CHAPTER 1

Introduction to Computers and Programming

TOPICS

1.1 Why Program?

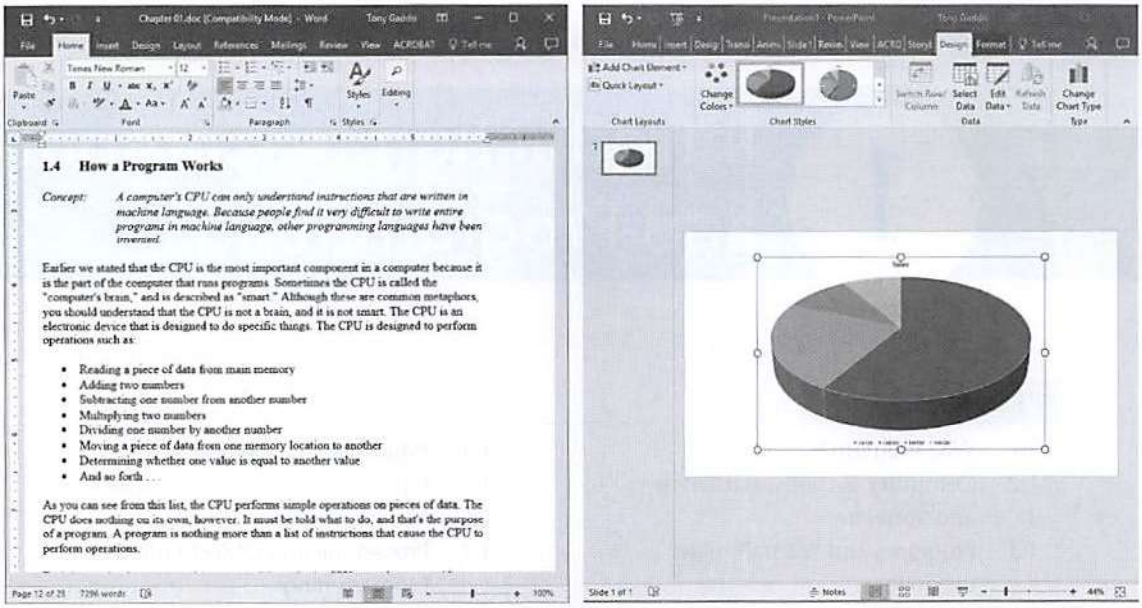
CONCEPT: Computers can do many different jobs because they are programmable.

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 e-mail, and participating in online classes. At work, people use computers to conduct business transactions, communicate with customers and coworkers, analyze data, make presentations, control machines in manufacturing facilities, and many many other tasks. At home, people use computers for tasks such as paying bills, shopping online, social networking, and playing computer games. And don't forget that smartphones, MP3 players, DVRs, car navigation systems, and many other devices are computers as well. 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 computers are not designed to do just one job, but 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 using Microsoft Word and PowerPoint, two commonly used programs.

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 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, academia, entertainment, and almost every other field.

Figure 1-1 A word processing program and a presentation program



Computer programming is both an art and a science. It is an art because every aspect of a program should be carefully designed. Listed below are a few of the things that must be designed for any real-world computer program:

- The logical flow of the instructions
- The mathematical procedures
- The appearance of the screens
- The way information is presented to the user
- The program's "user-friendliness"
- Documentation, help files, tutorials, and so on

There is also a scientific, or engineering, side to programming. Because programs rarely work right the first time they are written, a lot of testing, correction, and redesigning is required. This demands patience and persistence from the programmer. Writing software demands discipline as well. Programmers must learn special languages like C++ because computers do not understand English or other human languages. Languages such as C++ have strict rules that must be carefully followed.

Both the artistic and scientific nature of programming make writing computer software like designing a car: Both cars and programs should be functional, efficient, powerful, easy to use, and pleasing to look at.

1.2 Computer Systems: Hardware and Software

CONCEPT: All computer systems consist of similar hardware devices and software components. This section provides an overview of standard computer hardware and software organization.

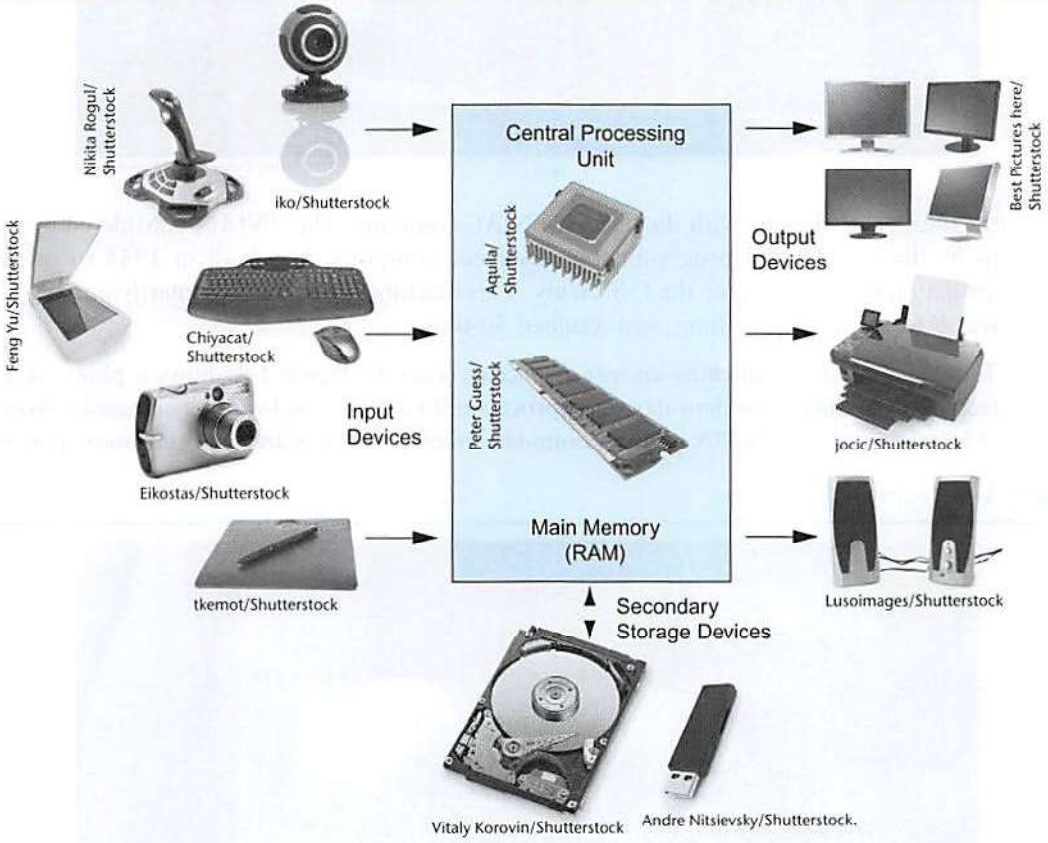
Hardware

Hardware refers to the physical components of which a computer is made. A computer, as we generally think of it, is not an individual device, but a system of devices. Like the instruments in a symphony orchestra, each device plays its own part. A typical computer system consists of the following major components:

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

The organization of a computer system is depicted in Figure 1-2.

Figure 1-2 Typical devices in a computer system

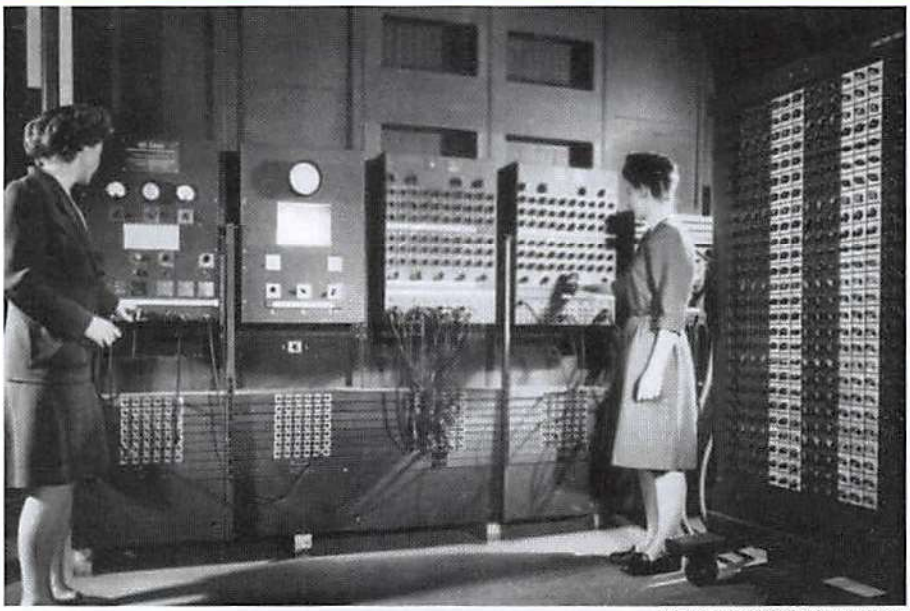


The CPU

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 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

Figure 1-3 The ENIAC computer

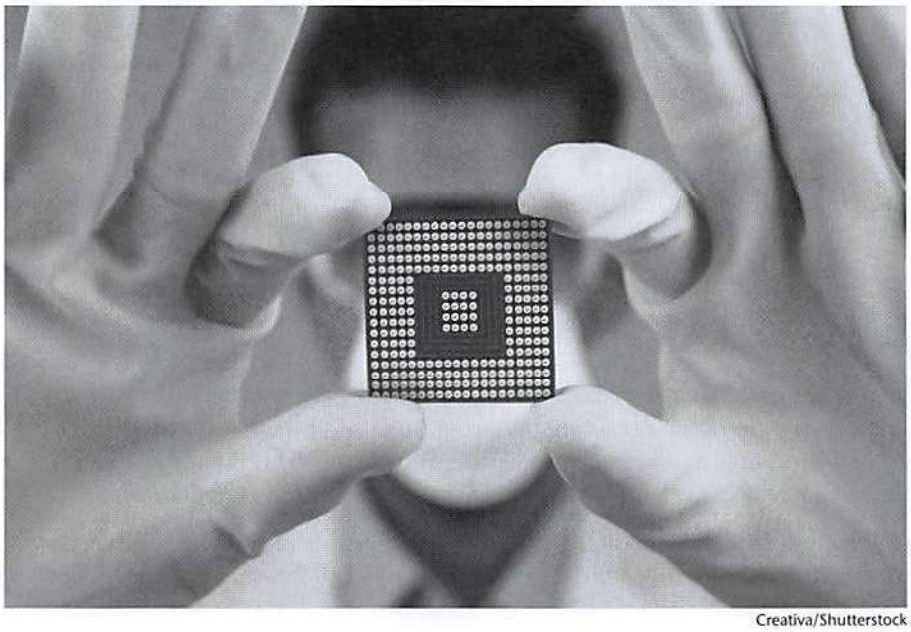


U.S. Army Center of Military History

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 electromechanical CPUs in early computers, microprocessors are also much more powerful.

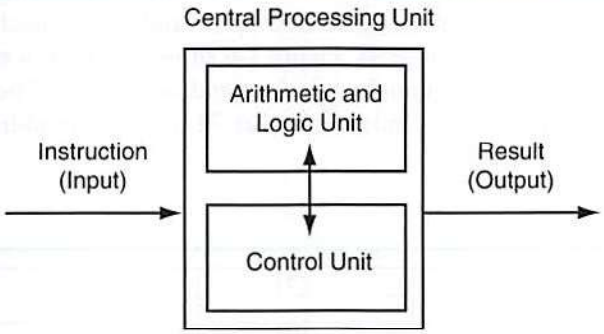
Figure 1-4 A microprocessor



Creativa/Shutterstock

The CPU's job is to fetch instructions, follow the instructions, and produce some result. Internally, the central processing unit consists of two parts: the *control unit* and the *arithmetic and logic unit (ALU)*. The control unit coordinates all of the computer's operations. It is responsible for determining where to get the next instruction and regulating the other major components of the computer with control signals. The arithmetic and logic unit, as its name suggests, is designed to perform mathematical operations. The organization of the CPU is shown in Figure 1-5.

Figure 1-5 Organization of a CPU



A program is a sequence of instructions stored in the computer's memory. When a computer is running a program, the CPU is engaged in a process known formally as the *fetch/decode/execute cycle*. The steps in the fetch/decode/execute cycle are as follows:

Fetch	The CPU's control unit fetches, from main memory, the next instruction in the sequence of program instructions.
Decode	The instruction is encoded in the form of a number. The control unit decodes the instruction and generates an electronic signal.
Execute	The signal is routed to the appropriate component of the computer (such as the ALU, a disk drive, or some other device). The signal causes the component to perform an operation.

These steps are repeated as long as there are instructions to perform.

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 with which the program is working. 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 small chips.

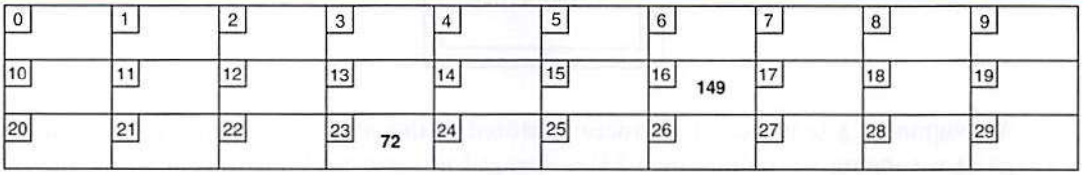
A computer's memory is divided into tiny storage locations known as bytes. One *byte* is enough memory to store only a letter of the alphabet or a small number. In order to do

anything meaningful, a computer must 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.

Each byte is assigned a unique number known as an *address*. The addresses are ordered from lowest to highest. A byte is identified by its address in much the same way a post office box is identified by an address. Figure 1-6 shows a group of memory cells with their addresses. In the illustration, sample data is stored in memory. The number 149 is stored in the cell with the address 16, and the number 72 is stored at address 23.

Figure 1-6 Memory



Secondary Storage

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 storage devices can be used to create backup copies of important data or to move data to another computer. For example, *USB (Universal Serial Bus) drives* and *SD (Secure Digital) memory cards* are small devices that appear in the system as disk drives. They 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.

Input Devices

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

Output Devices

Output is any information the computer sends to the outside world. It might be a sales report, a list of names, or a graphic image. The information is sent to an *output device*, which formats and presents it. Common output devices are screens, printers, and speakers. Storage devices can also be considered output devices because the CPU sends them data to be saved.

Software

If a computer is to function, software is not optional. Everything 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 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.
- *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*
  The software tools that programmers use to create, modify, and test software are referred to as *software development tools*. Compilers and integrated development environments, which we will discuss later in this chapter, 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, e-mail programs, web browsers, and game programs.

Checkpoint

1.1 Why is the computer used by so many different people, in so many different professions?

1.2 List the five major hardware components of a computer system.

1.3 Internally, the CPU consists of what two units?

1.4 Describe the steps in the fetch/decode/execute cycle.

1.5 What is a memory address? What is its purpose?

1.6 Explain why computers have both main memory and secondary storage.

1.7 What are the two general categories of software?

1.8 What fundamental set of programs control the internal operations of the computer's hardware?

1.9 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?

1.10 Word processing programs, spreadsheet programs, e-mail programs, web browsers, and game programs belong to what category of software?

1.3 Programs and Programming Languages

CONCEPT: **A program is a set of instructions a computer follows in order to perform a task. A programming language is a special language used to write computer programs.**

What Is a Program?

Computers are designed to follow instructions. A computer program is a set of instructions that tells the computer how to solve a problem or perform a task. For example, suppose we want the computer to calculate someone's gross pay. Here is a list of things the computer should do:

1. Display a message on the screen asking "How many hours did you work?"
2. Wait for the user to enter the number of hours worked. Once the user enters a number, store it in memory.
3. Display a message on the screen asking "How much do you get paid per hour?"
4. Wait for the user to enter an hourly pay rate. Once the user enters a number, store it in memory.
5. Multiply the number of hours by the amount paid per hour, and store the result in memory.
6. Display a message on the screen that tells the amount of money earned. The message must include the result of the calculation performed in Step 5.

Collectively, these instructions are called an *algorithm*. An algorithm is a set of well-defined steps for performing a task or solving a problem. Notice these steps are sequentially ordered. Step 1 should be performed before Step 2, and so forth. It is important that these instructions be performed in their proper sequence.

Although you and I might easily understand the instructions in the pay-calculating algorithm, it is not ready to be executed on a computer. A computer's CPU can only process instructions that are written in *machine language*. If you were to look at a machine language program, you would see a stream of *binary numbers* (numbers consisting of only 1s and 0s). The binary numbers form machine language instructions, which the CPU interprets as commands. Here is an example of what a machine language instruction might look like:

```
1011010000000101
```

As you can imagine, the process of encoding an algorithm in machine language is very tedious and difficult. In addition, each different type of CPU has its own machine language. If you wrote a machine language program for computer *A* then wanted to run it on computer *B*, which has a different type of CPU, you would have to rewrite the program in computer B's machine language.

Programming languages, which use words instead of numbers, were invented to ease the task of programming. A program can be written in a programming language, such as C++, which is much easier to understand than machine language. Programmers save their programs in text files, then use special software to convert their programs to machine language.

Program 1-1 shows how the pay-calculating algorithm might be written in C++.

The "Program Output with Example Input" shows what the program will display on the screen when it is running. In the example, the user enters 10 for the number of hours worked and 15 for the hourly pay rate. The program displays the earnings, which are $150.

NOTE: The line numbers that are shown in Program 1-1 are *not* part of the program. This book shows line numbers in all program listings to help point out specific parts of the program.

Program 1-1

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      double hours, rate, pay;
8
9      // Get the number of hours worked.
10     cout << "How many hours did you work? ";
11     cin >> hours;
12
13     // Get the hourly pay rate.
14     cout << "How much do you get paid per hour? ";
15     cin >> rate;
16
17     // Calculate the pay.
18     pay = hours * rate;
```

(program continues)

Program 1-1 *(continued)*

```
19
20       // Display the pay.
21       cout << "You have earned $" << pay << endl;
22       return 0;
23  }
```

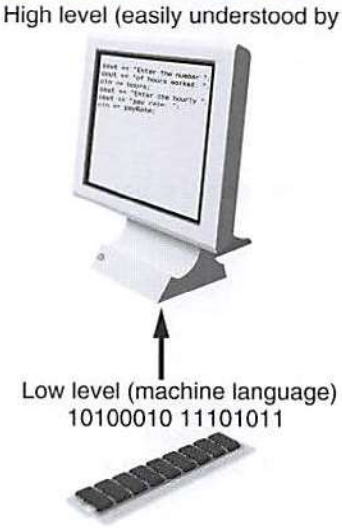
Program Output with Example Input Shown in Bold

```
How many hours did you work? 10 [Enter]
How much do you get paid per hour? 15 [Enter]
You have earned $150
```

Programming Languages

In a broad sense, there are two categories of programming languages: low-level and high-level. A low-level language is close to the level of the computer, which means it resembles the numeric machine language of the computer more than the natural language of humans. The easiest languages for people to learn are *high-level languages*. They are called "high-level" because they are closer to the level of human readability than computer readability. Figure 1-7 illustrates the concept of language levels.

Figure 1-7 Low-level versus high-level languages



Many high-level languages have been created. Table 1-1 lists a few of the well-known ones.

In addition to the high-level features necessary for writing applications such as payroll systems and inventory programs, C++ also has many low-level features. C++ is based on the C language, which was invented for purposes such as writing operating systems and compilers. Since C++ evolved from C, it carries all of C's low-level capabilities with it.

Table 1-1 Programming Languages

Language	Description
BASIC	Beginners All-purpose Symbolic Instruction Code. A general programming language originally designed to be simple enough for beginners to learn.
FORTRAN	Formula Translator. A language designed for programming complex mathematical algorithms.
COBOL	Common Business-Oriented Language. A language designed for business applications.
Pascal	A structured, general-purpose language designed primarily for teaching programming.
C	A structured, general-purpose language developed at Bell Laboratories. C offers both high-level and low-level features.
C++	Based on the C language, C++ offers object-oriented features not found in C. Also invented at Bell Laboratories.
C#	Pronounced "C sharp." A language invented by Microsoft for developing applications based on the Microsoft .NET platform.
Java	An object-oriented language that may be used to develop programs that run on many different types of devices.
JavaScript	JavaScript can be used to write small programs that run in webpages. 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 both 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	A Microsoft programming language and software development environment that allows programmers to quickly create Windows-based applications.

C++ is popular not only because of its mixture of low- and high-level features, but also because of its *portability*. This means that a C++ program can be written on one type of computer, then run on many other types of systems. This usually requires the program to be recompiled on each type of system, but the program itself may need little or no change.

NOTE: Programs written for specific graphical environments often require significant changes when moved to a different type of system. Examples of such graphical environments are Windows, the X-Window System, and the macOS operating system.

Source Code, Object Code, and Executable Code

When a C++ program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The statements written by the programmer are called *source code*, and the file they are saved in is called the *source file*.

After the source code is saved to a file, the process of translating it to machine language can begin. During the first phase of this process, a program called the *preprocessor* reads the source code. The preprocessor searches for special lines that begin with the # symbol. These lines contain commands that cause the preprocessor to modify the source code in some way.

During the next phase, the *compiler* steps through the preprocessed source code, translating each source code instruction into the appropriate machine language instruction. This process will uncover any *syntax errors* that may be in the program. Syntax errors are illegal uses of key words, operators, punctuation, and other language elements. If the program is free of syntax errors, the compiler stores the translated machine language instructions, which are called *object code*, in an *object file*.

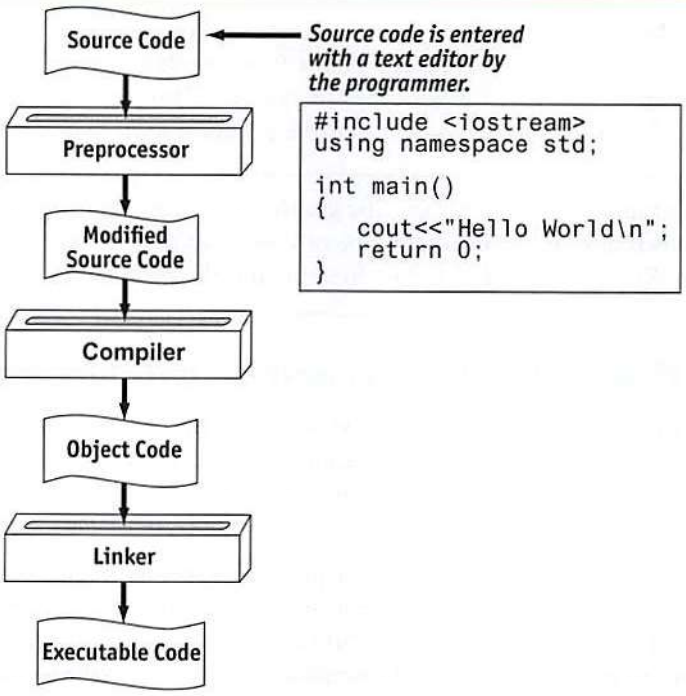
Although an object file contains machine language instructions, it is not a complete program. Here is why: C++ is conveniently equipped with a library of prewritten code for performing common operations or sometimes-difficult tasks. For example, the library contains hardware-specific code for displaying messages on the screen and reading input from the keyboard. It also provides routines for mathematical functions, such as calculating the square root of a number. This collection of code, called the *runtime library*, is extensive. Programs almost always use some part of it. When the compiler generates an object file, however, it does not include machine code for any runtime library routines the programmer might have used. During the last phase of the translation process, another program called the *linker* combines the object file with the necessary library routines. Once the linker has finished with this step, an *executable file* is created. The executable file contains machine language instructions, or *executable code*, and is ready to run on the computer.

Figure 1-8 illustrates the process of translating a C++ source file into an executable file.

The entire process of invoking the preprocessor, compiler, and linker can be initiated with a single action. For example, on a Linux system, the following command causes the C++ program named `hello.cpp` to be preprocessed, compiled, and linked. The executable code is stored in a file named `hello`.

```
g++ -o hello hello.cpp
```

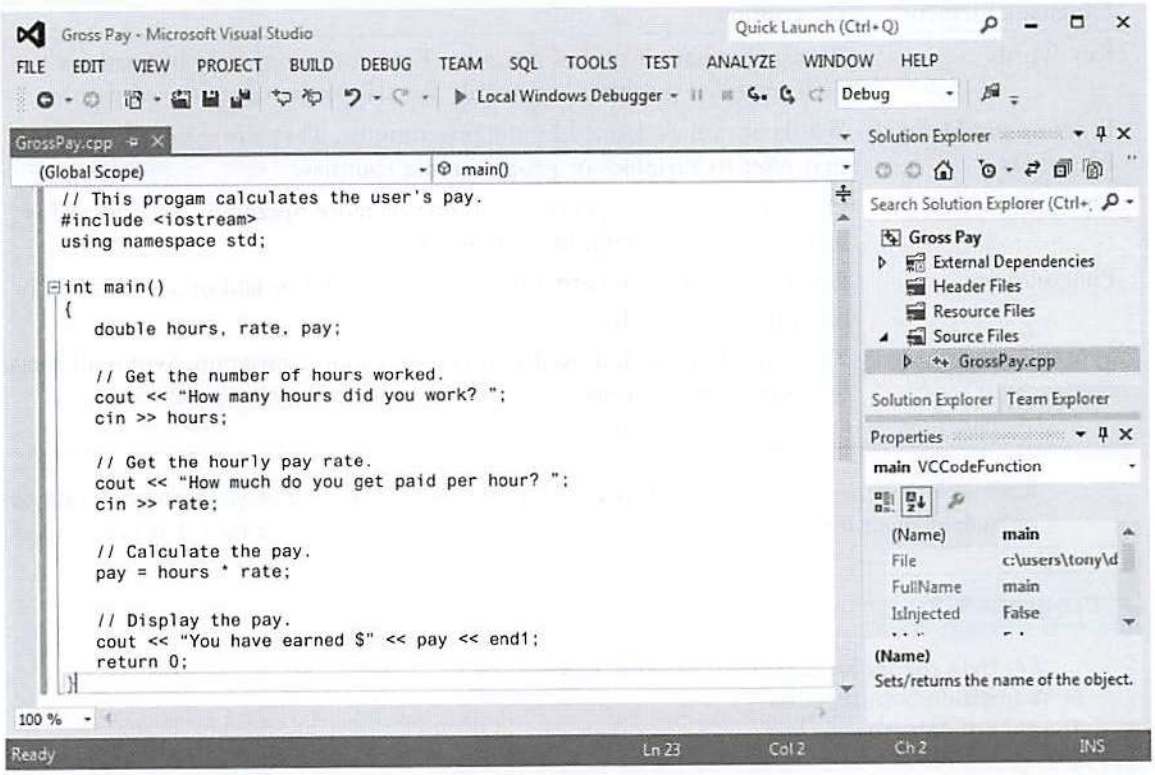
Figure 1-8 Translating a C++ source file to an executable file



Appendix F explains how compiling works in .Net. You can download Appendix F from the Computer Science Portal at www.pearsonhighered.com/gaddis.

Many development systems, particularly those on personal computers, have *integrated development environments (IDEs)*. These environments consist of a text editor, compiler, debugger, and other utilities integrated into a package with a single set of menus. Preprocessing, compiling, linking, and even executing a program is done with a single click of a button, or by selecting a single item from a menu. Figure 1-9 shows a screen from the Microsoft Visual Studio IDE.

Figure 1-9 An integrated development environment (IDE)



Checkpoint

1.11 What is an algorithm?

1.12 Why were computer programming languages invented?

1.13 What is the difference between a high-level language and a low-level language?

1.14 What does *portability* mean?

1.15 Explain the operations carried out by the preprocessor, compiler, and linker.

1.16 Explain what is stored in a source file, an object file, and an executable file.

1.17 What is an integrated development environment?

1.4 What Is a Program Made of?

CONCEPT: There are certain elements that are common to all programming languages.

Language Elements

All programming languages have a few things in common. Table 1-2 lists the common elements you will find in almost every language.

Table 1-2 Language Elements

Language Element	Description
Key Words	Words that have a special meaning. Key words may only be used for their intended purpose. Key words are also known as reserved words.
Programmer-Defined Identifiers	Words or names defined by the programmer. They are symbolic names that refer to variables or programming routines.
Operators	Operators perform operations on one or more operands. An operand is usually a piece of data, like a number.
Punctuation	Punctuation characters that mark the beginning or end of a statement, or separate items in a list.
Syntax	Rules that must be followed when constructing a program. Syntax dictates how key words and operators may be used, and where punctuation symbols must appear.

Let's look at some specific parts of Program 1-1 (the pay-calculating program) to see examples of each element listed in the table above. For your convenience, Program 1-1 is listed again.

Program 1-1

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      double hours, rate, pay;
8
9      // Get the number of hours worked.
10     cout << "How many hours did you work? ";
11     cin >> hours;
12
13     // Get the hourly pay rate.
14     cout << "How much do you get paid per hour? ";
15     cin >> rate;
16
17     // Calculate the pay.
18     pay = hours * rate;
```

```
19
20      // Display the pay.
21      cout << "You have earned $" << pay << endl;
22      return 0;
23  }
```

Key Words (Reserved Words)

Three of C++'s key words appear on lines 3 and 5: `using`, `namespace`, and `int`. The word `double`, which appears on line 7, is also a C++ key word. These words, which are always written in lowercase, each have a special meaning in C++ and can only be used for their intended purposes. As you will see, the programmer is allowed to make up his or her own names for certain things in a program. Key words, however, are reserved and cannot be used for anything other than their designated purposes. Part of learning a programming language is learning what the key words are, what they mean, and how to use them.

NOTE: The `#include <iostream>` statement in line 2 is a preprocessor directive.

NOTE: In C++, key words are written in all lowercase.

Programmer-Defined Identifiers

The words `hours`, `rate`, and `pay` that appear in the program on lines 7, 11, 15, 18, and 21 are programmer-defined identifiers. They are not part of the C++ language, but rather are names made up by the programmer. In this particular program, these are the names of variables. As you will learn later in this chapter, variables are the names of memory locations that may hold data.

Operators

On line 18 the following code appears:

```
pay = hours * rate;
```

The = and * symbols are both operators. They perform operations on pieces of data known as operands. The * operator multiplies its two operands, which in this example are the variables `hours` and `rate`. The = symbol is called the assignment operator. It takes the value of the expression on the right and stores it in the variable whose name appears on the left. In this example, the = operator stores in the `pay` variable the result of the `hours` variable multiplied by the `rate` variable. In other words, the statement says, "Make the `pay` variable equal to `hours` times `rate`, or "`pay` is assigned the value of `hours` times `rate`."

Punctuation

Notice lines 3, 7, 10, 11, 14, 15, 18, 21, and 22 all end with a semicolon. A semicolon in C++ is similar to a period in English: It marks the end of a complete sentence (or statement, as it is called in programming jargon). Semicolons do not appear at the end of every line in a C++ program, however. There are rules that govern where semicolons are required and

where they are not. Part of learning C++ is learning where to place semicolons and other punctuation symbols.

Lines and Statements

Often, the contents of a program are thought of in terms of lines and statements. A "line" is just that—a single line as it appears in the body of a program. Program 1-1 has 23 lines. Most of the lines contain something meaningful; however, some of the lines are empty. The blank lines are only there to make the program more readable.

A statement is a complete instruction that causes the computer to perform some action. Here is the statement that appears in line 10 of Program 1-1:

```
cout << "How many hours did you work? ";
```

This statement causes the computer to display the message "How many hours did you work?" on the screen. Statements can be a combination of key words, operators, and programmer-defined symbols. Statements often occupy only one line in a program, but sometimes they are spread out over more than one line.

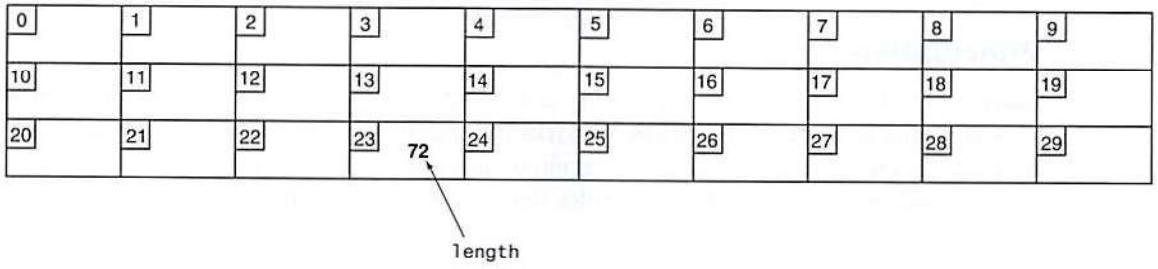
Variables

A variable is a named storage location in the computer's memory for holding a piece of information. The information stored in variables may change while the program is running (hence the name "variable"). Notice in Program 1-1 the words `hours`, `rate`, and `pay` appear in several places. All three of these are the names of variables. The `hours` variable is used to store the number of hours the user has worked. The `rate` variable stores the user's hourly pay rate. The `pay` variable holds the result of `hours` multiplied by `rate`, which is the user's gross pay.

NOTE: Notice the variables in Program 1-1 have names that reflect their purpose. In fact, it would be easy to guess what the variables were used for just by reading their names. This will be discussed further in Chapter 2.

Variables are symbolic names that represent locations in the computer's random-access memory (RAM). When information is stored in a variable, it is actually stored in RAM. Assume a program has a variable named `length`. Figure 1-10 illustrates the way the variable name represents a memory location.

Figure 1-10 A variable name represents a memory location



In Figure 1-10, the variable `length` is holding the value 72. The number 72 is actually stored in RAM at address 23, but the name `length` symbolically represents this storage location. If it helps, you can think of a variable as a box that holds information. In Figure 1-10, the number 72 is stored in the box named `length`. Only one item may be stored in the box at any given time. If the program stores another value in the box, it will take the place of the number 72.

Variable Definitions

In programming, there are two general types of data: numbers and characters. Numbers are used to perform mathematical operations, and characters are used to print data on the screen or on paper.

Numeric data can be categorized even further. For instance, the following are all whole numbers or integers:

```
5
7
-129
32154
```

The following are real or floating-point numbers:

```
3.14159
6.7
1.0002
```

When creating a variable in a C++ program, you must know what type of data the program will be storing in it. Look at line 7 of Program 1-1:

```
double hours, rate, pay;
```

The word `double` in this statement indicates that the variables `hours`, `rate`, and `pay` will be used to hold double precision floating-point numbers. This statement is called a *variable definition*. It is used to *define* one or more variables that will be used in the program and to indicate the type of data they will hold. The variable definition causes the variables to be created in memory, so all variables must be defined before they can be used. If you review the listing of Program 1-1, you will see that the variable definitions come before any other statements using those variables.

NOTE: Programmers often use the term "variable declaration" to mean the same thing as "variable definition." Strictly speaking, there is a difference between the two terms. A definition statement always causes a variable to be created in memory. Some types of declaration statements, however, do not cause a variable to be created in memory. You will learn more about declarations later in this book.

1.5 Input, Processing, and Output

CONCEPT: The three primary activities of a program are input, processing, and output.

Computer programs typically perform a three-step process of gathering input, performing some process on the information gathered, then producing output. Input is information a

program collects from the outside world. It can be sent to the program from the user, who is entering data at the keyboard or using the mouse. It can also be read from disk files or hardware devices connected to the computer. Program 1-1 allows the user to enter two pieces of information: the number of hours worked and the hourly pay rate. Lines 11 and 15 use the `cin` (pronounced "see in") object to perform these input operations:

```
cin >> hours;
cin >> rate;
```

Once information is gathered from the outside world, a program usually processes it in some manner. In Program 1-1, the hours worked and hourly pay rate are multiplied in line 18, and the result is assigned to the `pay` variable:

```
pay = hours * rate;
```

Output is information that a program sends to the outside world. It can be words or graphics displayed on a screen, a report sent to the printer, data stored in a file, or information sent to any device connected to the computer. Lines 10, 14, and 21 in Program 1-1 all perform output:

```
cout << "How many hours did you work? ";
cout << "How much do you get paid per hour? ";
cout << "You have earned $" << pay << endl;
```

These lines use the `cout` (pronounced "see out") object to display messages on the computer's screen. You will learn more details about the `cin` and `cout` objects in Chapter 2.

Checkpoint

1.18 Describe the difference between a key word and a programmer-defined identifier.

1.19 Describe the difference between operators and punctuation symbols.

1.20 Describe the difference between a program line and a statement.

1.21 Why are variables called "variable"?

1.22 What happens to a variable's current contents when a new value is stored there?

1.23 What must take place in a program before a variable is used?

1.24 What are the three primary activities of a program?

1.6 The Programming Process

CONCEPT: **The programming process consists of several steps, which include design, creation, testing, and debugging activities.**

Designing and Creating a Program

Now that you have been introduced to what a program is, it's time to consider the process of creating a program. Quite often, when inexperienced students are given programming assignments, they have trouble getting started because they don't know what to do first. If you find yourself in this dilemma, the steps listed in Figure 1-11 may help. These are the steps recommended for the process of writing a program.

Figure 1-11 Steps for writing a program

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools such as a hierarchy chart, flowcharts, or pseudocode to create a model of the program.
4. Check the model for logical errors.
5. Type the code, save it, and compile it.
6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.

The steps listed in Figure 1-11 emphasize the importance of planning. Just as there are good ways and bad ways to paint a house, there are good ways and bad ways to create a program. A good program always begins with planning.

With the pay-calculating program as our example, let's look at each of the steps in more detail.

1. **Clearly define what the program is to do.**

This step requires that you identify the purpose of the program, the information that is to be input, the processing that is to take place, and the desired output. Let's examine each of these requirements for the example program:

Purpose	To calculate the user's gross pay.
Input	Number of hours worked, hourly pay rate.
Process	Multiply number of hours worked by hourly pay rate. The result is the user's gross pay.
Output	Display a message indicating the user's gross pay.

2. **Visualize the program running on the computer.**

Before you create a program on the computer, you should first create it in your mind. Step 2 is the visualization of the program. Try to imagine what the computer screen looks like while the program is running. If it helps, draw pictures of the screen, with sample input and output, at various points in the program. For instance, here is the screen produced by the pay-calculating program:

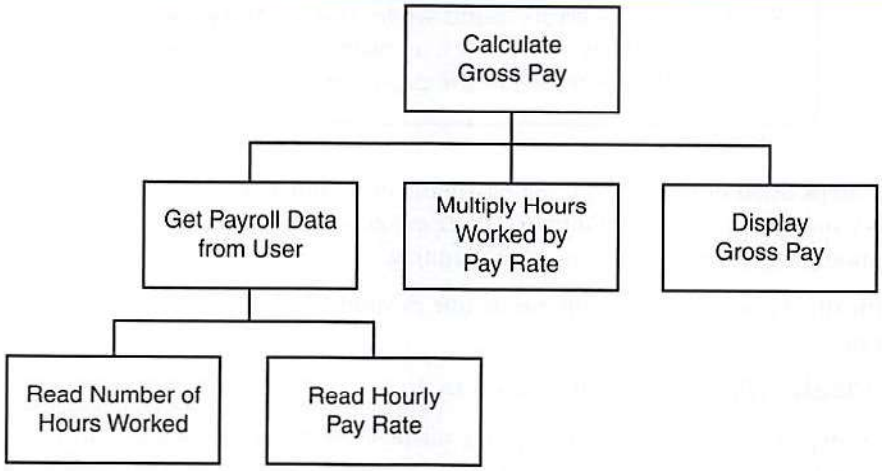
```
How many hours did you work? 10
How much do you get paid per hour? 15
You have earned $150
```

In this step, you must put yourself in the shoes of the user. What messages should the program display? What questions should it ask? By addressing these concerns, you will have already determined most of the program's output.

3. **Use design tools such as a hierarchy chart, flowcharts, or pseudocode to create a model of the program.**

While planning a program, the programmer uses one or more design tools to create a model of the program. Three common design tools are hierarchy charts, flowcharts, and pseudocode. A *hierarchy chart* is a diagram that graphically depicts the structure of a program. It has boxes that represent each step in the program. The boxes are connected in a way that illustrates their relationship to one another. Figure 1-12 shows a hierarchy chart for the pay-calculating program.

Figure 1-12 Hierarchy chart



A hierarchy chart begins with the overall task then refines it into smaller subtasks. Each of the subtasks is then refined into even smaller sets of subtasks, until each is small enough to be easily performed. For instance, in Figure 1-12, the overall task "Calculate Gross Pay" is listed in the top-level box. That task is broken into three subtasks. The first subtask, "Get Payroll Data from User," is broken further into two subtasks. This process of "divide and conquer" is known as *top-down design*.

VideoNote
Introduction to Flowcharting

A *flowchart* is a diagram that shows the logical flow of a program. It is a useful tool for planning each operation a program performs and the order in which the operations are to occur. For more information see Appendix C, Introduction to Flowcharting.

Pseudocode is a cross between human language and a programming language. Although the computer can't understand pseudocode, programmers often find it helpful to write an algorithm in a language that's "almost" a programming language, but still very similar to natural language. For example, here is pseudocode that describes the pay-calculating program:

VideoNote
Designing a Program with Pseudocode

Get payroll data.
Calculate gross pay.
Display gross pay.

Although the pseudocode above gives a broad view of the program, it doesn't reveal all the program's details. A more detailed version of the pseudocode follows:

Display "How many hours did you work?".
Input hours.
Display "How much do you get paid per hour?".
Input rate.
Store the value of hours times rate in the pay variable.
Display the value in the pay variable.

Notice the pseudocode contains statements that look more like commands than the English statements that describe the algorithm in Section 1.4 (What Is a Program Made of?). The pseudocode even names variables and describes mathematical operations.

4. **Check the model for logical errors.**

Logical errors are mistakes that cause the program to produce erroneous results. Once a hierarchy chart, flowchart, or pseudocode model of the program is assembled, it should be checked for these errors. The programmer should trace through the charts or pseudocode, checking the logic of each step. If an error is found, the model can be corrected before the next step is attempted.

5. **Type the code, save it, and compile it.**

Once a model of the program (hierarchy chart, flowchart, or pseudocode) has been created, checked, and corrected, the programmer is ready to write source code on the computer. The programmer saves the source code to a file and begins the process of translating it to machine language. During this step, the compiler will find any syntax errors that may exist in the program.

6. **Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.**

If the compiler reports any errors, they must be corrected. Steps 5 and 6 must be repeated until the program is free of compile-time errors.

7. **Run the program with test data for input.**

Once an executable file is generated, the program is ready to be tested for runtime errors. A runtime error is an error that occurs while the program is running. These are usually logical errors, such as mathematical mistakes.

Testing for runtime errors requires that the program be executed with sample data or sample input. The sample data should be such that the correct output can be predicted. If the program does not produce the correct output, a logical error is present in the program.

8. **Correct any errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.**

When runtime errors are found in a program, they must be corrected. You must identify the step where the error occurred and determine the cause. Desk-checking is a process that can help locate runtime errors. The term *desk-checking* means the programmer starts reading the program, or a portion of the program, and steps through each statement. A sheet of paper is often used in this process to jot down the current contents of all variables and sketch what the screen looks like after each output operation. When a variable's contents change, or information is displayed on the screen, this is noted. By stepping through each statement, many errors can be located and corrected. If an error is a result of incorrect logic (such as an improperly stated math formula), you must correct the statement or statements involved in

the logic. If an error is due to an incomplete understanding of the program requirements, then you must restate the program purpose and modify the hierarchy and/or flowcharts, pseudocode, and source code. The program must then be saved, recompiled, and retested. This means Steps 5 through 8 must be repeated until the program reliably produces satisfactory results.

9. **Validate the results of the program.**

When you believe you have corrected all the runtime errors, enter test data and determine whether the program solves the original problem.

What Is Software Engineering?

The field of software engineering encompasses the whole process of crafting computer software. It includes designing, writing, testing, debugging, documenting, modifying, and maintaining complex software development projects. Like traditional engineers, software engineers use a number of tools in their craft. Here are a few examples:

- Program specifications
- Charts and diagrams of screen output
- Hierarchy charts and flowcharts
- Pseudocode
- Examples of expected input and desired output
- Special software designed for testing programs

Most commercial software applications are very large. In many instances, one or more teams of programmers, not a single individual, develop them. It is important that the program requirements be thoroughly analyzed and divided into subtasks that are handled by individual teams, or individuals within a team.

In Step 3 of the programming process, you were introduced to the hierarchy chart as a tool for top-down design. The subtasks identified in a top-down design can easily become modules, or separate components of a program. If the program is very large or complex, a team of software engineers can be assigned to work on the individual modules. As the project develops, the modules are coordinated to finally become a single software application.

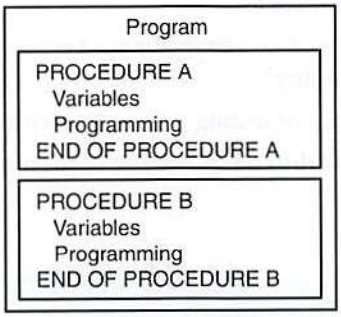
1.7 Procedural and Object-Oriented Programming

CONCEPT: **Procedural programming and object-oriented programming are two ways of thinking about software development and program design.**

C++ is a language that can be used for two methods of writing computer programs: *procedural programming* and *object-oriented programming*. This book is designed to teach you some of both.

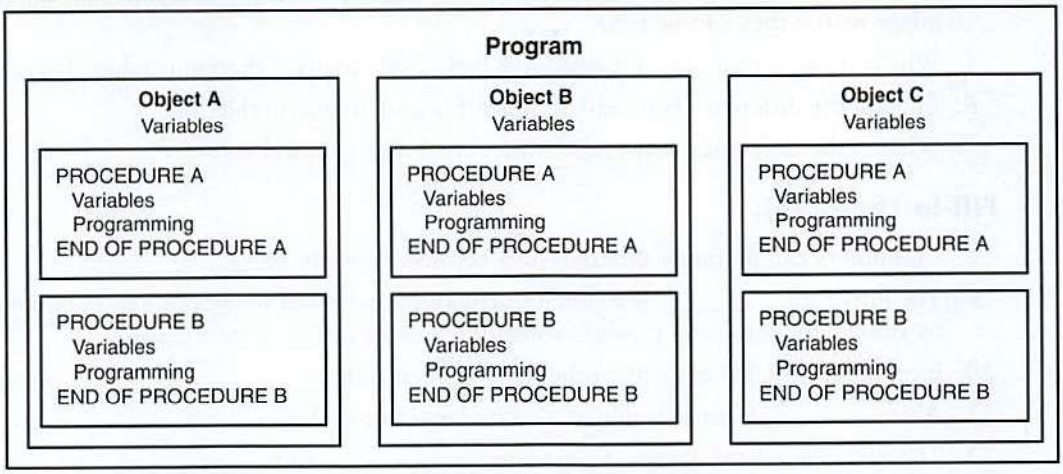
In procedural programming, the programmer constructs procedures (or functions, as they are called in C++). The procedures are collections of programming statements that perform a specific task. The procedures each contain their own variables and commonly share variables with other procedures. This is illustrated in Figure 1-13.

Figure 1-13 Procedures in a program



Procedural programming is centered on the procedure, or function. Object-oriented programming (OOP), on the other hand, is centered on the object. An object is a programming element that contains data and the procedures that operate on the data. It is a self-contained unit. This is illustrated in Figure 1-14.

Figure 1-14 Objects in a program



The objects contain, within themselves, both information and the ability to manipulate the information. Operations are carried out on the information in an object by sending the object a *message*. When an object receives a message instructing it to perform some operation, it carries out the instruction. As you study this text, you will encounter many other aspects of object-oriented programming.

Checkpoint

1.25 What four items should you identify when defining what a program is to do?

1.26 What does it mean to "visualize a program running"? What is the value of such an activity?

1.27 What is a hierarchy chart?

1.28 Describe the process of desk-checking.

1.29 Describe what a compiler does with a program's source code.

1.30 What is a runtime error?

1.31 Is a syntax error (such as misspelling a key word) found by the compiler or when the program is running?

1.32 What is the purpose of testing a program with sample data or input?

1.33 Briefly describe the difference between procedural and object-oriented programming.

Review Questions and Exercises

Short Answer

1. Both main memory and secondary storage are types of memory. Describe the difference between the two.
2. What is the difference between system software and application software?
3. What type of software controls the internal operations of the computer's hardware?
4. Why must programs written in a high-level language be translated into machine language before they can be run?
5. Why is it easier to write a program in a high-level language than in machine language?
6. Explain the difference between an object file and an executable file.
7. What is the difference between a syntax error and a logical error?

Fill-in-the-Blank

8. Computers can do many different jobs because they can be _____.
9. The job of the _____ is to fetch instructions, carry out the operations commanded by the instructions, and produce some outcome or resultant information.
10. Internally, the CPU consists of the _____ and the _____.
11. A(n) _____ is an example of a secondary storage device.
12. The two general categories of software are _____ and _____.
13. A program is a set of _____.
14. Since computers can't be programmed in natural human language, algorithms must be written in a(n) _____ language.
15. _____ is the only language computers really process.
16. _____ languages are close to the level of humans in terms of readability.
17. _____ languages are close to the level of the computer.
18. A program's ability to run on several different types of computer systems is called _____.
19. Words that have special meaning in a programming language are called _____.
20. Words or names defined by the programmer are called _____.
21. _____ are characters or symbols that perform operations on one or more operands.

22. _____ characters or symbols mark the beginning or end of programming statements, or separate items in a list.
23. The rules that must be followed when constructing a program are called _____.
24. A(n) _____ is a named storage location.
25. A variable must be _____ before it can be used in a program.
26. The three primary activities of a program are _____, _____, and _____.
27. _____ is information a program gathers from the outside world.
28. _____ is information a program sends to the outside world.
29. A(n) _____ is a diagram that graphically illustrates the structure of a program.

Algorithm Workbench

Draw hierarchy charts or flowcharts that depict the programs described below. (See Appendix C for instructions on creating flowcharts.)

30. Available Credit

    The following steps should be followed in a program that calculates a customer's available credit:

    1. Display the message "Enter the customer's maximum credit."
    2. Wait for the user to enter the customer's maximum credit.
    3. Display the message "Enter the amount of credit used by the customer."
    4. Wait for the user to enter the customer's credit used.
    5. Subtract the used credit from the maximum credit to get the customer's available credit.
    6. Display a message that shows the customer's available credit.

31. Sales Tax

    Design a hierarchy chart or flowchart for a program that calculates the total of a retail sale. The program should ask the user for:
    - The retail price of the item being purchased
    - The sales tax rate

    Once these items have been entered, the program should calculate and display:
    - The sales tax for the purchase
    - The total of the sale

32. Account Balance

    
    VideoNote
    Designing the Account Balance Program

    Design a hierarchy chart or flowchart for a program that calculates the current balance in a savings account. The program must ask the user for:
    - The starting balance
    - The total dollar amount of deposits made
    - The total dollar amount of withdrawals made
    - The monthly interest rate

    Once the program calculates the current balance, it should be displayed on the screen.

Predict the Result

Questions 33–35 are programs expressed as English statements. What would each display on the screen if they were actual programs?

VideoNote
Predicting the Result of Problem 33

33. The variable x starts with the value 0.
    The variable y starts with the value 5.
    Add 1 to x.
    Add 1 to y.
    Add x and y, and store the result in y.
    Display the value in y on the screen.

34. The variable j starts with the value 10.
    The variable k starts with the value 2.
    The variable l starts with the value 4.
    Store the value of j times k in j.
    Store the value of k times l in l.
    Add j and l, and store the result in k.
    Display the value in k on the screen.

35. The variable a starts with the value 1.
    The variable b starts with the value 10.
    The variable c starts with the value 100.
    The variable x starts with the value 0.
    Store the value of c times 3 in x.
    Add the value of b times 6 to the value already in x.
    Add the value of a times 5 to the value already in x.
    Display the value in x on the screen.

Find the Error

36. The following *pseudocode algorithm* has an error. The program is supposed to ask the user for the length and width of a rectangular room, then display the room's area. The program must multiply the width by the length in order to determine the area. Find the error.

    area = width × length.
    Display "What is the room's width?".
    Input width.
    Display "What is the room's length?".
    Input length.
    Display area.