5, C6, D2, E1, F4

CHAPTER 2

A1. True/False Statements

2.1 True **2.2** False **2.3** True **2.4** True **2.5** False **2.6** False **2.7** True
2.8 False **2.9** False **2.10** True **2.11** False **2.12** False **2.13** True

A2. Fill in the Blanks

2.1 System **2.2** Malware or virus **2.3** Unified Extensible Firmware Interface (UEFI).
2.4 CMOS **2.5** Bootstrapping **2.6** Operating system **2.7** Shell **2.8** Real-time
2.9 System **2.10** External, internal **2.11** `RENAME`, `MOVE` **2.12** Windows Update
2.13 Kernel **2.14** *[Ctrl-c]*, *[Ctrl-v]* **2.15** IP address **2.16** FTP
2.17 HTTP, `.htm`, `.html`

A3. Multiple-Choice Questions

2.1 C **2.2** C **2.3** C **2.4** A **2.5** B **2.6** D **2.7** D
2.8 B **2.9** E **2.10** B **2.11** A **2.12** D **2.13** D **2.14** D **2.15** D

A4. Mix and Match

2.1 A6, B5, C1, D2, E3, F4
2.2 A4, B6, C5, D1, E2, F3
2.3 A3, B4, C6, D5, E1, F2
2.4 A5, B1, C2, D4, E6, F3
2.5 A5, B4, C1, D2, E3
2.6 A4, B5, C2, D1, E3

CHAPTER 3

A1. True/False Statements

3.1 True **3.2** False **3.3** False **3.4** True **3.5** True **3.6** True **3.7** False
3.8 False **3.9** False **3.10** True **3.11** True **3.12** False

A2. Fill in the Blanks

3.1 2 **3.2** 255 **3.3** `0010 0101 0101` **3.4** Magnitude
3.5 7-bit **3.6** 1 **3.7** Procedural **3.8** Top-down
3.9 Boolean, true, false **3.10** Algorithm **3.11** Assembly **3.12** Linker
3.13 Interpreter

A3. Multiple-Choice Questions

3.1 E **3.2** A **3.3** C **3.4** D **3.5** D **3.6** E **3.7** C **3.8** C

A4. Mix and Match

3.1 A3, B4, C6, D5, E1, F2

CHAPTER 4

A1. True/False Statements

4.1 True **4.2** True **4.3** False **4.4** False **4.5** True **4.6** False **4.7** True
4.8 True **4.9** False **4.10** True **4.11** False **4.12** True **4.13** False

A2. Fill in the Blanks

4.1 UNIX **4.2** Compiler, linker **4.3** Directive **4.4** Symbolic constant
4.5 /*, */ **4.6** Syntax, semantic **4.7** `main`, `return` **4.8** Operators
4.9 Boolean **4.10** `.exe` or `.EXE` **4.11** `return`
4.12 `scanf` **4.13** Control string

A3. Multiple-Choice Questions

4.1 D **4.2** A **4.3** C **4.4** E **4.5** C **4.6** B **4.7** C **4.8** D

CHAPTER 5

A1. True/False Statements

5.1 True **5.2** False **5.3** False **5.4** True **5.5** True **5.6** False **5.7** True
5.8 False **5.9** False **5.10** True **5.11** True **5.12** True **5.13** False **5.14** False

A2. Fill in the Blanks

5.1 Reserved **5.2** Declaration, definition **5.3** Constant
5.4 `sizeof` **5.5** 2 and 4 **5.6** `0` and `0X` **5.7** 6 and 10
5.8 `int` **5.9** Escape sequence **5.10** Derived

A3. Multiple-Choice Questions

5.1 C **5.2** C **5.3** D **5.4** A **5.5** D **5.6** B **5.7** B

CHAPTER 6

A1. True/False Statements

6.1 False **6.2** False **6.3** True **6.4** True **6.5** False **6.6** False **6.7** True
6.8 True **6.9** False **6.10** False **6.11** False **6.12** True

A2. Fill in the Blanks

6.1 Unary **6.2** C **6.3** Precedence (or priority), associativity **6.4** Side effect
6.5 `int` **6.6** True, false **6.7** `k++`, `k = k + 1` **6.8** `==` **6.9** Ternary
6.10 20

A3. Multiple-Choice Questions

6.1 C **6.2** D **6.3** B **6.4** C **6.5** D **6.6** A **6.7** B
6.8 D **6.9** D **6.10** A

A4. Mix and Match

6.1 A2, B5, C4, D1, E3

CHAPTER 7

A1. True/False Statements

7.1 True **7.2** False **7.3** True **7.4** False **7.5** False **7.6** True
7.7 True **7.8** False **7.9** True **7.10** False

A2. Fill in the Blanks

7.1 Control expression **7.2** Curly braces **7.3** Null **7.4** `break`
7.5 Integer **7.6** `goto` **7.7** Ternary, `?` and `:`

A3. Multiple-Choice Questions

7.1 A **7.2** C **7.3** C **7.4** B **7.5** C (not a logical operator) **7.6** A

CHAPTER 8

A1. True/False Statements

8.1 True **8.2** False **8.3** True **8.4** False **8.5** False **8.6** True **8.7** False
8.8 True

A2. Fill in the Blanks

8.1 `break`, `continue` **8.2** Entry-controlled, exit-controlled **8.3** Infinite
8.4 Integer **8.5** `return`

A3. Multiple-Choice Questions

8.1 A (not a loop) **8.2** D **8.3** D **8.4** A **8.5** C **8.6** B

CHAPTER 9

A1. True/False Statements

9.1 True **9.2** True **9.3** False **9.4** False **9.5** True **9.6** False **9.7** True
9.8 True

A2. Fill in the Blanks

9.1 `-1`, `stdio.h` **9.2** Buffer **9.3** Archive **9.4** Smaller
9.5 Variable **9.6** Files, open **9.7** `ungetc` **9.8** `scanf`, `line`
9.9 Word, string

A3. Multiple-Choice Questions

9.1 A **9.2** C **9.3** B **9.4** D **9.5** D
9.6 A (not meant for an integer) **9.7** B **9.8** B

A4. Mix and Match

9.1 A3, B1, C2 **9.2** A4, B3, C5, D2, E1

CHAPTER 10

A1. True/False Statements

10.1 True **10.2** False **10.3** False **10.4** True **10.5** False **10.6** False **10.7** True
10.8 True

A2. Fill in the Blanks

10.1 Contiguously **10.2** Subscript or index **10.3** Operator **10.4** Binary
10.5 Array **10.6** Rows, columns **10.7** 200 **10.8** `a[2][9]`
10.9 Runtime

A3. Multiple-Choice Questions

10.1 A **10.2** C **10.3** C **10.4** D **10.5** B **10.6** A

CHAPTER 11

A1. True/False Statements

11.1 False **11.2** True **11.3** False **11.4** True **11.5** False **11.6** False **11.7** True
11.8 False **11.9** True **11.10** True **11.11** True **11.12** False

A2. Fill in the Blanks

11.1 Declaration (or prototype or signature), definition **11.2** `void`
11.3 Recursive **11.4** Declaration **11.5** Copied **11.6** Stack
11.7 Static **11.8** `return`, `exit` **11.9** Declared
11.10 Declared, defined **11.11** `auto` or automatic **11.12** `register`

A3. Multiple-Choice Questions

11.1 D **11.2** D **11.3** A **11.4** B **11.5** C **11.6** D

CHAPTER 12

A1. True/False Statements

12.1 False **12.2** True **12.3** True **12.4** False **12.5** True **12.6** False **12.7** True
12.8 False **12.9** True

A2. Fill in the Blanks

12.1 &, * **12.2** Address **12.3** Null **12.4** Pointers
12.5 Local **12.6** Contiguous **12.7** `void`

A3. Multiple-Choice Questions

12.1 D **12.2** C **12.3** A **12.4** C **12.5** B **12.6** C

CHAPTER 13

A1. True/False Statements

13.1 False **13.2** False **13.3** True **13.4** False **13.5** True **13.6** False
13.7 True **13.8** True **13.9** False **13.10** True **13.11** True

A2. Fill in the Blanks

13.1 NUL **13.2** `char *`, `char` **13.3** `gets`
13.4 `sscanf`, `sprintf` **13.5** `string.h` **13.6** `isalpha`, `isdigit` **13.7** `main`

A3. Multiple-Choice Questions

13.1 E **13.2** E **13.3** D **13.4** A **13.5** E **13.6** C

A4. Mix and Match

13.1 A3, B5, C4, D1, E2

CHAPTER 14

A1. True/False Statements

14.1 True **14.2** False **14.3** False **14.4** True **14.5** False **14.6** True **14.7** True
14.8 False **14.9** False **14.10** True **14.11** False **14.12** True **14.13** False **14.14** False
14.15 True

A2. Fill in the Blanks

14.1 Heterogeneous **14.2** User-defined **14.3** `typedef` **14.4** Self-referential
14.5 Union **14.6** Array **14.7** Largest, size **14.8** Bit, field
14.9 `enum`

A3. Multiple-Choice Questions

14.1 A **14.2** B **14.3** B **14.4** D **14.5** A **14.6** C

CHAPTER 15

A1. True/False Statements

15.1 False **15.2** True **15.3** False **15.4** False **15.5** True **15.6** True **15.7** False
15.8 True **15.9** False

A2. Fill in the Blanks

15.1 Operating system **15.2** Text, newline **15.3** `stdlib.h` **15.4** `w+`
15.5 `FILE` **15.6** `fscanf` **15.7** `errno`, `perror` **15.8** `feof`
15.9 `fwrite`

A3. Multiple-Choice Questions

15.1 D **15.2** C **15.3** A **15.4** E **15.5** A **15.6** D **15.7** B

A4. Mix and Match

15.1 A2, B1, C4, D3

CHAPTER 16

A1. True/False Statements

16.1 False **16.2** False **16.3** False **16.4** True **16.5** False **16.6** True **16.7** False
16.8 False **16.9** False **16.10** True **16.11** False **16.12** False **16.13** True **16.14** True
16.15 False

A2. Fill in the Blanks

16.1 `malloc`, `calloc` **16.2** Generic **16.3** Cast **16.4** `free`
16.5 Dangling pointer **16.6** `malloc`, `calloc` **16.7** `calloc` **16.8** Nodes
16.9 NULL **16.10** Abstract

A3. Multiple-Choice Questions

16.1 B **16.2** D **16.3** E **16.4** E **16.5** D **16.6** D

A4. Mix and Match

16.1 A3, B1, C2 **16.2** A2, B3, C1

CHAPTER 17

A1. True/False Statements

17.1 False **17.2** True **17.3** False **17.4** False **17.5** True **17.6** True **17.7** False
17.8 False **17.9** True **17.10** True **17.11** False **17.12** False **17.13** True **17.14** True

A2. Fill in the Blanks

17.1 `#` **17.2** Syntax **17.3** `#`, `##` **17.4** `\`
17.5 double, quotes, `#include` **17.6** `defined`, `!defined`. **17.7** Bits
17.8 Mask **17.9** Cleared, cleared **17.10** 8
17.11 Function pointer **17.12** Callback **17.13** Variable

A3. Multiple-Choice Questions

17.1 D **17.2** A **17.3** C **17.4** C **17.5** A **17.6** D **17.7** B
17.8 C **17.9** A **17.10** B **17.11** C

A4. Mix and Match

17.1 A4, B2, C1, D5, E3

APPENDIX

E Bibliography

It's a problem of plenty when it comes to selecting books for learning C. To be honest, I have discovered some useful features in *every* resource listed here. Some of these books could be out of print, but you should be able to locate their PDF versions on Google Books:

Computer Fundamentals and Programming in C, Second Edition by Reema Thareja (Oxford University Press), 2016, New Delhi

Computer Fundamentals and Programming in C, Second Edition by Pradip Dey and Manas Ghosh (Oxford University Press), 2013, New Delhi

Let Us C, Fifteenth Edition by Yashavant Kanetkar (BPB Publications), 2016, New Delhi

Programming in ANSI C, Seventh Edition by E. Balagurusamy (McGraw-Hill India), 2017, New Delhi

Computer Fundamentals & C Programming, Second Edition by E. Balagurusamy (McGraw-Hill India), 2017, New Delhi

The C programming Language, Second Edition by Brian Kernighan and Dennis Ritchie (Pearson), 1988, New Delhi

A Book on C, Fourth Edition by Al Kelly and IRA Pohl (Pearson), 1998, New Delhi

C Primer Plus, Third Edition by Stephen Prata (Techmedia/SAMS), 1999, New Delhi

Programming in ANSI C, Revised Edition by Stephen G. Kochan (SAMS Publishing), 1994, USA

Computer Science: A Structured Programming Approach Using C, Second Edition by Behrouz A. Forouzan and Richard F. Gilberg (Brooks/Cole), 2001, USA

Practical C Programming, Third Edition by Steve Oualline (O'Reilly), 1997, Mumbai

C How To Program, Seventh Edition by Deitel & Deitel (Pearson), 2013, New Delhi

On-line Resources

The GNU C Programming Tutorial (http://crasseux.com/books/ctutorial/) A desert-island choice that you simply can't do without.

The GNU C Reference Manual (http://www.gnu.com) A detailed comprehensive reference, but make sure that you ignore the non-standard GNU extensions.

comp.lang.c Frequently Asked Questions by Steve Summit *(http://c-faq.com)* Another must-have document that you'll find quite helpful.

Essential C by Nick Parlante (Stanford CS Education Library: *http://cslibrary.standford.edu/101/EssentialC.pdf)* A brilliant exposition of the basic concepts supplemented with several diagrams.

Index

E

F

T

U

V

W