PART I

Introduction

Chapter 1

Computers and Systems

1.1 The Physical Computer

It is not necessary to understand how a computer works in order to use one—but it helps. An elementary understanding of computer architecture helps demystify the nature and rules of programming languages and enables one to use these languages more wisely. It is essential for anyone who needs to attach devices to a computer or buy one wisely. The architecture of modern computers varies greatly from type to type, and new developments happen every year. Therefore, it is impossible to describe how all computers work. The following discussion is intended to give a general idea of the elements common or universal to personal computers and workstations today.

The main logical parts of a computer, diagrammed in Figure 1.1, can be roughly compared to parts of a human body:

- The **CPU** (central processing unit) is the brain of the computer.
- The computer's **RAM** (random access memory) chips are its memory.
- The **bus** is the nervous system; it carries information between the CPU and everything else in the computer.
- The **input** devices (e.g., keyboard) are the computer's senses.
- The **output** devices (e.g., monitor) are the computer's effectors (hands, voice).

1.1.1 The Processor

In a typical modern computer, the central processing unit (CPU) is the main element on the processor chip. The CPU controls and coordinates the whole machine. It contains a set of **registers**:

- The instruction register (IR) holds the current machine instruction.
- The instruction counter (IC) holds the address of the next machine instruction.
- The memory data register (MDR) holds the data currently in use.
- The memory address register (MAR) holds the address from which the data came.

One of its components is the **clock**, which ticks at a fixed rate and controls the fundamental speed at which all of the computer's operations work. The clock rate on a microprocessor chip is set as fast as is (conservatively) possible without causing processing errors. This setting is the megahertz (MHz) rating published by the manufacturer.

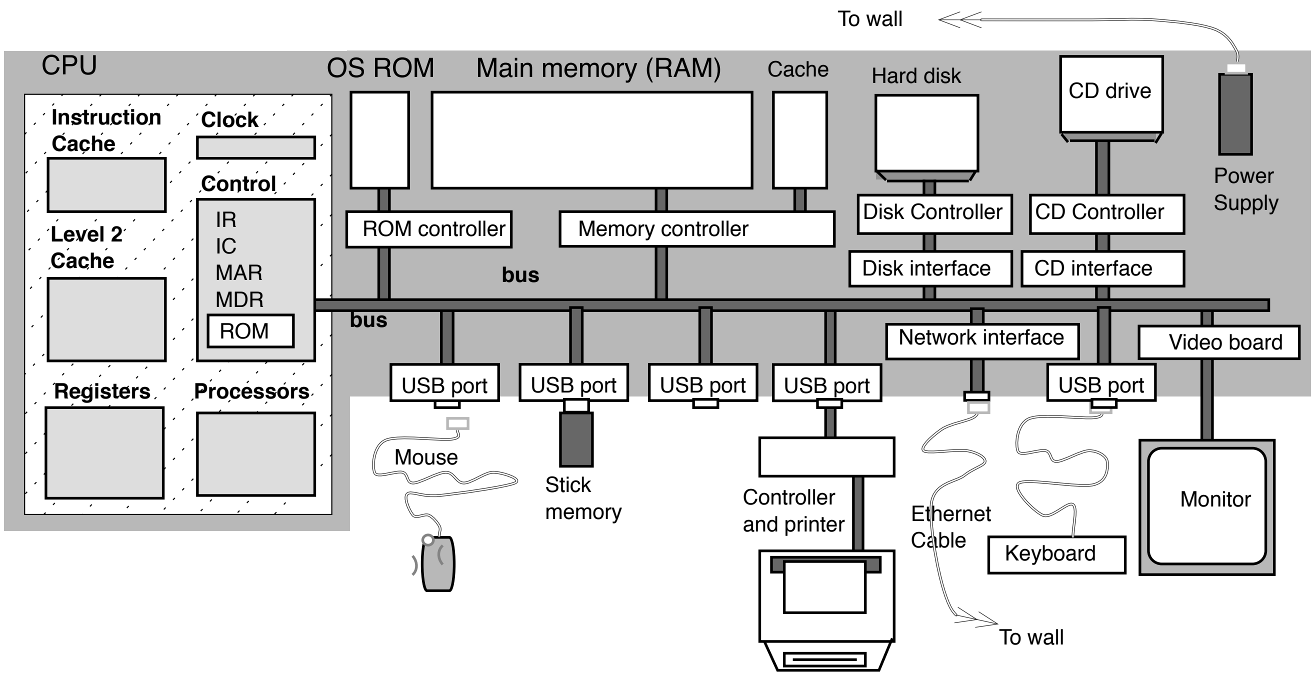


Figure 1.1. Basic architecture of a modern computer.

The ALU. The *arithmetic and logic unit* (ALU) is the part of the processor containing the many circuits that actually perform computations. Typically, an ALU includes instructions for addition, negation, and multiplication of integers; comparison; logical operations; and other actions. Many computers also have a **floating-point coprocessor**, for handling arithmetic operations on real (floating-point) numbers. Floating-point instructions are important for many scientific applications to achieve adequate accuracy at an acceptable speed. Taken together, this set of instructions forms the **machine language** for that particular processor.

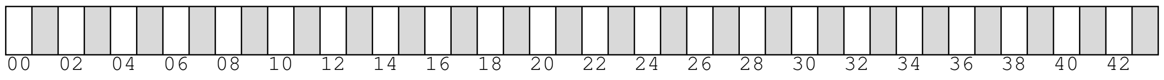
Control ROM and the instruction cycle. A small read-only memory inside the control unit contains instructions (called *microcode*) that control all parts of the CPU and define the actions of the instruction cycle, the ALU, and the instruction cache (discussed next).

To use a computer, we write a **program**, which is a series of instructions in some computer language. Before the program can be used or run, those instructions must be translated into machine language and the machine-language program must be loaded into the computer's main memory. Then, one at a time, the program's instructions are brought into the processor, decoded, and executed. A processor executes the program instructions in sequence until it comes to a "jump" instruction, which causes it to start executing the instructions in another part of the program. A typical instruction brings data into one of the registers, sends data out to the memory or to an output device, or executes some computation on the data in the registers.

1.1.2 The Memory

The memory of a computer consists of a very large number of storage locations called **bits**, which is short for "binary digits." Each bit can be turned either off (to store a 0) or on (to store a 1). All

- Memory is a very long series of bytes; we show only the first few here.
- Every byte has an address (only the even addresses are shown here).
- In this picture, even-address bytes are white; odd-address bytes are gray.
- We diagram the addresses outside the boxes because they are part of the hardware. They are *not* stored in memory.
- The addresses in the diagram begin with the address 0 and continue through 43.



This amount of memory could store

- 44 characters (one byte each), or
- 22 short integers (two bytes each), or
- 11 long integers or single precision floating-point numbers (four bytes each).

Figure 1.2. Main memory in a byte-addressable machine.

memory in a computer is made out of bits or groups of bits, and all computation is done on groups of bits. The bits in memory are organized into a series of locations, each with an address. In personal computers, each addressable location is eight bits long, called a **byte**. In larger computers and older computers, the smallest addressable unit often is larger than this; in a few machines, it is smaller. Figure 1.2 is a diagram of main memory, depicted as a sequence of boxes with addresses.

A byte could contain data of various types. It could contain one character, such as *'A'*, or a small number. The range of numbers that can be stored in one byte is from 0 to 255 or from −128 to +127, which is not large enough for most purposes. For this reason, bytes generally are grouped into longer units. Two bytes are long enough to contain any integer between 0 and 65,536, while four bytes can hold an integer as large as 2 billion.[1]

Traditionally, a **word** is the unit of data that can pass across the bus to or from main memory at one time.[2] Small computers usually have two-byte words; workstations and larger computers have words that are four bytes or longer.

A computer might have several different types of **memory** to achieve different balances among capacity, cost, speed, and convenience. The major types are cache memory, main memory, secondary memory, and auxiliary memory. These are diagrammed in Figure 1.3 and discussed next.

[1]Number representation is covered in Chapters 7 and 15.

[2]In machines where the bus transports only one byte, *word* means two bytes and *long word* means four bytes.

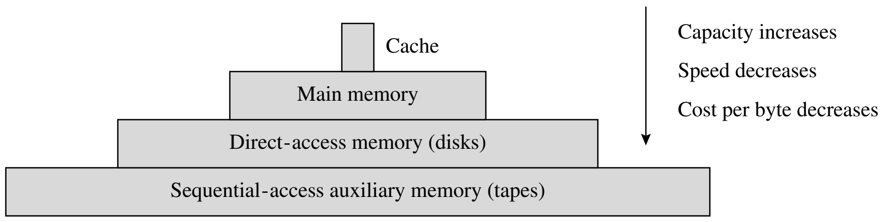


Figure 1.3. The memory hierarchy.

Cache memory. The fastest and most expensive kind of memory is a **cache**. Some machines have small caches to speed up access to frequently used data that are stored in the main memory. The first time an item is used, it is loaded into the cache. If it is used again soon, it is retrieved from the cache rather than from the main memory, reducing the access time. Cache memories are small because they are very expensive. Their small capacity limits the extent to which they can improve performance.

Main memory. The computer's main memory (sometimes called *RAM*, for random access memory) is where the active program (or programs) is kept with its data. Sometimes the operating system, which controls the computer,[3] also is kept in RAM; otherwise it is in read-only memory.

OS-ROM memory. **ROM** stands for "read-only memory." An operating system (OS) in ROM is a real convenience for the user for two reasons. First, it is installed in the computer and need not be brought in from a disk. This saves time when the system is turned on. Second, ROM is read-only memory, which means its contents can be read but not changed, either accidentally or on purpose. A partly debugged program sometimes can "run wild" and try to store things in memory locations allocated to some other process that is simultaneously loaded into the computer. With the operating system in ROM, most of the system is protected from this kind of random modification. The only remaining vulnerable part of the system is the area near memory address 0, which is called **low memory** and contains the locations used to communicate between the operating system and the input/output devices.

The disadvantage of using ROM for an operating system is that it is difficult and expensive to improve the system or correct errors in it. When a company wishes to release a new version of a ROM operating system, the code must be recorded on a set of ROM chips. The computer owner must buy a set, remove the old ROM chips, and install the new ones.

Direct-access memory. Direct-access memory, also called *secondary memory*, is needed because even the largest main memory cannot store all the information we need. Data files and software packages are kept in secondary memory when not in use.[4]

A **CD-ROM** (compact disk, read-only memory) is an optical disk storage device, like an audio disc except that it is used to store various types of data, not just music data. A CD-ROM reader contains a laser that reads the minute marks etched into the surface of the disk. Once a CD-ROM has been used to record data, it cannot be reused to record different data. Large collections of data of interest to many people are recorded and distributed on CD-ROMs.

Hard disks and diskettes are the most common direct-access memory devices today. Through the years, the physical size of these devices has decreased steadily, and the amount of information they can hold has greatly increased. As technology has progressed, we have been able to store the bits closer and closer together, enabling us to simultaneously decrease size and increase capacity.

Auxiliary memory. Today's disks can store large volumes of information: 2 gigabytes (2 billion bytes) now is a common disk capacity. However, most businesses and individuals find that 2 billion bytes of secondary memory is not enough to meet all of their needs. Larger, cheaper memory devices are needed to store backup (duplicate) copies of the files on the hard disk and infrequently used files. Magnetic tapes and tape drives meet this need for auxiliary storage. A tape has very large capacity but must be read or written sequentially. This makes retrieving a file from a tape that contains hundreds of files or recording a new file on the end of the tape very slow and inconvenient.

[3] Section 1.2 covers operating systems.

[4] As memory capacities have expanded, our desire to store information has kept pace, so that this statement is as true today as it was when direct-access memories were the size of today's main memories.

1.1.3 Input and Output Devices

Input and output devices permit communication between human beings and the CPU. The most common input devices include keyboards, mice, and track balls. Typically, output is displayed either on a video screen or via a printer. Since humans and the computer speak different languages, some translation is needed. Using a mouse allows the human to point or click at a portion of the screen to convey a screen position and an intent to the computer. Interactions with a keyboard, the screen, or a printer take place using the English language and some hardware-level translation.

In a simple view of the computer world, hitting the Z key on a keyboard causes a *Z* to go into the computer, after which that *Z* can be sent to the video screen or a printer and reappear as a *Z*. While this usually is true, it is very far from a direct or necessary connection.

Consider the keyboard. When you type the key in the lower left (usually marked with a *Z*), the information that goes into the keyboard controller is the coordinates of that key, not a *Z*. Somewhere a code table says "bottom row, first key means *Z*." When the same keyboard is used in Germany, the key in that position is marked with a *Y*, and the code table says "bottom row, first key means *Y*." Similarly, changing the wheel on a daisy-wheel printer changes the letter that prints on the paper.

Translation codes are arbitrary, and several codes are in common use. The most common character code for personal computers is named ASCII (American Standard Code for Information Interchange), a seven-bit code that supports upper- and lowercase alphabets, numerals, punctuation, and control characters. Each device has its own set of codes, but the codes built into one device are not necessarily the same as the codes built into the next device. Some characters, especially characters like tabs, formfeeds, and carriage returns, are handled differently by different devices and even by different pieces of system software running on the same device. For example, the character whose ASCII code is 12 is named *formfeed* in the ASCII code table. The idea of the code makers was that a printer would eject the paper when it received this code. The program in each printer's controller was supposed to look for this code and handle it, and most did. Today, also, some video controllers are programmed to respond to a formfeed character in an analogous way, by clearing the screen.

As a computer user, you need to be aware that equipment not designed to be used together might be incompatible in unexpected ways. You may find computer systems in which the label on the key that you type, the letter that shows on the screen, and the one that comes out of the printer are all different. This can happen because all three depend on software interpretation as well as hardware capabilities.

1.1.4 The Bus

The bus is the pathway between the processor and everything else. It consists of two sets of wires: one set has enough wires to transmit an address; the other set transmits data. When the processor needs a data item, it puts the address of the item in the memory address register (MAR) and issues a `fetch` command. The memory address register and the memory data register (MDR) sit at the end of the bus line (see Figure 1.1).

When a `fetch` command is given, the address goes from the MAR out over the bus's address lines and a copy of the required data comes back over the bus's data lines to the MDR. Similarly, to store information into the memory, the information and the target address are put into these two registers and a `store` command is given. The information goes out over the bus to the given address and replaces whatever used to be there.

Control of peripherals and interfaces. The input and output devices are attached to the bus lines. Each device has its own address and handles the information in its own way. Each device, in fact, has a different set of instruction codes that it can handle and a controller to carry out those instructions. A **device controller** is a small processor connected between the bus and the device that is used to control the action of the device. For example, consider a hard disk. A disk has a controller that understands how to get addresses, disk instructions, and information off the bus and how to put information and signals back on the bus. It knows how to carry out and oversee all the disk operations.

Making all the highly varied devices respond to instructions in a uniform way is the job of **device drivers**. A device driver consists of software that knows about the specific quirks and capabilities of a specific device. It translates the uniform system commands into a form that the device can handle prior to putting the commands on the bus lines. A different driver may be needed for every combination of operating system and hardware. For example, a UNIX driver for a SCSI[5] disk would translate the UNIX `read-disk` command to the SCSI format. The user becomes aware that device drivers exist when he or she wants to install a new device and must also install the appropriate driver.

Between a device's driver and its controller is an interface, a doorway between the bus and the device. There are many kinds of interfaces, with varied transmission properties, which can be classified into two general groups: serial and parallel. A **serial interface** transmits and receives bits one at a time. A **parallel interface** can transmit or receive a byte (or more) at a time, in parallel, over several wires. Parallel interfaces commonly are used for printers; serial interfaces for modems, certain printers, mice, and other slow-speed devices. A MIDI (musical instrument digital interface) is a serial interface used to communicate with electronic musical instruments such as synthesizers and keyboards.

1.1.5 Networks

Inside the computer, the various hardware components communicate with each other using the internal bus. It now is common practice to have computers communicate with each other to share resources and information. This is made possible through the use of networks, physical wires (often phone lines) along which electrical transmissions can occur. The extent of these networks is varied. A **local area network** typically joins together tens of computers in a lab or throughout a small company. A **global network**, such as the Internet, spans much greater distances and connects hundreds of thousands of machines but is truly just a joining together of the smaller networks via a set of gateway computers. A **gateway** computer is a bridge between a network such as the Internet on one side and a local network on the other side. This computer also often acts as a **firewall**, whose purpose is to keep illegal, unwanted, or dangerous transmissions out of the local environment.

Sharing resources. One use of networks is to let several computers share resources such as file systems, printers, and tape drives. The computers in such a network usually are connected in a **server-client** relationship, as illustrated in Figure 1.4. The server possesses the resource that is being shared. The clients, connected via a **hub** or **switched ethernet connection**, share the use of these resources. The user of a client machine may print out documents or access files as if the devices actually were physically connected to the local machine. This can provide the illusion of greater resources than actually exist, as well as present a uniform programming environment, independent of the actual machine used. This kind of sharing is less practical over larger networks due to delays caused by data transmissions through gateway machines.

Communication. The other typical use of networks is communication. E-mail has become a popular way to send letters and short notes to friends and business associates. Chat rooms provide the opportunity for more direct, interactive communication. The World Wide Web makes a wealth of information
available to the average user at home that used to be available only in distant libraries. It also provides new commercial opportunities; people now can shop for many items on the Web and are able to buy specialty items or get bargains that were previously unavailable to them.

Networks also have changed the workplace and work habits. Many professionals use network transmissions between home and office or between two office locations to gain access to essential information when they need it, wherever they are. This may include using a laptop computer that is connected to the network via a cellular phone.

[5]The small computer systems interface (SCSI) is a standard disk interface.

This is one way that a lab network might be set up. It is not drawn to scale; a hub is a small device that could fit on the corner of a table.

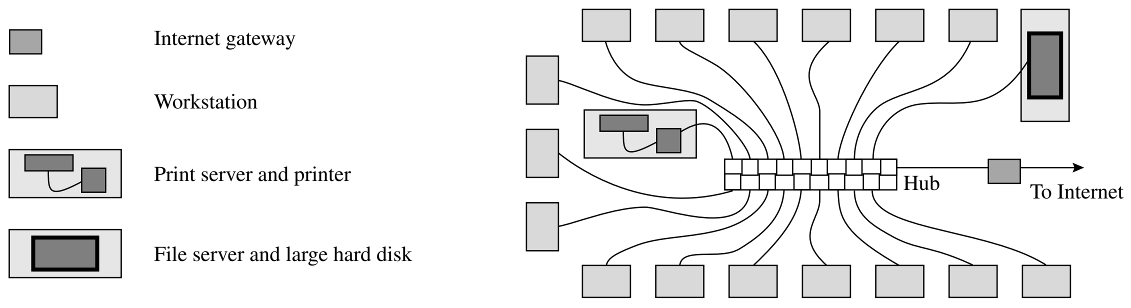


Figure 1.4. A local network in a student lab.

Distributed computing. Networks also are used to allow the computers to communicate between themselves. The complexity of many of today's problems requires the use of reserve computing power. This can be achieved by synchronizing the efforts of multiple computers, all working in parallel on separate components of a problem. A large distributed system may make use of thousands of computers. Synchronization and routing of information in such systems are major tasks, among the many performed by the operating system's software, as discussed next.

1.2 The Operating System

The **operating system** (OS) is the most important piece of system software and the first one you see when you turn on your machine. It is the master control program that enables you to use the hardware and communicate with the rest of the system software. The operating system has several major components, including the **system kernel**, which is the central control component; a **memory management system**, which allocates an area of memory for each program that is running; the **file system manager**, which organizes and controls use of the disks; **device drivers**, which control the hardware devices attached to the computer; and the **system libraries**, which contain all sorts of useful utility programs that can be called by user programs. In addition, a multiprocessing system (described later) has a process scheduler, which keeps track of programs waiting to be run and determines when and how long to run each one.

Command shells and windows. Operating systems can be divided into two categories: command-line interpreters and windowing systems. **Command shells**, or command-line interpreters, were invented first and can be used on small, simple machines. A command shell displays a system prompt at the left of the screen and waits for the user to type in a command. Then, the command is executed by calling up some piece of system or user software. When that program terminates, control goes back to the operating system and the system prompt again is displayed.

A newer idea is a **windowing system**. Window-based systems include the Apple Macintosh system, Microsoft Windows (which is an extension of DOS), and NextStep and X-Windows, which provide window interfaces for UNIX. Except for the Macintosh, these windowing systems can run side-by-side with a command shell and provide access to it. This is important because command-line interpreters generally provide capabilities that are not available within the window environment.

In a window environment, multiple windows can be displayed on the screen, including perhaps a command window. Windows can be used to display file directories, run programs, and so forth. The user accesses the contents of the windows using a mouse or some other point-and-click device. Seeing your files, moving and copying them, renaming them, and every other thing that you do is much easier in a window environment.

Multiprogramming systems. Another way to categorize systems is by whether they can run several programs at the same time or are limited to one at a time. Ordinary personal computers are limited to one process at a time. However, modern personal workstations and large computers, often called *mainframes*, have **multiprogramming** operating systems. UNIX is one of the best known and most widely used multiprogramming systems.

Workstations are capable of running a few processes concurrently, and mainframes often can support 50 or 100 users, running the programs in a **time-shared** manner. In time sharing, each user process is given a short slice of CPU time, then it waits while all the other users get their turns. This works because users spend much more time thinking and typing than running their programs. Any request by a program for input or output (I/O) also ends a time slice; the OS initiates the input or output, then selects another process to be run while the I/O happens. The process scheduler is the system component that coordinates and directs all this complex activity.

Each kind of computer must have its own custom-tailored operating system. Some systems are proprietary and have been implemented for only one manufacturer's models. For example, Apple's system for the Macintosh is jealously guarded against copying. Other systems, such as UNIX and DOS, are widely implemented or imitated. Increasingly, the computer owner has a choice about what system will be installed on his or her hardware.

The choice of hardware is important because it determines what software you can run and what diskettes you can read. Software and file systems are constructed to be compatible with a particular system environment, and they do not work with the wrong system. For example, you cannot read a UNIX diskette in a DOS system, and a C **compiler** that works under UNIX must be modified to work under DOS. Windowing systems and multiprogramming are powerful aides to program development. However, both consume large amounts of main memory, disk space, and processing time. Trying to run them with a machine that is not big enough or fast enough is a mistake.

1.3 Languages

The purpose of a computer language is to allow a human being to communicate with a computer. Human language and machine language are vastly different because human capabilities and machine capabilities are different. Each kind of computer has its own machine language that reflects the particular capabilities of that machine. Computer languages allow human beings to write instructions in a language that is more appropriate for human capabilities and can be translated into the machine language of many different kinds of machines.

1.3.1 Machine Language

Built into the CPU of each computer is a set of instructions that the hardware knows how to execute. The behavior of each instruction and its binary code are documented in the hardware manual. Technically, it is possible to program a computer by making lists of these codes. That is how programming was done 45 years ago.

A machine language program is a sequence of instructions, each of which consists of an instruction code, often followed by one or two register codes or memory addresses. In a machine, these are all represented in binary.[6] In the early days of computers, when people still wrote programs in machine language, they did not write them in binary because people were (and are) abominably bad at writing long strings of 1's and 0's without making errors. Instead, they used the octal number system,[7] in which information is represented as strings of digits between 0 and 7. Each octal digit translates directly into

[6]Binary is the base 2 number system. Information is represented as strings of bits (see Chapter 15).

[7]Octal is base 8.

three binary bits. Thus, a machine language program was written as a long series of lines, where each line was a string of octal digits.

1.3.2 Assembly Languages

When people had to use machine language, it took a very long time to write and debug a program. The next development was symbolic assembly language. Instead of writing octal codes, the programmer wrote symbolic codes for instructions and defined a name for each data-storage location. The three lines that follow show how a simple action might look when expressed in assembly language; this code adds two numbers and stores them in a variable named `sum`. The same addition expressed in C would be `sum = n1 + n2;`

```
ldreg *n1, d1   / Load first number into register d1.
add   *n2, d1   / Add second number to the register.
sto   d1, sum   / Store result in the location for sum.
```

A translator, called an **assembler**, analyzed the symbolic codes and assembled machine-code instructions by translating each symbol into its code and assigning memory locations for the data objects used by the program.

Every name and quoted string used in a program must be stored at some address in the computer's memory. To write in machine language, you manually assign an address to each object. Happily, assembly languages and high-level languages such as C free you from concern over these addresses. The programmer declares the names to be used at the top and defines each name by giving it a data type or quoted string value. When the assembler translates this into machine language, it assigns memory locations for these objects.

Assembly languages still are very important for writing programs so closely related to the hardware that high-level languages like C simply have no commands to express them. Many large systems are written primarily in a high-level language but contain some parts coded in assembly language. These portions are part of the system kernel. They work directly with parts of the machine and must operate as efficiently as possible.

1.3.3 High-Level Languages

Programming in an assembly language is very tedious. Furthermore, an assembly language is specific to one type of machine and probably very different from assembly languages for machines of other manufacturers. Therefore, assembly language programs are not portable—that is, they are not easily converted for use on other machines. In contrast, programs in languages like FORTRAN and C are highly portable, because compilers for these languages have been created for nearly every kind of computer.

Over the past 40 years many high-level languages have been developed, some of which have stood the test of time and some of which have not. Each instruction in a high-level language translates into several at the assembly or machine language level. Programs written in a high-level language appear much more like English and are more understandable to humans. In this section, we discuss a few widely used programming languages.

The C language. C is a relatively old[8] language that recently became very popular. It has characteristics of both high-level languages such as Pascal[9] and FORTRAN[10] and low-level languages such as assembly language. You still might see several dialects of C in older programs or books. In 1988, however, the American National Standards Institute (ANSI) adopted a standard for the language, known as ANSI C. This standard was adopted with a few minor changes by the International Standards Organization (ISO) in 1990 and amended in 1994. This new standard is known as ISO C. The phrase

[8]Created originally in 1972, the language has been updated and expanded several times.

[9]Pascal has been used most extensively as an instructional language in universities.

[10]Short for "formula translator," this language was developed primarily for doing scientific calculations.

standard C can be applied correctly to either version of the standard. The changeover to the standard language is nearly complete, so the beginning C student can safely focus all efforts on standard ISO C.

There are several reasons for the recent growth in the use of C:

- As the UNIX operating system has spread, so has C. C is the tool by which much of the power of UNIX is accessed. UNIX has spread because of the very large amount of valuable software that runs under it.
- From the beginning, C was a very powerful language and fun for the experienced programmer to use. However, it lacked certain important kinds of compile-time error checking and therefore was quite difficult for a beginner to use. ISO C incorporates important new error-checking features and has eliminated many hardware dependencies. It developed into a much better language and became suitable for both beginners and experienced programmers.
- C allows large programs to be written in separate modules. This makes it easier to manage large projects, greatly facilitates debugging, and makes it possible to reuse program modules that do common, useful jobs.
- The ISO C library is extensive and standardized. It contains functions for mathematical computation, input and output facilities, and various system utilities.

On the other hand, ISO C remains more error prone than languages of a similar age with similar features, such as Pascal and Ada.[11] It is popular among experienced programmers partly because of the features that cause this error-prone nature:

- C allows the programmer to write terse, compact code that can run very efficiently. Any programmer who is a slow typist appreciates this. Sometimes programmers even make a game of squeezing the unnecessary operations out of their code. On the negative side, compact code can be hard to read and understand unless comments are used liberally to explain it.
- The error-checking system in ISO C is less rigid and more permissive than in competing languages. Expert programmers claim that this permissiveness is an advantage and that the rigid mechanisms in other languages often "get in the way" when they want to do something unusual. However, these rigid systems are easier for the new programmer to understand and to use.
- C supports bit-manipulation operators that can select or change a single bit or group of bits in a number. These are very important in system programs that must interface to hardware devices that set and test values in specific memory locations. However, working with arbitrary machine addresses and bit patterns must be done with extreme care to avoid errors.
- No restrictions are placed on the use of pointers. (A **pointer** is a variable representing the location, as opposed to the value, of data.) This permits the use of some very efficient computational methods, at the potential cost of destroying information anywhere in memory when an error is made in setting a pointer value. Unfortunately, such errors are common, and many result in system crashes and the need to reboot the system.

C and FORTRAN. FORTRAN is a very old language that has been used by engineers and scientists since the infancy of computers. Originally, it was a language for scientific computation, and it still serves that purpose. Over the years, the language has been updated, revised, and expanded, but its primary focus remains high-performance numerical computation. A massive amount of scientific programming now exists in the form of FORTRAN libraries and FORTRAN application programs that are used, and shared, by scientists worldwide. The FORTRAN libraries are extremely efficient, reliable, and trusted.

Because many engineering departments are acquiring UNIX workstations, C is beginning to supplant FORTRAN-77 in many engineering applications. This has some advantages. FORTRAN-77 still bears

[11]Ada is a programming language developed in the 1970s to support large-scale, portable application systems.

the burden of being an old language. It is full of unnecessary complications and nonuniform conventions. Much of the space in a FORTRAN textbook is spent explaining how to *write* the language correctly. In contrast, a C textbook has a much simpler language to present and can spend more time explaining how to *use* the language well.

On the other hand, FORTRAN-77 is a "safer" language. A program can get into trouble in very few ways that will cause a system crash or cause the result of a seemingly correct expression to be nonsense. C is prone to these problems, even when the programmer avoids using the advanced parts of the language. When a C programmer begins to use pointers, debugging becomes substantially more difficult than it ever could be in FORTRAN-77. Nonetheless, C is here, and thousands of former FORTRAN programmers are beginning to use it. FORTRAN-to-C conversion programs exist and are being used to make the transition less costly.

C and C++. The C++ language extends C to eliminate more causes of error and provide software-engineering tools that are important for large projects. Also, C++ (but not C) is fully compatible with the FORTRAN libraries. This can be a very important consideration for a department switching from FORTRAN to C. C++ is a superset of C. The ordinary line-by-line code in a C++ program is written in C. The extensions involve the way code is organized into modules and the way these modules are used. The C++ extensions are a powerful tool for program organization and error prevention. However, since the entire C language is included as a subset of C++, any error that you can make in C also can be made in C++. The advantages of C++ are there only for those who know how to use them. For beginners, C++ is a more difficult and confusing language than C.

The differences between C and C++ become significant only for moderate to large programs, and only when C++ is used with proper object-oriented design techniques. All software-engineering techniques presented in this book are appropriate for use with both C and C++. The way in which C language elements are presented will lead toward an understanding of the design requirements for C++.

1.4 What You Should Remember

1.4.1 Major Concepts

- This chapter provides a brief description of computer hardware and software. It describes the parts of the machine a programmer must know to comprehend the operation of a program or buy a personal computer system wisely.
- Computer languages and the process of translation are discussed, and the C language is compared to FORTRAN and C++.

1.4.2 Vocabulary

The terms and concepts that follow have been introduced and described briefly. The first and second columns contain terms related to computer hardware and operating systems; the third column relates to the programming process.

CPU	device controller	program
register	device driver	machine language
memory	serial interface	operating system
cache	parallel interface	system kernel
ROM	local area network	system libraries
CD-ROM	global network	command shell
RAM	gateway	windowing system
bit	firewall	multiprogramming
byte	server	assembler
word	client	compiler
clock	memory management system	ANSI C
bus	file system manager	ISO C
hub	floating-point coprocessor	C++

1.5 Using Pencil and Paper

1.5.1 Self-Test Exercises

1. Which terms on the vocabulary list relate to the computer's processor?
2. Which terms on the vocabulary list relate to the memory of a computer?
3. Which terms on the vocabulary list relate to the peripherals of a computer?
4. Which terms on the vocabulary list relate to a computer network?
5. Which terms on the vocabulary list relate to system software?
6. For what does each of the following abbreviations stand?

a.	ALU	f.	ANSI	k.	LAN
b.	bit	g.	I/O	l.	MIDI
c.	CPU	h.	OS	m.	SCSI
d.	ISO	i.	ROM	n.	MHz
e.	ASCII	j.	WAN	o.	RAM

1.5.2 Using Pencil and Paper

1. Choose three terms from *each column* of the vocabulary list in Section 1.4.2. In your own words, give a brief definition for each (a total of nine definitions).
2. What computer will you use for the programming exercises in this course? What kind of processor chip does it have? How big is its main memory? What input and output devices are available for it? Is it attached to a computer network?
3. Have you used a local area network? Why? Have you used the Internet? For what purposes?
4. What operating system runs on the computer that you will use for the programming exercises in this course? Is this a multiprogramming system? What compiler will you use?
5. Explain the difference between

   (a) a byte and a word.
   (b) ROM and RAM.
   (c) cache memory and main memory.
   (d) a device controller and a device driver.
   (e) a compiler and an assembler.
   (f) a command shell and a windowing system.
   (g) a LAN and a WAN.
   (h) a gateway and a hub.

Chapter 2

Programs and Programming

2.1 What Is a Program?

A program is a set of instructions for a computer. Programs receive data, carry out the instructions, and produce useful results. More precisely, the computer **executes a program** by performing its instructions, one at a time, on the supplied data. The instructions a computer is capable of carrying out are very primitive—add two numbers, move a number from here to there in memory, and so forth. Large numbers of instructions are required to carry out even the simplest task, so some programs are thousands or even millions of instructions long.

Nowadays, programs are so big and complicated that they cannot be constructed without the help of the computer itself. A C **compiler** is a computer program whose purpose is to take a desired program coded in a programming language and generate the low-level instructions for the computer from that code. A C compiler is usually used through a program development system called an *integrated development environment*, or *IDE*, that includes an editor and provides windows for viewing code, results, and other relevant information.

The process of submitting a code file to a compiler for translation is called *compilation*. The programmer's code is called **source code** (or C *code* in this case). And the compiled program is called **object code** or **machine code**. One often blurs the distinction between source code and object code by using the word *program* to refer to either. Ideally, the intended meaning will always be clear from the context.

The C *programming language* is the notation used for writing C code. The purpose of this book is to help you learn how to write programs using the C programming language and to use a C compiler and an IDE to generate the instructions that will allow the computer to carry out the actions specified by your program.

Algorithms. Many programs are based on algorithms. An **algorithm** is a method for solving a well-structured problem, much as a recipe is a step-by-step process for cooking a particular dish. The method must be completely defined, unambiguous, and effective; that is, it must be capable of being carried out in a mechanical way. An algorithm must terminate; it cannot go on forever.[1] As long as these criteria are met, the algorithm may be specified in English, graphically, or by any other form of communication. Figure 2.1 is an example of a very simple algorithm specified in English.

Thousands of years ago, mathematicians and scientists invented algorithms to solve important problems such as computing areas of polygons and multiplying integers. More recently, algorithms have been developed for solving engineering, mathematical, and scientific problems such as summing a series of numbers, integrating functions, and computing trajectories. A major area of computer science focuses on the creation, analysis, and improvement of algorithms. Computer scientists have invented new algorithms for computing mathematical functions and organizing, sorting, and searching collections of data. The growing power of computer software is due largely to recently invented algorithms that solve problems better and faster.

[1]Some programs, generally called *systems programs*, are not based on algorithms and are designed to go on forever. Their purpose is to help operate the computer.

```
To find the average of a set of numbers do the following:
    Count the numbers in the set.
    Add all of them together to get their sum.
    Divide the sum by the count.
```

Figure 2.1. A simple algorithm: Find an average.

Algorithms are sometimes explained using a mixture of English and computer language called **pseudocode** (because it looks like computer code, but is not). Figures 2.3, 2.5, and 2.7 show how the simple English statement of the average algorithm from Figure 2.1 progressively is developed into a program specification (Figure 2.3), then into pseudocode (Figure 2.5) and finally into a C program (Figure 2.7).

Algorithms are also sometimes written in C or in a similar high-level language, or as graphical representations (known as **flowcharts**) of the sequence of calculations and decisions that form the algorithm. These forms (rather than pseudocode or English) are normally used when algorithms are published in textbooks or journals.

2.1.1 The Simplest Program

A very simple, yet complete, program is shown in Figure 2.2. All it does is print a greeting on the computer's screen. Even though this program is very short, it illustrates several important facts about C programs:

- The first line in this example is called a comment. It is ignored by the C compiler and is written only to provide information for human readers.
- The second line is a preprocessor command. It instructs the C compiler to bring in the declarations of the standard input and output functions so that this program can perform an output operation.
- Every C program must have a function named `main()`. Execution begins at the first statement in `main()` and continues until the `return` statement at the end. The statements are surrounded by a pair of braces.
- This program is a sequence of two statements. The first is a call on the `puts()` function to display a phrase on the computer's screen (`puts()` is the name of one of the standard output functions).
- The statement `return 0;` is always written at the end of a program; it returns control to the computer's operating system after the program's work is done.
- If you enter this program into your own C system, compile it, and run it, you should see the following message on your computer's screen:

  ```
  Hail and farewell, friend!
  ```

```c
// This simple program is where we begin.
#include <stdio.h>
int main( void )
{
    puts( "Hail and farewell, friend!" );
    return 0;
}
```

Figure 2.2. A simple program: Hail and farewell.

2.2 The stages of program development.

Some programs are short and can be written in a few minutes. However, most programs are larger and more complex than that, and often created by many people working for many months. Whether a program is small or large, the steps in creating it are the same:

1. **Define the problem.** The programmer must know the purpose and requirements of the program he or she is supposed to create.
2. **Design a testing process.** The programmer must figure out how to test the program and how to determine whether or not it is correct.
3. **Design a solution.** The programmer must plan a sequence of steps that starts with entering the input and ends by supplying the information that is required.
4. **Program construction.** After planning comes coding. The programmer codes each step in a computer language such as C, and enters the C code into a computer file using a text editor.
5. **Program translation.** The file of C code is processed by the C compiler. If there are no errors, the result is executable code.
6. **Verification.** The program is run, or executed, on data values from the test plan. The program normally displays results and the programmer must verify that the results are what was expected.

In reality, many of these steps are repeated in a cycle until all errors are eliminated from the program. In the rest of this chapter, each of these steps will be described in more detail.

2.2.1 Define the Problem

Most programs are written to solve problems, but you don't start by writing program code or even by planning a solution. Before you can begin to solve a problem, you must understand exactly the nature of the problem. You need to know what data you will start with (if any) and what answers you are supposed to produce.

Suppose a teacher asked a helper to write a small program to compute the average of a student's test scores. The helper might go through these stages in defining the problem:

First version of specification.
Goal: Compute a student's exam average.
Input: A list of exam scores.
Output: The average of the scores.

We also must decide how the output will be presented. First, all output should be labeled clearly with a word or phrase that explains its meaning. Also, it is a very good idea to "echo" every input as part of the output. This lets the user confirm that the input was typed and interpreted correctly and makes it clear which answer belongs with each input set. Spacing should be used to make the answers easy to find and easy to read.

Define constants and formulas. Some applications require the use of one or more formulas and, possibly, some numeric constants. This information is part of a complete specification, so we add the formula and update the output description:

Second version of specification.
Goal: Compute a student's exam average.
Input: A list of exam scores.
Output: Echo the inputs and display the average of the scores.
Formula: average = (sum of all scores) / (number of scores)

Goal:	Compute a student's exam average.
Input:	The user will enter exam scores interactively.
Output:	Echo the inputs and print their average.
Constant:	There are 3 exam scores.
Formula:	$Average = (score1 + score2 + score3)/3.0$
Other requirements:	The answer must be accurate to at least one decimal place.

Figure 2.3. Problem specification: Find the average exam score.

At this stage, the programmer should realize that more information is needed and go back to the teacher to find out how many exams the class had, and whether the program should do anything about strange scores like -1 or 1000. Assume that the teacher's answers are 3 exams, and (for now) don't worry about ridiculous scores. We now have a complete problem description, which is shown in Figure 2.3.

2.2.2 Design a Test Plan

Verification is an essential step in program development. When a program is first compiled and run, it generally contains errors due to mistakes in planning, unexpected circumstances, or simple carelessness. Although program testing and verification take place after a program is written and compiled, a **program verification** plan should be created much earlier in the development process, as soon as the program specifications are complete. Designing a test plan at this stage helps clarify the programmer's thoughts. It often uncovers special cases that must be handled and helps create an easily verified design for the solution.

A **test plan** consists of a list of input data sets and the correct program result for each, which often must be computed by hand. This list should test all the normal and abnormal conditions that the program could encounter. To develop a test plan, we start with the problem definition. The first few data sets (if possible) should have answers that are easy to compute in one's head. These test items often can be created by looking at the formulas and constants that are part of the problem definition and choosing values for the variables that make the calculations easy. Figure 2.3 shows a test plan for the exam average problem.

Next, all special cases should be listed. The test designer identifies these by looking at the computational requirements and limitations given in the problem definition. Extreme data values at or just beyond the specified limits should be added to the plan. The next step is to analyze what could go wrong with the program and enter data sets that would be likely to cause failure. For example, some data values might result in a division by 0.

The last data sets should contain data that might be entered during normal use of the program. Since the answers to such problems generally are harder to compute by hand, we list only one or two of them. The results for these items allow the programmer to refine the appearance and readability of the output.

After the coding step is complete, the programmer should return once more to the test plan. If necessary, more test cases should be added to ensure that every line of code in the program is tested at least once.

These guidelines have been stated in general terms so that they are applicable to a wide range of program testing situations. Because of this, the guidelines are somewhat vague and abstract. More examples of test

Case	**Score1**	**Score2**	**Score3**	**Answer**
Easy to compute	10	20	30	20
Special cases	100	100	100	100
	0	0	0	0
Typical cases	66	5	21	30.67
	90	82	89	87

Figure 2.4. Test plan for exam average program.

```
1. Declare names for real numbers $n1$, $n2$, $n3$, and $average$.
2. Display output titles that identify the program.
3. Instruct the user to enter three numbers.
4. Read $n1$, $n2$, and $n3$.
5. Add $n1$, $n2$, and $n3$ and divide the sum by 3.0. Store the result in $average$.
6. Display the three numbers we just read so that the user can verify that they were entered and read
   correctly.
7. Display the $average$.
8. Return to the system.
```

Figure 2.5. Pseudocode program sketch: Find an average.

plans are given on the website for this chapter and as part of the case studies in later chapters of this text. These examples will help you gain understanding of verification techniques and learn to build test plans for your own programs.

2.2.3 Design a Solution

We are now ready to begin writing the program. We must design a series of steps to go from the data given to the desired result. The steps of most simple programs follow this pattern:

- Display a title that identifies the program.
- Read in the data needed.
- Calculate the answer.
- Print the answer.

Following this guide and the specification, we write down a series of steps that will solve the problem. Many programmers like to write a *pseudocode* version first, which is a combination of English and programming terms. Figure 2.5 gives a pseudocode version of the exam average program.

2.3 The Development Environment

A program **development environment** consists of a set of system programs that enable a programmer to create, translate, and maintain programs. Many modern commercially available compilers are part of an *integrated development environment*, which includes an editor, compiler, linker, and run-time support system including a symbolic debugger. These system programs are used through a menu-driven shell. The stages of program creation are illustrated in Figure 2.6 and described next.

2.3.1 The Text Editor

The **text editor** is a tool used to enter a program into a computer file. Text editors permit a typist to enter lines of code or data into a file and make changes and rearrange parts easily. They lack the font and style commands and control over page layout included in a word processor because these "bells and whistles" are neither useful nor desirable in a program file. If you were to use a word processor and enter your program into a normal word-processor document file, the compiler would be unable to translate the program because of the embedded formatting commands.

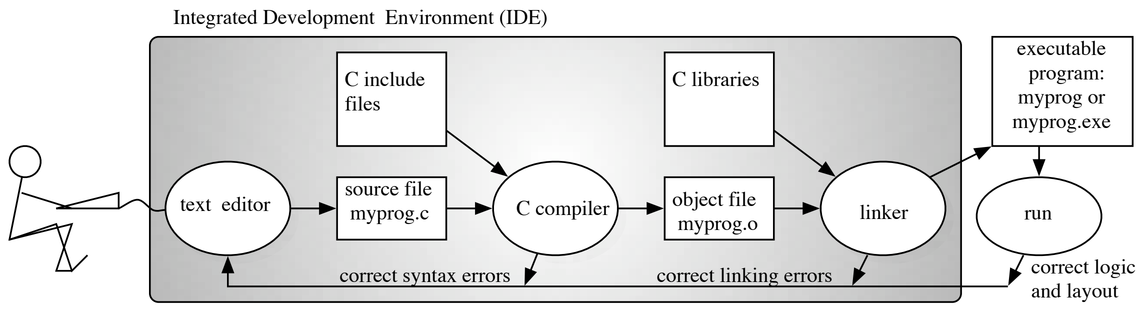


Figure 2.6. The stages of program translation.

When you enter a text editor to create a new program or data file, its screen will show a blank document. To type a program, just start at the top and type the lines of code. This is your source code. The first time you save the program file, you must give it a name that ends in *.c*. Be sure to save your file periodically and maintain a backup copy. When the program is complete, save it again. Data files are created similarly, except that they are given names ending in *.in* or *.dat*.

Text editors come in three general varieties. The very old editors, like the UNIX ed editor, are line editors. They operate on one line of a file at a time, by line number, and are slow and awkward to use. Somewhat newer are the full-screen editors, such as UNIX vi and older versions of emacs, which allow the programmer to move a cursor around the video screen and delete, insert, or change the text under the cursor. They also can delete, copy, or move large blocks of code. Some make periodic, automatic backup copies of your file, which is a tremendous help when the power unexpectedly goes off. To use one of these editors, the programmer enters a command into the operating system interpreter that contains the name of the program to be executed, the names of the files to be used, and sometimes other information regarding software options.

Modern text editors (including the new versions of emacs permit all the necessary text editing operations to be done using a mouse and a text window with scroll bars. They often display the text in color, using different colors for C's reserved words, user-defined words, and comments.

Use the most modern text editor you can find. Scroll bars and mice make a huge difference in the ease of use. A good editor encourages you to write good programs. You will be much more willing to make needed changes and reorganize program parts if it is not much work to do so.

2.3.2 The Translator

We use programming languages to communicate with computers. However, computers cannot understand our programming languages directly; they must be translated first. When we write source code in a symbolic computer language, a translator is used to analyze the code, follow the commands, find the declarations, assign memory addresses for the data objects, and determine the structure and meaning of each statement.

There are two kinds of translators: compilers and interpreters. After analyzing the source code, a compiler generates machine instructions that will carry out the meaning of the program at a later time. At **compile time**, it plans what memory locations will be needed and creates an object file that contains machine-language instructions to allocate this memory and carry out the actions of the program. Later, the object code is linked with the library, loaded into memory, and run. At **run time**, memory is allocated and information is stored in it. Then, the machine instructions are executed using these data.

An interpreter performs the translation and execution as a single step. After determining the meaning of each command, the interpreter carries it out. Some languages, such as BASIC and Java, normally are interpreted. Others, such as FORTRAN and C, normally are compiled. In general, compiled code is more efficient than interpreted code.

2.3.3 Linkers

An object file is not executable code. Before you can run your program, the linker (another system program) must be called to link it with the precompiled code in the C system libraries. Although the two processes *can* be done separately, linking is normally done automatically when compilation is successful. The result of linking is a **load module** or **executable file**; that is, a file that is ready to load into the computer's memory and execute. The programmer should give this file, which will be used to run the program, a convenient and meaningful name. In some systems, the name automatically will be the same as the name of the source file except that it ends in *.exe*. In UNIX, however, the name of the load module is *a.out* unless the user provides a better name as part of the compile command.

2.4 Source Code Construction

You have specified the whole project and designed an algorithm to do the computation. Now you are ready to write the actual code for your program. In the old days, when computers were scarce and expensive, programmers wrote out their code, in detail, on paper so that everything was complete before approaching the computer. This ensured a minimum amount of computer time was consumed. Today, computer time is no longer scarce, so experienced programmers often skip the paper-copy phase; they approach the computer with a detailed sketch of the code but without actually writing it in longhand. Others, especially beginners, still prefer to write the entire program on paper before sitting down at the computer. This permits them to separate the process of writing the code from the difficulties of dealing with a computer and a text editor. It also is helpful in getting "the big picture"; that is, seeing how all the parts of the program fit together. However, whether you are a beginner or an expert, it is important to have at least a well-developed and well-specified program sketch when you begin entering code into the computer. Programming without one is difficult and failure prone, because without a good plan, it is hard to know what to do and what order to do it in.

The steps in code construction are:

1. Create a project and a source file in a proper subdirectory.
2. Write or copy C code into the source file.
3. Build (compile, then link) the C code.
4. Correct compilation and linker errors and rebuild.
5. Run and test the executable program.

This text presents two methods for coding a program: Start from scratch or adapt an existing program to meet the new specification. The first several chapters rely primarily on adapting the programs given as examples in the text. In Chapter 9, after the fundamental concepts of programming have been covered, we discuss the process of building a program from scratch, using a specification.

Here, we continue with the exam average problem. An algorithm was given in Figure 2.1; a specification is given in Figure 2.3 a test plan in Figure 2.4 and a pseudocode program sketch in Figure 2.5. Now we are ready to write C code.

2.4.1 Create a Source File

To begin, you should create a new subdirectory (folder) on your disk for use only with this program. By the time a program is finished, it will have several component files (source code, backup, error file, object code, executable version, data, and output). Your life will be much simpler if you keep all the parts of a program together and separate them from all other programs.

Once you have made a directory for the new project, you are ready to create a new source code file in that directory. This can be done by copying and renaming an existing program you intend to modify or by starting with a blank document and storing it under your new program name. Then either type in your

```
#include <stdio.h>
int main( void )
{
    double n1, n2, n3;      // The three input numbers.
    double average;         // The average of the three numbers.

    puts( "Compute the average of 3 numbers" );
    puts( "--------------------------------" );
    printf( "Please input 3 numbers: " );
    scanf( "%lg%lg%lg", &n1, &n2, &n3 );   // Read the numbers.

    average = (n1 + n2 + n3) / 3.0;        // Average is sum / 3.
    printf( "The average of  %g, %g, and %g = %g \n\n",
            n1, n2, n3, average );         // Print answers.
    return 0;
}
```

Sample output:

```
Compute the average of 3 numbers
--------------------------------
Please input 3 numbers: 21 5 66
The average of  21, 5, and 66 = 30.6667
```

Figure 2.7. A C program.

new source code or modify the existing code. When you are finished, print out a copy of your file and look carefully at what you have done. (If your printer is inconvenient or produces output of poor quality, you may find it easier to examine the code on the video screen.) Mark any errors you find and use the text editor to correct them.

2.4.2 Coding.

With a firm idea of what the program must do, we can proceed to writing the steps in the C language. Figure 2.7 gives the C code that corresponds to the pseudocode in Figure 2.5. The notes, below, give a quick guide to the meaning of the C statements. Do not try to understand the C code at this time; all will be explained more fully in Chapter 3. This goal in this chapter is only to understand the process of program development, coding, translation, and execution.

1. Like all programs, this one starts with a command to include the declarations from the C input/output libary.
2. Next is the line that declares the beginning of the main program. That is followed by a pair of curly braces that enclose the body of the program (declarations and instructions).
3. The first two lines of the program body are declarations that create and name space to store four numbers, $n1$, $n2$, $n3$, and *average*.
4. The words enclosed in `/* ... */` on the right end of these lines are comments. They are not part of the C code, but are written to help the reader understand the code.
5. The next two lines print the program title and a line of dashes to make the title look nice. Compare the code to the output, at the bottom of the Figure.
6. The `printf()` statement Instructs the user to enter three numbers at the keyboard. This kind of instruction is called a *user prompt* or, simply, a *prompt*.
7. The `scanf()` statement reads three numbers from the keyboard and stores them in $n1$, $n2$, and $n3$. This ends the input phase of the program, and we leave a blank line in the code.

8. Calculations in C are written much as they are in mathematics. The next line calculates the average and stores it for future use.
9. The next two lines are a `printf()` statement that displays the three input numbers and the average. The first line of this statement has a quoted string called a *format* that shows how we want the answers to be laid out. Each `%g` marks a spot where a number should be inserted. The second line lists the numbers that we want to insert in those spots.
10. The last line of code returns control to the system.

2.4.3 Translate the Program and Correct Errors

Before you can run your program, it must be compiled (translated into machine language) and linked with the previously compiled library code modules.

Compiling. When we use the compiler, we tell it the name of a C source file to translate. It runs and produces various kinds of feedback. If the job was done perfectly, the compiler will display some indication of success and create a new object file, which contains the machine code instructions that correspond to the C code in the source file. However, it is rare that all the planning, design, coding, and typing are correct on the first try. Human beings are prone to making mistakes and compilers are completely intolerant of errors, even very small ones. Spelling, grammar, and punctuation must be perfect or the program cannot be translated. The result is that the compiler will stop in the middle of a translation with an error comment (or a list of them). The programmer must locate the error, fix it, and use the compiler again.

Correcting compile-time errors. When we first entered the average program, we made typographical errors in two lines near the end. The code typed in looked like this:

```
average = (n1 + n2 + n3) / 3.0
print( "The average of  %g, %g, and %g = %g \n\n",
        n1, n2, n3, average );
```

The compiler responded with these error comments:

```
Compiling mean.c (2 errors)
    mean.c:18: illegal statement, missing ‘;’ after ‘3.0’
    mean.c:19: parse error before "print"
```

The problem is very clear: the line that computes the average needs a semicolon. This was corrected and the code was recompilied.

Interpreting error comments. Error comments differ from system to system. Language designers try to make these comments clear and helpful. However, especially with C, identifying the source of the error is sometimes guesswork, and the comments are often obscure or even misleading. For example, the comment `undefined symbol` might indicate a misspelling of a symbol that is defined, and `illegal symbol line 90` might mean that the semicolon at the end of line 89 is missing. Worse yet, a missing punctuation mark might not be detected for many lines, and an extra one might not be detected at all—the program just will not work right.

If there is a list of error comments, do not try to remember them all. Transfer the error comments to Notepad or to a new file and either print it or use a part of your screen to display it[2]. Then fix the first several problems. Very often, one small error near the beginning of a program can cause the compiler to become confused and produce dozens of false error comments. After fixing a small number of errors, recompile and get a fresh error listing. Often, many of the errors will just go away.

It takes some practice to be able to interpret the meaning of error comments. Sometimes beginners interpret them literally and change the wrong thing. When this happens, a nearly correct program can get

[2]In some systems, using the print-screen button is convenient

This diagram shows how the constant values used in the program of Figure 2.7 might be stored in main memory. Later chapters will explain why.

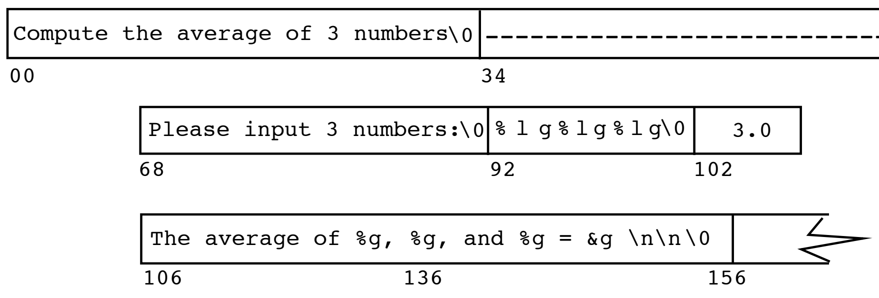

Compute the average of 3 numbers\0
---------------------------------\0
00
34
67
Please input 3 numbers:\0
% l g % l g % l g\0
3.0
68
92
102
The average of %g, %g, and %g = &g \n\n\0
106
136
156

Figure 2.8. Memory for constants.

worse and worse until it becomes really hard to fix. Anyone who is not sure of the meaning of an error comment should seek advice from a more experienced person.

Correcting linking errors. The linking process also may fail and generate an error comment. When a **linking error** happens, it usually means that the name of something in the library has been misspelled or that the linker cannot find the system libraries. The programmer should check the calls on the specified library function for correctness. If the compiler is newly installed, a knowledgable person should check that it was installed correctly. When we tried to link our program, the linker produced this error comment:

After fixing the semicolon error in our program, the compiler finished successfully, but the linker did not:

```
Compiling mean.c (1 warning)
    mean.c:19: warning: implicit declaration of function ‘print’
Linking /mean/build/mean (1 error, 1 warning)
    ld: warning prebinding disabled because of undefined symbols
    ld: Undefined symbols:
_print
```

The linker could not find a function named `print()` in the standard library. I changed it to `printf()` This time, both compilation and linking were successful and we were ready to run the program.

2.5 Program Execution and Testing

2.5.1 Execute the Program

When a program has been successfully compiled and linked, the testing phase begins. An integrated development environment usually provides a icon or a RUN button to run the program; if so, just click on it. If not, you can start execution by clicking the mouse on the program's icon or typing the program's name on a command line. The operating system will respond by loading the executable program into the computer's memory. The memory will then contain the program instructions, any constants or quoted strings in the program, and space for the program's data. Figure 2.8 shows how the constants and data might be laid out in the memory for the program in Figure 2.7.

When loading is complete, the operating system transfers control to the first line of the program. At this time, you should begin to see the output produced by your program. If it needs input, type the input and hit the Enter key. When your program is finished, control will return to the system. When the program's instructions are executed, the computer's devices perform input and output and its logic circuits perform calculations. These actions produce data and results that are stored in the memory. In an integrated development environment, the system screen may disappear while your program is running and appear again when it is finished.

A program can be tested by running it directly (left) or through an on-line debugger (right).

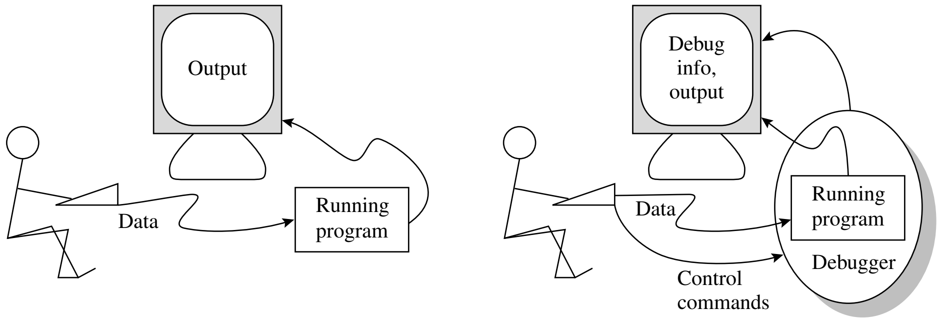


Figure 2.9. Testing a program.

When a program is running, the contents of the memory cells change every time a `read` or `store` instruction is executed. In some systems, a program can be run with a debugger. A **debugger** is another program that runs the program for you. As shown in Figure 2.9, you communicate with the debugger, and it interprets your program instructions one at a time, step by step. You can ask to see the results of computations and the contents of selected parts of memory after each step and, thus, monitor how the process is proceeding. Figure 2.10 shows, step by step, how the values stored in the memory change when the program in Figure 2.7 is executed.

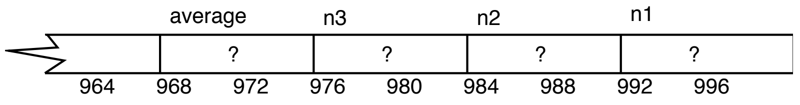
A debugger can be a powerful tool for figuring out why and how an error occurs. Even though the C language is fully standardized, C debuggers are not. Each one is different. Because of this, it is helpful to learn other ways to monitor the progress of a program. Such techniques are presented in the next section.

2.5.2 Test and Verify

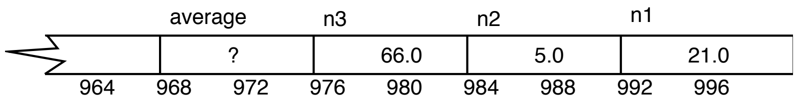
Many errors are caught by the compiler, some by the linker. When a program finally does compile and link without warnings, though, the process of finding the **logical errors** begins. The primary method for

This diagram shows how the values stored in main memory change when we run the program from Figure 2.7. A `?` indicates that any garbage value might exist at that location.

When the program is ready to run, the memory area for variables might look like this:



After returning from `scanf()`, with inputs of 21, 5, and 66,



After computing the formula and storing the answer,

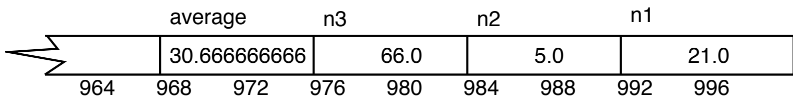


Figure 2.10. Memory during program execution.

detecting logical errors is to test the program on varied input data and look carefully at the answers it prints. They might or might not be correct. Sometimes a program operates correctly on some data sets and incorrectly on others. Sometimes an unusual data condition will cause a program to crash or begin running forever. In any case, *you must verify the correctness of your answers.*

You now are ready to use your testing plan. Run your program and enter each of the data sets in your plan, verifying each time that the computer got the same answer you got by hand calculation. Compare the correct answer on your test plan to the answer the computer has printed. Are they the same? We tested our program using the plan in Figure 2.4; the output from our fourth test is shown here:

```
Compute the average of 3 numbers
--------------------------------
Please input 3 numbers: 21 5 66
The average of  21, 5, and 66 = 30.6667
```

This answer (and the others) matched the expected answers, so we have some confidence that the program is correct. If the answer is not correct, why not? Where is the error—in the test plan or in the program code? If the test plan has the correct answer, you must analyze the code and find the reason for the error; that is, you must find the logical error, better known as a *program bug*. The useful debugging method of using printouts is described next. Another technique (parse trees) is given in Section 4.8.

Debugging: locating and correcting errors. As you start writing your own programs, you must learn how to find and eliminate programming errors. Debugging printouts are a powerful technique for locating a logical error in a program. When your program computes nonsense answers, add debugging printouts until you discover the first point at which the intermediate results are wrong. This shows you where the problem is. With that information (and possibly some expert help), you can deduce why the code is wrong and how to fix it. In an integrated development environment, the debugger can be used to provide similar information.

Beginners tend to form wrong theories about the causes of errors and change things that were right. Sometimes this process continues for hours until a program that originally was close to correct becomes a random mess of damaging patches. The best defense against this is to be sure you understand the reason for the error before you change anything. If you do not understand it, ask for help. Be sure to save a backup copy of your program when you modify it. This can be helpful if you need to undo a "correction."

Finishing the test. Whether you find an error or a needed improvement, the next step is to figure out how to fix it and start the edit–compile–link–test process all over again. Eventually the output for your first test case will be correct and acceptably readable. Then the whole testing cycle must begin again with the other items on the test plan: Enter a data set, inspect the results critically, and fix any problems that become evident. When the last test item produces correct, readable answers, you are done.

Polishing the presentation. Now that we are sure the answers are correct, we look at the output with a critical eye. Is it clear and easy to read? Should it be improved? Is everything spelled correctly? Does the spacing make sense? Small changes sometimes make big improvements in readability.

2.6 What You Should Remember

2.6.1 Major Concepts

Facts about programs.

1. **Translation.** Programs are written in a computer language and converted to machine code by a translator, called a *compiler*.

2. **Syntax.** Programs must conform exactly to the rules for spelling, grammar, and punctuation prescribed by the language. Not every piece of source code describes a valid program. To be valid, every word used in the source code must be defined and the words must be arranged according to strict rules.
3. **Errors.** An invalid program can contain three kinds of errors: compilation errors, linking errors, and run-time errors. Invalid program code results in **compilation errors** or **linking errors** when translated. A runtime error happens when the machine malfunctions or the program tries do do an illegal operation, such as dividing by zero.
4. **Semantics.** A program, as a whole, has a *meaning*. The meaning of a C program is its effect when it is translated and executed.[3] Ideally, the meaning of a program is what the programmer intended. If not, we say that the program contains a bug, or **logic error**.

The stages of program development.

1. Problem definition.
2. Design of a testing process.
3. Design of a solution.
4. Program construction.
5. Program translation.
6. Verification.

2.6.2 The moral of this story.

If you wish to avoid grief in your programming, you should take to heart these proverbs:

1. It is harder than it seems it ought to be.
2. The program itself is like the tip of an iceberg: It rests on top of a lot of invisible work and it will not float without that work.
3. Attention to detail pays off. Compilers expect perfection.
4. Debugging seldom is an easy process, and if a program is not well-structured, it can become a nightmare.
5. Every development step skipped is hours wasted. (Beginners rarely understand this.)
6. You cannot start a program the day before it is due and finish it on time.

2.6.3 Vocabulary

These are the most important terms and concepts presented or discussed in this chapter.

program	development environment	compile time
algorithm	text editor	compilation error
pseudocode	compiler	linking error
test plan	linker	execute a program
source code	debugger	run time
object code	debugging printouts	run-time error
executable file	statement	logical error
load module	declaration	program verification

[3]In some ways, the meaning of a C program depends on the hardware on which it is executed. Although it always will produce the same results when executed on the same machine with the same data, the results might be different when executed on a different kind of computer.

2.7 Exercises

2.7.1 Self-Test Exercises

1. Use a dictionary to find the meaning and pronounciation of the word "pseudo". Use that definition to explain the meaning of "pseudocode".
2. Explain why a programmer must use a text editor, not a word processor.
3. Explain why a beginner usually fails to finish a program on time when he or she starts it the day before it is due.
4. True or false: A program is ready to turn in when it compiles, runs, and produces results. Please explain your answer.
5. True or false: When a program finishes running, it returns control to the operating system. Please explain your answer.
6. Explain the difference between
   (a) an algorithm and a program.
   (b) a command and a declaration.
   (c) a compiler and an interpreter.
   (d) source code and object code
   (e) an object code file and a load module.

2.7.2 Using Pencil and Paper

1. A test plan is a list of data inputs and corresponding outputs that can be used to test a program. This list should start with input values that are easy to check in your head, then include input values that are special cases and values that might cause trouble, perhaps by being too big or too small.
   (a) Following the example in Figure 2.3, create a problem specification for a program to convert Fahrenheit temperatures to Celsius using the formula

   $$Celsius = (Fahrenheit - 32.0) * \frac{5.0}{9.0}$$

   Specify the following aspects: scope, input or inputs required, output required, formulas, constants, computational requirements, and limits (if any are necessary) on the range of legal inputs.
   (b) Write a test plan for this algorithm. Include inputs that can be checked in your head, those that will test any limits or special cases for this problem, and typical input values. For each input, list the correct output.
   (c) Using English or pseudocode, write an algorithm for this problem. Attempt to verify the correctness of your algorithm using the test plan. (Do not attempt to actually write the program in C.)
2. Explain the difference between
   (a) pseudocode and source code.
   (b) a text editor and a word processor.
   (c) compile time and run time.
   (d) program verification and program execution.
3. Given the short C program that follows,
   (a) Make a list of the memory variables in this program.
   (b) Which lines of code contain operations that change the contents of memory? What are those operations?
   (c) Which lines of code cause things to be displayed on the computer's screen?

```
int main( void )
{
    double square;
    double number;

    printf( "Enter a number:" );
    scanf( "%lg", &number );
    square = number * number;
    printf( "The square of %g is %g\n", number, square );
    return 0;
}
```

2.7.3 Using the Computer

1. Decide which computer system you will use for these exercises. If it is a shared system, find out how to create an account for yourself.

   (a) Create an account and a personal directory for your work.
   (b) Find out how to create a subdirectory on your system. Create one called `info`.
   (c) You will use a text editor to type in your programs and data files. Some C systems have a built-in text editor; others do not. Find out what text editor you will be using and how to access it. Create a text file (not a program) containing your name, address, and telephone number on separate lines. Next, write the brand of computer you are using and the name of the text editor. Then write a paragraph that describes your past experience with computers. Save this file in your `info` directory and email it to your teacher.
   (d) Find out how to print a file on your system. Print out and turn in the file you created in (c).

2. Given the short C program that follows,

   (a) Make a list of the memory variables in this program.
   (b) Which lines of code contain operations that change the contents of memory? What are those operations?

```
int main( void )
{
    double base;          /* Input variable. */
    double height;        /* Input variable. */
    double area;          /* To be calculated. */

    printf( "Enter base and height of triangle: " );
    scanf( "%lg", &base );
    scanf( "%lg", &height );
    area = base * height / 2.0;
    printf( "The area of the triangle is %g\n", area );
    return 0;
}
```

3. Write a simple program that will output your name, phone number, E-mail address, and academic major on separate lines.

Chapter 3

The Core of C

The C language is a powerful language that has been used to implement some of the world's most complex programs. However, by learning a modest number of fundamental concepts, a newcomer to C can write simple programs that do useful things. As with any language, the beginner needs a considerable amount of basic information before it is possible to read or write anything meaningful. In this chapter, we introduce some of the capabilities of C by using them in simple, but practical, programs and explaining how those programs work. These examples introduce all the concepts necessary to read and write simple programs, providing an overview of the central parts of the language. Later chapters will return to all of the topics introduced here and explain them in greater detail.

3.1 The Process of Compilation

In Chapter 2, we discussed the process of creating a C program. One of the stages in that process is compilation (Section 2.2). The compilation step can also be broken into stages, as shown in Figure 3.1. It helps a programmer to understand what these stages are, and the problems that can arise at each stage.

Stage	Purpose
1. Lexical analysis	Identify the words and symbols used in the code
2. Preprocessing	Carry out the preprocessor commands
3. Parsing	Analyze the structure of each declaration and statement, according to the official grammar of the language
4. Code generation	Select the appropriate machine operation codes and produce the actual machine instructions that will carry out the C-language program statements The object code can be saved in a file or linked with library code modules, as shown in Figure 2.6, to form an executable program.

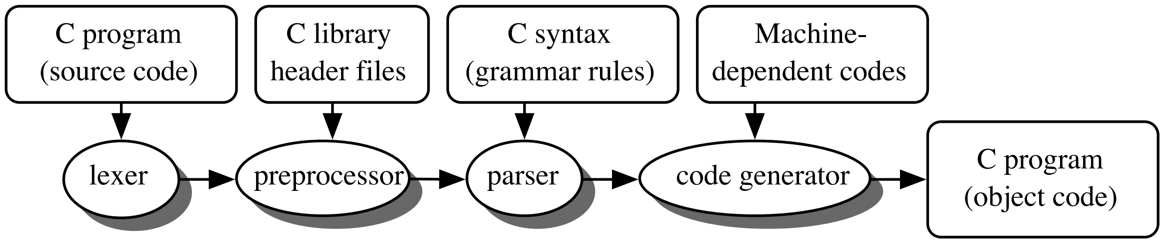
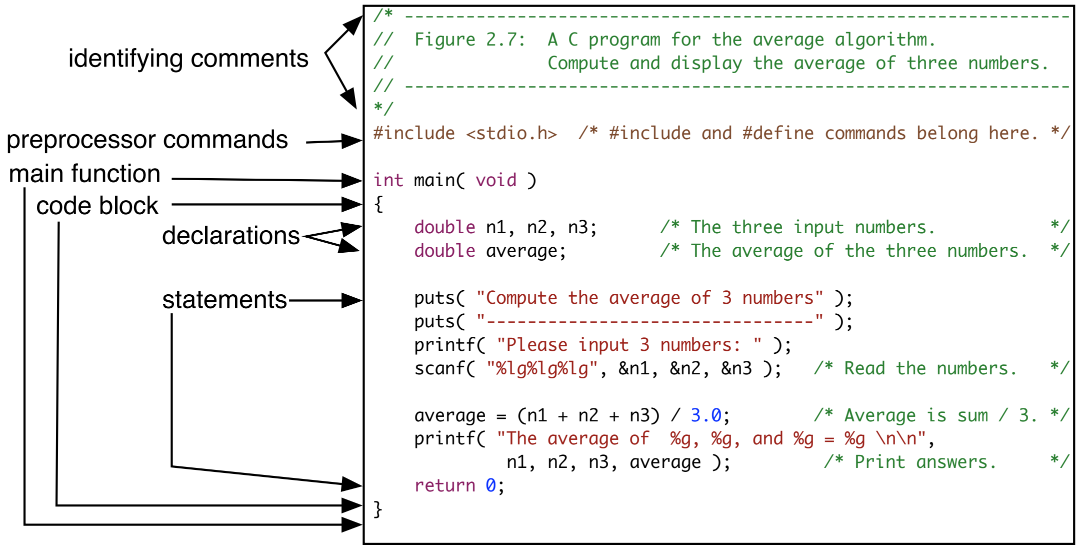


Figure 3.1. The stages of compilation.



```c
/* ---------------------------------------------------------------
//  Figure 2.7:  A C program for the average algorithm.
//               Compute and display the average of three numbers.
// ---------------------------------------------------------------
*/
#include <stdio.h>  /* #include and #define commands belong here. */

int main( void )
{
    double n1, n2, n3;       /* The three input numbers.           */
    double average;          /* The average of the three numbers.  */

    puts( "Compute the average of 3 numbers" );
    puts( "--------------------------------" );
    printf( "Please input 3 numbers: " );
    scanf( "%lg%lg%lg", &n1, &n2, &n3 );   /* Read the numbers.   */

    average = (n1 + n2 + n3) / 3.0;        /* Average is sum / 3. */
    printf( "The average of  %g, %g, and %g = %g \n\n",
             n1, n2, n3, average );         /* Print answers.     */
    return 0;
}
```

Figure 3.2. How to lay out a simple program.

Lexical analysis: Identify the words and symbols used in the code. Some words are built into C and form the core of the language; these are called *keywords* and are listed in Appendix **??**. New words can be defined by the programmer by giving a declaration. For example, in Figure 2.7, two declarations were used to define the words `n1, n2, n3`, and `average`.

Preprocessing: Carry out the preprocessor commands, which all start with the `#` symbol on the left end of the line. These commands are used to bring sets of C library declarations into your program. In this chapter, you will see that they can also be used to define symbolic names for constants.

Parsing: Analyze the structure of each declaration and statement according to the official grammar of the language. When the compiler parses your code it may discover missing punctuation, incomplete statements, bad arithmetic expressions, and various other kinds of structural errors.

Code generation: Select the appropriate machine operation codes and produce the actual machine instructions that will carry out the C-language program statements. If the compiler can parse your code correctly, and can find definitions for all of your symbols, it will generate machine code and store it in an object-code file.

3.2 The Parts of a Program

3.2.1 Terminology.

A C program is a series of comments, preprocessor commands, declarations, and function definitions. The function definitions contain further comments, statements, and possibly more declarations. Each of these program units is composed of a series of words and symbols. We define these terms very broadly in the next several paragraphs; their meaning gradually should become clearer as you look at the first several sample programs. As you read these definitions, you should refer to the diagram in Figure 3.2

Comments. At the top of a program, there should be a few lines that start with `//` or are enclosed between the symbols `/*` and `*/`. These lines are comments that supply information about the program's purpose and its author. Their purpose is to inform the human reader, not the computer, and so they are ignored by the

C Category	In English	Purpose
Identifiers	Nouns	Used to name objects
Data types	Adjectives	Used to describe the properties of an object
Operators	Verbs	Denote simple actions like add or multiply
Function calls	Verbs	Denote complex actions like finding a square root
Symbols	Punctuation	Symbols like a semicolon or # are used to mark the beginning or the end of a program unit
Symbols	Grouping	Pairs of parentheses, braces, quotes, and the like are used to enclose a meaningful unit of code

Figure 3.3. Words in C.

compiler. Comments also can and should appear throughout the program, wherever they could help a reader understand the form or function of the code.

Preprocessor commands. Also, ly a series of commands at the top of a program tell the C compiler how to process the pieces of a program and where to look for essential definitions. These **preprocessor commands** all start with the symbol # and are handled by the C preprocessor before the compiler starts to translate the code itself, as indicated in Figure 3.1.

Words. Many of the words used to write a program are defined by the C language standard; these are known as *keywords*. A list of keywords can be found in Appendix **??**. Other words are defined by the programmer. These words can be grouped into categories analogous to English parts of speech; the table in Figure 3.3 lists the kinds of words in C and their English analogs.

Declarations. The purpose of a **declaration** is to introduce a new word, or **identifier,** into the program's vocabulary. It is like a declarative sentence: it gives information about the objects a program will use. In C, a declaration must be written in the program prior to any statement that uses the word it defines.

Statements. A **statement** is like an imperative sentence; it expresses a complete thought and tells the compiler what to tell the computer to do. Just as an English sentence has a verb and a subject, a typical C statement contains one or more words that denote actions and one or several names of objects to use while doing these actions. The objects we study in this chapter are called *variables* and *constants*; the action words are assignment statements, arithmetic operations, and function calls. An entire program is like an essay, covering a topic from beginning to end.

Blocks. Sometimes, C statements are grouped together by enclosing them in braces. Such a group is called a *block*. Blocks are used with **control statements** to into action units that resemble paragraphs.

Putting it all together. This chapter shows how to use a subset of the C language we call the *beginner's toolbox*. It consists of the basic elements from each category: comments, preprocessor commands, declarations, objects, actions, and control statements. Each section will focus on one or a few elements and use them in a complete program. The accompanying program notes should draw your attention to the practical aspects of using each element in the given context.

The sample programs should help you gain a general familiarity with programming terminology and the C language and provide examples to guide the first few programming efforts. Each topic is revisited in more depth in later chapters. As you read this material, try to understand just the general purpose and form of each part. Then test your understanding by completing one of the skeletal programs given at the end of the chapter.

3.2.2 The main() program.

Every C program must have a function named `main()`, with a first line like the one shown here, followed by a block of code in braces. Identifying comments and preprocessor commands come before `main()`. The last line of code in `main()` is `return 0`. This much never changes.

The keyword `void` appears in the parentheses after `main()`. This means that `main()` does not receive any information from the operating system. (Some complex programs do receive data that way.) The type name `int` appears before the name `main()` to declare that `main()` will return a termination code to the operating system when the program finishes its work.

The `{` and `}` (curly bracket) symbols may be read as "begin" and "end." The braces and everything between them is called a **program block**. For simplicity, we usually use the shorter term **block**. Every function has a block that defines its actions, and the statements in this block are different for every program. They are carried out, in the order they are written, whenever the function is executed. A simple sequential program format is described below; it has three main phases: input, calculation and output. Most short programs follow this pattern.

```
int main( void )
{
    Variable declarations, each with a comment that describes its purpose.

    An output statement that identifies the program.
    Prompts and statements that read the input data.
    Statements that perform calculations and store results.
    Statements that echo input and display results for user.
    return 0;
}
```

3.3 An Overview of Variables, Constants, and Expressions

Before we can write a program that does anything useful, we need to know how to get information into and out of a program and how to store and refer to that information within the program. This section focuses on how a program can define and name objects and how those objects can be used in computations. Several ways are introduced to write variable declarations and constant definitions. Formal rules and informal guidelines for naming these objects are discussed and informal rules for diagramming objects are presented.

3.3.1 Values and Storage Objects.

A **data value** is one piece of information. Data values come in several types; the most basic of these are numbers, letters, and strings of letters. When a program is running in a computer, it reads and writes data values, computes them, and sends them through its circuits. A computed data value might stay for a while in a CPU register, be stored into a variable in memory, or be sent to an output device. C keeps track of where data values are when they are being moved around within the computer, but you must decide when to output or store a value (assign it to a variable). C also permits you to give a symbolic name to a value.

We use the term *storage object*, or simply *object* to refer to any area of memory in which a value can be stored[1]. Objects are created and named by declarations. Each one has a definite type and an address. Many, but not all, contain values (the rest contain garbage). Some are variable, some are constant.

3.3.2 Variables

A **variable** is an area of computer memory that has been given a name by the program and can be used to store, or remember, a value. You can visualize it as a box that can hold one data value at a time.

The programmer may choose any name (there are certain rules to follow when picking a name) for a variable, so long as it is not a keyword[2] in the language. Good programmers try to choose meaningful names that are not too long.

Each variable can contain a specific **type** of value, such as a letter or a number. In this chapter, we introduce the first three[3] data types:

- `char` is used to store characters (letters, punctuation, digits)
- `double` is used for real numbers such as 3.14 and .0056. This types can represent an immense range of numbers but does so with limited precision.
- `int` is used for integers such as 31 and 2006.

[1]This terminology is normal in C, and is consistent with the use of the term in C++. However, the term "object" in Java is used only for instances of a class.

[2]Keywords are words such as `main()` and `void` that have preset meanings to the translator and are reserved for specific uses. These are listed in Appendix **??**.

[3]Other types are introduced in Chapters 7, 15, and 11,

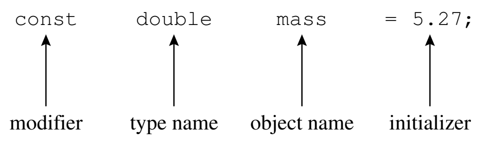


Figure 3.4. Syntax for declarations.

The amount of memory required for a variable depends on its type and the characteristics of the local computer system. Normally, simple variables are 1 to 8 bytes long. For example, an `int` (in many machines) uses four bytes of memory while a `double` occupies eight, and a `char` only one[4].

Declarations. A variable is created by a **declaration**. (Unlike some languages, C requires every variable name to be declared explicitly.) The declaration specifies the type of object that is needed, supplies a name for it, and directs the compiler to allocate space for it in memory. Use of a variable is restricted to the statements within that block that declares it. We say such variables are **local to the block** because they are not "visible" to other blocks.

Declarations can appear anywhere in your code[5]. Most declarations are written at the beginning of a block of code, just after the opening `{`. This placement makes the declarations easy to locate. Sometimes, however, we declare a variable in the middle of a block, especially when it is going to be used to control a loop or a switch[6].

Syntax for declarations. The simplest declaration is a type name followed by a variable name, ending with a semicolon. The following declaration creates an **integer variable** named `minutes`:

```
int minutes;
```

For each name declared, the compiler will create a storage object by allocating the right amount of memory for the specified type of data. The address of this storage object becomes associated with or *bound* to the variable name. Thereafter, when the programmer refers to the name, the compiler will use the associated address. Sometimes we need to refer explicitly to the address of a variable; in those contexts, we write an ampersand (which means "address of") in front of the variable name. For example, `&minutes` means "the address of the variable `minutes`."

The general form of a declaration, shown in Figure 3.4, contains four parts: zero or more modifiers,[7] a type name, an object name or names, and an optional initializer for each name. The parts are written in the order given, followed by a terminating semicolon, as diagrammed in Figure 3.4. The various different data types will be explored in the next few chapters. Object naming conventions are discussed shortly. An **initializer** gives a variable an initial value. This is done by using an `=` sign, followed by a value of the appropriate type.[8]

If multiple objects are declared, and possibly initialized, using a single statement, the objects are separated by commas. We have adopted a style in which most declaration lines declare one variable and have a comment explaining the use of that variable. This is a good way to write a program and make it self-documenting.

Declaration examples. Two examples of variable declarations were given in Chapter 2, Figure 2.7, which declares four `double` variables, `n1, n2, n3`and `average`. Figure 3.5 illustrates variations on the basic declaration syntax and shows how to draw diagrams of variables. The first line declares one variable of type `double`, as we have done many times already. This line tells C to allocate enough memory to store a `double` value and

[4]A full treatment of types `int` and `double` is given in Chapter 7 where we deal with representation, overflow, and the imprecise nature of floating point numbers.

[5]The C99 standard loosened the restrictions on placement of declarations.

[6]Loops and switches will be covered later in this chapter and in Chapter 6.

[7]In the C standard, the properties `const` and `volatile` are called *type qualifiers* and `extern`, `static`, `register`, and `auto` are called *storage class specifiers*. For simplicity, and because the distinctions are not important here, we refer to both as *modifiers*. The modifiers are optional and, with the exception of `const`, you need not understand or use them for the time being. We use `volatile` in Chapter 15. We discuss `static` in Chapter 10 and all storage classes, in general, in Chapter 19. Additionally, the modifier `extern` is important in complex, multimodule programs and is demonstrated in Chapter 21.

[8]The rules for initializers will be given as each type is considered.

Four variables are declared here and diagrammed below.

```
int main( void )
{
    double length;           // Length, in meters (uninitialized).
    double weight = 1.5;     // Weight, in kilograms (initialized to 1.5).
    int k, m = 0;            // One unitialized and one initialized variable.
    char gender = 'F'        // Use a char literal to initalize a char variable.
    ...                      // Rest of program goes here.
}
```

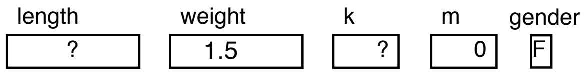


Figure 3.5. Simple declarations.

use that storage location whenever the name `length` is mentioned. The declaration does not put a value into the variable; it will contain whatever was left in that storage location by the program that previously ran on the computer. This value is unpredictable and meaningless in the present context. Formally, we say that it is an **undefined value**; informally, we call it **garbage**. The garbage stays there until a value is read into the variable (perhaps by `scanf()`) or stored in the variable by an assignment statement.

A variable is diagrammed, as in Figure 3.5, by drawing a box with the variable name just above the box and the current value inside it. The size of each box is proportional to the number of bytes of storage required on a representative C system. In the diagram of `length`, its undefined value is represented by a question mark. Variable diagrams, or **object diagrams**, are a concrete and visual representation of the abstractions that the program manipulates. We use them to visualize relationships among objects and to perform step-by-step traces of program execution. Object diagrams become increasingly important when we introduce compound data objects such as arrays, pointers, and data structures.

The second line in Figure 3.5 declares the variable `weight` and gives it an initial value. This shorthand notation combines the effect of a declaration and an assignment:

```
double weight;      // Weight, in kg.  Contains garbage.
...
weight = 1.5;       // Now weight contains value 1.5
```

As we progress to more complex programs, the ability to declare and initialize a variable in one step will become increasingly important.

The third line in Figure 3.5 declares two variables, `k` and `m`, and it initializes `m`. It does not initialize `k`. To do that, another `=` sign and value would be needed prior to the comma. This omission is a common mistake of beginning programmers. Combining two declarations in this manner on one line saves space and writing effort. However, it does not provide a place to put separate comments explaining the purpose of the variables and so should be done only when the meaning of the variables is the same or they are logically related.

Caution. C has a somewhat arbitrary restriction that all of the declarations in a program block must come immediately after the `{` that follows `main( void )` and before any statements. (This restriction is relaxed in C++.)

Assignment. An assignment statement is one way to put a value into a variable. The term *variable* is used because a program can change the value stored in a variable by assigning a new value to it. An **assignment** begins with the name of the variable that is to receive the value. This is followed by an = sign and a value to be stored (or an arithmetic expression that computes a value, see later). As an example,

```
minutes = 10;    // Store value 10 in variable "minutes".
```

3.3.3 Constants and Literals

Literals. In addition to variables, most programs contain **constants**, which represent values that do not change. A **literal constant** is one that is written, literally, in your code; in the assignment statement `radius = diameter/2.0;` `2.0` is a literal `double`. For each type of value in C, there is a way to write a literal constant.

These will be explained as we go through the various types. Here is a table of literal formats for the three types we have covered

C Syntax for Literal Constants

Type	Examples	Notes
`double`	`3.1, .045`	A string of digits with a decimal point.
`double`	`4.5E-02`	.045 written in scientific notation.
`int`	`32767`	A string of digits with an optional sign; no decimal point, no commas.
`char`	'A' or '%'	A single character enclosed in single quotes.

Symbolic constants. A **symbolic constant** is a constant to which we have given a name. There are two ways in C to create a symbolic constant: a `#define` command and a `const` declaration. Physical constants such as π often are given symbolic names. We can define the constant `PI` as

```
#define PI 3.1416
```

The `#define` commands usually are placed after the `#include` commands at the top of a program. It is sound programming practice to use `#define` to name constants rather than to write literal constants in a program, especially if they are used more than once in the code. This practice makes a program easier to debug because it is easier to locate and correct one constant definition than many occurrences of a literal number.

Note three important syntactic differences between `#define` commands and assignment statements: (1) the `#define` command does not end in a semicolon while the assignment statement does; (2) there is no `=` sign between the defined constant name and its value, whereas there always is an `=` sign between the variable and the value given to it; and (3) you cannot use the name of a constant on the left side of an assignment statement, only a variable.

Often, using symbolic names rather than literal constants saves trouble and prevents errors. This is true especially if the literal constant has many digits of precision, like a value for π, or if the constant might be changed in the future. Well-chosen descriptive names also make the program easier to read. For example, we might wish to use the temperature constant, `-273.15`, which is absolute zero in degrees Celsius. A name like `ABSO_C` is more meaningful than a string of digits.

Using `#define`. A `#define` command is generally used to define a physical constant such as `PI`, `GRAVITY` (see Figure 3.7) or `LITR_GAL` (see Figure 3.18), or to give a name to an arbitrarily defined value that is used throughout a program but might have to be changed at some future time. (The first example of such usage is the loop limit `N` in Figure 6.14.) We use a `#define` for an arbitrary constant so that changing the constant is easy if change becomes necessary. Changing one `#define` is easier than changing several references to the constant value, and finding the `#define` at the top of the file is easier than searching throughout the code for copies of a literal constant.

When you use a name that was defined with `#define`, you actually are putting a literal into your program. Commands that start with `#` in C are handled as a separate part of the language by a part of the compiler, called the *preprocessor*, that looks at (and possibly modifies) every line of source code before the main part of the compiler gets it. The preprocessor identifies the `#define` commands and uses them to build a table of defined terms, with their meanings. Thereafter, each time you refer to a defined symbol, the preprocessor removes it from your source code and replaces it by its meaning. The result is the same as if you wrote the meaning, not the symbol in your source code. For example, suppose a program contained these two lines of source code before preprocessing:

```
#define PI  3.1415927
area = PI * radius * radius;
```

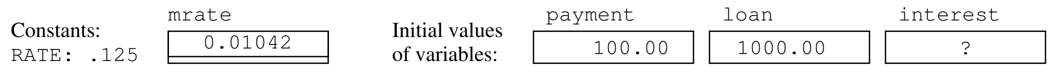
After preprocessing, the `#define` is gone. In some compilers, you can instruct the compiler to stop after preprocessing and write out the resulting code. If you were to look at the preceding assignment statement in that code, it would look like this:

```
area = 3.1415927 * radius * radius;
```

Note the following things about a `#define` command:

- *No* `=` sign appears between the constant's name and its value.

```
#define RATE .125              // Annual interest rate.
const double mrate = RATE/12;  // Monthly interest rate.
double payment = 100.00;       // Monthly payment.
double loan = 1000.00;         // Remaining unpaid principle amount.
double interest;               // Current month's interest.
```



```
interest = mrate * loan ;
loan = loan + interest - payment;
```

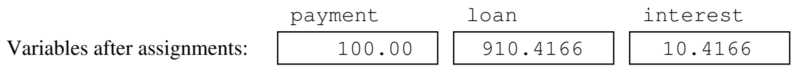


Figure 3.6. Using constants.

- The line does *not* end in a semicolon, and you do *not* write the type of the name. (The C translator will deduce the type from the literal.)
- In this example, `PI` is a `double` constant because it has a decimal point.

The C preprocessor always has been a source of confusion and program errors. However, these difficulties are caused by advanced features of the preprocessor, not by simple constant definitions like those shown here. A beginning programmer can use `#define` to name constants with no difficulty. The programmer usually is unaware of the substitutions made by the preprocessor and does not see the version with the literal constants.

The `const` qualifier. You also can create a constant by writing the `const` *modifier* at the beginning of a variable declaration. This creates an object that is like a variable in every way except that you cannot change its value (so it is not really like a variable at all).[9]. A `const` declaration is like a variable declaration except that it starts with the keyword `const` and it must have an initializer.

A symbolic name is one way to clarify the purpose of a constant and the meaning of the statement that uses the constant. That C provides two different ways to give a symbolic name to a constant value (`#define` and `const`) might seem strange. C does so because each kind of constant can be used to do some advanced things that the other cannot. We recommend the following guidelines for defining constants:[10]

- Use `#define` to name constants of simple built-in types.
- Use `const` to define anything that depends on another constant.

This usage is illustrated in Figure 3.6. The lines in this figure are excerpted from a program that computes a payment table for a loan. The program uses `#define` for the annual interest rate, because while it is constant for this particular loan, it is likely to change for future loans. By placing the `#define` at the top of the program, we make it easy to locate and edit the interest rate at that future time. The monthly rate is one-twelfth of the annual rate; we declare it as a `const double` initialized to `RATE/12`. We use `const` rather than `#define` because the definition involves a computation.

The loan amount and the monthly interest are defined as variables, because they decrease each month. The monthly payment normally will be $100 but we do not define it as a constant because the loan payment on the final month will be smaller.

Following the declarations in the figure are diagrams for the objects declared. The variables `payment`, `loan`, and `interest` are diagrammed as variable boxes. The striped box indicates that `mrate` is a constant variable and cannot be changed. No box is drawn for `RATE` because it is implemented as a literal. The lower line in the diagram shows the changes in the variables produced by the two assignment statements.

[9]A `const` variable has an address. This becomes important in advanced programs and in C++.
[10]These guidelines are consistent with the advanced uses of constants and with usage in C++.

3.3.4 Names and Identifiers

As a programmer works on the problem specification and begins writing code to solve a problem, he or she must analyze what variables are needed and invent names for them. No two programmers will do this in exactly the same way. They probably will choose different names, since naming is wholly arbitrary. They might even use different types of variables or a different number of variables, since the same goals can often be accomplished in many ways. In this section, we introduce guidelines and formal syntactic rules for naming objects.

The technical term for a name is an **identifier**. You can use almost any name for an object, subject to the following constraints. These are absolute rules about names:

1. It must begin with a letter or underscore.
2. The rest of the name must be made of letters, digits, and underscores.
3. You cannot use C keywords such as `double` and `while` to name your own objects. You also should avoid the names of functions and constants in the C library, such as `sin()` and `scanf()`.
4. C is case sensitive, so `Volume` and `volume` are different names.
5. Some compilers limit names to 31 characters. Very old C compilers use only the first eight characters of a name.

These are guidelines for names:

1. Use one-letter names such as `x, t`, or `v` to conform to standard engineering and scientific notation. Writing `d = r * t` is better for our purposes than writing the lengthier `distance = rate * time`. Otherwise, avoid single-letter names.
2. When you have two similar quantities, such as two time instances, you might call them `t1` and `t2`. Otherwise, avoid using such similar names.
3. Use names of moderate length. Most names should be between 2 and 12 letters long.
4. Avoid names that look like numbers; `O`, `l`, and `I` are very bad names.
5. Use underscores to make compound names easier to read: `tot_vol` or `total_volume` is clearer than `totalvolume`.
6. Try to invent meaningful names; `x_coord` and `y_coord` are better names than `var1` and `var2`.
7. Do not use names that are very similar, such as `metric_distance` and `meter_distance` or `my_var` and `my_varr`.

3.3.5 Arithmetic and Formulas

Arithmetic formulas, which are called **expressions**, are written in C in a notation very much like standard mathematical notation, using the **operators** `+` (add), `-` (subtract), `*` (multiply), and `/` (divide). Normal mathematical operator **precedence** is supported; that is, multiplication and division will be performed before addition and subtraction. As in mathematics, parentheses may be used for grouping[11]. These operators are combined with variable names (such as `radius`), constants (such as `PI`), or **literal** values such as 3.14 or 10 to form expressions. The result of an expression can be used for output or it can be stored in memory by using assignment. For example, the following statement computes the area of a circle with radius `r` and stores the result in the variable `area`:

```
    area = 3.1416 * r * r;
```

Using a constant definition (`#define PI 3.1416`) permits us to rewrite the area formula thus:

```
    area = PI * r * r;
```

[11]Many more operators and the details of operator precedence and associativity are given in Chapter 4

3.4 Simple Input and Output

A call on an input function is one way to put a value in a variable; we call an output function when we want to see the value stored there. The input and output facilities in C are some of the most complex parts of the language, yet we need to use them as part of even the simplest programs. The best way to start is to learn to do input and output in a few simple ways. In this chapter, we focus on the elementary use of just three I/O functions:

Function name	Meaning of name	Purpose of function
`puts()`	Put string and newline	To write a message and a newline on the screen
`printf()`	Print output using a format	To write messages and data values.
`scanf()`	Scan input using a format	To read a data value from the keyboard

These functions (and many others) are in the standard input/output library (`stdio`), which is "added" to the program when you write `#include <stdio.h>`. Several examples given in this chapter use these functions with the hope that you can successfully imitate them.[12]

Streams and buffers. Input and output in C are handled using **streams**. An *input stream* is a device-independent connection between a program and a specified source of input data; an *output stream* connects a program to a destination for data. In this book, we use streams connected to data files or devices such as the computer's keyboard and monitor screen. Three streams are predefined in standard C: one for input (`stdin`), one for output (`stdout`), and one for error comments (`stderr`). The standard output stream is directed by default to the operator's video screen but can be redirected to a printer or a file.[13] The standard input stream normally is connected to the keyboard but also can be redirected to get information from a file.

Input and output devices are designed to handle chunks of data. For example, a keyboard delivers an entire line of characters to the computer when you hit Enter, and a hard disk is organized into clusters that store 1,000 or more characters. When we read from or write to a disk, the entire cluster is read or written. On the other hand, programs typically read data only a few numbers at a time and write data one number or one line at a time. To bridge the gap between the needs of the program and the characteristics of the hardware, C uses buffers. Every stream has its own **buffer**, an area of main memory large enough to hold a quantity of input or output data appropriate for the stream's device.

When the program uses `scanf()` to read data, the input comes out of the buffer for the `stdin` stream. If that buffer is empty, the system will stop and wait for the user to enter more data. If a user types more data than are called for, the extra input remains in the input buffer until the next use of `scanf()`.

Similarly, when the program uses `printf()` to produce output, that output goes to the output buffer and stays there until the program prints a newline character (denoted by `\n`) or until the program stops sending output and switches to reading input. This permits a programmer to build an output line one number at a time, until it is complete, then display the entire line. However, the most common use of `printf()` is to print an entire line at one time.

3.4.1 Formats.

Some input and output functions, like `puts()`, read or write a single value of a fixed type. A call on `puts()`, which outputs a single string, is very simple: We write the message enclosed in double quotes inside the parentheses following the function name. For example, we might write this lines as the first statement in a program that computes square roots:

```
    puts( "Compute the square root of a number." );
```

Other input and output functions, including `scanf()` and `printf()`, can read or write a list of values of varying types. Using these functions is more complex than using `puts()` because two kinds of things may be written inside the parentheses. First comes a **format** string, which describes the form of the data. It describes

[12] A note about notation: In writing about C functions, we often use the name of a function separately, outside the context of program code. At these times, it is customary to write an empty pair of parentheses after the function name. The parentheses remind us that we are talking about a function rather than some other kind of entity. In a program, they distinguish a function call (with parentheses) and a reference to a function (without).

[13] The complexities of streams will be explored in Chapter 14.

how many items will be read or written and the type of each item. Following that is either a list of addresses for the input data (for `scanf()`) or a list of expressions whose results are to be printed (for `printf()`). The complete set of rules for formats is long and detailed.[14] Fortunately it is not necessary to understand formats fully in order to use them. In this chapter, we show how to write simple formats for three types of data.[15]

A format is a string (a series of characters enclosed in double quotes) that describes the form and quantity of the data to be processed. Input and output formats are quite different and will be described separately. Both, however, contain conversion specifiers. The **conversion specifiers** in the format tell the input or output function how many data items to process and what type of data is stored in each item. Each conversion specifier starts with a percent sign and ends with a code letter(s) that represents the type of the data.

Data Type	**Input Specifier**	**Output Specifier**	**Notes**
`char`	`" %c"`	`"%c"`	Type a space between the quote and the `%` sign for input.
`int`	`"%i"`	`"%i"`	`"%d"` also works.
`double`	`"%lg"`	`"%g"`	Use `%lg` for input, but only `%g"` (without the letter `l`) for output.

Using input formats. An input format is a series of conversion specifiers enclosed in quotes; for example, the format string `"%i"` could be used to read one integer value and `"%i%i"` could be used to read two integers that are separated by spaces on the input line. In a call on `scanf()`, the format is written first, followed by a list of the addresses of the variables that will receive the data after they are read. Here are two complete calls on `scanf()`:

```
scanf( "%i", &minutes );
scanf( " %c%i%lg", &gender, &age, &weight );
```

The first line tells `scanf()` to read one integer value and store it at the memory location allotted to the variable named `minutes`. The second line tells `scanf()` to read a character an integer, and a real number. The character will be stored at the address of the variable named `gender`, the integer in the variable named `age`, and the real number in the variable named `weight`.

Using output formats. Output formats contain both conversion specifiers and words that the programmer wishes to see interspersed within the data. Therefore, output formats are longer and more complex than input formats. An appropriate output format to print the data just read might look like this:

```
"Gender: %c  Age: %i  Weight: %g\n"
```

The words and the spaces are written exactly the way they should appear on the screen. The `%c`, `%i` and `%g` tell `printf()` where to insert the data values in the sentence. The `\n` represents a newline character. We need it with `printf()` to cause the output cursor to go to a new line. (The `\n` is not needed with `puts()`.) Hence, most `printf()` formats end in a newline character. Make this a habit: *Use* `\n` *at the end of every format string to send the information to your screen immediately and prepare for the next line.*[16]

Following the format string in a call on `printf()` is the list of variables or expressions whose values we want to write (do not use the ampersand for output). Exactly one item should appear in this list for each `%` in the format. A complete call on `printf()` might look like this:

```
printf( "Gender: %c  Age: %i  Weight: %g\n", gender, age, weight );
```

This tells `printf()` to print the values stored in the variables gender, age, and weight. These values will appear in the output after labels that tell the meaning of each number. The output from this line might be

```
Gender: M  Age: 21  Weight: 178.5
```

[14]These rules can be found in any standard reference manual, such as S. Harbison and G. Steele, *C: A Reference Manual*, 4th ed (Englewood Cliffs, NJ: Prentice-Hall, 1995).

[15]As other types of data are introduced in Chapters 7 through 11, more details about formats will be presented.

[16]The exception is a format string used to display a user prompt. These normally end in a colon and a space so that the screen cursor does not move to a new line and the user types the input on the same line as the prompt.

3.5 A Program with Calculations

Now you know how to define constants and variables, how to get letters and numbers into and out of the computer's memory, and how to do simple calculations. You know, in general, the form of a program. In this section, we combine all these elements in a simple program in Figure 3.7 to illustrate variable declarations, assignments, input, output, and calculations. In this program, some parts of the code are boxed. Program notes corresponding to these boxes are given below.

The problem. A grapefruit is dropped from the top of a very tall building[17]. It has no initial velocity since it is dropped, not thrown. The force of gravity accelerates the fruit. Determine the velocity of the fruit and the distance it has fallen after `t` seconds. The time, `t`, is read as an input.

Notes on Figure 3.7: Grapefruits and gravity.

Introductory comments.

- The first two lines of "Grapefruits and Gravity" are a comment block. The first line of such a block starts with `/*` and the last line ends with Other lines in the block do not need any special mark at the beginning, but using ** or `//` (as shown here) makes an attractive heading.

[17]This problem will serve as the basis for a series of example programs in the following sections.

```c
// Determine the velocity and distance traveled by a grapefruit
// with no initial velocity after being dropped from a building.
//

#include <stdio.h>
#define GRAVITY 9.81         // gravitational acceleration (m/s^2)

int main( void )
{
    double t;   // elapsed time during fall (s)
    double y;   // distance of fall (m)
    double v;   // final velocity (m/s)

    printf( "\nWelcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );

    printf( " Input seconds: " );  // prompt for the time, in seconds.
    scanf ( "%lg", &t );           // keyboard input for time

    y = .5 * GRAVITY * t * t;      // calculate distance of the fall
    v = GRAVITY * t;               // velocity of grapefruit at impact

    printf( "    Time of fall = %g seconds \n", t );
    printf( "    Distance of fall = %g meters \n", y );
    printf( "    Velocity of the object = %g m/s \n", v );

    return 0;

}
```

Figure 3.7. Grapefruits and gravity.

- Good programmers put comments at the top of a program to identify the program, its author, and the date or version number. Comments are also written throughout the code to explain the purpose of each group of statements. The compiler does not use the comments—we write them for the benefit of the humans who will read the code.

First box: preprocessor commands.

- Any line that starts with a `#`, called a *preprocessor command*, is handled by the preprocessor before the compiler translates the code.
- In this program, as in most, we ask the preprocessor to bring in the file `stdio.h` and include the contents of that file as part of the program. This is the header (basically, the table of contents) for the standard I/O library. This command makes the standard input-output library functions available for use.
- We use a `#define` command to give a symbolic name to the constant for the acceleration of gravity at sea level.

Second box: the declarations.

- Declarations are used to define the names of variables that will be used in statements later in the program.
- We declare three `double` variables named `t, y`, and `v`. This instructs the compiler to allocate enough space in memory to store three real values and use the appropriate location every time we refer to `t, y`, or `v`. On most common machines, 8 bytes of memory will be allocated for each variable.
- Note that `t, y`, and `v` all are variables of type `double`. When you have several variables of the same type, you also declare them on separate lines, as shown, or you may declare them all on one line like this: `double t, y, v;`. In this problem, we declare one variable per line so that there is space for a comment that explains the meaning or purpose of each variable. This is good programming practice.

Third box: printing an output title.

- A well-designed program displays a title and a greeting message so that the user knows that execution has started.
- We could use either `puts()` or `printf()` to print the title, but if we use `printf()` we must end the string with a newline character if we want the output cursor to move down to the next line.
- Here, we print a 3-line title, using three lines of code. Where we break the code, we end the line with a quote mark and start the next with indentation and a quote mark.
- Program execution begins with the first box following the declarations, proceeding sequentially through the other boxes to the end of the code. This is studied more fully in the next section.

Fourth box: user input.

- A **prompt** is a message displayed on the video screen that tells the user what to do. A program must display a prompt whenever it needs input from the human user.
- In this prompt, we ask the user to type in the time it took the grapefruit to hit the ground. We *do not* put a newline character at the end of the message because we want the input to appear on the same line as the prompt. We *do* leave a space after the colon to make the output easy to read. The user sees

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 10
```

- In the call on `scanf()`, we send two pieces of information to the `scanf()` function: the format string and the address of the variable to receive the input. This causes the system to scan the `stdin` input stream to find and read a value for `t` (time, in seconds). This value may be entered with or without a decimal point.
- The `%` sign in the format is the beginning of a conversion specifier. It tells us to read one item and specifies the type of that item. We use `%lg`, for "long general-format real," to read a value of type `double`. (The letter between the `%` and the `g` is a lowercase *letter* `l`, not a numeral 1.)
- After the format comes the address for storing the input data. In this case, the data will be stored at `&t`, the address of the variable `t`.

Fifth box: calculations.

- The symbol `=` is used to store a value into a variable. Here we store the result of the calculation `.5 * GRAVITY * t * t`, which is the standard equation for distance traveled under the influence of a constant force after a given time, in the variable named `y`.
- The calculation for height, `y`, uses several `*` operators. The `*` means "multiply." When there is a series of `*` operators, they are executed left to right.
- The second formula is standard for computing the terminal velocity of an object with no initial velocity under the influence of a constant force.

Sixth box: the output.

- First, the input value (time) is echoed onto the video screen to convince the user that the calculations were done with the correct value.
- Two `printf()` statements write the answers to the screen. We use a `%g` specifier to write a `double` value. Note that this is different from a `scanf()` format, where we need `%lg` for a `double`.
- The `printf()` function leaves the output cursor on the same output line unless you put a newline character, `\n`, in the format string. In this case, we use a `\n` to put the velocity value on a different line from the distance.
- This would be a typical output:

```
 Welcome.
 Calculate the height from which a grapefruit fell
 given the number of seconds that it was falling.

 Input seconds: 10
    Time of fall = 10 seconds
    Distance of fall = 490.5 meters
    Velocity of the object = 98.1 m/s

Gravity has exited with status 0.
```

Last box: termination.

- It is sound programming practice to display a termination comment. This leaves no doubt in the mind of the user that all program actions have been completed normally.
- At the top of every program, we write `int main( void )`. This tells the operating system to expect to receive a termination code from your program.
- The last line in every program should be a `return 0;` statement. The zero is a termination code that tells the system that termination was normal.

3.6 The Flow of Control

All the examples we have considered so far execute the code (instructions) line by line from beginning to end. However, in most practical programs, it is necessary to follow alternative paths of execution, depending on the data values and other conditions. For example, suppose you want to calculate the velocity of an object, as in Figure 3.7, but you want to prohibit cases that make no physical sense; that is, negative values of time. Two potential courses of action could be taken: (1) Do the calculation and display the answer or (2) comment on the illegal input data and skip the calculation. Another reason we need nonsequential execution is to enable a program to analyze many data values by repeating one block of code.

When a C program is executed, action starts at the first statement following the declarations. For example, in Figure 3.7, execution starts with the `printf()` statement. From there, execution proceeds to the next statement and the next, in order, until it reaches the `return` statement at the program, at which point control returns to the operating system. This is called **simple sequential execution**. *Control statements* are used to create conditional branching and repetitive paths of execution in a program. *Flow diagrams* are used as graphical illustrations of the execution paths that these control statements create.

A **flow diagram** is an unambiguous, complete diagram of all possible sequences in which the program statements might be executed. We use flow diagrams to illustrate the flow of control through the statements of a program or part of a program. This is not so necessary for simple sequential execution. However, This is a flow diagram of the program in Figure 2.7. It illustrates simple straight-line control and the way we begin and end the diagram of a function.

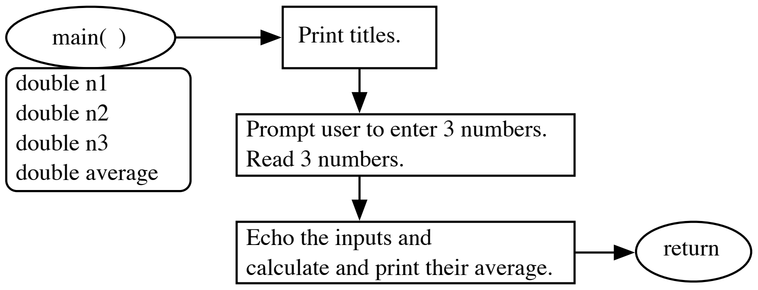


Figure 3.8. A complete flow diagram of a simple program.

a two-dimensional graphic representation can greatly aid comprehension when control statements are used to implement more complex sequencing. A few basic rules for flow diagrams follow and are illustrated in Figure 3.8:

1. A flow diagram for a complete program begins with an oval "start" box at the top and ends with an oval return box at the bottom.
2. Declarations need not be diagrammed unless they contain an initializer (used to give the variable a starting value and discussed more in the next chapter). It is probably clearer, though, if you include one round-cornered box below the start oval that lists the variables declared within the function.
3. The purpose of each statement is written in the appropriate kind of box, each depending on the nature of the action. Since the purpose of the diagram is to clarify the logic of the function, we use English or pseudocode, not C, to describe the actions of the program. Such a diagram could be translated into languages other than C.
4. Simple assignment statements and function calls are written in rectangular boxes. Several of these may be written in the same box if they are sequential and relate to each other.
5. Arrows connect boxes in the sequence in which they will be executed, from start to finish. In the diagram of a complete function, no arrow is left dangling in space and no box is left unattached. All arrow heads must end at a box of some sort, except in the diagram of a program fragment, where the beginning and ending arrows might be left unattached.
6. The diagrams are laid out so that flow generally moves down or to the right. However, you may change this convention if it simplifies your layout or makes it clearer.
7. No arrow ever branches spontaneously. Every tail has exactly one head.

As another example, Figure 3.9 shows the diagram of the program in Figure 3.7. The graph consists of five nodes connected by arrows that indicate the flow of control during execution:

1. A start oval at the top, attached to a box listing the variable declarations.
2. A rectangular box listing statements that print a title and prompt for and read the input.
3. A box that calculates the values for distance and velocity using appropriate formulas.
4. A box that echoes the input and outputs the answers.
5. A return oval at the bottom that terminates the program.

Note that these boxes correspond roughly to the boxed units of code in the program.

3.7 Asking Questions: Conditional Statements

A large part of the power of a computer is the ability to take different actions in different situations. For this purpose, all computers have instructions that do various kinds of conditional branching. These instructions are represented in C by the `if` and `if...else` statements.

This is a flow diagram of the program in Figure 3.7.

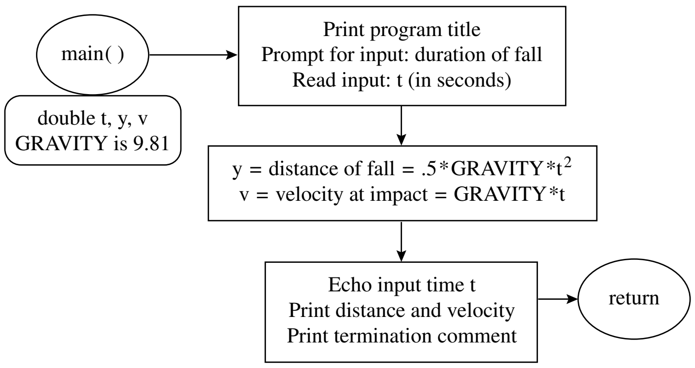

main( )
double t, y, v
GRAVITY is 9.81
Print program title
Prompt for input: duration of fall
Read input: t (in seconds)
y = distance of fall = .5*GRAVITY*t²
v = velocity at impact = GRAVITY*t
Echo input time t
Print distance and velocity
Print termination comment
return

Figure 3.9. Diagram of the grapefruits and gravity program.

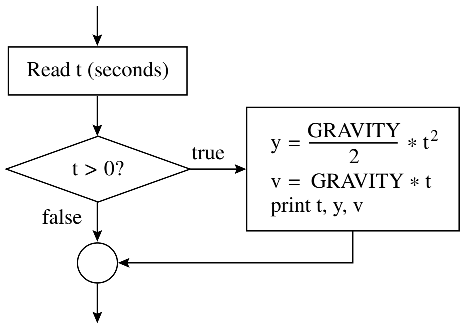
3.7.1 The Simple `if` Statement

In a simple `if` statement, the keyword `if` is followed by an expression in parentheses, called the *condition*, followed by a single statement or a block of statements in braces, called the *true clause*. At run time, the expression is evaluated and its result is interpreted as either true or false. If true, the block of statements following the condition is executed. If false, that block of statements is skipped. Execution continues with the rest of the program following the conditional.

This control pattern is illustrated by the program in Figure 3.10. We use a simple `if` statement here to test whether the input data make sense (the input is a valid time value). If so, we process the data; if not, we do nothing.

Notes on Figure 3.10. Asking a question.

1. The outer box contains the entire simple `if` statement, which consists of
   - The keyword `if`.
   - The condition `(t > 0)`.
   - A block (inner box) containing two assignment statements and three `printf()` statements.
2. As shown in the flow diagram, below, there is only one control path into the `if` unit and one path out. Control branches at the `if` condition but rejoins immediately below the true clause.



3. We use an `if` statement here to test whether the input data makes sense (the time is positive). If so, we follow the `true` path in the diagram and process the data. if not, we follow the `false` path in the diagram and do nothing.

   In either case, the next statement executed will be the final `puts()`. A sample output from the `false` path follows. Note that the user gets no answers at all and no explanation of why. This is not a good human–machine interface; it will be improved in the next program example.

```
Welcome.
Calculate the height from which a grapefruit fell
```

```
 given the number of seconds that it was falling.

 Input seconds:  -1

Gravity has exited with status 0.
```

4. The inner box (the true clause) shows what happens when `t`, the time, is positive: we execute the statements that calculate and print the answers. A sample output from this path might be

```
 Welcome.
 Calculate the height from which a grapefruit fell
 given the number of seconds that it was falling.

 Input seconds: 10
    Time of fall = 10 seconds
    Distance of fall = 490.5 meters
    Velocity of the object = 98.1 m/s

Gravity has exited with status 0.
```

3.7.2 The `if...else` Statement

An `if...else` statement is like a simple `if` statement but more powerful because it lets us specify an alternative block of statements to execute when the condition is false. It consists of

This solves the same problem as Figure 3.7, except that we screen out invalid inputs.

```
// ------------------------------------------------------------------------
// Determine the velocity and distance traveled by a grapefruit
// with no initial velocity after being dropped from a building.
//
#include <stdio.h>
#define GRAVITY  9.81    // gravitational acceleration (m/s^2)

int main( void )
{
    double t;            // elapsed time during fall (s)
    double y;            // distance of fall (m)
    double v;            // final velocity (m/s)

    printf( " \n\n Welcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );

    printf( " Input seconds: " );  // prompt for the time, in seconds.
    scanf( "%lg", &t );            // keyboard input for time

    if (t > 0) {                   // check for valid data.

        y = .5 * GRAVITY * t * t;  // calculate distance of the fall
        v = GRAVITY * t;           // velocity of grapefruit at impact
        printf( "    Time of fall = %g seconds \n",  t );
        printf( "    Distance of fall = %g meters \n",  y );
        printf( "    Velocity of the object = %g m/s \n",  v );

    }
    return 0;
}
```

Figure 3.10. Asking a question.

The keyword `if`.
An expression in parentheses, called the *condition*.
A statement or block of statements, called the *true clause*.
The keyword `else`.
A statement or block of statements, called the *false clause*.

At run time, the condition will be evaluated and either the true or the false clause will be executed (the other will be skipped), depending on the result.

The syntactic difference between an `if...else` and a simple `if` statement is that the true clause of an `if...else` statement is followed immediately by the keyword `else`, while the true clause of a simple `if` statement is not. There must not be a semicolon before the `else`.

One common use of the `if...else` statement is to validate input data, it often is necessary to perform more than one test. For example, one might need to test two inputs or test whether an input lies between minimum and maximum acceptable values. To do this, we can use a series of `if...else` statements, as demonstrated in Figure 3.11, which is an extension of the program in Figure 3.10. In this program, we make two tests using two `if...else` statements in a row. We ensure that the input both is positive and does not exceed a reasonable limit, in this case 60 seconds. Using `if...else` in this manner, we can test as many criteria as needed.

The flow diagram corresponding to this new version of the program is given in Figure 3.12. In this diagram, you can see the column of diamond shapes typical of a chain of `if` statements. The `true` clause actions form another sequence to the right of the tests in the diamonds, and the final `false` clause terminates the diamond sequence. All the paths come together in the bubble before the termination message.

Notes on Figure 3.11: Testing the limits.

First box: maximum time.

- This program checks for inputs that are too large as well as those that are negative.
- Since the upper limit is an arbitrary number, it might be necessary to change it in the future. To make such changes easy, we define this limit, `MAX`, at the top of the program and use the symbolic name in the code.

Large outer box: the `if...else` ***chain.***

- Before computations are made, we inspect the data for two errors (first two inner boxes). We skip the remaining tests and the calculations if either error is discovered. This method of error handling avoids doing a computation with meaningless data, such as the negative value in the previous version of the program.
- If no errors are found, we execute the code in the third inner box.

First inner box: negative input values.

- As in the prior example, the input must be positive to correspond to physical reality. The first `if` statement handles this.
- The `true` clause of this `if` statement prints an error comment. Following that, control goes to the `puts()` statement at the bottom of the program, skipping over the `else if` test and the `else` clause.

Second inner box: input values that are too large.

- The elapsed time also must be reasonable. We compare the input value to a maximum allowable value, as defined in the problem specification (60 seconds in this case). The second `if` statement handles this.
- The `true` clause of this `if` statement prints an error comment. Then control skips the `else` clause and goes to the `puts()` at the bottom of the program.
- Here is a sample of the program's error handling:

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 100
Error: time must be <= 60 seconds.
```

Third inner box: the computation and output.

- If the input seems valid, we calculate and print the distance and velocity using the given formulas. Then we echo the input and print the answers.
- Here is the output (without the titles) from a run with valid data:

```
Welcome.
Calculate the height from which a grapefruit fell
given the number of seconds that it was falling.

Input seconds: 20
   Time of fall = 20 seconds
   Distance of fall = 1962 meters
   Velocity of the object = 196.2 m/s
```

This solves the same problem as Figure 3.10, except that we limit valid inputs to be less than 1 minute.

```
#include <stdio.h>
#define GRAVITY  9.81       // Gravitational acceleration (m/s^2)

#define MAX     60          // Upper bound on time of fall.

int main( void )
{
    sdouble t;     // elapsed time during fall (s)
    double y;      // distance of fall (m)
    double v;      // final velocity (m/s)

    printf( " \n\n Welcome.\n"
            " Calculate the height from which a grapefruit fell\n"
            " given the number of seconds that it was falling.\n\n" );

    printf( " Input seconds: " );   // prompt for the time, in seconds.
    scanf( "%lg", &t );             // keyboard input for time

    if (t < 0) {                    // check for negative input
        printf( " Error: time must be positive.\n\n" );
    }

    else if (t > MAX) {             // Is time value too big?
        printf( " Error: time must be <= %i seconds.\n\n", MAX );
    }

    else {  // Input is valid; calculate distance and velocity
        y = .5 * GRAVITY * t * t;  // calculate distance of the fall
        v = GRAVITY * t;           // velocity of grapefruit at impact
        printf( "    Time of fall = %g seconds \n", t );
        printf( "    Distance of fall = %g meters \n", y );
        printf( "    Velocity of the object = %g m/s \n", v );
    }

    return 0;
}
```

Figure 3.11. Testing the limits.

This is a diagram of the program from Figure 3.11.

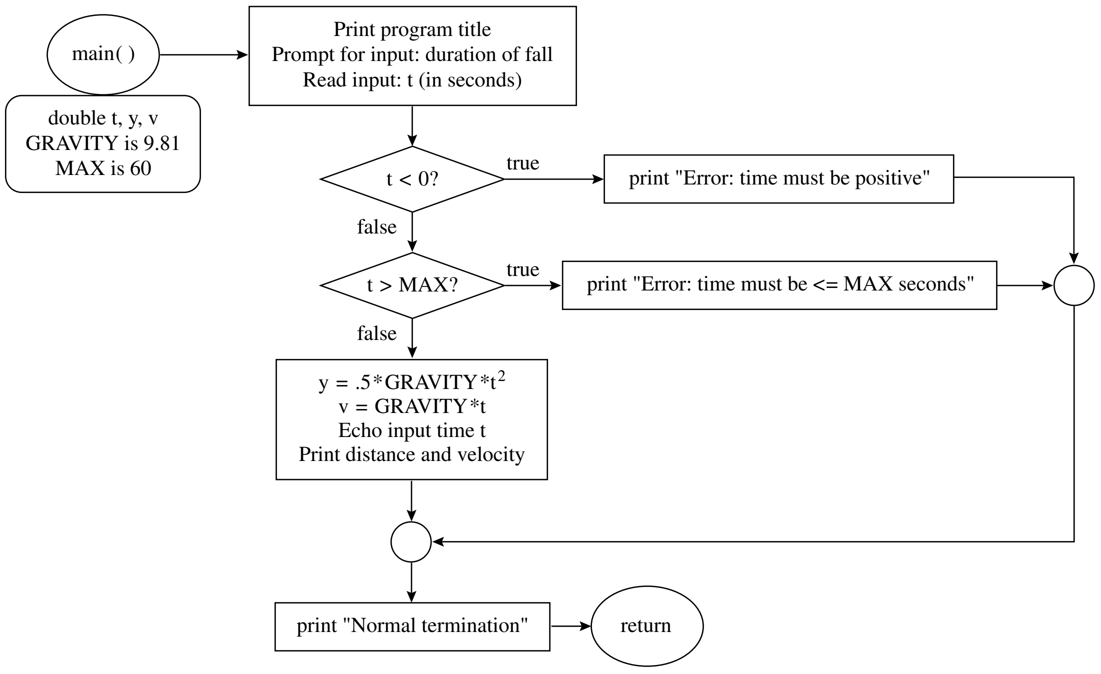

main( )
double t, y, v
GRAVITY is 9.81
MAX is 60
Print program title
Prompt for input: duration of fall
Read input: t (in seconds)
t < 0?
true
print "Error: time must be positive"
false
t > MAX?
true
print "Error: time must be <= MAX seconds"
false
y = .5*GRAVITY*t²
v = GRAVITY*t
Echo input time t
Print distance and velocity
print "Normal termination"
return

Figure 3.12. Flow diagram for Testing the limits.

3.7.3 Which `if` Should Be Used?

We looked at three different ways to use `if` statements. These three control patterns are appropriate for different kinds of applications.

The simple `if` statement. The primary uses of the simple `if` statement:

- Processing a subset of the data items. Sometimes a collection of data contains some items that are relevant to the current process and others that are not of interest. We need to process the relevant items and ignore the others. For example, suppose that a file contains data from a whole census but the programmer is interested only in people over the age of 18. The entire file would need to be read but only a subset of the data items would be selected and processed.
- Handling data items that require extra work. Sometimes a few items in a data set require extra processing. For instance, a cash register program must compute the sales tax on taxable items but not on every item. It must test whether each product is taxable and, if so, compute and add the tax. If not, nothing happens.
- Testing for an error condition or goal condition within a loop and leaving the loop if that condition is found. This use of `if` with `break` will be discussed in Chapter 6.

The `if...else` statement. The primary applications of the `if...else` statement:

- Choosing one from a series of alternatives. Sometimes two or more alternative actions are possible and one must be selected. In this case, we often use a series of `if...else` statements to test a series of conditions and select the action corresponding to the first test whose result is `true`.

  For example, consider a program that computes the roots of a quadratic equation: $ax^2 + bx + c = 0$. If a is zero, the equation is linear, and should be solved by the method for linear equations. If both a and b are zero, the equation is degenerate and has no roots. Otherwise, the equation is quadratic. In this case, the program must first calculate the "discriminant" of the equation; namely, $d = b^2 - 4ac$. If this

With braces the code is spread out:

```
if (age > 18) {
    adults = adults + 1;         // Count the adults.
}
else {
    kids = kids + 1;             // Count the children.
}
```

Without braces the code is more compact:

```
if (age > 18)    adults = adults + 1;  // Count the adults.
else             kids = kids + 1;      // Count the children.
```

Figure 3.13. The `if` statement with and without braces.

value is positive, the roots of the equation are real numbers; if it is negative, the roots are complex. Both cases are valid but require different processing and different output statements. An `if...else` statement would be used to test the discriminant. Then the `true` clause would execute the code for one kind of roots and the `false` clause would contain the code for the other.

The program for solving quadratic equations is on the website: look for a link labelled QuadRoots. This multi-brancheddecision pattern will also be incorporated into programs in later chapters; one example is shown in Figure 13.28.

- Normal processing versus nonfatal error handling. In this control pattern, an input value is read and tested for legality. If it would cause a run-time error, an error message is given and the input is not processed in the usual way. This control pattern is illustrated by the outer box and diagram in Figure 3.11.
- Validating a series of inputs. We can use a series of `if...else` statements to test a series of inputs. The `true` clause of each statement would print an error comment, and the `false` clause of the last statement would process the validated data. This decision pattern is incorporated into Figure 3.11.

3.7.4 Options for Syntax and Layout (Optional Topic)

Normally each part of an `if` statement is written on a separate line and all the lines are indented except the `if`, the `else`, and the closing curly bracket. However, a C compiler does not care how you lay out your code.

Indentation. Inconsistent or missing indentation does not cause any trouble, it simply makes the program hard for a human being to read. Since the compiler determines the structure of the statement solely from the punctuation (semicolons and braces), an extra or omitted semicolon can completely change the meaning of the statement. Therefore, consistent style is important, both to make programs easier to modify and to help avoid punctuation errors.

The condition. The condition in parentheses after the keyword `if` can be any expression; it does not need to be a comparison. Whatever the expression, it will be evaluated. A zero result will be treated as false and a nonzero result will be treated as true. (Note, therefore, that any number except 0 is considered to be *true.*) The name of a variable, or even a call on an input function, is a legal (and commonly used) kind of condition.

Very local variables. Technically, we could declare variables inside any block, even one that is part of an `if` statement. However, this is not an appropriate style for a simple program.

Curly braces. A `true` or `false` clause can consist of either a single statement or a block of statements enclosed in braces. If it consists of a single statement, the braces { and } may be omitted. This can make the code shorter and clearer if the resulting clause fits entirely on the same line as the keyword `if` or `else`. Figure 3.13 illustrates this issue. In it, the same `if` statement is written with and without braces. The first version is preferred by many experts, even though the braces are not required. However, others feel that the second version, without braces, is easier to read because it is written on one line as a single, complete thought. In either case, consistency makes code easier to read; a program becomes visually confusing if one part of an `if` statement has `{` and `}` and the other does not.

3.8 Loops and Repetition

The `if` statement lets us make choices. We test a condition and execute one block of code or another based on the outcome. Another kind of control structure is the loop, which lets us execute a block of statements any number of times (zero or more). Conditionals and loops are two of the three fundamental control structures in a programming language.[18] Loops are important because they allow us to write code once and have a program execute it many times, each time with different data. The more times a computation must be done, the more we gain from writing a loop to do it, rather than doing it by hand or writing the same formula over and over in a program.

C provides three types of loop statements that repeatedly execute a block of code: `while`, `do`, and `for`. The `while` statement is the most basic and is introduced first, in Figure 3.14.[19]

3.8.1 A Counting Loop

Every loop has a set of actions to repeat, called the **loop body**, and a loop test to determine whether to repeat those actions again or end the repetition. In this chapter, we study loops based on counting. In these loops, a variable is set to an initial value, then increased or decreased each time around the loop until it reaches a goal value. We call this variable a **loop counter** because it counts the repetitions and ends the loop when we have repeated it enough times. Our next example, shown in Figure 3.14, introduces a counting loop implemented using a `while` statement. It is a very simple program whose purpose is to make the repetition visible. The corresponding flow diagram used for `while` loops follows the code in the figure.

Notes on Figure 3.14: Countdown.

First box: the loop variable.
A counter is an integer variable used to count some quantity such as the number of repetitions of a loop.

Second box: the loop.

- Before entering a `while` loop, we must initialize the variable that will be used to control it. Here we scan an initial value into `days_left`, the counter variable.
- A `while` statement has three parts, in the following order: the keyword `while`, a condition in parentheses, and a body. The loop body consists of either a single statement or a block of statements enclosed in braces, as shown in the program.
- To execute a `while` loop, first evaluate the condition. If the result is true, the loop body will be executed once and the condition will be retested.
- This sequence of execution is illustrated by the flow diagram at the bottom of Figure 3.14. In the diagram, the `while` loop is an actual closed loop. Control will go around this loop until the condition becomes false, at which point control will pass out of the loop. In this case, control will go to the `puts()` statement. The same diamond shape used to represent a test in an `if` statement is used to represent the loop termination test. The presence of a closed cyclic path in the diagram shows that this is a loop, rather than a conditional control statement.
- The statement `days_left = days_left - 1` tells us to use the old value of `days_left` to compute a new one. Read this expression as "`days_left gets days_left minus 1`" or "`days_left becomes days_left minus 1`". (Do not call this assignment operation *equals*, or you are likely to become confused with a test for equality.) In detail, this statement means
  - Fetch the current value of `days_left`.
  - Subtract 1 from it.
  - Store the result back into the variable `days_left`.

[18]The third basic control structure, functions, will be introduced in Chapter 5.
[19]The other two loop statements will be presented in Chapter 6.

- Each time we execute the body of the loop, the value stored in `days_left` will decrease by 1.
- Sample output:

```
How many days are there until the exam? 4
Days left: 4. Study now.
Days left: 3. Study now.
Days left: 2. Study now.
Days left: 1. Study now.
This is it!  I hope you are ready.
```

- Control will leave the loop when the condition becomes false; that is, the first time that `days_left` is tested after its value reaches 0. This will happen before displaying the 0.

Third box: after the loop.
After leaving the loop, control goes to the statements that follow it. The call on `puts()` displays a message at the bottom of the output. The `return` statement returns to the operating system with a code of 0, indicating successful termination.

This program demonstrates the concept of a counting loop. After inputting N, a number of days, it counts downward from N to 0.

```
#include <stdio.h>

int main( void )
{
    int days_left;                     // The loop counter.

    printf( " How many days are there until the exam? " );
    scanf( "%i", &days_left);           // Initialize the loop counter.

    while (days_left > 0)   {          // Count downward from 20 to 1.
        printf( " Days left: %i. Study now.\n", days_left );
        days_left = days_left - 1;     // Decrement the counter.
    }

    puts( " This is it!  I hope you are ready." );
    return 0;
}
```

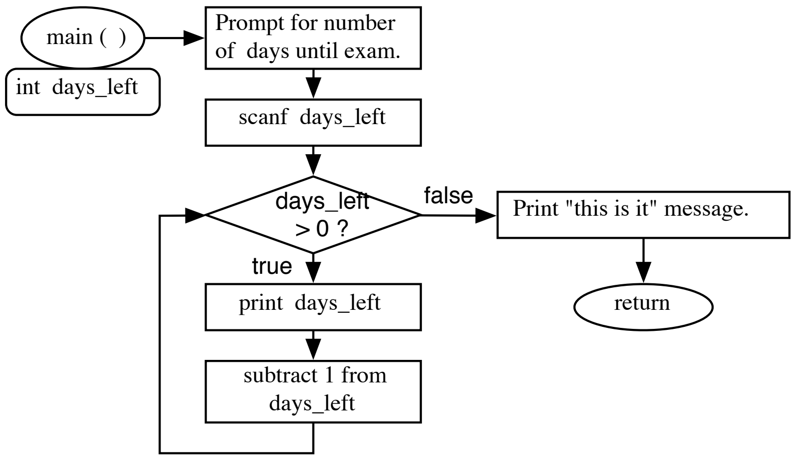

main ( )
int days_left
Prompt for number of days until exam.
scanf days_left
days_left > 0 ?
false
true
Print "this is it" message.
print days_left
return
subtract 1 from days_left

Figure 3.14. Countdown.

3.8.2 An Input Data Validation Loop

Interactive computer users are prone to errors; it is very difficult to hit the right keys all the time. It is common to lean on the keyboard or to hold a key down too long and type 991 instead of 91. A crash-proof program must be prepared for bad input, sometimes called *garbage input*. A much-repeated saying is “garbage in, garbage out” (GIGO); that is, the results of a program cannot be meaningful if the input was erroneous or illegal. Although it is impossible to identify all bad input, some kinds can be identified by comparing the data to the range of expected, legal data values. When an illegal data item is identified, the program can quit or ask for re-entry of the data.

A good interactive program uses input prompts to let the user know what specific inputs or what kinds of

```
// --------------- Compute the average speed of your car on a trip.
#include <stdio.h>
int main( void )
{
    int begin_miles, end_miles   // Odometer readings.
    int miles;                   // Total miles traveled.
    double hours, minutes        // Duration of trip.
    double speed;                // Average miles per hour for trip.
    puts( "\n Miles Per Hour Computation \n" );

    printf( " Odometer reading at beginning of trip: " );
    scanf( "%i", &begin_miles );
    while (begin_miles < 0) {
        printf( " Re-enter; odometer reading must be positive: " );
        scanf( "%i", &begin_miles );
    }

    printf( " Odometer reading at end of trip: " );
    scanf( "%i", &end_miles );
    while (end_miles <= begin_miles) {
        printf( " Re-enter; input must be > previous reading: " );
        scanf( "%i", &end_miles );
    }

    printf( " Duration of trip in hours and minutes: " );
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + (minutes / 60);
    while (hours < 0.0) {
        printf( " Re-enter; hours and minutes must be >= 0.\n" );
        scanf( "%lg%lg", &hours, &minutes );
        hours = hours + (minutes / 60);
    }

    miles = end_miles - begin_miles;
    speed = miles / hours;
    printf( " Average speed was %g \n", speed );
    return 0;
}
```

Figure 3.15. Input validation using `while`.

inputs are acceptable and what to do to end the process. If an input clearly is faulty, the program should (at a minimum) refuse to process it. Preferably, it should explain the error and give the user as many chances as needed to enter good data. One way to do this is the data validation loop.

A **data validation loop** (shown in Figure 3.15) prompts the user to enter a particular data item. It then reads the item and checks whether the input meets criteria for a reasonable or legal value. If so, the loop exits; if not, the user is informed of the error and reprompted. This continues until the user enters an acceptable value.

Notes on Figure 3.15: Input validation using `while`.

First box: A valid odometer reading.

- Before the beginning of a `while` data validation loop, the program must prompt the user for input and read it.
- The `while` loop tests the input. If it is good, the loop body will be skipped; otherwise, control enters the loop. Numbers entered must be small enough to be stored in an integer variable.
- The loop must print an error comment that indicates what is wrong with the input, reprompt the user, and read another input.
- Note that we need two prompts and two `scanf()` statements for this control pattern: one before the loop and another inside the loop.

Second and third boxes: validating the other input.

- Data validation loops all follow a pattern similar to the first loop. The major difference between the first two loops is that the mileage read by the first loop is used in the validation test of the second.
- In the third loop, a computation must be made before the data can be tested. Like the `scanf()` statement, this computation must be written twice, once before entering the loop and again at the end of the loop.

The fourth box: correct data. The box produced this output when correct data were supplied:

```
Miles Per Hour Computation

Odometer reading at beginning of trip:  061234
Odometer reading at end of trip:  061475
Duration of trip in hours and minutes:  4 51
Average speed was 49.6907
```

Faulty data. Here are the results of supplying two kinds of invalid data (greeting and closing comments have been omitted):

```
Odometer reading at beginning of trip: -1
Re-enter; odometer reading must be positive: 061234
Odometer reading at end of trip: 061521
Duration of trip in hours and minutes: 5 28
Average speed was 52.5

Odometer reading at beginning of trip: 023498
Odometer reading at end of trip: 022222
Re-enter; input must be > previous reading: -32222
Re-enter; input must be > previous reading: 032222
Duration of trip in hours and minutes: 148 43
Average speed was 58.6618
```

3.9 An Application

We have analyzed several small programs; now it is time to show how to start with a problem and synthesize a program to solve it. We will create a program for Joe Smith, the owner of a gas station in Niagara Falls, New York. Joe advertises that his prices are cheaper than those of his competitor, Betty, across the border in

1. **Problem scope:** Write a short program for Joe that will compute the price per gallon for one grade of gas, in U.S. funds, that is equivalent to Betty's price for that grade of gas.
2. **Inputs:** (1) The current exchange rate, in U.S. dollars per Canadian dollar. This rate varies daily. (2) The Canadian prices per liter for one grade of gasoline.
3. **Constants:** The number of liters in 1 gallon = 3.78544
4. **Formula:**
$$\frac{\$\_US}{gallon} = \frac{\$\_Canadian}{liter} * \frac{liters}{gallon} * \frac{\$\_US}{\$\_Canadian}$$
5. **Output required:** Echo the inputs and print the equivalent U.S. price.
6. **Computational requirements:** All the inputs will be real numbers.
7. **Limitations:** The exchange rate and the price for gas should be positive. If the user enters an incorrect input, an error message should be displayed and another opportunity given to enter correct input.

Figure 3.16. Problem specification: Gas prices.

Niagara Falls, Ontario. To be sure that his claim is true, he computes the U.S. equivalent of Betty's rates daily. Joe's employee reads Betty's pump prices on the way to work each day, and Joe uses his Internet connection to look up the current exchange rate (U.S. dollars per Canadian dollar). Canadian gas is priced in Canadian dollars per liter. U.S. gas is priced in U.S. dollars per gallon. The conversion is too complicated for Joe to do accurately in his head. Figure 3.16 defines the problem and specifies the scope and properties of the desired solution. We will write a solution step by step. As we go along, we will write a comment for every declaration and any part of the code we defer to a later step.

Step 1. Writing the specification.
Sometimes you will be given a specification, like that in Figure 3.16, and you can begin to plan your strategy based on it. Much of the time, though, you will be given only a general description, as in the previous paragraph. You can fill in many of the details of the specification directly, but you might need to look up in a reference book things like constants or formulas, and you may need to decide the level of accuracy to maintain in your calculations. Until you have completed this step, you will be wasting time by trying to jump into writing the program.

Step 2. Creating a test plan.
Before beginning to write the program, we plan how we will test it. The first test case should be something that can be computed in one's head. We note that one of the simplest computations will occur when the exchange rate is 1.0. Then, if the price per liter is the same as the number of gallons per liter, the price per gallon should be \$1.00. We enter this set of numbers as the first line of the test plan in Figure 3.17. We enter the inverse case as the second line of the table: For a price of \$1.00 Canadian per liter, the U.S. price should be the same as the conversion factor for liters per gallon.

As a third test case, we enter an unacceptable conversion rate; we expect to see an error comment in response. We also must test the program's response to an invalid gas price, so we add a line with a negative price. This is called "black-box testing": the test values are drawn from the specification or from general knowledge of the kinds of data that often cause trouble. They could be chosen by someone with no knowledge of the code itself. (The code could be inside a black box.)

Finally, we enter a typical conversion rate and a typical price per liter, expecting to see an answer that is consistent with real prices. We use a hand calculator to compute the correct answer. We now have five lines in our test plan, which is enough for a simple program that tests for acceptable inputs.

Step 3. Starting the program.
First, we write the parts that remain the same from program to program, that is, the `#include` command

Rate	Can.$/liter	U.S.$/gallon
1.0	$0.26417	$1.00
1.0	$1.00	$3.78544
−0.001	$1.00	Error
1.0	−$0.87	Error
0.7412	$0.55	$1.543173

Figure 3.17. Test plan: Gas prices.

and the first and last lines of `main()` with the greeting message and termination code. The dots in the code represent the unfinished parts of the program that will be filled in by later coding steps.

```
#include <stdio.h>
...                // Space for #defines.
int main( void )
{
    ...            // Space for declarations.
    puts( "\n Gas Price Conversion Program \n" );
    ...            // Input statements.
    ...            // Computations.
    ...            // Output statements.
    return 0;
}
```

Step 4. Reading the data.
The exchange rate is a number with decimal places, so we declare a `double` variable to store it and put the declaration at the top of `main()`:

```
double US_per_Can;   // Exchange rate, $_US / $_Canadian
```

We decide to use a data validation loop to prompt for and read the current exchange rate, so we write down the parts of a `while` validation loop that always are the same, modifying the loop test, prompts, formats, and variable names, as appropriate, for our current application. This code goes into the `main()` program in the second spot marked by the dots.

```
printf( " Enter the exchange rate, $US per $Can: " );
scanf( "%lg", &US_per_Can );
while (US_per_Can < 0.0) {
    printf( " Re-enter; rate must be positive: " );
    scanf( "%lg", &US_per_Can );
}
```

When writing the calls on `scanf()`, remember to use `%lg` in the format for type `double` and put the ampersand before the variable name.

Next, we must read and validate the Canadian gas price. We declare a variable with a name that reminds us that the input is the price in Canadian dollars for a liter. We also remember to declare a variable for the price in U.S. dollars.

```
double C_liter;         // Canadian dollars per liter
double D_gallon;        // US dollars per gallon
```

Now we write another data validation loop, modifying the loop test, prompts, formats, and variable names, as needed. We write it in the program after the first loop.

```
printf( " Canadian price per liter: " );
scanf( "%lg", &C_liter );
while (C_liter < 0.0) {
    printf( " Re-enter; price must be positive: " );
    scanf( "%lg", &C_liter );
}
```

Step 5. Converting the gasoline price.
We defined a constant for liters per gallon at the very top of the program. Now we are ready to compute the price per gallon. Remember that we do not use an `=` sign or a semicolon in a `#define` command:

```
#define LITR_GAL   3.78544
```

We check the conversion formula given in the specification, making sure that the units do cancel out and leave us with dollars per gallon. Then we write the code for it and a `printf()` statement to print the answers:

```
D_gallon = C_liter * LITR_GAL * US_per_Can;
printf( "\n Canada: $%g  USA: $%g \n", C_liter, D_gallon );
```

Step 6. Testing the completed program.
The finished program is shown in Figure 3.18. Now we run the program and enter the first data set from the test plan. The results are

```
Gas Price Conversion Program
```

```
// ----------------------------------------------------------------
// Compute the equivalent prices for Canadian gas and U.S. gas
//
#include <stdio.h>
#define LITR_GAL   3.78544

int main( void )
{
    double US_per_Can;        // Exchange rate, $_US / $_Canadian
    double C_liter;           // Canadian dollars per liter
    double D_gallon;          // US dollars per gallon

    puts( "\n Gas Price Conversion Program \n" );
    printf( " Enter the exchange rate, $US per $Can: " );
    scanf( "%lg", &US_per_Can );
    while (US_per_Can < 0.0 ) {
        printf( " Re-enter; rate must be positive.\n " );
        scanf( "%lg", &US_per_Can );
    }

    printf( " Canadian price per liter: " );
    scanf( "%lg", &C_liter );
    while (C_liter < 0.0 ) {
        printf( " Re-enter; price must be positive: " );
        scanf( "%lg", &C_liter );
    }

    D_gallon = C_liter * LITR_GAL * US_per_Can;
    printf( "\n Canada: $%g  USA: $%g \n", C_liter, D_gallon );

    return 0;
}
```

Figure 3.18. Problem solution: Gas prices.

```
Enter the exchange rate, $US per $Can: 1.0
Canadian price per liter: .26417

Canada: $0.26417   USA: $1
```

Here is a test run using ordinary data (the last line in the test plan):

```
Gas Price Conversion Program

Enter the exchange rate, $US per $Can: .7412
Canadian price per liter: .55

Canada: $0.55  USA: $1.54317
```

Finally, we run the program twice again to test the error handling (the greeting message has been omitted):

```
Enter the exchange rate, $US per $Can:  -0.001
Re-enter; rate must be positive:  1.001
Canadian price per liter:  1.00

Canada: $1  USA: $3.78923
------------------------------------------
Enter the exchange rate, $US per $Can: 1.0
Canadian price per liter: -.087
Re-enter; price must be positive: 2.30

Canada: $2.3  USA: $8.70651
```

3.10 What You Should Remember

3.10.1 Major Concepts

The C language contains many facilities not mentioned yet, and those that have been introduced can be used in many more ways than demonstrated here. It can take years for a programmer to become truly expert in this language. In the face of this complexity, a beginner copes by starting with simple applications, mastering the basic concepts, and learning only the most important details. In this chapter, we have taken a preliminary look at the most basic and important elements of the C language and how they can be combined into a simple but complete program. These are grouped into related areas and summarized.

- **Language elements:**
  The C language contains elements (called commands, operators, and functions) that can be translated to groups of instructions built into a computer's hardware. It provides names or symbols for actions such as *add* (`+`) and *compare* (`==`) but not for complex activities like *solve this equation* or abstract activities such as *think*.
- Overall program structure:
  - An `#include` command is needed at the top of your program to allow your program to use system library functions.
  - A program must have a `main()` function.
  - The `main()` function starts with a series of declarations.
  - A series of statements follows the declarations.
  - A program should start with a statement that prints a greeting and gives instructions for the user.
- Types, objects, and declarations:
  - A program uses the computer's memory to create abstract models of real-world objects such as people, buildings, or numbers. Declarations are used to create and name these objects and may also give them initial values.
  - Declarations are grouped at the top of a program block.

  - Variables and `const` variables are objects; their names are used like nouns in English as the subjects and objects of actions.
  - An object has a name, a location, and a value. The compiler assigns the location for the object. We can give it a value by initializing it in the declaration or by assigning a value to it later. An object that has not been given a value is said to be *uninitialized* or to contain *garbage*.
  - Every object has a type. Types are like adjectives in English: The type of an object describes its properties and how it may be used. The three basic data types seen so far are `int`, `char`, and `double`.
  - A constant object must be initialized in the declaration and its value cannot be changed.
  - A `#define` command can be written at the top of a program to create a symbolic name for a literal constant.
- Simple statements:
  - The programmer combines operations and functions into a series of sentence-like statements that describe the actions to be carried out and the order in which they must happen.
  - Each statement tells the computer what to do next and what variables and constants to use in the process.
  - When a program is executed, the instructions in the program are run in order, from beginning to end. That sequential order can be modified by control instructions that allow for choices and repetition of blocks of statements.
  - The `scanf()` statements perform input. They let a human being communicate with a computer program. If a program requires the user to enter data, the input statement should be preceded by an output statement that displays a user prompt.
  - The `puts()` and `printf()` statements perform output. These statements let a computer program communicate with a human being.
  - An assignment statement can perform a calculation and store the result in a variable so that it can be used later. In general, calculations follow the basic rules of mathematics.
- Compound statements:
  - Statements can be grouped into blocks with braces.
  - The simple `if` statement is a conditional control statement. It has a condition and one block of code that is executed when the condition is true.
  - The `if...else` statement is a conditional control statement that has a condition and two blocks of code, a true clause and a false clause. When an `if...else` statement is executed, the condition is tested first and this determines which block of code is executed.
  - The `while` statement is a looping control statement. It has a condition and one block of code. The condition is tested first, and if the condition is true, the block is executed. Then the condition is retested. Execution and testing are repeated as long as the condition remains true.
  - The counting loops seen so far require that a counter variable be initialized prior to the loop and updated in some manner each time through the loop.

3.10.2 Programming Style

- A comment should follow each variable declaration to explain the purpose of the variable.
- Input data should be checked for validity: Garbage input causes garbage output.
- When writing your own programs, it often helps to model your work after a sample program that does a similar task. This makes it easier to find a combination of input, calculation, output, and control statements that are consistent with each other and work gracefully together.
- Indentation is important for readability. A programmer should adopt a meaningful indentation style and follow it consistently.
- Line up the words `if` and `else` with the `}` braces that close each clause in the same column. Indent all the statements in each clause. This assures a neat appearance and helps a reader find the end of the clause.[20]

[20]This layout scheme is advocated by *Recommended C Style and Coding Standards*, guidelines published in 1994 by experts at Bell Laboratories.

- If the clause to be executed in one part of an `if` statement is short and the other part is long, put the short clause first. This helps the reader see the whole picture easily. For example, suppose the program must test an input value to determine whether it is in the legal range. If it is legal, several statements will be used to process it. If it is illegal, the program will print an error comment and terminate execution, which takes only two lines of code. This program should be written with the error clause first (immediately following the `if` test) and the normal processing following the `else`.
- Where possible, avoid writing the same statement twice. This makes the program clearer and easier to debug. For example, do not put the same statement in both clauses of an `if` statement; put it before the `if` or after it.
- If both the `true` clause and the `false` clause are single statements, the entire `if` statement can be written on two lines without braces (`{` and `}`) to begin and end the clauses. If either clause is longer than one line, both clauses should be written with braces.

3.10.3 Sticky Points and Common Errors

- When you compile a program and get compile-time error comments, look at the first one first. One small error early in the program can produce dozens of error comments; fixing that single error often will make many comments go away.
- If you misspell a word, it becomes a different word in the eyes of the compiler. This is the first thing to check when you do not understand a compile-time or link-time error comment.
- An extra semicolon after the condition in an `if` or `while` statement will end the statement, and the code that should be within the `if` or `while` statement will be outside it. For example, suppose the `while` statement in the countdown program (Figure 3.14) were written incorrectly:

```
m = ITERATIONS;
while (m > 0);
{   printf( " %i.  \n", m );
    m = m - 1;
}
```

  The programmer will expect to see 20 lines printed on the page, with the first line numbered 20 and the last numbered 1. Instead, the semicolon ends the loop, which therefore has no body at all. The update line `m = m - 1;` is outside the loop and cannot be reached. The program will become an infinite loop because nothing *within the loop* will decrement the loop variable, `m`.
- A missing semicolon will not be discovered until the compiler begins working on the next line. It will tell you that there is an error, but give the wrong line number. Always check the previous line if you get a puzzling error comment about syntax.
- Quotation marks, braces, and comment-begin and -end marks come in pairs. If the second mark of a pair is omitted, the compiler will interpret all of the program up to the next closing mark as part of the comment or quote. It will produce odd and unpredictable error comments.
- If the output seems to make no sense, the first thing to check is whether the declared type of each variable on the output list matches the conversion specification used to print it. An error anywhere in an output format can affect everything after that on the line. If this does not correct the problem, add more diagnostic printouts to your program to display every data item calculated or read as input. If the input values are correct and the calculated values are wrong, check your formulas for precedence errors and check your function calls for errors.
- If the input values are wrong when you echo them, make sure you have ampersands before the names of the variables in the `scanf()` statement. Check also for type errors in the conversion specifiers in the format.

3.10.4 New and Revisited Vocabulary

These important terms and concepts were presented in this chapter:

lexical analysis	precision	assignment (=)
preprocessor command	modifier	expression
program block	identifier	operators (+, −, ∗, /)
declaration	local name	precedence
statement	undefined value	sequential execution
keyword	garbage	control statement
literal constant	object diagram	condition
symbolic constant	string	loop test
constant variable	prompt	loop body
variable	stream	loop counter
data value	buffer	data validation loop
data type	format	flow diagram
initializer	conversion specifier	black-box testing

The following C keywords and functions were presented in this chapter:

`#include`	`while` loop	`double`
`#define`	`\n` (newline character)	`int`
`main()`	`&` (address of)	`char`
`return` statement	`<stdio.h>`	`const`
`{...}` (block)	`stdin`	`scanf()`
`/*...` (comment)	`stdout`	`puts()`
`if...else` statement	`stderr`	`printf()`

3.10.5 Where to Find More Information

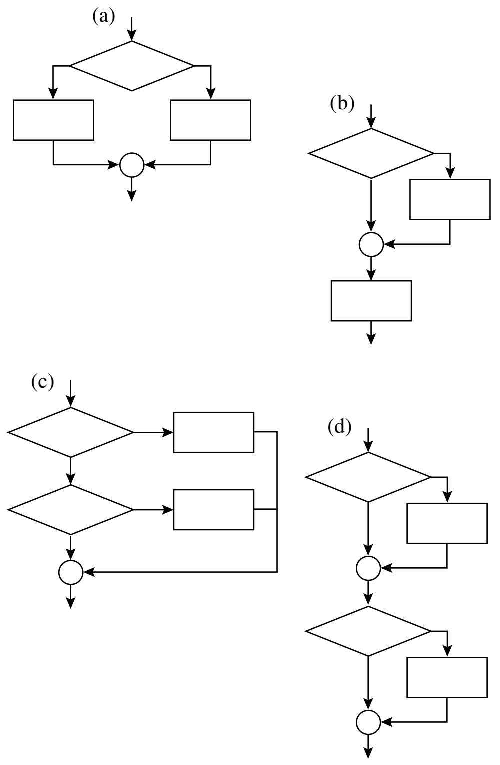
- The complete set of standard C keywords is given in Appendix ??.
- A full discussion of types `int` and `double` is given in Chapter 7 where we deal with representation, overflow, and the imprecise nature of floating point numbers.
- Additional simple and compound types are introduced in Chapters 10 through 14.
- Storage classes are introduced and used as follows:
  - `const`: Introduced here and used hereafter.
  - `volatile`: Introduced and used in Chapter 13.
  - `auto`: Introduced in Chapter 19, used throughout.
  - `extern`: Introduced in Chapter 19, used in Chapter 21.
  - `static`: Introduced in Chapter 19, used in Chapter 21.
  - `register`: Introduced in Chapter 19, not used in a program.
- Many more operators and the details of operator precedence and associativity are given in Chapter 4.
- Program design, with pseudocode and top-down development, is revisited in Chapters 5, 6, and 9.
- Operators associated with pointers are given in Chapter 11, those for bit manipulation are given in Chapter 15, those for accessing a `struct` in chapter 13, and the conditional operator is explained in Appendix ??.
- The third basic control structure, functions, will be introduced in Chapter 5
- The other two loop statements will be presented in Chapter 6.
- As other types of data are introduced in Chapters 7 through 11, more details about formats will be presented.
- The complexities of streams will be explored in Chapter 14.

3.11 Exercises

3.11.1 Self-Test Exercises

1. Four code fragments and four flow diagrams follow. Note that each diagram has two action boxes and one or two question boxes. All four represent distinct patterns of logic. Match each code fragment to the corresponding flow diagram and show how the code fits into the boxes.

```
(1)
if (radius < 0) {
volume = 0;
}
if (height < 0) {
volume = 0;

(2)
if (radius < 0) {
volume = 0;
}
else if (height < 0) {
volume = 0;
}

(3)
if (t < 1) {
v = t;
}
else {
v = 1;
}

(4)
if (rad > 100)
puts( "Too big" );
area = PI * rad * rad;
```



2. What is wrong with each of the following declarations?

   (a) `int d; a = 5;`
   (b) `doubel h;`
   (c) `int h = 2.5;`
   (d) `const double g;`
   (e) `character middle_initial;`
   (f) `double h = 2.0 * x;`
   (g) `integer k = 0;`
   (h) `char gender = "M";`

3. What conversion specifier do you use in an output format for an `int` variable? For a `double` variable? For a character?

4. Find the error in each of the following declarations.

   (a) `integer count;`
   (b) `real weight ;`
   (c) `int k; count;`
   (d) `character gender;`
   (e) `duble age;`

5. What happens when you omit the ampersand (address of) before a variable name in a `scanf()` statement? To find out, delete an ampersand in one of the sample programs. Try to compile and run the resulting program.
6. What happens when you type an ampersand before a variable name in a `printf()` statement? Try it.
7. What happens when you type a comment-begin mark but forget to type (or mistype) the matching comment-end mark? To find out, delete a comment-end mark in one of the sample programs and try to compile the result.
8. What happens when you type a semicolon after the closing parenthesis in a simple `if` statement? Try it.
9. What is wrong with each of the following `if` statements? They are supposed to identify and print out the middle value of three `double` values, `x`, `y`, and `z`.

(a)
```
if (x < y < z) printf( "y=%g", y );
else if (y < x < z) printf( "x=%g", x );
else printf( "z=%g", z );
```

(b)
```
if (x < y)
    if (y < z) printf( "y=%g", y );
    if (z < y) printf( "z=%g", z );
else
    if (x < z) printf( "x=%g", x );
    if (z < x) printf( "z=%g", z );
```

(c)
```
if (x > y)
{   if (x < z);
    printf( "x=%g", x );
    else printf( "z=%g", z );
}
else
{   if (y < z);
    printf( "y=%g", y );
    else printf( "z=%g", z );
}
```

3.11.2 Using Pencil and Paper

1. What does your compiler do when you misspell a keyword such as `while` or `else`? What happens when you misspell a function name such as `main()` or `scanf()`? Try it.
2. What happens when you type a semicolon after the closing parenthesis in a `while` statement? Try it.
3. What conversion specifier do you use in an input format for an `int` variable? For a `double` variable? For a character variable?
4. Find the error in each of the following preprocessor commands.
   (a) `#include <stdio>`
   (b) `#define NORMAL = 98.6`
   (c) `#include stdio.h`
   (d) `#define TOP 1,000`
   (e) `#define LOOPS 10;`
   (f) `#include <studio.h>`
5. Given the declarations on the first three lines that follow, find the error in each of the following statements:

```
int age, count;
double price, weight;
char gender;
```

(a) `scanf( "%g", &price );`
(b) `scanf( "%c", &gender );`
(c) `scanf( "%d", &weight );`
(d) `printf( "%i", &count );`
(e) `printf( "%lg", price );`
(f) `printf( "%c", gendre );`

6. Draw a flow diagram of the following program and use your diagram to trace its execution. What is the output?

```
#include <stdio.h>
int main( void )
{
    int k, m;
    k = 0;
    m = 1;
    while (k <= 3) {
        k = k + 1;
        m = m * 3;
    }
    printf( "k = %i m = %i \n", k, m );
    return 0;
}
```

7. In the following program, circle each error and show how to correct it:

```
#include (stdio.h)
int main (void)
{
    integer k;
    double x, y, z

    printf( Enter an integer: );
    scanf( "%i", k );
    printf( "Enter a double: );
    scanf( "%g", &X );
    printf( "Enter two integers: );
    scanf( "%lg", &y, &z );
    printf( "k= %i X= %g \n", &k, &x );
    printf( "y= %i z= %g \n" y, z );
}
```

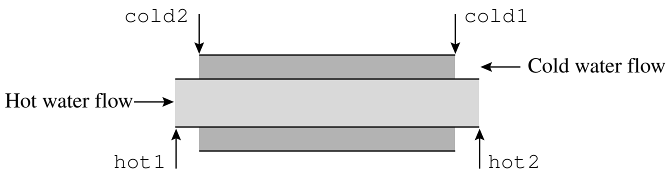
8. Draw a flow diagram for each set of statements, then trace their execution and show the output. Use these values for the variables: `w=3, x=1, y=2, z=0`.

(a)
```
if (x < y) z=1; if (x > w) z=2;
printf(" %i\n", z);
```
(b)
```
if (x < y) z=1; else if (x > w) z=2;
printf(" %i\n", z);
```
(c)
```
if (w < x) z=1; else if (w > y) z=2; else z=3;
printf(" %i \n", z);
```

3.11.3 Using the Computer

1. Heat transfer.

   Complete the following program for heat transfer that already has been designed and partially coded. The program analyzes the results from a heat transfer experiment in which hot water flows through a pipe and cold water flows in the opposite direction through a surrounding, concentric pipe. Thermocouples are placed at the beginning and end of both pipes to measure the temperature of the water coming in and going out, as diagrammed here.



Thermocouples are at both ends of both pipes

This is a standard engineering calculation. To calculate the heat transfer, we need the average hot temperature, the average cold temperature, and the change in temperature from input to output for both hot and cold pipes.

Your job is to start with the incomplete program in Figure 3.19, fill in the boxes according to the instructions that follow in each box, and make the program work. Use the following test plan.

Test Plan	**Data**				**Answers**			
	Inlets		**Outlets**		**Means**		**Differences**	
Test Objective	**Hot**	**Cold**	**Hot**	**Cold**	**Hot**	**Cold**	**Hot**	**Cold**
Easy to calculate	100	0	50	50	75	25	−50	50
Another legal input	120	35	100	50	110	42.5	−20	15

(a) Get a copy of the file `fex_heat.c`, which contains the partially completed program in Figure 3.19. Write program statements to implement the actions described in each of the comment boxes. Delete the existing comments and add informative ones of your own.

(b) Compile your code and test it using the test plan given above. Prepare proper documentation for the project including the source code, output from running the program on the test data, and hand calculations that prove the output is correct.

2. Temperature Conversion.

Complete a program for temperature conversion that has been designed and partially coded. The program will read in a temperature in degrees Fahrenheit, convert it into the equivalent temperature in degrees Celsius, and display this result. Your job is to start with the incomplete program in Figure 3.20 fill in the boxes according to the instructions given below in each box, and make the program work.

(a) Make an appropriate test plan for your program following the layout scheme below.

Test objective	Input Fahrenheit	Output Celsius
Easy-to-calculate input		
Another legal input		
Minimum legal input		
Out-of-range input		

(b) Start coding with a copy of the file `fex_temp.c`, which contains the partially completed program in Figure 3.20. Following the instructions given in each comment box, write program statements to implement the actions described. Delete the existing comments and add informative ones of your own. Compile and run your code, testing it according to the first line of your plan. Check the answer. If it is correct, print it out and go on to the rest of your test plan. If it is incorrect, fix it and test it again until it is correct. When the answers are correct, print out the program and its output on these tests, and hand them in with the test plan.

3. Sales Tax.

   The text website contains a complete program to solve the problem described below. However, the program will not compile and may have logical errors. Download the program and debug it, according to your specification. Hand in the debugged program and its output.

   Given the cost of a purchase, and a character 'T' for taxable or 'N' for non-taxable, compute the sales tax and total price. Let the tax rate be 6%.

4. Gasket area.

   A specification for a program that computes the area of a ring gasket is given in Figure 3.21. From this, the flow diagram in Figure 3.22 was constructed.

   (a) Develop a test plan for this program based on the problem specification in Figure 3.21.

```
// This program will compute heat characteristics of a concentric-pipe
// heat exchanger, including mean hot and cold temperatures, and the
// differences in temperatures of the ends of the pipes.

#include <stdio.h>
int main( void )
{
    double hot1;         // hot inlet temperature
    double hot2;         // hot outlet temperature
    double cold1;        // cold inlet temperature
    double cold2;        // cold outlet temperature
    double mean_hot;     // average of hot temperatures
    double mean_cold;    // average of cold temperatures
    double dthot;        // difference in hot temperatures
    double dtcold;       // difference in cold temperatures

    puts( "Heat Transfer Experiment" );

   Prompt user to enter four inlet and outlet temperatures.
   Use a separate prompt for each temperature you read.
   Use scanf() to read each and store in the appropriate variable.

   Calculate the mean of the hot temperatures (their sum divided by 2)
       and the mean of the cold temperatures.  Save each value in an
       appropriate variable.
   Calculate dthot (difference in hot temperatures = the hot outlet
       temperature minus the hot inlet temperature) and dtcold.
       Again save these in the appropriate variables.
   Put comments on these lines describing the calculations.

   Write printf() statements to echo the four input temperatures.
   Write printf() statements to print the four calculated numbers.
   Label each output clearly, so we can tell what it means.
   Use \n and spaces in your formats to make the output easy to read.

    return 0;
}
```

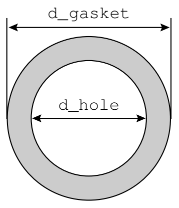
Figure 3.19. Sketch for the heat transfer program.

```
// ------------------------------------------------------------------
// This program will convert temperatures from degrees Fahrenheit into
// degrees Celsius.  Input temperature must be above absolute zero.

#include <stdio.h>

Define a constant for absolute zero in Fahrenheit (-459.67).

int main( void )
{
    double fahr;        // Temperature in Fahrenheit degrees
    double cels;        // Temperature in Celsius degrees

    puts( "\n Temperature Conversion Program" );

    Prompt the user to enter a temperature in degrees Fahrenheit.
    Use scanf() to read it and store it in the appropriate variable.
    Write a printf() statement that will echo the input.

    Test whether the input temperature is less than absolute zero.
    If so, print an error comment.
    If not, calculate the temperature in degrees Celsius according
    to the formula $C = \frac{5.0}{9.0} * (F - 32.0)$ and print the answer.
    Label the output clearly.
    Use newlines and spaces in your formats to make the output
    easy to read.

    return 0;
}
```

Figure 3.20. Sketch for the temperature conversion program.

(a) **Problem scope:** Write a program that will calculate the surface area of a ring gasket.

(b) **Inputs:** The outer diameter of the gasket, `d_gasket`, and the diameter of the hole in the center of the gasket, `d_hole`. Both diameters should be given in centimeters.



Formula:

$$\texttt{area} = \frac{\pi \times (\texttt{d\_gasket}^2 - \texttt{d\_hole}^2)}{4}$$

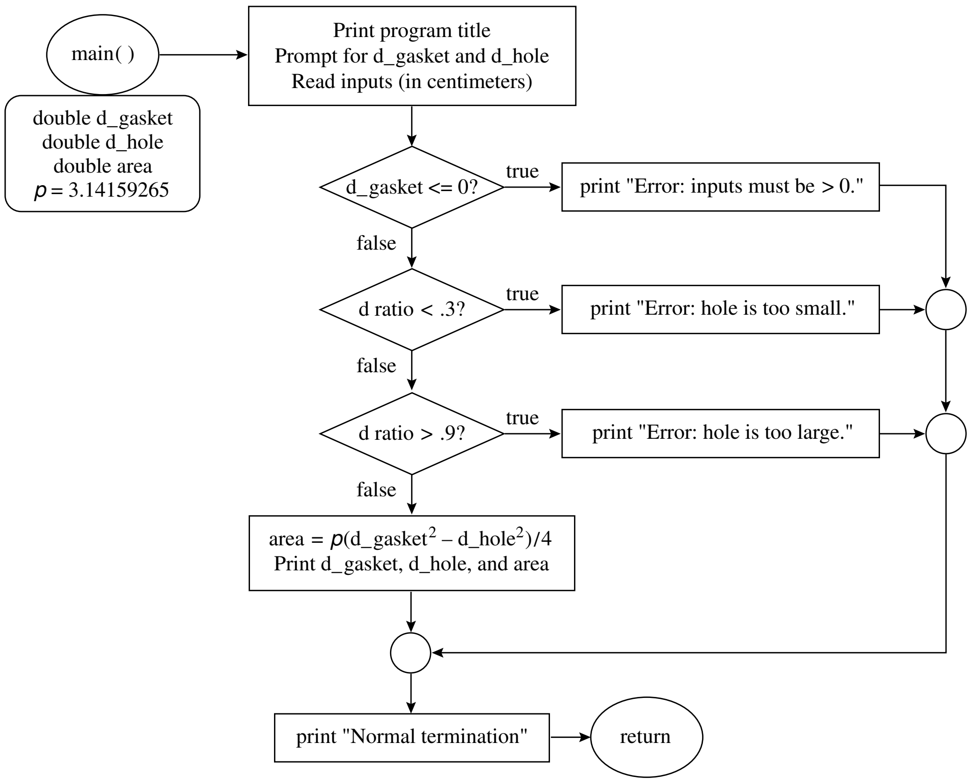
(c) **Limitations:** The value of `d_gasket` must be nonnegative. The ratio of `d_hole` to `d_gasket` must be greater than 0.3 and less than 0.9.

(d) **Constant:** π= 3.14159265

(e) **Output required:** The surface area of the gasket. (Echo the inputs also.)

(f) **Computational requirements:** All the inputs will be real numbers.

Figure 3.21. Problem specification: Gasket area.


main( )
double d_gasket
double d_hole
double area
p = 3.14159265
Print program title
Prompt for d_gasket and d_hole
Read inputs (in centimeters)
d_gasket <= 0?
true
print "Error: inputs must be > 0."
false
d ratio < .3?
true
print "Error: hole is too small."
false
d ratio > .9?
true
print "Error: hole is too large."
false
area = p(d_gasket2 – d_hole2)/4
Print d_gasket, d_hole, and area
print "Normal termination"
return

[.7]

Figure 3.22. Flow diagram for the gasket area program.

(b) Write a program based on the algorithm in Figure 3.22.
(c) Make sure all of your prompts and output labels are clear and easy to read.
(d) Test your program using the test plan you devised.
(e) Turn in the source code and output results from your test.

5. Skyscraper.

   The text website contains a complete program to solve the problem described below. However, the specification and test plan are missing and the program will not compile and may have logical errors. Write a specification and test plan, then download the program and debug it, according to your specification. Hand in your work plus the debugged program and its output.

   Assume that the ground floor of a skyscraper has 12-foot ceilings, while other floors of the building have 8-foot ceilings. Also, the thickness in between every floor is 3 feet. On top of the building is a 20-foot flagpole. If the building has N stories altogether, and N is given as an input to your program, calculate the height of a blinking red light at the top of the flagpole and print out this height.

6. Weight conversion.

   Write a program that will read a weight in pounds and convert it to grams. Print both the original weight and the converted value. There are 454 grams in a pound. Design and carry out a test plan for this program.

7. Distance conversion.

   Convert a distance from miles to kilometers. There are 5,280 feet per mile, 12 inches per foot, 2.54 centimeters per inch, and 100,000 centimeters per kilometer.

8. More Temperature Conversion. Revise the specification, test plan and program from problem 4 so that the user can enter a temperature in either Celsius or Fahrenheit, and the program will convert it to the other system. The inputs should be a temperature and a character, "C" for Celsius or "F" for Fahrenheit. Use an `if` statement to test which letter was entered, then perform the appropriate conversion. Echo the input and label the output properly. The formula for conversion from Celsius to Fahrenheit is

$$Fahrenheit = Celsius \times \frac{9}{5} + 32$$

9. Meaningful change.

   Write a program that will input the cost of an item from the user and output the amount of change due if the customer pays for the purchase with a $20 bill. What kinds of problems might this program have? (Think about unexpected inputs.) Design a test plan to detect these problems. Use an `if` statement to validate your data to prevent meaningless outputs.

10. A snow job.

    Your snow blower clears a swath 2 feet wide. Given the length and width of your rectangular driveway, calculate how many feet you will need to walk to clear away all the snow. Be sure to include the steps you take when you turn the snow blower around. Write a program that contains validation loops for the two input dimensions, echoes the input, and prints out the calculated distance.

11. Plusses and minuses.

    Your program will be used as part of a quality control system in a factory that makes metal rods of precise lengths. The current batch of rods is supposed to be 10 cm long. An automated measuring device measures the length of each rod as it comes off the production line. These measurements are automatically sent to a computer as input. Some rods are slightly shorter and some slightly longer than the target length. Your program must read the measurements (use `scanf()`). If the rod is too short, add it to a `short_total`; otherwise, add it to a `long_total`. Also, count it with either the `short_counter` or the `long_counter`. After each batch of 20 rods, print out your totals, counters, and the average length of the rods in the batch.