

Introduction to Computers and Programming

TOPICS

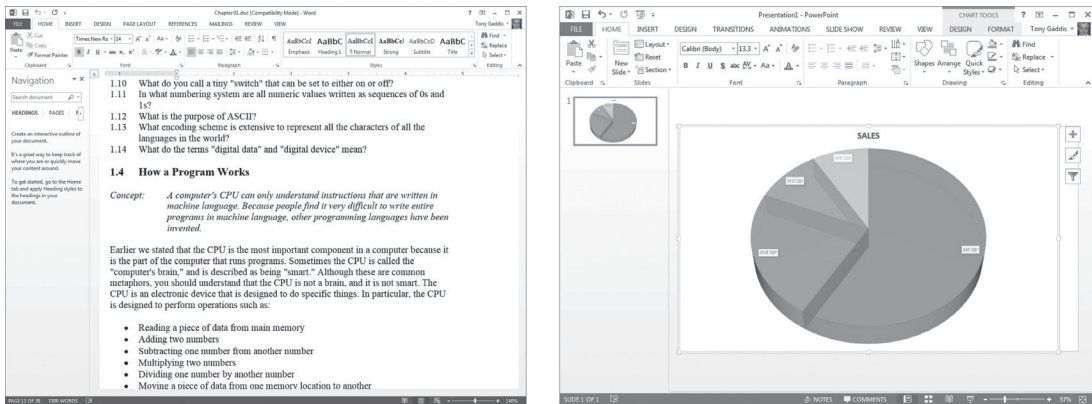
- | | |
|------------------------------|-------------------------|
| 1.1 Introduction | 1.4 How a Program Works |
| 1.2 Hardware | 1.5 Types of Software |
| 1.3 How Computers Store Data | |

1.1

Introduction

Think about some of the different ways that people use computers. In school, students use computers for tasks such as writing papers, searching for articles, sending email, and participating in online classes. At work, people use computers to analyze data, make presentations, conduct business transactions, communicate with customers and coworkers, control machines in manufacturing facilities, and many other things. At home, people use computers for tasks such as paying bills, shopping online, communicating with friends and family, and playing computer games. And don't forget that smart phones, tablets, MP3 players, car navigation systems, and many other devices are computers too. The uses of computers are almost limitless in our everyday lives.

Computers can do such a wide variety of things because they can be programmed. This means that computers are not designed to do just one job, but to do any job that their programs tell them to do. A *program* is a set of instructions that a computer follows to perform a task. For example, Figure 1-1 shows screens from two commonly used programs: Microsoft Word and PowerPoint.

Figure 1-1 Commonly used programs (Courtesy of Microsoft Corporation)

Programs are commonly referred to as *software*. Software is essential to a computer because without software, a computer can do nothing. All of the software that we use to make our computers useful is created by individuals known as programmers or software developers. A *programmer*, or *software developer*, is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, you will find programmers working in business, medicine, government, law enforcement, agriculture, academics, entertainment, and almost every other field.

This book introduces you to the fundamental concepts of computer programming. Before we begin exploring those concepts, you need to understand a few basic things about computers and how they work. This chapter will build a solid foundation of knowledge that you will continually rely on as you study computer science. First, we will discuss the physical components that computers are commonly made of. Next, we will look at how computers store data and execute programs. Finally, we will discuss the major types of software that computers use.

1.2

Hardware

CONCEPT: The physical devices that a computer is made of are referred to as the computer's hardware. Most computer systems are made of similar hardware devices.

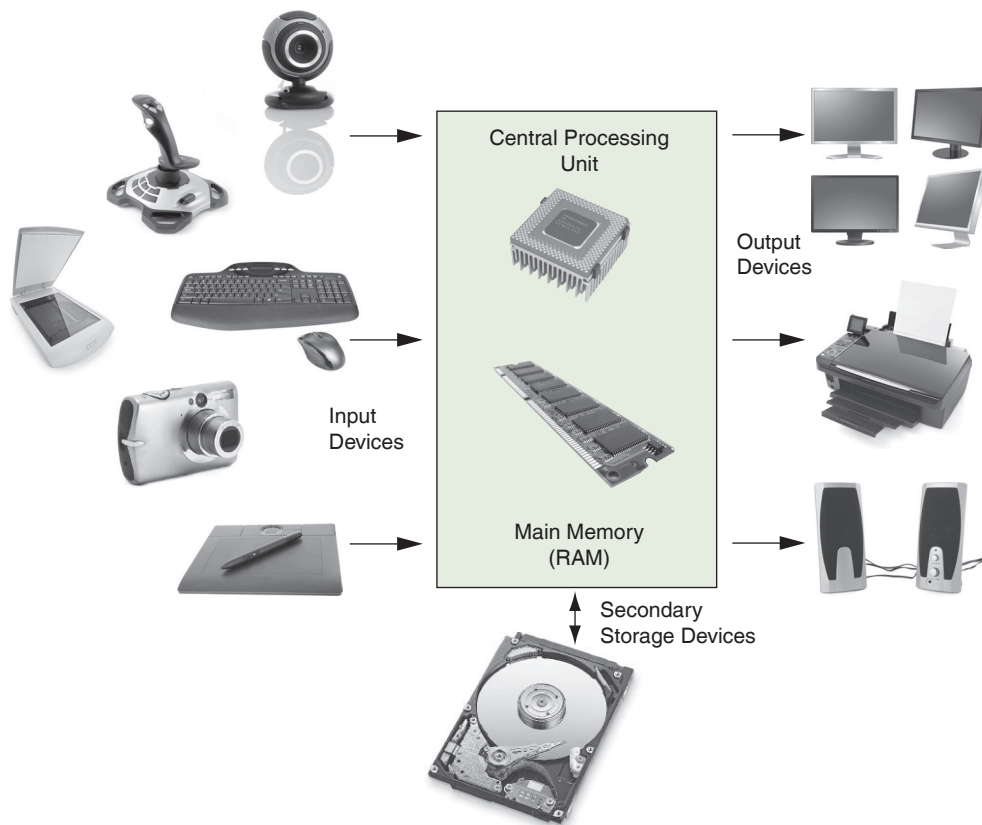
The term *hardware* refers to all of the physical devices, or *components*, that a computer is made of. A computer is not one single device, but a system of devices that all work together. Like the different instruments in a symphony orchestra, each device in a computer plays its own part.

If you have ever shopped for a computer, you've probably seen sales literature listing components such as microprocessors, memory, disk drives, video displays, graphics cards, and so on. Unless you already know a lot about computers, or at least have a

friend who does, understanding what these different components do can be confusing. As shown in Figure 1-2, a typical computer system consists of the following major components:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices
- Input devices
- Output devices

Figure 1-2 Typical components of a computer system (all photos © Shutterstock)



Let's take a closer look at each of these components.

The CPU

When a computer is performing the tasks that a program tells it to do, we say that the computer is *running* or *executing* the program. The *central processing unit*, or *CPU*, is the part of a computer that actually runs programs. (The CPU is often referred to as the *processor*.) The CPU is the most important component in a computer because without it, the computer could not run software.

In the earliest computers, CPUs were huge devices made of electrical and mechanical components such as vacuum tubes and switches. Figure 1-3 shows such a device. The two women in the photo are working with the historic ENIAC computer. The *ENIAC*, considered by many to be the world's first programmable electronic computer, was built in 1945 to calculate artillery ballistic tables for the U.S. Army. This machine, which was primarily one big CPU, was 8 feet tall, 100 feet long, and weighed 30 tons.

Today, CPUs are small chips known as *microprocessors*. Figure 1-4 shows a photo of a lab technician holding a modern-day microprocessor. In addition to being much smaller than the old electro-mechanical CPUs in early computers, microprocessors are also much more powerful.

Figure 1-3 The ENIAC computer (Courtesy of US Army Center of Military History)

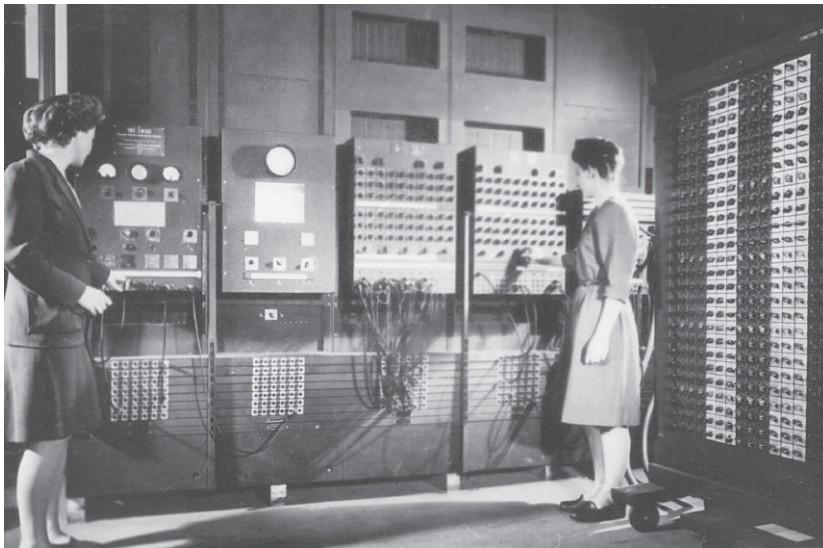


Figure 1-4 A lab technician holds a modern microprocessor (Courtesy of Chris Ryan/OJO Images/Getty Images)

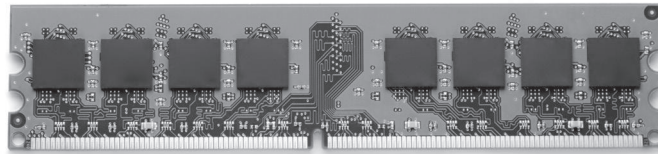


Main Memory

You can think of *main memory* as the computer's work area. This is where the computer stores a program while the program is running, as well as the data that the program is working with. For example, suppose you are using a word processing program to write an essay for one of your classes. While you do this, both the word processing program and the essay are stored in main memory.

Main memory is commonly known as *random-access memory*, or *RAM*. It is called this because the CPU is able to quickly access data stored at any random location in RAM. RAM is usually a *volatile* type of memory that is used only for temporary storage while a program is running. When the computer is turned off, the contents of RAM are erased. Inside your computer, RAM is stored in chips, similar to the ones shown in Figure 1-5.

Figure 1-5 Memory chips (photo © Garsya/Shutterstock)



NOTE: Another type of memory that is stored in chips inside the computer is *read-only memory*, or *ROM*. A computer can read the contents of ROM, but it cannot change its contents, or store additional data there. ROM is *nonvolatile*, which means that it does not lose its contents, even when the computer's power is turned off. ROM is typically used to store programs that are important for the system's operation. One example is the computer's startup program, which is executed each time the computer is started.

Secondary Storage Devices

Secondary storage is a type of memory that can hold data for long periods of time, even when there is no power to the computer. Programs are normally stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory records, is saved to secondary storage as well.

The most common type of secondary storage device is the disk drive. A traditional *disk drive* stores data by magnetically encoding it onto a circular disk. *Solid state drives*, which store data in solid-state memory, are increasingly becoming popular. A solid state drive has no moving parts, and operates faster than a traditional disk drive. Most computers have some sort of secondary storage device, either a traditional disk drive or a solid state drive, mounted inside their case. External disk drives, which connect to one of the computer's communication ports, are also available. External disk drives can be used to create backup copies of important data or to move data to another computer.

In addition to external disk drives, many types of devices have been created for copying data, and for moving it to other computers. *Universal Serial Bus drives*, or *USB drives*, are small devices that plug into the computer's USB port, and appear to the system as a disk

drive. These drives do not actually contain a disk, however. They store data in a special type of memory known as *flash memory*. USB drives, which are also known as *memory sticks* and *flash drives*, are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the *CD* (compact disc) and the *DVD* (digital versatile disc) are also used for data storage. Data is not recorded magnetically on an optical disc, but is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they are good mediums for creating backup copies of data.



NOTE: In recent years, *cloud storage* has become a popular way to store data. When you store data in the cloud, you are storing it on a remote server via the Internet, or via a company's private network. When your data is stored in the cloud, you can access it from many different devices, and from any location where you have a network connection. Cloud storage can also be used to back up important data that is stored on a computer's disk.

Input Devices

Input is any data the computer collects from people and from other devices. The component that collects the data and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, touchscreen, scanner, microphone, and digital camera. Disk drives and optical drives can also be considered input devices because programs and data are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any data the computer produces for people or for other devices. It might be a sales report, a list of names, or a graphic image. The data is sent to an *output device*, which formats and presents it. Common output devices are video displays and printers. Disk drives and CD recorders can also be considered output devices because the system sends data to them in order to be saved.



Checkpoint

- 1.1 What is a program?
- 1.2 What is hardware?
- 1.3 List the five major components of a computer system.
- 1.4 What part of the computer actually runs programs?
- 1.5 What part of the computer serves as a work area to store a program and its data while the program is running?
- 1.6 What part of the computer holds data for long periods of time, even when there is no power to the computer?
- 1.7 What part of the computer collects data from people and from other devices?
- 1.8 What part of the computer formats and presents data for people or other devices?

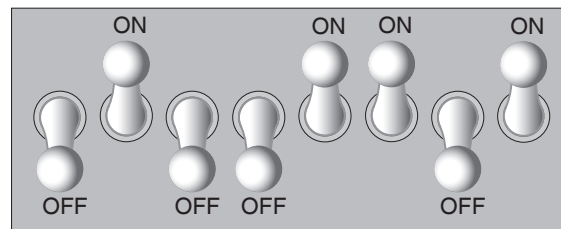
1.3 How Computers Store Data

CONCEPT: All data that is stored in a computer is converted to sequences of 0s and 1s.

A computer's memory is divided into tiny storage locations known as *bytes*. One byte is only enough memory to store a letter of the alphabet or a small number. In order to do anything meaningful, a computer has to have lots of bytes. Most computers today have millions, or even billions, of bytes of memory.

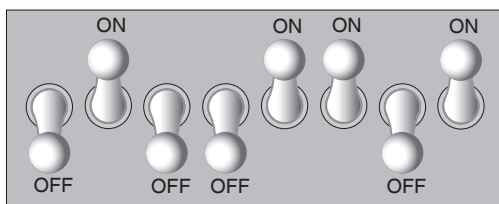
Each byte is divided into eight smaller storage locations known as bits. The term *bit* stands for *binary digit*. Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer systems, bits are tiny electrical components that can hold either a positive or a negative charge. Computer scientists think of a positive charge as a switch in the *on* position, and a negative charge as a switch in the *off* position. Figure 1-6 shows the way that a computer scientist might think of a byte of memory: as a collection of switches that are each flipped to either the on or off position.

Figure 1-6 Think of a byte as eight switches

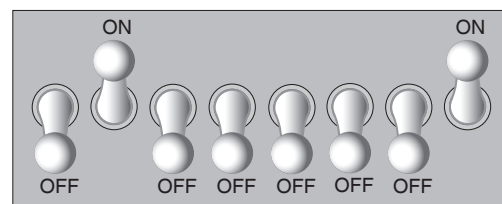


When a piece of data is stored in a byte, the computer sets the eight bits to an on/off pattern that represents the data. For example, the pattern shown on the left in Figure 1-7 shows how the number 77 would be stored in a byte, and the pattern on the right shows how the letter A would be stored in a byte. In a moment you will see how these patterns are determined.

Figure 1-7 Bit patterns for the number 77 and the letter A



The number 77 stored in a byte.



The letter A stored in a byte.

Storing Numbers

A bit can be used in a very limited way to represent numbers. Depending on whether the bit is turned on or off, it can represent one of two different values. In computer systems, a bit that is turned off represents the number 0 and a bit that is turned on represents the number 1. This corresponds perfectly to the *binary numbering system*. In the binary numbering system (or *binary*, as it is usually called) all numeric values are written as sequences of 0s and 1s. Here is an example of a number that is written in binary:

10011101

The position of each digit in a binary number has a value assigned to it. Starting with the rightmost digit and moving left, the position values are 2^0 , 2^1 , 2^2 , 2^3 , and so forth, as shown in Figure 1-8. Figure 1-9 shows the same diagram with the position values calculated. Starting with the rightmost digit and moving left, the position values are 1, 2, 4, 8, and so forth.

Figure 1-8 The values of binary digits as powers of 2

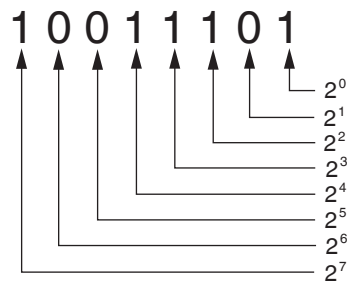
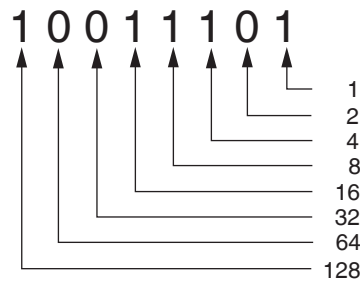


Figure 1-9 The values of binary digits



To determine the value of a binary number you simply add up the position values of all the 1s. For example, in the binary number 10011101, the position values of the 1s are 1, 4, 8, 16, and 128. This is shown in Figure 1-10. The sum of all of these position values is 157. So, the value of the binary number 10011101 is 157.

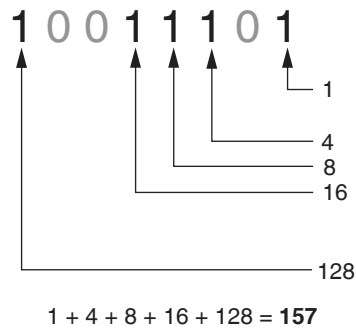
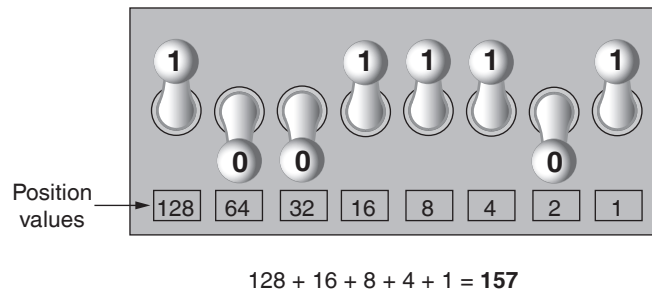
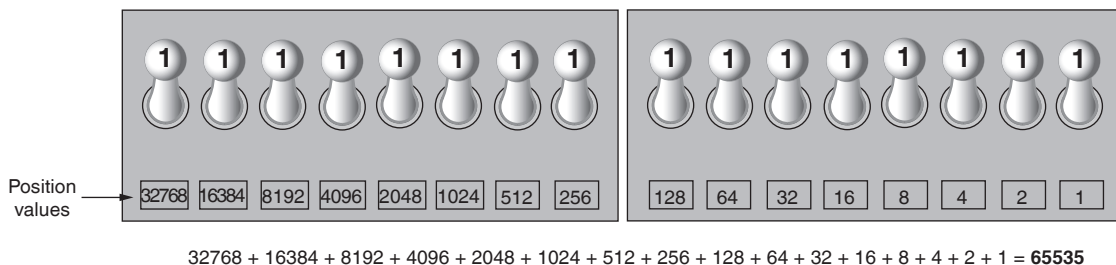
Figure 1-10 Determining the value of 1001101

Figure 1-11 shows how you can picture the number 157 stored in a byte of memory. Each 1 is represented by a bit in the on position, and each 0 is represented by a bit in the off position.

Figure 1-11 The bit pattern for 157

When all of the bits in a byte are set to 0 (turned off), then the value of the byte is 0. When all of the bits in a byte are set to 1 (turned on), then the byte holds the largest value that can be stored in it. The largest value that can be stored in a byte is $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$. This limit exists because there are only eight bits in a byte.

What if you need to store a number larger than 255? The answer is simple: use more than one byte. For example, suppose we put two bytes together. That gives us 16 bits. The position values of those 16 bits would be $2^0, 2^1, 2^2, 2^3$, and so forth, up through 2^{15} . As shown in Figure 1-12, the maximum value that can be stored in two bytes is 65,535. If you need to store a number larger than this, then more bytes are necessary.

Figure 1-12 Two bytes used for a large number



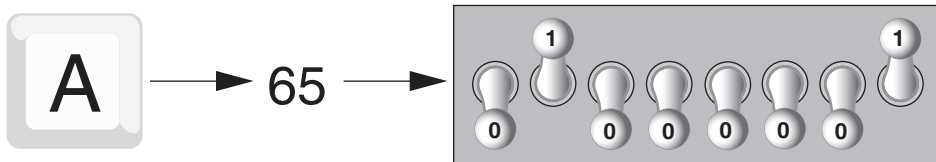
TIP: In case you're feeling overwhelmed by all this, relax! You will not have to actually convert numbers to binary while programming. Knowing that this process is taking place inside the computer will help you as you learn, and in the long term this knowledge will make you a better programmer.

Storing Characters

Any piece of data that is stored in a computer's memory must be stored as a binary number. That includes characters, such as letters and punctuation marks. When a character is stored in memory, it is first converted to a numeric code. The numeric code is then stored in memory as a binary number.

Over the years, different coding schemes have been developed to represent characters in computer memory. Historically, the most important of these coding schemes is *ASCII*, which stands for the *American Standard Code for Information Interchange*. ASCII is a set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters. For example, the ASCII code for the uppercase letter A is 65. When you type an uppercase A on your computer keyboard, the number 65 is stored in memory (as a binary number, of course). This is shown in Figure 1-13.

Figure 1-13 The letter A is stored in memory as the number 65



TIP: The acronym ASCII is pronounced “askee.”

In case you are curious, the ASCII code for uppercase B is 66, for uppercase C is 67, and so forth. Appendix A shows all of the ASCII codes and the characters they represent.

The ASCII character set was developed in the early 1960s, and was eventually adopted by most of all computer manufacturers. ASCII is limited, however, because it defines codes for only 128 characters. To remedy this, the Unicode character set was developed in the early 1990s. *Unicode* is an extensive encoding scheme that is compatible with ASCII, and can also represent the characters of many of the world's languages. Today, Unicode is quickly becoming the standard character set used in the computer industry.

Advanced Number Storage

Earlier you read about numbers and how they are stored in memory. While reading that section, perhaps it occurred to you that the binary numbering system can be used to represent only integer numbers, beginning with 0. Negative numbers and real numbers (such as 3.14159) cannot be represented using the simple binary numbering technique we discussed.

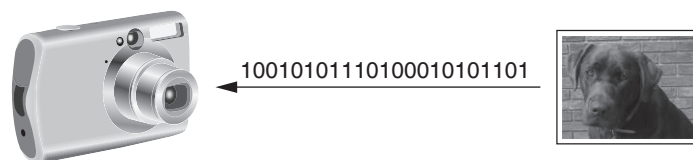
Computers are able to store negative numbers and real numbers in memory, but to do so they use encoding schemes along with the binary numbering system. Negative numbers are encoded using a technique known as *two's complement*, and real numbers are encoded in *floating-point notation*. You don't need to know how these encoding schemes work, only that they are used to convert negative numbers and real numbers to binary format.

Other Types of Data

Computers are often referred to as digital devices. The term *digital* can be used to describe anything that uses binary numbers. *Digital data* is data that is stored in binary, and a *digital device* is any device that works with binary data. In this section, we have discussed how numbers and characters are stored in binary, but computers also work with many other types of digital data.

For example, consider the pictures that you take with your digital camera. These images are composed of tiny dots of color known as *pixels*. (The term pixel stands for *picture element*.) As shown in Figure 1-14, each pixel in an image is converted to a numeric code that represents the pixel's color. The numeric code is stored in memory as a binary number.

Figure 1-14 A digital image is stored in binary format (photo on the right courtesy of Tony Gaddis)



The music that you play on your CD player, iPod, or MP3 player is also digital. A digital song is broken into small pieces known as *samples*. Each sample is converted to a binary number, which can be stored in memory. The more samples that a song is divided into, the more it sounds like the original music when it is played back. A CD-quality song is divided into more than 44,000 samples per second!



Checkpoint

- 1.9 What amount of memory is enough to store a letter of the alphabet or a small number?
- 1.10 What do you call a tiny “switch” that can be set to either on or off?

- 1.11 In what numbering system are all numeric values written as sequences of 0s and 1s?
- 1.12 What is the purpose of ASCII?
- 1.13 What encoding scheme is extensive to represent all the characters of all the languages in the world?
- 1.14 What do the terms “digital data” and “digital device” mean?

1.4

How a Program Works

CONCEPT: A computer’s CPU can only understand instructions that are written in machine language. Because people find it very difficult to write entire programs in machine language, other programming languages have been invented.

Earlier, we stated that the CPU is the most important component in a computer because it is the part of the computer that runs programs. Sometimes the CPU is called the “computer’s brain,” and is described as being “smart.” Although these are common metaphors, you should understand that the CPU is not a brain, and it is not smart. The CPU is an electronic device that is designed to do specific things. In particular, the CPU is designed to perform operations such as the following:

- Reading a piece of data from main memory
- Adding two numbers
- Subtracting one number from another number
- Multiplying two numbers
- Dividing one number by another number
- Moving a piece of data from one memory location to another
- Determining whether one value is equal to another value
- And so forth . . .

As you can see from this list, the CPU performs simple operations on pieces of data. The CPU does nothing on its own, however. It has to be told what to do, and that’s the purpose of a program. A program is nothing more than a list of instructions that cause the CPU to perform operations.

Each instruction in a program is a command that tells the CPU to perform a specific operation. Here’s an example of an instruction that might appear in a program:

```
10110000
```

To you and me, this is only a series of 0s and 1s. To a CPU, however, this is an instruction to perform an operation.¹ It is written in 0s and 1s because CPUs only understand instructions that are written in *machine language*, and machine language instructions are always written in binary.

¹ The example shown is an actual instruction for an Intel microprocessor. It tells the microprocessor to move a value into the CPU.

A machine language instruction exists for each operation that a CPU is capable of performing. For example, there is an instruction for adding numbers; there is an instruction for subtracting one number from another; and so forth. The entire set of instructions that a CPU can execute is known as the CPU's *instruction set*.



NOTE: There are several microprocessor companies today that manufacture CPUs. Some of the more well-known microprocessor companies are Intel, AMD, and Motorola. If you look carefully at your computer, you might find a tag showing a logo for its microprocessor.

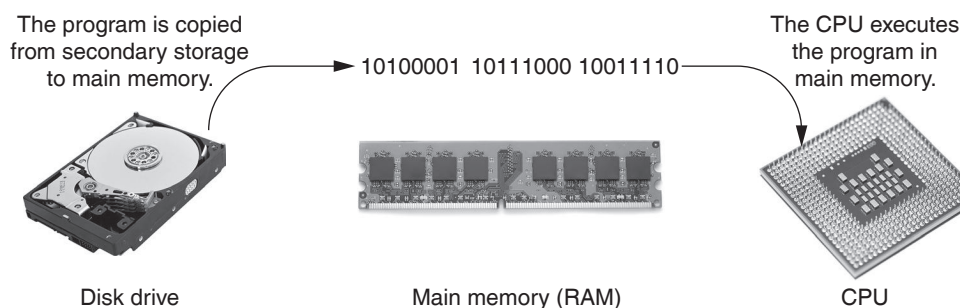
Each brand of microprocessor has its own unique instruction set, which is typically understood only by microprocessors of the same brand. For example, Intel microprocessors understand the same instructions, but they do not understand instructions for Motorola microprocessors.

The machine language instruction that was previously shown is an example of only one instruction. It takes a lot more than one instruction, however, for the computer to do anything meaningful. Because the operations that a CPU knows how to perform are so basic in nature, a meaningful task can be accomplished only if the CPU performs many operations. For example, if you want your computer to calculate the amount of interest that you will earn from your savings account this year, the CPU will have to perform a large number of instructions, carried out in the proper sequence. It is not unusual for a program to contain thousands, or even a million or more machine language instructions.

Programs are usually stored on a secondary storage device such as a disk drive. When you install a program on your computer, the program is typically copied to your computer's disk drive from a CD-ROM, or perhaps downloaded from a Web site.

Although a program can be stored on a secondary storage device such as a disk drive, it has to be copied into main memory, or RAM, each time the CPU executes it. For example, suppose you have a word processing program on your computer's disk. To execute the program you use the mouse to double-click the program's icon. This causes the program to be copied from the disk into main memory. Then, the computer's CPU executes the copy of the program that is in main memory. This process is illustrated in Figure 1-15.

Figure 1-15 A program is copied into main memory and then executed (Courtesy Lefteris Papaulakis/ Shutterstock, Garsya/ Shutterstock and marpan/ Shutterstock)

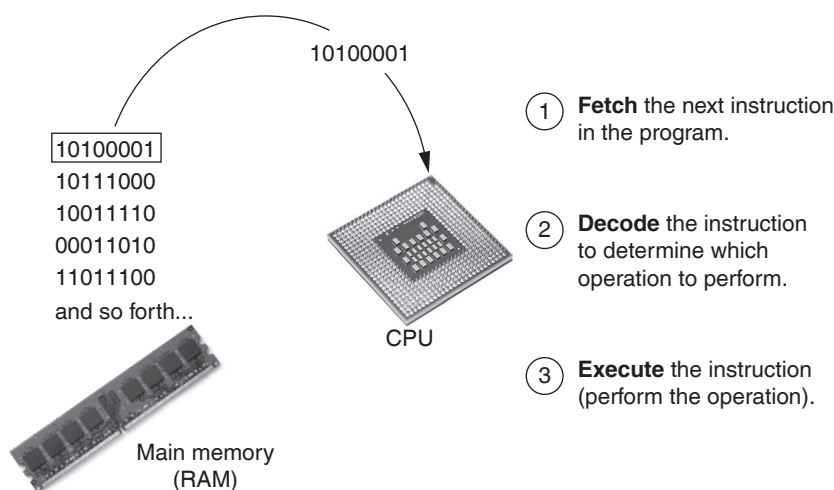


When a CPU executes the instructions in a program, it is engaged in a process that is known as the *fetch-decode-execute cycle*. This cycle, which consists of three steps, is repeated for each instruction in the program. The steps are:

1. **Fetch.** A program is a long sequence of machine language instructions. The first step of the cycle is to fetch, or read, the next instruction from memory into the CPU.
2. **Decode.** A machine language instruction is a binary number that represents a command that tells the CPU to perform an operation. In this step the CPU decodes the instruction that was just fetched from memory, to determine which operation it should perform.
3. **Execute.** The last step in the cycle is to execute, or perform, the operation.

Figure 1-16 illustrates these steps.

Figure 1-16 The fetch-decode-execute cycle (Courtesy Garsya/Shutterstock, marpan/ Shutterstock)



From Machine Language to Assembly Language

Computers can only execute programs that are written in machine language. As previously mentioned, a program can have thousands, or even a million or more binary instructions, and writing such a program would be very tedious and time consuming. Programming in machine language would also be very difficult because putting a 0 or a 1 in the wrong place will cause an error.

Although a computer's CPU only understands machine language, it is impractical for people to write programs in machine language. For this reason, *assembly language* was created in the early days of computing² as an alternative to machine language. Instead of using binary numbers for instructions, assembly language uses short words that are known as *mnemonics*. For example, in assembly language, the mnemonic *add* typically means to add numbers, *mul* typically means to multiply numbers, and *mov* typically means to move a value to a location in memory. When a programmer uses assembly language to write a program, he or she can write short mnemonics instead of binary numbers.

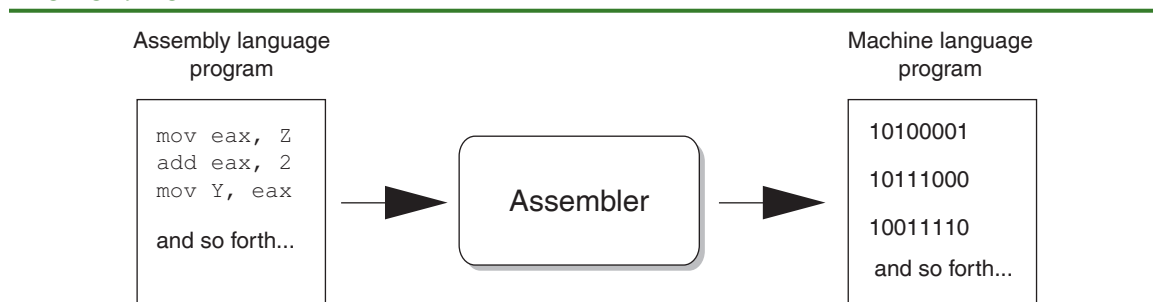
² The first assembly language was most likely developed in the 1940s at Cambridge University for use with a historical computer known as the EDSAC.



NOTE: There are many different versions of assembly language. It was mentioned earlier that each brand of CPU has its own machine language instruction set. Each brand of CPU typically has its own assembly language as well.

Assembly language programs cannot be executed by the CPU, however. The CPU only understands machine language, so a special program known as an *assembler* is used to translate an assembly language program to a machine language program. This process is shown in Figure 1-17. The machine language program that is created by the assembler can then be executed by the CPU.

Figure 1-17 An assembler translates an assembly language program to a machine language program



High-Level Languages

Although assembly language makes it unnecessary to write binary machine language instructions, it is not without difficulties. Assembly language is primarily a direct substitute for machine language, and like machine language, it requires that you know a lot about the CPU. Assembly language also requires that you write a large number of instructions for even the simplest program. Because assembly language is so close in nature to machine language, it is referred to as a *low-level language*.

In the 1950s, a new generation of programming languages known as *high-level languages* began to appear. A high-level language allows you to create powerful and complex programs without knowing how the CPU works, and without writing large numbers of low-level instructions. In addition, most high-level languages use words that are easy to understand. For example, if a programmer were using COBOL (which was one of the early high-level languages created in the 1950s), he or she would write the following instruction to display the message "Hello world" on the computer screen:

Display "Hello world"

Doing the same thing in assembly language would require several instructions, and an intimate knowledge of how the CPU interacts with the computer's video circuitry. As you can see from this example, high-level languages allow programmers to concentrate on the tasks they want to perform with their programs rather than the details of how the CPU will execute those programs.

Since the 1950s, thousands of high-level languages have been created. Table 1-1 lists several of the more well-known languages. If you are working toward a degree in computer science or a related field, you are likely to study one or more of these languages.

Table 1-1 Programming languages

Language	Description
Ada	Ada was created in the 1970s, primarily for applications used by the U.S. Department of Defense. The language is named in honor of Countess Ada Lovelace, an influential and historical figure in the field of computing.
BASIC	B eginners A ll-purpose Symbolic Instruction Code is a general-purpose language that was originally designed in the early 1960s to be simple enough for beginners to learn. Today, there are many different versions of BASIC.
FORTRAN	F ORMula T RANslator was the first high-level programming language. It was designed in the 1950s for performing complex mathematical calculations.
COBOL	C ommon B usiness- O riented L anguage was created in the 1950s, and was designed for business applications.
Pascal	Pascal was created in 1970, and was originally designed for teaching programming. The language was named in honor of the mathematician, physicist, and philosopher Blaise Pascal.
C and C++	C and C++ (pronounced “c plus plus”) are powerful, general-purpose languages developed at Bell Laboratories. The C language was created in 1972 and the C++ language was created in 1983.
C#	Pronounced “c sharp.” This language was created by Microsoft around the year 2000 for developing applications based on the Microsoft .NET platform.
Java	Java was created by Sun Microsystems (a company that is now owned by Oracle) in the early 1990s. It can be used to develop programs that run on a single computer or over the Internet from a Web server.
JavaScript™	JavaScript, created in the 1990s, can be used in Web pages. Despite its name, JavaScript is not related to Java.
Python	Python is a general-purpose language created in the early 1990s. It has become popular in business and academic applications.
Ruby	Ruby is a general-purpose language that was created in the 1990s. It is increasingly becoming a popular language for programs that run on Web servers.
Visual Basic	Visual Basic (commonly known as VB) is a Microsoft programming language and software development environment that allows programmers to create Windows®-based applications quickly. VB was originally created in the early 1990s.

Each high-level language has its own set of words that the programmer must learn in order to use the language. The words that make up a high-level programming language are known as *key words* or *reserved words*. Each key word has a specific meaning, and cannot be used for any other purpose. You previously saw an example of a COBOL statement that uses the key word `display` to print a message on the screen. In the Python language the word `print` serves the same purpose.

In addition to key words, programming languages have *operators* that perform various operations on data. For example, all programming languages have math operators that perform arithmetic. In Java, as well as most other languages, the `+` sign is an operator that adds two numbers. The following adds 12 and 75:

12 + 75

In addition to key words and operators, each language also has its own *syntax*, which is a set of rules that must be strictly followed when writing a program. The syntax rules dictate how key words, operators, and various punctuation characters must be used in a program. When you are learning a programming language, you must learn the syntax rules for that particular language.

The individual instructions that you use to write a program in a high-level programming language are called *statements*. A programming statement can consist of key words, operators, punctuation, and other allowable programming elements, arranged in the proper sequence to perform an operation.



NOTE: Human languages also have syntax rules. Do you remember when you took your first English class, and you learned all those rules about infinitives, indirect objects, clauses, and so forth? You were learning the syntax of the English language.

Although people commonly violate the syntax rules of their native language when speaking and writing, other people usually understand what they mean. Unfortunately, computers do not have this ability. If even a single syntax error appears in a program, the program cannot be executed.

Compilers and Interpreters

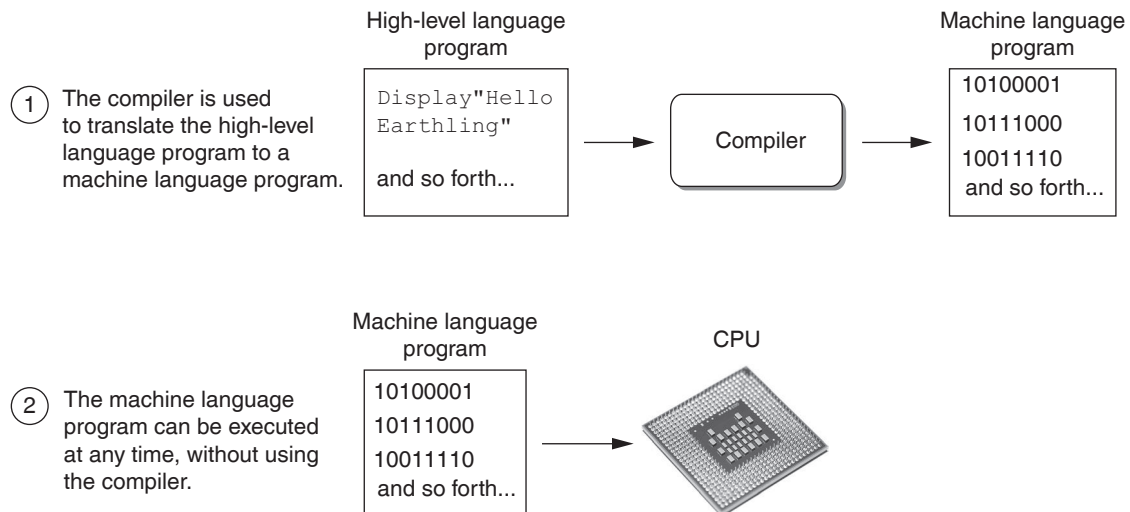


VideoNote
Compiling and
Executing a
Program

Because the CPU understands only machine language instructions, programs that are written in a high-level language must be translated into machine language. Once a program has been written in a high-level language, the programmer will use a compiler or an interpreter to make the translation.

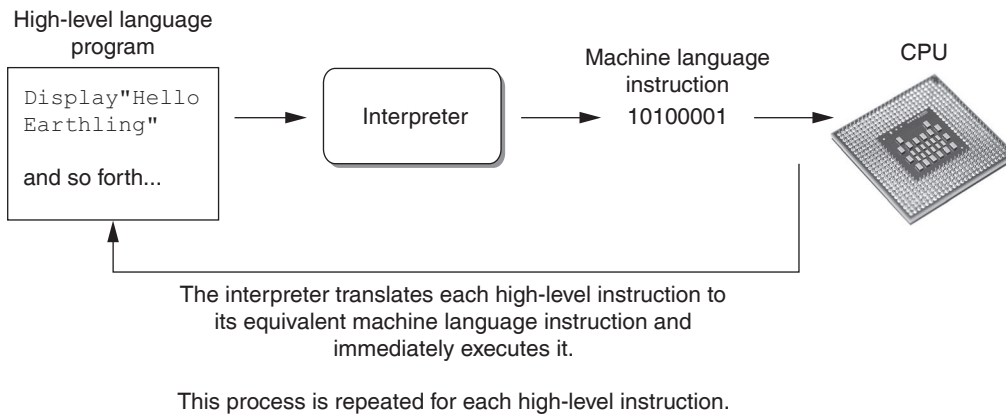
A *compiler* is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed. This is shown in Figure 1-18. As shown in the figure, compiling and executing are two different processes.

Figure 1-18 Compiling a high-level program and executing it (Courtesy marpan/ Shutterstock)



An *interpreter* is a program that both translates and executes the instructions in a high-level language program. As the interpreter reads each individual instruction in the program, it converts it to a machine language instruction and then immediately executes it. This process repeats for every instruction in the program. This process is illustrated in Figure 1-19. Because interpreters combine translation and execution, they typically do not create separate machine language programs.

Figure 1-19 Executing a high-level program with an interpreter (Courtesy marpan/ Shutterstock)



NOTE: Programs that are compiled generally execute faster than programs that are interpreted because a compiled program is already translated entirely to machine language when it is executed. A program that is interpreted must be translated at the time it is executed.

The statements that a programmer writes in a high-level language are called *source code*, or simply *code*. Typically, the programmer types a program's code into a text editor and then saves the code in a file on the computer's disk. Next, the programmer uses a compiler to translate the code into a machine language program, or an interpreter to translate and execute the code. If the code contains a syntax error, however, it cannot be translated. A *syntax error* is a mistake such as a misspelled key word, a missing punctuation character, or the incorrect use of an operator. When this happens the compiler or interpreter displays an error message indicating that the program contains a syntax error. The programmer corrects the error and then attempts once again to translate the program.

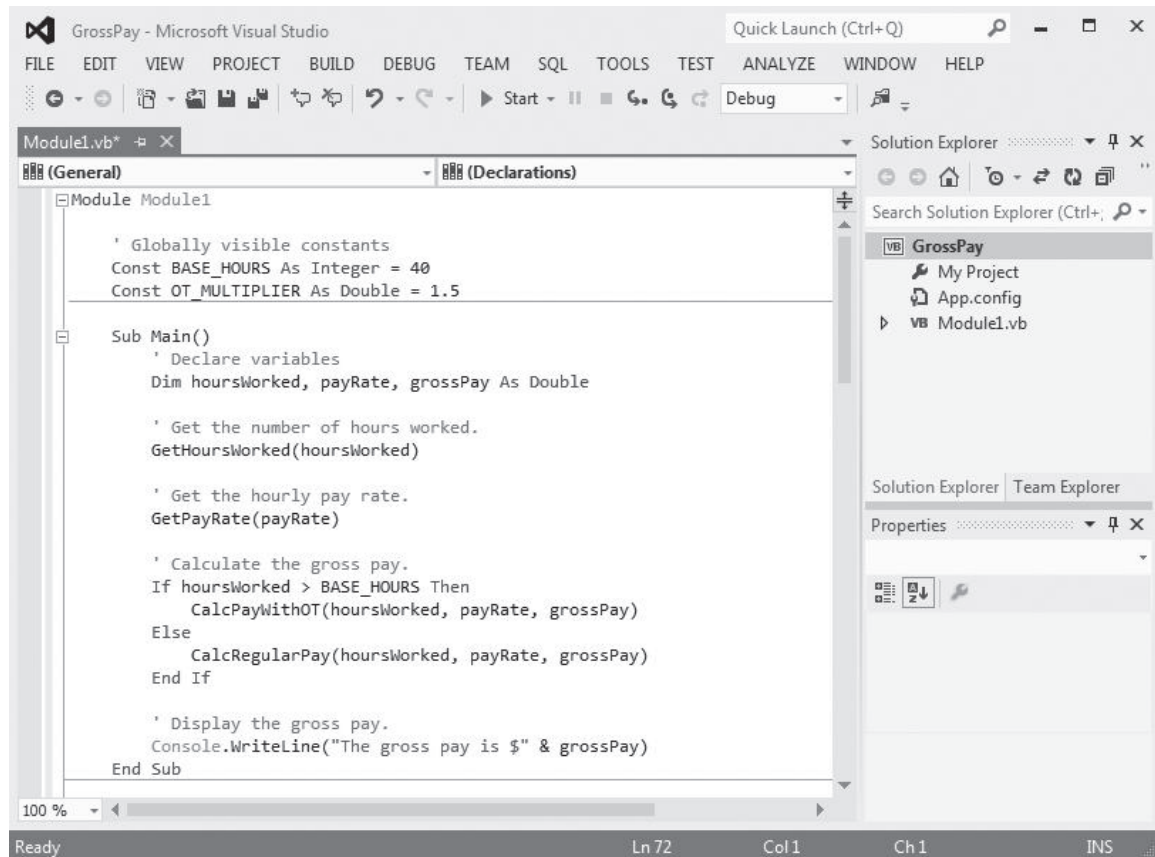
Integrated Development Environments

Although you can use a simple text editor such as Notepad (which is part of the Windows operating system) to write a program, most programmers use specialized software packages called *integrated development environments* or *IDEs*. Most IDEs combine the following programs into one software package:

- A text editor that has specialized features for writing statements in a high-level programming language
- A compiler or interpreter
- Useful tools for testing programs and locating errors

Figure 1-20 shows a screen from Microsoft Visual Studio, a popular IDE for developing programs in the C++, Visual Basic, and C# languages. Eclipse™, NetBeans, Dev-C++, and jGRASP™ are a few other popular IDEs.

Figure 1-20 An integrated development environment (photo courtesy of Microsoft Corporation)



Checkpoint

- 1.15 A CPU understands instructions that are written only in what language?
- 1.16 A program has to be copied into what type of memory each time the CPU executes it?
- 1.17 When a CPU executes the instructions in a program, it is engaged in what process?
- 1.18 What is assembly language?
- 1.19 What type of programming language allows you to create powerful and complex programs without knowing how the CPU works?

- 1.20 Each language has a set of rules that must be strictly followed when writing a program. What is this set of rules called?
- 1.21 What do you call a program that translates a high-level language program into a separate machine language program?
- 1.22 What do you call a program that both translates and executes the instructions in a high-level language program?
- 1.23 What type of mistake is usually caused by a misspelled key word, a missing punctuation character, or the incorrect use of an operator?

1.5

Types of Software

CONCEPT: Programs generally fall into one of two categories: system software or application software. System software is the set of programs that control or enhance the operation of a computer. Application software makes a computer useful for everyday tasks.

If a computer is to function, software is not optional. Everything that a computer does, from the time you turn the power switch on until you shut the system down, is under the control of software. There are two general categories of software: system software and application software. Most computer programs clearly fit into one of these two categories. Let's take a closer look at each.

System Software

The programs that control and manage the basic operations of a computer are generally referred to as *system software*. System software typically includes the following types of programs:

Operating Systems. An *operating system* is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all of the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer. Examples of operating systems that are widely used today are Windows, Mac OS, iOS, Android, and Linux.

Utility Programs. A *utility program* performs a specialized task that enhances the computer's operation or safeguards data. Examples of utility programs are virus scanners, file compression programs, and data backup programs.

Software Development Tools. *Software development tools* are the programs that programmers use to create, modify, and test software. Assemblers, compilers, and interpreters are examples of programs that fall into this category.

Application Software

Programs that make a computer useful for everyday tasks are known as *application software*. These are the programs that people normally spend most of their time running on their computers. Figure 1-1, at the beginning of this chapter, shows screens from two

commonly used applications—Microsoft Word, a word processing program, and Microsoft PowerPoint, a presentation program. Some other examples of application software are spreadsheet programs, email programs, Web browsers, and game programs.



Checkpoint

- 1.24 What fundamental set of programs controls the internal operations of the computer's hardware?
- 1.25 What do you call a program that performs a specialized task, such as a virus scanner, a file compression program, or a data backup program?
- 1.26 Word processing programs, spreadsheet programs, email programs, Web browsers, and game programs belong to what category of software?

Review Questions

Multiple Choice

1. A(n) _____ is a set of instructions that a computer follows to perform a task.
 - a. compiler
 - b. program
 - c. interpreter
 - d. programming language
2. The physical devices that a computer is made of are referred to as _____.
 - a. hardware
 - b. software
 - c. the operating system
 - d. tools
3. The part of a computer that runs programs is called _____.
 - a. RAM
 - b. secondary storage
 - c. main memory
 - d. the CPU
4. Today, CPUs are small chips known as _____.
 - a. ENIACs
 - b. microprocessors
 - c. memory chips
 - d. operating systems
5. The computer stores a program while the program is running, as well as the data that the program is working with, in _____.
 - a. secondary storage
 - b. the CPU
 - c. main memory
 - d. the microprocessor

6. This is a volatile type of memory that is used only for temporary storage while a program is running.
 - a. RAM
 - b. secondary storage
 - c. the disk drive
 - d. the USB drive
7. A type of memory that can hold data for long periods of time—even when there is no power to the computer—is called _____.
 - a. RAM
 - b. main memory
 - c. secondary storage
 - d. CPU storage
8. A component that collects data from people or other devices and sends it to the computer is called _____.
 - a. an output device
 - b. an input device
 - c. a secondary storage device
 - d. main memory
9. A video display is a(n) _____.
 - a. output device
 - b. input device
 - c. secondary storage device
 - d. main memory
10. A _____ is enough memory to store a letter of the alphabet or a small number.
 - a. byte
 - b. bit
 - c. switch
 - d. transistor
11. A byte is made up of eight _____.
 - a. CPUs
 - b. instructions
 - c. variables
 - d. bits
12. In a(n) _____ numbering system, all numeric values are written as sequences of 0s and 1s.
 - a. hexadecimal
 - b. binary
 - c. octal
 - d. decimal
13. A bit that is turned off represents the following value: _____.
 - a. 1
 - b. -1
 - c. 0
 - d. “no”

14. A set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters is _____.
 - a. binary numbering
 - b. ASCII
 - c. Unicode
 - d. ENIAC
15. An extensive encoding scheme that can represent the characters of many of the languages in the world is _____.
 - a. binary numbering
 - b. ASCII
 - c. Unicode
 - d. ENIAC
16. Negative numbers are encoded using the _____ technique.
 - a. two's complement
 - b. floating-point
 - c. ASCII
 - d. Unicode
17. Real numbers are encoded using the _____ technique.
 - a. two's complement
 - b. floating-point
 - c. ASCII
 - d. Unicode
18. The tiny dots of color that digital images are composed of are called _____.
 - a. bits
 - b. bytes
 - c. color packets
 - d. pixels
19. If you were to look at a machine language program, you would see _____.
 - a. Java code
 - b. a stream of binary numbers
 - c. English words
 - d. circuits
20. In the _____ part of the fetch-decode-execute cycle, the CPU determines which operation it should perform.
 - a. fetch
 - b. decode
 - c. execute
 - d. immediately after the instruction is executed
21. Computers can only execute programs that are written in _____.
 - a. Java
 - b. assembly language
 - c. machine language
 - d. C++

22. The _____ translates an assembly language program to a machine language program.
 - a. assembler
 - b. compiler
 - c. translator
 - d. interpreter
23. The words that make up a high-level programming language are called _____.
 - a. binary instructions
 - b. mnemonics
 - c. commands
 - d. key words
24. The rules that must be followed when writing a program are called _____.
 - a. syntax
 - b. punctuation
 - c. key words
 - d. operators
25. A(n) _____ program translates a high-level language program into a separate machine language program.
 - a. assembler
 - b. compiler
 - c. translator
 - d. utility

True or False

1. Today, CPUs are huge devices made of electrical and mechanical components such as vacuum tubes and switches.
2. Main memory is also known as RAM.
3. Any piece of data that is stored in a computer's memory must be stored as a binary number.
4. Images, like the ones you make with your digital camera, cannot be stored as binary numbers.
5. Machine language is the only language that a CPU understands.
6. Assembly language is considered a high-level language.
7. An interpreter is a program that both translates and executes the instructions in a high-level language program.
8. A syntax error does not prevent a program from being compiled and executed.
9. Windows, Mac OS, iOS, Android, and Linux are all examples of application software.
10. Word processing programs, spreadsheet programs, email programs, Web browsers, and games are all examples of utility programs.

Short Answer

1. Why is the CPU the most important component in a computer?
2. What number does a bit that is turned on represent? What number does a bit that is turned off represent?
3. What would you call a device that works with binary data?
4. What are the words that make up a high-level programming language called?
5. What are the short words that are used in assembly language called?
6. What is the difference between a compiler and an interpreter?
7. What type of software controls the internal operations of the computer's hardware?

Exercises

1. Appendix D shows how to convert a decimal number to binary. Use the technique shown in Appendix D to convert the following decimal numbers to binary:

11

65

100

255

2. Use what you've learned about the binary numbering system in this chapter to convert the following binary numbers to decimal:

1101

1000

101011

3. Look at the ASCII chart in Appendix A and determine the codes for each letter of your first name.
4. Use the Web to research the history of the BASIC, C++, Java, and Python programming languages, and answer the following questions:
 - Who was the creator of each of these languages?
 - When was each of these languages created?
 - Was there a specific motivation behind the creation of these languages? If so, what was it?



VideoNote
Converting
Binary to
Decimal

Input, Processing, and Output

TOPICS

- | | | | |
|-----|--------------------------------------|-----|--|
| 2.1 | Designing a Program | 2.6 | Hand Tracing a Program |
| 2.2 | Output, Input, and Variables | 2.7 | Documenting a Program |
| 2.3 | Variable Assignment and Calculations | 2.8 | Designing Your First Program |
| 2.4 | Variable Declarations and Data Types | 2.9 | Focus on Languages: Java, Python,
and C++ |
| 2.5 | Named Constants | | |

2.1

Designing a Program

CONCEPT: Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudocode and flowcharts to create models of programs.

In Chapter 1 you learned that programmers typically use high-level languages to write programs. However, all professional programmers will tell you that a program should be carefully designed before the code is actually written. When programmers begin a new project, they never jump right in and start writing code as the first step. They begin by creating a design of the program.

After designing the program, the programmer begins writing code in a high-level language. Recall from Chapter 1 that each language has its own rules, known as syntax, that must be followed when writing a program. A language's syntax rules dictate things such as how key words, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.

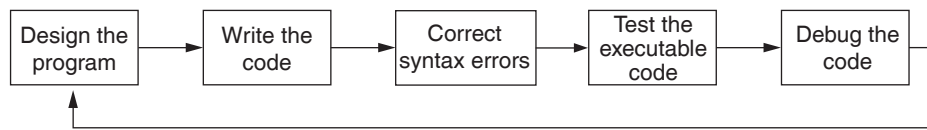
If the program contains a syntax error, or even a simple mistake such as a misspelled key word, the compiler or interpreter will display an error message indicating what the error is. Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all of the syntax errors and simple

typing mistakes have been corrected, the program can be compiled and translated into a machine language program (or executed by an interpreter, depending on the language being used).

Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A *logic error* is a mistake that does not prevent the program from running, but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.)

If there are logic errors, the programmer *debugs* the code. This means that the programmer finds and corrects the code that is causing the error. Sometimes during this process, the programmer discovers that the original design must be changed. This entire process, which is known as the *program development cycle*, is repeated until no errors can be found in the program. Figure 2-1 shows the steps in the process.

Figure 2-1 The program development cycle



This book focuses entirely on the first step of the program development cycle: designing the program. The process of designing a program is arguably the most important part of the cycle. You can think of a program's design as its foundation. If you build a house on a poorly constructed foundation, eventually you will find yourself doing a lot of work to fix the house! A program's design should be viewed no differently. If your program is designed poorly, eventually you will find yourself doing a lot of work to fix the program.

Designing a Program

The process of designing a program can be summarized in the following two steps:

1. Understand the task that the program is to perform.
2. Determine the steps that must be taken to perform the task.

Let's take a closer look at each of these steps.

Understand the Task That the Program Is to Perform

It is essential that you understand what a program is supposed to do before you can determine the steps that the program will perform. Typically, a professional programmer gains this understanding by working directly with the customer. We use the term *customer* to describe the person, group, or organization that is asking you to write a program. This could be a customer in the traditional sense of the word, meaning someone who is paying you to write a program. It could also be your boss, or the manager of a department within your company. Regardless of who it is, the customer will be relying on your program to perform an important task.

To get a sense of what a program is supposed to do, the programmer usually interviews the customer. During the interview, the customer will describe the task that the program

should perform, and the programmer will ask questions to uncover as many details as possible about the task. A follow-up interview is usually needed because customers rarely mention everything they want during the initial meeting, and programmers often think of additional questions.

The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements. A *software requirement* is simply a single function that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.



TIP: If you choose to become a professional software developer, your customer will be anyone who asks you to write programs as part of your job. As long as you are a student, however, your customer is your instructor! In every programming class that you will take, it's practically guaranteed that your instructor will assign programming problems for you to complete. For your academic success, make sure that you understand your instructor's requirements for those assignments and write your programs accordingly.

Determine the Steps That Must Be Taken to Perform the Task

Once you understand the task that the program will perform, you begin by breaking down the task into a series of steps. This is similar to the way you would break down a task into a series of steps that another person can follow. For example, suppose your little sister asks you how to boil water. Assuming she is old enough to be trusted around the stove, you might break down that task into a series of steps as follows:

1. Pour the desired amount of water into a pot.
2. Put the pot on a stove burner.
3. Turn the burner to high.
4. Watch the water until you see large bubbles rapidly rising. When this happens, the water is boiling.

This is an example of an *algorithm*, which is a set of well-defined logical steps that must be taken to perform a task. Notice that the steps in this algorithm are sequentially ordered. Step 1 should be performed before Step 2, and so on. If your little sister follows these steps exactly as they appear, and in the correct order, she should be able to boil water successfully.

A programmer breaks down the task that a program must perform in a similar way. An algorithm is created, which lists all of the logical steps that must be taken. For example, suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee. Here are the steps that you would take:

1. Get the number of hours worked.
2. Get the hourly pay rate.
3. Multiply the number of hours worked by the hourly pay rate.
4. Display the result of the calculation that was performed in Step 3.

Of course, this algorithm isn't ready to be executed on the computer. The steps in this list have to be translated into code. Programmers commonly use two tools to help them accomplish this: pseudocode and flowcharts. Let's look at each of these in more detail.

Pseudocode

Recall from Chapter 1 that each programming language has strict rules, known as syntax, that the programmer must follow when writing a program. If the programmer writes code that violates these rules, a syntax error will result and the program cannot be compiled or executed. When this happens, the programmer has to locate the error and correct it.

Because small mistakes like misspelled words and forgotten punctuation characters can cause syntax errors, programmers have to be mindful of such small details when writing code. For this reason, programmers find it helpful to write their programs in pseudocode (pronounced “sue doe code”) before they write it in the actual code of a programming language.

The word *pseudo* means fake, so *pseudocode* is fake code. It is an informal language that has no syntax rules, and is not meant to be compiled or executed. Instead, programmers use pseudocode to create models, or “mock-ups” of programs. Because programmers don’t have to worry about syntax errors while writing pseudocode, they can focus all of their attention on the program’s design. Once a satisfactory design has been created with pseudocode, the pseudocode can be translated directly to actual code.

Here is an example of how you might write pseudocode for the pay calculating program that we discussed earlier:

```
Display "Enter the number of hours the employee worked."  
Input hours  
Display "Enter the employee's hourly pay rate."  
Input payRate  
Set grossPay = hours * payRate  
Display "The employee's gross pay is $", grossPay
```

Each statement in the pseudocode represents an operation that can be performed in any high-level language. For example, all languages provide a way to display messages on the screen, read input that is typed on the keyboard, and perform mathematical calculations. For now, don’t worry about the details of this particular pseudocode program. As you progress through this chapter you will learn more about each of the statements that you see here.

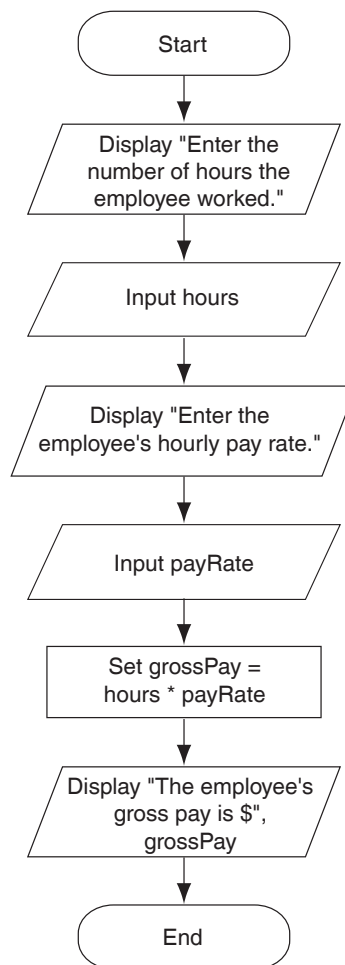


NOTE: As you read the examples in this book, keep in mind that pseudocode is not an actual programming language. It is a generic way to write the statements of an algorithm, without worrying about syntax rules. If you mistakenly write pseudocode into an editor for an actual programming language, such as Python or Visual Basic, errors will result.

Flowcharts

Flowcharting is another tool that programmers use to design programs. A *flowchart* is a diagram that graphically depicts the steps that take place in a program. Figure 2-2 shows how you might create a flowchart for the pay calculating program.

Notice that there are three types of symbols in the flowchart: ovals, parallelograms, and rectangles. The ovals, which appear at the top and bottom of the flowchart, are called

Figure 2-2 Flowchart for the pay calculating program

terminal symbols. The *Start* terminal symbol marks the program's starting point and the *End* terminal symbol marks the program's ending point.

Between the terminal symbols are parallelograms, which are used for both *input symbols* and *output symbols*, and rectangles, which are called *processing symbols*. Each of these symbols represents a step in the program. The symbols are connected by arrows that represent the "flow" of the program. To step through the symbols in the proper order, you begin at the *Start* terminal and follow the arrows until you reach the *End* terminal. Throughout this chapter we will look at each of these symbols in greater detail. For your reference, Appendix B summarizes all of the flowchart symbols that we use in this book.

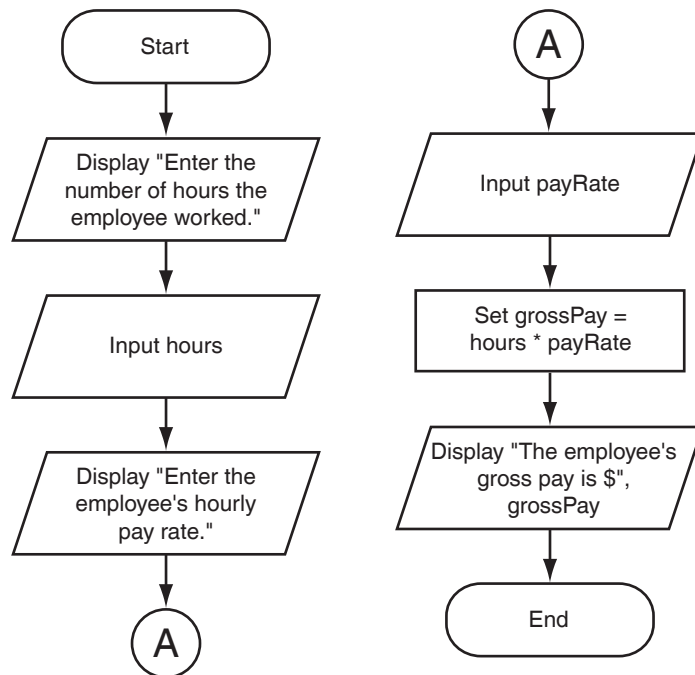
There are a number of different ways that you can draw flowcharts, and your instructor will most likely tell you the way that he or she prefers you to draw them in class. Perhaps the simplest and least expensive way is to simply sketch the flowchart by hand with pencil and paper. If you need to make your hand-drawn flowcharts look more professional, you can visit your local office supply store (or possibly your campus bookstore) and purchase a flowchart template, which is a small plastic sheet that has the flowchart symbols cut into it. You can use the template to trace the symbols onto a piece of paper.

The disadvantage of drawing flowcharts by hand is that mistakes have to be manually erased, and in many cases, require that the entire page be redrawn. A more efficient and professional way to create flowcharts is to use software. There are several specialized software packages available that allow you to create flowcharts.

Flowchart Connector Symbols

Often, a flowchart is too long to fit on a page. Sometimes you can remedy this by breaking the flowchart into two or more smaller flowcharts, and placing them side-by-side on the page. When you do this, you use a *connector symbol* to connect the pieces of the flowchart. A connector symbol is a small circle with a letter or number written inside it. Figure 2-3 shows an example of a flowchart with a connector symbol.

Figure 2-3 Flowchart with a connector symbol

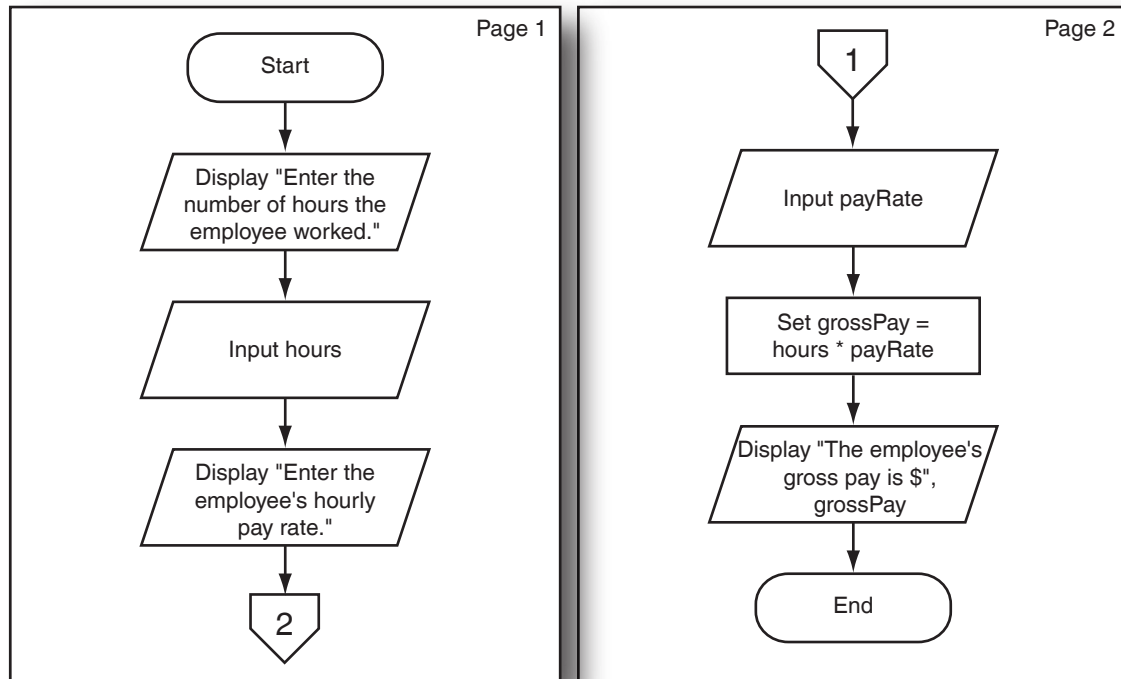


In Figure 2-3, the (A) connector symbol indicates that the second flowchart segment begins where the first flowchart segment ends.

When a flowchart is simply too large to fit on a single page, you can break the flowchart into parts, and place the parts on separate pages. You then use the *off-page connector symbol* to connect the pieces of the flowchart. The off-page connector symbol is the “home plate” shape, with a page number shown inside it. If the connector is at an exit point in the flowchart, the number indicates the page where the next part of the flowchart is located. If the connector is at an entry point in the flowchart, it indicates the page where the previous part of the flowchart is located. Figure 2-4 shows an example. In the figure, the flowchart on the left is on page 1. At the bottom of the flowchart is an off-page connector indicating that the flowchart continues on page 2. In

the flowchart on the right (which is page 2), the off-page connector at the top indicates that the previous part of the flowchart is on page 1.

Figure 2-4 Flowchart with an off-page connector



NOTE: Flowcharting symbols and techniques can vary from one book to another, or from one software package to another. If you are using specialized software to draw flowcharts, you might notice slight differences between some of the symbols that it uses, compared to some of the symbols used in this book.



Checkpoint

- 2.1 Who is a programmer's customer?
- 2.2 What is a software requirement?
- 2.3 What is an algorithm?
- 2.4 What is pseudocode?
- 2.5 What is a flowchart?
- 2.6 What are each of the following symbols in a flowchart?
 - Oval
 - Parallelogram
 - Rectangle

2.2

Output, Input, and Variables

CONCEPT: Output is data that is generated and displayed by the program. Input is data that the program receives. When a program receives data, it stores it in variables, which are named storage locations in memory.

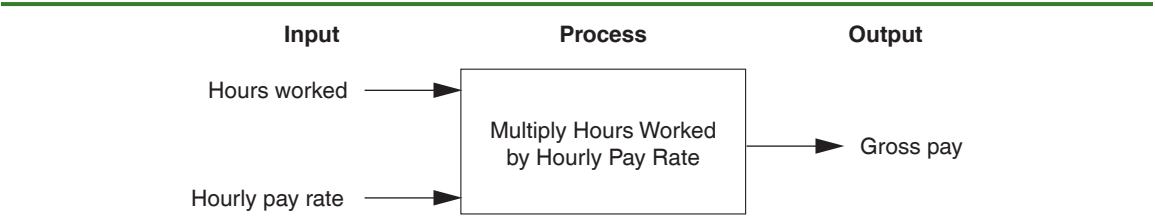
Computer programs typically perform the following three-step process:

- 1. Input is received.
- 2. Some process is performed on the input.
- 3. Output is produced.

Input is any data that the program receives while it is running. One common form of input is data that is typed on the keyboard. Once input is received, some process, such as a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output.

Figure 2-5 illustrates these three steps in the pay calculating program that we discussed earlier. The number of hours worked and the hourly pay rate are provided as input.

Figure 2-5 The input, processing, and output of the pay calculating program



The program processes this data by multiplying the hours worked by the hourly pay rate. The results of the calculation are then displayed on the screen as output.

IPO Charts

An IPO chart is a simple but effective tool that programmers commonly use while designing programs. IPO stands for *input*, *processing*, and *output*, and an *IPO chart* describes the input, processing, and output of a program. These items are usually laid out in columns. The input column shows a description of the data that is required as input. The processing column shows a description of the process, or processes, that the program performs. The output column describes the output that is produced by the program. For example, Figure 2-6 shows an IPO chart for the pay calculating program.

In the remainder of this section, you will look at some simple programs that perform output and input. In the next section, we will discuss how to process data.

Displaying Screen Output

Perhaps the most fundamental thing that you can do in a program is to display a message on the computer screen. As previously mentioned, all high-level languages provide

Figure 2-6 IPO chart for the pay calculating program

IPO Chart for the Pay Calculating Program		
Input	Processing	Output
Number of hours worked Hourly pay rate	Multiply the number of hours worked by the hourly pay rate. The result is the gross pay.	Gross pay

a way to display screen output. In this book, we use the word `Display` to write pseudo-code statements for displaying output on the screen. Here is an example:

```
Display "Hello world"
```

The purpose of this statement is to display the message *Hello world* on the screen. Notice that after the word `Display`, we have written `Hello world` inside quotation marks. The quotation marks are not to be displayed. They simply mark the beginning and the end of the text that we wish to display.

Suppose your instructor tells you to write a pseudocode program that displays your name and address on the computer screen. The pseudocode shown in Program 2-1 is an example of such a program.

Program 2-1



```
Display "Kate Austen"  
Display "1234 Walnut Street"  
Display "Asheville, NC 28899"
```

It is important for you to understand that the statements in this program execute in the order that they appear, from the top of the program to the bottom. This is shown in Figure 2-7. If you translated this pseudocode into an actual program and ran it, the first statement would execute, followed by the second statement, and followed by the third statement. If you try to visualize the way this program’s output would appear on the screen, you should imagine something like that shown in Figure 2-8. Each `Display` statement produces a line of output.



NOTE: Although this book uses the word `Display` for an instruction that displays screen output, some programmers use other words for this purpose. For example, some programmers use the word `Print`, and others use the word `Write`. Pseudocode has no rules that dictate the words that you may or may not use.

Figure 2-7 The statements execute in order (Courtesy of Microsoft Corporation)

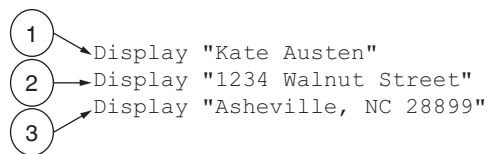


Figure 2-8 Output of Program 2-1 (Courtesy of Microsoft Corporation)

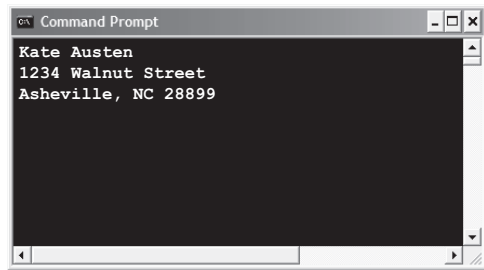
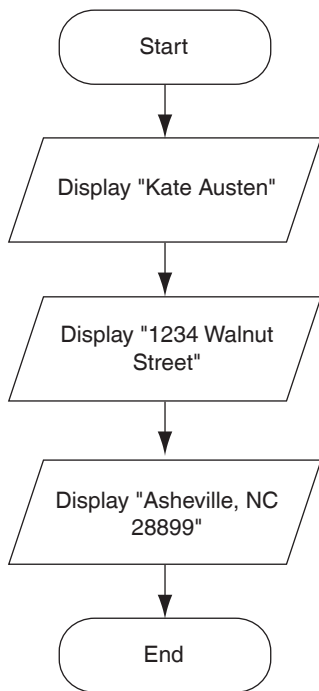


Figure 2-9 shows the way you would draw a flowchart for this program. Notice that between the *Start* and *End* terminal symbols there are three parallelograms. A parallelogram can be either an output symbol or an input symbol. In this program, all three parallelograms are output symbols. There is one for each of the `Display` statements.

Figure 2-9 Flowchart for Program 2-1



Sequence Structures

It was mentioned earlier that the statements in Program 2-1 execute in the order that they appear, from the top of the program to the bottom. A set of statements that execute in the order that they appear is called a *sequence structure*. In fact, all of the programs that you will see in this chapter are sequence structures.

A *structure*, also called a *control structure*, is a logical design that controls the order in which a set of statements executes. In the 1960s, a group of mathematicians proved that only three program structures are needed to write any type of program. The simplest of these structures is the sequence structure. Later in this book, you will learn about the other two structures—decision structures and repetition structures.

Strings and String Literals

Programs almost always work with data of some type. For example, Program 2-1 uses the following three pieces of data:

```
"Kate Austen"  
"1234 Walnut Street"  
"Asheville, NC 28899"
```

These pieces of data are sequences of characters. In programming terms, a sequence of characters that is used as data is called a *string*. When a string appears in the actual code of a program (or in pseudocode, as it does in Program 2-1) it is called a *string literal*. In program code, or pseudocode, a string literal is usually enclosed in quotation marks. As mentioned earlier, the quotation marks simply mark where the string begins and ends.

In this book, we will always enclose string literals in double quote marks ("). Most programming languages use this same convention, but a few use single quote marks (').

Variables and Input



VideoNote
Variables and Input

Quite often a program needs to store data in the computer's memory so it can perform operations on that data. For example, consider the typical online shopping experience: You browse a Web site and add the items that you want to purchase to the shopping cart. As you add items to the shopping cart, data about those items is stored in memory. Then, when you click the checkout button, a program running on the Web site's computer calculates the total of all the items you have in your shopping cart, applicable sales taxes, shipping costs, and the total of all these charges. When the program performs these calculations, it stores the results in the computer's memory.



Programs use variables to store data in memory. A *variable* is a storage location in memory that is represented by a name. For example, a program that calculates the sales tax on a purchase might use a variable named `tax` to hold that value in memory. And a program that calculates the distance from Earth to a distant star might use a variable named `distance` to hold that value in memory.

In this section, we will discuss a basic input operation: reading data that has been typed on the keyboard. When a program reads data from the keyboard, usually it stores that data in a variable so it can be used later by the program. In pseudocode we will read

data from the keyboard with the Input statement. As an example, look at the following statement, which appeared earlier in the pay calculating program:

```
Input hours
```

The word **Input** is an instruction to read a piece of data from the keyboard. The word **hours** is the name of the variable in which that data will be stored. When this statement executes, two things happen:

- The program pauses and waits for the user to type something on the keyboard, and then press the  key.
- When the  key is pressed, the data that was typed is stored in the **hours** variable.

Program 2-2 is a simple pseudocode program that demonstrates the Input statement. Before we examine the program, we should mention a couple of things. First, you will notice that each line in the program is numbered. The line numbers are not part of the pseudocode. We will refer to the line numbers later to point out specific parts of the program. Second, the program's output is shown immediately following the pseudocode. From now on, all pseudocode programs will be shown this way.


Program 2-2



```
1 Display "What is your age?"
2 Input age
3 Display "Here is the value that you entered:"
4 Display age
```

Program Output (with Input Shown in Bold)

```
What is your age?
24 [Enter]
Here is the value that you entered:
24
```

The statement in line 1 displays the string "What is your age?" Then, the statement in line 2 waits for the user to type a value on the keyboard and press . The value that is typed will be stored in the **age** variable. In the example execution of the program, the user has entered 24. The statement in line 3 displays the string "Here is the value that you entered:" and the statement in line 4 displays the value that is stored in the **age** variable.

Notice that in line 4 there are no quotation marks around **age**. If quotation marks were placed around **age**, it would have indicated that we want to display the word *age* instead of the contents of the **age** variable. In other words, the following statement is an instruction to display the contents of the **age** variable:

```
Display age
```

This statement, however, is an instruction to display the word *age*:

```
Display "age"
```



NOTE: In this section, we have mentioned the user. The *user* is simply any hypothetical person that is using a program and providing input for it. The user is sometimes called the *end user*.

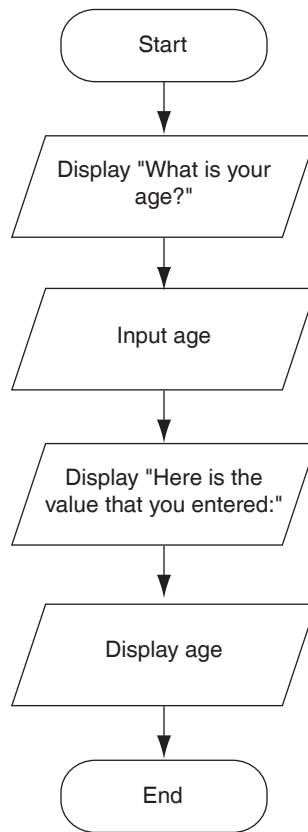
Figure 2-10 Flowchart for Program 2-2

Figure 2-10 shows a flowchart for Program 2-2. Notice that the Input operation is also represented by a parallelogram.

Variable Names

All high-level programming languages allow you to make up your own names for the variables that you use in a program. You don't have complete freedom in naming variables, however. Every language has its own set of rules that you must abide by when creating variable names.

Although the rules for naming variables differ slightly from one language to another, there are some common restrictions:

- Variable names must be one word. They cannot contain spaces.
- In most languages, punctuation characters cannot be used in variable names. It is usually a good idea to use only alphabetic letters and numbers in variable names.
- In most languages, the first character of a variable name cannot be a number.

In addition to following the programming language rules, you should always choose names for your variables that give an indication of what they are used for. For example, a variable that holds the temperature might be named `temperature`, and a variable that holds a car's speed might be named `speed`. You may be tempted to give variables names like `x` and `b2`, but names like these give no clue as to what the variable's purpose is.

Because a variable's name should reflect the variable's purpose, programmers often find themselves creating names that are made of multiple words. For example, consider the following variable names:

```
grosspay
payrate
hotdogssoldtoday
```

Unfortunately, these names are not easily read by the human eye because the words aren't separated. Because we can't have spaces in variable names, we need to find another way to separate the words in a multiword variable name, and make it more readable to the human eye.

One way to do this is to use the underscore character to represent a space. For example, the following variable names are easier to read than those previously shown:

```
gross_pay
pay_rate
hot_dogs_sold_today
```

Another way to address this problem is to use the *camelCase* naming convention. camelCase names are written in the following manner:

- You begin writing the variable name with lowercase letters.
- The first character of the second and subsequent words is written in uppercase.

For example, the following variable names are written in camelCase:

```
grossPay
payRate
hotDogsSoldToday
```

Because the camelCase convention is very popular with programmers, we will use it from this point forward. In fact, you have already seen several programs in this chapter that use camelCase variable names. The pay calculating program shown at the beginning of the chapter uses the variable name `payRate`. Later in this chapter, Program 2-9 uses the variable names `originalPrice` and `salePrice`, and Program 2-11 uses the variable names `futureValue` and `presentValue`.



NOTE: This style of naming is called camelCase because the uppercase characters that appear in a name are sometimes reminiscent of a camel's humps.

Displaying Multiple Items with One Display Statement

If you refer to Program 2-2 you will see that we used the following two `Display` statements in lines 3 and 4:

```
Display "Here is the value that you entered:"
Display age
```

We used two `Display` statements because we needed to display two pieces of data. Line 3 displays the string literal "Here is the value that you entered:" and line 4 displays the contents of the `age` variable.

Most programming languages provide a way to display multiple pieces of data with one statement. Because this is a common feature of programming languages, frequently we will write `Display` statements in our pseudocode that display multiple items. We will simply separate the items with a comma, as shown in line 3 of Program 2-3.

Program 2-3

```
1 Display "What is your age?"
2 Input age
3 Display "Here is the value that you entered: ", age
```

Program Output (with Input Shown in Bold)

```
What is your age?
24 [Enter]
Here is the value that you entered: 24
```

Take a closer look at line 3 of Program 2-3:

```
Display "Here is the value that you entered: ", age
```

↑
Notice the space.

Notice that the string literal "Here is the value that you entered: " ends with a space. That is because in the program output, we want a space to appear after the colon, as shown here:

```
Here is the value that you entered: 24
```

↑
Notice the space.

In most cases, when you are displaying multiple items on the screen, you want to separate those items with spaces between them. Most programming languages do not automatically print spaces between multiple items that are displayed on the screen. For example, look at the following pseudocode statement:

```
Display "January", "February", "March"
```

In most programming languages, such as statement would produce the following output:
JanuaryFebruaryMarch

To separate the strings with spaces in the output, the `Display` statement should be written as:

```
Display "January ", "February ", "March"
```

String Input

Programs 2-2 and 2-3 read numbers from the keyboard, which were stored in variables by `Input` statements. Programs can also read string input. For example, the pseudocode in Program 2-4 uses two `Input` statements: one to read a string and one to read a number.

Program 2-4

```

1 Display "Enter your name."
2 Input name
3 Display "Enter your age."
4 Input age
5 Display "Hello ", name
6 Display "You are ", age, " years old."

```

Program Output (with Input Shown in Bold)

```

Enter your name.
Andrea [Enter]
Enter your age.
24 [Enter]
Hello Andrea
You are 24 years old.

```

The Input statement in line 2 reads input from the keyboard and stores it in the `name` variable. In the example execution of the program, the user entered Andrea. The Input statement in line 4 reads input from the keyboard and stores it in the `age` variable. In the example execution of the program, the user entered 24.

Prompting the User

Getting keyboard input from the user is normally a two-step process:

1. Display a prompt on the screen.
2. Read a value from the keyboard.

A *prompt* is a message that tells (or asks) the user to enter a specific value. For example, the pseudocode in Program 2-3 gets the user to enter his or her age with the following statements:

```

Display "What is your age?"
Input age

```

In most programming languages, the statement that reads keyboard input does not display instructions on the screen. It simply causes the program to pause and wait for the user to type something on the keyboard. For this reason, whenever you write a statement that reads keyboard input, you should also write a statement just before it that tells the user what to enter. Otherwise, the user will not know what he or she is expected to do. For example, suppose we remove line 1 from Program 2-3, as follows:

```

Input age
Display "Here is the value that you entered: ", age

```

If this were an actual program, can you see what would happen when it is executed? The screen would appear blank because the Input statement would cause the program to wait for something to be typed on the keyboard. The user would probably think the computer was malfunctioning.

The term *user-friendly* is commonly used in the software business to describe programs that are easy to use. Programs that do not display adequate or correct instructions are frustrating to use, and are not considered user-friendly. One of the simplest things that you can do to increase a program's user-friendliness is to make sure that it displays clear, understandable prompts prior to each statement that reads keyboard input.



TIP: Sometimes we computer science instructors jokingly tell our students to write programs as if “Uncle Joe” or “Aunt Sally” were the user. Of course, these are not real people, but imaginary users who are prone to making mistakes if not told exactly what to do. When you are designing a program, you should imagine that someone who knows nothing about the program’s inner workings will be using it.



Checkpoint

- 2.7 What are the three operations that programs typically perform?
- 2.8 What is an IPO chart?
- 2.9 What is a sequence structure?
- 2.10 What is a string? What is a string literal?
- 2.11 A string literal is usually enclosed inside a set of what characters?
- 2.12 What is a variable?
- 2.13 Summarize three common rules for naming variables.
- 2.14 What variable naming convention do we follow in this book?
- 2.15 Look at the following pseudocode statement:
 Input temperature
 What happens when this statement executes?
- 2.16 Who is the user?
- 2.17 What is a prompt?
- 2.18 What two steps usually take place when a program prompts the user for input?
- 2.19 What does the term *user-friendly* mean?

2.3

Variable Assignment and Calculations

CONCEPT: You can store a value in a variable with an assignment statement. The value can be the result of a calculation, which is created with math operators.

Variable Assignment

In the previous section, you saw how the Input statement gets a value typed on the keyboard and stores it in a variable. You can also write statements that store specific values in variables. The following is an example, in pseudocode:

```
Set price = 20
```

This is called an assignment statement. An *assignment statement* sets a variable to a specified value. In this case, the variable `price` is set to the value 20. When we write an assignment statement in pseudocode, we will write the word `Set`, followed by the name

of the variable, followed by an equal sign (=), followed by the value we want to store in the variable. The pseudocode in Program 2-5 shows another example.

Program 2-5

```
1 Set dollars = 2.75
2 Display "I have ", dollars, " in my account."
```

Program Output

I have 2.75 in my account.

In line 1, the value 2.75 is stored in the `dollars` variable. Line 2 displays the message “I have 2.75 in my account.” Just to make sure you understand how the `Display` statement in line 2 is working, let’s walk through it. The word `Display` is followed by three pieces of data, so that means it will display three things. The first thing it displays is the string literal “I have “. Next, it displays the contents of the `dollars` variable, which is 2.75. Last, it displays the string literal “ in my account.”

Variables are called “variable” because they can hold different values while a program is running. Once you set a variable to a value, that value will remain in the variable until you store a different value in the variable. For example, look at the pseudocode in Program 2-6.

Program 2-6

```
1 Set dollars = 2.75
2 Display "I have ", dollars, " in my account."
3 Set dollars = 99.95
4 Display "But now I have ", dollars, " in my account!"
```

Program Output

I have 2.75 in my account.
But now I have 99.95 in my account!

Line 1 sets the `dollars` variable to 2.75, so when the statement in line 2 executes, it displays “I have 2.75 in my account.” Then, the statement in line 3 sets the `dollars` variable to 99.95. As a result, the value 99.95 replaces the value 2.75 that was previously stored in the variable. When line 4 executes, it displays “But now I have 99.95 in my account!” This program illustrates two important characteristics of variables:

- A variable holds only one value at a time.
- When you store a value in a variable, that value replaces the previous value that was in the variable.



NOTE: When writing an assignment statement, all programming languages require that you write the name of the variable that is receiving the value on the left side of the = operator. For example, the following statement is incorrect:

Set 99.95 = dollars ← This is an error!

A statement such as this would be considered a syntax error.



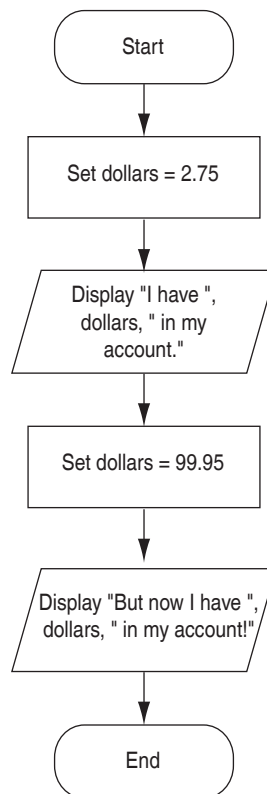
NOTE: In this book, we have chosen to start variable assignment statements with the word **Set** because it makes it clear that we are setting a variable to a value. In most programming languages, however, assignment statements do not start with the word **Set**. In most languages, an assignment statement looks similar to the following:

```
dollars = 99.95
```

If your instructor allows it, it is permissible to write assignment statements without the word **Set** in your pseudocode. Just be sure to write the name of the variable that is receiving the value on the left side of the equal sign.

In flowcharts, an assignment statement appears in a processing symbol, which is a rectangle. Figure 2-11 shows a flowchart for Program 2-6.

Figure 2-11 Flowchart for Program 2-6



Performing Calculations

Most real-world algorithms require calculations to be performed. A programmer's tools for performing calculations are *math operators*. Programming languages commonly provide the operators shown in Table 2-1.



Table 2-1 Common math operators¹

Symbol	Operator	Description
+	Addition	Adds two numbers
—	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the quotient
MOD	Modulus	Divides one number by another and gives the remainder
^	Exponent	Raises a number to a power

Programmers use the operators shown in Table 2-1 to create math expressions. A *math expression* performs a calculation and gives a value. The following is an example of a simple math expression:

12 + 2

The values on the right and left of the + operator are called *operands*. These are values that the + operator adds together. The value that is given by this expression is 14.

Variables may also be used in a math expression. For example, suppose we have two variables named `hours` and `payRate`. The following math expression uses the * operator to multiply the value in the `hours` variable by the value in the `payRate` variable:

`hours * payRate`

When we use a math expression to calculate a value, normally we want to save that value in memory so we can use it again in the program. We do this with an assignment statement. Program 2-7 shows an example.

Program 2-7

```
1 Set price = 100
2 Set discount = 20
3 Set sale = price - discount
4 Display "The total cost is $", sale
```

Program Output

The total cost is \$80

Line 1 sets the `price` variable to 100, and line 2 sets the `discount` variable to 20. Line 3 sets the `sale` variable to the result of the expression `price - discount`. As you can see from the program output, the `sale` variable holds the value 80.

¹ In some programming languages, the % character is used as the modulus operator, and sometimes the ** characters are used as the exponent operator.



In the Spotlight:

Calculating Cell Phone Overage Fees

Suppose your cell phone calling plan allows you to use 700 minutes per month. If you use more than this limit in a month, you are charged an overage fee of 35 cents for each excess minute. Your phone shows you the number of excess minutes that you have used in the current month, but it does not show you how much your overage fee currently is. Until now, you've been doing the math the old-fashioned way (with pencil and paper, or with a calculator), but you would like to design a program that will simplify the task. You would like to be able to enter the number of excess minutes, and have the program perform the calculation for you.

First, you want to make sure that you understand the steps that the program must perform. It will be helpful if you closely look at the way you've been solving this problem, using only paper and pencil, or calculator:

Manual Algorithm (Using pencil and paper, or calculator)

1. You get the number of excess minutes that you have used.
2. You multiply the number of excess minutes by 0.35.
3. The result of the calculation is your current overage fee.

Ask yourself the following questions about this algorithm:

Question: What input do I need to perform this algorithm?

Answer: I need the number of excess minutes.

Question: What must I do with the input?

Answer: I must multiply the input (the number of excess minutes) by 0.35. The result of that calculation is the overage fee.

Question: What output must I produce?

Answer: The overage fee.

Now that you have identified the input, the process that must be performed, and the output, you can write the general steps of the program's algorithm:

Computer Algorithm

1. Get the number of excess minutes as input.
2. Calculate the overage fee by multiplying the number of excess minutes by 0.35.
3. Display the overage fee.

In Step 1 of the computer algorithm, the program gets the number of excess minutes from the user. Any time a program needs the user to enter a piece of data, it does two things: (1) it displays a message prompting the user for the piece of data, and (2) it reads the data that the user enters on the keyboard, and stores that data in a variable. In pseudocode, Step 1 of the algorithm will look like this:

Display "Enter the number of excess minutes."

Input excessMinutes

Notice that the Input statement stores the value entered by the user in a variable named `excessMinutes`.

In Step 2 of the computer algorithm, the program calculates the overage fee by multiplying the number of excess minutes by 0.35. The following pseudocode statement performs this calculation, and stores the result in a variable named `overageFee`:

```
Set overageFee = excessMinutes * 0.35
```

In Step 3 of the computer algorithm, the program displays the overage fee. Because the overage fee is stored in the `overageFee` variable, the program will display a message that shows the value of the `overageFee` variable. In pseudocode we will use the following statement:

```
Display "Your current overage fee is $", overageFee
```

Program 2-8 shows the entire pseudocode program, with example output. Figure 2-12 shows the flowchart for this program.

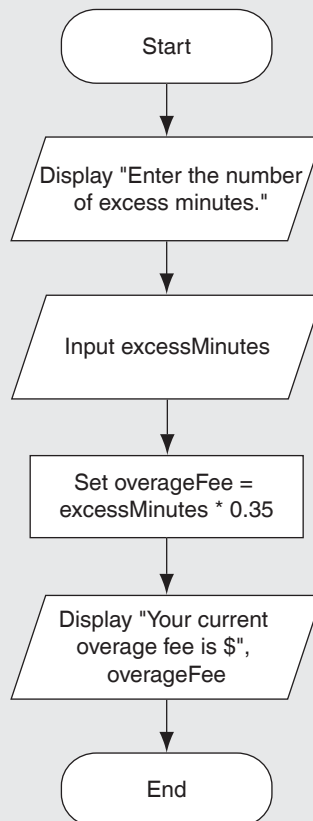
Program 2-8

```
1 Display "Enter the number of excess minutes."  
2 Input excessMinutes  
3 Set overageFee = excessMinutes * 0.35  
4 Display "Your current overage fee is $", overageFee
```

Program Output (with Input Shown in Bold)

```
Enter the number of excess minutes.  
100 [Enter]  
Your current overage fee is $35
```

Figure 2-12 Flowchart for Program 2-8





In the Spotlight:

Calculating a Percentage

Determining percentages is a common calculation in computer programming. In mathematics, the % symbol is used to indicate a percentage, but most programming languages don't use the % symbol for this purpose. In a program, you usually have to convert a percentage to a decimal number. For example, 50 percent would be written as 0.5 and 2 percent would be written as 0.02.

Let's step through the process of writing a program that calculates a percentage. Suppose a retail business is planning to have a storewide sale where the prices of all items will be 20 percent off. We have been asked to write a program to calculate the sale price of an item after the discount is subtracted. Here is the algorithm:

1. Get the original price of the item.
2. Calculate 20 percent of the original price. This is the amount of the discount.
3. Subtract the discount from the original price. This is the sale price.
4. Display the sale price.

In Step 1 we get the original price of the item. We will prompt the user to enter this data on the keyboard. Recall from the previous section that prompting the user is a two-step process: (1) display a message telling the user to enter the desired data, and (2) reading that data from the keyboard. We will use the following pseudocode statements to do this. Notice that the value entered by the user will be stored in a variable named `originalPrice`.

```
Display "Enter the item's original price."  
Input originalPrice
```

In Step 2, we calculate the amount of the discount. To do this we multiply the original price by 20 percent. The following statement performs this calculation and stores the result in the `discount` variable.

```
Set discount = originalPrice * 0.2
```

In Step 3, we subtract the discount from the original price. The following statement does this calculation and stores the result in the `salePrice` variable.

```
Set salePrice = originalPrice - discount
```

Last, in Step 4, we will use the following statement to display the sale price:

```
Display "The sale price is $", salePrice
```

Program 2-9 shows the entire pseudocode program, with example output. Figure 2-13 shows the flowchart for this program.

Program 2-9



```
1 Display "Enter the item's original price."  
2 Input originalPrice  
3 Set discount = originalPrice * 0.2  
4 Set salePrice = originalPrice - discount  
5 Display "The sale price is $", salePrice
```

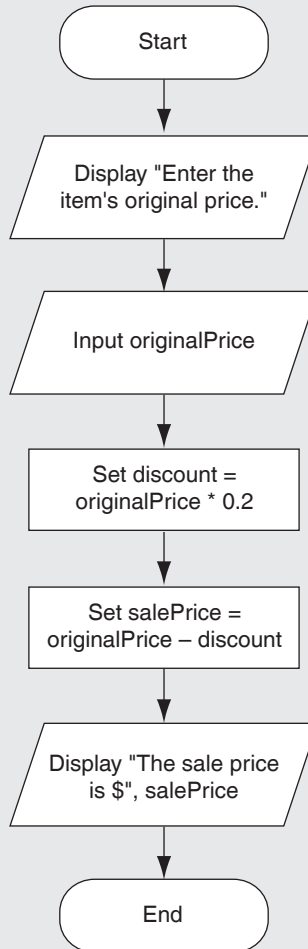
Program Output (with Input Shown in Bold)

Enter the item's original price.

100 [Enter]

The sale price is \$80

Figure 2-13 Flowchart for Program 2-9



The Order of Operations

It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, the variable *x*, 21, and the variable *y* to the variable *answer*.

Set *answer* = 17 + *x* + 21 + *y*

Some expressions are not that straightforward, however. Consider the following statement:

Set *outcome* = 12 + 6 / 3

What value will be stored in `outcome`? The number 6 is used as an operand for both the addition and division operators. The `outcome` variable could be assigned either 6 or 14, depending on when the division takes place. The answer is 14 because the *order of operations* dictates that the division operator works before the addition operator does.

In most programming languages, the order of operations can be summarized as follows:

1. Perform any operations that are enclosed in parentheses.
2. Perform any operations that use the exponent operator to raise a number to a power.
3. Perform any multiplications, divisions, or modulus operations as they appear from left to right.
4. Perform any additions or subtractions as they appear from left to right.

Mathematical expressions are evaluated from left to right. When two operators share an operand, the order of operations determines which operator works first. Multiplication and division are always performed before addition and subtraction, so the statement

```
Set outcome = 12 + 6 / 3
```

works like this:

1. 6 is divided by 3, yielding a result of 2
2. 12 is added to 2, yielding a result of 14

It could be diagrammed as shown in Figure 2-14.

Figure 2-14 The order of operations at work

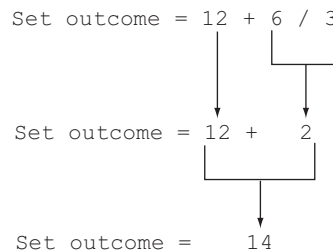


Table 2-2 shows some other sample expressions with their values.

Table 2-2 Some expressions and their values

Expression	Value
$5 + 2 * 4$	13
$10 / 2 - 3$	2
$8 + 12 * 2 - 4$	28
$6 - 3 * 2 + 7 - 1$	6

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the variables `a` and `b` are added together, and their sum is divided by 4:

```
Set result = (a + b) / 4
```

Without the parentheses, however, `b` would be divided by 4 and the result added to `a`. Table 2-3 shows more expressions and their values.

Table 2-3 More expressions and their values

Expression	Value
<code>(5 + 2) * 4</code>	28
<code>10 / (5 - 3)</code>	5
<code>8 + 12 * (6 - 2)</code>	56
<code>(6 - 3) * (2 + 7) / 3</code>	9



NOTE: Parentheses can be used to enhance the clarity of a math expression, even when they are unnecessary to get the correct result. For example, look at the following statement:

`Set fahrenheit = celsius * 1.8 + 32`

Even when it is unnecessary to get the correct result, we can insert parentheses to clearly show that `celsius * 1.8` happens first in the math expression:

`Set fahrenheit = (celsius * 1.8) + 32`

In the Spotlight: Calculating an Average



Determining the average of a group of values is a simple calculation: You add all of the values and then divide the sum by the number of values. Although this is a straightforward calculation, it is easy to make a mistake when writing a program that calculates an average. For example, let's assume that the variables `a`, `b`, and `c` each hold a value and we want to calculate the average of those values. If we are careless, we might write a statement such as the following to perform the calculation:

`Set average = a + b + c / 3`

Can you see the error in this statement? When it executes, the division will take place first. The value in `c` will be divided by 3, and then the result will be added to `a + b`. That is not the correct way to calculate an average. To correct this error we need to put parentheses around `a + b + c`, as shown here:

`Set average = (a + b + c) / 3`

Let's step through the process of writing a program that calculates an average. Suppose you have taken three tests in your computer science class, and you want to write a program that will display the average of the test scores. Here is the algorithm:

1. Get the first test score.
2. Get the second test score.
3. Get the third test score.
4. Calculate the average by adding the three test scores and dividing the sum by 3.
5. Display the average.

In Steps 1, 2, and 3 we will prompt the user to enter the three test scores. We will store those test scores in the variables `test1`, `test2`, and `test3`. In Step 4 we will calculate the average of the three test scores. We will use the following statement to perform the calculation and store the result in the `average` variable:

```
Set average = (test1 + test2 + test3) / 3
```

Last, in Step 5, we display the average. Program 2-10 shows the pseudocode for this program, and Figure 2-15 shows the flowchart.

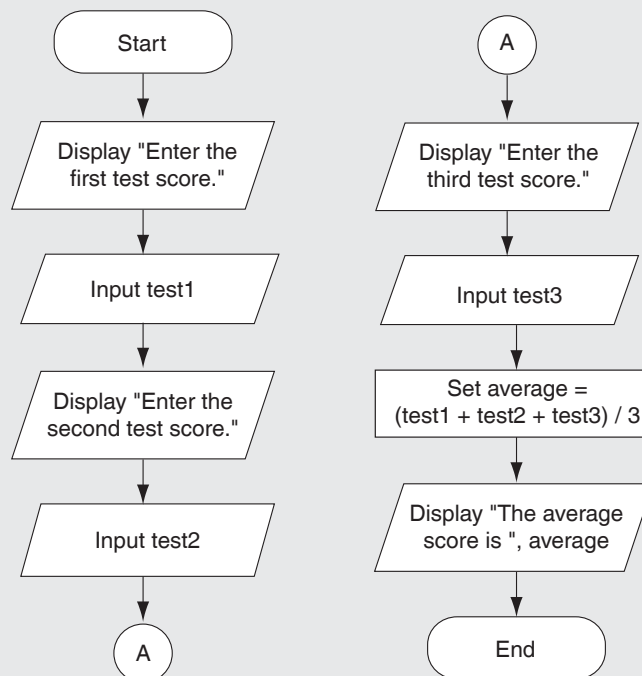
Program 2-10

```
1 Display "Enter the first test score."
2 Input test1
3 Display "Enter the second test score."
4 Input test2
5 Display "Enter the third test score."
6 Input test3
7 Set average = (test1 + test2 + test3) / 3
8 Display "The average score is ", average
```

Program Output (with Input Shown in Bold>

```
Enter the first test score.
90 [Enter]
Enter the second test score.
80 [Enter]
Enter the third test score.
100 [Enter]
The average score is 90
```

Figure 2-15 Flowchart for Program 2-10



Advanced Arithmetic Operators:
Exponent and Modulus

In addition to the basic math operators for addition, subtraction, multiplication, and division, many languages provide an exponent operator and a modulus operator. The `^` symbol is commonly used as the exponent operator, and its purpose is to raise a number to a power. For example, the following pseudocode statement raises the `length` variable to the power of 2 and stores the result in the `area` variable:

```
Set area = length^2
```

The word `MOD` is used in many languages as the modulus operator. (Some languages use the `%` symbol for the same purpose.) The modulus operator performs division, but instead of returning the quotient, it returns the remainder. The following statement assigns 2 to `leftover`:

```
Set leftover = 17 MOD 3
```

This statement assigns 2 to `leftover` because 17 divided by 3 is 5 with a remainder of 2. You will not use the modulus operator frequently, but it is useful in some situations. It is commonly used in calculations that detect odd or even numbers, determine the day of the week, measure the passage of time, and other specialized operations.

Converting Math Formulas to Programming Statements

You probably remember from algebra class that the expression $2xy$ is understood to mean 2 times x times y . In math, you do not always use an operator for multiplication. Programming languages, however, require an operator for any mathematical operation. Table 2-4 shows some algebraic expressions that perform multiplication and the equivalent programming expressions.

Table 2-4 Algebraic expressions

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	<code>6 * B</code>
$(3)(12)$	3 times 12	<code>3 * 12</code>
$4xy$	4 times x times y	<code>4 * x * y</code>

When converting some algebraic expressions to programming expressions, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following formula:

$$x = \frac{a + b}{c}$$

To convert this to a programming statement, $a + b$ will have to be enclosed in parentheses:

```
Set x = (a + b) / c
```

Table 2-5 shows additional algebraic expressions and their pseudocode equivalents.

Table 2-5 Algebraic expressions and pseudocode statements

Algebraic Expression	Pseudocode Statement
$y = 3\frac{x}{2}$	Set <code>y = x / 2 * 3</code>
$z = 3bc + 4$	Set <code>z = 3 * b * c + 4</code>
$a = \frac{x + 2}{a - 1}$	Set <code>a = (x + 2) / (a - 1)</code>

In the Spotlight:

Converting a Math Formula to a Programming Statement



Suppose you want to deposit a certain amount of money into a savings account, and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have \$10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the present value, or the amount that you need to deposit today.
- F is the future value that you want in the account. (In this case, F is \$10,000.)
- r is the annual interest rate.
- n is the number of years that you plan to let the money sit in the account.

It would be nice to write a computer program to perform the calculation, because then we can experiment with different values for the terms. Here is an algorithm that we can use:

1. Get the desired future value.
2. Get the annual interest rate.
3. Get the number of years that the money will sit in the account.
4. Calculate the amount that will have to be deposited.
5. Display the result of the calculation in Step 4.

In Steps 1 through 3, we will prompt the user to enter the specified values. We will store the desired future value in a variable named `futureValue`, the annual interest rate in a variable named `rate`, and the number of years in a variable named `years`.

In Step 4, we calculate the present value, which is the amount of money that we will have to deposit. We will convert the formula previously shown to the following pseudocode statement. The statement stores the result of the calculation in the `presentValue` variable.

```
Set presentValue = futureValue / (1 + rate)^years
```

In Step 5, we display the value in the `presentValue` variable. Program 2-11 shows the pseudocode for this program, and Figure 2-16 shows the flowchart.