



# 1

## **AN OVERVIEW OF COMPUTERS AND LOGIC**

### **After studying Chapter 1, you should be able to:**

- Understand computer components and operations
- Describe the steps involved in the programming process
- Describe the data hierarchy
- Understand how to use flowchart symbols and pseudocode statements
- Use and name variables
- Use a sentinel, or dummy value, to end a program
- Use a connector symbol
- Assign values to variables
- Recognize the proper format of assignment statements
- Describe data types
- Understand the evolution of programming techniques

## UNDERSTANDING COMPUTER COMPONENTS AND OPERATIONS

Hardware and software are the two major components of any computer system. **Hardware** is the equipment, or the devices, associated with a computer. For a computer to be useful, however, it needs more than equipment; a computer needs to be given instructions. The instructions that tell the computer what to do are called **software**, or programs, and are written by programmers. This book focuses on the process of writing these instructions.

### TIP



Software can be classified as application software or system software. Application software comprises all the programs you apply to a task—word-processing programs, spreadsheets, payroll and inventory programs, and even games. System software comprises the programs that you use to manage your computer—operating systems, such as Windows, Linux, or UNIX. This book focuses on the logic used to write application software programs, although many of the concepts apply to both types of software.

Together, computer hardware and software accomplish four major operations:

1. Input
2. Processing
3. Output
4. Storage

Hardware devices that perform **input** include keyboards and mice. Through these devices, **data**, or facts, enter the computer system. **Processing** data items may involve organizing them, checking them for accuracy, or performing mathematical operations on them. The piece of hardware that performs these sorts of tasks is the **central processing unit**, or **CPU**. After data items have been processed, the resulting information is sent to a printer, monitor, or some other **output** device so people can view, interpret, and use the results. Often, you also want to store the output information on storage hardware, such as magnetic disks, tapes, compact discs, or flash media. Computer software consists of all the instructions that control how and when the data items are input, how they are processed, and the form in which they are output or stored.

### TIP



Data includes all the text, numbers, and other information that are processed by a computer. However, many computer professionals reserve the term “information” for data that has been processed. For example, your name, Social Security number, and hourly pay rate are data items, but your paycheck holds information.

Computer hardware by itself is useless without a programmer’s instructions, or software, just as your stereo equipment doesn’t do much until you provide music on a CD or tape. You can buy prewritten software that is stored on a disk or that you download from the Internet, or you can write your own software instructions. You can enter instructions into a computer system through any of the hardware devices you use for data; most often, you type your instructions using a keyboard and store them on a device such as a disk or CD.

You write computer instructions in a computer **programming language**, such as Visual Basic, C#, C++, Java, or COBOL. Just as some people speak English and others speak Japanese, programmers also write programs in different

languages. Some programmers work exclusively in one language, whereas others know several and use the one that seems most appropriate for the task at hand.

No matter which programming language a computer programmer uses, the language has rules governing its word usage and punctuation. These rules are called the language's **syntax**. If you ask, "How the get to store do I?" in English, most people can figure out what you probably mean, even though you have not used proper English syntax. However, computers are not nearly as smart as most people; with a computer, you might as well have asked, "Xpu mxv ot dodnm cadf B?" Unless the syntax is perfect, the computer cannot interpret the programming language instruction at all.

Every computer operates on circuitry that consists of millions of on/off switches. Each programming language uses a piece of software to translate the specific programming language into the computer's on/off circuitry language, or **machine language**. The language translation software is called a **compiler** or **interpreter**, and it tells you if you have used a programming language incorrectly. Therefore, syntax errors are relatively easy to locate and correct—the compiler or interpreter you use highlights every syntax error. If you write a computer program using a language such as C++ but spell one of its words incorrectly or reverse the proper order of two words, the translator lets you know that it found a mistake by displaying an error message as soon as you try to translate the program.

## TIP

Although there are differences in how compilers and interpreters work, their basic function is the same—to translate your programming statements into code the computer can use. When you use a compiler, an entire program is translated before it can execute; when you use an interpreter, each instruction is translated just prior to execution. Usually, you do not choose which type of translation to use—it depends on the programming language. However, there are some languages for which both compilers and interpreters are available.

A program without syntax errors can be executed on a computer, but it might not produce correct results. For a program to work properly, you must give the instructions to the computer in a specific sequence, you must not leave any instructions out, and you must not add extraneous instructions. By doing this, you are developing the **logic** of the computer program. Suppose you instruct someone to make a cake as follows:

```
Stir
Add two eggs
Add a gallon of gasoline
Bake at 350 degrees for 45 minutes
Add three cups of flour
```

Even though you have used the English language syntax correctly, the instructions are out of sequence, some instructions are missing, and some instructions belong to procedures other than baking a cake. If you follow these instructions, you are not going to end up with an edible cake, and you may end up with a disaster. Logical errors are much more difficult to locate than syntax errors; it is easier for you to determine whether "eggs" is spelled incorrectly in a recipe than it is for you to tell if there are too many eggs or if they are added too soon.

## TIP

Programmers often call logical errors semantic errors. For example, if you misspell a programming language word, you commit a syntax error, but if you use an otherwise correct word that does not make any sense in the current context, you commit a **semantic error**.

Just as baking directions can be given correctly in French, German, or Spanish, the same logic of a program can be expressed in any number of programming languages. This book is almost exclusively concerned with the logic development process. Because this book is not concerned with any specific language, the programming examples could have been written in Japanese, C++, or Java. The logic is the same in any language. For convenience, the book uses English!

Once instructions have been input to the computer and translated into machine language, a program can be **run**, or **executed**. You can write a program that takes a number (an input step), doubles it (processing), and tells you the answer (output) in a programming language such as Java or C++, but if you were to write it using English-like statements, it would look like this:

```
Get inputNumber.
Compute calculatedAnswer as inputNumber times 2.
Print calculatedAnswer.
```

**TIP** □ □ □ □ You will learn about the odd elimination of the space between words like “input” and “Number” and “calculated” and “Answer” in the next few pages.

The instruction to `Get inputNumber` is an example of an input operation. When the computer interprets this instruction, it knows to look to an input device to obtain a number. Computers often have several input devices, perhaps a keyboard, a mouse, a CD drive, and two or more disk drives. When you learn a specific programming language, you learn how to tell the computer which of those input devices to access for input. Logically, however, it doesn't really matter which hardware device is used, as long as the computer knows to look for a number. The logic of the input operation—that the computer must obtain a number for input, and that the computer must obtain it before multiplying it by two—remains the same regardless of any specific input hardware device. The same is true in your daily life. If you follow the instruction “Get eggs from store,” it does not really matter if you are following a handwritten instruction from a list or a voice-mail instruction left on your cell phone—the process of getting the eggs, and the result of doing so, are the same.

**TIP** □ □ □ □ Many computer professionals categorize disk drives and CD drives as storage devices rather than input devices. Such devices actually can be used for input, storage, and output.

Processing is the step that occurs when the arithmetic is performed to double the `inputNumber`; the statement `Compute calculatedAnswer as inputNumber times 2` represents processing. Mathematical operations are not the only kind of processing, but they are very typical. After you write a program, the program can be used on computers of different brand names, sizes, and speeds. Whether you use an IBM, Macintosh, Linux, or UNIX operating system, and whether you use a personal computer that sits on your desk or a mainframe that costs hundreds of thousands of dollars and resides in a special building in a university, multiplying by 2 is the same process. The hardware is not important; the processing will be the same.

In the number-doubling program, the `Print calculatedAnswer` statement represents output. Within a particular program, this statement could cause the output to appear on the monitor (which might be a flat panel screen or a cathode-ray tube), or the output could go to a printer (which could be laser or ink-jet), or the output could be written to a disk or CD. The logic of the process called “Print” is the same no matter what hardware device you use.

Besides input, processing, and output, the fourth operation in any computer system is storage. When computers produce output, it is for human consumption. For example, output might be displayed on a monitor or sent to a printer. Storage, on the other hand, is meant for future computer use (for example, when data items are saved on a disk).

Computer storage comes in two broad categories. All computers have **internal storage**, often referred to as **memory**, **main memory**, **primary memory**, or **random access memory (RAM)**. This storage is located inside the system unit of the machine. (For example, if you own a microcomputer, the system unit is the large case that holds your CD or other disk drives. On a laptop computer, the system unit is located beneath the keyboard.) Internal storage is the type of storage most often discussed in this book.

Computers also use **external storage**, which is **persistent** (relatively permanent) storage on a device such as a floppy disk, hard disk, flash media, or magnetic tape. In other words, external storage is outside the main memory, not necessarily outside the computer. Both programs and data sometimes are stored on each of these kinds of media.

To use computer programs, you must first load them into memory. You might type a program into memory from the keyboard, or you might use a program that has already been written and stored on a disk. Either way, a copy of the instructions must be placed in memory before the program can be run.

A computer system needs both internal memory and external storage. Internal memory is needed to run the programs, but internal memory is **volatile**—that is, its contents are lost every time the computer loses power. Therefore, if you are going to use a program more than once, you must store it, or **save** it, on some nonvolatile medium. Otherwise, the program in main memory is lost forever when the computer is turned off. External storage (usually disks or tape) provides a nonvolatile (or persistent) medium.

## TIP □ □ □ □

Even though a hard disk drive is located inside your computer, the hard disk is not main, internal memory. Internal memory is temporary and volatile; a hard drive is permanent, nonvolatile storage. After one or two “tragedies” of losing several pages of a typed computer program due to a power failure or other hardware problem, most programmers learn to periodically save the programs they are in the process of writing, using a nonvolatile medium such as a disk.

Once you have a copy of a program in main memory, you want to execute, or run, the program. To do so, you must also place any data that the program requires into memory. For example, after you place the following program into memory and start to run it, you need to provide an actual **inputNumber**—for example, 8—that you also place in main memory.

```
Get inputNumber.  
Compute calculatedAnswer as inputNumber times 2.  
Print calculatedAnswer.
```

The **inputNumber** is placed in memory at a specific memory location that the program will call **inputNumber**. Then, and only then, can the **calculatedAnswer**, in this case 16, be calculated and printed.

**TIP**

Computer memory consists of millions of numbered locations where data can be stored. The memory location of `inputNumber` has a specific numeric address, for example, 48604. Your program associates `inputNumber` with that address. Every time you refer to `inputNumber` within a program, the computer retrieves the value at the associated memory location. When you write programs, you seldom need to be concerned with the value of the memory address; instead, you simply use the easy-to-remember name you created.

Computer programmers often refer to memory addresses using hexadecimal notation, or base 16. Using this system, they might use a value like 42FF01A to refer to a memory address. Despite the use of letters, such an address is still a number. When you use the hexadecimal numbering system, the letters A through F stand for the values 10 through 15.

## UNDERSTANDING THE PROGRAMMING PROCESS

A programmer's job involves writing instructions (such as the three instructions in the doubling program in the preceding section), but a professional programmer usually does not just sit down at a computer keyboard and start typing. The programmer's job can be broken down into six programming steps:

1. Understanding the problem
2. Planning the logic
3. Coding the program
4. Using software to translate the program into machine language
5. Testing the program
6. Putting the program into production

## UNDERSTANDING THE PROBLEM

Professional computer programmers write programs to satisfy the needs of others. Examples could include a Human Resources Department that needs a printed list of all employees, a Billing Department that wants a list of clients who are 30 or more days overdue on their payments, and an office manager who wants to be notified when specific supplies reach the reorder point. Because programmers are providing a service to these users, programmers must first understand what it is the users want.

Suppose the director of human resources says to a programmer, "Our department needs a list of all employees who have been here over five years, because we want to invite them to a special thank-you dinner." On the surface, this seems like a simple enough request. An experienced programmer, however, will know that he or she may not yet understand the whole problem. Does the director want a list of full-time employees only, or a list of full- and part-time employees together? Does she want people who have worked for the company on a month-to-month contractual basis over the past five years, or only regular, permanent employees? Do the listed employees need to have worked for the organization for five years as of today, as of the date of the dinner, or as of some other cutoff date? What about an employee who worked three years, took a two-year leave of absence, and has been back for three years? Does he or she qualify? The programmer cannot make any of these decisions; the user is the one who must address these questions.

More decisions still might be required. For example, what does the user want the report of five-year employees to look like? Should it contain both first and last names? Social Security numbers? Phone numbers? Addresses? Is all this data available? Several pieces of documentation are often provided to help the programmer understand the problem. This documentation includes print layout charts and file specifications, which you will learn about in Chapter 3.

Really understanding the problem may be one of the most difficult aspects of programming. On any job, the description of what the user needs may be vague—worse yet, the user may not even really know what he or she wants, and users who think they know what they want frequently change their minds after seeing sample output. A good programmer is often part counselor, part detective!

## **PLANNING THE LOGIC**

The heart of the programming process lies in planning the program's logic. During this phase of the programming process, the programmer plans the steps of the program, deciding what steps to include and how to order them. You can plan the solution to a problem in many ways. The two most common planning tools are flowcharts and pseudocode. Both tools involve writing the steps of the program in English, much as you would plan a trip on paper before getting into the car, or plan a party theme before going shopping for food and favors.

### **TIP**



You may hear programmers refer to planning a program as “developing an algorithm.” An **algorithm** is the sequence of steps necessary to solve any problem. You will learn more about flowcharts and pseudocode later in this chapter.

The programmer doesn't worry about the syntax of any particular language at this point, just about figuring out what sequence of events will lead from the available input to the desired output. Planning the logic includes thinking carefully about all the possible data values a program might encounter and how you want the program to handle each scenario. The process of walking through a program's logic on paper before you actually write the program is called **desk-checking**. You will learn more about planning the logic later; in fact, this book focuses on this crucial step almost exclusively.

## **CODING THE PROGRAM**

Once the programmer has developed the logic of a program, only then can he or she write the program in one of more than 400 programming languages. Programmers choose a particular language because some languages have built-in capabilities that make them more efficient than others at handling certain types of operations. Despite their differences, programming languages are quite alike—each can handle input operations, arithmetic processing, output operations, and other standard functions. The logic developed to solve a programming problem can be executed using any number of languages. It is only after a language is chosen that the programmer must worry about each command being spelled correctly and all of the punctuation getting into the right spots—in other words, using the correct *syntax*.

Some very experienced programmers can successfully combine the logic planning and the actual instruction writing, or **coding**, of the program in one step. This may work for planning and writing a very simple program, just as you can plan and write a postcard to a friend using one step. A good term paper or a Hollywood screenplay, however, needs planning before writing, and so do most programs.

Which step is harder, planning the logic or coding the program? Right now, it may seem to you that writing in a programming language is a very difficult task, considering all the spelling and grammar rules you must learn. However, the planning step is actually more difficult. Which is more difficult: thinking up the twists and turns to the plot of a best-selling mystery novel, or writing a translation of an already written novel from English to Spanish? And who do you think gets paid more, the writer who creates the plot or the translator? (Try asking friends to name any famous translator!)

## **USING SOFTWARE TO TRANSLATE THE PROGRAM INTO MACHINE LANGUAGE**

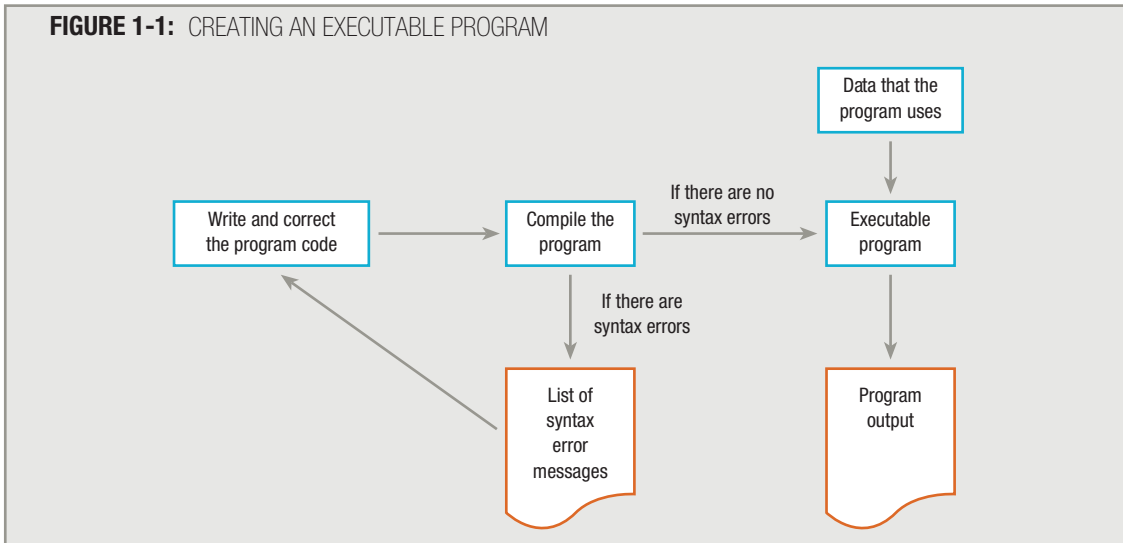
Even though there are many programming languages, each computer knows only one language, its machine language, which consists of many 1s and 0s. Computers understand machine language because computers themselves are made up of thousands of tiny electrical switches, each of which can be set in either the on or off state, which is represented by a 1 or 0, respectively.

Languages like Java or Visual Basic are available for programmers to use because someone has written a translator program (a compiler or interpreter) that changes the English-like **high-level programming language** in which the programmer writes into the **low-level machine language** that the computer understands. If you write a programming language statement incorrectly (for example, by misspelling a word, using a word that doesn't exist in the language, or using "illegal" grammar), the translator program doesn't know what to do and issues an error message identifying a **syntax error**, or misuse of a language's grammar rules. You receive the same response when you speak nonsense to a human-language translator. Imagine trying to look up a list of words in a Spanish-English dictionary if some of the listed words are misspelled—you can't complete the task until the words are spelled correctly. Although making errors is never desirable, syntax errors are not a major concern to programmers, because the compiler or interpreter catches every syntax error, and the computer will not execute a program that contains them.

A computer program must be free of syntax errors before you can execute it. Typically, a programmer develops a program's logic, writes the code, and then compiles the program, receiving a list of syntax errors. The programmer then corrects the syntax errors, and compiles the program again. Correcting the first set of errors frequently reveals a new set of errors that originally were not apparent to the compiler. For example, if you could use an English compiler and submit the sentence `The gr1 go to school`, the compiler at first would point out only one syntax error to you. The second word, `gr1`, is illegal because it is not part of the English language. Only after you corrected the word `girl` would the compiler find another syntax error on the third word, `go`, because it is the wrong verb form for the subject `girl`. This doesn't mean `go` is necessarily the wrong word. Maybe `girl` is wrong; perhaps the subject should be `girls`, in which case `go` is right. Compilers don't always know exactly what you mean, nor do they know what the proper correction should be, but they do know when something is wrong with your syntax.

When writing a program, a programmer might need to recompile the code several times. An executable program is created only when the code is free of syntax errors. When you run an executable program, it typically also might require input data. Figure 1-1 shows a diagram of this entire process.



**FIGURE 1-1:** CREATING AN EXECUTABLE PROGRAM

## **TESTING THE PROGRAM**

A program that is free of syntax errors is not necessarily free of **logical errors**. For example, the sentence `The girl goes to school`, although syntactically perfect, is not logically correct if the girl is a baby or a dropout.

Once a program is free from syntax errors, the programmer can test it—that is, execute it with some sample data to see whether the results are logically correct. Recall the number-doubling program:

```
Get inputNumber.  
Compute calculatedAnswer as inputNumber times 2.  
Print calculatedAnswer.
```

If you provide the value 2 as input to the program and the answer 4 prints, you have executed one successful test run of the program.

However, if the answer 40 prints, maybe it's because the program contains a logical error. Maybe the second line of code was mistyped with an extra zero, so that the program reads:

```
Get inputNumber.  
Compute calculatedAnswer as inputNumber times 20.  
Print calculatedAnswer.
```

The error of placing 20 instead of 2 in the multiplication statement caused a logical error. Notice that nothing is syntactically wrong with this second program—it is just as reasonable to multiply a number by 20 as by 2—but if the programmer intends only to double the `inputNumber`, then a logical error has occurred.

Programs should be tested with many sets of data. For example, if you write the program to double a number and enter 2 and get an output value of 4, that doesn't necessarily mean you have a correct program. Perhaps you have typed this program by mistake:

```
Get inputNumber.
Compute calculatedAnswer as inputNumber plus 2.
Print calculatedAnswer.
```

An input of 2 results in an answer of 4, but that doesn't mean your program doubles numbers—it actually only adds 2 to them. If you test your program with additional data and get the wrong answer—for example, if you use a 3 and get an answer of 5—you know there is a problem with your code.

Selecting test data is somewhat of an art in itself, and it should be done carefully. If the Human Resources Department wants a list of the names of five-year employees, it would be a mistake to test the program with a small sample file of only long-term employees. If no newer employees are part of the data being used for testing, you don't really know if the program would have eliminated them from the five-year list. Many companies don't know that their software has a problem until an unusual circumstance occurs—for example, the first time an employee has more than nine dependents, the first time a customer orders more than 999 items at a time, or when (in an example that was well-documented in the popular press) a new century begins.

## **PUTTING THE PROGRAM INTO PRODUCTION**

Once the program is tested adequately, it is ready for the organization to use. Putting the program into production might mean simply running the program once, if it was written to satisfy a user's request for a special list. However, the process might take months if the program will be run on a regular basis, or if it is one of a large system of programs being developed. Perhaps data-entry people must be trained to prepare the input for the new program, users must be trained to understand the output, or existing data in the company must be changed to an entirely new format to accommodate this program. **Conversion**, the entire set of actions an organization must take to switch over to using a new program or set of programs, can sometimes take months or years to accomplish.

### **TIP**



You might consider maintaining programs as a seventh step in the programming process. After programs are put into production, making required changes is called maintenance. Maintenance is necessary for many reasons: for example, new tax rates are legislated, the format of an input file is altered, or the end user requires additional information not included in the original output specifications. Frequently, your first programming job will require maintaining previously written programs. When you maintain the programs others have written, you will appreciate the effort the original programmer put into writing clear code, using reasonable variable names, and documenting his or her work.

You might consider retiring the program as the eighth and final step in the programming process. A program is retired when it is no longer needed by an organization—usually when a new program is in the process of being put into production.

## UNDERSTANDING THE DATA HIERARCHY

Some very simple programs require very simple data. For example, the number-doubling program requires just one value as input. Most business programs, however, use much more data—inventory files list thousands of items, personnel and customer files list thousands of people. When data items are stored for use on computer systems, they are often stored in what is known as a **data hierarchy**, where the smallest usable unit of data is the character. **Characters** are letters, numbers, and special symbols, such as “A”, “7”, and “\$”. Anything you can type from the keyboard in one keystroke (including a space or a tab) is a character. Characters are made up of smaller elements called bits, but just as most human beings can use a pencil without caring whether atoms are flying around inside it, most computer users can store characters without caring about these bits.

**TIP** □ □ □ □ Computers also recognize characters you cannot enter from the keyboard, such as foreign alphabet characters like  $\phi$  or  $\Sigma$ .

Characters are grouped together to form a field. A **field** is a single data item, such as `lastName`, `streetAddress`, or `annualSalary`. For most of us, an “S”, an “m”, an “i”, a “t”, and an “h” don’t have much meaning individually, but if the combination of characters makes up your last name, “Smith”, then as a group, the characters have useful meaning.

Related fields are often grouped together to form a record. **Records** are groups of fields that go together for some logical reason. A random name, address, and salary aren’t very useful, but if they’re your name, your address, and your salary, then that’s your record. An inventory record might contain fields for item number, color, size, and price; a student record might contain ID number, grade point average, and major.

Related records, in turn, are grouped together to form a file. **Files** are groups of records that go together for some logical reason. The individual records of each student in your class might go together in a file called STUDENTS. Records of each person at your company might be in a file called PERSONNEL. Items you sell might be in an INVENTORY file.

Some files can have just a few records; others, such as the file of credit-card holders for a major department-store chain or policyholders of an insurance company, can contain thousands or even millions of records.

Finally, many organizations use database software to organize many files. A **database** holds a group of files, often called **tables**, that together serve the information needs of an organization. Database software establishes and maintains relationships between fields in these tables, so that users can write questions called **queries**. Queries pull related data items together in a format that allows businesspeople to make managerial decisions efficiently. Chapter 16 of the Comprehensive version of this text covers database creation.

In summary, you can picture the data hierarchy, as shown in Figure 1-2.

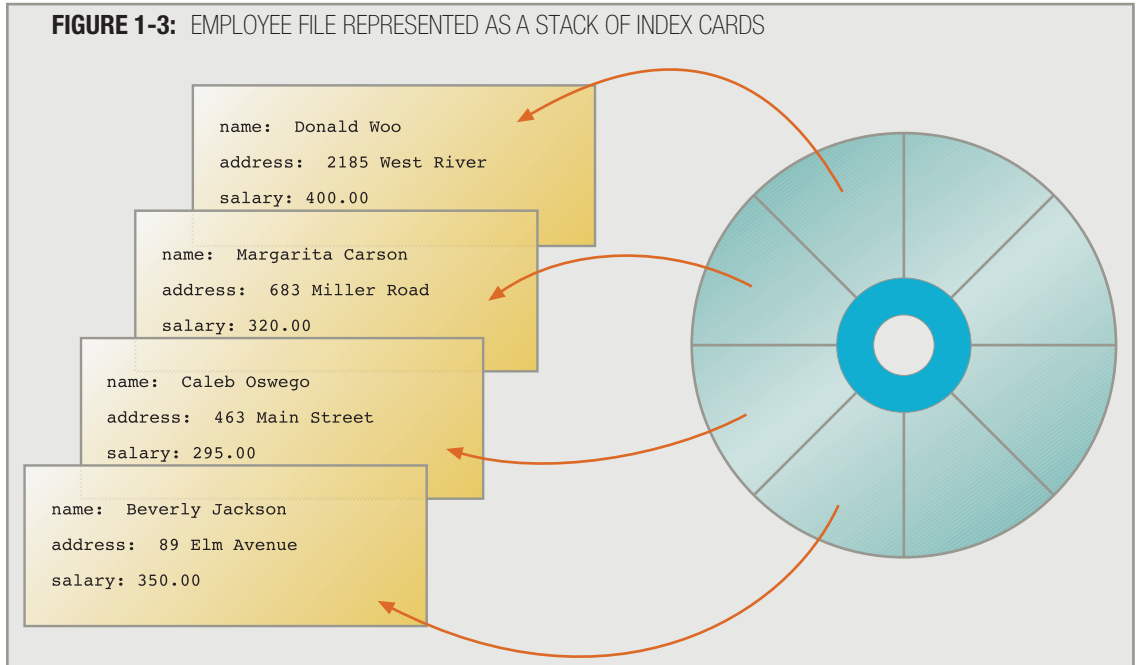
**FIGURE 1-2:** THE DATA HIERARCHY

```

Database
  File
    Record
      Field
        Character
  
```

A database contains many files. A file contains many records. Each record in a file has the same fields. Each record's fields contain different data items that consist of one or more stored characters in each field.

As an example, you can picture a file as a set of index cards, as shown in Figure 1-3. The stack of cards is the EMPLOYEE file, in which each card represents one employee record. On each card, each line holds one field—**name**, **address**, or **salary**. Almost all the program examples in this book use files that are organized in this way.



## USING FLOWCHART SYMBOLS AND PSEUDOCODE STATEMENTS

When programmers plan the logic for a solution to a programming problem, they often use one of two tools, **flowcharts** or **pseudocode** (pronounced "sue-doe-code"). A flowchart is a pictorial representation of the logical steps it takes to solve a problem. **Pseudocode** is an English-like representation of the same thing. *Pseudo* is a prefix that means "false," and to *code* a program means to put it in a programming language; therefore, *pseudocode* simply means "false code," or sentences that appear to have been written in a computer programming language but don't necessarily follow all the syntax rules of any specific language.

You have already seen examples of statements that represent pseudocode earlier in this chapter, and there is nothing mysterious about them. The following five statements constitute a pseudocode representation of a number-doubling problem:

```
start
  get inputNumber
  compute calculatedAnswer as inputNumber times 2
  print calculatedAnswer
stop
```

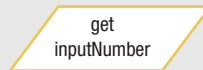
Using pseudocode involves writing down all the steps you will use in a program. Usually, programmers preface their pseudocode statements with a beginning statement like “start” and end them with a terminating statement like “stop”. The statements between “start” and “stop” look like English and are indented slightly so that “start” and “stop” stand out. Most programmers do not bother with punctuation such as periods at the end of pseudocode statements, although it would not be wrong to use them if you prefer that style. Similarly, there is no need to capitalize the first word in a sentence, although you might choose to do so. This book follows the conventions of using lowercase letters for verbs that begin pseudocode statements and omitting periods at the end of statements.

Some professional programmers prefer writing pseudocode to drawing flowcharts, because using pseudocode is more similar to writing the final statements in the programming language. Others prefer drawing flowcharts to represent the logical flow, because flowcharts allow programmers to visualize more easily how the program statements will connect. Especially for beginning programmers, flowcharts are an excellent tool to help visualize how the statements in a program are interrelated.

Almost every program involves the steps of input, processing, and output. Therefore, most flowcharts need some graphical way to separate these three steps. When you create a flowchart, you draw geometric shapes around the individual statements and connect them with arrows.

When you draw a flowchart, you use a parallelogram to represent an **input symbol**, which indicates an input operation. You write an input statement, in English, inside the parallelogram, as shown in Figure 1-4.

**FIGURE 1-4:** INPUT SYMBOL



## TIP

When you want to represent entering two or more values in a program, you can use one or multiple flowchart symbols or pseudocode statements—whichever seems more reasonable and clear to you. For example, the pseudocode to input a user’s name and address might be written as:

```
get inputName
get inputAddress
```

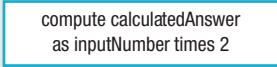
or as:

```
get inputName, inputAddress
```

The first version implies two separate input operations, whereas the second implies a single input operation retrieving two data items. If your application will accept user input from a keyboard, using two separate input statements might make sense, because the user will type one item at a time. If your application will accept data from a storage device, obtaining all the data at once is more common. Logically, either format represents the retrieval of two data items. The end result is the same in both cases—after the statements have executed, `inputName` and `inputAddress` will have received values from an input device.

Arithmetic operation statements are examples of processing. In a flowchart, you use a rectangle as the **processing symbol** that contains a processing statement, as shown in Figure 1-5.

**FIGURE 1-5:** PROCESSING SYMBOL



```
compute calculatedAnswer
as inputNumber times 2
```

To represent an output statement, you use the same symbol as for input statements—the **output symbol** is a parallelogram, as shown in Figure 1-6.

**FIGURE 1-6:** OUTPUT SYMBOL



```
print
calculatedAnswer
```

## TIP



As with input, output statements can be organized in whatever way seems most reasonable. A program that prints the length and width of a room might use the statement:

```
print length
print width
```

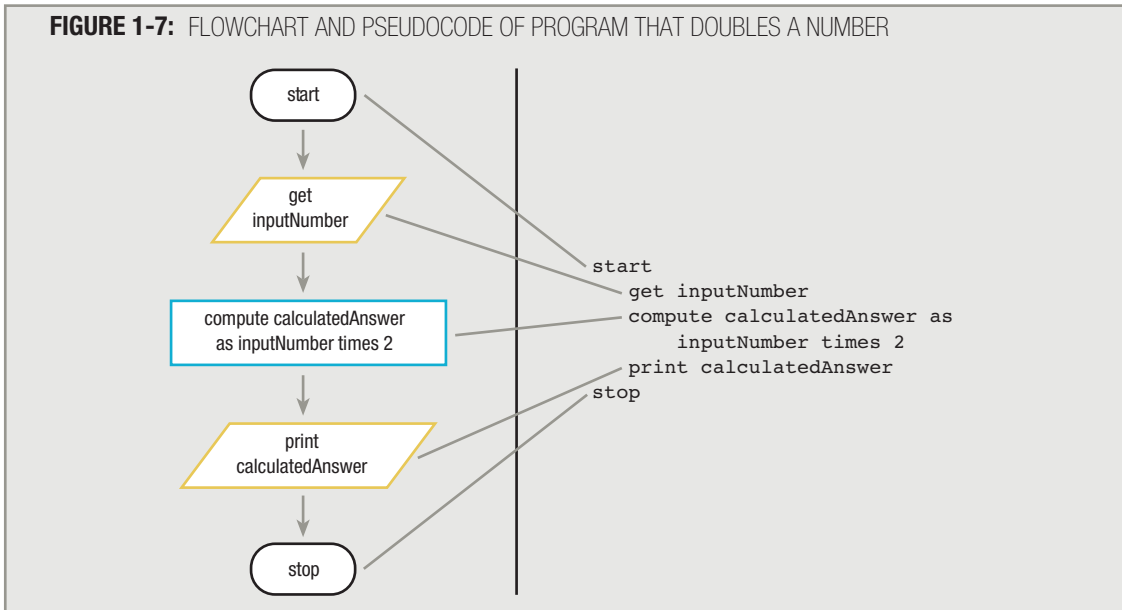
or:

```
print length, width
```

In some programming languages, using two print statements places the output values on two separate lines on the monitor or printer, whereas using a single print statement places the values next to each other on the same line. This book follows the convention of using one print statement per line of output.

To show the correct sequence of these statements, you use arrows, or **flowlines**, to connect the steps. Whenever possible, most of a flowchart should read from top to bottom or from left to right on a page. That's the way we read English, so when flowcharts follow this convention, they are easier for us to understand.

To be complete, a flowchart should include two more elements: a **terminal symbol**, or start/stop symbol, at each end. Often, you place a word like “start” or “begin” in the first terminal symbol and a word like “end” or “stop” in the other. The standard terminal symbol is shaped like a racetrack; many programmers refer to this shape as a **lozenge**, because it resembles the shape of a medicated candy lozenge you might use to soothe a sore throat. Figure 1-7 shows a complete flowchart for the program that doubles a number, and the pseudocode for the same problem.

**FIGURE 1-7:** FLOWCHART AND PSEUDOCODE OF PROGRAM THAT DOUBLES A NUMBER

**TIP** □ □ □ □ | Programmers seldom create both pseudocode and a flowchart for the same problem. You usually use one or the other.

The logic for the program represented by the flowchart and pseudocode in Figure 1-7 is correct no matter what programming language the programmer eventually uses to write the corresponding code. Just as the same statements could be translated into Italian or Chinese without losing their meaning, they also can be coded in C#, Java, or any other programming language.

After the flowchart or pseudocode has been developed, the programmer only needs to: (1) buy a computer, (2) buy a language compiler, (3) learn a programming language, (4) code the program, (5) attempt to compile it, (6) fix the syntax errors, (7) compile it again, (8) test it with several sets of data, and (9) put it into production.

“Whoa!” you are probably saying to yourself. “This is simply not worth it! All that work to create a flowchart or pseudocode, and *then* all those other steps? For five dollars, I can buy a pocket calculator that will double any number for me instantly!” You are absolutely right. If this were a real computer program, and all it did was double the value of a number, it simply would not be worth all the effort. Writing a computer program would be worth the effort only if you had many—let’s say 10,000—numbers to double in a limited amount of time—let’s say the next two minutes. Then, it would be worth your while to create a computer program.

Unfortunately, the number-doubling program represented in Figure 1-7 does not double 10,000 numbers; it doubles only one. You could execute the program 10,000 times, of course, but that would require you to sit at the computer telling it to run the program over and over again. You would be better off with a program that could process 10,000 numbers, one after the other.

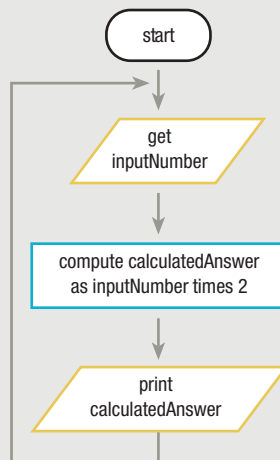
One solution is to write the program as shown in Figure 1-8 and execute the same steps 10,000 times. Of course, writing this program would be very time-consuming; you might as well buy the calculator.

**FIGURE 1-8:** INEFFICIENT PSEUDOCODE FOR PROGRAM THAT DOUBLES 10,000 NUMBERS

```
start
  get inputNumber
  compute calculatedAnswer as inputNumber times 2
  print calculatedAnswer
  get inputNumber
  compute calculatedAnswer as inputNumber times 2
  print calculatedAnswer
  get inputNumber
  compute calculatedAnswer as inputNumber times 2
  print calculatedAnswer
  . . . and so on
```

A better solution is to have the computer execute the same set of three instructions over and over again, as shown in Figure 1-9. With this approach, the computer gets a number, doubles it, prints the answer, and then starts over again with the first instruction. The same spot in memory, called `inputNumber`, is reused for the second number and for any subsequent numbers. The spot in memory named `calculatedAnswer` is reused each time to store the result of the multiplication operation. The logic illustrated in the flowchart shown in Figure 1-9 contains a major problem—the sequence of instructions never ends. You will learn to handle this problem later in this chapter.

**FIGURE 1-9:** FLOWCHART OF INFINITE NUMBER-DOUBLING PROGRAM





## USING AND NAMING VARIABLES

Programmers commonly refer to the locations in memory called `inputNumber` and `calculatedAnswer` as variables. **Variables** are memory locations, whose contents can vary or differ over time. Sometimes, `inputNumber` can hold a 2 and `calculatedAnswer` will hold a 4; at other times, `inputNumber` can hold a 6 and `calculatedAnswer` will hold a 12. It is the ability of memory variables to change in value that makes computers and programming worthwhile. Because one memory location can be used over and over again with different values, you can write program instructions once and then use them for thousands of separate calculations. *One* set of payroll instructions at your company produces each individual's paycheck, and *one* set of instructions at your electric company produces each household's bill.

The number-doubling example requires two variables, `inputNumber` and `calculatedAnswer`. These can just as well be named `userEntry` and `programSolution`, or `inputValue` and `twiceTheValue`. As a programmer, you choose reasonable names for your variables. The language interpreter then associates the names you choose with specific memory addresses.

A variable name is also called an **identifier**. Every computer programming language has its own set of rules for naming identifiers. Most languages allow both letters and digits within variable names. Some languages allow hyphens in variable names—for example, `hourly-wage`. Others allow underscores, as in `hourly_wage`. Still others allow neither. Some languages allow dollar signs or other special characters in variable names (for example, `hourly$`); others allow foreign alphabet characters, such as  $\pi$  or  $\Omega$ .

### TIP □ □ □ □

You also can refer to a variable name as a **mnemonic**. In everyday language, a mnemonic is a memory device, like the sentence “Every good boy does fine,” which makes it easier to remember the notes that occupy the lines on the staff in sheet music. In programming, a variable name is a device that makes it easier to reference a memory address.

### TIP □ □ □ □

Different languages put different limits on the length of variable names, although in general, newer languages allow longer names. For example, in some very old versions of BASIC, a variable name could consist of only one or two letters and one or two digits. You could have some cryptic variable names like `hw` or `a3` or `re02`. Fortunately, most modern languages allow variable names to be much longer; in the newest versions of C++, C#, and Java, the length of identifiers is virtually unlimited. Variable names in these languages usually consist of lowercase letters, don't allow hyphens, but do allow underscores, so you can use a name like `price_of_item`. These languages are case sensitive, so `HOURLYWAGE`, `hourlywage`, and `hourlyWage` are considered three separate variable names, although the last example, in which the new word begins with an uppercase letter, is easiest to read. Most programmers who use the more modern languages employ the format in which multiple-word variable names are run together, and each new word within the variable name begins with an uppercase letter. This format is called **camel casing**, because such variable names, like `hourlyWage`, have a “hump” in the middle. The variable names in this text are shown using camel casing.

Even though every language has its own rules for naming variables, when designing the logic of a computer program, you should not concern yourself with the specific syntax of any particular computer language. The logic, after all, works with any language. The variable names used throughout this book follow only two rules:

1. *Variable names must be one word.* The name can contain letters, digits, hyphens, underscores, or any other characters you choose, with the exception of *spaces*. Therefore, `x` is a legal variable name, as is `rate`, as is `interestRate`. The variable name `interest rate` is not allowed because of the space. No programming language allows spaces within a variable name. If you see a name such as `interest rate` in a flowchart or pseudocode, you should assume that the programmer is discussing two variables, `interest` and `rate`, each of which individually would be a fine variable name.

### TIP



As a convention, this book begins variable names with a lowercase letter. You might find programming texts in languages such as Visual Basic and C++ in which the author has chosen to begin variable names with an uppercase letter. As long as you adopt a convention and use it consistently, your programs will be easier to read and understand.

### TIP



When you write a program using an editor that is packaged with a compiler, the compiler may display variable names in a different color from the rest of the program. This visual aid helps your variable names stand out from words that are part of the programming language.

2. *Variable names should have some appropriate meaning.* This is not a rule of any programming language. When computing an interest rate in a program, the computer does not care if you call the variable `g`, `u84`, or `fred`. As long as the correct numeric result is placed in the variable, its actual name doesn't really matter. However, it's much easier to follow the logic of a program with a statement in it like `compute finalBalance as equal to initialInvestment times interestRate` than one with a statement in it like `compute someBanana as equal to j89 times myFriendLinda`. You might think you will remember how you intended to use a cryptic variable name within a program, but several months or years later when a program requires changes, you, and other programmers working with you, will appreciate clear, descriptive variable names.

Notice that the flowchart in Figure 1-9 follows these two rules for variables: both variable names, `inputNumber` and `calculatedAnswer`, are one word, and they have appropriate meanings. Some programmers have fun with their variable names by naming them after friends or creating puns with them, but such behavior is unprofessional and marks those programmers as amateurs. Table 1-1 lists some possible variable names that might be used to hold an employee's last name and provides a rationale for the appropriateness of each one.

### TIP



Another general rule in all programming languages is that variable names may not begin with a digit, although usually they may contain digits. Thus, in most languages `budget2013` is a legal variable name, but `2013Budget` is not.

**TABLE 1-1:** VALID AND INVALID VARIABLE NAMES FOR AN EMPLOYEE'S LAST NAME

Suggested variable names for employee's last name	Comments
<code>employeeLastName</code>	Good
<code>employeeLast</code>	Good—most people would interpret <code>Last</code> as meaning <code>Last Name</code>
<code>empLast</code>	Good— <code>emp</code> is short for employee
<code>emlstnam</code>	Legal—but cryptic
<code>lastNameOfTheEmployeeInQuestion</code>	Legal—but awkward
<code>last name</code>	Not legal—embedded space
<code>employeelastname</code>	Legal—but hard to read without camel casing

## ENDING A PROGRAM BY USING SENTINEL VALUES

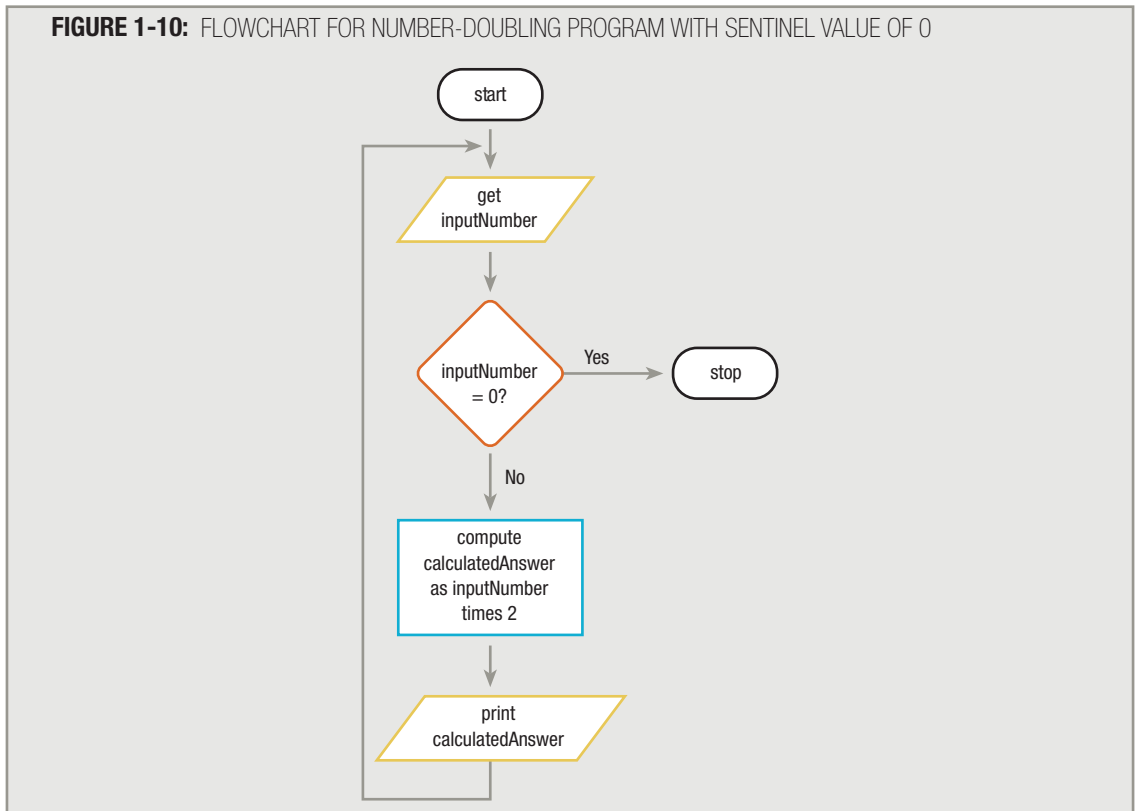
Recall that the logic in the flowchart for doubling numbers, shown in Figure 1-9, has a major flaw—the program never ends. This programming situation is known as an **infinite loop**—a repeating flow of logic with no end. If, for example, the input numbers are being entered at the keyboard, the program will keep accepting numbers and printing doubles forever. Of course, the user could refuse to type in any more numbers. But the computer is very patient, and if you refuse to give it any more numbers, it will sit and wait forever. When you finally type in a number, the program will double it, print the result, and wait for another. The program cannot progress any further while it is waiting for input; meanwhile, the program is occupying computer memory and tying up operating system resources. Refusing to enter any more numbers is not a practical solution. Another way to end the program is simply to turn the computer off. But again, that's neither the best nor an elegant way to bring the program to an end.

A superior way to end the program is to set a predetermined value for `inputNumber` that means “Stop the program!” For example, the programmer and the user could agree that the user will never need to know the double of 0 (zero), so the user could enter a 0 when he or she wants to stop. The program could then test any incoming value contained in `inputNumber` and, if it is a 0, stop the program. Testing a value is also called making a **decision**.

You represent a decision in a flowchart by drawing a **decision symbol**, which is shaped like a diamond. The diamond usually contains a question, the answer to which is one of two mutually exclusive options—often yes or no. All good computer questions have only two mutually exclusive answers, such as yes and no or true and false. For example, “What day of the year is your birthday?” is not a good computer question because there are 366 possible answers. But “Is your birthday June 24?” is a good computer question because, for everyone in the world, the answer is either yes or no.

**TIP** □ □ □ □ A yes-or-no decision is called a **binary decision**, because there are two possible outcomes.

The question to stop the doubling program should be “Is the `inputNumber` just entered equal to 0?” or “`inputNumber = 0?`” for short. The complete flowchart will now look like the one shown in Figure 1-10.

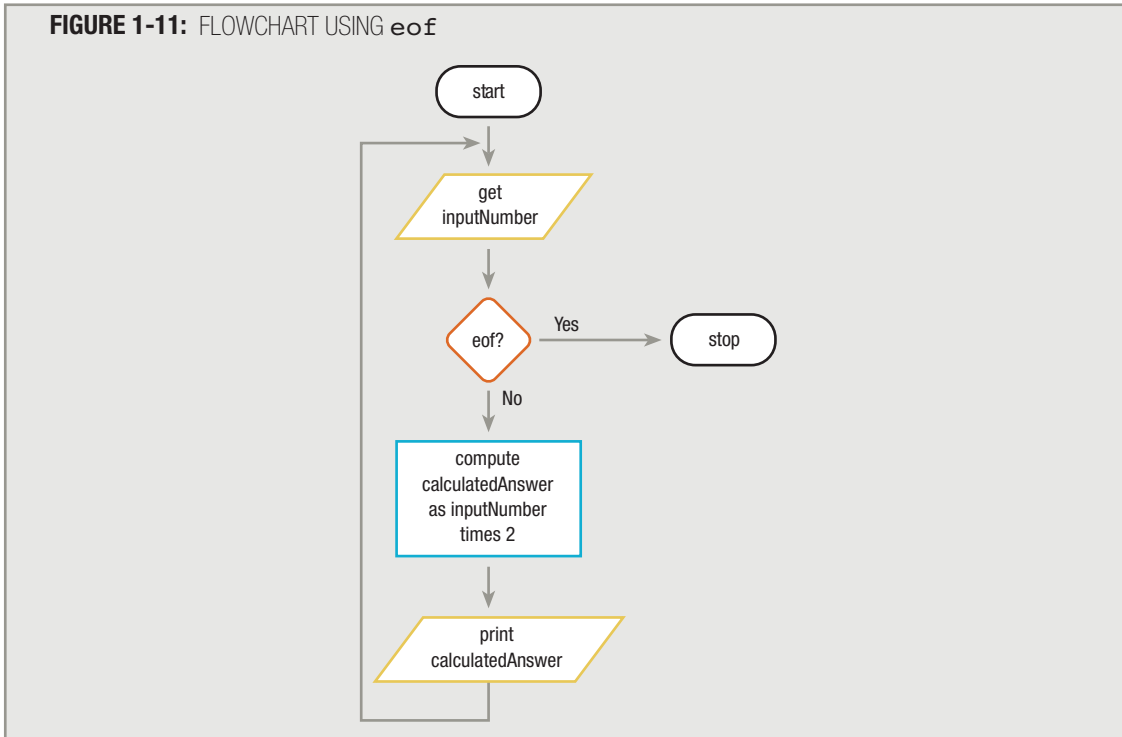


One drawback to using 0 to stop a program, of course, is that it won't work if the user does need to find the double of 0. In that case, some other data-entry value that the user never will need, such as 999 or  $-1$ , could be selected to signal that the program should end. A preselected value that stops the execution of a program is often called a **dummy value** because it does not represent real data, but just a signal to stop. Sometimes, such a value is called a **sentinel value** because it represents an entry or exit point, like a sentinel who guards a fortress.

Not all programs rely on user data entry from a keyboard; many read data from an input device, such as a disk or tape drive. When organizations store data on a disk or other storage device, they do not commonly use a dummy value to signal the end of the file. For one thing, an input record might have hundreds of fields, and if you store a dummy record in every file, you are wasting a large quantity of storage on “non-data.” Additionally, it is often difficult to choose sentinel values for fields in a company's data files. Any `balanceDue`, even a zero or a negative number, can be a legitimate value, and any `customerName`, even “ZZ”, could be someone's name. Fortunately, programming languages can

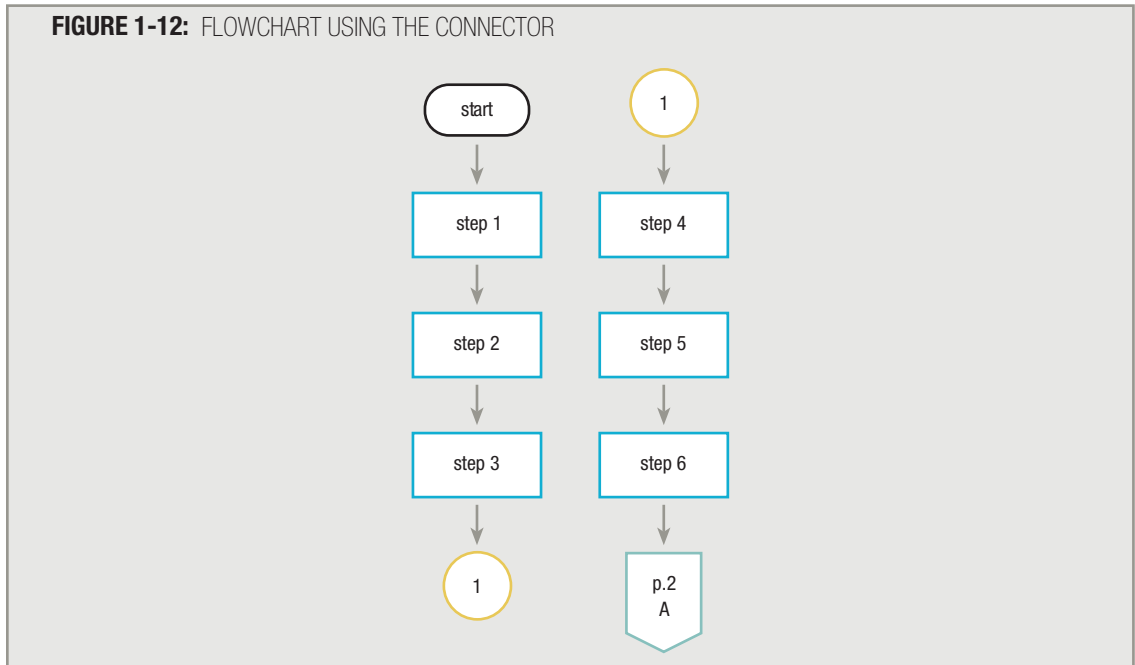
recognize the end of data in a file automatically, through a code that is stored at the end of the data. Many programming languages use the term **eof** (for “end of file”) to talk about this marker that automatically acts as a sentinel. This book, therefore, uses **eof** to indicate the end of data, regardless of whether the code is a special disk marker or a dummy value such as 0 that comes from the keyboard. Therefore, the flowchart and pseudocode can look like the examples shown in Figure 1-11.

**FIGURE 1-11:** FLOWCHART USING **eof**



## USING THE CONNECTOR

By using just the input, processing, output, decision, and terminal symbols, you can represent the flowcharting logic for many diverse applications. When drawing a flowchart segment, you might use another symbol, the **connector**. You can use a connector when limited page size forces you to continue a flowchart in an unconnected location or on another page. If a flowchart has six processing steps and a page provides room for only three, you might represent the logic as shown in Figure 1-12.



By convention, programmers use a circle as an on-page connector symbol, and a symbol that looks like a square with a pointed bottom as an off-page connector symbol. The on-page connector at the bottom of the left column in Figure 1-12 tells someone reading the flowchart that there is more to the flowchart. The circle should contain a number or letter that can then be matched to another number or letter somewhere else, in this case on the right. If a large flowchart needed more connectors, new numbers or letters would be assigned in sequence (1, 2, 3... or A, B, C...) to each successive pair of connectors. The off-page connector at the bottom of the right column in Figure 1-12 tells a reader that there is more to the flowchart on another page.

When you are creating your own flowcharts, you should avoid using any connectors, if at all possible; flowcharts are more difficult to follow when their segments do not fit together on a page. Some programmers would even say that if a flowchart must connect to another page, it is a sign of poor design. Your instructor or future programming supervisor may require that long flowcharts be redrawn so you don't need to use the connector symbol. However, when continuing to a new location or page is unavoidable, the connector provides the means.

## ASSIGNING VALUES TO VARIABLES

When you create a flowchart or pseudocode for a program that doubles numbers, you can include the statement `compute calculatedAnswer as inputNumber times 2`. This statement incorporates two actions. First, the computer calculates the arithmetic value of `inputNumber times 2`. Second, the computed value is

stored in the `calculatedAnswer` memory location. Most programming languages allow a shorthand expression for **assignment statements** such as `compute calculatedAnswer as inputNumber times 2`. The shorthand takes the form `calculatedAnswer = inputNumber * 2`. The equal sign is the **assignment operator**; it always requires the name of a memory location on its left side—the name of the location where the result will be stored.

## TIP

When they write pseudocode or draw a flowchart, most programmers use the asterisk (\*) to represent multiplication. When you write pseudocode, you can use an X or a dot for multiplication (as most mathematicians do), but you will be using an unconventional format. This book will always use an asterisk to represent multiplication.

According to the rules of algebra, a statement like `calculatedAnswer = inputNumber * 2` should be exactly equivalent to the statement `inputNumber * 2 = calculatedAnswer`. That's because in algebra, the equal sign always represents equivalency. In most programming languages, however, the equal sign represents assignment, and `calculatedAnswer = inputNumber * 2` means “multiply `inputNumber` by 2 and store the result in the variable called `calculatedAnswer`.” Whatever operation is performed to the right of the equal sign results in a value that is placed in the memory location to the left of the equal sign. Therefore, the incorrect statement `inputNumber * 2 = calculatedAnswer` means to attempt to take the value of `calculatedAnswer` and store it in a location called `inputNumber * 2`, but there can't be a location called `inputNumber * 2`. For one thing, you should recognize that the expression `inputNumber * 2` can't be a variable because it has spaces in it. For another, a location can't be multiplied. Its contents can be multiplied, but the location itself cannot be. The backward statement `inputNumber * 2 = calculatedAnswer` contains a syntax error, no matter what programming language you use; a program with such a statement will not execute.

## TIP

When you create an assignment statement, it may help to imagine the word “let” in front of the statement. Thus, you can read the statement `calculatedAnswer = inputNumber * 2` as “Let `calculatedAnswer` equal `inputNumber` times two.” The BASIC programming language allows you to use the word “let” in such statements. You might also imagine the word “gets” or “receives” in place of the assignment operator. In other words, `calculatedAnswer = inputNumber * 2` means both `calculatedAnswer gets inputNumber * 2` and `calculatedAnswer receives inputNumber * 2`.

Computer memory is made up of millions of distinct locations, each of which has an address. Fifty or sixty years ago, programmers had to deal with these addresses and had to remember, for instance, that they had stored a salary in location 6428 of their computer. Today, we are very fortunate that high-level computer languages allow us to pick a reasonable “English” name for a memory address and let the computer keep track of where it is. Just as it is easier for you to remember that the president lives in the White House than at 1600 Pennsylvania Avenue, Washington, D.C., it is also easier for you to remember that your salary is in a variable called `mySalary` than at memory location 6428104.

Similarly, it does not usually make sense to perform mathematical operations on names given to memory addresses, but it does make sense to perform mathematical operations on the *contents* of memory addresses. If you live in

`blueSplitLevelOnTheCorner`, adding 1 to that would be meaningless, but you certainly can add 1 person to the number of people already in that house. For our purposes, then, the statement `calculatedAnswer = inputNumber * 2` means exactly the same thing as the statement `calculate inputNumber * 2` (that is, double the contents in the memory location named `inputNumber`) and store the result in the memory location named `calculatedAnswer`.

## TIP



Many programming languages allow you to create named constants. A named constant is a named memory location, similar to a variable, except its value never changes during the execution of a program. If you are working with a programming language that allows it, you might create a constant for a value such as `PI = 3.14` or `COUNTY_SALES_TAX_RATE = .06`. Many programmers follow the convention of using camel casing for variable identifiers but all capital letters for constant identifiers.

## UNDERSTANDING DATA TYPES

Computers deal with two basic types of data—text and numeric. When you use a specific numeric value, such as 43, within a program, you write it using the digits and no quotation marks. A specific numeric value is often called a **numeric constant**, because it does not change—a 43 always has the value 43. When you use a specific text value, or string of characters, such as “Amanda”, you enclose the **string constant**, or **character constant**, within quotation marks.

## TIP



Some languages require single quotation marks surrounding character constants, whereas others require double quotation marks. Many languages, including C++, C#, and Java, reserve single quotes for a single character such as ‘A’, and double quotes for a character string such as “Amanda”.

Similarly, most computer languages allow at least two distinct types of variables. A variable’s **data type** describes the kind of values the variable can hold and the types of operations that can be performed with it. One type of variable can hold a number, and is often called a **numeric variable**. A numeric variable is one that can have mathematical operations performed on it; it can hold digits, and usually can hold a decimal point and a sign indicating positive or negative if you want. In the statement `calculatedAnswer = inputNumber * 2`, both `calculatedAnswer` and `inputNumber` are numeric variables; that is, their intended contents are numeric values, such as 6 and 3, 150 and 75, or –18 and –9.

Most programming languages have a separate type of variable that can hold letters of the alphabet and other special characters such as punctuation marks. Depending on the language, these variables are called **character**, **text**, or **string variables**. If a working program contains the statement `lastName = "Lincoln"`, then `lastName` is a character or string variable.

Programmers must distinguish between numeric and character variables, because computers handle the two types of data differently. Therefore, means are provided within the syntax rules of computer programming languages to tell the



computer which type of data to expect. How this is done is different in every language; some languages have different rules for naming the variables, but with others you must include a simple statement (called a **declaration**) telling the computer which type of data to expect.

Some languages allow for several types of numeric data. Languages such as C++, C#, Visual Basic, and Java distinguish between **integer** (whole number) numeric variables and **floating-point** (fractional) numeric variables that contain a decimal point. Thus, in some languages, the values 4 and 4.3 would be stored in different types of numeric variables.

Some programming languages allow even more specific variable types, but the character versus numeric distinction is universal. For the programs you develop in this book, assume that each variable is one of the two broad types. If a variable called `taxRate` is supposed to hold a value of 2.5, assume that it is a numeric variable. If a variable called `inventoryItem` is supposed to hold a value of “monitor”, assume that it is a character variable.

**TIP** □ □ □ □ Values such as “monitor” and 2.5 are called constants or literal constants because they never change. A variable value *can* change. Thus, `inventoryItem` can hold “monitor” at one moment during the execution of a program, and later you can change its value to “modem”.

**TIP** □ □ □ □ Some languages allow you to invent your own data type. In Chapter 12 of the Comprehensive version of this book, you will learn that object-oriented programming languages allow you to create new data types called classes.

By convention, this book encloses character data like “monitor” within quotation marks to distinguish the characters from yet another variable name. Also by convention, numeric data values are not enclosed within quotation marks. According to these conventions, then, `taxRate = 2.5` and `inventoryItem = "monitor"` are both valid statements. The statement `inventoryItem = monitor` is a valid statement only if `monitor` is also a character variable. In other words, if `monitor = "color"`, and subsequently `inventoryItem = monitor`, then the end result is that the memory address named `inventoryItem` contains the string of characters “color”.

Every computer handles text or character data differently from the way it handles numeric data. You may have experienced these differences if you have used application software such as spreadsheets or database programs. For example, in a spreadsheet, you cannot sum a column of words. Similarly, every programming language requires that you distinguish variables as to their correct type, and that you use each type of variable appropriately. Identifying your variables correctly as numeric or character is one of the first steps you have to take when writing programs in any programming language. Table 1-2 provides you with a few examples of legal and illegal variable assignment statements.

**TIP** □ □ □ □ The process of naming program variables and assigning a type to them is called **making declarations**, or **declaring variables**. You will learn how to declare variables in Chapter 4.

**TABLE 1-2:** SOME EXAMPLES OF LEGAL AND ILLEGAL ASSIGNMENTS

*Assume lastName and firstName are character variables.*

*Assume quizScore and homeworkScore are numeric variables.*

Examples of valid assignments	Examples of invalid assignments	Explanation of invalid examples
<code>lastName = "Parker"</code>	<code>lastName = Parker</code>	If <code>Parker</code> is the last name, it requires quotes. If <code>Parker</code> is a named string variable, this assignment would be allowed.
<code>firstName = "Laura"</code>	<code>"Parker" = lastName</code>	Value on left must be a variable name, not a constant
<code>lastName = firstName</code>	<code>lastName = quizScore</code>	The data types do not match
<code>quizScore = 86</code>	<code>homeworkScore = firstName</code>	The data types do not match
<code>homeworkScore = quizScore</code>	<code>homeworkScore = "92"</code>	The data types do not match
<code>homeworkScore = 92</code>	<code>quizScore = "zero"</code>	The data types do not match
<code>quizScore = homeworkScore + 25</code>	<code>firstName = 23</code>	The data types do not match
<code>homeworkScore = 3 * 10</code>	<code>100 = homeworkScore</code>	Value on left must be a variable name, not a constant

## UNDERSTANDING THE EVOLUTION OF PROGRAMMING TECHNIQUES

People have been writing computer programs since the 1940s. The oldest programming languages required programmers to work with memory addresses and to memorize awkward codes associated with machine languages. Newer programming languages look much more like natural language and are easier for programmers to use. Part of the reason it is easier to use newer programming languages is that they allow programmers to name variables instead of using awkward memory addresses. Another reason is that newer programming languages provide programmers with the means to create self-contained modules or program segments that can be pieced together in a variety of ways. The oldest computer programs were written in one piece, from start to finish; modern programs are rarely written that way—they are created by teams of programmers, each developing his or her own reusable and connectable program procedures. Writing several small modules is easier than writing one large program, and most large tasks are easier when you break the work into units and get other workers to help with some of the units.

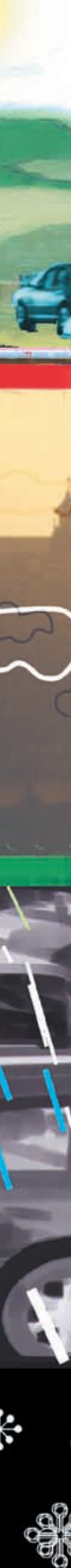


You will learn to create program modules in Chapter 3.

Currently, there are two major techniques used to develop programs and their procedures. One technique, called **procedural programming**, focuses on the procedures that programmers create. That is, procedural programmers focus on the actions that are carried out—for example, getting input data for an employee and writing the calculations needed to produce a paycheck from the data. Procedural programmers would approach the job of producing a paycheck by breaking down the paycheck-producing process into manageable subtasks.

The other popular programming technique, called **object-oriented programming**, focuses on objects, or “things,” and describes their features, or attributes, and their behaviors. For example, object-oriented programmers might design a payroll application by thinking about employees and paychecks, and describing their attributes (such as last name or check amount) and behaviors (such as the calculations that result in the check amount).

With either approach, procedural or object-oriented, you can produce a correct paycheck, and both techniques employ reusable program modules. The major difference lies in the focus the programmer takes during the earliest planning stages of a project. Object-oriented programming employs a large vocabulary; you can learn this terminology in Chapter 13 of the Comprehensive version of this book. For now, this book focuses on procedural programming techniques. The skills you gain in programming procedurally—declaring variables, accepting input, making decisions, producing output, and so on—will serve you well whether you eventually write programs in a procedural or object-oriented fashion, or in both.



## CHAPTER SUMMARY

- Together, computer hardware (equipment) and software (instructions) accomplish four major operations: input, processing, output, and storage. You write computer instructions in a computer programming language that requires specific syntax; the instructions are translated into machine language by a compiler or interpreter. When both the syntax and logic of a program are correct, you can run, or execute, the program to produce the desired results.
- A programmer's job involves understanding the problem, planning the logic, coding the program, translating the program into machine language, testing the program, and putting the program into production.
- When data items are stored for use on computer systems, they are stored in a data hierarchy of character, field, record, file, and database.
- When programmers plan the logic for a solution to a programming problem, they often use flowcharts or pseudocode. When you draw a flowchart, you use parallelograms to represent input and output operations, and rectangles to represent processing.
- Variables are named memory locations, the contents of which can vary. As a programmer, you choose reasonable names for your variables. Every computer programming language has its own set of rules for naming variables; however, all variable names must be written as one word without embedded spaces, and should have appropriate meaning.
- Testing a value involves making a decision. You represent a decision in a flowchart by drawing a diamond-shaped decision symbol containing a question, the answer to which is either yes or no. You can stop a program's execution by using a decision to test for a sentinel value.
- A connector symbol is used to continue a flowchart that does not fit together on a page, or must continue on an additional page.
- Most programming languages use the equal sign to assign values to variables. Assignment always takes place from right to left.
- Programmers must distinguish between numeric and character variables, because computers handle the two types of data differently. A variable declaration tells the computer which type of data to expect. By convention, character data values are included within quotation marks.
- Procedural and object-oriented programmers approach program problems differently. Procedural programmers concentrate on the actions performed with data. Object-oriented programmers focus on objects and their behaviors and attributes.

## KEY TERMS

**Hardware** is the equipment of a computer system.

**Software** consists of the programs that tell the computer what to do.

**Input** devices include keyboards and mice; through these devices, data items enter the computer system. Data can also enter a system from storage devices such as magnetic disks and CDs.

**Data** includes all the text, numbers, and other information that are processed by a computer.

**Processing** data items may involve organizing them, checking them for accuracy, or performing mathematical operations on them.

The **central processing unit**, or **CPU**, is the piece of hardware that processes data.

Information is sent to a printer, monitor, or some other **output** device so people can view, interpret, and work with the results.

**Programming languages**, such as Visual Basic, C#, C++, Java, or COBOL, are used to write programs.

The **syntax** of a language consists of its rules.

**Machine language** is a computer's on/off circuitry language.

A **compiler** or **interpreter** translates a high-level language into machine language and tells you if you have used a programming language incorrectly.

You develop the **logic** of the computer program when you give instructions to the computer in a specific sequence, without leaving any instructions out or adding extraneous instructions.

A **semantic error** occurs when a correct word is used in an incorrect context.

The **running**, or **executing**, of a program occurs when the computer actually uses the written and compiled program.

**Internal storage** is called **memory**, **main memory**, **primary memory**, or **random access memory (RAM)**.

**External storage** is **persistent** (relatively permanent) storage outside the main memory of the machine, on a device such as a floppy disk, hard disk, or magnetic tape.

Internal memory is **volatile**—that is, its contents are lost every time the computer loses power.

You **save** a program on some nonvolatile medium.

An **algorithm** is the sequence of steps necessary to solve any problem.

**Desk-checking** is the process of walking through a program solution on paper.

**Coding** a program means writing the statements in a programming language.

**High-level programming languages** are English-like.

**Machine language** is the **low-level** language made up of 1s and 0s that the computer understands.

A **syntax error** is an error in language or grammar.

**Logical errors** occur when incorrect instructions are performed, or when instructions are performed in the wrong order.

**Conversion** is the entire set of actions an organization must take to switch over to using a new program or set of programs.

The **data hierarchy** represents the relationship of databases, files, records, fields, and characters.

**Characters** are letters, numbers, and special symbols such as "A", "7", and "\$".

A **field** is a single data item, such as `lastName`, `streetAddress`, or `annualSalary`.

**Records** are groups of fields that go together for some logical reason.

**Files** are groups of records that go together for some logical reason.

A **database** holds a group of files, often called **tables**, that together serve the information needs of an organization.

**Queries** are questions that pull related data items together from a database in a format that enhances efficient management decision making.

A **flowchart** is a pictorial representation of the logical steps it takes to solve a problem.

**Pseudocode** is an English-like representation of the logical steps it takes to solve a problem.

**Input symbols**, which indicate input operations, are represented as parallelograms in flowcharts.

**Processing symbols** are represented as rectangles in flowcharts.

**Output symbols**, which indicate output operations, are represented as parallelograms in flowcharts.

**Flowlines**, or arrows, connect the steps in a flowchart.

A **terminal symbol**, or start/stop symbol, is used at each end of a flowchart. Its shape is a **lozenge**.

**Variables** are memory locations, whose contents can vary or differ over time.

A variable name is also called an **identifier**.

A **mnemonic** is a memory device; variable identifiers act as mnemonics for hard-to-remember memory addresses.

**Camel casing** is the format for naming variables in which multiple-word variable names are run together, and each new word within the variable name begins with an uppercase letter.

An **infinite loop** is a repeating flow of logic without an ending.

Testing a value is also called making a **decision**.

You represent a decision in a flowchart by drawing a **decision symbol**, which is shaped like a diamond.

A yes-or-no decision is called a **binary decision**, because there are two possible outcomes.

A **dummy value** is a preselected value that stops the execution of a program. Such a value is sometimes called a **sentinel value** because it represents an entry or exit point, like a sentinel who guards a fortress.

Many programming languages use the term **eof** (for “end of file”) to talk about an end-of-data file marker.

A **connector** is a flowchart symbol used when limited page size forces you to continue the flowchart elsewhere on the same page or on the following page.

An **assignment statement** stores the result of any calculation performed on its right side to the named location on its left side.

The equal sign is the **assignment operator**; it always requires the name of a memory location on its left side.

A **numeric constant** is a specific numeric value.

A **string constant**, or **character constant**, is enclosed within quotation marks.

A variable's **data type** describes the kind of values the variable can hold and the types of operations that can be performed with it.

**Numeric variables** hold numeric values.

**Character, text, or string variables** hold character values. If a working program contains the statement `lastName = "Lincoln"`, then `lastName` is a character or string variable.

A **declaration** is a statement that names a variable and tells the computer which type of data to expect.

**Integer** values are whole-number, numeric variables.

**Floating-point** values are fractional, numeric variables that contain a decimal point.

The process of naming program variables and assigning a type to them is called **making declarations**, or **declaring variables**.

The technique known as **procedural programming** focuses on the procedures that programmers create.

The technique known as **object-oriented programming** focuses on objects, or “things,” and describes their features, or attributes, and their behaviors.

## REVIEW QUESTIONS

1. **The two major components of any computer system are its \_\_\_\_\_.**
  - a. input and output
  - b. data and programs
  - c. hardware and software
  - d. memory and disk drives
2. **The major computer operations include \_\_\_\_\_.**
  - a. hardware and software
  - b. input, processing, output, and storage
  - c. sequence and looping
  - d. spreadsheets, word processing, and data communications
3. **Another term meaning “computer instructions” is \_\_\_\_\_.**
  - a. hardware
  - b. software
  - c. queries
  - d. data
4. **Visual Basic, C++, and Java are all examples of computer \_\_\_\_\_.**
  - a. operating systems
  - b. hardware
  - c. machine languages
  - d. programming languages

5. **A programming language's rules are its \_\_\_\_\_.**
  - a. syntax
  - b. logic
  - c. format
  - d. options
  
6. **The most important task of a compiler or interpreter is to \_\_\_\_\_.**
  - a. create the rules for a programming language
  - b. translate English statements into a language such as Java
  - c. translate programming language statements into machine language
  - d. execute machine language programs to perform useful tasks
  
7. **Which of the following is a typical input instruction?**
  - a. `get accountNumber`
  - b. `calculate balanceDue`
  - c. `print customerIdentificationNumber`
  - d. `total = janPurchase + febPurchase`
  
8. **Which of the following is a typical processing instruction?**
  - a. `print answer`
  - b. `get userName`
  - c. `pctCorrect = rightAnswers / allAnswers`
  - d. `print calculatedPercentage`
  
9. **Which of the following is not associated with internal storage?**
  - a. main memory
  - b. hard disk
  - c. primary memory
  - d. volatile
  
10. **Which of the following pairs of steps in the programming process is in the correct order?**
  - a. code the program, plan the logic
  - b. test the program, translate it into machine language
  - c. put the program into production, understand the problem
  - d. code the program, translate it into machine language
  
11. **The two most commonly used tools for planning a program's logic are \_\_\_\_\_.**
  - a. flowcharts and pseudocode
  - b. ASCII and EBCDIC
  - c. Java and Visual Basic
  - d. word processors and spreadsheets



12. **The most important thing a programmer must do before planning the logic to a program is \_\_\_\_\_.**
- decide which programming language to use
  - code the problem
  - train the users of the program
  - understand the problem
13. **Writing a program in a language such as C++ or Java is known as \_\_\_\_\_ the program.**
- translating
  - coding
  - interpreting
  - compiling
14. **A compiler would find all of the following programming errors except \_\_\_\_\_.**
- the misspelled word "pprint" in a language that includes the word "print"
  - the use of an "X" for multiplication in a language that requires an asterisk
  - a `newBalanceDue` calculated by adding a `customerPayment` to an `oldBalanceDue` instead of subtracting it
  - an arithmetic statement written as `regularSales + discountedSales = totalSales`
15. **Which of the following is true regarding the data hierarchy?**
- files contain records
  - characters contain fields
  - fields contain files
  - fields contain records
16. **The parallelogram is the flowchart symbol representing \_\_\_\_\_.**
- input
  - output
  - both a and b
  - none of the above
17. **Which of the following is not a legal variable name in any programming language?**
- `semester grade`
  - `fall2005_grade`
  - `GradeInCIS100`
  - `MY_GRADE`
18. **In flowcharts, the decision symbol is a \_\_\_\_\_.**
- parallelogram
  - rectangle
  - lozenge
  - diamond

19. The term “eof” represents \_\_\_\_\_.
- a standard input device
  - a generic sentinel value
  - a condition in which no more memory is available for storage
  - the logical flow in a program
20. The two broadest types of data are \_\_\_\_\_.
- internal and external
  - volatile and constant
  - character and numeric
  - permanent and temporary

### FIND THE BUGS

Since the early days of computer programming, program errors have been called “bugs.” The term is often said to have originated from an actual moth that was discovered trapped in the circuitry of a computer at Harvard University in 1945. Actually, the term “bug” was in use prior to 1945 to mean trouble with any electrical apparatus; even during Thomas Edison’s life, it meant an “industrial defect.” However, the process of finding and correcting program errors has come to be known as debugging.

Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. **This pseudocode segment is intended to describe computing your average score of two classroom tests.**

```
input midtermGrade
input finalGrade
average = (inputGrade + final) / 3
print average
```

2. **This pseudocode segment is intended to describe computing the number of miles per gallon you get with your automobile.**

```
input milesTraveled
input gallonsOfGasUsed
gallonsOfGasUsed / milesTravelled = milesPerGallon
print milesPerGal
```

3. **This pseudocode segment is intended to describe computing the cost per day and the cost per week for a vacation.**

```
input totalDollarsSpent
input daysOnTrip
costPerDay = totalMoneySpent * daysOnTrip
weeks = daysOnTrip / 7
costPerWeek = daysOnTrip / numberOfWeeks
print costPerDay, week
```

**EXERCISES****1. Match the definition with the appropriate term.**

- |                              |             |
|------------------------------|-------------|
| 1. Computer system equipment | a. compiler |
| 2. Another word for programs | b. syntax   |
| 3. Language rules            | c. logic    |
| 4. Order of instructions     | d. hardware |
| 5. Language translator       | e. software |

**2. In your own words, describe the steps to writing a computer program.****3. Consider a student file that contains the following data:**






LAST NAME	FIRST NAME	MAJOR	GRADE POINT AVERAGE
Andrews	David	Psychology	3.4
Broederdorf	Melissa	Computer Science	4.0
Brogan	Lindsey	Biology	3.8
Carson	Joshua	Computer Science	2.8
Eisfelder	Katie	Mathematics	3.5
Faris	Natalie	Biology	2.8
Fredricks	Zachary	Psychology	2.0
Gonzales	Eduardo	Biology	3.1

**Would this set of data be suitable and sufficient to use to test each of the following programs?**

**Explain why or why not.**

- a. a program that prints a list of Psychology majors
  - b. a program that prints a list of Art majors
  - c. a program that prints a list of students on academic probation—those with a grade point average under 2.0
  - d. a program that prints a list of students on the dean's list
  - e. a program that prints a list of students from Wisconsin
  - f. a program that prints a list of female students
- 4. Suggest a good set of test data to use for a program that gives an employee a \$50 bonus check if the employee has produced more than 1,000 items in a week.**
  - 5. Suggest a good set of test data for a program that computes gross paychecks (that is, before any taxes or other deductions) based on hours worked and rate of pay. The program computes gross as hours times rate, unless hours are over 40. If so, the program computes gross as regular rate of pay for 40 hours, plus one and a half times the rate of pay for the hours over 40.**
  - 6. Suggest a good set of test data for a program that is intended to output a student's grade point average based on letter grades (A, B, C, D, or F) in five courses.**
  - 7. Suggest a good set of test data for a program for an automobile insurance company that wants to increase its premiums by \$50 per month for every ticket a driver receives in a three-year period.**

8. Assume that a grocery store keeps a file for inventory, where each grocery item has its own record. Two fields within each record are the name of the manufacturer and the weight of the item. Name at least six more fields that might be stored for each record. Provide an example of the data for one record. For example, for one product the manufacturer is DeMonte, and the weight is 12 ounces.
9. Assume that a library keeps a file with data about its collection, one record for each item the library lends out. Name at least eight fields that might be stored for each record. Provide an example of the data for one record.
10. Match the term with the appropriate shape.

1. Input	A.	
2. Processing	B.	
3. Decision	C.	
4. Terminal	D.	
5. Connector	E.	

11. Which of the following names seem like good variable names to you? If a name doesn't seem like a good variable name, explain why not.
  - a. c
  - b. cost
  - c. costAmount
  - d. cost amount

- e. `cstofdnghbsns`
- f. `costOfDoingBusinessThisFiscalYear`
- g. `cost2004`

**12. If `myAge` and `yourRate` are numeric variables, and `departmentCode` is a character variable, which of the following statements are valid assignments? If a statement is not valid, explain why not.**

- a. `myAge = 23`
- b. `myAge = yourRate`
- c. `myAge = departmentCode`
- d. `myAge = "departmentCode"`
- e. `42 = myAge`
- f. `yourRate = 3.5`
- g. `yourRate = myAge`
- h. `yourRate = departmentCode`
- i. `6.91 = yourRate`
- j. `departmentCode = Personnel`
- k. `departmentCode = "Personnel"`
- l. `departmentCode = 413`
- m. `departmentCode = "413"`
- n. `departmentCode = myAge`
- o. `departmentCode = yourRate`
- p. `413 = departmentCode`
- q. `"413" = departmentCode`

**13. Complete the following tasks:**

- a. Draw a flowchart to represent the logic of a program that allows the user to enter a value. The program multiplies the value by 10 and prints the result.
- b. Write pseudocode for the same problem.

**14. Complete the following tasks:**

- a. Draw a flowchart to represent the logic of a program that allows the user to enter a value that represents the radius of a circle. The program calculates the diameter (by multiplying the radius by 2), and then calculates the circumference (by multiplying the diameter by 3.14). The program prints both the diameter and the circumference.
- b. Write pseudocode for the same problem.

**15. Complete the following tasks:**

- a. Draw a flowchart to represent the logic of a program that allows the user to enter two values. The program prints the sum of the two values.
- b. Write pseudocode for the same problem.

**16. Complete the following tasks:**

- a. Draw a flowchart to represent the logic of a program that allows the user to enter three values. The first value represents hourly pay rate, the second represents the number of hours worked this pay period, and the third represents the percentage of gross salary that is withheld. The program multiplies the hourly pay rate by the number of hours worked, giving the gross pay; then, it multiplies the gross pay by the withholding percentage, giving the withholding amount. Finally, it subtracts the withholding amount from the gross pay, giving the net pay after taxes. The program prints the net pay.
- b. Write pseudocode for the same problem.

**DETECTIVE WORK**

1. **Even Shakespeare referred to a “bug” as a negative occurrence. Name the work in which he wrote, “Warwick was a bug that fear’d us all.”**
2. **What are the distinguishing features of the programming language called Short Code? When was it invented?**
3. **What is the difference between a compiler and an interpreter? Under what conditions would you prefer to use one over the other?**

**UP FOR DISCUSSION**

1. **Which is the better tool for learning programming—flowcharts or pseudocode? Cite any educational research you can find.**
2. **What is the image of the computer programmer in popular culture? Is the image different in books than in TV shows and movies? Would you like that image for yourself?**