M01_SAVI1346_07_SB_C01.fm Page 1 Friday, January 4, 2008 5:01 PM

1 Introduction to Computers and C++ Programming

The whole of the development and operation of analysis are now capable of being executed by machinery.... As soon as an Analytical Engine exists, it will necessarily guide the future course of science.

CHARLES BABBAGE (1792–1871)

INTRODUCTION

In this chapter we describe the basic components of a computer, as well as the basic technique for designing and writing a program. We then show you a sample C++ program and describe how it works.

1.1 COMPUTER SYSTEMS

software

hardware

A set of instructions for a computer to follow is called a program. The collection of programs used by a computer is referred to as the **software** for that computer. The actual physical machines that make up a computer installation are referred to as **hardware.** As we will see, the hardware for a computer is conceptually very simple. However, computers now come with a large array of software to aid in the task of programming. This software includes editors, translators, and managers of various sorts. The resulting environment is a complicated and powerful system. In this book we are concerned almost exclusively with software, but a brief overview of how the hardware is organized will be useful.

Hardware

PCs, workstations, and mainframes

There are three main classes of computers: *PCs*, *workstations*, and *mainframes*. A **PC (personal computer)** is a relatively small computer designed to be used by one person at a time. Most home computers are PCs, but PCs are also widely used in business, industry, and science. A **workstation** is essentially a larger and more powerful PC. You can think of it as an "industrial-strength" PC. A **mainframe** is an even larger computer that typically requires some support staff and generally is shared by more than one user. The distinctions between PCs, workstations, and mainframes are not precise, but the terms are commonly used and do convey some very general information about a computer.

network

A **network** consists of a number of computers connected, so that they may share resources such as printers, and may share information. A network might contain a number of workstations and one or more mainframes, as well as shared devices such as printers.

For our purposes in learning programming, it will not matter whether you are working on a PC, a mainframe, or a workstation. The basic configuration of the computer, as we will view it, is the same for all three types of computers.

M01_SAVI1346_07_SB_C01.fm Page 3 Friday, January 4, 2008 5:01 PM

The hardware for most computer systems is organized as shown in Display 1.1. The computer can be thought of as having five main components: the *input device(s)*, the *output device(s)*, the *processor* (also called the *CPU*, for *central processing unit*), the *main memory*, and the *secondary memory*. The processor, main memory, and sometimes even secondary memory are normally housed in a single cabinet. The processor and main memory form the heart of a computer and can be thought of as an integrated unit. Other components connect to the main memory and operate under the direction of the processor. The arrows in Display 1.1 indicate the direction of information flow.

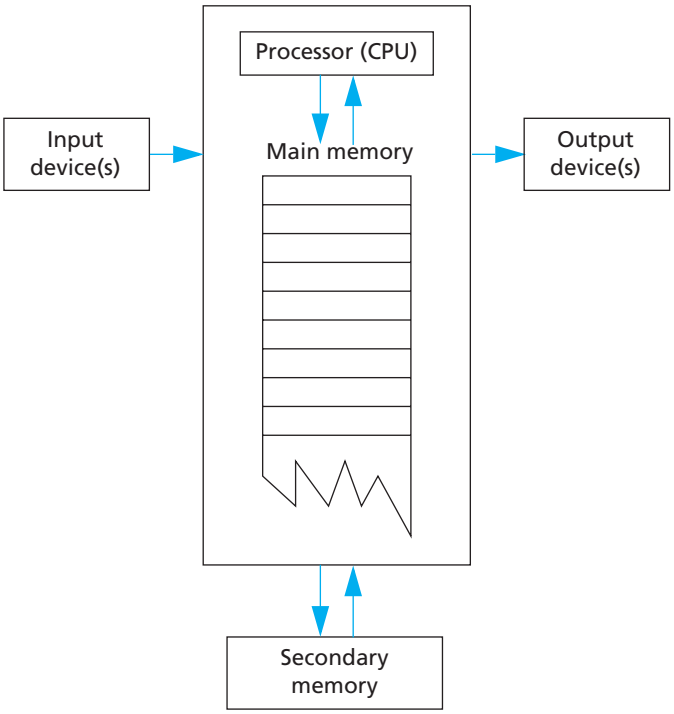
input devices

An **input device** is any device that allows a person to communicate information to the computer. Your primary input devices are likely to be a keyboard and a mouse.

output devices

An **output device** is anything that allows the computer to communicate information to you. The most common output device is a display screen, referred to as a *monitor*. Quite often, there is more than one output device. For example, in addition to the monitor, your computer probably is connected to a printer for producing output on paper. The keyboard and monitor are sometimes thought of as a single unit called a *terminal*.

DISPLAY 1.1 Main Components of a Computer



M01_SAVI1346_07_SB_C01.fm Page 4 Friday, January 4, 2008 5:01 PM

In order to store input and to have the equivalent of scratch paper for performing calculations, computers are provided with *memory*. The program that the computer executes is also stored in this memory. A computer has two forms of memory, called *main memory* and *secondary memory*. The program that is being executed is kept in main memory, and main memory is, as the name implies, the most important memory. **Main memory** consists of a long list of numbered locations called *memory locations*; the number of memory locations varies from one computer to another, ranging from a few thousand to many millions, and sometimes even into the billions. Each memory location contains a string of zeros and ones. The contents of these locations can change. Hence, you can think of each memory location as a tiny blackboard on which the computer can write and erase. In most computers, all memory locations contain the same number of zero/one digits. A digit that can assume only the values zero or one is called a **binary digit** or a **bit.** The memory locations in most computers contain eight bits (or some multiple of eight bits). An eight-bit portion of memory is called a **byte,** so we can refer to these numbered memory locations as *bytes*. To rephrase the situation, you can think of the computer's main memory as a long list of numbered memory locations called *bytes*. The number that identifies a byte is called its **address.** A data item, such as a number or a letter, can be stored in one of these bytes, and the address of the byte is then used to find the data item when it is needed.

main memory

bit

byte

address

If the computer needs to deal with a data item (such as a large number) that is too large to fit in a single byte, it will use several adjacent bytes to hold the data item. In this case the entire chunk of memory that holds the data item is still called a **memory location.** The address of the first of the bytes that make up this memory location is used as the address for this larger memory location. Thus, as a practical matter, you can think of the computer's main memory as a long list of memory locations of *varying sizes*. The size of each of these locations is expressed in bytes and the address of the first byte is used as the address (name) of that memory location. Display 1.2 shows a picture of a hypothetical computer's main memory. The sizes of the memory locations are not fixed, but can change when a new program is run on the computer.

memory location

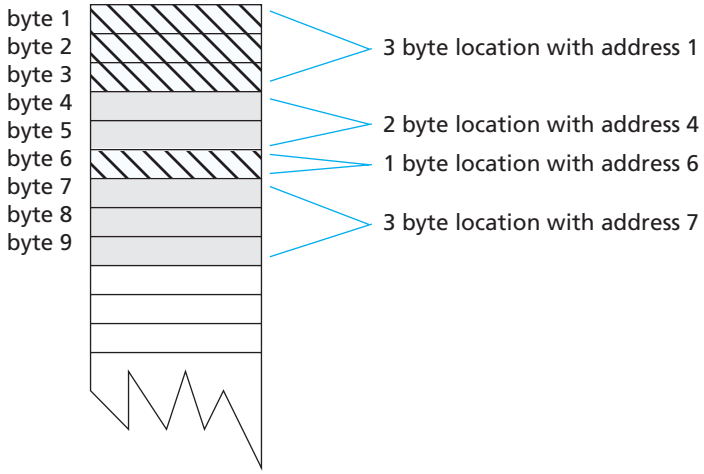
The fact that the information in a computer's memory is represented as zeros and ones need not be of great concern to you when programming in C++

Bytes and Addresses

Main memory is divided into numbered locations called **bytes.** The number associated with a byte is called its **address.** A group of consecutive bytes is used as the location for a data item, such as a number or letter. The address of the first byte in the group is used as the address of this larger memory location.

M01_SAVI1346_07_SB_C01.fm Page 5 Friday, January 4, 2008 5:01 PM

DISPLAY 1.2 Memory Locations and Bytes



(or in most other programming languages). There is, however, one point about this use of zeros and ones that will concern us as soon as we start to write programs. The computer needs to interpret these strings of zeros and ones as numbers, letters, instructions, or other types of information. The computer performs these interpretations automatically according to certain coding schemes. A different code is used for each different type of item that is stored in the computer's memory: one code for letters, another for whole numbers, another for fractions, another for instructions, and so on. For example, in one commonly used set of codes, 01000001 is the code for the letter A and also for the number 65. In order to know what the string 01000001 in a particular location stands for, the computer must keep track of which code is currently being used for that location. Fortunately, the programmer seldom needs to be concerned with such codes and can safely reason as though the locations actually contained letters, numbers, or whatever is desired.

Why Eight?

A **byte** is a memory location that can hold eight bits. What is so special about eight? Why not ten bits? There are two reasons why eight is special. First, eight is a power of 2. (8 is 2^3.) Since computers use bits, which have only two possible values, powers of two are more convenient than powers of 10. Second, it turns out that eight bits (one byte) are required to code a single character (such as a letter or other keyboard symbol).

M01_SAVI1346_07_SB_C01.fm Page 6 Friday, January 4, 2008 5:01 PM

The memory we have been discussing up until now is the main memory. Without its main memory, a computer can do nothing. However, main memory is only used while the computer is actually following the instructions in a program. The computer also has another form of memory called *secondary memory* or *secondary storage*. (The words *memory* and *storage* are exact synonyms in this context.) **Secondary memory** is the memory that is used for keeping a permanent record of information after (and before) the computer is used. Some alternative terms that are commonly used to refer to secondary memory are *auxiliary memory*, *auxiliary storage*, *external memory*, and *external storage*.

secondary memory

Information in secondary storage is kept in units called **files,** which can be as large or as small as you like. A program, for example, is stored in a file in secondary storage and copied into main memory when the program is run. You can store a program, a letter, an inventory list, or any other unit of information in a file.

files

Several different kinds of secondary memory can be attached to a single computer. The most common forms of secondary memory are *hard disks*, *diskettes*, *CDs*, *DVDs* and *removable flash memory drives*. (**Diskettes** are also sometimes referred to as *floppy disks*.) **CDs** (compact discs) used on computers are basically the same as those used to record and play music, while **DVDs** (digital video discs) are the same as those used to play videos. CDs and DVDs for computers can be read-only so that your computer can read, but cannot change, the data on the disc; CDs and DVDs for computers can also be read/write, which can have their data changed by the computer. Information is stored on hard disks and diskettes in basically the same way as it is stored on CDs and DVDs. **Hard disks** are fixed in place and are normally not removed from the disk drive. Diskettes and CDs can be easily removed from the disk drive and carried to another computer. Diskettes and CDs have the advantages of being inexpensive and portable, but hard disks hold more data and operate faster. Other forms of secondary memory are also available, but this list covers most forms that you are likely to encounter.

CDs, DVDs, disks, and diskettes

Main memory is often referred to as **RAM** or **random access memory.** It is called *random access* because the computer can immediately access the data in any memory location. Secondary memory often requires **sequential access,** which means that the computer must look through all (or at least very many) memory locations until it finds the item it needs.

RAM

The **processor** (also known as the **central processing unit,** or **CPU**) is the "brain" of the computer. When a computer is advertised, the computer company tells you what *chip* it contains. The **chip** is the processor. The processor follows the instructions in a program and performs the calculations specified by the program. The processor is, however, a very simple brain. All it can do is follow a set of simple instructions provided by the programmer. Typical processor instructions say things like "Interpret the zeros and ones as numbers, and then add the number in memory location 37 to the number in memory location 59, and put the answer in location 43," or "Read a letter of input, convert it to its code as a string of zeros and ones, and place it in

processor

chip

M01_SAVI1346_07_SB_C01.fm Page 7 Friday, January 4, 2008 5:01 PM

memory location 1298." The processor can add, subtract, multiply, and divide and can move things from one memory location to another. It can interpret strings of zeros and ones as letters and send the letters to an output device. The processor also has some primitive ability to rearrange the order of instructions. Processor instructions vary somewhat from one computer to another. The processor of a modern computer can have as many as several hundred available instructions. However, these instructions are typically all about as simple as those we have just described.

Software

You do not normally talk directly to the computer, but communicate with it through an *operating system*. The **operating system** allocates the computer's resources to the different tasks that the computer must accomplish. The operating system is actually a program, but it is perhaps better to think of it as your chief servant. It is in charge of all your other servant programs, and it delivers your requests to them. If you want to run a program, you tell the operating system the name of the file that contains it, and the operating system runs the program. If you want to edit a file, you tell the operating system the name of the file and it starts up the editor to work on that file. To most users the operating system is the computer. Most users never see the computer without its operating system. The names of some common operating systems are *UNIX*, *DOS*, *Linux*, *Windows*, *Mac OS*, and *VMS*.

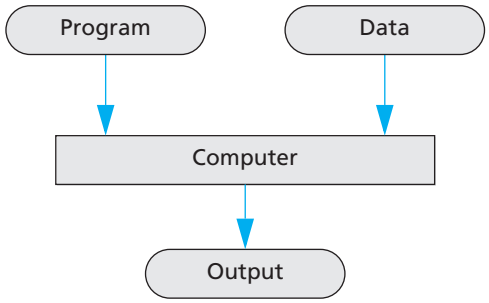
operating system

A **program** is a set of instructions for a computer to follow. As shown in Display 1.3, the input to a computer can be thought of as consisting of two parts, a program and some data. The computer follows the instructions in the program, and in that way, performs some process. The **data** is what we conceptualize as the input to the program. For example, if the program adds two numbers, then the two numbers are the data. In other words, the data is the input to the program, and both the program and the data are input to the computer (usually via the operating system). Whenever we give a computer

program

data

DISPLAY 1.3 Simple View of Running a Program



M01_SAVI1346_07_SB_C01.fm Page 8 Friday, January 4, 2008 5:01 PM

running a program

executing a program

both a program to follow and some data for the program, we are said to be **running the program** on the data, and the computer is said to **execute the program** on the data. The word *data* also has a much more general meaning than the one we have just given it. In its most general sense it means any information available to the computer. The word is commonly used in both the narrow sense and the more general sense.

High-Level Languages

There are many languages for writing programs. In this text we will discuss the C++ programming language and use it to write our programs. C++ is a high-level language, as are most of the other programming languages you are likely to have heard of, such as C, Java, Pascal, Visual Basic, FORTRAN, COBOL, Lisp, Scheme, and Ada. **High-level languages** resemble human languages in many ways. They are designed to be easy for human beings to write programs in and to be easy for human beings to read. A high-level language, such as C++, contains instructions that are much more complicated than the simple instructions a computer's processor (CPU) is capable of following.

high-level language

low-level language

The kind of language a computer can understand is called a **low-level language.** The exact details of low-level languages differ from one kind of computer to another. A typical low-level instruction might be the following:

```
ADD X Y Z
```

This instruction might mean "Add the number in the memory location called `X` to the number in the memory location called `Y`, and place the result in the memory location called `Z`." The above sample instruction is written in what is called **assembly language.** Although assembly language is almost the same as the language understood by the computer, it must undergo one simple translation before the computer can understand it. In order to get a computer to follow an assembly language instruction, the words need to be translated into strings of zeros and ones. For example, the word `ADD` might translate to 0110, the `X` might translate to 1001, the `Y` to 1010, and the `Z` to 1011. The version of the above instruction that the computer ultimately follows would then be:

assembly language

```
0110 1001 1010 1011
```

Assembly language instructions and their translation into zeros and ones differ from machine to machine.

machine language

Programs written in the form of zeros and ones are said to be written in **machine language,** because that is the version of the program that the computer (the machine) actually reads and follows. Assembly language and machine language are almost the same thing, and the distinction between them will not be important to us. The important distinction is that between

M01_SAVI1346_07_SB_C01.fm Page 9 Friday, January 4, 2008 5:01 PM

machine language and high-level languages like C++: Any high-level language program must be translated into machine language before the computer can understand and follow the program.

Compilers

A program that translates a high-level language like C++ to a machine language is called a **compiler.** A compiler is thus a somewhat peculiar sort of program, in that its input or data is some other program, and its output is yet another program. To avoid confusion, the input program is usually called the **source program** or **source code,** and the translated version produced by the compiler is called the **object program** or **object code.** The word **code** is frequently used to mean a program or a part of a program, and this usage is particularly common when referring to object programs. Now, suppose you want to run a C++ program that you have written. In order to get the computer to follow your C++ instructions, proceed as follows. First, run the compiler using your C++ program as data. Notice that in this case, your C++ program is not being treated as a set of instructions. To the compiler, your C++ program is just a long string of characters. The output will be another long string of characters, which is the machine-language equivalent of your C++ program. Next, run this machine-language program on what we normally think of as the data for the C++ program. The output will be what we normally conceptualize as the output of the C++ program. The basic process is easier to visualize if you have two computers available, as diagrammed in Display 1.4. In reality, the entire process is accomplished by using one computer two times.

compiler

source program
object program
code

Compiler

A **compiler** is a program that translates a high-level language program, such as a C++ program, into a machine-language program that the computer can directly understand and execute.

The complete process of translating and running a C++ program is a bit more complicated than what we show in Display 1.4. Any C++ program you write will use some operations (such as input and output routines) that have already been programmed for you. These items that are already programmed for you (like input and output routines) are already compiled and have their object code waiting to be combined with your program's object code to produce a complete machine-language program that can be run on the computer. Another program, called a **linker,** combines the object code for these program pieces with the object code that the compiler produced from your C++ program.

linker

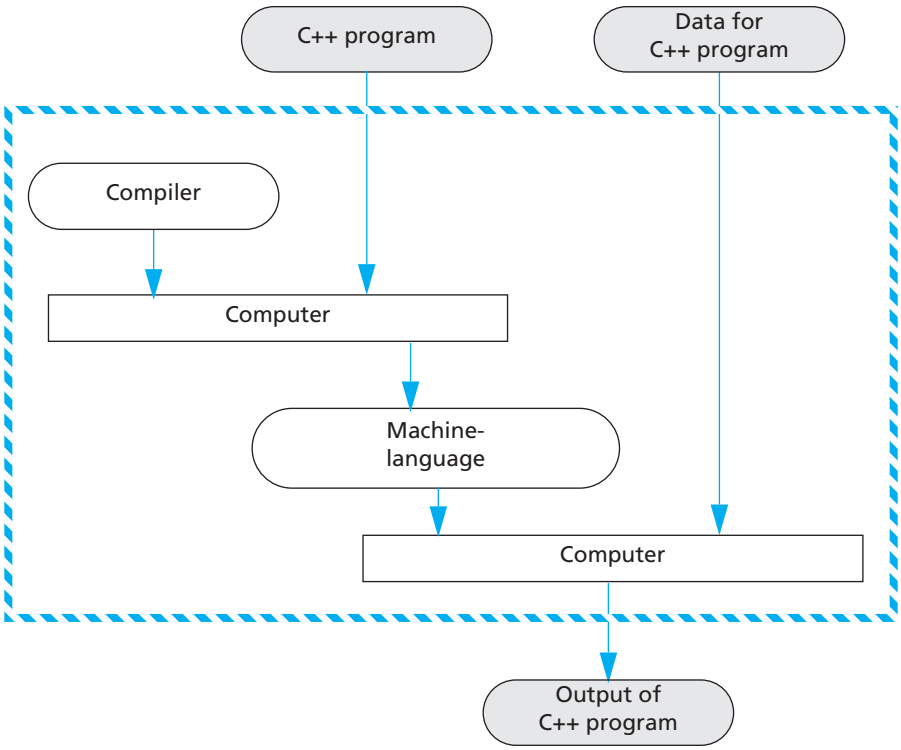
M01_SAVI1346_07_SB_C01.fm Page 10 Friday, January 4, 2008 5:01 PM

The interaction of the compiler and the linker are diagrammed in Display 1.5. In routine cases, many systems will do this linking for you automatically. Thus, you may not need to worry about linking in very simple cases.

Linking

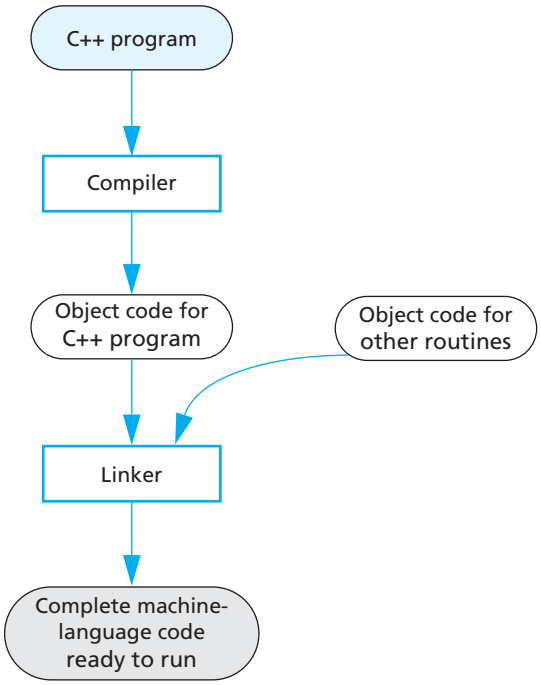
The object code for your C++ program must be combined with the object code for routines (such as input and output routines) that your program uses. This process of combining object code is called **linking** and is done by a program called a **linker.** For simple programs, linking may be done for you automatically.

DISPLAY 1.4 Compiling and Running a C++ Program (Basic Outline)



M01_SAVI1346_07_SB_C01.fm Page 11 Friday, January 4, 2008 5:01 PM

DISPLAY 1.5 Preparing a C++ Program for Running



SELF-TEST EXERCISES

1. What are the five main components of a computer?
2. What would be the data for a program to add two numbers?
3. What would be the data for a program that assigns letter grades to students in a class?
4. What is the difference between a machine-language program and a high-level language program?
5. What is the role of a compiler?
6. What is a source program? What is an object program?
7. What is an operating system?
8. What purpose does the operating system serve?

M01_SAVI1346_07_SB_C01.fm Page 12 Friday, January 4, 2008 5:01 PM

9. Name the operating system that runs on the computer you use to prepare programs for this course.

10. What is linking?

11. Find out whether linking is done automatically by the compiler you use for this course.

History Note

Charles Babbage

The first truly programmable computer was designed by **Charles Babbage,** an English mathematician and physical scientist. Babbage began the project sometime before 1822 and worked on it for the rest of his life. Although he never completed the construction of his machine, the design was a conceptual milestone in the history of computing. Much of what we know about Charles Babbage and his computer design comes from the writings of his colleague

Ada Augusta

Ada Augusta. Ada Augusta was the daughter of the poet Byron and was the Countess of Lovelace. Ada Augusta is frequently given the title of the first computer programmer. Her comments, quoted in the opening of the next section, still apply to the process of solving problems on a computer. Computers are not magic and do not, at least as yet, have the ability to formulate sophisticated solutions to all the problems we encounter. Computers simply do what the programmer orders them to do. The solutions to problems are carried out by the computer, but the solutions are formulated by the programmer. Our discussion of computer programming begins with a discussion of how a programmer formulates these solutions.

1.2 PROGRAMMING AND PROBLEM-SOLVING

> *The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.*
>
> ADA AUGUSTA, COUNTESS OF LOVELACE (1815–1852)

In this section we describe some general principles that you can use to design and write programs. These principles are not particular to C++. They apply no matter what programming language you are using.

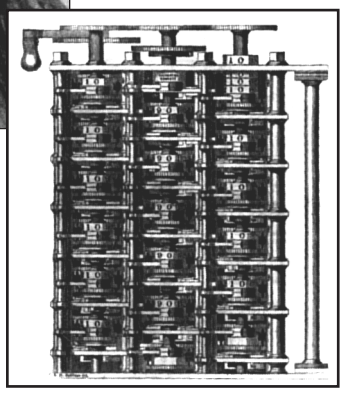
Algorithms

When learning your first programming language it is easy to get the impression that the hard part of solving a problem on a computer is translating your

M01_SAVI1346_07_SB_C01.fm Page 13 Friday, January 4, 2008 5:01 PM

▲
Ada Augusta,
Countess of Lovelace and
the first computer programmer

▲ Charles Babbage



◀ A model of
Babbage's
computer

ideas into the specific language that will be fed into the computer. This definitely is not the case. The most difficult part of solving a problem on a computer is discovering the method of solution. After you come up with a method of solution, it is routine to translate your method into the required language, be it C++ or some other programming language. It is therefore helpful to temporarily ignore the programming language and to concentrate instead on formulating the steps of the solution and writing them down in plain English, as if the instructions were to be given to a human being rather than a computer. A sequence of instructions expressed in this way is frequently referred to as an *algorithm*.

A sequence of precise instructions which leads to a solution is called an **algorithm.** Some approximately equivalent words are *recipe*, *method*, *directions*, **algorithm**

M01_SAVI1346_07_SB_C01.fm Page 14 Friday, January 4, 2008 5:01 PM

procedure, and *routine*. The instructions may be expressed in a programming language or a human language. Our algorithms will be expressed in English and in the programming language C++. A computer program is simply an algorithm expressed in a language that a computer can understand. Thus, the term *algorithm* is more general than the term *program*. However, when we say that a sequence of instructions is an algorithm, we usually mean that the instructions are expressed in English, since if they were expressed in a programming language we would use the more specific term *program*. An example may help to clarify the concept.

Display 1.6 contains an algorithm expressed in English. The algorithm determines the number of times a specified name occurs on a list of names. If the list contains the winners of each of last season's football games and the name is that of your favorite team, then the algorithm determines how many games your team won. The algorithm is short and simple but is otherwise very typical of the algorithms with which we will be dealing.

The instructions numbered 1 through 5 in our sample algorithm are meant to be carried out in the order they are listed. Unless otherwise specified, we will always assume that the instructions of an algorithm are carried out in the order in which they are given (written down). Most interesting algorithms do, however, specify some change of order, usually a repeating of some instruction again and again such as in instruction 4 of our sample algorithm.

origin of the word algorithm

The word *algorithm* has a long history. It derives from the name of a ninth-century Persian mathematician and astronomer al-Khowarizmi. He wrote a famous textbook on the manipulation of numbers and equations. The book was entitled *Kitab al-jabr w'almuqabala*, which can be translated as *Rules for reuniting and reducing*. The similar-sounding word *algebra* was derived from the arabic word *al-jabr*, which appears in the title of the book and which is often translated as *reuniting* or *restoring*. The meanings of the words *algebra* and *algorithm* used to be much more intimately related than they are today. Indeed, until modern times, the word *algorithm* usually referred only to

DISPLAY 1.6 An Algorithm

Algorithm that determines how many times a name occurs in a list of names:

1. Get the list of names.
2. Get the name being checked.
3. Set a counter to zero.
4. Do the following for each name on the list:
   Compare the name on the list to the name being checked,
   and if the names are the same, then add one to the counter.
5. Announce that the answer is the number indicated by the counter.

M01_SAVI1346_07_SB_C01.fm Page 15 Friday, January 4, 2008 5:01 PM

algebraic rules for solving numerical equations. Today the word *algorithm* can be applied to a wide variety of kinds of instructions for manipulating symbolic as well as numeric data. The properties that qualify a set of instructions as an algorithm now are determined by the nature of the instructions rather than by the things manipulated by the instructions. To qualify as an algorithm, a set of instructions must completely and unambiguously specify the steps to be taken and the order in which they are taken. The person or machine carrying out the algorithm does exactly what the algorithm says, neither more nor less.

Algorithm

An **algorithm** is a sequence of precise instructions that leads to a solution.

Program Design

Designing a program is often a difficult task. There is no complete set of rules, no algorithm to tell you how to write programs. Program design is a creative process. Still, there is the outline of a plan to follow. The outline is given in diagrammatic form in Display 1.7. As indicated there, the entire program-design process can be divided into two phases, the *problem-solving phase* and the *implementation phase*. The result of the **problem-solving phase** is an algorithm, expressed in English, for solving the problem. To produce a program in a programming language such as C++, the algorithm is translated into the programming language. Producing the final program from the algorithm is called the **implementation phase.**

problem-solving phase

implementation phase

The first step is to be certain that the task—that you want your program to do—is completely and precisely specified. Do not take this step lightly. If you do not know exactly what you want as the output of your program, you may be surprised at what your program produces. Be certain that you know what the input to the program will be and exactly what information is supposed to be in the output, as well as what form that information should be in. For example, if the program is a bank accounting program, you must know not only the interest rate, but also whether interest is to be compounded annually, monthly, daily, or whatever. If the program is supposed to write poetry, you need to determine whether the poems can be in free verse or must be in iambic pentameter or some other meter.

Many novice programmers do not understand the need to design an algorithm before writing a program in a programming language, such as C++, and so they try to short-circuit the process by omitting the problem-solving phase entirely, or by reducing it to just the problem definition part. This seems reasonable. Why not "go for the mark" and save time? The answer is that *it does not save time!* Experience has shown that the two-phase process will produce a

M01_SAVI1346_07_SB_C01.fm Page 16 Friday, January 4, 2008 5:01 PM

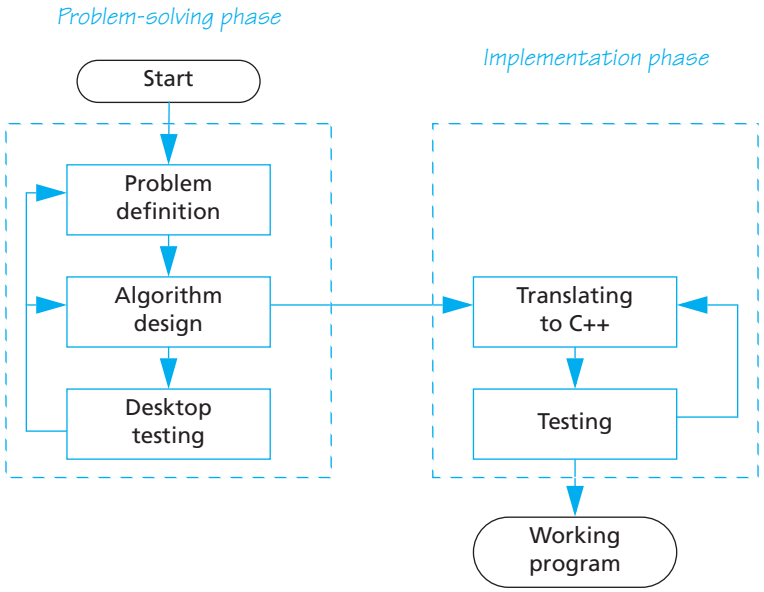
correctly working program faster. The two-phase process simplifies the algorithm design phase by isolating it from the detailed rules of a programming language such as C++. The result is that the algorithm design process becomes much less intricate and much less prone to error. For even a modest-size program, it can represent the difference between a half day of careful work and several frustrating days of looking for mistakes in a poorly understood program.

The implementation phase is not a trivial step. There are details to be concerned about, and occasionally some of these details can be subtle, but it is much simpler than you might at first think. Once you become familiar with C++ or any other programming language, the translation of an algorithm from English into the programming language becomes a routine task.

As indicated in Display 1.7, testing takes place in both phases. Before the program is written, the algorithm is tested, and if the algorithm is found to be deficient, then the algorithm is redesigned. That desktop testing is performed by mentally going through the algorithm and executing the steps yourself. On large algorithms this will require a pencil and paper. The C++ program is tested by compiling it and running it on some sample input data. The compiler will give error messages for certain kinds of errors. To find other types of errors, you must somehow check to see whether the output is correct.

The process diagrammed in Display 1.7 is an idealized picture of the program design process. It is the basic picture you should have in mind, but

DISPLAY 1.7 Program Design Process



M01_SAVI1346_07_SB_C01.fm Page 17 Friday, January 4, 2008 5:01 PM

reality is sometimes more complicated. In reality, mistakes and deficiencies are discovered at unexpected times, and you may have to back up and redo an earlier step. For example, as shown in Display 1.7, testing the algorithm might reveal that the definition of the problem was incomplete. In such a case you must back up and reformulate the definition. Occasionally, deficiencies in the definition or algorithm may not be observed until a program is tested. In that case you must back up and modify the problem definition or algorithm and all that follows them in the design process.

Object-Oriented Programming

The program design process that we outlined in the previous section represents a program as an algorithm (set of instructions) for manipulating some data. That is a correct view, but not always the most productive view. Modern programs are usually designed using a method known as *object-oriented programming*, or **OOP.** In OOP a program is viewed as a collection of interacting objects. The methodology is easiest to understand when the program is a simulation program. For example, for a program to simulate a highway interchange, the objects might represent the automobiles and the lanes of the highway. Each object has algorithms that describe how it should behave in different situations. Programming in the OOP style consists of designing the objects and the algorithms they use. When programming in the OOP framework the term *Algorithm design* in Display 1.7 would be replaced with the phrase *Designing the objects and their algorithms*.

OOP

The main characteristics of OOP are *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation is usually described as a form of information hiding or abstraction. That description is correct, but perhaps an easier to understand characterization is to say that encapsulation is a form of simplification of the descriptions of objects. Inheritance has to do with writing reusable program code. Polymorphism refers to a way that a single name can have multiple meanings in the context of inheritance. Having made those statements, we must admit that they hold little meaning for readers who have not heard of OOP before. However, we will describe all these terms in detail later in this book. C++ accommodates OOP by providing **classes,** a kind of data type combining both data and algorithms.

class

The Software Life Cycle

Designers of large software systems, such as compilers and operating systems, divide the software development process into six phases collectively known as the **software life cycle.** The six phases of this life cycle are:

software life cycle

1. Analysis and specification of the task (problem definition)
2. Design of the software (object and algorithm design)

M01_SAVI1346_07_SB_C01.fm Page 18 Friday, January 4, 2008 5:01 PM

3. Implementation (coding)
4. Testing
5. Maintenance and evolution of the system
6. Obsolescence

We did not mention the last two phases in our discussion of program design because they take place after the program is finished and put into service. However, they should always be kept in mind. You will not be able to add improvements or corrections to your program unless you design it to be easy to read and easy to change. Designing programs so that they can be easily modified is an important topic that we will discuss in detail when we have developed a bit more background and a few more programming techniques. The meaning of obsolescence is obvious, but it is not always easy to accept. When a program is not working as it should and cannot be fixed with a reasonable amount of effort, it should be discarded and replaced with a completely new program.

SELF-TEST EXERCISES

12. An algorithm is approximately the same thing as a recipe, but some kinds of steps that would be allowed in a recipe are not allowed in an algorithm. Which steps in the following recipe would be allowed in an algorithm?

```
Place 2 teaspoons of sugar in mixing bowl.
Add 1 egg to mixing bowl.
Add 1 cup of milk to mixing bowl.
Add 1 ounce of rum, if you are not driving.
Add vanilla extract to taste.
Beat until smooth.
Pour into a pretty glass.
Sprinkle with nutmeg.
```

13. What is the first step you should take when creating a program?

14. The program design process can be divided into two main phases. What are they?

15. Explain why the problem-solving phase should not be slighted.

M01_SAVI1346_07_SB_C01.fm Page 19 Friday, January 4, 2008 5:01 PM

1.3 INTRODUCTION TO C++

Language is the only instrument of science...

SAMUEL JOHNSON (1709–1784)

In this section we introduce you to the C++ programming language, which is the programming language used in this book.

Origins of the C++ Language

The first thing that people notice about the C++ language is its unusual name. Is there a C programming language, you might ask? Is there a C– or a C–– language? Are there programming languages named A and B? The answer to most of these questions is no. But the general thrust of the questions is on the mark. There is a B programming language; it was not derived from a language called A, but from a language called BCPL. The C language was derived from the B language, and C++ was derived from the C language. Why are there two pluses in the name C++? As you will see in the next chapter, ++ is an operation in the C and C++ languages, so using ++ produces a nice pun. The languages BCPL and B do not concern us. They are earlier versions of the C programming language. We will start our description of the C++ programming language with a description of the C language.

The C programming language was developed by Dennis Ritchie of AT&T Bell Laboratories in the 1970s. It was first used for writing and maintaining the UNIX operating system. (Up until that time UNIX systems programs were written either in assembly language or in B, a language developed by Ken Thompson, who is the originator of UNIX.) C is a general-purpose language that can be used for writing any sort of program, but its success and popularity are closely tied to the UNIX operating system. If you wanted to maintain your UNIX system, you needed to use C. C and UNIX fit together so well that soon not just systems programs, but almost all commercial programs that ran under UNIX were written in the C language. C became so popular that versions of the language were written for other popular operating systems; its use is not limited to computers that use UNIX. However, despite its popularity, C is not without its shortcomings.

The C language is peculiar because it is a high-level language with many of the features of a low-level language. C is somewhere in between the two extremes of a very high-level language and a low-level language, and therein lies both its strengths and its weaknesses. Like (low-level) assembly language, C language programs can directly manipulate the computer's memory. On the other hand, C has many features of a high-level language, which makes it easier to read and write than assembly language. This makes C an excellent

M01_SAVI1346_07_SB_C01.fm Page 20 Friday, January 4, 2008 5:01 PM

choice for writing systems programs, but for other programs (and in some sense even for systems programs), C is not as easy to understand as other languages; also, it does not have as many automatic checks as some other high-level languages.

To overcome these and other shortcomings of C, Bjarne Stroustrup of AT&T Bell Laboratories developed C++ in the early 1980s. Stroustrup designed C++ to be a better C. Most of C is a subset of C++, and so most C programs are also C++ programs. (The reverse is not true; many C++ programs are definitely not C programs.) Unlike C, C++ has facilities to do *object-oriented programming*, which is a recently developed and very powerful programming technique, described earlier in this chapter.

A Sample C++ Program

Display 1.8 contains a simple C++ program and the screen display that might be generated when a *user* runs and interacts with this program. The person who runs a program is called the **user.** The text typed in by the user is shown in boldface to distinguish it from the text written by the program. On the actual screen both texts would look alike. The person who writes the program is called the **programmer.** Do not confuse the roles of the user and the programmer. The user and the programmer might or might not be the same person. For example, if you write and then run a program, you are both the programmer and the user. With professionally produced programs, the programmer (or programmers) and the user are usually different persons.

user

programmer

In the next chapter we will explain in detail all the C++ features you need to write programs like the one in Display 1.8, but to give you a feel for how a C++ program works, we will now give a brief description of how this particular program works. If some of the details are a bit unclear, do not worry. In this section, we just want to give you a feel for what a C++ program is.

beginning of program

The beginning and end of our sample program contain some details that need not concern us yet. The program begins with the following lines:

```
#include <iostream>
using namespace std;

int main( )
{
```

For now we will consider these lines to be a rather complicated way of saying "The program starts here."

`return 0;`

The program ends with the following two lines:

```
    return 0;
}
```

For a simple program, these two lines simply mean "The program ends here."

M01_SAVI1346_07_SB_C01.fm Page 21 Friday, January 4, 2008 5:01 PM

DISPLAY 1.8 A Sample C++ Program

```
1  #include <iostream>
2  using namespace std;

3  int main( )
4  {
5      int number_of_pods, peas_per_pod, total_peas;

6      cout << "Press return after entering a number.\n";
7      cout << "Enter the number of pods:\n";
8      cin >> number_of_pods;
9      cout << "Enter the number of peas in a pod:\n";
10     cin >> peas_per_pod;

11     total_peas = number_of_pods * peas_per_pod;

12     cout << "If you have ";
13     cout << number_of_pods;
14     cout << " pea pods\n";
15     cout << "and ";
16     cout << peas_per_pod;
17     cout << " peas in each pod, then\n";
18     cout << "you have ";
19     cout << total_peas;
20     cout << " peas in all the pods.\n";

21     return 0;
22 }
```

Sample Dialogue

```
Press return after entering a number.
Enter the number of pods:
10
Enter the number of peas in a pod:
9
If you have 10 pea pods
and 9 peas in each pod, then
you have 90 peas in all the pods.
```

The lines in between these beginning and ending lines are the heart of the program. We will briefly describe these lines, starting with the following line:

```
int number_of_pods, peas_per_pod, total_peas;
```

M01_SAVI1346_07_SB_C01.fm Page 22 Friday, January 4, 2008 5:01 PM

variable declarations

This line is called a **variable declaration.** This variable declaration tells the computer that `number_of_pods`, `peas_per_pod`, and `total_peas` will be used as names for three *variables*. Variables will be explained more precisely in the next chapter, but it is easy to understand how they are used in this program. In this program, the **variables** are used to name numbers. The word that starts this line, `int`, is an abbreviation for the word *integer* and it tells the computer that the numbers named by these variables will be integers. An **integer** is a whole number, like 1, 2, −1, −7, 0, 205, −103, and so forth.

variables

integer

statements

The remaining lines are all instructions that tell the computer to do something. These instructions are called **statements** or **executable statements.** In this program each statement fits on exactly one line. That need not be true, but for very simple programs, statements are usually listed one per line.

`cin` and `cout`

Most of the statements begin with either the word `cin` or `cout`. These statements are input statements and output statements. The word `cin`, which is pronounced "see-in," is used for input. The statements that begin with `cin` tell the computer what to do when information is entered from the keyboard. The word `cout`, which is pronounced "see-out," is used for output; that is, for sending information from the program to the terminal screen. The letter `c` is there because the language is C++. The arrows, written `<<` or `>>`, tell you the direction that data is moving. The arrows, `<<` and `>>`, are called 'insert' and 'extract,' or 'put to' and 'get from,' respectively. For example, consider the line:

```
cout << "Press return after entering a number.\n";
```

This line may be read, 'put `"Press...number.\n"` to `cout`' or simply 'output `"Press...number.\n"`'. If you think of the word `cout` as a name for the screen (the output device), then the arrows tell the computer to send the string in quotes to the screen. As shown in the sample dialogue, this causes the text contained in the quotes to be written to the screen, The `\n` at the end of the quoted string tells the computer to start a new line after writing out the text. Similarly, the next line of the program also begins with `cout`, and that program line causes the following line of text to be written to the screen:

`\n`

```
Enter the number of pods:
```

The next program line starts with the word `cin`, so it is an input statement. Let's look at that line:

```
cin >> number_of_pods;
```

This line may be read, 'get `number_of_pods` from `cin`' or simply 'input `number_of_pods`'.

If you think of the word `cin` as standing for the keyboard (the input device), then the arrows say that input should be sent from the keyboard to the variable `number_of_pods`. Look again at the sample dialogue. The next line shown has a `10` written in bold. We use bold to indicate something typed in at the keyboard. If you type in the number `10`, then the `10` appears on the screen.

M01_SAVI1346_07_SB_C01.fm Page 23 Friday, January 4, 2008 5:01 PM

If you then press the Return key (which is also sometimes called the *Enter key*), that makes the `10` available to the program. The statement which begins with `cin` tells the computer to send that input value of `10` to the variable `number_of_pods`. From that point on, `number_of_pods` has the value `10`; when we see `number_of_pods` later in the program, we can think of it as standing for the number `10`.

Consider the next two program lines:

```
cout << "Enter the number of peas in a pod:\n";
cin >> peas_per_pod;
```

These lines are similar to the previous two lines. The first sends a message to the screen asking for a number. When you type in a number at the keyboard and press the Return key, that number becomes the value of the variable `peas_per_pod`. In the sample dialogue, we assume that you type in the number `9`. After you type in `9` and press the Return key, the value of the variable `peas_per_pod` becomes `9`.

The next nonblank program line, shown below, does all the computation that is done in this simple program:

```
total_peas = number_of_pods * peas_per_pod;
```

The asterisk symbol, `*`, is used for multiplication in C++. So this statement says to multiply `number_of_pods` and `peas_per_pod`. In this case, `10` is multiplied by `9` to give a result of `90`. The equal sign says that the variable `total_peas` should be made equal to this result of `90`. This is a special use of the equal sign; its meaning here is different than in other mathematical contexts. It gives the variable on the left-hand side a (possibly new) value; in this case it makes `90` the value of `total_peas`.

The rest of the program is basically more of the same sort of output. Consider the next three nonblank lines:

```
cout << "If you have ";
cout << number_of_pods;
cout << " pea pods\n";
```

These are just three more output statements that work basically the same as the previous statements that begin with `cout`. The only thing that is new is the second of these three statements, which says to output the variable `number_of_pods`. When a variable is output, it is the value of the variable that is output. So this statement causes a `10` to be output. (Remember that in this sample run of the program, the variable `number_of_pods` was set to `10` by the user who ran the program.) Thus, the output produced by these three lines is:

```
If you have 10 pea pods
```

Notice that the output is all on one line. A new line is not begun until the special instruction `\n` is sent as output.

M01_SAVI1346_07_SB_C01.fm Page 24 Friday, January 4, 2008 5:01 PM

The rest of the program contains nothing new, and if you understand what we have discussed so far, you should be able to understand the rest of the program.

PITFALL Using the Wrong Slash in \n

backslash

When you use a `\n` in a `cout` statement be sure that you use the **backslash,** which is written `\`. If you make a mistake and use `/n` rather than `\n`, the compiler will not give you an error message. Your program will run, but the output will look peculiar. ■

PROGRAMMING TIP Input and Output Syntax

If you think of `cin` as a name for the keyboard or **in**put device and think of `cout` as a name for the screen or the **out**put device, then it is easy to remember the direction of the arrows `>>` and `<<`. They point in the direction that data moves. For example, consider the statement:

```
cin >> number_of_pods;
```

In the above statement, data moves from the keyboard to the variable `number_of_pods`, and so the arrow points from `cin` to the variable.

On the other hand, consider the output statement:

```
cout << number_of_pods;
```

In this statement the data moves from the variable `number_of_pods` to the screen, so the arrow points from the variable `number_of_pods` to `cout`. ■

Layout of a Simple C++ Program

line breaks and spaces

The general form of a simple C++ program is shown in Display 1.9. As far as the compiler is concerned, the **line breaks** and **spacing** need not be as shown there and in our examples. The compiler will accept any reasonable pattern of line breaks and indentation. In fact, the compiler will even accept most unreasonable patterns of line breaks and indentation. However, a program should always be laid out so that it is easy to read. Placing the opening brace, `{`, on a line by itself and also placing the closing brace, `}`, on a line by itself will make these punctuations easy to find. Indenting each statement and placing each statement on a separate line makes it easy to see what the program instructions are. Later on, some of our statements will be too long to fit on one line and then we will use a slight variant of this pattern for indenting and line breaks. You should follow the pattern set by the examples in this book, or follow the pattern specified by your instructor if you are in a class.

M01_SAVI1346_07_SB_C01.fm Page 25 Friday, January 4, 2008 5:01 PM

DISPLAY 1.9 Layout of a Simple C++ Program

```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      Variable_Declarations
7
8      Statement_1
9      Statement_2
10       ...
11     Statement_Last
12
13     return 0;
14 }
```

In Display 1.8, the variable declarations are on the line that begins with the word `int`. As we will see in the next chapter, you need not place all your variable declarations at the beginning of your program, but that is a good default location for them. Unless you have a reason to place them somewhere else, place them at the start of your program as shown in Display 1.9 and in the sample program in Display 1.8. The **statements** are the instructions that are followed by the computer. In Display 1.8, the statements are the lines that begin with `cout` or `cin`, and the one line that begins with `total_peas` followed by an equal sign. Statements are often called **executable statements.** We will use the terms *statement* and *executable statement* interchangeably. Notice that each of the statements we have seen ends with a semicolon. The semicolon in statements is used in more or less the same way that the period is used in English sentences; it marks the end of a statement.

statement

executable statement

For now you can view the first few lines as a funny way to say "this is the beginning of the program." But we can explain them in a bit more detail. The first line

#include

```
#include <iostream>
```

is called an `include` **directive.** It tells the compiler where to find information about certain items that are used in your program. In this case `iostream` is the name of a library that contains the definitions of the routines that handle input from the keyboard and output to the screen; `iostream` is a file that contains some basic information about this library. The linker program that we discussed earlier in this chapter combines the object code for the library `iostream` and the object code for the program you write. For the library `iostream` this will probably happen automatically on your system. You will eventually use other libraries as well, and when you use them, they will have

include directive

M01_SAVI1346_07_SB_C01.fm Page 26 Friday, January 4, 2008 5:01 PM

to be named in directives at the start of your program. For other libraries, you may need to do more than just place an `include` directive in your program, but in order to use any library in your program, you will always need to at least place an `include` directive for that library in your program. Directives always begin with the symbol `#`. Some compilers require that directives have no spaces around the `#`; so it is always safest to place the `#` at the very start of the line and not include any space between the `#` and the word `include`.

The following line further explains the `include` directive that we just explained.

```
using namespace std;
```

This line says that the names defined in `iostream` are to be interpreted in the "standard way" (`std` is an abbreviation of *standard*). We will have more to say about this line a bit later in this book.

`int main( )`

The third and fourth nonblank lines, shown next, simply say that the main part of the program starts here:

```
int main( )
{
```

The correct term is *main function*, rather than *main part*, but the reason for that subtlety will not concern us until Chapter 4. The braces `{` and `}` mark the beginning and end of the main part of the program. They need not be on a line by themselves, but that is the way to make them easy to find and we will therefore always place each of them on a line by itself.

`return 0;`

The next-to-last line

```
return 0;
```

says to "end the program when you get to here." This line need not be the last thing in the program, but in a very simple program it makes no sense to place it anywhere else. Some compilers will allow you to omit this line and will figure out that the program ends when there are no more statements to execute. However, other compilers will insist that you include this line, so it is best to get in the habit of including it, even if your compiler is happy without it. This line

return statement

is called a **return statement** and is considered to be an executable statement because it tells the computer to do something; specifically, it tells the computer to end the program. The number `0` has no intuitive significance to us yet, but must be there; its meaning will become clear as you learn more about C++. Note that even though the `return` statement says to end the program, you still must add a closing brace `}` at the end of the main part of your program.

PITFALL Putting a Space before the `include` File Name

Be certain that you do not have any extra space between the < and the `iostream` file name (Display 1.9) or between the end of the file name and the closing >.

M01_SAVI1346_07_SB_C01.fm Page 27 Friday, January 4, 2008 5:01 PM

The compiler `include` directive is not very smart: It will search for a file name that starts or ends with a space! The file name will not be found, producing an error that is quite difficult to find. You should make this error deliberately in a small program, then compile it. Save the message that your compiler produces so you know what the error message means the next time you get that error message. ■

Compiling and Running a C++ Program

Video Note
Compiling and Running a C++ Program

In the previous section you learned what would happen if you ran the C++ program shown in Display 1.8. But where is that program and how do you make it run?

You write a C++ program using a text editor in the same way that you write any other document such as a term paper, a love letter, a shopping list, or whatever. The program is kept in a file just like any other document you prepare using a text editor. There are different text editors, and the details of how to use the text editor will vary from one text editor to another, so we cannot say too much more about your text editor. You should consult the documentation for your editor.

The way that you compile and run a C++ program also depends on the particular system you are using, so we will discuss these points in only a very general way. You need to learn how to give the commands to compile, link, and run a C++ program on your system. These commands can be found in the manuals for your system and by asking people who are already using C++ on your system. When you give the command to compile your program, this will produce a machine-language translation of your C++ program. This translated version of your program is called the *object code* for your program. The object code for your program must be linked (that is, combined) with the object code for routines (such as input and output routines) that are already written for you. It is likely that this linking will be done automatically, so you do not need to worry about linking. But on some systems, you may be required to make a separate call to the linker. Again, consult your manuals or a local expert. Finally, you give the command to run your program; how you give that command also depends on the system you are using, so check with the manuals or a local expert.

■ PROGRAMMING TIP Getting Your Program to Run

Different compilers and different environments might require a slight variation in some details of how you set up a file with your C++ program. Obtain a copy of the program in Display 1.10. It is available for downloading over the Internet. (See the preface for details.) Alternatively, *very carefully* type in the program yourself. Compile the program. If you get an error message, check your typing, fix any typing mistakes, and recompile the file. Once the program compiles with no error messages, try running the program.

M01_SAVI1346_07_SB_C01.fm Page 28 Friday, January 4, 2008 5:01 PM

DISPLAY 1.10 Testing Your C++ Setup

```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      cout << "Testing 1, 2, 3\n";
7      return 0;
8  }
9
```

If you cannot compile and run this program, then see the programming tip entitled "Getting Your Program to Run." It suggests some things to do to get your C++ programs to run on your particular computer setup.

Sample Dialogue

```
Testing 1, 2, 3
```

If you get the program to compile and run normally, you are all set. You do not need to do anything different from the examples shown in the book. If this program does not compile or does not run normally, then read on. In what follows we offer some hints for dealing with your C++ setup. Once you get this simple program to run normally, you will know what small changes to make to your C++ program files in order to get them to run on your system.

If your program seems to run, but you do not see the output line

```
Testing 1, 2, 3
```

then, in all likelihood, the program probably did give that output, but it disappeared before you could see it. Try adding the following to the end of your program, just before the line `return 0;` these lines should stop your program to allow you to read the output.

```
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
```

The part in braces should then read as follows:

```
cout << "Testing 1, 2, 3\n";
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
return 0;
```

For now you need not understand these added lines, but they will be clear to you by the end of Chapter 2.

If the program does not compile or run at all, then try changing

M01_SAVI1346_07_SB_C01.fm Page 29 Friday, January 4, 2008 5:01 PM


```
#include <iostream>
```

by adding `.h` to the end of `iostream`, so it reads as follows:

```
#include <iostream.h>
```

If your program requires `iostream.h` instead of `iostream`, then you have an old C++ compiler and should obtain a more recent compiler.

If your program still does not compile and run normally, try deleting

```
using namespace std;
```

If your program still does not compile and run, then check the documentation for your version of C++ to see if any more "directives" are needed for "console" input/output.

If all this fails, consult your instructor if you are in a course. If you are not in a course or you are not using the course computer, check the documentation for your C++ compiler or check with a friend who has a similar computer setup. The necessary change is undoubtedly very small and, once you find out what it is, very easy. ■

SELF-TEST EXERCISES

16. If the following statement were used in a C++ program, what would it cause to be written on the screen?

```
cout << "C++ is easy to understand.";
```

17. What is the meaning of `\n` as used in the following statement (which appears in Display 1.8)?

```
cout << "Enter the number of peas in a pod:\n";
```

18. What is the meaning of the following statement (which appears in Display 1.8)?

```
cin >> peas_per_pod;
```

19. What is the meaning of the following statement (which appears in Display 1.8)?

```
total_peas = number_of_pods * peas_per_pod;
```

20. What is the meaning of this directive?

```
#include <iostream>
```

M01_SAVI1346_07_SB_C01.fm Page 30 Friday, January 4, 2008 5:01 PM

21. What, if anything, is wrong with the following `#include` directives?
    a. `#include <iostream >`
    b. `#include < iostream>`
    c. `#include <iostream>`

1.4 TESTING AND DEBUGGING

"And if you take one from three hundred and sixty-five, what remains?"
"Three hundred and sixty-four, of course."
Humpty Dumpty looked doubtful. "I'd rather see that done on paper," he said.

LEWIS CARROLL, *Through the Looking-Glass*

bug
debugging

A mistake in a program is usually called a **bug,** and the process of eliminating bugs is called **debugging.** There is colorful history of how this term came into use. It occurred in the early days of computers, when computer hardware was extremely sensitive. Rear Admiral Grace Murray Hopper (1906–1992) was "the third programmer on the world's first large-scale digital computer." (Denise W. Gurer, "Pioneering women in computer science" CACM 38(1):45–54, January 1995.) While Hopper was working on the Harvard Mark I computer under the command of Harvard professor Howard H. Aiken, an unfortunate moth caused a relay to fail. Hopper and the other programmers taped the deceased moth in the logbook with the note "First actual case of bug being found." The logbook is currently on display at the Naval Museum in Dahlgren, Virginia. This was the first documented computer bug. Professor Aiken would come into the facility during a slack time and inquire if any numbers were being computed. The programmers would reply that they were debugging the computer. For more information about Admiral Hopper and other persons in computing, see Robert Slater, *Portraits in Silicon,* MIT Press, 1987. Today, a bug is a mistake in a program. In this section we describe the three main kinds of programming mistakes and give some hints on how to correct them.

Kinds of Program Errors

syntax error

The compiler will catch certain kinds of mistakes and will write out an error message when it finds a mistake. It will detect what are called **syntax errors,** because they are, by and large, violation of the syntax (that is, the grammar rules) of the programming language, such as omitting a semicolon.

If the compiler discovers that your program contains a syntax error, it will tell you where the error is likely to be and what kind of error it is likely to be. If the compiler says your program contains a syntax error, you can be confident that it does. However, the compiler may be incorrect about either the location or the nature of the error. It does a better job of determining the location of an error, to within a line or two, than it does of determining the source of the error.

M01_SAVI1346_07_SB_C01.fm Page 31 Friday, January 4, 2008 5:01 PM

This is because the compiler is guessing at what you meant to write down and can easily guess wrong. After all, the compiler cannot read your mind. Error messages subsequent to the first one have a higher likelihood of being incorrect with respect to either the location or the nature of the error. Again, this is because the compiler must guess your meaning. If the compiler's first guess was incorrect, this will affect its analysis of future mistakes, since the analysis will be based on a false assumption.

error messages versus warning messages

If your program contains something that is a direct violation of the syntax rules for your programming language, the compiler will give you an **error message.** However, sometimes the compiler will give you only a **warning message,** which indicates that you have done something that is not, technically speaking, a violation of the programming language syntax rules, but that is unusual enough to indicate a likely mistake. When you get a warning message, the compiler is saying, "Are you sure you mean this?" At this stage of your development, you should treat every warning as if it were an error until your instructor approves ignoring the warning.

run-time error

There are certain kinds of errors that the computer system can detect only when a program is run. Appropriately enough, these are called **run-time errors.** Most computer systems will detect certain run-time errors and output an appropriate error message. Many run-time errors have to do with numeric calculations. For example, if the computer attempts to divide a number by zero, that is normally a run-time error.

If the compiler approved of your program and the program ran once with no run-time error messages, this does not guarantee that your program is correct. Remember, the compiler will only tell you if you wrote a syntactically (that is, grammatically) correct C++ program. It will not tell you whether the program does what you want it to do. Mistakes in the underlying algorithm or in translating the algorithm into the C++ language are called **logic errors.** For example, if you were to mistakenly use the addition sign + instead of the multiplication sign * in the program in Display 1.8, that would be a logic error. The program would compile and run normally, but would give the wrong answer. If the compiler approves of your program and there are no run-time errors, but the program does not perform properly, then undoubtedly your program contains a logic error. Logic errors are the hardest kind to diagnose, because the computer gives you no error messages to help find the error. It cannot reasonably be expected to give any error messages. For all the computer knows, you may have meant what you wrote.

logic error

PITFALL Assuming Your Program Is Correct

In order to test a new program for logic errors, you should run the program on several representative data sets and check its performance on those inputs. If the program passes those tests, you can have more confidence in it, but this is still not an absolute guarantee that the program is correct. It still

M01_SAVI1346_07_SB_C01.fm Page 32 Friday, January 4, 2008 5:01 PM

may not do what you want it to do when it is run on some other data. The only way to justify confidence in a program is to program carefully and so avoid most errors. ■

SELF-TEST EXERCISES

22. What are the three main kinds of program errors?

23. What kinds of errors are discovered by the compiler?

24. If you omit a punctuation symbol (such as a semicolon) from a program, an error is produced. What kind of error?

25. Omitting the final brace } from a program produces an error. What kind of error?

26. Suppose your program has a situation about which the compiler reports a warning. What should you do about it? Give the text's answer, and your local answer if it is different from the text's. Identify your answers as the text's or as based on your local rules.

27. Suppose you write a program that is supposed to compute the interest on a bank account at a bank that computes interest on a daily basis, and suppose you incorrectly write your program so that it computes interest on an annual basis. What kind of program error is this?

CHAPTER SUMMARY

- The collection of programs used by a computer is referred to as the **software** for that computer. The actual physical machines that make up a computer installation are referred to as **hardware.**
- The five main components of a computer are the input device(s), the output device(s), the processor (CPU), the main memory, and the secondary memory.
- A computer has two kinds of memory: main memory and secondary memory. Main memory is used only while the program is running. Secondary memory is used to hold data that will stay in the computer before and/or after the program is run.
- A computer's main memory is divided into a series of numbered locations called **bytes.** The number associated with one of these bytes is called the **address** of the byte. Often several of these bytes are grouped together to

M01_SAVI1346_07_SB_C01.fm Page 33 Friday, January 4, 2008 5:01 PM

form a larger memory location. In that case, the address of the first byte is used as the address of this larger memory location.

- A **byte** consists of eight binary digits, each either zero or one. A digit that can only be zero or one is called a **bit.**
- A **compiler** is a program that translates a program written in a high-level language like C++ into a program written in the machine language that the computer can directly understand and execute.
- A sequence of precise instructions that leads to a solution is called an **algorithm.** Algorithms can be written in English or in a programming language, like C++. However, the word *algorithm* is usually used to mean a sequence of instructions written in English (or some other human language, such as Spanish or Arabic).
- Before writing a C++ program, you should design the algorithm (method of solution) that the program will use.
- Programming errors can be classified into three groups: syntax errors, run-time errors, and logic errors. The computer will usually tell you about errors in the first two categories. You must discover logic errors yourself.
- The individual instructions in a C++ program are called **statements.**
- A variable in a C++ program can be used to name a number. (Variables are explained more fully in the next chapter.)
- A statement in a C++ program that begins with `cout <<` is an output statement, which tells the computer to output to the screen whatever follows the <<.
- A statement in a C++ program that begins with `cin >>` is an input statement.

Answers to Self-Test Exercises

1. The five main components of a computer are the input device(s), the output device(s), the processor (CPU), the main memory, and the secondary memory.
2. The two numbers to be added.
3. The grades for each student on each test and each assignment.
4. A machine-language program is a low-level language consisting of zeros and ones that the computer can directly execute. A high-level language is

M01_SAVI1346_07_SB_C01.fm Page 34 Friday, January 4, 2008 5:01 PM

written in a more English-like format and is translated by a compiler into a machine-language program that the computer can directly understand and execute.

5. A compiler translates a high-level language program into a machine-language program.

6. The high-level language program that is input to a compiler is called the source program. The translated machine-language program that is output by the compiler is called the object program.

7. An operating system is a program, or several cooperating programs, but is best thought of as the user's chief servant.

8. An operating system's purpose is to allocate the computer's resources to different tasks the computer must accomplish.

9. Among the possibilities are the Macintosh operating system Mac OS, Windows 2000, Windows XP, VMS, Solaris, SunOS, UNIX (or perhaps one of the UNIX-like operating systems such as Linux). There are many others.

10. The object code for your C++ program must be combined with the object code for routines (such as input and output routines) that your program uses. This process of combining object code is called linking. For simple programs this linking may be done for you automatically.

11. The answer varies, depending on the compiler you use. Most UNIX and UNIX-like compilers link automatically, as do the compilers in most integrated development environments for Windows and Macintosh operating systems.

12. The following instructions are too vague for use in an algorithm:

```
Add vanilla extract to taste.

Beat until smooth.

Pour into a pretty glass.

Sprinkle with nutmeg.
```

The notions of "to taste," "smooth," and "pretty" are not precise. The instruction "sprinkle" is too vague, since it does not specify how much nutmeg to sprinkle. The other instructions are reasonable to use in an algorithm.

13. The first step you should take when creating a program is to be certain that the task to be accomplished by the program is completely and precisely specified.

M01_SAVI1346_07_SB_C01.fm Page 35 Friday, January 4, 2008 5:01 PM

14. The problem-solving phase and the implementation phase.

15. Experience has shown that the two-phase process produces a correctly working program faster.

16. `C++ is easy to understand.`

17. The symbols `\n` tell the computer to start a new line in the output so that the next item output will be on the next line.

18. This statement tells the computer to read the next number that is typed in at the keyboard and to send that number to the variable named `peas_per_pod`.

19. This statement says to multiply the two numbers in the variables `number_of_pods` and `peas_per_pod`, and to place the result in the variable named `total_peas`.

20. The `#include <iostream>` directive tells the compiler to fetch the file `iostream`. This file contains declarations of `cin`, `cout`, the insertion (`<<`) and extraction (`>>`) operators for I/O (input and output). This enables correct linking of the object code from the iostream library with the I/O statements in the program.

21. a. The extra space after the `iostream` file name causes a *file-not-found* error message.
    b. The extra space before the `iostream` file name causes a *file-not-found* error message.
    c. This one is correct.

22. The three main kinds of program errors are syntax errors, run-time errors, and logic errors.

23. The compiler detects syntax errors. There are other errors that are not technically syntax errors that we are lumping with syntax errors. You will learn about these later.

24. A syntax error.

25. A syntax error.

26. The text states that you should take warnings as if they had been reported as errors. You should ask your instructor for the local rules on how to handle warnings.

27. A logic error.

M01_SAVI1346_07_SB_C01.fm Page 36 Friday, January 4, 2008 5:01 PM

PROGRAMMING PROJECTS

1. Using your text editor, enter (that is, type in) the C++ program shown in Display 1.8. Be certain to type the first line exactly as shown in Display 1.8. In particular, be sure that the first line begins at the left-hand end of the line with no space before or after the # symbol. Compile and run the program. If the compiler gives you an error message, correct the program and recompile the program. Do this until the compiler gives no error messages. Then run your program.

2. Modify the C++ program you entered in Programming Project 1. Change the program so that it first writes the word `Hello` to the screen and then goes on to do the same things that the program in Display 1.8 does. You will only have to add one line to the program to make this happen. Recompile the changed program and run the changed program. Then change the program even more. Add one more line that will make the program write the word `Good-bye` to the screen at the end of the program. Be certain to add the symbols `\n` to the last output statement so that it reads as follows:

   ```
   cout << "Good-bye\n";
   ```

   (Some systems require that final `\n`, and your system may be one of the systems that requires a final `\n`.) Recompile and run the changed program.

3. Further modify the C++ program that you already have modified in Programming Project 2. Change the multiplication sign * in your C++ program to a division sign /. Recompile the changed program. Run the program. Enter a zero input for "number of peas in a pod." Notice the run time error message due to division by zero.

4. Modify the C++ program that you entered in Programming Project 1. Change the multiplication sign * in your C++ program to an addition sign +. Recompile and run the changed program. Notice that the program compiles and runs perfectly fine, but the output is incorrect. That is because this modification is a logic error.

5. Write a C++ program that reads in two integers and then outputs both their sum and their product. One way to proceed is to start with the program in Display 1.8 and to then modify that program to produce the program for this project. Be certain to type the first line of your program exactly the same as the first line in Display 1.8. In particular, be sure that the first line begins at the left-hand end of the line with no space before or after the # symbol. Also, be certain to add the symbols `\n` to the last output statement in your program. For example, the last output statement might be the following:

   ```
   cout << "This is the end of the program.\n";
   ```

M01_SAVI1346_07_SB_C01.fm Page 37 Friday, January 4, 2008 5:01 PM

(Some systems require that final `\n`, and your system may be one of these.)

Video Note
Solution to Programming Project 1.6

6. The purpose of this exercise is to produce a catalog of typical syntax errors and error messages that will be encountered by a beginner, and to continue acquainting you with the programming environment. This exercise should leave you with a knowledge of what error to look for when given any of a number of common error messages.

   Your instructor may have a program for you to use for this exercise. If not, you should use a program from one of the previous Programming Projects.

   Deliberately introduce errors to the program, compile, record the error and the error message, fix the error, compile again (to be sure you have the program corrected), then introduce another error. Keep the catalog of errors and add program errors and messages to it as you continue through this course.

   The sequence of suggested errors to introduce is:

   a. Put an extra space between the < and the `iostream` file name.
   b. Omit one of the < or > symbols in the `include` directive.
   c. Omit the `int` from `int main()`.
   d. Omit or misspell the word `main`.
   e. Omit one of the `()`, then omit both the `()`.
   f. Continue in this fashion, deliberately misspelling identifiers (`cout`, `cin`, and so on). Omit one or both of the << in the `cout` statement; leave off the ending curly brace `}`.

7. Write a program that prints out C S ! in large block letters inside a border of *s followed by two blank lines then the message Computer Science is Cool Stuff. The output should look as follows:


```
**************************************************

     C C C           S S S S       !!
   C       C       S         S     !!
  C               S             S  !!
 C                S              S !!
 C                 S             S !!
 C                  S S S S      S !!
 C                               S !!
  C                             S  !!
   C       C       S         S
     C C C           S S S S       00

**************************************************
    Computer Science is Cool Stuff!!!
```

M01_SAVI1346_07_SB_C01.fm Page 38 Friday, January 4, 2008 5:01 PM

8. Write a program that allows the user to enter a number of quarters, dimes, and nickels and then outputs the monetary value of the coins in cents. For example, if the user enters 2 for the number of quarters, 3 for the number of dimes, and 1 for the number of nickels, then the program should output that the coins are worth 85 cents.

9. Write a program that allows the user to enter a time in seconds and then outputs how far an object would drop if it is in freefall for that length of time. Assume that the object starts at rest, there is no friction or resistance from air, and there is a constant acceleration of 32 feet per second due to gravity. Use the equation:

$$\text{distance} = \frac{\text{acceleration} \times \text{time}^2}{2}$$

   You should first compute the product and then divide the result by 2 (The reason for this will be discussed later in the book).