chapter 1

Overview of Programming and Problem Solving

Goals

- To be able to define *computer program*, *algorithm*, and *high-level programming language*.
- To be able to list the basic stages involved in writing a computer program.
- To be able to distinguish between machine code and Bytecode.
- To be able to describe what compilers and interpreters are and what they do.
- To be able to describe the compilation, execution, and interpretation processes.
- To be able to list the major components of a computer and describe how they work together.
- To be able to distinguish between hardware and software.
- To be able to discuss some of the basic ethical issues confronting computing professionals.
- To be able to apply an appropriate problem-solving method for developing an algorithmic solution to a problem.

1.1 Overview of Programming

com·put·er \kəm-'pyüt-ər\ *n. often attrib* (1646): one that computes; *specif*: a programmable electronic device that can store, retrieve, and process data*

What a brief definition for something that has, in just a few decades, changed the way of life in industrialized societies! Computers touch all areas of our lives: paying bills, driving cars, using the telephone, shopping. In fact, it might be easier to list those areas of our lives in which we do *not* use computers. You are probably most familiar with computers through the use of games, word processors, Web browsers, and other programs. Be forewarned: This book is not just about using computers. This is a text to teach you how to program them.

What is Programming?

Much of human behavior and thought is characterized by logical sequences of actions applied to objects. Since infancy, you have been learning how to act, how to do things; and you have learned to expect certain behavior from other people.

A lot of what you do every day you do automatically. Fortunately, it is not necessary for you to consciously think of every step involved in a process as simple as turning a page by hand:

1. Lift hand.
2. Move hand to right side of book.
3. Grasp top-right corner of page.
4. Move hand from right to left until page is positioned so that you can read what is on the other side.
5. Let go of page.

Think how many neurons must fire and how many muscles must respond, all in a certain order or sequence, to move your arm and hand. Yet you do it unconsciously.

Much of what you do unconsciously you once had to learn. Watch how a baby concentrates on putting one foot before the other while learning to walk. Then watch a group of three-year-olds playing tag.

On a broader scale, mathematics never could have been developed without logical sequences of steps for manipulating symbols to solve problems and prove theorems. Mass production would never have worked without operations taking place on component parts in a certain order. Our whole civilization is based on the order of things and actions.

---

*By permission. From *Merriam-Webster's Collegiate Dictionary*, Tenth Edition © 1994 by Merriam-Webster Inc.

We create order, both consciously and unconsciously, through a process called programming. This book is concerned with the programming of one tool in particular, the electronic computer.

Programming Planning or scheduling the performance of a task or an event

Electronic computer A programmable device that can store, retrieve, and process data

Data Information in a form a computer can use

Information Any knowledge that can be communicated

Data type The specification of how information is represented in the computer as data and the set of operations that can be applied to it

Computer programming The process of specifying the data types and the operations for a computer to apply to data in order to solve a problem

Computer program Data type specifications and instructions for carrying out operations that are used by a computer to solve a problem

Notice that the key word in the definition of computer is *data*. Computers manipulate data. When you write a program (a plan) for a computer, you specify the properties of the data and the operations that can be applied to it. Those operations are then combined as necessary to solve a problem. Data is information in a form the computer can use—for example, numbers and letters. Information is any knowledge that can be communicated, including abstract ideas and concepts such as "the Earth is round."

Data comes in many different forms: letters, words, integer numbers, real numbers, dates, times, coordinates on a map, and so on. Virtually any kind of information can be represented as data, or as a combination of data and operations on it. Each kind of data in the computer is said to have a specific data type. For example, if we say that two data items are of type `Integer`, we know now they are represented in memory and that we can apply arithmetic operations to them.

Just as a concert program lists the pieces to be performed and the order in which the players perform them, a computer program lists the types of data that are to be used and the sequence of steps the computer performs on them. From now on, when we use the words *programming* and *program*, we mean computer programming and computer program.

The computer allows us to do tasks more efficiently, quickly, and accurately than we could by hand—if we could do them by hand at all. In order for this powerful machine to be a useful tool, it must be programmed. That is, we must specify what we want done and how. We do this through programming.

How Do We Write a Program

A computer is not intelligent. It cannot analyze a problem and come up with a solution. A human (the *programmer*) must analyze the problem, develop the instructions for solving the problem, and then have the computer carry out the instructions. What's the advantage of using a computer if it cannot solve problems? Once we have written a solution for the computer, the computer can repeat the solution very quickly and consistently, again and again. The computer frees people from repetitive and boring tasks.

To write a program for a computer to follow, we must go through a two-phase process: *problem solving* and *implementation* (see Figure 1.1).

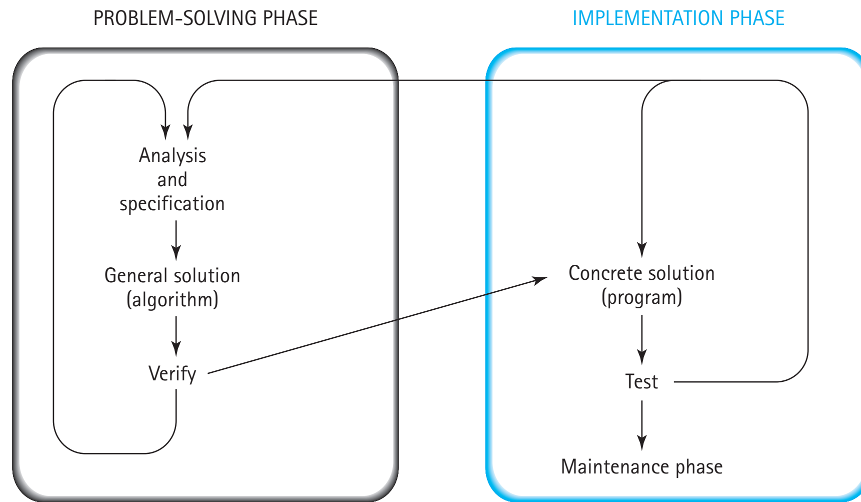


Figure 1.1 *Programming process*

Problem-Solving Phase

1. *Analysis and Specification.* Understand (define) the problem and what the solution must do.
2. *General Solution (Algorithm).* Specify the required data types and the logical sequences of steps that solve the problem.
3. *Verify.* Follow the steps exactly to see if the solution really does solve the problem.

Implementation Phase

1. *Concrete Solution (Program).* Translate the algorithm (the general solution) into a programming language.
2. *Test.* Have the computer follow the instructions. Then manually check the results. If you find errors, analyze the program and the algorithm to determine the source of the errors, and then make corrections.

Once a program has been written, it enters a third phase: *maintenance.*

Maintenance Phase

1. *Use.* Use the program.
2. *Maintain.* Modify the program to meet changing requirements or to correct any errors that show up while using it.

The programmer begins the programming process by analyzing the problem, breaking it into manageable pieces, and developing a general solution for each piece called an

algorithm. The solutions to the pieces are collected together to form a program that solves the original problem. Understanding and analyzing a problem take up much more time than Figure 1.1 implies. They are the heart of the programming process.

Algorithm Instructions for solving a problem or subproblem in a finite amount of time using a finite amount of data

If our definitions of a computer program and an algorithm look similar, it is because a program is simply an algorithm that has been written for a computer.

An algorithm is a written or verbal description of a logical sequence of actions applied to objects. We use algorithms every day. Recipes, instructions, and directions are all examples of algorithms that are not programs.

When you start your car, you follow a step-by-step procedure. The algorithm might look something like this:

1. Insert the key.
2. Make sure the transmission is in Park (or Neutral).
3. Turn the key to the start position.
4. If the engine starts within six seconds, release the key to the ignition position.
5. If the engine doesn't start in six seconds, release the key and gas pedal, wait ten seconds, and repeat Steps 3 through 5, but not more than five times.
6. If the car doesn't start, call the garage.

Without the phrase "but not more than five times" in Step 5, you could be trying to start the car forever. Why? Because if something is wrong with the car, repeating Steps 3 through 5 over and over will not start it. This kind of never-ending situation is called an *infinite loop*. If we leave the phrase "but not more than five times" out of Step 5, the procedure doesn't fit our definition of an algorithm. An algorithm must terminate in a finite amount of time for all possible conditions.

Suppose a programmer needs an algorithm to determine an employee's weekly wages. The algorithm reflects what would be done by hand:

1. Look up the employee's pay rate.
2. Determine the hours worked during the week.
3. If the number of hours worked is less than or equal to 40, multiply the hours by the pay rate to calculate regular wages.
4. If the number of hours worked is greater than 40, multiply 40 by the pay rate to calculate regular wages, and then multiply the difference between the hours worked and 40 by $1\frac{1}{2}$ times the pay rate to calculate overtime wages.
5. Add the regular wages to the overtime wages (if any) to determine total wages for the week.

The steps the computer follows are often the same steps you would use to do the calculations by hand.

After developing a general solution, the programmer tests the algorithm, "walking through" each step manually with paper and pencil. If the algorithm doesn't work, the programmer repeats the problem-solving process, analyzing the problem again and coming up with another algorithm. Often the second algorithm is just a variation of the first. When the programmer is satisfied with the algorithm, he or she translates it into a programming language. We use the Visual Basic programming language in this book.

A programming language is a simplified form of English (with math symbols) that adheres to a strict set of grammatical rules. English is far too complicated and ambiguous for today's computers to follow. Programming languages, because they limit vocabulary and grammar, are much simpler.

Programming language A set of rules, symbols, and special words used to construct a computer program

Code Data type specifications and instructions for a computer that are written in a programming language

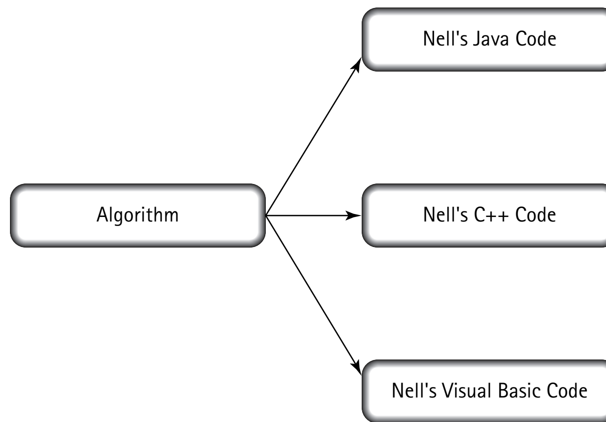
Although a programming language is simple in form, it is not always easy to use. Try giving someone directions to the nearest airport using a limited vocabulary of no more than 25 words, and you begin to see the problem. Programming forces you to write very simple, exact instructions.

Translating an algorithm into a programming language is called *coding* the algorithm. The products of the translation—the code for all the algorithms in the problem—are tested by collecting them into a program and running (*executing*) the program on the computer. If the program fails to produce the desired results, the programmer must debug it—that is, determine what is wrong and then modify the program, or even one or more of the algorithms, to fix it. The combination of coding and testing the algorithms is called *implementation*.

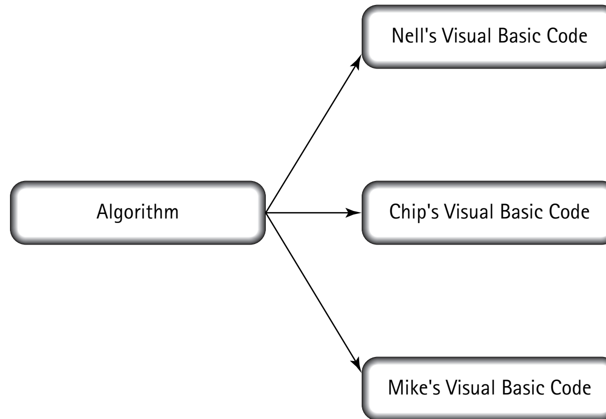
Code is the product of translating an algorithm into a programming language. The term *code* can refer to a complete program or to any portion of a program.

There is no single way to implement an algorithm. For example, an algorithm can be translated into more than one programming language. Each translation produces a different implementation (see Figure 1.2a). Even when two people translate an algorithm into the same programming language, they are likely to come up with different implementations (see Figure 1.2b). Why? Because every programming language allows the programmer some flexibility in how an algorithm is translated. Given this flexibility, people adopt their own *styles* in writing programs, just as they do in writing short stories or essays. Once you have some programming experience, you develop a style of your own. Throughout this book, we offer tips on good programming style.

Some people try to speed up the programming process by going directly from the problem definition to coding the program (see Figure 1.3). A shortcut here is very tempting and at first seems to save a lot of time. However, for many reasons that become obvious to you as you read this book, this kind of shortcut actually takes more time and effort. Developing a general solution before you write a program helps you manage the problem, keep your thoughts straight, and avoid mistakes. If you don't take the time at the beginning to think out and polish your algorithm, you spend a lot of extra time debugging and revising your program. So *think first and*



a. Algorithm translated into different languages



b. Algorithm translated by different people

Figure 1.2 *Differences in implementation*

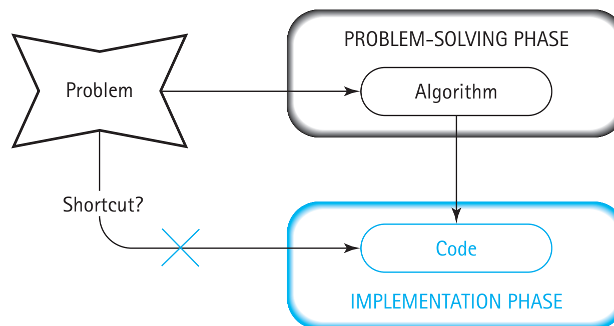


Figure 1.3 *Programming shortcut?*

code later! The sooner you start coding, the longer it takes to write a program that works.

Once a program has been put into use, it is often necessary to modify it. Modification may involve fixing an error that is discovered during the use of the program or changing the program in response to changes in the user's requirements. Each time the program is modified, it is necessary to repeat the problem-solving and implementation phases for those aspects of the program that change. This phase of the programming process is known as *maintenance* and actually accounts for the majority of the effort expended on most programs. For example, a program that is implemented in a few months may need to be maintained over a period of many years. Thus it is a cost-effective investment of time to carefully develop the initial problem solution and program implementation. Together, the problem-solving, implementation, and maintenance phases constitute the program's *life cycle*.

Documentation The written text and comments that make a program easier for others to understand, use, and modify

In addition to solving the problem, implementing the algorithm, and maintaining the program, writing documentation is an important part of the programming process. Documentation includes written explanations of the problem being solved and the organization of the solution, comments embedded within the program itself, and user manuals that describe how to use the program. Many different people are likely to work on a program over a long period of time. Each of those people must be able to read and understand the code.

Theoretical Foundations

Binary Representation of Data

In a computer, data is represented electronically by pulses of electricity. Electric circuits, in their simplest form, are either on or off. Usually, a circuit that is on represents the number 1; a circuit that is off represents the number 0. Any kind of data can be represented by combinations of enough 1s and 0s. We simply have to choose which combination represents each piece of data we are using. For example, we could arbitrarily choose the pattern 1101000110 to represent the word *Basic*.

Data is represented by 1s and 0s in binary form. The binary (base–2) number system uses only 1s and 0s to represent numbers. (The decimal (base–10) number system uses the digits 0 through 9.) The word *bit* (short for binary digit) often is used to refer to a single 1 or 0. So the pattern 1101000110 has 10 bits. A binary number with 10 bits can represent 2^{10} (1,024) different patterns. A byte is a group of eight bits; it can represent 2^8 (256) patterns. Inside the computer, each character (such as the letter A, the letter g, or a question mark) is usually represented by a byte.[1] Groups of 16, 32, and 64 bits are generally referred to as words.

(continued)

Binary Representation of Data

The process of assigning bit patterns to pieces of data is called coding—the same name we give to the process of translating an algorithm into a programming language. The names are the same because the only language that the first computers recognized was binary in form. Thus, in the early days of computers, programming meant translating both data and algorithms into patterns of 1s and 0s.

Binary coding schemes are still used inside the computer to represent both the instructions that it follows and the data that it uses. For example, 16 bits can represent the decimal integers from 0 to $2^{16} - 1$ (65,535). More complicated coding schemes are necessary to represent negative numbers, real numbers, and numbers in scientific notation. Characters can be represented by bit combinations, in one coding scheme, 01001101 represents *M* and 01101101 represents *m*.

The patterns of bits that represent data vary from one family of computers to another. Even on the same computer, different programming languages can use different binary representations for the same data. A single programming language may even use the same pattern of bits to represent different things in different contexts. (People do this too. The four letters that form the word *tack* have different meanings depending on whether you are talking about upholstery, sailing, sewing, paint, or horseback riding.) The point is that patterns of bits by themselves are meaningless. It is the way in which the patterns are used that gives them their meaning.

Fortunately, we no longer have to work with binary coding schemes. Today the process of coding is usually just a matter of writing down the data in letters, numbers, and symbols. The computer automatically converts these letters, numbers, and symbols into binary form. Still, as you work with computers, you continually run into numbers that are related to powers of 2—numbers like 256, 32,768, 65,536—reminders that the binary number system is lurking somewhere nearby.

[1]Most programming languages use the American Standard Code for Information Interchange (ASCII) to represent the English alphabet and other symbols. ASCII characters are stored in a single byte. Visual Basic recognizes both ASCII and a newer standard called Unicode, which includes the alphabets of many other human languages. A single Unicode character takes up two bytes in the computer's memory.

1.2 How Is a Program Converted into a Form That a Computer Can Use?

In the computer, all data, whatever its form, is stored and used in binary codes, strings of 1s and 0s. Instructions and data are stored together in the computer's memory using these binary codes. If you were to look at the binary codes representing instructions and data in memory, you could not tell the difference between them; they are distinguished

Machine language The language, made up of binary-coded instructions, that is used directly by the computer

Assembly language A low-level programming language in which a mnemonic is used to represent each of the machine language instructions for a particular computer

Assembler A program that translates an assembly language program into machine code

Compiler A program that translates a program written in a high-level language into machine code

Source code Data type specifications and instructions written in a high-level programming language

Object code A machine language version of source code

only by the manner in which the computer uses them. It is thus possible for the computer to process its own instructions as a form of data.

When computers were first developed, the only programming language available was the primitive instruction set built into each machine, the machine language, or *machine code.*

Even though most computers perform the same kinds of operations, their designers choose different sets of binary codes for each instruction. So the machine code for one family of computers is not the same as for another.

When programmers used machine language for programming, they had to enter the binary codes for the various instructions, a tedious process that was prone to error. Moreover, their programs were difficult to read and modify. In time, assembly languages were developed to make the programmer's job easier.

Instructions in an assembly language are in an easy-to-remember form called a mnemonic (pronounced "ni- ' män – ik"). Typical instructions for addition and subtraction might look like this:

Assembly Language	Machine Language
ADD	100101
SUB	010011

Although assembly language is easier for humans to work with, the computer cannot directly execute the instructions. Because a computer can process its own instructions as a form of data, it is possible to write a program to translate assembly language into machine code. Such a program is called an assembler.

Assembly language is a step in the right direction, but it still forces programmers to think in terms of individual machine instructions. Eventually, computer scientists developed high-level programming languages. These languages are easier to use than assembly languages or machine code because they are closer to English and other natural languages (see Figure 1.4).

A program called a compiler translates algorithms written in certain high-level languages (Visual Basic, C++, Java, Pascal, and Ada, for example) into machine language. If you write a program in a high-level language, you can run it on any computer that has the appropriate compiler. This is possible because most high-level languages are *standardized*, which means that an official description of the language exists.

The text of an algorithm written in a high-level language is called source code. To the compiler, source code is just input data. It translates the source code into a machine language form called object code (see Figure 1.5).

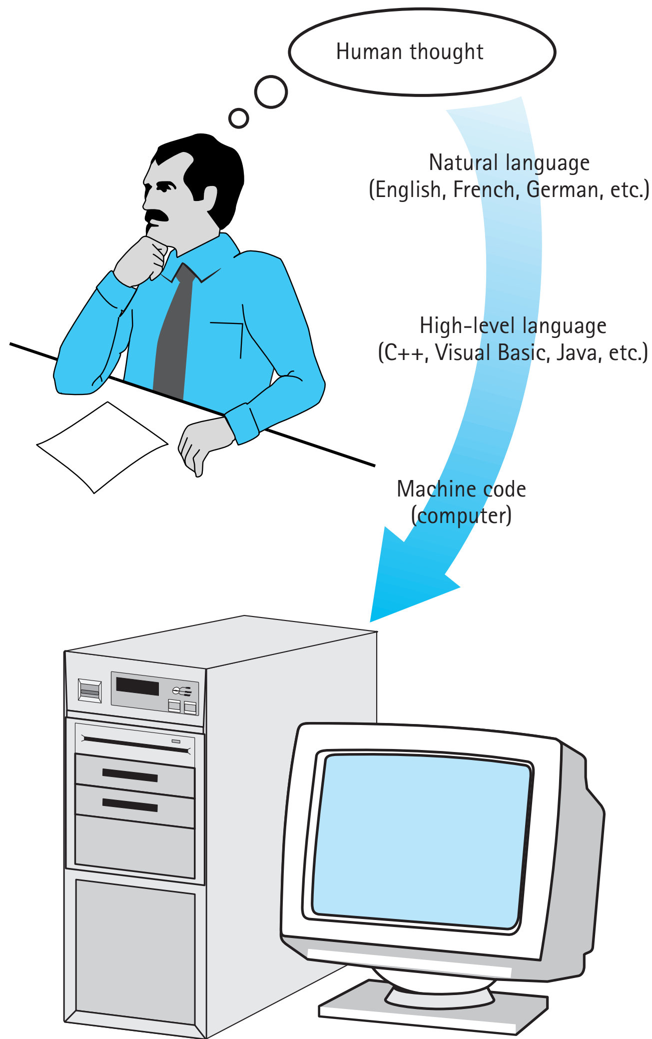


Figure 1.4 *Levels of abstraction*

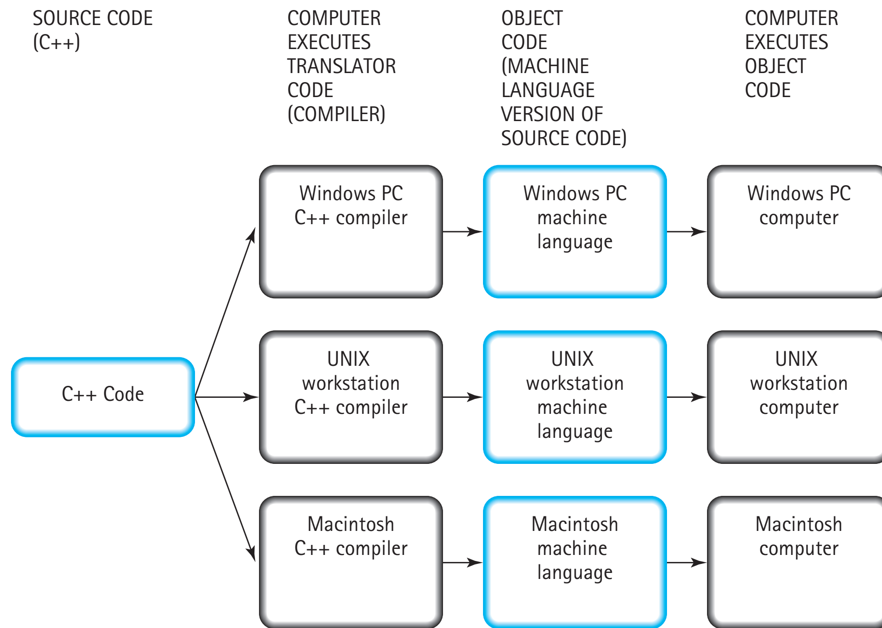


Figure 1.5 *High-level programming languages allow programs to be compiled on different systems.*

A benefit of standardized high-level languages is that they allow you to write *portable* (or *machine independent*) code. As Figure 1.5 emphasizes, a single C++ program can be run on different machines, whereas a program written in assembly language or machine language is not portable from one computer to another. Because each computer family has its own machine language, a machine language program written for computer A may not run on computer B.

Visual Basic takes a somewhat different approach than we have described. Visual Basic programs are translated into a standard machine language called Bytecode.

Bytecode A standard machine language into which Visual Basic source code is compiled

However, there are no computers that actually use Bytecode as their machine language. In order for a computer to run Bytecode programs, it must have another program called the Common Language Runtime (CLR) that serves as a language interpreter for the program. Just as an interpreter of human languages listens to words spoken in one language and speaks a translation of them in a language that another person understands, the CLR reads the Bytecode machine language instructions and translates them into machine language operations that the particular computer executes. Interpretation is done as the program is running, one instruction at a time. It is not the same as compilation, which is a separate step that

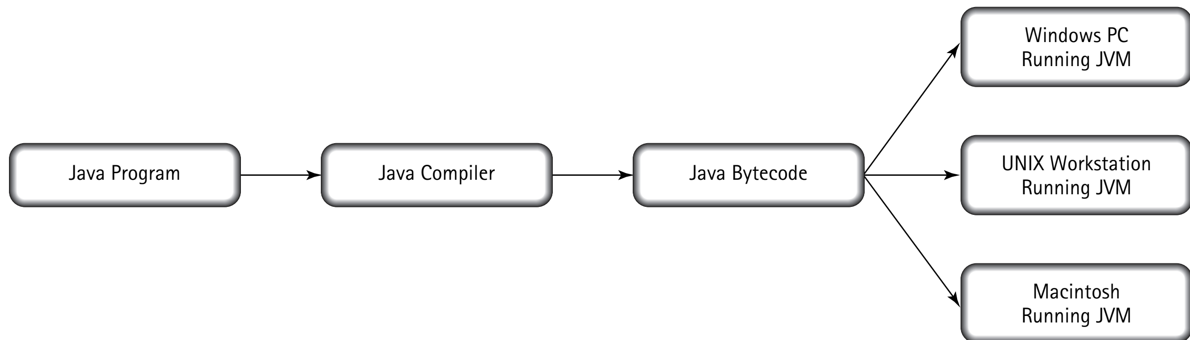


Figure 1.6 *Java compiler produces Bytecode that can be run on any machine with the JVM.*

translates all of the instructions in a program prior to execution. Figure 1.6 shows how the Visual Basic translation process works.

1.3 How Is Interpreting a Program Different from Executing It?

There is a significant distinction between direct execution and interpretation of a program. A computer can directly execute a program that is compiled into machine language. The CLR is one such machine language program that is directly executed. The computer cannot directly execute Bytecode. It must execute the CLR to interpret each Bytecode instruction in order to run the compiled Visual Basic program. The CLR does not produce machine code, like a compiler, but instead it reads each Bytecode instruction and gives the computer a corresponding series of operations to perform. Because each Bytecode instruction must first be interpreted, the computer cannot run Bytecode programs as quickly as it can execute machine language. Slower execution is the price we pay for increased portability.

Direct execution The process by which a computer performs the actions specified in a machine language program

Interpretation The translation, while a program is running, of non-machine language instructions (such as Bytecode) into executable operations

1.4 How Is Compilation Related to Interpretation and Execution?

It is important to understand that *compilation* and *execution* are two distinct processes. During compilation, the computer runs the compiler program. During execution, the object program is loaded into the computer's memory unit, replacing the compiler program. The computer then directly executes the object program, doing whatever the program instructs it to do (see Figure 1.7).

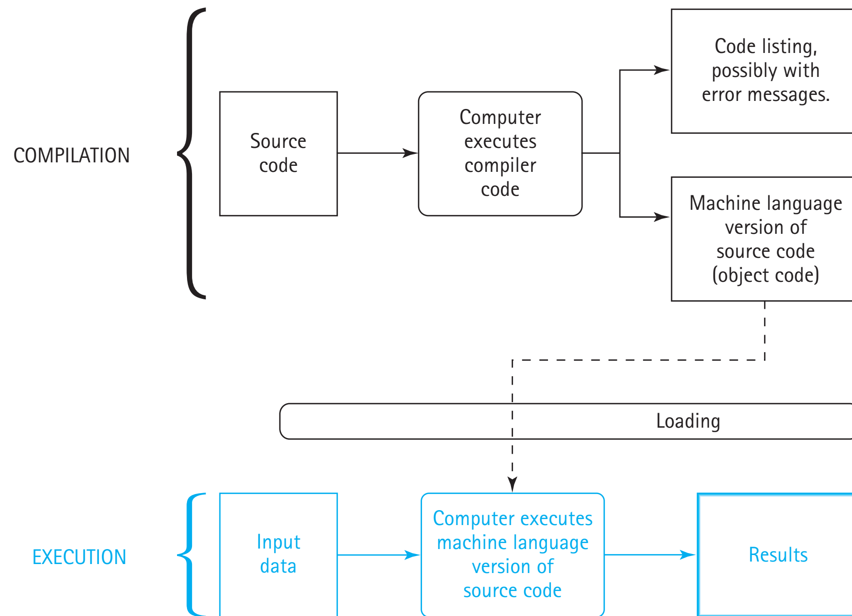


Figure 1.7 *Compilation/Execution*

We can use the CLR as an example of the process shown in Figure 1.7. The CLR is written in a high-level programming language such as C++ and then compiled into machine language. The machine language is loaded into the computer's memory, and the CLR is executed. Its input is a Visual Basic program that has been compiled into Bytecode. Its results are the series of actions that would take place if the computer could directly execute Bytecode. Figure 1.8 illustrates this process.

As you look at Figure 1.8, it is important to understand that the output from the compiler can be saved for future use. Once the CLR and the Visual Basic program have been compiled, they can be used over and over without being compiled again. You never need to compile the CLR because that has already been done for you. We have shown its compilation in Figure 1.8 simply to illustrate the difference between the traditional compile-execute steps and the compile-interpret steps that are used with Visual Basic.

Viewed from a different perspective, the CLR makes the computer look like a different computer that has Bytecode as its machine language. The computer itself hasn't changed—it is still the same collection of electronic circuits—but the CLR makes it appear that it has become a different machine. When a program is used to make one computer act like another computer, we call it a virtual machine. For convenience we may refer to the computer as "executing a Visual Basic program, " but keep in mind this is just shorthand for saying that "the computer is executing the CLR running a Visual Basic program."

Virtual machine A program that makes one computer act like another

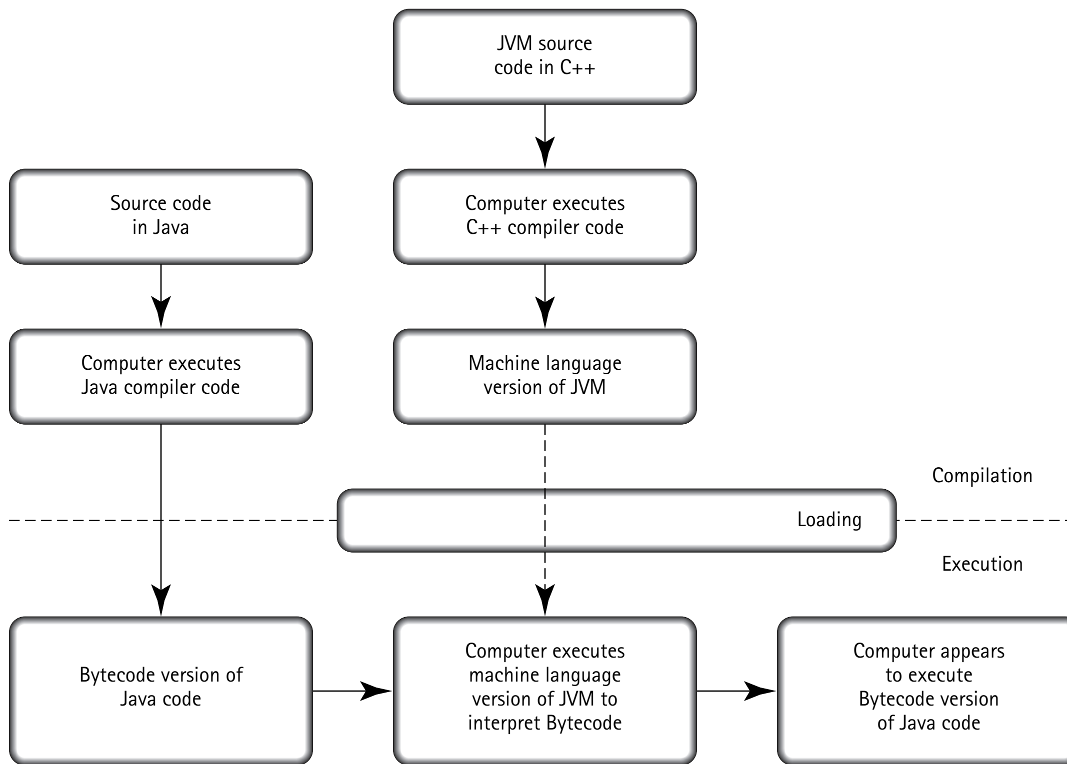


Figure 1.8 *Compilation and execution of JVM combined with compilation and interpretation of Bytecode*

1.5 What Kinds of Instructions Can Be Written in a Programming Language?

The instructions in a programming language reflect the operations a computer can perform:

- A computer can transfer data from one place to another.
- A computer can get data from an input device (a keyboard or mouse, for example) and write data to an output device (a screen, for example).
- A computer can store data into and retrieve data from its memory and secondary storage (parts of a computer that we discuss in the next section).
- A computer can compare data values for equality or inequality and make decisions based on the result.
- A computer can perform arithmetic operations (addition and subtraction, for example) very quickly.
- A computer can branch to a different section of the instructions.

Programming languages require that we use certain *control structures* to express algorithms as source code. There are four basic ways of structuring statements (instructions) in most programming languages: by sequence, selection, loop, and with subprograms. Visual Basic adds a fifth way: asynchronously (see Figure 1.9). A *sequence* is a series of statements that are executed one after another. *Selection*, the conditional control structure, executes different statements depending on certain conditions. The repetitive control structure, the *loop*, repeats statements while certain conditions are met. The *subprogram* allows us to structure our code by breaking it into smaller units. *Asynchronous* control lets us write code that handles events that originate outside of our program, such as the user clicking a button on the screen with their mouse. Each of these ways of structuring statements controls the order in which the computer executes the statements, which is why they are called control structures.

Assume you're driving a car. Going down a straight stretch of road is like following *sequence* of instructions. When you come to a fork in the road, you must decide which way to go and then take one or the other branch of the fork. This is what the computer does when it encounters a *selection control structure* (sometimes called a *branch* or *decision*) in a program. Sometimes you have to go around the block several times to find a place to park. The computer does the same sort of thing when it encounters a *loop* in a program.

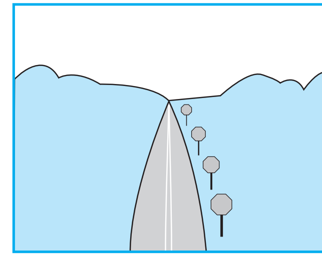
A *subprogram* is a named sequence of instructions written separately from the main program. When the program executes an instruction that refers to the name of the subprogram, the code for the subprogram is executed. When the subprogram has finished executing, execution of the program resumes at the next instruction. Suppose, for example, that every day you go to work at an office. The directions for getting from home to work form a procedure called "Go to the office." It makes sense, then, for someone to give you directions to a meeting by saying "Go to the office, then go four blocks west," without listing all the steps you have to take to get to the office. Subprograms allow us to write parts of our programs separately and then assemble them into final form. They can greatly simplify the task of writing large programs.

Responding to asynchronous events is like working as a pizza delivery person. You wait around the dispatch station with all of the other delivery people. The dispatcher calls your name and gives you some pizzas and a delivery address. You go deliver the pizzas and return to the dispatch station. At the same time, other delivery people may be out driving.[2] The term *asynchronous* means "not at the same time." In this context it refers to the fact that the user can, for example, click the mouse on the screen at any time while the program is running. The mouse click does not have to happen at some particular time corresponding to certain instructions within the program. The event shown in Figure 1.9 is like the dispatcher. You do not have to write it as part of your

---

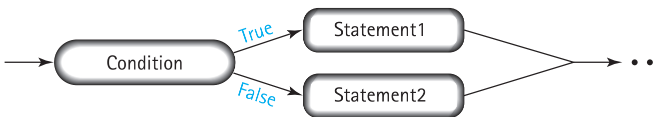
[2] Visual Basic actually allows us to write general asynchronous programs using a construct called a *thread*. Threaded programs are beyond the scope of this text. We restrict our use of asynchronous structures to handling events.

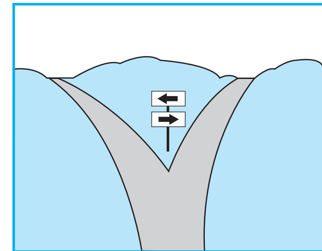
SEQUENCE



SELECTION (also called *branch* or *decision*)

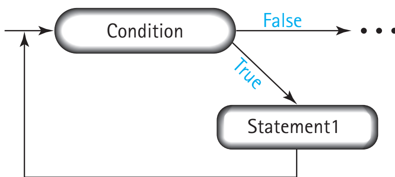
IF condition THEN statement1 ELSE statement2

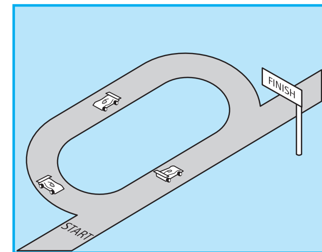




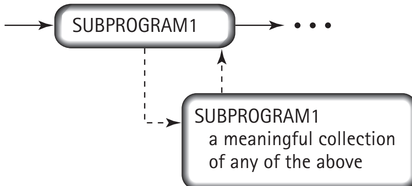
LOOP (also called *repetition* or *iteration*)

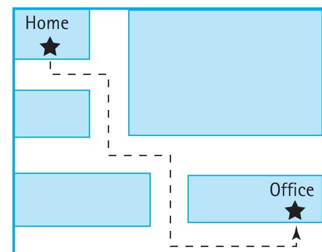
WHILE condition DO statement1

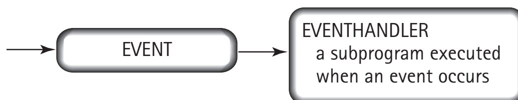




SUBPROGRAM (also called *procedure*, *function*, *method*, or *subroutine*)







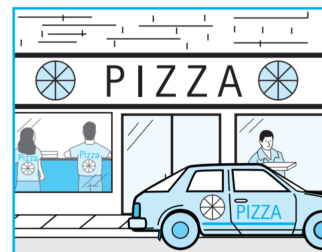


Figure 1.9 *Basic control structures of programming languages*

program because it is included in Visual Basic. You simply write the event-handler subprograms that the event calls.

Object-Oriented Programming Languages

Early programming languages focused their attention on the operations and control structures of programming. These procedural languages paid little explicit attention to the relationships between the operations and the data. At that time, a typical computer program used only simple data types such as integer and real numbers, which have obvious sets of operations defined by mathematics. Those operations were built directly into early programming languages. As people gained experience with the programming process, they began to realize that in solving complex problems, it is helpful to define new types of data, such as dates and times; these aren't a standard part of a programming language. Each new type of data typically has an associated set of operations, such as determining the number of days between two dates.

Procedural languages thus evolved to include the feature of *extensibility:* the capability to define new data types. However, they still treated the data and operations as separate parts of the program. A programmer could define a data type to represent the time of day and then write a subprogram to compute the number of minutes between two times, but could not explicitly indicate that the two are related.

Modern programming languages such as Visual Basic allow us to collect a data type and its associated operations into a single entity called an *object*. They are thus called *object-oriented programming languages*. The advantage of an object is that it makes the relationship between the data type and operations explicit. The result is that an object is a complete, self-contained unit that can be reused again in other programs. Reusability enables us to write a significant portion of our programs using existing objects, thereby saving us a considerable amount of time and effort.

Object A collection of data values and associated operations

Class A description of an object that specifies the types of data values that it can hold and the operations that it can perform

Instantiate To create an object based on the description supplied by a class

A class is a description of one or more like objects. Classes are usually collected into groups called *namespaces*. When we need an object in a program, we instantiate the class that describes the object. That is, we tell the Visual Basic compiler to provide us with one or more of the objects described by a specified class. One characteristic of an object-oriented programming language is having a large library of classes. In this text we present only a small subset of the classes that are available from the Visual Basic library. It is easy to be overwhelmed by the sheer size of Visual Basic's library, but many of those objects are highly specialized and unnecessary for learning the essential concepts of programming.

In the next few chapters we consider how to write simple codes that instantiate just a few of the classes in Visual Basic's library. By Chapter 7 we develop enough of the basic concepts of programming to start designing our own classes. Leading up to that point, we learn how to write some specific classes using patterns of Visual Basic code that we provide.

1.6 What Is a Computer?

You can learn a programming language, how to write programs, and how to run (execute) these programs without knowing much about computers. But if you know something about the parts of the computer, you can better understand the effect of each instruction in a programming language.

Most computers have six basic components: the memory unit, the arithmetic/logic unit, the control unit, input devices, output devices, and auxiliary storage devices. Figure 1.10 is a stylized diagram of the basic components of a computer.

The memory unit is an ordered sequence of storage cells, each capable of holding a piece of data. Each memory cell has a distinct address to which we refer in order to store data into it or retrieve data from it. These storage cells are called *memory cells*, or *memory locations*.[3] The memory unit holds data (input data or the product of computation) and instructions (programs), as shown in Figure 1.11.

Memory unit Internal data storage in a computer

[3]The memory unit is also referred to as RAM, an acronym for random-access memory (because we can access any location at random).

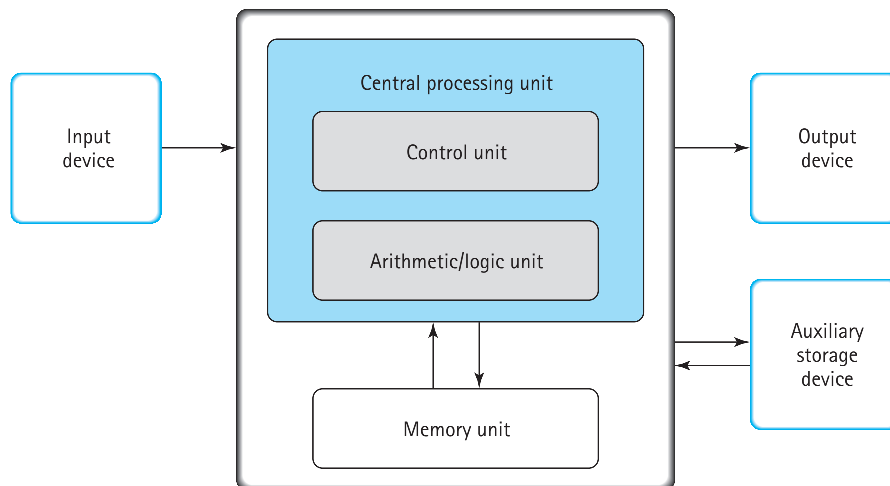


Figure 1.10 *Basic components of a computer*

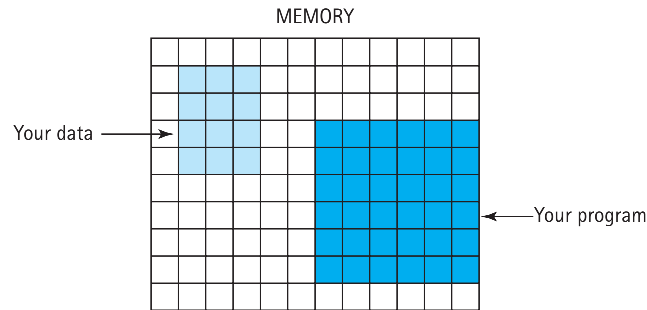


Figure 1.11 *Memory*

Central processing unit (CPU) The part of the computer that executes the instructions (program) stored in memory; made up of the arithmetic/logic unit and the control unit

Arithmetic/Logic unit (ALU) The component of the central processing unit that performs arithmetic and logical operations

Control unit The component of the central processing unit that controls the actions of the other components so that instructions (the program) are executed in the correct sequence

Input/Output (I/O) devices The parts of the computer that accept data to be processed (input) and present the results of that processing (output)

The part of the computer that follows instructions is called the central processing unit (CPU). The CPU usually has two components. The arithmetic/logic unit (ALU) performs arithmetic operations (addition, subtraction, multiplication, and division) and logical operations (comparing two values). The control unit controls the actions of the other components so that program instructions are executed in the correct order.

For us to use computers, we must have some way of getting data into and out of them. Input/Output (I/O) devices accept data that have been prepared (input) and present data that have been processed (output). A keyboard is a common input device. Another is the mouse. A video display is a common output device, as are printers and liquid crystal display (LCD) screens.

For the most part, computers simply move and combine data in memory. The many types of computers differ primarily in the size of their memories, the speed with which data can be recalled, the efficiency with which data can be moved or combined, and limitations on I/O devices.

When a program is executing, the computer proceeds through a series of steps, the *fetch-execute cycle*:

1. The control unit retrieves (*fetches*) the next coded instruction from memory.
2. The instruction is translated into control signals.
3. The control signals tell the appropriate unit (arithmetic/logic unit, memory, I/O device) to perform (*execute*) the instruction.
4. The sequence repeats from Step 1.

Computers can have a wide variety of peripheral devices. An auxiliary storage device, or *secondary storage device*, holds coded data for the computer until we actually want to use the data. Instead of inputting data every time, we can input it once and have the computer store it in an auxiliary storage device. Whenever we need to use the data, we tell the computer to transfer the data from the auxiliary storage device to its memory. An auxiliary storage device therefore serves as both an input and an output device. A *disk drive* is a cross between a compact disk player and a tape recorder. It uses a thin disk made out of magnetic material. A read/write head (similar to the record/playback head in a tape recorder) travels across the spinning disk, retrieving or recording data. A CD-ROM or a DVD-ROM drive use a laser to read information stored optically on a plastic disk. The CD-R and DVD-RAM forms of the CD and DVD can be used to both read and write data. A *magnetic tape drive* is like a tape recorder and is most often used to back up (make a copy of) the data on a disk in case the disk is ever damaged.

Peripheral device An input, output, or auxiliary storage device attached to a computer

Auxiliary storage device A device that stores data in encoded form outside the computer's main memory

Hardware The physical components of a computer

Software Computer programs; the set of all programs available on a computer

Interface A connecting link at a shared boundary that allows independent systems to meet and act on or communicate with each other

Interactive system A system that allows direct communication between user and computer

Operating system A set of programs that manages all of the computer's resources

Together, all of these physical components are known as *hardware*. The programs that allow the hardware to operate are called *software*. Hardware is usually fixed in design; software is easily changed. In fact, the ease with which software can be manipulated is what makes the computer such a versatile, powerful tool.

In addition to the programs that we write or purchase, there are programs in the computer that are designed to simplify the user/computer interface, making it easier for us to use the machine. The interface between user and computer is a set of I/O devices—for example, the keyboard, mouse, and a screen—that allows the user to communicate with the computer. We work with the keyboard, mouse, and screen on our side of the interface boundary; wires attached to the keyboard and the screen carry the electronic pulses that the computer works with on its side of the interface boundary. At the boundary itself is a mechanism that translates information for the two sides.

When we communicate directly with the computer through an interface, we are using an interactive system. Interactive systems allow direct entry of programs and data and provide immediate feedback to the user. In contrast, *batch systems* require that all data be entered before a program is run and provide feedback only after a program has been executed. In this text we focus on interactive systems, although in Chapter 9 we discuss file-oriented programs, which share certain similarities with batch systems.

The set of programs that simplifies the user/computer interface and improves the efficiency of processing is called *system software*. It includes the CLR and the Visual Basic compiler as well as the operating system and the editor (see Figure 1.12). The operating system manages all of the computer's resources. It can input programs, call

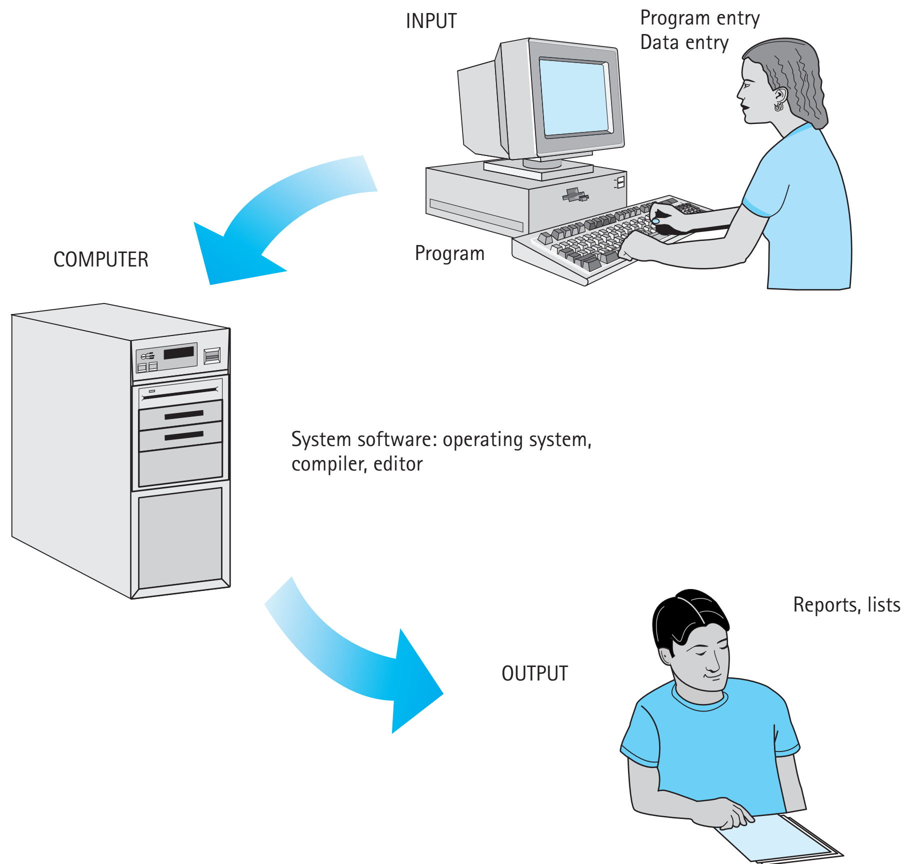


Figure 1.12 *User/Computer interface*

Editor An interactive program used to create and modify source programs or data

the compiler, execute object programs, and carry out any other system commands. The editor is an interactive program used to create and modify source programs or data.

1.7 Ethics and Responsibilities in the Computing Profession

Every profession operates with a set of ethics that help to define the responsibilities of people who practice the profession. For example, medical professionals have an ethical responsibility to keep information about their patients confidential. Engineers have an ethical responsibility to their employers to protect proprietary information, but they also have a responsibility to protect the public and the environment from harm that may result from their work. Writers are ethically bound not to plagiarize the work of others, and so on.

The computer presents us with a vast new range of capabilities that can affect people and the environment in dramatic ways. It thus challenges society with many new ethical issues. Some of our existing ethical practices apply to the computer, whereas other situations require new ethical rules. In some cases, there may not be established guidelines, but it is up to you to decide what is ethical. In this section we examine some common situations encountered in the computing profession that raise particular ethical issues.

A professional in the computing industry, like any other professional, has knowledge that enables him or her to do certain things that others cannot do. Knowing how to access computers, how to program them, and how to manipulate data gives the computer professional the ability to create new products, solve important problems, and help people to manage their interactions with the ever-more-complex world that we all live in. Knowledge of computers can be a powerful means to effecting positive change.

Knowledge also can be used in unethical ways. A computer can be programmed to trigger a terrorist's bomb, to sabotage a competitor's production line, or to steal money. Although these blatant examples make an extreme point and are unethical in any context, there are more subtle examples that are unique to computers.

Software Piracy

Computer software is easy to copy. But just like books, software is usually copyrighted—it is illegal to copy software without the permission of its creator. Such copying is called software piracy.

Software piracy The unauthorized copying of software for either personal use or use by others

Copyright laws exist to protect the creators of software (and books and art) so that they can make a profit from the effort and money spent developing the software. A major software package can cost millions of dollars to develop, and this cost (along with the cost of producing the package, shipping it, supporting customers, and allowing for retailer markup) is reflected in the purchase price. If people make unauthorized copies of the software, then the company loses those sales and either has to raise its prices to compensate or spend less money to develop improved versions of the software—in either case, a desirable piece of software becomes harder to obtain.

Software pirates sometimes rationalize their software theft with the excuse that they're just making one copy for their own use. It's not that they're selling a bunch of bootleg copies, after all. But if thousands of people do the same, then it adds up to millions of dollars in lost revenue for the company, which leads to higher prices for everyone.

Computing professionals have an ethical obligation to not engage in software piracy and to try to stop it from occurring. You never should copy software without permission. If someone asks you for a copy of a piece of software, you should refuse to supply it. If someone says they just want to "borrow" the software so they can "try it out," tell them they are welcome to try it out on your machine (or at a retailer's shop) but not to make a copy.

This rule isn't restricted to duplicating copyrighted software; it includes plagiarism of all or part of code. If someone gives you permission to copy some code, then

just like any responsible writer, you should acknowledge that person with a citation in the code.

Privacy of Data

The computer enables the compilation of databases containing useful information about people, companies, geographic regions, and so on. These databases allow employers to issue payroll checks, banks to cash a customer's check at any branch, the government to collect taxes, and mass merchandisers to send out junk mail. Even though we may not care for every use of databases, they generally have positive benefits. However, they also can be used in negative ways.

For example, a car thief who gains access to the state motor vehicle registry could print out a "shopping list" of valuable car models together with their owners' addresses. An industrial spy might steal customer data from a company database and sell it to a competitor. Although these are obviously illegal acts, computer professionals face other situations that are not so obvious.

Suppose your job includes managing the company payroll database. In that database are the names and salaries of the employees in the company. You might be tempted to poke around in the database and see how your salary compares to your associates—this act is unethical and an invasion of your associates' right to privacy. This information is confidential. Any information about a person that is not clearly public should be considered confidential. An example of public information is a phone number listed in a telephone directory. Private information includes any data that has been provided with an understanding that it will be used only for a specific purpose (such as the data on a credit card application).

A computer professional has a responsibility to avoid taking advantage of special access that he or she may have to confidential data. The professional also has a responsibility to guard that data from unauthorized access. Guarding data can involve such simple things as shredding old printouts, keeping backup copies in a locked cabinet, not using passwords that are easy to guess (such as a name or word), and more complex measures such as *encryption* (keeping it stored in a secret coded form).

Use of Computer Resources

If you've ever bought a computer, you know that it costs money. A personal computer can be relatively inexpensive, but it's still a major purchase. Larger computers can cost millions of dollars. Operating a PC may cost a few dollars a month for electricity and an occasional outlay for paper, disks, and repairs. Larger computers can cost tens of thousands of dollars per month to operate. Regardless of the type of computer, whoever owns it has to pay these costs. They do so because the computer is a resource that justifies its expense.

The computer is an unusual resource because it is valuable only when a program is running. Thus, the computer's time is really the valuable resource. There is no significant physical difference between a computer that is working and one that is sitting idle. Thus, unauthorized use of a computer is different from unauthorized use of a car. If one person uses another's car without permission, that individual must take possession of it

physically—that is, steal it. If someone uses a computer without permission, the computer isn't physically stolen, but just as in the case of car theft, the owner is being deprived of a resource that he or she is paying for.

For some people, theft of computer resources is a game—like joyriding in a car. The thief really doesn't want the resources; it is just the challenge of breaking through a computer's security system and seeing how far he or she can get without being caught. Success gives a thrilling boost to this sort of person's ego. Many computer thieves think that their actions are acceptable if they don't do any harm, but whenever real work is displaced from the computer by such activities, then harm is clearly being done. If nothing else, the thief is trespassing in the computer owner's property. By analogy, consider that even though no physical harm may be done by someone who breaks into your bedroom and takes a nap while you are away, such an action is certainly disturbing to you because it poses a threat of potential physical harm. In this case, and in the case of breaking into a computer, mental harm can be done.

Other thieves can be malicious. Like a joyrider who purposely crashes a stolen car, these people destroy or corrupt data to cause harm. They may feel a sense of power from being able to hurt others with impunity. Sometimes these people leave behind programs that act as time bombs, to cause harm long after they have gone. Another kind of program that may be left is a virus—a program that replicates itself, often with the goal of spreading to other computers. Viruses can be benign, causing no other harm than to use up some resources. Others can be destructive and cause widespread damage to data. Incidents have occurred in which viruses have cost billions of dollars in lost computer time and data.

Virus Code that replicates itself, often with the goal of spreading to other computers without authorization, and possibly with the intent of doing harm

Computing professionals have an ethical responsibility never to use computer resources without permission. This includes activities such as doing personal work on an employer's computer. We also have a responsibility to help guard resources to which we have access—by using unguessable passwords and keeping them secret, by watching for signs of unusual computer use, by writing programs that do not provide loopholes in a computer's security system, and so on.

Software Engineering

Humans have come to depend greatly on computers in many aspects of their lives. That reliance is fostered by the perception that computers function reliably; that is, they work correctly most of the time. However, the reliability of a computer depends on the care that is taken in writing its software.

Errors in a program can have serious consequences. Here are a few examples of real incidents involving software errors. An error in the control software of an F-18 jet fighter caused it to flip upside down the first time it flew across the equator. A rocket launch went out of control and had to be blown up because there was a comma typed in place of a period in its control software. A radiation therapy machine killed several patients because a software error caused the machine to operate at full power when the operator typed certain commands too quickly.

Even when the software is used in less critical situations, errors can have significant effects. Examples of such errors include

- An error in a word processor that causes your term paper to be lost just hours before it is due
- An error in a statistical program that causes a scientist to draw a wrong conclusion and publish a paper that must later be retracted
- An error in a tax preparation program that produces an incorrect return, leading to a fine

Software engineering The application of traditional engineering methodology and techniques to the development of software

Programmers thus have a responsibility to develop software that is free from errors. The process that is used to develop correct software is known as software engineering.

Software engineering has many aspects. The software life cycle described at the beginning of this chapter outlines the stages in the development of software. Different techniques are used at each of these stages. We address many of the techniques in this text. In Chapter 5 we introduce methodologies for developing correct algorithms. We discuss strategies for testing and validating programs in every chapter. We use a modern programming language that enables us to write readable, well-organized programs, and so on. Some aspects of software engineering, such as the development of a formal, mathematical specification for a program, are beyond the scope of this text.

1.8 Problem-Solving Techniques

You solve problems every day, often unaware of the process you are going through. In a learning environment, you usually are given most of the information you need: a clear statement of the problem, the necessary input, and the required output. In real life, the process is not always so simple. You often have to define the problem yourself and then decide what information you have to work with and what the results should be.

After you understand and analyze a problem, you must come up with a solution—an algorithm. Earlier we defined an algorithm as a step-by-step procedure for solving a problem in a finite amount of time with a finite amount of data. Although you work with algorithms all the time, most of your experience with them is in the context of *following* them. You follow a recipe, play a game, assemble a toy, or take medicine. In the problem-solving phase of computer programming, you will be *designing* algorithms, not following them. This means you must be conscious of the strategies you use to solve problems in order to apply them to programming problems.

Ask Questions

If you are given a task orally, you ask questions—When? Why? Where?—until you understand exactly what you have to do. If your instructions are written, you might put

question marks in the margin, underline a word or a sentence, or in some other way indicate that the task is not clear. Your questions may be answered by a later paragraph, or you might have to discuss them with the person who gave you the task.

These are some of the questions you might ask in the context of programming:

- What do I have to work with—that is, what are my data?
- What do the data items look like?
- What are the operations to be performed on the data?
- How much data is there?
- How will I know when I have processed all the data?
- What should my output look like?
- What special error conditions might come up?

Look for Things That Are Familiar

Never reinvent the wheel. If a solution exists, use it. If you've solved the same or a similar problem before, just repeat your solution. People are good at recognizing similar situations. We don't have to learn how to go to the store to buy milk, then to buy eggs, and then to buy candy. We know that going to the store is always the same; only what we buy is different.

In programming, certain problems occur again and again in different guises. A good programmer immediately recognizes a subtask he or she has solved before and plugs in the solution. For example, finding the daily high and low temperatures is really the same problem as finding the highest and lowest grades on a test. You want the largest and smallest values in a set of numbers (see Figure 1.13).

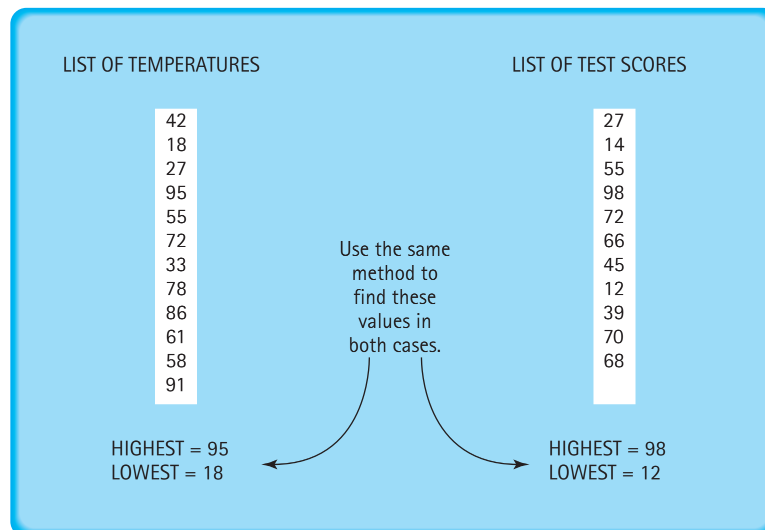


Figure 1.13 *Look for things that are familiar.*

In Chapter 8, we see how this problem-solving strategy can be implemented in Visual Basic using a mechanism called *inheritance*, which allows us to define a new object that adds to the capabilities of an existing object.

Solve by Analogy

Often a problem reminds you of one you have seen before. You may find solving the problem at hand easier if you remember how you solved the other problem. In other words, draw an analogy between the two problems. For example, a solution to a perspective projection problem from an art class might help you figure out how to compute the distance to a landmark when you are on a cross-country hike. As you work your way through the new problem, you come across things that are different than they were in the old problem, but usually these are just details that you can deal with one at a time.

Analogy is really just a broader application of the strategy of looking for things that are familiar. When you are trying to find an algorithm for solving a problem, don't limit yourself to computer-oriented solutions. Step back and try to get a larger view of the problem. Don't worry if your analogy doesn't match perfectly—the only reason for starting with an analogy is that it gives you a place to start (see Figure 1.14). The best programmers are people who have broad experience in solving all kinds of problems.

Means-Ends Analysis

Often the beginning state and the ending state are given; the problem is to define a set of actions that can be used to get from one to the other. Suppose you want to go from Boston, Massachusetts to Austin, Texas. You know the beginning state (you are in Boston) and the ending state (you want to be in Austin). The problem is how to get from one to the other.

In this example, you have lots of choices. You can fly, walk, hitchhike, ride a bike, or whatever. The method you choose depends on your circumstances. If you're in a hurry, you'll probably decide to fly.

Once you've narrowed down the set of actions, you have to work out the details. It may help to establish intermediate goals that are easier to meet than the overall goal. Let's say there is a really cheap, direct flight to Austin out of Newark, New Jersey. You might decide to divide the trip into legs: Boston to Newark and then Newark to Austin. Your intermediate goal is to get from Boston to Newark. Now you only have to examine the means of meeting that intermediate goal (see Figure 1.15).

A library catalog system can give insight into how to organize a parts inventory.

Figure 1.14 *Analogy*

Start: Boston **Goal:** Austin	**Means:** *Fly*, walk, hitchhike, bike, drive, sail, bus
Start: Boston **Goal:** Austin	**Revised Means:** Fly to Chicago and then Austin; *fly to Newark and then Austin:* fly to Atlanta and then Austin
Start: Boston **Intermediate Goal:** Newark **Goal:** Austin	**Means to Intermediate Goal:** *Commuter flight*, walk, hitchhike, bike, drive, sail, bus
Solution: Take commuter flight to Newark and then catch cheap flight to Austin	

Figure 1.15 *Means-ends analysis*

The overall strategy of means-ends analysis is to define the ends and then to analyze your means of getting between them. The process translates easily to computer programming. You begin by writing down what the input is and what the output should be. Then you consider the actions a computer can perform and choose a sequence of actions that can transform the input into the results.

Divide and Conquer

We often break up large problems into smaller units that are easier to handle. Cleaning the whole house may seem overwhelming; cleaning the rooms one at a time seems much more manageable. The same principle applies to programming. We break up a large problem into smaller pieces that we can solve individually (see Figure 1.16). In

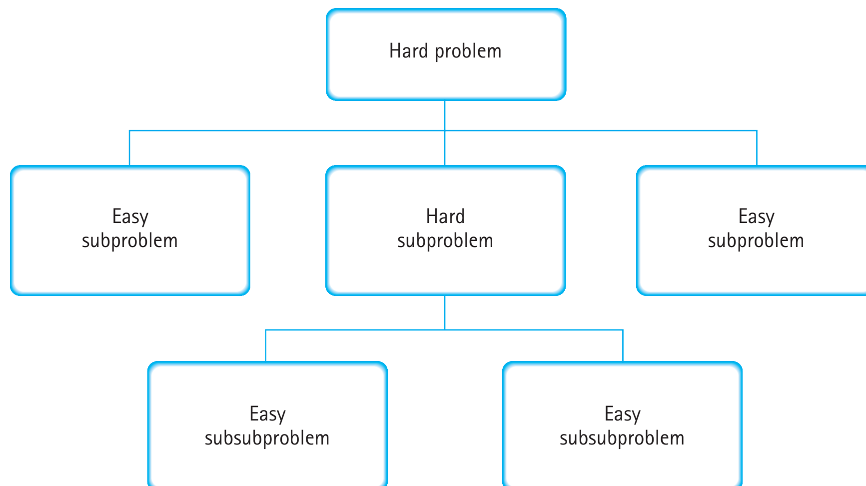


Figure 1.16 *Divide and conquer*

fact, the object-oriented design and functional decomposition methodologies, which we describe in Chapter 5, are both based on the principle of divide and conquer.

The Building-Block Approach

Another way of attacking a large problem is to see if any solutions for smaller pieces of the problem exist. It may be possible to put some of these solutions together end to end to solve most of the big problem. This strategy is just a combination of the look-for-familiar-things and divide-and-conquer approaches. You look at the big problem and see that it can be divided into smaller problems for which solutions already exist. Solving the big problem is just a matter of putting the existing solutions together, like mortaring together blocks to form a wall (see Figure 1.17).

With an object-oriented programming language, we often solve a problem by first looking in the class library to see what solutions have been developed previously and then writing a small amount of additional code to put the pieces together. As we see later, this problem-solving technique is the basis for the methodology called *object-oriented design.*

Merging Solutions

Another way to combine existing solutions is to merge them on a step-by-step basis. For example, to compute the average of a list of values, we must both sum and count the values. If we already have separate solutions for summing values and for counting the number of values, we can combine them. But if we first do the summing and then do the counting, we have to read the list twice. We can save steps if we merge these two solutions: read a value and then add it to the running total and add 1 to our count

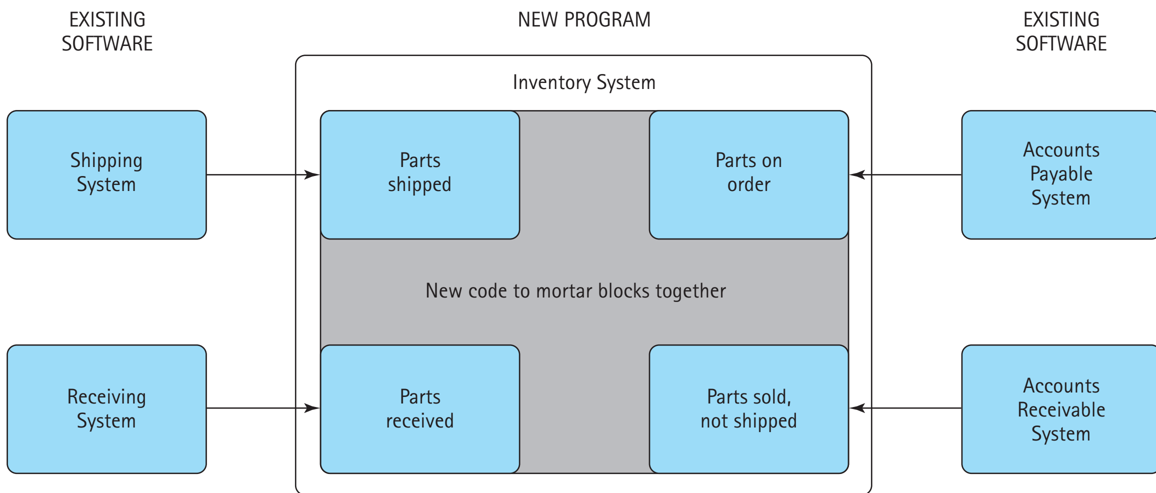


Figure 1.17 *Building-block approach*

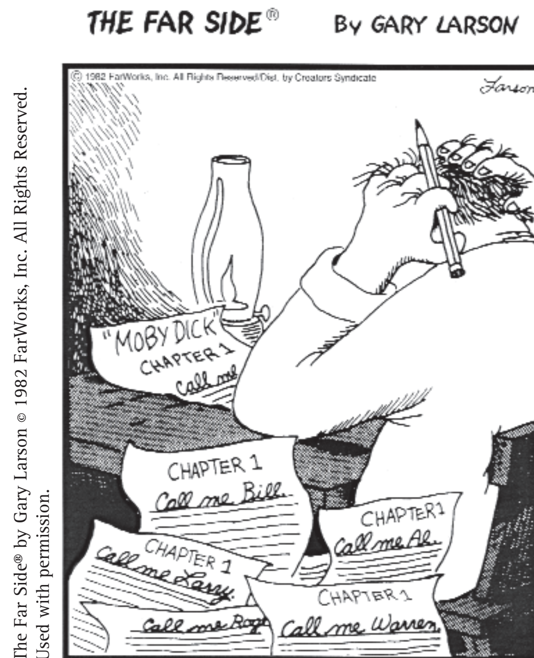
before going on to the next value. Whenever the solutions to subproblems duplicate steps, think about merging them instead of joining them end to end.

Mental Blocks: The Fear of Starting

Writers are all too familiar with the experience of staring at a blank page, not knowing where to begin. Programmers have the same difficulty when they first tackle a big problem. They look at the problem and it seems overwhelming (see Figure 1.18).

Remember that you always have a place to begin solving any problem: Write it down on paper in your own words so that you understand it. Once you paraphrase the problem, you can focus on each of the subparts individually instead of trying to tackle the entire problem at once. This process gives you a clearer picture of the overall problem. It helps you see pieces of the problem that look familiar or that are analogous to other problems you have solved. And it pinpoints areas where something is unclear, where you need more information.

As you write down a problem, you tend to group things together into small, understandable chunks of data and operations, which may be natural places to split the problem up—to divide and conquer. Your description of the problem may collect all of the information about data and results into one place for easy reference. Then you can see the beginning and ending states necessary for means-ends analysis.



The Far Side® by Gary Larson © 1982 FarWorks, Inc. All Rights Reserved.
Used with permission.

Figure 1.18 *Mental block.*

Most mental blocks are caused by not really understanding the problem. Rewriting the problem in your own words is a good way to focus on the subparts of the problem, one at a time, and to understand what is required for a solution.

Algorithmic Problem Solving

Coming up with an algorithm for solving a particular problem is not always cut-and-dried. It fact, it is usually a trial-and-error process requiring several attempts and refinements. We test each attempt to see if it really solves the problem. If it does, fine. If it doesn't, we try again. We typically use a combination of the techniques we've described to solve any nontrivial problem.

Remember that the computer can only do certain things. Your primary concern, then, is how to make the computer transform, manipulate, calculate, or process the input data to produce the desired output. If you keep in mind the allowable instructions and data types in your programming language, you won't design an algorithm that is difficult or impossible to code.

In the case study that follows, we develop a program for calculating employees' weekly wages. It typifies the thought processes involved in writing an algorithm and coding it as a program, and it shows you what a complete Visual Basic program looks like.

Problem-Solving Case Study

A Company `Payroll` Program

Problem A company needs a program to figure its weekly payroll. The input data, consisting of each employee's identification number, pay rate, and hours worked, is in the file datafile.dat in secondary storage. The program should input the data for each employee, calculate the weekly wages, save the input information for each employee along with the weekly wages in a file, and display the total wages for the week on the screen, so that the payroll clerk can transfer the appropriate amount into the payroll account.

Discussion At first glance, this seems like a simple problem. But if you think about how you would do it by hand, you see that you need to ask questions about the specifics of the process: What is the employee data that we need and how is the data written to the file? How are wages computed? In what file should the results be stored? How does the program know that all of the employees have been processed? How should the total be displayed?

- The data for each employee includes an employee identification number, the employee's hourly pay rate, and the hours worked. Each data value is written on a separate line.
- Wages equal the employees pay rate times the number of hours worked up to 40 hours. If an employee worked more than 40 hours, wages equal the employee's pay rate times 40 hours, plus $1\frac{1}{2}$ times the employee's regular pay rate times the number of hours worked above 40.
- The results should be stored in a file called payfile.dat.

- The program knows to finish the processing when there is no more data in the input file.
- The total should be shown in a window on the screen that can be closed by the user.

We begin by *looking for things that are familiar.* An experienced Visual Basic programmer immediately recognizes that this problem contains many different objects that are represented as classes in the Visual Basic library. The input file and the output file are objects, as is the window in which the total payroll is displayed on the screen. The employee identification number, pay rate, hours worked, wages earned, and total wages are objects in the problem that we must find a way to represent in our algorithm. Here's a list of the objects we've identified:

- Input file, `datafile.dat`, represented by one of the Visual Basic file classes
- Output file, `payfile.dat`, represented by another Visual Basic file class
- Display window, represented by a Visual Basic `Messagebox` class
- Employee identification number
- Pay rate
- Hours worked
- Wages
- Total wages

Now that we know the objects we are working with, we need to fit them together with operations that enable them to exchange information. The operations coordinate the behavior of the objects in a way that solves the problem, like the choreography that coordinates ballet dancers moving around a stage, interacting with each other.

Let's apply the *divide-and-conquer* approach to identify the main operations in which our objects must participate. It's clear that there are two main steps to be accomplished. One is to process the input file, and the other is to display the total on the screen. Let's look at each of those steps separately, once again applying *divide-and-conquer.*

First we consider the processing of the data in the input file. Each data set represents one employee, and we process each employee's data in turn. There are three obvious steps in almost any problem of this type. For each person, we must:

1. Get the data.
2. Compute the results.
3. Output the results.

Our first step is to get the data. (By *get*, we mean *read* or *input* the data.) We need three pieces of data for each employee: employee identification number, hourly pay rate, and number of hours worked. Each data value is written in the input file. Therefore, to input the data, we take these steps:

Read the employee number.

Read the pay rate.

Read the number of hours worked.

The next step is to compute the wages. Let's expand this step with *means-ends analysis.* Our starting point is the set of data values that was input; our desired ending, the payroll for

the week. The means at our disposal are the basic operations that the computer can perform, which include calculation and control structures. Let's begin by working backward from the end.

We know that there are two formulas for computing wages: one for regular hours and one for overtime. If there is no overtime, wages are simply the pay rate times the number of hours worked. If the number of hours worked is greater than 40, however, wages are 40 times the pay rate, plus the number of overtime hours times $1\frac{1}{2}$ times the pay rate. The overtime hours are computed by subtracting 40 from the total number of hours worked. Here are the two formulas:

We now have the means to compute the wages for each person. Our intermediate goal is then to execute the correct formula given the input data. We must decide which formula to use and employ a branching control structure to make the computer execute the appropriate formula. The decision that controls the branching structure is simply whether more than 40 hours have been worked. We now have the means to get from our starting point to the desired end. To figure the wages, then, we take the following steps:

```
If hours worked is greater than 40.0, then
    wages = (40.0 × pay rate) + (hours worked − 40.0) × 1.5 × pay rate
otherwise
    wages = hours worked × pay rate
```

The last step, outputting the results, is simply a matter of directing the computer to write the employee number, the pay rate, the number of hours worked, and the wages into payfile.dat:

```
Write the employee number, pay rate, hours worked, and wages into payfile.dat
```

We now have an algorithm that processes the data for one employee. We need to extend this algorithm to handle all of the employees. Let's use *the building-block approach* to enclose our three main steps (getting the data, computing the wages, and outputting the results) within a looping structure that continues until each employee has been processed. Once we have computed the wages for one employee, we need to add them to a running total so that we can display it at the end of processing. Our algorithm now coordinates the behavior of the objects to accomplish the first of our two major steps.

The second major step is to display the total wages and stop the program. We again use *divide-and-conquer* to break this into a series of steps:

```
Call a Messagebox object
Pass the total to the Messagebox
Show the Messagebox on the screen
Stop the program
```

Finally, we must take care of housekeeping chores. Before we start processing, we must prepare the input file for reading, prepare the output file to receive the results, and set the running total to zero.

What follows is the complete algorithm. Calculating the wages is written as a separate subalgorithm that is defined below the main algorithm. Notice that the algorithm is simply a very precise description of the same steps you would follow to do this process by hand.

Main Algorithm

Prepare to read a list of employee information (open file object datafile.dat)
Prepare to write a list of employees' wages (open file object payfile.dat)
Set the total payroll to zero
while there is more data in file dataFile
    Read employee number
    Read the pay rate
    Read the number of hours worked
    Calculate pay
    Add the employee's wages to the total payroll
    Write the employee number, pay rate, hours worked, and wages into the list (file payFile)
Call a Messagebox object
Pass the total to the Messagebox
Display the Messagebox on the screen
Stop the program

Subalgorithm for Calculating Pay

if number of hours worked is greater than 40.0, then
    wages = (40.0 × pay rate) + (hours worked − 40.0) × 1.5 × pay rate
else
    wages = hours worked × pay rate

Before we implement this algorithm, we should test it by hand, simulating the algorithm with specific data values. Case Study Follow-Up Exercise 2 asks you to carry out this test.

What follows is the Visual Basic program for this algorithm. It's here to give you an idea of what you'll be learning. If you've had no previous exposure to programming, you probably won't understand most of the program. Don't worry; you will soon. In fact, throughout this book as we introduce new constructs, we refer you back to the `Payroll` program. One more thing: The remarks following apostrophes (') are called comments. They are here to help you understand the program; the compiler ignores them.

```
Imports System.io

Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim empNum As String
    Dim payRate As Double
    Dim hours As Double
    Dim wages As Double
    Dim total As Double = 0
    Dim theFile As File
    Dim dataFile As StreamReader
    Dim payFile As StreamWriter
#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()
        dataFile = theFile.OpenText("datafile.dat")
        payFile = theFile.CreateText("payfile.dat")
        While (dataFile.Peek <> -1)
            empNum = dataFile.ReadLine()
            payRate = CDbl(dataFile.ReadLine())
            hours = CDbl(dataFile.ReadLine())
            wages = CalcPay(payRate, hours)
            total = total + wages
            ' Put results into payFile
            payFile.WriteLine(empNum & " " & payRate & " " & hours _
                              & " " & wages)
        End While
        MessageBox.Show("Total payroll for the week is $" & total & _
                        " Close window to exit program.")
        payFile.Close()
        dataFile.Close()
        'Add any initialization after the InitializeComponent() call

    End Sub
    Private Function calcPay(ByVal payRate As Double, _
    ByVal hours As Double)
        Const MAX_HOURS As Double = 40
        Const OVERTIME As Double = 1.5
        If (hours > MAX_HOURS) Then
            Return (MAX_HOURS * payRate) + (hours - MAX_HOURS) * _
                    payrate * OVERTIME
```

```
        Else
            Return hours * payRate
        End If
    End Function
    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing _
        As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
#End Region

End Class
```

Given the following input in file `datafile.dat`:

```
534923445
6.54
45
103428439
12.82
38
131909545
8.20
52
739219803
10.00
40
```

The program outputs the following in file `payfile.dat`:

```
534923445 6.54 45 310.65
103428439 12.82 38 487.16
131909545 8.2 52 475.6
739219803 10 40 400
```

And then it displays the following window:

For such a simple task, the length of this program is rather daunting. Don't worry. A large part of this program involves preparation for inputting and outputting data. These steps become second nature to you very shortly, and you can use the same algorithmic steps again and again (the building-block approach).

Summary

We think nothing of turning on the television and sitting down to watch it. It's a communication tool we use to enhance our lives. Computers are becoming as common as televisions, just a normal part of our lives. And like televisions, computers are based on complex principles but are designed for easy use.

Computers are unintelligent; they must be told what to do. A true computer error is extremely rare (usually due to a component malfunction or an electrical fault). Because we tell the computer what to do, most errors in computer-generated output are really human errors.

Computer programming is the process of planning a sequence of steps for a computer to apply to data. It involves a problem-solving phase and an implementation phase. After analyzing a problem, we develop and test a general solution (algorithm). This general solution becomes a concrete solution—our program—when we write it in a high-level programming language. The sequence of instructions that makes up our program is then either compiled into machine code (the language the computer uses) or Bytecode (the language the Common Language Runtime, or CLR, uses). After correcting any errors or "bugs" that show up during testing, our program is ready to use.

Once we begin to use the program, it enters the maintenance phase. Maintenance involves correcting any errors discovered while the program is being used and changing the program to reflect changes in the user's requirements.

Data and instructions are represented as binary numbers (numbers consisting of just 1s and 0s) in electronic computers. The process of converting data and instructions into a form usable by the computer is called coding.

A programming language reflects the range of operations a computer can perform. In this text, you will learn to write programs in the high-level programming language called Visual Basic. The basic control structures in the Visual Basic programming language—sequence, selection, loop, subprogram, and asynchronous—are based on the fundamental operations of the computer. Visual Basic provides the ability to collect data

and operations into self-contained units called objects that can be reused in other programs.

Computers are composed of six basic parts: the memory unit, the arithmetic/logic unit, the control unit, input devices, output devices, and auxiliary storage devices. The arithmetic/logic unit and control unit together are called the central processing unit. The physical parts of the computer are called hardware. The programs that are executed by the computer are called software.

System software is a set of programs designed to simplify the user/computer interface. It includes the compiler, the operating system, the CLR, and the editor.

Computing professionals are guided by a set of ethics, as are members of other professions. Among the responsibilities that we have are: copying software only with permission and including attribution to other programmers when we make use of their code, guarding the privacy of confidential data, using computer resources only with permission, and carefully engineering our programs so that they work correctly.

We've said that problem solving is a integral part of the programming process. Although you may have little experience programming computers, you have lots of experience solving problems. The key is to stop and think about the strategies that you use to solve problems, and then use those strategies to devise workable algorithms. Among those strategies are asking questions, looking for things that are familiar, solving by analogy, applying means-ends analysis, dividing the problem into subproblems, using existing solutions to small problems to solve a larger problem, merging solutions, and paraphrasing the problem in order to overcome a mental block.

The computer is widely used today in science, engineering, business, government, medicine, production of consumer goods, and the arts. Learning to program in Visual Basic can help you use this powerful tool effectively.

Quick Check

The Quick Check is intended to help you decide if you've met the goals set forth at the beginning of each chapter. If you understand the material in the chapter, the answer to each question should be fairly obvious. After reading a question, check your response against the answers listed at the end of the Quick Check. If you don't know an answer or don't understand the answer that's provided, turn to the page(s) listed at the end of the question to review the material.

1. What is a computer program? (p. 3)
2. What are the three phases in a program's life cycle? (p. 4)
3. Is an algorithm the same as a program? (pp. 4–5)
4. What is a programming language? (p. 6)
5. What are the advantages of using a high-level programming language? (pp. 12–13)
6. What does a compiler do? (p. 10)
7. What is the difference between machine code and Bytecode? (pp. 11–13)
8. What part does the Common Language Runtime play in the compilation and interpretation process? (pp. 13–15)
9. Name the five basic ways of structuring statements in Visual Basic. (pp. 15–17)

10. What are the six basic components of a computer? (pp. 19–22)
11. What is the difference between hardware and software? (pp. 20–21)
12. In what regard is theft of computer time like stealing a car? How are the two crimes different? (pp. 22–26)
13. What is the divide-and-conquer approach? (p. 29)

Answers

1. A computer program is a sequence of instructions performed by a computer. 2. The three phases of a program's life cycle are problem-solving, implementation, and maintenance. 3. No. All programs are algorithms, but not all algorithms are programs. 4. A set of rules, symbols, and special words used to construct a program. 5. A high-level programming language is easier to use than an assembly language or a machine language, and programs written in a high-level language can be run on many different computers. 6. The compiler translates a program written in a high-level programming language to either object code or Bytecode. 7. Machine code is the native binary language that is directly executed by any particular computer. Bytecode is a standard portable machine language that is executed by the Common Language Runtime, but it is not directly executed by the computer. 8. It translates the Bytecode instructions into operations that are executed by the computer. 9. Sequence, selection, loop, subprogram, and asynchronous. 10. The basic components of a computer are the memory unit, arithmetic/logic unit, control unit, input and output devices, and auxiliary storage devices. 11. Hardware is the physical components of the computer; software is the collection of programs that run on the computer. 12. Both crimes deprive the owner of access to a resource. A physical object is taken in a car theft, whereas time is the thing being stolen from the computer owner. 13. The divide-and-conquer approach is a problem-solving technique that breaks a large problem into smaller, simpler subproblems.

Exam Preparation Exercises

1. Explain why the following series of steps is not an algorithm, then rewrite the series so it is.

   Shampooing

   1. Rinse.
   2. Lather.
   3. Repeat.

2. Describe the input and output files used by a compiler.
3. In the following recipe for chocolate pound cake, identify the steps that are branches (selection) and loops, and the steps that are references to subalgorithms outside the algorithm.

```
Preheat the oven to 350 degrees
Line the bottom of a 9-inch tube pan with wax paper
Sift 2¾ c flour, ¾ t cream of tartar, ½ t baking soda, 1½ t salt, and 1¾ c sugar into a large bowl
Add 1 c shortening to the bowl
If using butter, margarine, or lard, then
   add 2/3 c milk to the bowl,
else
   (for other shortenings) add 1 c minus 2 T of milk to the bowl
Add 1 t vanilla to the mixture in the bowl
If mixing with a spoon, then
   see the instructions in the introduction to the chapter on cakes,
else
   (for electric mixers) beat the contents of the bowl for 2 minutes at medium speed, scraping the
        bowl and beaters as needed
Add 3 eggs plus 1 extra egg yolk to the bowl
Melt 3 squares of unsweetened chocolate and add to the mixture in the bowl
Beat the mixture for 1 minute at medium speed
Pour the batter into the tube pan
Put the pan into the oven and bake for 1 hour and 10 minutes
Perform the test for doneness described in the introduction to the chapter on cakes
Repeat the test once each minute until the cake is done
Remove the pan from the oven and allow the cake to cool for 2 hours
Follow the instructions for removing the cake from the pan, given in the introduction to the chapter
        on cakes
Sprinkle powdered sugar over the cracks on top of the cake just before serving
```

4. Put a check next to each item below that is a peripheral device.
   _______ a. Disk drive
   _______ b. Arithmetic/logic unit
   _______ c. Magnetic tape drive
   _______ d. Printer
   _______ e. CD-ROM drive
   _______ f. Memory
   _______ g. Auxiliary storage device
   _______ h. Control unit
   _______ i. LCD display
   _______ j. Mouse

5. Next to each item below, indicate whether it is hardware (H) or software (S).
   _______ a. Disk drive
   _______ b. Memory
   _______ c. Compiler
   _______ d. Arithmetic/logic unit
   _______ e. Editor
   _______ f. Operating system
   _______ g. Object program
   _______ h. Common Language Runtime
   _______ i. Central processing unit
6. Means-ends analysis is a problem-solving strategy.
   a. What are three things you must know in order to apply means-ends analysis to a problem?
   b. What is one way of combining this technique with the divide-and-conquer strategy?
7. Show how you would use the divide-and-conquer approach to solve the problem of finding a job.
8. Distinguish between information and data.

Programming Warm-Up Exercises

1. Write an algorithm for driving from where you live to the nearest airport that has regularly scheduled flights. Restrict yourself to a vocabulary of 50 words plus numbers and place names. You must select the appropriate set of words for this task. An example of a vocabulary is given in Appendix A, the list of reserved words (words with special meanings) in the Visual Basic programming language. The purpose of this exercise is to give you practice in writing simple, exact instructions with an equally small vocabulary.
2. Write an algorithm for making a peanut butter and jelly sandwich, using a vocabulary of just 50 words (you choose the words). Assume that all ingredients are in the refrigerator and that the necessary tools are in a drawer under the kitchen counter. The instructions must be very simple and exact because the person making the sandwich has no knowledge of food preparation and takes every word literally.
3. In Exercise 1 above, identify the sequential, conditional, repetitive, and subprograms steps.

Case Study Follow-Up Exercises

1. Using Figure 1.16 as a guide, construct a divide-and-conquer diagram of the Problem-Solving Case Study, A Company `Payroll` Program.
2. Use the following data set to test the payroll algorithm presented on page 35. Follow each step of the algorithm just as it is written, as if you were a computer. Then check your results by hand to be sure that the algorithm is correct.

ID Number	Pay Rate	Hours Worked
327	8.30	48
201	6.60	40
29	12.50	40
166	9.25	51
254	7.00	32

3. In the Company `Payroll` Program case study, we used means-ends analysis to develop the subalgorithm for calculating pay. What are the *ends* in the analysis? That is, what information did we start with and what information did we want to end up with?
4. In the `Payroll` program, certain remarks are preceded by the symbol `'`. What are these remarks called, and what does the compiler do with them? What is their purpose?