Part 1 Fundamentals

Chapter 1
Introduction to Computer Programming

1.1 History and Hardware

The process of using a machine to perform computations is almost as old as recorded history. The earliest such tool was the abacus—a device as common in China today as handheld calculators are in the United States. Both of these machines, however, require direct human involvement to be used. To add two numbers with an abacus requires the movement of beads on the device, while adding two numbers with a calculator requires that the operator push both the numbers and the addition operator keys.

The first recorded attempt at creating a programmable computing machine was by Charles Babbage in England in 1822. Ada Byron, daughter of the poet Lord Byron, developed a set of instructions that could, if the machine were ever built, be used to operate the machine. Although this machine, which Babbage called an analytical engine, was not successfully built in his lifetime, the concept of a programmable machine remained. It was partly realized in 1937 at Iowa State University by Dr. John V. Atanasoff and a graduate student named Clifford Berry, using electronic components. (See Figure 1.1.) The machine was known as the ABC, which stood for Atanasoff-Berry Computer, but required a human operator to manipulate external wiring to perform the desired operations. Thus, the goal of internally storing a replaceable set of instructions had still not been achieved.

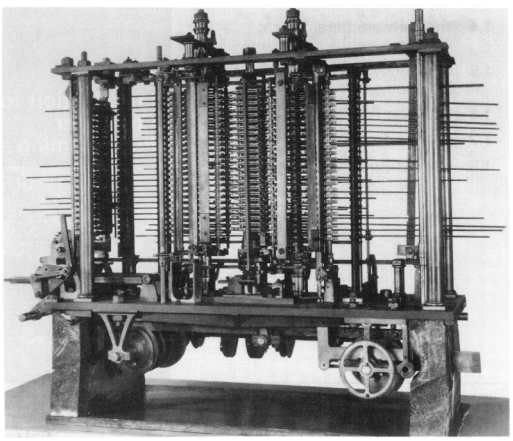

Figure 1.1 Charles Babbage's analytical engine

The outbreak of World War II led to a more concentrated development of the computer, beginning in late 1939. One of the pioneers of this work was Dr. John W. Mauchly of the Moore School of Engineering at the University of Pennsylvania. Dr. Mauchly, who had visited Dr. Atanasoff and seen his ABC machine, began working with J. Presper Eckert on a computer called ENIAC (Electrical Numerical Integrator and Computer). Funding for this project was provided by the U.S. government. One of the early functions performed by this machine was the calculation of

trajectories for ammunition fired from large guns. When completed in 1946, ENIAC contained 18,000 vacuum tubes, weighed approximately 30 tons, and could perform 5,000 additions or 360 multiplications in one second (see Figure 1.2).

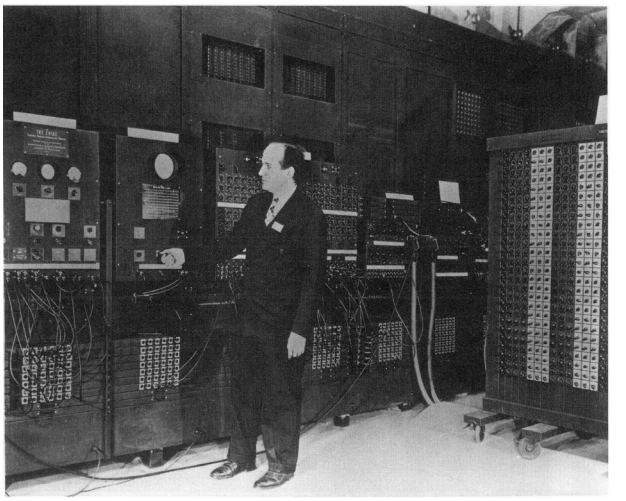

Figure 1.2 ENIAC

While work was progressing on ENIAC using vacuum tubes, work on a computer named the Mark I was being done at Harvard University using mechanical relay switches (see Figure 1.3). The Mark I was completed in 1944, but could only perform six multiplications in one second. Both of these machines, however, like the ABC, required external wiring to perform the desired operations.

Figure 1.3 Mark I

The final goal of a stored program computer, where instructions as well as data are stored internally within the machine, was achieved at Cambridge University in England on May 6, 1949, with the successful operation of the EDSAC (Electronic Delayed Storage Automatic Computer). In addition to performing calculations, the EDSAC could store both data and the instructions that directed the computer's operation. The EDSAC incorporated a form of memory, whose principles where developed by John Von Neumann, that allowed it to retrieve an instruction and then retrieve the data needed to carry out the instruction. This same design and operating principle is still used by the majority of computers manufactured today. The only features that have significantly changed are the sizes and speeds of the components used to make a computer, and the type of programs that are stored in it. Collectively, the components used to make a computer are referred to as **hardware**, while the programs are known as **software**.

Computer Hardware

Computers are constructed from physical components referred to as hardware. The purpose of this hardware is to facilitate the storage and processing of data under the direction of a stored program. If computer hardware could store data using the same symbols that humans do, the number 126, for example, would be stored using the symbols 1, 2, and 6. Similarly, the letter that we recognize as "A" would be stored using this same symbol. Unfortunately, a computer's internal components require a different number and letter representation. It is worthwhile to understand why computers cannot use our symbols and then see how numbers are represented within the machine. This will make it easier to understand the actual parts of a computer used to store and process this data.

Bits and Bytes

The smallest and most basic data item in a computer is a **bit**. Physically, a bit is really a switch that can be either open or closed. The convention we will follow is that the open position is represented by 0 and the closed position by 1.[1]

Historical Note

Binary ABC

For several years, Dr. Atanasoff agonized over the design of a computing machine to help his Iowa State University graduate students solve complex equations. He considered building a machine based on binary numbers-the most natural system to use with electromechanical equipment that had one of two easily recognizable states, on and off-but feared people would not use a machine that was not based on the familiar and comfortable decimal system. Finally, on a cold evening at a roadhouse in Illinois in 1937, he determined that it had to be done the simplest and least expensive way, with binary digits (bits). Over the next two years, he and graduate student Clifford Berry built the first electronic digital computer, called the ABC (Atanasoff-Berry Computer). Since that time the vast majority of computers have been binary machines.

A single bit that can represent the values 0 and 1, by itself, has limited usefulness. All computers, therefore, group a set number of bits together both for storage and transmission. The grouping of 8 bits to form a larger unit is an almost universal computer standard and is referred to as a **byte**. A single byte, where each of the 8 bits is either 0 or 1, can represent any one of 256 distinct patterns. These consist of the pattern 00000000 (all eight switches open) to the pattern 11111111 (all eight switches closed) and all possible combinations of 0s and 1s in between. Each of these patterns can be used to represent a letter of the alphabet, other single characters (a dollar sign, comma, etc.), a single digit, or numbers containing more than one digit. The collections of patterns consisting of 0s and 1s used to represent letters, single digits, and other single characters are called **character codes** (one such code, ASCII, is presented in Section 2.1).

Character codes are extremely useful for items such as names and addresses, and any text that must be processed. It almost never is used, however, for arithmetic data. There are two reasons for this. First, converting a decimal number into a character code requires an individual code for each digit. For large numbers, this can waste a computer's memory space. The more basic reason, however, is that the decimal numbering system, which is based on the number 10, is inherently not supported by a computer's internal hardware. Recall that a bit, which is a computer's basic memory component, can take on only one of two possible states, open and closed, which is represented as a 0 and a 1. This would indicate that a numbering system based on these two states makes more sense, and in fact, this is the case. Section 1.8 presents the most commonly used base-two numbering system.

The idea of a computer's internal numbering system differing from our decimal system should not come as a surprise. For example, you are probably already familiar with two other numbering systems, and can easily recognize the following:

1 This convention, unfortunately, is rather arbitrary, and you will frequently encounter the reverse correspondence where the open and closed positions are represented as 1 and 0, respectively.

- Roman numeral: XIV
- Hash mark system: /// /// ////

Components

All computers, from large supercomputers costing millions of dollars to smaller desktop personal computers costing hundreds of dollars, must perform a minimum set of tasks and provide the capability to:

1. Accept input, both data and instructions.
2. Display output, both textual and numerical.
3. Store data and instructions.
4. Perform arithmetic and logic operations on either the input or stored data.
5. Monitor, control, and direct the overall operation and sequencing of the system.

Figure 1.4 illustrates the computer components that support these capabilities and collectively form a computer's hardware.

Historical Note

The Turing Machine

In the 1930s and 1940s, Alan Mathison Turing (1912-1954) and others developed a theory that described what a computing machine should be able to do. Turing's theoretical machine, known as the Turing Machine, contains the minimum set of operations for solving programming problems. Turing had hoped to prove that all problems could be solved by a set of instructions given to such a hypothetical computer. What he succeeded in proving was that some problems cannot be solved by any machine, just as some problems cannot be solved by any person. Turing's work formed the foundation of computer theory before the first electronic computer was built. His contributions to the team that developed the critical code-breaking computers during World War II led directly to the practical implementation of his theories.

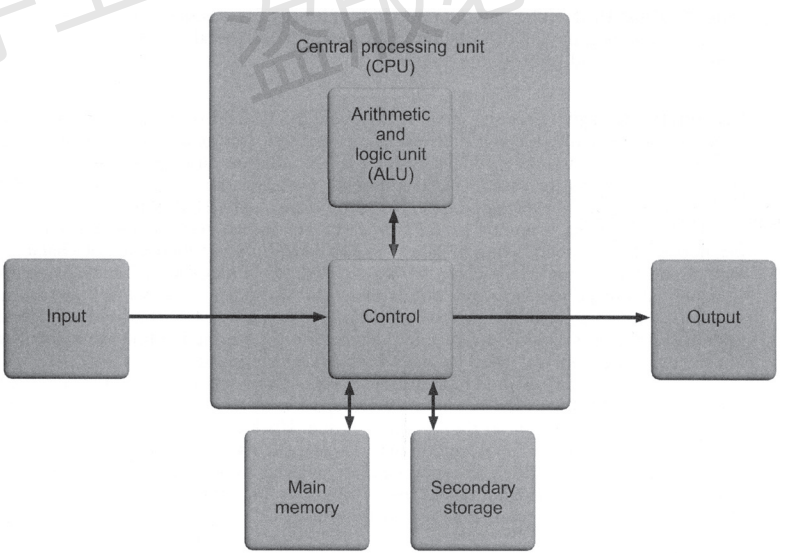

Figure 1.4 Basic hardware units of a computer

Main Memory Unit This unit stores data and instructions as a sequence of bytes. A program must reside in main memory if it is to operate the computer. Main memories combine 1 or more bytes into a single unit, referred to as a **word**. Although larger word sizes facilitate an increase in overall speed and capacity, this increase is achieved by an increase in the computer's complexity.

Early personal computers (PCs), such as the Apple IIe and Commodore machines, internally stored and transmitted words consisting of single bytes. The first IBM PCs used word sizes consisting of 2 bytes each, while more current Intel-based PCs store and process words consisting of 4 bytes each.

The arrangement of words in a computer's memory can be compared to the arrangement of suites in a large hotel, where each suite is made up of rooms of the same size. Just as each suite has a unique room number that allows patrons to locate and identify it, each word in a computer's memory has a unique numerical address. Like room numbers, word addresses are always positive unsigned whole numbers that are used for location and identification purposes. Also, like hotel rooms with connecting doors that form larger suites, words can be combined to form larger units to accommodate different-sized data types.

As a physical device, main memories are constructed as random access memory, or RAM. This means that every section of memory can be accessed randomly as quickly as any other section. Main memory is also **volatile**; whatever is stored in it is lost when the computer's power is turned off. Your programs and data are always stored in RAM when your program is being executed. The size of the computer's RAM is usually specified in terms of how many bytes of RAM are available to the user. PC memories currently start at 512 million bytes (denoted as megabytes or MB).

A second type of memory is read only memory, or ROM. ROM is nonvolatile; its contents are not lost when the power goes off. As such, ROM always contains fundamental instructions that cannot be lost or changed by the casual computer user. These instructions include those necessary for starting the computer's operation when the power is first turned on, and for holding any other instructions the manufacturer requires to be permanently accessible when the computer is operating.

Central Processing Unit The central processing unit (CPU) consists of two essential subunits, the control unit and the **Arithmetic and Logic Unit (ALU)**. The control unit directs and monitors the overall operation of the computer. It keeps track of where in memory the next instruction resides, issues the signals needed to both read data from and write data to other units in the system, and executes all instructions. The ALU performs all of the computations, such as addition, subtraction, comparisons, and so on, that a computer provides.

The CPU is the central element of a computer and its most expensive part. Currently, CPUs are constructed as a single microchip, which is referred to as a **microprocessor**. Figure 1.5 illustrates the size and internal structure of a state-of-the-art microprocessor chip used in current notebook computers.

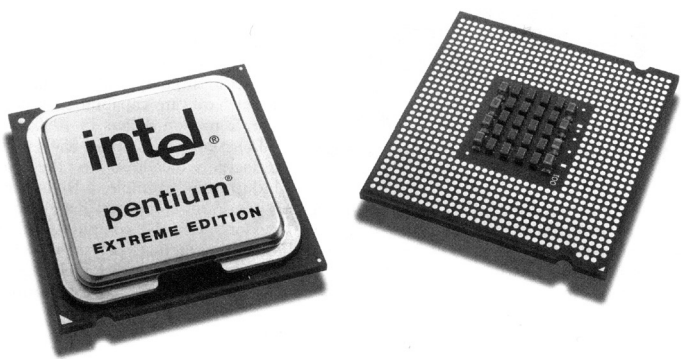

Figure 1.5 State-of-the-art Intel microprocessor

Input/Output Unit The input/output (I/O) unit provides access to the computer, allowing it to input and output data. It is the interface to which peripheral devices, such as keyboards, console screens, and printers, are attached.

Secondary Storage Because main RAM memory in large quantities is still relatively expensive and volatile, it is not practical as a permanent storage area for programs and data. Secondary or auxiliary storage devices are used for this purpose. Although data has been stored on punched cards, paper tape, and other media in the past, virtually all secondary storage is now done on magnetic tape, magnetic disks, and CD-ROMs.

The surfaces of magnetic tapes and disks are coated with a material that can be magnetized to store data. Current tapes are capable of storing thousands of characters per inch of tape, and a single tape may store up to hundreds of megabytes. Tapes, by nature, are a sequential storage media, which means that they allow data to be written or read in one sequential stream from beginning to end. Should you want to access a block of data in the middle of the tape, all preceding data on the tape must be scanned to find the block. Because of this, tapes are primarily used for mass backup of historical data.

A more convenient method of rapidly accessing stored data is provided by a **direct access storage device (DASD)**, which allows a computer to read or write any one file or program independent of its position on the storage medium. Until the recent advent of the CD, the most popular DASD was the magnetic disk. A **magnetic hard disk** consists of either a single rigid platter or several platters that spin together on a common spindle. A movable access arm positions the read and write mechanisms over, but not quite touching, the recordable surfaces. Such a configuration is shown in Figure 1.6.

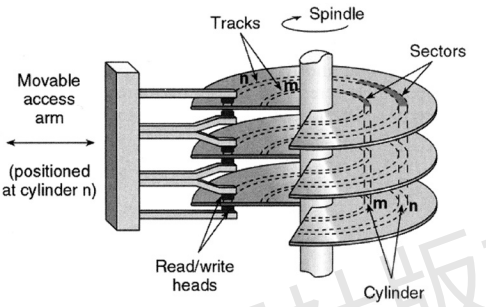

Figure 1.6 Internal structure of a hard disk drive

Initially, the most common magnetic disk storage device was the removable **floppy disk**. The most popular size for these is 3.5 inches in diameter, with a capacity of 1.44 megabytes. More recent removable disks, known as Zip disks, have capacities of 250 megabytes, with 650 to 700 megabyte compact disks (CDs) currently being the auxiliary storage devices of choice. Concurrent with the vast increase in storage capacity has been an equally significant increase in processing speed and a dramatic decrease in computer size and cost. Computer hardware capabilities that cost more than a million dollars in 1950 can now be purchased for less than $500. If the same reductions occurred in the automobile industry, for example, a Rolls-Royce could now be purchased for $10! The processing speeds of current computers have also increased by a factor of thousands over their 1950's predecessors, with the computational speeds of current computers being measured in both millions of instructions per second (MIPS) and billions of instructions per second (BIPS). For comparison, Figure 1.7 shows an early desktop IBM PC of the 1980s, while Figure 1.8 illustrates a current IBM notebook computer.

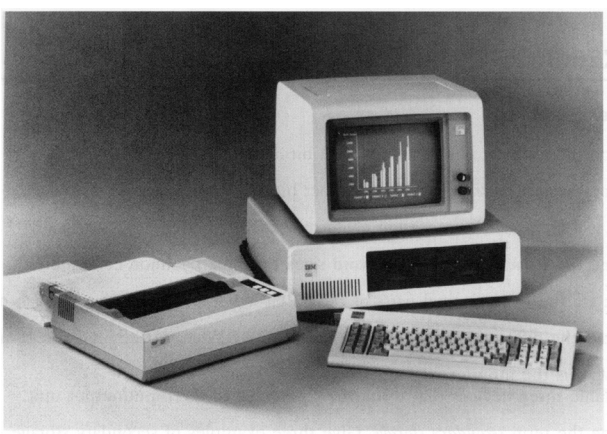

Figure 1.7 An original (1980s) IBM personal computer

Figure 1.8 A current IBM notebook computer

EXERCISES 1.1

Short Answer Questions

1. Define the term *bit*. What values can a bit assume?
2. Define the term *byte*. How many distinct bit patterns can a byte assume?
3. How is a byte used to represent characters in a computer?
4. Define the term *word*. Give the word sizes for some common computers.
5. What are the two principal parts of the CPU? What is the function of each part?
6. What is the difference between RAM and ROM? What do they have in common?
7. a. What is the input/output unit?
   b. Name three devices that would be connected to the input/output unit.
8. Define the term *secondary storage*. Give three examples of secondary storage.
9. What is the difference between sequential storage and direct access storage? What is the advantage of direct access storage?
10. Define the term *microprocessor*. Name three ways that microprocessors are used in everyday life.

1.2 Programming Languages

A computer, such as the modern notebook shown in Figure 1.8, is a machine made of physical components. Like other machines, such as an airplane, automobile, or lawn mower, a computer must be turned on and then driven, or controlled, to do the task it was meant to do. How this gets done is what distinguishes computers from other types of machinery.

In an automobile, for example, control is provided by the driver, who sits inside of and directs the car. In a computer, the driver is called a **program**. More formally, a **computer program** is a structured combination of data and instructions that is used to operate a computer and produce a specific result. Another term for a program or set of programs is **software**. We will use both terms interchangeably throughout this text.

Programming is the process of writing these instructions in a language that the computer can respond to and that other programmers can understand. The set of instructions that can be used to construct a program is called a **programming language**. Available programming languages come in a variety of forms and types. Each of these different forms and types was designed to make the programming process easier, to capitalize on a special feature of the hardware, or to meet a special requirement of an application. At a fundamental level, however, all programs must ultimately be converted into a machine language program, which is the only type of program that can actually operate a computer.

Historical Note

Ada Augusta Byron, Countess of Lovelace

Ada Byron, daughter of the Romantic poet Lord Byron, was a colleague of Charles Babbage in his attempt throughout the mid-1800s to build an analytical engine. It was Ada's task to develop the algorithms-solutions to problems in the form of step-by-step instructions-that would allow the engine to compute the values of mathematical functions. Babbage's machine was not successfully built in his lifetime, primarily because the technology of the time did not allow mechanical parts to be constructed with necessary tolerances. Nonetheless, Ada is recognized as the first computer programmer. She published a collection of notes that established the basis for computer programming. The modern Ada programming language is named in her honor.

Machine Language

An **executable program** is a program that can operate a computer. Such programs are always written as a sequence of binary numbers, which is a computer's internal language, and are also referred to as **machine language programs**. An example of a simple machine language program containing two instructions is:

```
11000000000000000001000000000010
11110000000000000010000000000011
```

Each sequence of binary numbers that constitutes a machine language instruction consists of, at a minimum, two parts: an instruction part and a data part. The instruction part, which is referred to as the **opcode** (short for operation code), is usually at the beginning of each binary number and tells the computer the operation to be performed, such as add, subtract, multiply, and so on. The remaining part of the number provides information about the data.

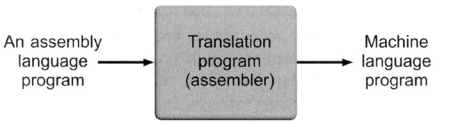

Figure 1.9 Assembly programs must be translated

Assembly Language

Because each class of computer, such as IBM PCs, Apple Macintoshes, and Hewlett-Packard computers, has its own particular machine language, it is very tedious and time consuming to write machine language programs. One of the first advances in programming was the substitution of word-like symbols, such as ADD, SUB, MUL, for the binary opcodes, and both decimal numbers and labels for memory addresses. For example, in the following set of instructions, word-like symbols are used to add two numbers (referred to as `first` and `second`), multiply the result by a third number known as `factor`, and store the result as answer:

```
LOAD first
ADD second
MUL factor
STORE answer
```

Programming languages that use this type of symbolic notation are referred to as **assembly languages**. Because computers can only execute machine language programs, the set of instructions contained within an assembly language program must be translated into a machine language program before it can be executed on a computer (Figure 1.9). Translator programs that translate assembly language programs into machine language programs are known as **assemblers**.

Low-and High-Level Languages

Both machine and assembly languages are classified as **low-level languages**. This is because both of these language types use instructions that are directly tied to one type of computer.[2]

As such, an assembly language program is limited in that it can only be used with the specific computer type for which the program is written. Such programs do, however, permit using special features of each particular computer type such as an IBM, Apple, or Hewlett-Packard computers, and generally execute at the fastest speed possible.

In contrast to low-level languages, a high-level language uses instructions that resemble human languages, such as English, and can be run on all computers, regardless of manufacturer. C, C++, Visual Basic, and Java are all high-level languages. For example, using C, the assembly language instructions used in the preceding section to add two numbers and multiply by a third number can be written as:

```
answer = (first + second) * factor;
```

Programs written in a computer language (high or low level) are referred to interchangeably as both **source programs** and **source code**. Once a program is written in a high-level language, it must also, like a low-level assembly program, be translated into the machine language of the computer on which it will be run. This translation can be accomplished in two ways.

When each statement in a high-level source program is translated individually and executed immediately upon translation, the programming language is called an **interpreted language**, and the program doing the translation is called an **interpreter**.

2 In actuality, the low-level language is defined for the processor around which the computer is constructed.

When all of the statements in a high-level source program are translated as a complete unit before any individual statement is executed, the programming language is called a **compiled language**. In this case, the program doing the translation is called a **compiler**. Both compiled and interpreted versions of a single language can exist, although typically one predominates. For example, although interpreted versions of C exist, C is predominantly a compiled language.

Figure 1.10 illustrates the process by which a C source program is compiled into a machine language executable program. (This figure does not show the essential steps of planning and analyzing the program design, which should take place before a line of code is typed. You'll learn how to plan a program design in Section 1.4.) As shown, the programmer types the source program using an editor program. This is effectively a word processing program that is part of the development environment supplied by the compiler.

Translation of the C source code into an executable program begins with the compiler. The output produced by the compiler is called an **object program**, which is a machine language version of the source code. Your source code may also make use of additional previously compiled code. This can consist of compiled code provided by another programmer or compiled code provided by a compiler, such as the mathematical code for finding a square root. In such cases this additional machine language code must be combined with the object program to create a final executable program. It is the task of the **linker** to accomplish these steps. The result of the linking process is a completed machine language program, containing all of the code required by your program, ready for execution. It is this final machine language program that is the executable program. The last step in the process is to load this machine language program into the computer's main memory for actual execution.

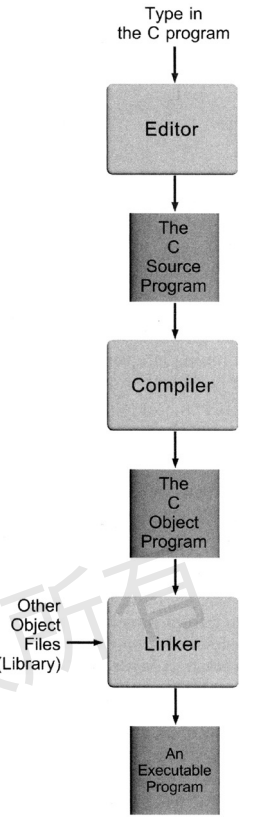

Figure 1.10 Programming steps to create an executable C program

Procedural and Object-Oriented Languages

High-level languages initially were all procedural languages. In a **procedural language**, the available instructions are used to create self-contained units, referred to as **procedures**. The purpose of a procedure is to accept data as input and transform the data in some manner to produce a specific result as an output. Effectively, each procedure moves the data one step closer to the final desired output along the path shown in Figure 1.11.

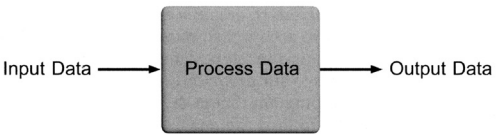

Figure 1.11 Basic procedural operations

The programming process illustrated in Figure 1.11 directly mirrors the input, processing, and output hardware units that are used to construct a computer. This was not accidental, because high-level procedural languages were initially designed to match and directly control corresponding hardware units. Each computer language tends to refer to its procedures by a different name. For example, in C, a procedure is referred to as a **function**; in Java, a procedure is referred to as a **method**; while in C++, the terms **method** and **function** are both used.

A well-written procedure consists of individual instructions that are grouped into specific internal structures according to well-established guidelines. (These guidelines are presented in Section 1.4.) Procedures conforming to these structure guidelines are known as **structured procedures**. High-level procedural languages, such as C, which effectively enforce adherence to these structures, are referred to as **structured languages**. Because of this, we will frequently use the term structured language to denote any high-level procedural languages, such as C, that enforces structured procedures.

Until the early 1990s, all new high-level languages were typically structured languages. More recently, a second approach, known as object orientation, has taken center stage. These languages, which consist of C++, Java, Visual Basic, and C#, are known as **object-oriented languages**.

One of the motivations for object-oriented languages was the development of graphical screens and support for graphical user interfaces (GUIs) capable of displaying multiple windows containing both graphical shapes and text. In such an environment, each window on the screen can be considered an object with associated characteristics, such as color, position, and size. An object-oriented program must first define the objects it will manipulate; this includes describing the general characteristics of the objects and then specifying specific procedures to manipulate them, such as changing size and position and transferring data between objects. Object-oriented languages, however, still retain and incorporate structured features in their procedures. Thus, C++, Java, and C# all used the basic structured procedure types found in C. In fact, C++ was specifically designed as an extension to C that included objects. These extensions are presented in Chapter 15.

C is unique in that it is a structured language that is still used extensively in the arena of these newer languages. In fact, C, in many cases, is preferred by professional programmers, especially when designing programs that handle extensive amounts of data, when dealing with programs that need to be developed quickly or require very targeted results, and for constructing intricate operating system programs. Thus, either as a foundation for learning the newer object-oriented programming languages, or as a programming language that can be used for many applications in its own right, learning C is a valuable addition to a programmer's knowledge base.

Application and System Software

Two logical categories of computer programs are application software and system software. **Application software** consists of programs written to perform particular tasks required by users. Most of the examples in this book would be considered application software.

System software is the collection of programs that must be readily available to any computer system to enable the computer to operate. In the early computer environments of the 1950s and 1960s, a user had to initially load the system software by hand to prepare the computer to do anything. This was done with rows of switches on a front panel. Those initial hand-entered commands were said to boot the computer, a term derived from the expression "pulling oneself up by the bootstraps." Today, the so-called **bootstrap loader** is internally contained in ROM and is a permanent, automatically executed component of the computer's system software.

Collectively, the set of system programs used to operate and control a computer is called the **operating system**. Tasks handled by modern operating systems include memory management; allocation of CPU time; control of input and output units such as the keyboard, screen, and printers; and the management of all secondary storage devices. Many operating systems handle very large programs, as well as multiple users concurrently, by dividing programs into segments, or pages, that are moved between the disk and memory as needed. Such operating systems permit more than one user to run a program on the computer, which gives each user the impression that the computer and peripherals are his or hers alone. This is referred to as a **multiuser system**. Additionally, many operating systems, including most windowed environments, permit each user to run multiple programs. Such operating systems are referred to as both **multiprogrammed** and **multitasking** systems.

The Development of C

The C language was initially developed in the 1970s at AT&T Bell Laboratories by Ken Thompson, Dennis Ritchie, and Brian Kernighan. It has an extensive set of capabilities that permits it to be written as a high-level structured language, while providing abilities to directly access the internal hardware of a computer. Known as the "professional programmer's language," C permits a programmer to "see into" a computer's memory and directly alter data that is stored within his or her own computer's memory. The standard that defines the C language is maintained by the American National Standards Institute (ANSI).

In the 1980s, Bjarne Stroustrup (working at AT&T) developed C++ as an extension to C. This new language contained the extensive set of capabilities provided by C, but was an object-oriented program. This close relationship

between C and C++ explains why C++ programs incorporate significant amounts of structured C-type code. Many C programs are now, in fact, written using C++ but are restricted to using only those features that are uniquely defined for the C language.

EXERCISES 1.2

Short Answer Questions

1. Define the following terms:
   a. computer program
   b. programming
   c. programming language
   d. high-level language
   e. low-level language
   f. machine language
   g. assembly language
   h. procedure-oriented language
   i. object-oriented language
   j. source program
   k. compiler
   l. assembler
2. a. Describe the difference between high- and low-level languages.
   b. Describe the difference between procedure and object-oriented languages.
3. Describe the difference between assemblers, interpreters, and compilers.
4. a. Assuming the following operation codes:
   `11000000` means add the 1st operand to the 2nd operand
   `10100000` means subtract the 1st operand from the 2nd operand
   `11110000` means multiply the 2nd operand by the 1st operand
   `11010000` means divide the 2nd operation by the 1st operand
   translate the following instructions into English:

Opcode	Address of 1st Operand	Address of 2nd Operand
11000000	000000000001	0000000000010
11110000	000000000010	0000000000011
10100000	000000000100	0000000000011
11010000	000000000101	0000000000011

   b. Assuming the following locations contain the following data, determine the result produced by the instructions listed in Question 4a.

Address	Initial Value (in Decimal) Stored at this Address
00000000001	5
00000000010	3
00000000011	6
00000000100	14
00000000101	4

5. Rewrite the machine-level instructions listed in Question 4a using assembly language notation. Use the symbolic names ADD, SUB, MUL, and DIV for addition, subtraction, multiplication, and division operations, respectively. In writing the instructions, use decimal values for the addresses.

6. Assuming that A = 10, B = 20, and C = .6, determine the numerical result of the following set of assembly language-type statements. For this exercise, assume that the LOAD instruction is equivalent to entering a value into the display of a calculator, and that ADD means add and MUL means multiply by.

```
LOAD A
ADD B
MUL C
```

1.3 Algorithms

Before a C program is written, the programmer must clearly understand what data is to be used, the desired result, and the steps to be used to produce this result. The steps selected to produce the result is referred to as an algorithm. More precisely, an algorithm is defined as a step-by-step sequence of instructions that describes how the data are to be processed to produce the desired outputs. In essence, an algorithm answers the question: "What method will you use to solve this problem?"

Only after we clearly understand the data that we will be using and select an algorithm (the specific steps required to produce the desired result) can we code the program. Seen in this light, programming is the translation of a selected algorithm into a language that the computer can use. In our case, we will always convert our steps into C language.

To illustrate an algorithm, we will consider a simple problem. Assume that a program must calculate the sum of all whole numbers from 1 through 100. Figure 1.12 illustrates two methods we could use to find the required sum. Each method constitutes an algorithm.

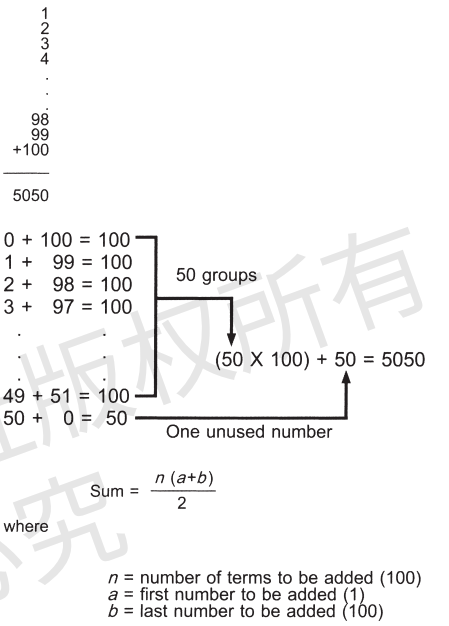

Figure 1.12 Summing the numbers 1 through 100

Historical Note

Al-Khwarizmi

About 825 A.D. one of the first great mathematicians, Mohammed Ibn Musa al-Khwarizmi, wrote a treatise called Ilm al-jabr wa'l muqabalah ("The Science of Reduction and Calculation"). The word algorithm is derived from al-Khwarismi's name, and our word algebra is derived from the word al-jabr in the title of his work.

Historical Note

The Young Gauss

German mathematical genius Johann Carl Fredrich Gauss (1777-1855) professed that he could "Reckon" before he could talk. When only 2 years old, he discovered an error in his father's business records.

One day in school, young Johann's teacher asked his class to add up the numbers between 1 and 100. To the chagrin of the teacher, who had thought the task would keep the class busy for a while, Gauss almost instantly wrote the number on his slate and exclaimed, "There it is!" He had reasoned that the series of numbers could be written forward and backward and added term-by-term to get 101 50 times. Thus, the sum was 100/2 x 101, and Gauss, at the age of 10 had discovered that $1 + 2 + \ldots + n = (n / 2)(n+1)$.

When solving such a problem, most people do not bother to list the possible alternatives in a detailed step-by-step manner (as shown in Figure 1.12) and then select one of the algorithms to solve the problem. That's because most people do not think algorithmically; they think heuristically. For example, if you had to change a flat tire on your car,

you would not think of all the steps required-you would simply change the tire or call someone else to do the job. This is an example of heuristic thinking.

Unfortunately, computers do not respond to heuristic commands. A general statement such as "add the numbers from 1 to 100" means nothing to a computer, because the computer can respond only to algorithmic commands written in an acceptable language such as C. To program a computer successfully, you must clearly understand this difference between algorithmic and heuristic commands. A computer is an "algorithm-responding" machine; it is not a "heuristic-responding" machine. You cannot tell a computer to change a tire or to add the numbers from 1 through 100. Instead, you must give the computer a detailed, step-by-step set of instructions that, collectively, forms an algorithm. For example, the set of instructions

```
Set n equal to 100
Set a equal to 1
Set b equal to 100
Calculate sum = n(a+ b)/2
Display the sum
```

form a detailed method, or algorithm, for determining the sum of the numbers from 1 through 100. Notice that these instructions are not a computer program. Unlike a program, which must be written in a language the computer can understand, an algorithm can be written or described in various ways.

When English phrases are used to describe the algorithm (the processing steps), as in this example, the description is called **pseudocode**. When mathematical equations are used, the description is called a **formula**. When pictures that employ specifically defined shapes are used, the description is called a **flowchart**. A flowchart provides a pictorial representation of the algorithm using the symbols shown in Figure 1.13. Figure 1.14 illustrates the use of these symbols in depicting an algorithm for determining the average of three numbers.

SYMBOL	NAME	DESCRIPTION
	Terminal	Indicates the beginning or end of an algorithm
	Input/Output	Indicates an input or output operation
	Process	Indicates computation or data manipulation
	Flow Lines	Connects the flowchart symbols and indicates the logic flow
	Decision	Indicates a decision point in an algorithm
	Loop	Indicates the initial, final, and increment values of a loop
	Predefined Process	Indicates a predefined process, as in calling a sorting process
	Connector	Indicates an entry to or exit from another part of the flowchart

Figure 1.13 Flowchart symbols

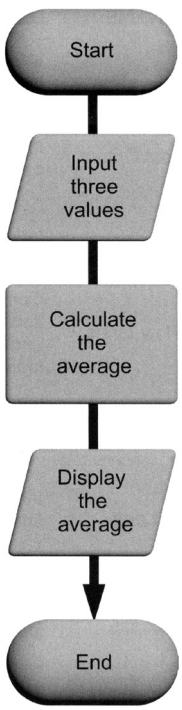

Figure 1.14 Flowchart for calculating the average of three numbers

Because flowcharts are cumbersome to revise, using pseudocode to express an algorithm's logic has gained increased acceptance among programmers. Unlike flowcharts, for which standard symbols are defined, there are no standard rules for constructing pseudocode. Any short English phrase may be used to describe an algorithm using

pseudocode. For example, the following is acceptable pseudocode describing the steps needed to compute the average of three numbers:

```
Input the three numbers into the computer
Calculate the average by adding the numbers and dividing the sum by three
Display the average
```

Only after the programmer selects an algorithm and understands the required steps can he or she write the algorithm using computer-language statements.

Once selected, an algorithm must be converted into a form that can be used by a computer. Converting an algorithm into a computer program, using a language such as C, is called **coding the algorithm** (see Figure 1.15). The program instructions resulting from this step are referred to as **program code**, or simply **code**, for short. The remainder of this text, starting in the next chapter, is devoted mostly to showing you how to develop algorithms and how to express these algorithms in C.

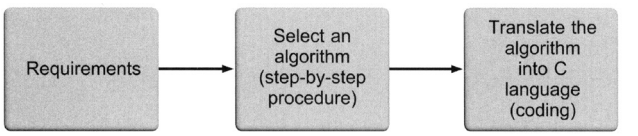

Figure 1.15 Coding an algorithm

EXERCISES 1.3

Short Answer Questions

1. Determine and list a step-by-step procedure to complete the following tasks. (Note: There is no single correct answer for each of these tasks. This exercise is designed to give you practice in converting heuristic commands into equivalent algorithms and making the shift between the thought processes involved in the two types of thinking.)
   a. Fix a flat tire
   b. Make a telephone call
   c. Go to the store and purchase a loaf of bread
   d. Roast a turkey
2. Determine and write an algorithm (list the steps) required to interchange the contents of two cups of liquid. Assume that a third cup is available to temporarily hold the contents of either cup. Each cup should be rinsed before any new liquid is poured into it.
3. Write a detailed set of step-by-step instructions, in English, to calculate the dollar amount of money in a piggybank that contains *h* half-dollars, *q* quarters, *n* nickels, *d* dimes, and *p* pennies.
4. Write a detailed set of step-by-step instructions, in English, to find the smallest number in a group of three integer numbers.
5. a. Write a detailed set of step-by-step instructions, in English, to calculate the fewest number of paper bills needed to pay a bill of amount TOTAL. For example, if TOTAL were $97, the bills, in U.S. currency, would consist of one $50 bill, two $20 bills, one $5 bill, and two $1 bills. Assume that only $100, $50, $20, $10, $5, and $1 bills are available if you are using U.S currency; otherwise, use the currency of the country you are living in.
   b. Assume that the bill is to be paid only in $1 bills.
6. a. Write an algorithm to locate the first occurrence of the name "Jones" in a list of names arranged in random order.
   b. Discuss how to improve your algorithm for Question 6a if the list of names was arranged in alphabetical order.
7. Determine and write an algorithm to determine the total occurrences of the letter e in any sentence.
8. Determine and write an algorithm to sort three numbers into ascending (from lowest to highest) order.

1.4 The Software Development Process

As modern society becomes more complex, so does its problems. Thus, problem solving has become a way of life. Issues such as solid waste disposal, global warming, international finance, pollution, and nuclear proliferation are relatively new, and solutions to these problems now challenge our best technology and human capabilities.

Most problem solutions require considerable planning and forethought if the solution is to be appropriate and efficient. This is true for most programming problems as well. For example, imagine trying to construct software for a cellular telephone network or creating an inventory management program for a department store by trial-and-error. Such a solution would be expensive at best, disastrous at worst, and practically unrealistic.

Each field of study has its own name for the systematic method used to design solutions to problems. In science this method is referred to as the **scientific method**, while in engineering disciplines the method is referred to as the **systems approach**. The technique used by professional software developers for understanding the problem that is being solved and for creating an effective and appropriate software solution is called the **software development process**. This process consists of the following four phases:

```
Phase I: Specify the program's requirements
Phase II: Design and development
      Step 1: Analyze the problem
      Step 2: Select an overall solution algorithm
      Step 3: Write the program
      Step 4: Test and correct the program
Phase III: Documentation
Phase IV: Maintenance
```

As shown in Figure 1.16, the first three phases frequently refine and interact with each other until a final design and program have been developed. Also, within the design and development phase itself you may discover that the problem has not been completely specified or analyzed, and further work in an earlier step is required to complete the program. Each of these phases are discussed in the following sections.

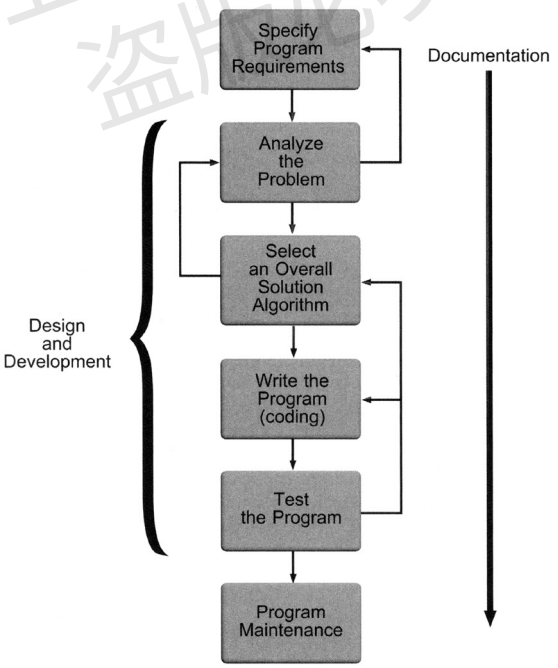

Figure 1.16 The software development process

Phase I: Specify the Program's Requirements

This phase begins with either a statement of a problem or a specific request for a program, which is referred to as a **program requirement**. Your task is to ensure that the program requirement is clearly stated and that you understand

what is to be achieved. For example, suppose you receive a brief e-mail from your supervisor that says: *We need a program to provide information about circles*.

This is not a clearly defined requirement. It does not specify a well-defined problem because it does not tell us exactly what information is required. It would be a major mistake to begin immediately writing a program to solve this poorly formulated problem. To clarify and define the problem statement, your first step would be to contact your supervisor to define exactly what information is to be produced (its outputs) and what data is to be provided (the inputs).

Suppose you do this and learn that what is really desired is a program to calculate and display the circumference of a circle when given the radius. Once you have what you think to be a clear statement of what is required, you may proceed to the next step.

Phase II: Design and Development

Once a program specification has been completed, the design and development phase, which forms the heart of the programming process, is begun. This phase consists of the following four steps:

Step 1: Analyze the Problem This step is required to ensure that the problem has, in fact, been clearly specified and understood, and to provide the necessary information for selecting an algorithm to solve the problem. The problem is clearly defined only when you understand

- The outputs that must be produced
- The input data that is required to create the desired outputs
- The formulas relating the inputs to the outputs

At the conclusion of the analysis, each of these three items must be clearly defined.

In performing an analysis, many new programmers prefer to determine the input data first, and then determine the desired outputs later, while professional programmers tend to work the other way around. It may seem odd to you to jump ahead to thinking about outputs first, but it is the outputs that the program is supposed to produce; they are the whole purpose of constructing a program in the first place. Knowing this goal and keeping it in mind will keep you focused on what is important in the program. However, if you are more comfortable initially determining the input data, do so.

Step 2: Select an Overall Solution Algorithm In this step, you determine and select an algorithm that will solve the problem. Sometimes determining this overall solution algorithm is quite easy, and sometimes it can be complex. For example, a program for determining the dollar value of the change in one's pocket or determining the area of a rectangle is quite simple. The construction of an inventory tracking and control system for a manufacturing company, however, is more complex. In these more complete cases, the initial solution algorithm is typically improved and refined until it specifies the complete solution in considerable detail. An example of this refinement is provided later in this section.

In its most general form, an overall solution algorithm applicable to most C programs is:

```
Get the inputs to the problem
Calculate the desired outputs
Report the results of the calculation
```

These three tasks are the primary responsibility of almost every problem, and we shall refer to this algorithm as the **Problem-Solver Algorithm**. A diagram of this algorithm is shown in Figure 1.17.

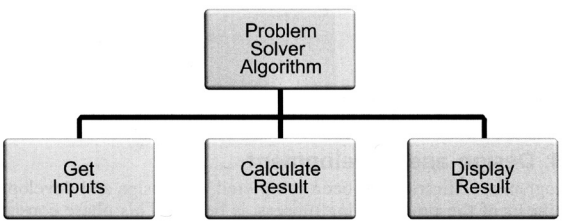

Figure 1.17 The Problem-Solver Algorithm

For example, if you were required to calculate the circumference of a circle with a given radius, the Problem-Solver Algorithm becomes:

```
Set a radius value, r
Calculate the circumference, C, using the formula C = 2πr
Display the calculated value for C
```

For small applications where only one or more calculations must be performed, the Problem-Solver Algorithm by itself is usually sufficient. For larger programs, however, you will have to refine the initial algorithm and organize it into smaller algorithms, with specifications for how these smaller algorithms will interface with each other. This is accomplished by the process of refinement, which is described next.

Refining the Algorithm For larger applications, the initial solution algorithm, which typically starts with the Problem-Solver Algorithm, must be refined and organized into smaller algorithms, with specifications for how the algorithms interface with each other. To achieve this goal, the description of the solution starts from the highest level (topmost) requirement and proceeds downward to the parts that must be constructed to achieve this requirement.

To make this more meaningful, suppose a computer program is required to track the number of parts in inventory. The required output for this program is a description of all parts carried in inventory and the number of units of each item in stock. The inputs are the initial inventory quantity of each part, the number of items sold, the number of items returned, and the number of items purchased.

The program designer could initially organize the overall solution algorithm for this program into the three Problem-Solver Algorithm sections illustrated on the bottom line in Figure 1.18. This is called a **first-level structure diagram** for the algorithm, because it represents the first attempt at an initial, but not yet sufficiently detailed, structure for a solution algorithm.

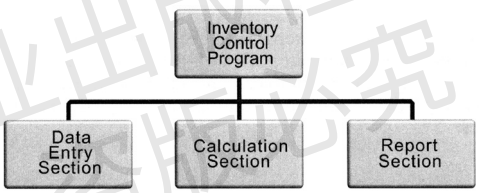

Figure 1.18 First-level structure diagram

Once you have developed an initial algorithm structure, you can then refine it until the tasks indicated in the boxes are completely defined. For example, both the data entry sections in Figure 1.18 would be further refined to specify provisions for entering the data. Because it is the system designer's responsibility to plan for contingencies and human error, provisions must also be made for changing incorrect data after an entry has been made and for deleting a previously entered value altogether. Similar subdivisions for the report section can also be made.

Figure 1.19 illustrates a second-level structure diagram for an inventory tracking system that includes these further refinements.

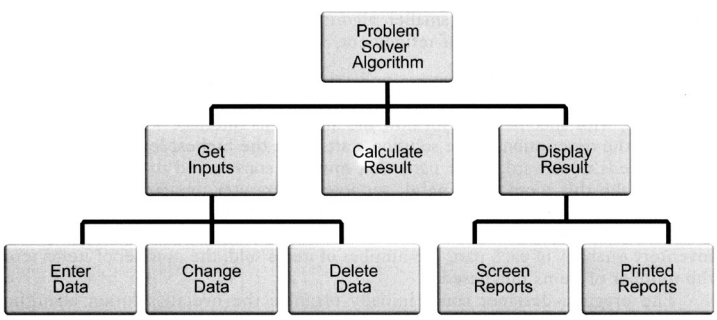

Figure 1.19 Second-level refinement structure diagram

Notice that the design takes on a treelike structure where the levels branch out as we move from the top of the structure to the bottom. When the design is complete, each task designated in the lower boxes typically represents simple algorithms that are used by algorithms higher up in the structure. This type of algorithm development is referred to as a **top-down algorithm development**; it starts at the topmost level and proceeds to develop more and more detailed algorithms as it proceeds to the final set of algorithms.

Step 3: Write the Program Writing the program involves translating the chosen solution algorithm into a C language computer program. This step is also referred to as coding the algorithm.

If the analysis and solution steps have been correctly performed, writing the program becomes rather mechanical in nature. In a well-designed program, the statements making up the program will, however, conform to certain well-defined structures that have been defined in the solution step. These structures control how the program executes and consist of the following types:

1. Sequence
2. Selection
3. Iteration
4. Invocation

Sequence defines the order in which instructions are executed by the program. It specifies that instructions will be executed in the order they appear in the code, unless specifically altered by one of the other structures.

Selection provides the capability to make a choice between different instructions, depending on the result of some condition. For example, the value of a number can be checked before a division is performed. If the number is zero, the division will not be performed and a warning message will be issued to the user; otherwise, a division will take place. Selection capabilities and how they are coded in C are presented in Chapter 4.

Repetition, which is also referred to as **looping** and **iteration**, provides the ability for the same operation to be repeated based on the value of a condition. For example, grades might be repeatedly entered and added until a negative grade is entered. In this case the entry of a negative grade is the condition that signifies the end of the repetitive input and addition of grades. At that point a calculation of an average for all the grades entered could be performed. Repetition capabilities and how they are coded in C are presented in Chapter 5.

Invocation involves invoking, or summoning into action, specific sections of code as they are needed. Invocation capabilities and how they are coded in C are presented in Chapters 6 and 7.

Step 4: Test and Correct the Program The purpose of testing is to verify that a program works correctly and actually fulfills its requirements. In theory, testing would reveal all existing program errors (in computer terminology, a program error is called a bug[3]). In practice, this would require checking all possible combinations of statement execution. Because of the time and effort required, this is usually an impossible goal except for extremely simple programs. (We illustrate why this is generally an impossible goal in Section 4.8.)

Exhaustive testing is simply not feasible for most programs. For this reason, different methods of testing have evolved. At its most basic level, testing involves a conscious effort to ensure that a program works correctly and produces meaningful results. You must think carefully about what the test is meant to achieve and the data you will use in the test. If testing reveals an error (bug), you must initiate the process of debugging, which includes locating, correcting, and verifying the correction. It is important to realize that *although testing may reveal the presence of an error, it does not necessarily indicate the absence of one.* Thus, *the fact that a test revealed one bug does not indicate that another one is not lurking somewhere else in the program*.

To catch and correct errors in a program, it is important to develop a set of test data that determines whether the program gives correct answers. In fact, an accepted step in formal software testing is to plan the test procedures and create meaningful test data before writing the code. This helps the programmer be more objective about what the

3 The derivation of this term is rather interesting. When a program stopped running on the MARK I at Harvard University in September 1945, Grace Hopper traced the malfunction to a dead insect that had gotten into the electrical circuits. She recorded the incident in her logbook at 15:45 hours as "Relay #70.... (moth) in relay. First actual case of bug being found."

program must do, because it essentially circumvents any subconscious temptation after coding to avoid test data that will cause the program to fail. The tests should examine every possible situation under which a program will be used. This means testing with data within a reasonable range, using data that are at the limits of what is acceptable, and testing with invalid data that the program should detect and report as invalid data. In fact, developing good verifications tests and data for sophisticated problems can be more difficult than writing the program code itself.

Phase III: Documentation

In practice, most programmers forget many of the details of their own programs a few months after they have finished working on them. If they or other programmers must subsequently make modifications to the program, much valuable time can be lost figuring out just how the original program works. Good documentation prevents this from occurring.

So much work becomes useless or lost, and so many tasks must be repeated because of inadequate documentation that it could be argued that documenting your work is the most important step in problem solving. Actually, many of the critical documents are created during the analysis, design, coding, and testing steps. Completing the documentation requires collecting these documents, adding additional material, and presenting it in a form that is most useful to you and your organization.

Although not everybody classifies them in the same way, there are essentially six documents for every problem solution:

1. The requirements statement
2. A description of the algorithms that were coded
3. Comments within the code itself
4. A description of modification and changes made over time
5. Sample test runs, which include the inputs used for each run and the output obtained from the run
6. A user's manual, which is a detailed explanation of how to use the program

"Putting yourself in the shoes" of a member of a large organization's team that might use your work-anyone from the secretary to the programmer to the user-should help you to make the content and design of the important documentation clear. The documentation phase formally begins in phase I and continues into the maintenance phase.

Phase IV: Maintenance

This phase is concerned with ongoing correction of problems, revisions to meet changing needs, and adding new features. Maintenance is often the major effort, the primary source of revenue, and the longest lasting of the engineering phases. While development may take days or months, maintenance may continue for years or decades. The better the documentation is, the more efficiently this phase can be performed and the happier the customer and user will be.

Figure 1.20 illustrates the relative proportion of time attributable to maintenance as compared to development and design.

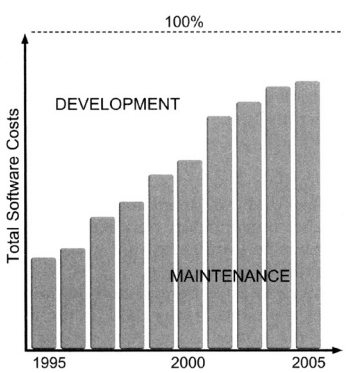

Figure 1.20 Maintenance is the dominant software cost

Figure 1.20 shows that the maintenance of existing programs currently accounts for approximately 70 percent of all programming costs. Students generally find this strange because they are accustomed to solving a problem and moving on to a different one. Commercial and scientific fields, however, do not operate this way. In these fields, one application or idea is typically built on a previous one and may require months or years of work. This is especially true in programming. Once an application is written, which may take weeks or months, maintenance may continue for years as new features are needed. Advances in technology such as communication, networking, fiber optics, and new graphical displays constantly demand updated software products.

How easily a program can be maintained (corrected, modified, or enhanced) is related to the ease with which the program can be read and understood. This, as you have learned, depends on the care with which the program was designed and the availability of high-quality documentation.

Backup

Although not part of the formal software development process, making and keeping backup copies of your work when writing a program is critical. In the course of revising a program, you may easily change the current working version of a program beyond recognition. Backup copies allow you to recover the last stage of work with a minimum of effort. The final working version of a useful program should be backed up at least twice. In this regard, another useful programming proverb is "Backup is unimportant if you don't mind starting all over again." The three fundamental rules of maintaining a program are:

1. *backup!*
2. *Backup!!*
3. *BACKUP!!!*

Many organizations keep at least one backup on site where it can be easily retrieved, and another backup copy either in a fireproof safe or at a remote location.

EXERCISES 1.4

Short Answer Questions

1. List and describe the four principle phases in the software development process.
2. An e-mail note from your department head, Ms. R. Karp, says: "Solve our inventory problems."
   a. What's your first task?
   b. How would you accomplish this task?
3. Suppose you are asked to create a C program that calculates the amount, in dollars, contained in a piggybank. The bank contains half-dollars, quarters, dimes, nickels, and pennies. Do not attempt to code this program. Instead, answer the following questions:
   a. For this programming problem, how many outputs are required?
   b. How many inputs does this problem have?
   c. Determine a formula for converting the input items into output items.
   d. Test the formula written for Question 3c using the following sample data: half dollars = 0, quarters = 17, dimes = 24, nickels = 16, pennies = 12.
4. Suppose you are asked to create a C program that calculates the value of distance, in miles, given the relationship *distance = rate * elapsed time*. Do not attempt to code this program. Instead, answer the following questions:
   a. For this programming problem, how many outputs are required?
   b. How many inputs does this problem have?
   c. What is the formula for converting the input items into output items?
   d. Test the formula written for Question 4c using the following sample data: rate = 55 miles per hour and elapsed time = 2.5 hours.
   e. How must the formula you determined in Question 4c be modified if the elapsed time is given in minutes instead of hours?

5. Suppose you are asked to create a C program that determines the value of Ergies, given the relationship
   Ergies = Fergies * Lergies
   (*Note*: All terms in this formula are for fictitious items.) Do not attempt to code this program. Instead, answer the following questions:
   a. For this programming problem, how many outputs are required?
   b. How many inputs does this problem have?
   c. What is the formula for converting the input items into output items?
   d. Test the formula written for Question 5c using the following sample data: Fergies = 14.65 and Lergies = 4.
6. Suppose you are asked to create a C program that displays the following name and address:
   Mr. J. Swanson
   63 Seminole Way
   Dumont, NJ 07030
   Do not attempt to code this program. Instead, answer the following questions:
   a. For this programming problem, how many lines of output are required?
   b. How many inputs does this problem have?
   c. Is there a formula for converting the input items into output items? Why or why not?
7. Suppose you are asked to create a C program that determines the distance a car has traveled after 10 seconds assuming the car is initially traveling at 88 feet per second (this equals 60 miles per hour) and the driver applies the brakes to uniformly decelerate at a rate of 12 feet/sec^2. Use the fact that distance = $st - (1/2)dt^2$, where s is the initial speed of the car, d is the deceleration, and t is the elapsed time. Do not attempt to code this program. Instead, answer the following questions:
   a. For this programming problem, how many outputs are required?
   b. How many inputs does this problem have?
   c. What is the formula for converting the input items into output items?
   d. Test the algorithm written for Question 7c using the data given in the problem.
   Many programming problems require both initial negotiations with a client and additional information that the client desires before programming even begins. Questions 8 through 13 are intended to familiarize you with some of the situations you may face, especially if you do any freelance programming.
8. Many people requesting a program or system for the first time consider coding to be the most important aspect of program development. They feel that they know what they need and think that the programmer can begin coding with minimal analysis time. As a programmer, what pitfalls can you envision in working with such people?
9. Many people requesting a program try to contract with programmers for a fixed fee (total amount to be paid is fixed in advance). What is the advantage to the user in having this arrangement? What is the advantage to the programmer in having this arrangement? What are some disadvantages to both user and programmer in this arrangement?
10. Many freelance programmers prefer to work on an hourly rate basis. Why do you think this is so? Under what conditions would it be advantageous for a programmer to give a client a fixed price for the programming effort?
11. People who have experience hiring programmers generally expect a clearly written statement of programming work to be done, including a complete description of what the program will do, delivery dates, payment schedules, and testing requirements. What is the advantage to the user in requiring this? What is the advantage to a programmer in working under this arrangement? What disadvantages does this arrangement pose for both client and programmer?
12. A computer store has asked you to write a sales recording program. The store is open six days a week, each sale requires an average of 100 characters, and, on average, the store makes 15 sales per day. The owner of the store has asked you to determine how many characters must be kept for all sales records for a two-year period. What estimate would you provide to the owner?

13. You are creating a sales recording system for a new client. Each sale input to the system requires that the operator type in a description of the item sold, the name and address of the firm buying the item, the value of the item, and a code for the person making the trade. This information consists of a maximum of 300 characters. The client will have to hire one or more data entry people to convert their existing manual records into computer records. They currently have more than 5,000 sales records, and have asked you how much time it would take to enter these records into your new system. What estimate would you provide? (*Hint*: To solve this problem you must make an assumption about the number of words per minute that an average typist can type and the average number of characters per word.)

1.5 Case Study: Design and Development

In this section we apply the design and development phase (phase II) of the software development process to the following program requirements specification:

The circumference, C, of a circle is given by the formula C = 2πr, where π is the constant 3.1416 (accurate to four decimal places), and r is the radius of the circle. Using this information, write a C program to calculate the circumference of a circle that has a 2-inch radius.

Step 1: Analyze the Problem

This step verifies that the program specification is complete and that we have a complete understanding of what is required.

a. Determine the Desired Outputs In determining the desired outputs, concentrate on words in the requirements statement such as calculate, print, determine, find, or compare. For our sample program requirement above, the key phrase is "to calculate the circumference of a circle." This identifies an output item (the circumference of a circle). Since there are no other such phrases in the problem, only one output item is required.

b. Determine the Input Items After clearly identifying the desired outputs, you must identify all input items. (If you are more comfortable identifying inputs before outputs, do so.) It is essential, at this stage, to distinguish between input items and input values. An input item is the name of an input quantity, while an input value is a specific number or quantity that can be used as the input item. For example, in our program requirement, the input item is the radius of the circle. Although this input item has a specific numerical value in this problem (the known quantity, which has a value of 2), actual input values are generally not important at this stage.

Input values are not needed at this point because the relationship between inputs and outputs is typically independent of specific input values. The formula depends on knowing the output and input items and whether there are any special constrains. Notice that the relationship between the two, which will be expressed by a formula, is correct regardless of any specific values assigned to the input items (unless there are specific constraints where the formula is valid). Although we cannot produce an actual numerical value for the output item without specific values for the input items, the correct relationship between inputs and outputs is expressed by the formulas relating the two, which are listed next.

c. List the Formulas Relating the Inputs to the Outputs The final step is to determine how to create the output from the inputs. This is answered by knowing the formulas between output and input quantities. In this case the relationship is provided by the single formula C = 2πr, where C is the output item and r is the input item. Again, be aware that the formula does not require listing specific input values; it simply identifies the relationship between input and output items.

If you are unsure how to obtain the required outputs from the given inputs, you need a clearer requirements statement. In other words, you need to obtain more information about the problem.

d. Perform a Hand Calculation Having listed the formulas, the next step is to check the formula using specific input values. Performing a manual calculation, either by hand or using a calculator, helps to ensure that you really do understand the problem. An added feature of doing a manual calculation is that the results can be used later to verify program operation in the testing step. Then, when the final program is used with other data, you will have established a degree of confidence that a correct result is being calculated.

It is in this step that we need specific input values that can be assigned to the input items used by the formula to produce the desired output. For this problem one input value is given: a radius of 2 inches. Substituting this value into the formula, we obtain a circumference = 2 (3.1416)(2) = 12.5664 inches for the circle.

Step 2: Select an Overall Solution Algorithm

The general Problem-Solver Algorithm presented in the previous section is:

```
Get the inputs to the problem
Calculate the desired outputs
Report the results of the calculation
```

For determining the circumference of a circle, this algorithm becomes:

```
Set the radius value to 2
Calculate the circumference, C, using the formula C = 2πr
Display the calculated value for C
```

Step 3: Write the Program

Program 1.1 illustrates the code for the algorithm for determining the circumference of a circle having a radius of 2. Because we have not yet introduced the C language, the program will be unfamiliar to you; however, you should be able to understand what the key individual lines are accomplishing. To help you in this, line numbers have been provided. These line numbers are never part of a C program, but will always be inserted for easily identifying individual C statements.

In this case the program follows a sequential order, where each statement is executed in strict sequential order, one after another. To help you understand it, however, comments, which are the text between the /* and */ symbols, have been included in lines 5, 7, and 8. Although all of the program lines are explained fully in the next chapter, for now, pay attention to these three lines and use the comments to relate these lines to the algorithm selected in the analysis step.

In line 5 the names `radius` and `circumference` are defined for use by the program. The program attaches no significance to these names (the names `r` and `c`, `x` and `y`, or `in` and `out`, for example, could just as well have been defined), but names that are more descriptive and have some meaning to the actual problem should always be chosen. In line 7 a value is assigned to the name `radius`, while in line 8 a value is computed for the item named `circumference`. Finally, in line 9 the value of the circumference is printed.

Program 1.1

```
 1  #include <stdio.h>
 2
 3  int main()
 4  {
 5    float radius, circumference;  /* declare an input and output item */
 6
 7    radius = 2.0;    /* set a value for the radius */
 8    circumference = 2.0 * 3.1416 * radius;  /* calculate the circumference */
 9    printf("The circumference of the circle is %f\n", circumference);
10
11    return 0;
12  }
```

When Program 1.1 is executed, the following output is produced:

```
The circumference of the circle is 12.566400
```

Step 4: Test and Correct the Program

Because only one calculation is performed by the program, testing Program 1.1 really means verifying that the single output is correct. Because the output agrees with our prior hand calculation, we can now use the program to calculate the circumference of circles with different radii and have confidence in the results.

EXERCISES 1.5

Short Answer Questions

1. Assuming a programmer used the names `rad` and `cir` in Program 1.1 instead of the names `radius` and `circumference`, determine the lines that would have to be modified in the program and rewrite these lines to reflect the change in names.
2. Assume that the area of a circle is required in addition to its circumference. What modifications do you think would have to be made to Program 1.1 to accommodate this additional requirement?
3. You have been asked to write a program that converts 86 kilometers to miles, where the formula for converting kilometers to miles is *miles* = *.625*(*kilometers*). Do not attempt to code this program. Instead, answer the following questions:
   a. Write a complete program requirements specification for this problem.
   b. Determine the output required by the program.
   c. Determine how many inputs the program will have.
   d. What is the formula for converting the input items into output items?
   e. Do a hand calculation for the given input value.
   f. Provide a solution algorithm for this problem.
4. In 1627, Manhattan Island was sold to Dutch settlers for approximately $24. Suppose you are asked to create a program that answers this question: If the proceeds of that sale had been deposited in a Dutch bank paying 5 percent interest, compounded annually, what is the principal balance at the end of 2006? The program should display the following output, with an actual value replacing the x's: Balance as of December 31, 2006, is xxxxxx. Do not attempt to code this program. Instead, answer the following questions:
   a. Write a complete program requirements specification for this problem.
   b. Determine the output required by the program.
   c. Determine how many inputs the program will have.
   d. Determine a formula for converting the input items into output items.
   e. Do a hand calculation for the given input value.
   f. Provide a solution algorithm for this problem.
5. Suppose you are asked to create a C program that calculates and displays the weekly gross pay and net pay of two individuals. The first individual is paid an hourly rate of $8.43, and the second individual is paid an hourly rate of $5.67. Both individuals have 20 percent of their gross pay withheld for income tax purposes, and both pay 2 percent of their gross pay, before taxes, for medical benefits. Do not attempt to code this program. Instead, answer the following questions.
   a. For this programming problem, how many outputs are required?
   b. What is the formula for converting the input items into output items?
   c. How many inputs does this problem have?
   d. Test the formula written for Question 5c using the following sample data: The first person works 40 hours during the week and the second person works 35 hours.
   e. Provide a solution algorithm for this problem.
6. The formula for the standard normal deviate, z, used in statistical applications is:

   $z = (X - \mu)/\sigma$

   where μ refers to a mean value and σ to a standard deviation. Suppose you are asked to write a program that calculates and displays the value of the standard normal deviate where $X = 85.3$, $\mu = 80$, and $\sigma = 4$. Do not attempt to code this program. Instead, answer the following questions:
   a. For this programming problem, how many outputs are required?
   b. How many inputs does this problem have?
   c. What is the formula for converting the input items into output items?

d. Test the formula written for Question 6c using the data given in the problem.

e. Provide a solution algorithm for this problem.

7. The equation describing exponential growth is:

   $y = e^x$

   Suppose you are asked to create a program that calculates the value of y. Do not attempt to code this program. Instead, answer the following questions:

   a. For this programming problem, how many outputs are required?

   b. How many inputs does this problem have?

   c. What is the formula converting the input items into output items?

   d. Test the formula written for Question 7c assuming $e = 2.718$ and $x = 10$.

   e. Provide a solution algorithm for this problem.

1.6 Common Programming Errors

Part of learning any programming language is making the elementary mistakes that other beginning programmers have made before you. Each language has its own set of common programming errors waiting for the unwary, and these mistakes can be quite frustrating. The most common errors associated with the material presented in this chapter are as follows:

1. Rushing to write and execute a program without spending sufficient time learning about the problem or designing an appropriate algorithm. In this regard, it is worthwhile to remember the programming proverb: "*It is impossible to construct a successful program for a problem that is not fully understood*." A similar and equally valuable proverb is "*The sooner you start programming an application, the longer it usually takes to debug and complete*."
2. Forgetting to back up a program. Almost all new programmers make this mistake until they lose a program that has taken considerable time to code.
3. Not understanding that computers respond only to explicitly defined algorithms. Telling a computer to add a group of numbers is quite different than telling a friend to add the numbers. The computer must be given the precise instructions for doing the addition in a programming language.

1.7 Chapter Summary

1. The first attempt at creating a self-operating computational machine was attempted by Charles Babbage in 1822. The concept became a reality with the Atanasoff-Berry Computer built in 1937 at Iowa State University, which was the first computer to use a binary numbering scheme to store and manipulate data.

   The earliest large-scale digital computer was the ENIAC, built in 1946 at the Moore School of Engineering at the University of Pennsylvania. This machine, however, required external wiring to direct its operation.

   The first computer to employ the concept of a stored program was the EDSAC, built at Cambridge University in England. The operating principals used in the design of this machine, developed by the mathematician John Von Neumann, are still used by the majority of computers manufactured today.
2. The physical components used in constructing a computer are called hardware.
3. The programs used to operate a computer are referred to as software.
4. Programming languages come in a variety of forms and types. Machine language programs, also known as executable programs, contain the binary codes that can be executed by a computer. Assembly languages permit the use of symbolic names for mathematical operations and memory addresses. Programs written in assembly languages must be converted to machine language, using translator programs called assemblers, before the programs can be executed. Assembly and machine languages are referred to as low-level languages.
5. Compiler and interpreter languages are referred to as high-level languages. This means that they are written using instructions that resemble a written language, such as English, and can be run on a variety of computer types.

Compiler languages require a compiler to translate the program into a machine language form, while interpreter languages require an interpreter to do the translation.

6. An algorithm is a step-by-step sequence of instructions that must terminate and describes how to perform an operation to produce the desired output.
7. The software development procedure consists of the following four phases:
   - Specification of the program's requirements
   - Design and development
   - Documentation
   - Maintenance
8. The design and development phase consists of four well-defined steps:
   - Analyze the problem
   - Select an overall solution algorithm
   - Write the program
   - Test and correct the program
9. Writing (or coding) a program consists of translating the solution algorithm into a computer language such as C.
10. Four fundamental control structures used in writing a program are:
    - Sequence
    - Selection
    - Iteration
    - Invocation
11. Although making copies of a program is not formally a part of the software development process, it is essential that you always keep at least one copy of a program. This copy is referred to as a backup copy, or backup, for short.

1.8 Chapter Appendix: Numerical Storage Codes

The most commonly used code for storing integers within a computer's memory unit is called **two's complement**. This code associates each integer (both positive and negative) with a specific pattern of bits.

A bit pattern, such as 10001011, that is considered to be a numeric value, is referred to as a **binary number**. The easiest way to determine the decimal value of an integer binary number is to first construct a simple device called a **value box**. Figure 1.21 illustrates a value box for a single byte. (For convenience, in the following discussion we will assume words consisting of a single byte, although the procedure also applies to larger-sized words.)

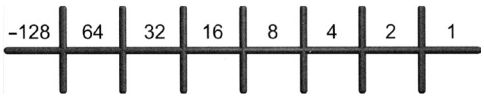

Figure 1.21 An 8-bit value box for two's complement conversion

Mathematically, each value in the box illustrated in Figure 1.21 represents an increasing power of two. Since two's complement numbers must be capable of representing both positive and negative integers, the leftmost position, in addition to having the largest absolute magnitude, also has a negative sign.

Conversion of any binary number, for example 10001101, into its equivalent decimal value simply requires inserting the bit pattern in the value box and adding the only those values that have a 1 under them. Thus, as illustrated in Figure 1.22, the bit pattern 10001101 represents the integer number −115.

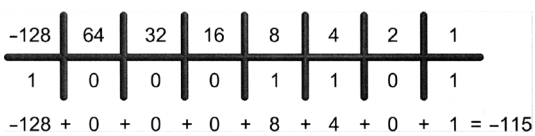

Figure 1.22 Converting 10001101 to a base 10 number

Reviewing the value box shows that any binary number with a leading 1 represents a negative number, and any bit pattern with a leading 0 represents a positive number. The value box can also be used in reverse, to convert a base 10 integer number into its equivalent binary bit pattern. Some conversions, in fact, can be made by inspection. For example, the base 10 number −125 is obtained by adding 3 to −128. Thus, the binary representation of −125 is 10000011, which equals −128 + 2 + 1. Similarly, the two's complement representation of the number 40 is 00101000, which is 32 plus 8.

Although the value box conversion method is deceptively simple, the method is directly related to the mathematical basis of two's complement binary numbers. The original name of the two's complement binary code was the **weighted-sign binary code**, which correlates directly to the value box. As the name "weighted sign" implies, each bit position has a weight, or value, of two raised to a power and a sign. The signs of all bits except the leftmost bit are positive and the sign of the leftmost or most significant bit is negative.