In this chapter, we will introduce some fundamental concepts of computer programming. We will briefly discuss the nature of programming and describe the basic building blocks that form the structure of almost all computer programs. You will learn to construct simple programs using statements (instructions) that are generic in nature. These generic statements are similar to those used in many computer languages.

After reading this chapter, you will be able to do the following:

- Understand what programs are, and in a very general way, how they are written [Section 1.1]
- Apply the problem-solving strategy for computer programming problems [Section 1.1]
- Understand the basic components of a computer program: input, processing, and output [Sections 1.2, 1.3]
- Write generic program statements that perform input, processing, and output operations [Sections 1.2, 1.3]
- Name and use constants and variables in a program [Section 1.2]
- Use basic arithmetic operations (addition, subtraction, multiplication, division, modulus, and exponentiation) in a program [Section 1.3]
- Understand how computers use the hierarchy of operations [Section 1.3]
- Understand what the String and Character data types are and how characters and strings are represented in the computer [Section 1.4]
- Use the concatenation operator [Section 1.4]
- Understand what the Integer data type is and how signed and unsigned integers are represented in the computer [Section 1.5]
- Understand what the Floating Point data type is, how it is represented in the computer, and how this differs from the Integer data type [Section 1.6]
You Are Already a Programmer!

Imagine that you need new running shoes. How would you go about getting them? You might check your bank balance or your wallet to see if you have enough money and then take a trip to the mall or sports store and browse the shoe departments. You'd try on some shoes, and if you found a pair that was the right size, color, and style, you'd probably buy them on the spot. Your plan of action would be to check for cash, browse the stores, try on shoes, and buy a pair.

But what if you need a new car? You wouldn't just check your wallet for cash, browse car dealerships, drive a few models, and buy one. You'd probably formulate a more complicated plan of action. You may check your credit or arrange for a bank loan before you shop. Or you may first do some research on various cars, prices, and loan interest rates and repayment plans. Then you might peruse car ads in newspapers or online and take some test drives. Regardless of how you finally buy the car, you'll follow some plan of action, albeit a different one from the plan to buy a pair of running shoes.

Really, a computer program is simply a list of instructions carried out by a computer to accomplish a particular task. If the task is simple and short, the program will be relatively simple and short. If the task is large and intricate, the program will be relatively large and intricate. You can think of a program as any plan of action to attain a certain end.

You create a plan of action to accomplish your everyday tasks. As a computer programmer, you will create a plan of action to accomplish a task that a computer can do. But right now, from this point of view, in everyday life we are all programmers!

1.1 What Is Programming?

As we go about our daily lives, we encounter problems that are best attacked using a systematic plan of action. Often this plan involves a step-by-step process for resolving the problem. For example, suppose you’ve invited an out-of-town friend to your new house. To help make the trip go smoothly, you provide detailed instructions for getting to your house. In effect, you create a program to be carried out by your friend. Such a program might look like the following:

Take Interstate 80 West and exit at Springfield Road.
At the traffic light at the end of the off-ramp, turn right onto Springfield.
Go two miles on Springfield and make a left turn onto Midvale Street.
Travel three blocks on Midvale and then turn right onto Harvey Drive.
Go about one half block to my house at 456 Harvey Drive.
A General Problem-Solving Strategy

To create a suitable plan of action to solve a particular problem such as providing travel instructions for a friend, it is often useful to apply the following general problem-solving strategy:

1. Try to understand the problem completely. If you don’t fully understand the problem, it will be difficult, or even impossible, to create a viable plan to solve it.
2. Devise a plan of action to solve the problem. Provide precise step-by-step instructions for doing so.
3. Carry out the plan.
4. Review the results. Did the plan work? Did it solve the given problem?

Of course, at any stage of this process, you may see a flaw in what you’ve done and have to return to a previous step to reevaluate or modify things.

Now let’s apply this problem-solving strategy to another everyday programming problem you may encounter. Suppose you decide to take a trip to Paris. You will fly there, stay in a hotel, and explore the city. You don’t vacation in France very often, so you want to be sure that the trip happens without problems or unpleasant surprises. To ensure that everything goes smoothly, you make a list of the things you must do before you leave and then you check off the items as you do them. In other words, you create a program for planning your trip and then execute that program. Here’s how you might use the general problem-solving strategy just described as you make your travel preparations:

1. Understand the problem. First, you have to do some research about Paris. What airlines fly there and how much will a ticket cost? What hotels are available, where are they located, and what are their prices? What is the weather like this time of year? What kinds of clothes should you bring? How will you get around the city? You will probably investigate the public transportation system and compare it to renting a car. Do you speak French? If not, you will want to buy a French-English dictionary and maybe even some language tapes. You can probably think of many more things to research before flying across the Atlantic.

2. Devise a plan of action. At this point you need to create a pre-trip list of things to do. In effect, you need to create a program. The list might include the following items, although you can probably think of many more:
   - Get airline tickets and make hotel reservations
   - Be sure that your passport is in order
   - Buy euros
   - Listen to language tapes
   - Decide what to pack (this generates, in effect, a few subprograms—things you need to do first, before you can decide what to pack)
     - List items you want to take with you
Decide what to pack in your checked luggage and what to pack in your carry-on bag—but wait!—you don’t have a carry-on bag so you must...
- Shop for a new carry-on bag—another subprogram!
- Stop your newspaper and mail delivery while you’re gone
- Inform your family and friends of your itinerary

3. Carry out your plan. Before you leave on your trip, perform each of the tasks on your list. In other words, execute your program.

4. Review the results. Once you have performed the listed tasks (i.e., carried out your plan), but before you leave on your trip, it would be a good idea to review your preparations. For example, you would do the following:
- Check to see that the information on the tickets and reservations is correct
- Check that you’ve packed all the things you might need
- Imagine that you’re on your trip in order to determine if you’ve forgotten to make any necessary preparations

In the course of devising a plan of action, you may discover that you don’t have enough information—you don’t fully understand the problem. Or in carrying out the plan or reviewing the results, you may find that you have to change your plan. These modifications to previous steps are almost inevitable when applying our problem-solving strategy. We say that problem-solving is a cyclic process because we often return to the beginning or redo previous work before arriving at a satisfactory solution.

Creating Computer Programs: The Program Development Cycle

The general process for creating a computer program mimics the general problem-solving strategy that we have just outlined: understand the problem, devise a plan, carry out the plan, and review the results. When you solve a problem with a computer program, this strategy takes the following form:

1. Analyze the problem. Determine what information you are given, what results you need to get, what information you may need to get those results, and in general terms, how to proceed from the known data to the desired results.

2. Design a program to solve the problem. This is the heart of the program development process. Depending on how hard or complex the problem is, it might take one person a few hours or it might take a large team of programmers many months to carry out this step.

3. Code the program. Write statements (program code) in a particular computer language that implement the design created in Step 2. The result of this step is the program.

4. Test the program. Run the program to see if it actually solves the given problem.
This process of analysis, design, coding, and testing forms the core of what is known as the **program development cycle**. The word *cycle* is used here because as with the general problem-solving process, we often have to return to previous steps as we discover flaws in subsequent ones. We will discuss the program development cycle in more detail in Chapter 2.

In order to develop a program to solve a specific problem, you must know and understand programming concepts in general and a specific language in particular. In this book, we will concentrate on the first two steps of the program development cycle—especially the program design phase. Our goal is to introduce programming concepts and program design generically, without referring to a specific language. We will present some short examples in several programming languages (C++, Java, or Visual Basic); these examples are for illustration purposes only. The text focuses on the programming concepts and logic that are relevant to all programming languages.

One of the best things about learning to program is that even though there are many different programming languages, the basic concepts of programming are the same regardless of what language you use. In fact, once you understand these programming concepts and learn one programming language, it is fairly easy to learn a new language. But first things first. Here we concentrate on the basic building blocks of programming—the concepts. After you have learned these general ideas it will be easy to implement them and the entire program development process using a particular computer language.

**Self Check for Section 1.1**

1. List the steps in the general problem-solving strategy described in this section.
2. Provide a precise list of instructions for traveling from your school to your home.
3. List the steps in the program development cycle.
4. What is the significance of the word *cycle* in the term *program development cycle*?

### 1.2 Basic Programming Concepts

In the previous section, you planned a trip to Paris. One of the most important things you need to do is prepare for a very long plane ride. You want to be sure you have a lot of good music to listen to so you decide to download music from iTunes. But your funds are limited so you need to figure out how many songs you can buy with your limited resources. We will use this project as the basis for solving a simple programming problem and we will refer to this sample problem throughout the remainder of Chapter 1. First we will use this problem to illustrate the notions of data *input*, *constants*, and *variables*. Then we will return to this same problem in each of the next two sections when we discuss the concepts of *processing*, *output*, and types of *data*. 
**A Simple Program**

You want to develop a program that will help you quickly see how much money you are spending as you download music. Today, the price of downloading one song is 99 cents ($0.99). You need a program that will tell you how much the price of 8 or 10 or any number of songs will be. You know how to do this calculation by hand with pencil and paper or with a calculator. To prepare to write the computer program, you write down the following instructions for using a calculator to perform the computation:

- Enter the number of songs to be purchased into the calculator.
- Press the multiply (×) key.
- Enter 0.99.
- Press the equals (=) key.

The total cost will appear in the display.

This calculation will only be valid if a song costs 99 cents. However, if the price changes, the steps needed to find the total cost of one or more songs remains the same. The only thing that would need to change would be the cost of one song.

**The Music Purchase Program**

The instructions shown here are not very different from those in a computer program designed to perform this task. The Music Purchase program instructions, or statements, might look like the following:

- Input the number of songs you wish to download today: Songs
- Compute the cost of the purchase:
- Set DollarPrice = 0.99 * Songs
- Display the value of DollarPrice

The asterisk (*) in the next-to-last line represents multiplication.

In this book, we will refer to a list of instructions like the one shown above as a program, but it is not a computer program in the strict sense of the word. Computer programs are written in programming languages and follow an exact syntax. The syntax of a computer language is its rules of usage. If you don’t use correct syntax, your program won’t work. For example, some computer languages use a semi-colon (;) to tell the computer that a statement is ended. If you type a comma (,) or a colon (:) in that language by accident, you will get an error message or the program may not work at all. However, the semi-colon may not be the end of statement indicator in another language. Each language has its own syntax and you must learn the syntax in order to write programs in a specific language.

The list of statements shown in the Music Purchase statements is referred to as pseudocode. The prefix pseudo (pronounced sue-dough) means not real. Examples 1.1 and 1.2 show actual computer programs in BASIC and C++ that would solve the currency conversion problem.

Java is a very popular high-level language. Example 1.1 shows Java code that performs the currency conversion.
Example 1.1 Using Java to Code the Music Purchase Program

Java Program Code

```java
public static void main(String[] args) {
    int Songs = 0;
    float DollarPrice = 0.0;
    Scanner scanner = New Scanner(System.in);
    println("Enter the number of songs you wish to purchase today.");
    Songs = scanner.nextInt();
    DollarPrice = 0.99 * Songs;
    println(DollarPrice);
}
```

- Lines 1 and 2 are how Java says start a program.
- Line 3 tells the computer that there will be a number used in this program that will always be a whole number (an integer) and its value will be stored in a variable named Songs.
- Line 4 tells the computer that there will be a number used in this program that will have a decimal part (a floating point number) and its value will be stored in a variable named DollarPrice.
- Line 5 is a Java instruction that allows input from a keyboard.
- Line 6 displays a request on the screen so the user knows to enter the number of songs to be purchased.
- Line 7 takes the value the user enters and stores it in the variable named Songs.
- Line 8 does the computation. It multiplies Songs by 0.99 and stores the result in the variable named DollarPrice.
- Line 9 displays the cost of the purchase.
- Line 10 is how Java says the end.

Example 1.2 does the same thing as Example 1.1, but uses C++ program code.

Example 1.2 Using C++ to Code the Music Purchase Program

C++ Program Code (slightly simplified)

```cpp
void main(void) {
    int Songs;
    float DollarPrice;
    cout << "Enter the number of songs you wish to purchase today."
    cin >> Songs;
```
Let’s compare this code to the previous Java code. Once we sift out all the words that are specific to the language, we can see that the actual programming logic is the same for both languages.

- Lines 1 and 2 are how C++ says *start a program*.
- Line 3 tells the computer that there will be a number used in this program that will always be a whole number (an integer) and its value will be stored in a variable named *Songs*.
- Line 4 tells the computer that there will be a number used in this program that will have a decimal part (a floating point number) and its value will be stored in a variable named *DollarPrice*.
- Line 5 displays a request on the screen so that the user knows he should enter the number of songs he wishes to purchase.
- Line 6 takes in the value that the user enters and stores it in the variable named *Songs*.
- Line 7 does the computation. It multiplies the *Songs* by 0.99 and stores the resulting number in a variable named *DollarPrice*. Note that this line does exactly the same thing as line 8 in the Java program, and it virtually looks the same. It is the essence of this program! The actual calculation, which is mainly what we wanted when we wrote this program, uses the same logic in both languages.
- Line 8 does the same thing as line 9 in Java—it displays the resulting cost of the purchase.
- Lines 9 and 10 are how C++ says *the end*.

Our simple program illustrates the basic structure of most computer programs. They input data, process it, and output the results. Here the word *data* refers to the numbers, words, or more generally, any collection of symbols that is manipulated by a program.

### Data Input

The input operation in a program transmits data from an outside source to the program. Often this data is typed at the computer keyboard by the person using the program. For example, the `cout` statement of the C++ program shown in Example 1.2 causes execution of this program to pause and displays the text *Enter the number of songs you wish to purchase today:* on the screen. The program waits until the user types something. In computer lingo, the program *prompts* the user to enter a value. Different programming languages have different prompts—symbols or methods of telling the user to enter data. At this point for example, the user might type `78`, indicating that he wants to buy 78 songs. In the C++ program, the `cin` statement causes the program execution to resume, and `Songs` takes on the value 78. These actions are referred to as *entering data from the keyboard.*
The Input and Write Statements

In this book, we will use a statement that begins with the word Write to display messages and other information on the screen. We will use a statement that begins with the word Input to allow the user to enter data from the keyboard. If we follow the word Input with a variable, then it will be assumed that the value the user types will be stored in that variable. Therefore, although each language has its specific rules and instructions, we will use a general form. Our Music Purchase program, written in our own general pseudocode, is shown in Example 1.3.

Example 1.3 Music Purchase Program Pseudocode

```
Write "Enter the number of songs you wish to purchase: "
Input Songs
Set DollarPrice = 0.99 * Songs  \(\text{(computes the total cost)}\)
Write DollarPrice
```

The first statement (Write) causes the message within the quotation marks to appear on the screen. The second statement (Input) causes execution to pause. Now the user clearly understands what data must be entered. When execution resumes, the Input statement also causes Songs to take on the value entered. After the calculation is completed, the next Write statement will cause the result to be displayed on the screen.

The last Write statement in Example 1.3 says Write DollarPrice. What will be displayed? The word DollarPrice won’t be displayed, but the value of DollarPrice will be.

In other words, if the user wanted to download three songs, he would enter 3 after the Write prompt. The calculation would multiply 3 by 0.99 and the result would be 2.97. Three songs downloaded from iTunes will cost $2.97. So DollarPrice now has the value 2.97. The statement Write DollarPrice will cause 2.97 to be displayed on the screen.

Use Input Prompts

When you want the user to input data to a program, you should always provide a prompt indicating that data is needed and you should explain what kind of data is required. If you don’t use a prompt, then the user will not know what kind of data to enter or even be aware, in most cases, that execution has paused!

If you want the user to enter several data items, you can use several Input statements, each preceded by a prompt. In some cases however, a single input is enough. For example, suppose that at some point in a program you want the user to enter three numbers. This can be done by the following pair of statements:

```
Write "Enter three numbers."
Input Number1, Number2, Number3
```

In this case, Number1 takes on the first value entered, Number2 the second value, and Number3 the third value.
Other Forms of Input
Input from the keyboard is very common, but data may be input into a program by other means as well. In some programs the user inputs information by clicking or moving the mouse. For example, to draw a straight line in a graphics program, the user might click a symbol representing a line and then click the locations of the line’s endpoints. Another common form of input doesn’t involve the user at all—data can be transmitted to a program from a data file stored on disk (as discussed in Chapter 8).

Program Variables and Constants
We have been using variables in all the examples up to now, but it’s important to discuss variables—what they are, how to use them, how to name them—in more detail. Variables are such an integral part of all computer programs, it’s impossible to describe a sample program without referring to one, as we have seen in the previous examples. On the other hand, it’s almost impossible to talk about a variable without putting it in some context—in this case, a bit of program code. So you know something about variables already.

Let’s return to our little Music Purchase program for a moment. If we had not used a variable, we still could have written a program to calculate the cost of three songs at $0.99 per song. A computer can do this conversion for us—a calculator does it all the time. We enter 3 into a calculator, press the multiplication sign, enter 0.99, press the = key, and get 2.97 as the result. If we wanted to buy 46 songs, we could do the same thing with our calculator. Because 46 * 0.99 = 45.54, we would know that it would cost $45.54 to download 46 songs from iTunes at $0.99 per song. But that’s not what programming is all about. A computer program should be written so that the user doesn’t have to repeat the steps to get another result. The Music Purchase program we wrote will find the price of 1 song, 12 songs, 836 songs, or however many songs we decide to buy because we use variables instead of actual values.

When we write a program, most of the time we don’t know the actual numbers or other data that the user will enter while running (executing) the program. Therefore, we assign the input data to a program variable. A variable is called a variable because it can vary! It is a quantity that can change value during the execution of a program. Whenever we need to refer to that data in a subsequent program statement, we simply refer to its variable name. At this point the value of the variable—the number or other data value it represents—will be used in this statement.

In the Music Purchase program, the input variable is Songs. If in running the program the user enters 100, this value is assigned to Songs. Then if later in the program the expression $0.99 * Songs occurs, the computer will multiply 0.99 by 100. Note that in the expression above, the number 0.99 cannot be changed during program execution; it is referred to as a program constant.

Sometimes you may want to give a constant a name. When you do, it is called a named constant. Let’s say you are writing a program to calculate the total cost of purchases made by customers for a small online business. First your program will add the purchases and then it will calculate the sales tax. Your program might use
this tax rate value in several places. You might want to display a subtotal, the tax rate, and tax amount on a shopping cart page and later, display these amounts along with the shipping cost.

If you enter the value of the tax as a number (a constant) and the tax rate changes, you would have to go through your program code and change it each time the constant is typed into the code. On the other hand, if you give the tax rate a variable name and use this variable name wherever you need to use the tax rate in a calculation, when the tax rate changes you would only have to change the value in one place. You would simply change the one line in the code where you assigned the constant value to the variable name and everywhere in the code where that variable was used, the new value would be in effect.

Variables Names
As the programmer, you choose the name of the variable. However, you need to know what types of names are acceptable and what types are unacceptable. The rules are different for different languages, but the following rules are the same for all:

- All variable names must be one word.
- Underscores are allowed and hyphens are usually allowed, but spaces are never allowed.
  - Miles_traveled is fine but Miles traveled is not
  - Cost-per-gallon is fine but cost per gallon is not
- Variable names can be long; in fact, many languages allow more than 200 characters. But remember, you will have to type the variable name throughout the program. If you make the name too long or too complicated, you are asking for another reason to make an error.
  - The variable name the_Tax_Rate_On_Clothing_Bought_for_Children_under_Six is a lot harder to remember than tax_rate_1.
- Many programmers use uppercase letters to distinguish one word from another within a single name.
  - MilesTraveled works as well as Miles_Traveled and is easier to type
  - CostPerGallon works as well as Cost_Per_Gallon and is easier to type
- Most languages allow numbers to be part of variable names, but variable names should never start with a number.
  - TaxRate_1 is fine but 1_TaxRate is not
  - Destination2 is fine but 2Destination is not

Variable Names Should Be Meaningful!
If you name your variables the perfectly acceptable names of variableNumber_1, variableNumber_2, variableNumber_3, and so forth, you will soon find that you are spending most of your time trying to remember what each variable represents, instead of fine-tuning your program.

It’s best to name variables with the shortest possible name that allows you some meaningful representation and does not conflict with any of the rules stated here or any of the rules of your specific programming language. For example, we could
have used \( x \) or \( PP \) or even \( George \) to represent the number of songs to be purchased in Music Purchase program. However, to minimize confusion you should give meaningful names to variables. So instead of \( Songs \), we might have used \( NumberOfSongs \) or perhaps \( MySongs \). But we must be consistent! If you use \( NumberOfSongs \) in the input statement, then the name of this variable in the next statement, currently

\[
\text{Set } DollarPrice = 0.99 \times Songs
\]

must also be changed to

\[
\text{Set } DollarPrice = 0.99 \times NumberOfSongs
\]

What's Really Going On with Variables in the Computer?

Technically, a program variable is the name for a storage location in the computer's internal memory, and the value of a variable is the contents of that location. It might help to think of the storage locations as mailboxes. Each variable can be thought of as the name printed on a certain mailbox, and the value of a variable as its contents. For example, the following is a picture of what the computer's memory might look like after the Input statement in the Music Purchase program is executed and the user wants to download 78 songs:

<table>
<thead>
<tr>
<th>Songs</th>
<th>DollarPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td></td>
</tr>
</tbody>
</table>

Notice that the DollarPrice mailbox is empty, indicating that at this point in the program DollarPrice has not yet been assigned a value. At the end of the program, after the conversion has been made, the picture would look like the following:

<table>
<thead>
<tr>
<th>Songs</th>
<th>DollarPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>77.22</td>
</tr>
</tbody>
</table>

If you run the program a second time to purchase a different number of songs, the picture will change. The contents of the Songs mailbox will be replaced by the new number of songs to be purchased, and after the calculation is made the contents of the DollarPrice mailbox will be replaced by the new value. This process is discussed in more detail in Section 1.3.

Self Check for Section 1.2

1.5 What are the three components that make up the basic structure of most computer programs?

1.6 Write a pair of statements, the first of which is an appropriate prompt to input a temperature in degrees Fahrenheit (use Temperature for the variable).
1.7 Suppose a program is to calculate the final (maturity) value of an investment. You will be given the amount invested, the rate of interest, and the length of time that the money is invested.

a. What data must be input to this program?

b. Give reasonable names for each of the input variables.

c. Give Input and Write statements that prompt for and input the data for this problem.

1.8 What (if anything) is wrong with each of the following variable names?

a. Sales Tax

b. 1_2_3

c. TheCowJumpedOverTheMoon

d. OneName

1.3 Data Processing and Output

In this section, we will continue the discussion of the fundamental building blocks that make up a program: input, processing, and output. In Section 1.2, we concentrated on data input; here we will discuss the concepts of processing and output.

Processing Data

Let’s return to the current version of the Music Purchase program, which was introduced in Section 1.2:

Write "Enter the number of songs you wish to purchase: ">
Input Songs
Compute the total cost:
Set DollarPrice = 0.99 * Songs
Write DollarPrice

The Set Statement

After the user has input the value of Songs, the instruction

Set DollarPrice = 0.99 * Songs

is executed. This statement comprises the processing part of the program. It accomplishes two things as follows:

1. It multiplies the value of Songs (the number of songs the user wishes to buy) by 0.99 (the cost of one song). Notice that we use the asterisk (*) for the multiplication symbol.

2. It assigns the resulting value of the expression on the right of the equals sign to the variable, DollarPrice, on the left. The value of Songs does not change. For this reason, it is called an assignment statement. For example, if the value of Songs is 100 when this statement is executed, then the expression on the right side is computed to be 99.00 and is assigned to the variable DollarPrice.
Assigning and Reassigning Values to Variables

What happens if a variable that already has a value is assigned a new one? For example, suppose a program has a variable named NumberX and contains the following statements:

```
Set NumberX = 45
Set NumberX = 97
```

In such a case, NumberX first contains the value of 45, and in the next statement the value of 45 is replaced by 97. NumberX will keep the value of 97 until something else in the program reassigns it a new value. In terms of storage locations, when the second of the two assignment statements is executed the value currently stored in location NumberX (45) is erased and the new value (97) is stored in its place.

Assignment statements can sometimes look a little strange. For example, a common program statement looks as follows:

```
Set Counter = Counter + 1
```

Although this looks confusing, if you look carefully, it’s easy to see what happens here. First the right side is evaluated, and 1 is added to the current value of the variable Counter. Then this new value is assigned to the variable on the left, Counter. The net result is to add one to the previous value of Counter. So if the value of Counter were equal to 23 prior to execution of this statement, its value would be 24 afterward. Using a variable in this manner is very common in programs; therefore, it’s very important to understand this single statement.

Operations on Data

The * symbol used to denote multiplication is an example of an arithmetic operator. Almost all programming languages use at least four basic arithmetic operators—addition, subtraction, multiplication, and division. Some languages contain other arithmetic operators, such as exponentiation (taking a number to a power) and modulus.

The modulus operator may seem odd at first, but as you start to write programs you will see its many uses. The modulus operator is a positive integer that returns the remainder after dividing one number by another. The symbol commonly used to denote the modulus operator is the percent sign (%) or the abbreviation MOD. We will use the % to designate the modulus operator in this text. Some examples using the modulus operator appear in Example 1.4.

Example 1.4 The Modulus Operator

1. What is $15 \% 2$?
   - $15$ divided by $2 = 7$ with a remainder of $1$ so $15 \% 2 = 1$
2. What is $39 \% 4$?
   - $39$ divided by $4 = 9$ with a remainder of $3$ so $39 \% 4 = 3$
3. What is $21 \% 7$?
   - $21$ divided by $7$ is $3$ with no remainder so $21 \% 7 = 0$
Table 1.1 gives examples of the six arithmetic operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Computer Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>$2 + 3 = 5$</td>
</tr>
<tr>
<td>Subtraction</td>
<td>–</td>
<td>$7 - 3 = 4$</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>$5 * 4 = 20$</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>$12 / 3 = 4$</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>^</td>
<td>$2 ^ 3 = 8$</td>
</tr>
<tr>
<td>Modulus</td>
<td>%</td>
<td>$14 % 4 = 2$</td>
</tr>
</tbody>
</table>

For example, to convert a temperature in degrees Fahrenheit to degrees Celsius we use the following formula:

$$C = \frac{5(F - 32)}{9}$$

However, in a programming language (and in this book), the formula is written as follows:

$$C = 5 * (F - 32) / 9$$

In this example, the $C$ is the variable name that represents the value of degrees in Celsius and the $F$ is the variable name that represents the value of degrees in Fahrenheit. To determine the value assigned to the variable $C$ when the value of $F$ is 77, we proceed as follows, substituting the value 77 for $F$:

$$C = 5 * (77 - 32) / 9$$
$$= 5 * (45) / 9$$
$$= 225 / 9$$
$$= 25$$

**Hierarchy of Operations**

Notice that if the parentheses were missing in the last example we would get a different result: $5 * 77 - 32 / 9 = 385 - 32 / 9$, which is approximately 381.4.

The reason the two answers differ is due to the **hierarchy of operations**. The rules of arithmetic tell us that the order in which arithmetic operations are performed (i.e., their hierarchy) is as follows:

1. Perform the operations in parentheses (from the inside out, if there are parentheses within parentheses)
2. Perform exponentiations
3. Do multiplications, divisions, and modulus (from left to right if there is more than one)
4. Do additions and subtractions (from left to right if there is more than one)

Unless you specify something different, the computer will apply the hierarchy of operations to any mathematical expression in a program. The best way to write a
mathematical expression is to put parentheses around parts of the expression that you want evaluated together. You will never get an error if you use a set of parentheses where none is needed, but you may get an error if you omit parentheses when a set should be included. Examples 1.5 and 1.6 demonstrate how parentheses can make a big difference in the answer to even the simplest mathematical calculations.

Example 1.5 Using the Hierarchy of Operations

Given the following arithmetic expression: \(6 + 8 / 2 \times 4\)

a. Evaluate without any parentheses:
\[
6 + 8/2 \times 4 = 6 + 4 \times 4 = 6 + 16 = 22
\]

b. Evaluate with parentheses:
\[
6 + 8/(2 \times 4) = 6 + 8/8 = 6 + 1 = 7
\]

c. Evaluate with different parentheses:
\[
(6 + 8)/2 \times 4 = 14/2 \times 4 = 7 \times 4 = 28
\]

d. Evaluate with two sets of parentheses:
\[
(6 + 8)/(2 \times 4) = 14/8 = 16/8 = 1.75
\]

It’s obvious that parentheses can make quite a difference in your results!

Example 1.6 One More Example for Emphasis

Given the following arithmetic expression: \(20/5 + 5 \times 4 - 3\)

a. Evaluate without parentheses:
\[
20/5 + 5 \times 4 - 3 = 4 + 20 - 3 = 21
\]

b. Evaluate with parentheses:
\[
20/(5 + 5) \times 4 - 3 = 20/10 \times 4 - 3 = 2 \times 4 - 3 = 8 - 3 = 5
\]

c. Evaluate with more parentheses:
\[
(20/(5 + 5) \times 4) - 3 = (20/(5 + 20)) - 3 = 20/25 - 3 = 0.8 - 3 = -2.2
\]
d. Evaluate with parentheses:

\[
(20 / 23 \% 6) + (5 * (4 - 3)) = 4 + (5 * 1) \\
= 4 + 5 \\
= 9
\]

By now you should understand why it’s important to know the hierarchy of operations and write arithmetic expressions in your programs with great care.

### Computers Have Limits

It’s generally accepted that computers can do calculations very quickly, very accurately, and normally without complaint. But it’s important to realize that computers can only handle a finite range of numbers and finite amounts of data. The actual range of values that a computer can process varies from computer to computer, from language to language, and from programmer to programmer (depending on how variables are declared). But even the biggest, fastest computer cannot process any and all numbers. For example, in C++ the largest value a 10-byte number can take on is \(10^{4952}\). This is 10 multiplied by itself 4,952 times. This is certainly an enormous number, but it’s smaller than \(10^{5000}\).

We will briefly discuss the exact ways the computer stores numbers in Sections 1.4 and 1.5. You should know, without going into too much detail, that the computer can process a finite range of numbers and therefore, is only as precise as its range allows. This is not a problem for most normal uses, but care must be taken in high-level computations.

### Data Output

A program’s output is data sent by the program to the screen, printer, or another destination such as a file. The output normally consists, at least in part, of the results of the program’s processing component. In the Music Purchase program the output is produced by the following statement:

Display the value of DollarPrice

### The Write Statement Revisited

Recall from Section 1.2 that we used a `Write` statement to display messages on the screen. We will also use this statement to display the values of variables on the screen. So from now on, we will write a statement like the `Display` statement above as follows:

Write DollarPrice

When this statement is executed, the current value of the variable DollarPrice is displayed and then the cursor, which indicates the current position in the text, moves to the beginning of the next line on the screen. For example, if the values of DollarPrice and Songs are 9.90 and 10 respectively, then the following pair of statements:

Write DollarPrice
Write Songs
produces the output

```
9.90
10
```
on the screen.

When outputting the values of variables, it is often useful to display text on the same screen line. For example, if the user inputs 10 for the number of songs he wishes to buy in the Music Purchase program as we have written it so far, the only output that would appear on the screen would be **9.90**. It would be more informative to display the following:

```
The cost of your purchase is 9.90
```

We will accomplish this by using the following statement:

```
Write "The cost of your purchase is " + DollarPrice
```

This statement displays the text included between the quotation marks followed by the value of the variable. As an alternative we could use the following statement:

```
Write "The cost of your purchase is " + DollarPrice + "dollars."
```
to produce the following output:

```
The cost of your purchase is 9.90 dollars.
```

Notice that the text to appear on the screen is enclosed in quotes. In this book, text and variables are separated by `+` symbols, which are not displayed as part of the output. Each programming language has its own special statements to create the kinds of screen output described above.

**Annotate Your Output**

If the output of your program consists of numbers, you should also include explanatory text as output. In other words, **annotate** your output as demonstrated in Example 1.7 so that the user will understand the significance of these numbers.

---

**Example 1.7 Annotating Your Output**

Let's assume that you have written a program to compute the temperature in degrees Celsius when degrees Fahrenheit is input. In very general terms, your program would follow the following logic:

```
Write "Enter temperature in degrees Fahrenheit to convert to Celsius:"
Input DegreesFahrenheit (get degrees to be converted)
DegreesCelsius = 5*(DegreesFahrenheit - 32)/9 (calculate)
Write DegreesCelsius (output result)
```

If a user inputs 77 degrees Fahrenheit, the answer would be 25 degrees Celsius. The following shows the screen output:

```
25
```
It would be far better to explain to the user what the result means by outputting some explanatory text. The improved program would look as follows:

```
Write "Enter temperature in degrees Fahrenheit to convert to Celsius:"
Input DegreesFahrenheit (get degrees to be converted)
Set DegreesCelsius = 5*(DegreesFahrenheit - 32)/9 (calculate)
Write DegreesFahrenheit + " degrees Fahrenheit" (output result)
Write "converts to " + DegreesCelsius + " degrees Celsius" (output result)
```

Now the output would look like the following and would make more sense to the user:

77 degrees Fahrenheit
converts to 25 degrees Celsius.

Example 1.8 uses all the ideas presented so far in this chapter.

**Example 1.8 Putting It All Together**

We will input two numbers and store their values in variables named `Number1` and `Number2`. Then we will display the average of the two numbers. This calculation is done by adding the numbers and dividing the sum by two.

```
1  Write "Enter two numbers."
2  Input Number1
3  Input Number2
4  Set Average = (Number1 + Number2)/2
5  Write "The average of"
6  Write Number1 + " and " + Number2
7  Write "is " + Average
```

The following shows what happens in the program shown in Example 1.8:

- Line 1 contains the `Write` statement, which is a prompt for the `Input` statement that follows it.
- Lines 2 and 3 are the `Input` statements. When the user types in the first number, its value is stored in the variable named `Number1`. When the user types in the second number, its value is stored in the variable named `Number2`.
- Line 4 now calculates and stores the value of the variable named `Average`. The value of `Average` is set to be the sum of the values of `Number1` and `Number2` divided by 2. Note the use of parentheses here. If for example, `Number1 = 10` and `Number2 = 12`, then `Average = (10 + 12)/2`, which is 11. But without the parentheses, the computer would follow the hierarchy of operations and do the division before the sum. Then we would have `Average = 10 + 12/2`, which would be `10 + 6` or 16. To get the result we want—the average of the two numbers—we must use parentheses to show that we want the addition to be done before the division.
- Lines 5, 6, and 7 produce the output. Line 5 outputs straight text. The
Self Check for Section 1.3

1.9 If the temperature is 95 degrees Fahrenheit, use the formula in this section to find the resulting temperature in degrees Celsius.

1.10 If \( X = 2 \) and \( Y = 3 \), give the value of each of the following expressions:
   a. \((2 \times X - 1) ^ 2 + Y\)
   b. \(X \times Y + 10 \times X / (7 - Y)\)
   c. \((4 + (2 ^ Y)) \times (X + 1) / Y\)
   d. \((19 \% Y) \times Y / X \times 2\)

1.11 If \( \text{Number} \) is a variable and has the value 5 before the execution of the following statement:
   \[
   \text{Set Number} = \text{Number} + 2
   \]
   what is the value of \( \text{Number} \) after the execution of this statement?

1.12 If \( \text{Songs} = 100 \) and \( \text{DollarPrice} = 99.00 \), write statements that use these variables to produce the following output on the screen:
   a. 100 songs will cost $99.00
   b. The number of songs to be downloaded is 100
      The cost for this purchase in dollars is 99.00

1.13 Write a program (like the Music Purchase program of this section) that inputs a temperature in degrees Fahrenheit and outputs the corresponding temperature in degrees Celsius. Use \( \text{DegreesF} \) and \( \text{DegreesC} \) for your variable names. (Hint: use the formula given earlier in this section.)
1.4 Character and String Data

Computer languages make use of two fundamental types of data, or data types: numeric data and character string (or alphanumeric) data. Numeric data can be further divided into two major types—integer and floating point—and both of these are discussed in Sections 1.5 and 1.6. In this section, we concentrate on the character string data type.

Loosely speaking, a character is any symbol that can be typed at the keyboard. This includes the letters of the alphabet, both uppercase and lowercase, numbers, punctuation marks, spaces, and some of the extra characters that a computer keyboard contains, such as the pipe key (|), the various types of brackets (curly brackets { } or square brackets [ ]), and other special characters like $, &,<,>, and so forth. A character string (or more simply, a string) is a sequence of characters. In most programming languages, strings are enclosed in quotation marks. We will follow this practice in this book—in fact, we already have. For example, the Music Purchase program contains the following statement:

Write "Enter the number of songs you wish to purchase:"

A single character is also considered to be a string, so "B" and "g" are strings. But a string may be void of any characters; if this is the case it's called the null string (or empty string) and is represented by two consecutive quotation marks ("""). The length of a string is simply the number of characters in it. For example, the string "B$? 12" has length 6 because the space between the ? and 1 counts as a character. A single character, such as "Y", has length 1 and the null string has length 0.

The Declare Statement

As you have learned, a variable is a name for a storage location in the computer's memory. For example, if your program contains a variable named Color (that you plan to use to store the user's favorite color), there will be a storage location inside the computer set aside to hold its value. When you ask the user to type in his or her favorite color, that information is stored in the location which you have named Color. Therefore, before you begin to use a variable, you must explain to the computer that you want a storage location set aside and you want to give that location a name. In other words, you must declare your variable.

Different programming languages use different syntax to declare variables. In this text we use the word Declare. Before using any variables in our pseudocode, they must be declared with a statement that includes the keyword Declare, the variable name, and the variable type. In this section you will learn about the Character or String data type and, in Sections 1.5 and 1.6, you will learn about the two numerical data types.

A Declare statement, in this text, takes the following form:

Declare VariableName As DataType
The Character and String Data Types

Most programming languages contain a Character data type. A single character takes little storage space in the computer’s memory and characters are easy to manipulate. There are some times when it is most efficient to store a variable as a Character data type, but you must be certain that only a single character will be stored in that variable. For example, if a user is required to type in either the letter Y or the letter N in response to a yes-or-no question, then the Character data type is most appropriate. More often, however, we will use the String data type for non-numeric data. In this book, we will use a statement such as:

```
Declare Response As Character
```

to declare a variable (in this case the variable is named Response) to be of Character type.

Most programming languages such as Visual Basic and C++ contain a String data type. In these languages, we can declare variables to be of String type in an appropriate statement within the program. In other languages, strings are constructed as arrays of characters. In this book, to define a string variable we will use a statement such as the following, in which the variable is UserName:

```
Declare UserName As String
```

Operating on Strings

Like numbers, strings can be input, processed, and output. We will use Input and Write statements to input and display the values of character and string variables in Example 1.9.

Example 1.9 Using String Variables

If UserName is declared to be a String variable, then the statement

```
Input UserName
```

allows the user to enter a string at the keyboard and then assigns that string to the variable UserName.

The statement

```
Write UserName
```

displays the value of UserName on the screen.

Any data stored in a string variable is treated as text. Therefore, if you declare a variable named ItemNumber as a String variable and then assign ItemNumber = "12" you cannot do any mathematical operations on the variable.

In Section 1.3, we introduced six arithmetic operators: addition, subtraction, multiplication, division, modulus, and exponentiation. Each of these operators acts on two given numbers to produce a numeric result. Many programming languages include at least one string operator, concatenation, which takes two strings and joins them to produce a string result. The symbol that is often used to concatenate two strings is the plus sign, +. For example, if String1 = "Part" and String2 = "Time", then the statement
Set `NewString = String1 + String2` assigns the string "PartTime" to the string variable `NewString`. In other words, the value stored in the variable `String1` is **concatenated** with the value stored in `String2` to form the new string, which is now stored in the variable named `NewString`.

The `+` sign is used to represent addition and concatenation in a computer program. However, the computer will never be confused by this. If the variables in a statement are numeric, the `+` sign will signify addition. If the variables are string data, the `+` sign will represent concatenation.

What do you think the computer would display if you used the `+` operator on a string variable and a numeric variable? For example, let's assume that `ItemName` and `TextString` are declared as string variables and `ItemCost` is a number. Let's assume the following values are inputted for these variables:

```
ItemName = "Cashmere sweater "
TextString = "will cost $ 
ItemCost = 125
```

Then the output, given the following statement that uses the concatenation operator twice, would be as shown:

**Statement:** `Write ItemName + TextString + ItemCost`

**Display on Screen:** Cashmere sweater will cost $ 125

The program shown in Example 1.10 illustrates concatenating string variables.

**Example 1.10 Concatenating String Variables**

The program below inputs a first and last name from the user as string variables, makes use of the concatenation operator to create a string of the form Last name, First name, and then displays this string.

```pascal
1 Declare FirstName As String
2 Declare LastName As String
3 Declare FullName As String
4 Write "Enter the person's first name: "
5 Input FirstName
6 Write "Enter the person's last name: 
7 Input LastName
8 Set FullName = LastName + ", " + FirstName
9 Write "The person's full name is: " FullName
```

Let's run through the execution of the program:

- Lines 1, 2, and 3 declare our three variables, `FirstName`, `LastName`, and `FullName`, to be of the string data type.
- Line 4 contains the first `Write` statement, which is an input prompt. It displays a message on the screen asking the user to input a first name.
- Line 5 is the first `Input` statement, which assigns the text (the string) entered by the user to the variable `FirstName`.  

---

• Lines 6 and 7 contain the next Write and Input statements, which prompt for and assign the value of the variable LastName.

• In line 8, when the assignment (Set) statement is executed, two things happen. First, the right side is evaluated. The value of LastName (a string input by the user) is concatenated with the string consisting of a comma and a blank space. Then, that string is concatenated with the first name entered by the user. Second, this new string is assigned to the variable FullName.

• Finally, in line 9 the value of the string variable FullName is displayed on the screen. Note that the quotation marks surrounding the text are not displayed. To display quotation marks on the screen, you must use the rules specific to each programming language to do this.

If the user entered FirstName = "Sam" and LastName = "Smith" the result is FullName = "Smith, Sam" and the characters shown on the screen are:

The person's full name is: Smith, Sam

Self Check for Section 1.4

1.14 What is the difference between the Character data type and the String data type?

1.15 What would be the result of the following operation, given that the variables named JackOne and JillTwo are of String type?
Set JackOne = "3"
Set JillTwo = "5"
Write JackOne + JillTwo

1.16 What would be the result of the following operation, given that the variable named JackOne is of String type and JillTwo is of Character type?
Set JackOne = "Jackie"
Set JillTwo = "J"
Write JackOne + JillTwo

1.17 T  F The + operator is used to mean either addition or concatenation.

1.18 Suppose String1 is a string variable with String1 = "Step". What would be displayed if the following program were run?
Set GetThere = String1 + "-by-" + String1
Write GetThere

1.5 Integer Data

Most programming languages allow at least two types of numeric data to be used in programs: **integers** and **floating point numbers**. These types of data are stored differently and they take up different amounts of space in the computer’s memory. Integer data consists of all the whole numbers, negative, zero, and positive. Floating point data consists of all numbers that include a decimal part. In this sec-
tion, we will discuss integer data and how this data type is stored in the computer. Floating point data is discussed in the next section.

In programming, an integer is a positive, negative, or zero whole number written without using a decimal point. For example, the numbers 430, -17, and 0 are integers. While the number 8 is an integer, the number 8.0 is not considered an integer in programming. It has a decimal part, even though the value of the decimal part is 0. Because they are relatively simple numbers, integers take up relatively little storage space in the computer’s memory.

As we have seen in Section 1.4, before a variable is used in a program it should be declared to be of a particular type, which in this textbook is accomplished by using the Declare statement. Specifically, to declare the variable Number to be of type Integer, we will use the following statement:

Declare Number As Integer

Declaring Data Types in C++ and Visual Basic

In most programming languages, a variable can or must be declared (defined) to be of a particular type by placing the proper statement within the program. For example, to declare the variable Number to be of Integer type in C++, we use the following statement:

```cpp
int Number;
```

and in Visual Basic the statement

```vb
Dim Number As Integer
```

does the same thing. When the program is run, this statement tells the computer to set aside a storage location of the proper size for the integer variable Number.

In some languages, a variable can simultaneously be declared as Integer type and assigned a value. When we assign a variable an initial value, we say that we initialize the variable. For example, in C++, to declare Number to be an integer and assign it an initial value of 50, we use the following:

```cpp
int Number = 50;
```

Operations on Integers

The six arithmetic operators (+, -, *, /, %, ^) discussed in Section 1.3 may be used on calculations with integers. The result of adding, subtracting, multiplying, or taking the modulus of a pair of integers is another integer. The result of exponentiation (when we raise an integer to a power that is a positive integer) is also an integer. However, dividing one integer by another may result in a non-integer value as the following shows:

- The results of these five operators performed on two integers will always be another integer:
  
  $5 + 2 = 7 \quad 5 - 2 = 3 \quad 5 \times 2 = 10 \quad 5^2 = 25 \quad 5 \% 2 = 1$

- The result of the division operator on two integers is not normally an integer: $5 \div 2 = 2.5$
Next, we explain why we say that the results of the division operator on two integers is not normally an integer. When you divide two integers, the result may or may not be an integer. For example, \(24 \div 8 = 3\) and \(3\), of course, is an integer. But \(22 \div 8 = 2.75\) and 2.75 is not an integer. Computer programming languages have their own ways of dealing with this situation. When the division operator \((/\)) is applied to two integers, and the result is not mathematically an integer, the language may treat the result as an integer or not, depending on the language. Some languages will truncate the fractional part, which means that the fractional part is simply discarded.

For example, suppose that \texttt{Number1} and \texttt{Number2} have been declared to be integers and assigned the values 22 and 8, respectively. Then we have the following:

- In Visual Basic the result of computing \texttt{Number1/Number2} is 2.75.
- In C++ and Java \texttt{Number1/Number2} is the integer 2. In these languages the result of dividing 22 by 8—2.75—is truncated to the integer 2. Its fractional part, .75, is discarded.

In this book, we take the latter approach, so that \(22/8 = 2\).

### The Binary Number System

The number system we all use in everyday life is the decimal number system. It developed from the fact that we have ten fingers so it is natural for us to count by tens. As you will soon see, \(10\) is a key number in the decimal system, and we refer to this system as a base 10 system. The number 23, for example, is 2 tens and 3 ones. The number 4,657 is 4 thousands, 6 hundreds, 5 tens, and 7 ones.

Remember that 1 is \(10^0\), 10 is \(10^1\), \(100\) is \(10^2\), \(1,000\) is \(10^3\), and so on. Each column in the decimal system represents the base of the system \((10)\) raised to a power. The one’s column represents 10 raised to the 0 power, the ten’s column represents 10 raised to the 1st power. The hundred’s column represents 10 raised to the 2nd power, the thousand’s column is 10 raised to the 3rd power, and so on. Table 1.2 shows what our decimal system really represents.

<table>
<thead>
<tr>
<th>Power</th>
<th>Base 10 Value</th>
<th>Column Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>ones</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>tens</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>hundred-thousands</td>
</tr>
<tr>
<td>3</td>
<td>1,000</td>
<td>thousands</td>
</tr>
<tr>
<td>4</td>
<td>10,000</td>
<td>ten-thousands</td>
</tr>
<tr>
<td>5</td>
<td>100,000</td>
<td>hundred-thousands</td>
</tr>
<tr>
<td>6</td>
<td>1,000,000</td>
<td>millions</td>
</tr>
<tr>
<td>7</td>
<td>10,000,000</td>
<td>ten-millions</td>
</tr>
</tbody>
</table>

However, the decimal system is not the only possible number system. There are many possibilities. We could use a number system based on powers of 7, of 23, or of any number. However, in dealing with computers, the binary system, which is based on powers of 2 and uses only the symbols 0 and 1, is particularly useful. This is due to the fact that all functions in a computer depend on two possibilities, a high or a low voltage in a circuit, which can be represented mathematically as either a \(1\) (on or high) or \(0\) (off or low). Thus, in order to understand how a computer stores numbers, we need to understand the binary number system.

1This section is optional and may be skipped if desired. For more detail and more examples on this material, refer to Appendix A and Appendix B.
The binary number system uses 2 as a base. In the binary system, instead of a one’s column (10), a ten’s column (100), a hundred’s column (1000), and so on, we have columns that use the base of 2 raised to the powers. So we have a one’s column (2^0), a two’s column (2^1), a four’s column (2^2), an eight’s column (2^3), and so on.

While there are ten possible digits in the decimal system, there are only two possible digits in the binary system. In the decimal system, you can have a zero in the one’s column, or a 1, a 2, a 3, all the way up to 9. The number after 9 in the decimal system is 10, or a 1 in the ten’s column and a 0 in the one’s column. In the binary system, you can have a zero in the one’s column or a 1 in the one’s column. But the next number, 2, is actually a 1 in the two’s column and a 0 in the one’s column; that is, in the binary system, 2 is represented by 10. This is true for all the columns in the binary system. All binary numbers are composed of 0s and 1s.

In computer terms, each 0 or 1 represents a single bit.

With any number system, you can have as many columns as you want. However, for our purposes, we will not deal with more than 16 columns. Table 1.3 shows the first eight columns of the binary system.

| Table 1.3 The First Eight Columns of the Binary System |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|     2^7         |     2^6         |     2^5         |     2^4         |     2^3         |     2^2         |     2^1         |     2^0         |
| 128’s           | 64’s            | 32’s            | 16’s            | 8’s             | 4’s             | 2’s             | 1’s             |

Identifying the Base

The number 1 in binary is written the same way as the number 1 in decimal and the number 0 is also the same in both decimal and binary systems. However, this is where the similarities end. The decimal number 2 in binary is actually written 10 because it means there is one 2 in the two’s column and nothing in the one’s column. The decimal number 3 is written 11 in binary because there is one 2 in the two’s column and one 1 in the one’s column. The decimal number 4 is written 100 in binary, 5 is 101 in binary, and so on.

So how can you tell if the number 101 represents one-hundred-and-one in decimal or five in binary? There are several different conventions used to indicate the base of a number. In this book we will use a subscript. 101_10 means the number is one-hundred-and-one in base 10 (the decimal system) and 101_2 means the number is 5 in base 2 (the binary system). Converting the base 10 numbers 2, 3, 4, and 5 to base 2 is shown:

- 2_{10} = 1 \times 2^1 + 0 \times 2^0 = 102
- 3_{10} = 1 \times 2^1 + 1 \times 2^0 = 112
- 4_{10} = 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 1002
- 5_{10} = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1012

Example 1.11 shows how the decimal numbers from 0 to 15 are represented in binary notation.
Example 1.11 Binary Numbers from 0 to 15

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (_{10})</td>
<td>0000(_2)</td>
</tr>
<tr>
<td>1 (_{10})</td>
<td>0001(_2)</td>
</tr>
<tr>
<td>2 (_{10})</td>
<td>0010(_2)</td>
</tr>
<tr>
<td>3 (_{10})</td>
<td>0011(_2)</td>
</tr>
<tr>
<td>4 (_{10})</td>
<td>0100(_2)</td>
</tr>
<tr>
<td>5 (_{10})</td>
<td>0101(_2)</td>
</tr>
<tr>
<td>6 (_{10})</td>
<td>0110(_2)</td>
</tr>
<tr>
<td>7 (_{10})</td>
<td>0111(_2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 (_{10})</td>
<td>1000(_2)</td>
</tr>
<tr>
<td>9 (_{10})</td>
<td>1001(_2)</td>
</tr>
<tr>
<td>10 (_{10})</td>
<td>1010(_2)</td>
</tr>
<tr>
<td>11 (_{10})</td>
<td>1011(_2)</td>
</tr>
<tr>
<td>12 (_{10})</td>
<td>1100(_2)</td>
</tr>
<tr>
<td>13 (_{10})</td>
<td>1101(_2)</td>
</tr>
<tr>
<td>14 (_{10})</td>
<td>1110(_2)</td>
</tr>
<tr>
<td>15 (_{10})</td>
<td>1111(_2)</td>
</tr>
</tbody>
</table>

Appendix A explains how to convert any decimal number to a binary number.

Integer Representation

There are actually three categories of integers. Positive integers start at 1 and go up to infinity \{1, 2, 3, 4, 5, ...\}. Negative integers start at -1 and go down to negative infinity \{-1, -2, -3, -4, -5, ...\}. Zero is not included in either classification, but often the positive integers and zero are grouped together and called nonnegative integers.

Unsigned Integers

An unsigned integer is an integer without the positive (+) or negative (-) specified. In a system that only has unsigned numbers, it would be impossible to do the following subtraction: \(8 - 23\). The answer (-15) just doesn’t exist in this system. This would not be very practical for math problems, but there are many times using the unsigned integer format works well in a computer program. For example, if you wanted to assign an integer as an identification number for each student in a large college, you would not need any negative numbers. If you were writing a program that allowed a user to select a state in the United States and each state was identified by an integer, again, you would not need any negative integers.

While the range of possible integers that exists in the world is from negative infinity to positive infinity, a computer cannot possibly store all integers in this range. To do so, the computer's memory would have to be infinitely large! Computers, therefore, have a specific range of integers that they can process.

The smallest number that can be represented in unsigned integer format is 0 and the largest is the maximum unsigned integer. This maximum number may differ from one computer to the next. The maximum unsigned integer depends on the number of bits the computer allocates to store an unsigned integer.

For example, Table 1.4 shows the possible integers that can be represented by a computer that allocates 2 bits to represent an unsigned integer.

\(^2\)This section is optional and may be skipped if desired. For more detail and more examples on this material, refer to Appendix A and Appendix B.
Therefore, the range of unsigned integers that can be represented by 2 bits is 0 through 3. There’s not much you can do with such a limited range. However, if a computer allocates 4 bits to represent one unsigned integer, as shown in Table 1.5, the range is somewhat larger.

Table 1.4 Unsigned Integers That Can Be Represented by 2 Bits

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
</tbody>
</table>

The range of unsigned integers, given 4 bits to represent a single unsigned integer, is 0 through 15. This is four times as large as a 2-bit representation, but still quite small. If a computer allocates 8 bits for a single unsigned integer, the range increases from 0 to 255. This is because the largest decimal integer that can be represented by an 8-bit binary integer is 255. There is a mathematical formula that expresses the range of unsigned integers available to a computer. It is as follows:

**Range of Unsigned Integers:** \( 0 \ldots (2^N - 1) \) where \( N \) = the number of bits allocated to represent a single unsigned integer.

Here is what it means. If you have 4 bits in a binary integer, the largest decimal integer you can represent in binary is 15 (1111). The largest number that can be represented is one less than 16 because 16 is represented by a one in the next (2⁴) column. If you have 8 bits in a binary integer, the largest decimal integer you can represent in binary is 255 (11111111). Since 256₁₀ = 10000000₀₂, this exceeds the 8-bit allocation. The largest number that can be represented is one less than 2⁸. The smallest unsigned binary integer that can be represented, no matter how many bits you use, is always 0. Therefore, the formula for the range, as given above, means that the range goes from 0 to one less than \( 2^N \) where \( N \) represents the number of bits allocated to one unsigned integer.

Computers use groups of bits, called bytes, as we have discussed earlier. Bytes are always some multiple of 8 bits. Although we can theoretically calculate the range of unsigned integers for any number of bits using the formula above, in reality the only ranges of interest are normally those for which \( N \) is 8, 16, 32, or 64.
Example 1.12 Using the Formula to Find the Range of Unsigned Integers

What is the range of unsigned integers that can be represented by a computer that uses 24-bit bytes? We see, from Table 1.6, that the ranges are given for 16-bit bytes and 32-bit bytes. If we want to find the range of unsigned integers that can be represented if a computer uses 24-bit bytes, we must use the following formula:

\[ \text{Range} = 0 \ldots (2^N - 1) \]

where \( N \) = the number of bits allocated

Substituting 24 for \( N \), we get:

\[ \text{Range} = 0 \ldots (2^{24} - 1) \]
\[ = 0 \ldots (16,777,216 - 1) \]
\[ = 0 \ldots 16,777,215 \]

Table 1.6 Range of Unsigned Integers for Bytes of Various Sizes

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0 \ldots 255</td>
</tr>
<tr>
<td>16</td>
<td>0 \ldots 65,535</td>
</tr>
<tr>
<td>32</td>
<td>0 \ldots 4,294,967,295</td>
</tr>
<tr>
<td>64</td>
<td>0 \ldots 1.844674 \times 10^{19}</td>
</tr>
</tbody>
</table>

(Note: 1.844674 \times 10^{19} = 18,446,740,000,000,000,000—a really, really big number!)

Therefore, a computer with a 24-bit byte system can represent integers from 0 through 16,777,215. While this may be a large number, it would not even be large enough to assign ID numbers to the population of New York State. Clearly, computers need to be able to deal with very large numbers.

Signed Integers: Sign-and-Magnitude Format

Positive integers and zero are represented in the computer by converting the value of the number to binary format. However, we also need a way to represent negative numbers. The sign-and-magnitude format provides one way to do this. In sign-and-magnitude format, the number is also represented in binary but divided into two parts. The leftmost bit is reserved to represent the sign—positive or negative. The other bits represent the magnitude (or the absolute value) of the integer. For example, the sign-and-magnitude number 0111₂ represents +7 and the number 1111₂ represents −7.

However, there is a problem with this method of representing integers. Since the leftmost bit is reserved for the sign of the number, the binary number 0000₂ represents +0 while 1000₂ represents −0. This can be a problem for programmers and is one of the reasons why computers usually use a different method to represent integers.

We have already seen that the range of unsigned integers available in a 4-bit location is 0 through 15. However, when you reserve the leftmost bit for the sign, as in the sign-and-magnitude format, the number of bits available to represent the magnitude of the number decreases by one. Thus, the range decreases by a power of...
2. In a 4-bit allocation, using sign-and-magnitude format, you can now only represent integers from \(-7\) through \(+7\). Therefore, the maximum positive value is half of what it is for an unsigned integer. As shown in Example 1.13, the mathematical formula that expresses the range of sign-and-magnitude integers available to a computer is

\[
\text{Range: } -(2^{n-1} - 1) \ldots + (2^{n-1} - 1) \text{ where } N = \text{the number of bits allocated to represent a single sign-and-magnitude integer.}
\]

**Example 1.13 The Range of Sign-and-Magnitude Integers: A Comparison**

What is the range of integers that can be represented with sign-and-magnitude form, by a computer that uses 24-bit bytes? This time we must use the formula:

\[
\text{Range: } -(2^{N-1} - 1) \ldots + (2^{N-1} - 1) \text{ where } N = \text{the number of bits allocated to represent a single sign-and-magnitude integer.}
\]

Substituting 24 for \(N\), we get:

\[
\text{Range} = -(2^{24-1} - 1) \ldots (2^{24-1} - 1) = -(2^{23} - 1) \ldots (2^{23} - 1) = -(8,388,608 - 1) \ldots (8,388,608 - 1) = -8,388,607 \ldots 8,388,607
\]

Therefore, a computer with a 24-bit byte system can represent integers from \(-8,388,607\) through \(+8,388,607\). This represents the same physical amount of numbers but it cuts the largest number possible by half. Using sign-and-magnitude integers, a 24-bit byte system would not allow us enough room to assign ID numbers to the population of New York City. But computers need to deal with both negative numbers and very large numbers.

As you may imagine, most calculations done on a computer should be able to handle both positive and negative numbers. Sometimes using unsigned integers is valuable, especially since unsigned integers allow the user to utilize the whole bit allocation for storing the number because the leftmost bit does not have to be reserved for the sign.

The main advantage of sign-and-magnitude, as shown in Example 1.14 is that it is simple to convert from decimal to binary and vice-versa. Sign-and-magnitude format may be used for applications that do not need to perform arithmetic operations on numbers. For example, when changing an analog signal to digital, a positive or negative number is assigned to a signal. Since no computations will be done with these numbers, sign-and-magnitude may work well for such an application.

**Example 1.14 Sign-and-Magnitude Numbers**

Find the 4-bit binary representations of the following decimal numbers:

1. \(+3_{10}\): Simply convert \(3_{10}\) to binary to get \(11_2\). However, since this is a 4-bit number, you must add two zeros to the left to make 4 bits. Since this
is a positive number, the leftmost bit remains a zero:
\[ +13_{10} = 0011_2 \]
b. \(-6_{10}\) : First, convert \(6_{10}\) to binary to get \(110_2\). Put one zero to the left to make 4 bits. Then change the leftmost zero to a one to indicate that this is a negative number:
\[ -6_{10} = 1110_2 \]
c. \(-1_{10}\) : First, convert \(1_{10}\) to binary to get \(1_2\). Then add zeros to the left to make 4 bits. Finally, change the leftmost bit to a one to indicate the negative sign:
\[ -1_{10} = 1001_2 \]

When the sign does matter, sign-and-magnitude format is not often used. There are two main difficulties with this format, along with the fact that range is often not very large. First, as we have seen, the number zero has two representations in this format (see Appendix B for further explanation). This could cause errors and confusion in a computer program. Another reason sign-and-magnitude is not often used is because arithmetic operations like addition and subtraction are difficult to perform on numbers stored in this manner.

There are two other formats used to store signed integers. These are the one's complement and the two’s complement. The two’s complement is the most often used method, but it is also the most difficult to understand. Both methods are discussed in depth in Appendix B.

Self Check for Section 1.5

Self Check Questions 1.19–1.24 refer to material in optional sections.

1.19 Which of the following is not considered an integer in programming?
   a. 6
   b. 0
   c. -53
   d. 2.0

1.20 What is the decimal value of the following unsigned binary numbers?
   a. 02
   b. 102
   c. 00102
   d. 01112

1.21 The maximum unsigned integer that can be represented by a computer that uses 4 bits to store a single integer is ________.

1.22 T F The smallest number that can be represented in unsigned integer format is always zero.
1.23 Give one reason why the following formats might be used to represent integers and one reason why each might be avoided:
   a. unsigned format
   b. sign-and-magnitude format

1.24 The decimal number 12 is represented as an 8-bit binary number as \texttt{00001100}_2. How would the decimal number \texttt{-12} be represented, using the sign-and-magnitude format?

### 1.6 Floating Point Data

In programming, a floating point number is, roughly speaking, any number that is not of integer type. This includes all numbers with fractional parts, such as 4.6, \(-34.876, 6\frac{1}{2}\) and 7.0. A floating point number differs from an integer because all floating point numbers have both an integer part and a fractional part.

When you begin programming, you will see that you must declare your numbers as either integers or floating point numbers. This is because the computer both stores and manipulates integers in a different manner from floating point numbers.

You might wonder why we can’t simply represent all numbers as floating point and leave the fractional part as zero for integers. There are many reasons why programs use both integer and floating point numbers. Probably the most important reason is space and time. We have already seen how the range of possible representations changes dramatically with the size of memory allotted to store a number. For example, a 16-bit memory location can store up to \(+65,535\). When you add signed integers (negative and positive), the range of positive integers is reduced by half. Since a floating point number must have two parts, the integer part and the decimal part, some bits would have to be reserved for the decimal part (even if they were all zero), further reducing the range. Thus, it would not be practical to store any usable range of numbers in a 16-bit location. For applications that do not require the use of numbers with decimal parts, integer representations take up a lot less space in the computer's memory. If the computer does not have to deal with a large memory space, it will work faster. So integer representations allow the computer to save space and time.

There are many types of applications done by a computer that only need integers. For these, it would be inefficient to use floating points. For example, many Web sites count the number of “hits” (the number of times people access the site). This example, and numerous other types of counting applications, uses integers. Assigning numerical identification to customers in a database, to students in a university, or to serial numbers on products are all examples of where integer representation makes sense.

Of course, there are many more instances where integers simply won’t do the job. Most scientific calculations require floating point numbers. Any financial application requires floating point numbers to store dollar amounts. When you begin
to write computer programs, you will find that you use both integers and floating points in most programs.

**The Declare Statement Revisited**

In this text, when we declare variables in our programs we will use statements like the following:

```java
Declare Price As Float
Declare Counter As Integer
Declare Color As String
```

Many programming languages also allow you to declare several variables in one statement, so long as all the variables in that statement have the same data type. To declare several variables of the same data type at once, we will use the following:

```java
Declare Number1, Number2 As Integer
Declare FirstName, LastName, FullName As String
Declare Price, DiscountRate As Float
```

**Be Careful with Data Type Declarations**

In the Music Purchase program, the value of `DollarPrice` might be an integer, but it might not, so we should declare this variable to be of type `Float`. This can be done by placing the statement

```java
Declare DollarPrice As Float
```

at the beginning of the program.

There may be times when you are not sure what data type your variable is. For example, say you want to declare a variable named `Age` that will hold the value of a user's age. Should this variable be declared as `Integer` or as `Float`? If you are absolutely sure that your program will require the age input to be a whole number and that no internal computations within the program will cause that variable to become a number with a decimal part, then you may want to declare `Age` as `Integer`. After all, as we have learned, integers take up less room in the computer's memory.

However, if there is a possibility that some computation could result in the value of `Age` becoming a floating point number, then it is always safer to declare `Age` as `Float`. Saving a little space by using `Integers` where `Floats` are needed is not worth the risk of errors in your program.

**Naming Conventions for Variables**

We have said that the programmer decides how to name the variables in a program, within the parameters allowed. Although this is true, there are naming conventions that are used by many programmers. Some naming conventions may be common in one language and not in another; yet there are conventions that are common to all. Some programmers like to identify the data type in variable names. For example, instead of using `Number1` as the name of an integer variable,
intNumber1 might be used, where the identifier int indicates that this variable is of type Integer. Other examples might be as follows:

- **strName** for a string variable, where the identifier str indicates that this variable is of type String
- **fltPrice** for a Floating Point variable, where fit indicates that this is a Floating Point variable

Regardless of what you name your variables or what convention you follow, you still must declare the type, as in the following:

- Declare **Number1** As Integer
- Declare **Name** As String
- Declare **Price** As Float

### The Last Word on the Music Purchase Program

Now we have all the tools we need to present the final version of the Music Purchase program that was introduced in Section 1.2. This version makes use of our Declare, Input, Set, and Write statements, and looks as follows:

```plaintext
Declare Songs As Integer
Declare DollarPrice As Float
Write "Enter the number of songs you wish to purchase: "
Input Songs
Set DollarPrice = 0.99 * Songs
Write "The price of your purchase is " + DollarPrice
```

### Types of Floating Point Numbers

In this section, we delve a little deeper into how floating point numbers are represented. We begin with a short review of the types of floating point numbers. There are two types of numbers that can be represented by floating point. The first is the set of **rational numbers**. A rational number is any number that can be written as an integer divided by another integer. Example 1.15 shows some rational numbers.

#### Example 1.15 Some Rational Numbers

<table>
<thead>
<tr>
<th>Fraction</th>
<th>Decimal Form</th>
<th>Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5/2</td>
<td>2.5</td>
<td>20742/10000</td>
</tr>
<tr>
<td>43/4</td>
<td>10.75</td>
<td>86/10</td>
</tr>
<tr>
<td>2.0</td>
<td>2/1</td>
<td>0.8754</td>
</tr>
</tbody>
</table>

A number whose decimal representation eventually becomes periodic (i.e., the same sequence of digits repeats indefinitely) is called a repeating decimal.

---

This section is optional and may be skipped if desired. For more detail and more examples on this material, refer to Appendix A and Appendix B.
Therefore, the number $1/3$ is a repeating decimal. If you try to write $1/3$ as a decimal, the 3s will repeat indefinitely. Another example is $1/11$. If you convert this to a decimal representation, you get $0.090909\ldots$ and the sequence, 09, repeats indefinitely. All rational numbers have either finite decimal expansions (like $5\frac{1}{2} = 11/2 = 5.5$) or repeating decimals (like $1/11 = 0.090909\ldots$).

The second type of floating point number is the set of irrational numbers. An irrational number is a number that cannot be expressed as a fraction because the fractional part would go on for infinity without ever repeating a sequence. A repeating sequence is called a period. Irrational numbers have decimal expansions that neither terminate nor become periodic. Example 1.16 shows some irrational numbers.

**Example 1.16 Some Irrational Numbers**

$\sqrt{2} = 1.4142135\ldots$  \quad $\pi = 3.1415926535\ldots$

In programming, all rational and irrational numbers are represented as floating point numbers.

**A Word about Accuracy**

If you were asked to divide $9.00$ evenly between three people, it is clear that each person would get $3.00. But if you were asked to evenly divide $10.00$ among the same three people, it would be impossible. Two people would get $3.33$ and one would have to get $3.34$. There is no way to evenly divide any number that is not a multiple of 3 by 3. That is because $1/3$ is a repeating decimal. If you needed to represent $1/3$ in a computer, you could not keep repeating 3s until infinity. Let’s say you were allowed four places to represent the decimal part of a number. Then $1/3$ would be represented as $0.3333$. This number has the exact same value as $0.3330$. But $1/3$ does not equal $0.3330$. Computer programs do not calculate with complete accuracy!

We can see from Example 1.17 that a computer is not always accurate. In fact, sometimes the computer may not be accurate when doing relatively simple calculations. Depending on various factors, a number as simple as 2.0 may actually be stored in the computer as 1.99999999 or 2.00000001. (The material in Appendices B and C can help you understand how this might happen.) This difference may not affect the results in most cases, but it is important to realize that a computer is not always 100% accurate!

**Example 1.17 Your Computer May Not Be as Accurate as You Think!**

a. Calculate the circumference of a circle with a radius of 10 inches, using four places in the representation of $\pi$:

- The formula for the circumference of a circle is $C = 2 \cdot \pi \cdot R$, where $C$ is the circumference and $R$ is the radius of the circle.


- Using four places to represent the fractional part of a number, \( \pi = 3.1416 \). This is because the first five digits of the fractional part of \( \pi \) are 14159 and, when using only four digits, we round up to 1416.
- So \( C = 2 \times 3.1416 \times 10 = 62.8320 \).

b. Calculate the circumference of a circle with a radius of 10 inches, using six places in the representation of \( \pi \):
- Using six places to represent the fractional part of a number, \( \pi = 3.141592 \).
- So \( C = 2 \times 3.1416 \times 10 = 62.831840 \).

And 62.831840 does not equal 62.8320.

**What & Why** The fact that a computer may not be perfectly accurate is no reason to stop trusting computers. In the real world, we can never measure anything with absolute accuracy. For example, if you weigh yourself on a digital scale, the reading is only as accurate as the number of fractional digits your scale allows. With a scale that is advertised to display accurately to half-pounds, a display of 134.5 pounds means you may actually weigh between 134\( \frac{1}{4} \) and 134\( \frac{3}{4} \) pounds. Anything less than 134\( \frac{1}{4} \) would round down to 134.0 and anything greater than 134\( \frac{3}{4} \) would round up to 135.0 on the scale. The weight difference between the lowest possible reading on such a scale and the highest is \( \frac{1}{2} \) pound.

So your scale actually weighs with accuracy of \( \pm \frac{1}{2} \) pound. A computer that stores the number 2.0 as 1.9999999 or 2.00000001 is accurate to within \( \pm 0.00000002 \). And that’s a lot more accurate than your bathroom scale!

**Floating Point Representation**

The **IEEE Standard** is the most widely accepted standard for representation of floating point numbers. IEEE stands for the Institute of Electrical and Electronics Engineers, Inc. and it is the world’s leading professional association for the advancement of technology.

When a floating point number is converted to binary using the IEEE Standard, the number is said to be normalized. **Normalization** is a way to represent floating point numbers in a consistent manner that allows the computer to perform operations more easily.

A normalized binary number consists of three parts: the sign part, the exponential part, and the mantissa. An in-depth discussion of normalized numbers and the **sign-exponent-mantissa form** is available in Appendix C. However, today it is the rare programmer who will ever have to actually deal with binary floating point numbers. It is good for you to understand what a normalized floating point looks like so you can get a feel for what is happening when you write a program that uses floating point numbers. For the purposes of this text, however, you do not need to know how to convert a floating point number from base 10 to base 2.

---

\( ^4 \)This section is optional and may be skipped if desired. For more detail and more examples on this material, refer to Appendix C.
There are two ways to store a number using the IEEE Standards: as single precision and as double precision. A single precision number uses 32 bits and a double precision number uses 64 bits. A single precision number can easily be stored in one or two storage locations on most computers. Double precision allows for a much larger range of numbers and allows for much greater accuracy than single precision but, as you remember, nothing is ever 100% accurate! Example 1.18 shows what single precision normalized floating point numbers look like in binary.

Example 1.18  A Peek Inside: Normalized Floating Point Numbers

In the following examples, the first bit represents the sign. A 0 (zero) respresents a positive number and a 1 represents a negative number. The next 8 bits are the exponent and the last 23 bits are the mantissa.

a. The number \(-14.510\) looks like this in single precision sign-exponent-mantissa form:
   \[ 1 \ 10000010 \ 11010000000000000000000 \]

b. The number \(+\frac{7}{8}\) or \(0.875\) looks like this in single precision sign-exponent-mantissa form:
   \[ 0 \ 01111110 \ 11000000000000000000000 \]

c. The number \(-\frac{3}{4}\) or \(0.75\) looks like this in single precision sign-exponent-mantissa form:
   \[ 1 \ 01111110 \ 10000000000000000000000 \]

Since a single precision number uses 32 bits, all the bits must be accounted for in any floating point number. The leftmost bit designates the sign. The next 8 bits are the exponent (which is expressed in a specific form, discussed in Appendix C). The last 23 bits are reserved for the mantissa (also discussed in more depth in Appendix C). That’s why there is such a long trail of 0s in these examples; all 32 bits must be used.

Integers versus Floating Point Numbers

The number \(7.0\) is a floating point number. The number \(7\) is an integer. These numbers have exactly the same value, but they are stored differently in the computer and are treated differently within a program. Just something to keep in mind....

Although variables declared to be of type Integer cannot take on non-integer values (such as \(2.75\)), variables declared to be of type Float can take on integer values. For example, if NumberFloat is of type Float and NumberInt is of type Integer, then the following statements apply:

- Set \(\text{NumberFloat} = 5.5\) is normally valid (that is, correct)
- Set \(\text{NumberInt} = 5.5\) is normally invalid (incorrect since 5.5 is not an integer)

In most programming languages, a variable is declared to be of type Float by using the proper statement within the program. For example, to declare the variable...
Number to be of type `float` in C++ and Java we use the following statement:

```c
float Number;
```

but in Visual Basic we use the following statement:

```vb
Dim Number As Double
```

or

```vb
Dim Number As Decimal
```

These statements cause the computer to allocate the amount of memory needed to store the `Floating Point` variable named `Number`. Typically, this is twice as much memory as needed for an integer variable.

Actually, most programming languages offer several options for declaring floating point numbers, but the specifics of why one option is used over another is not relevant to this text. It is only significant to know that if you want a number to be used in a program as a `Floating Point` data type, you must declare it as such.

---

**Self Check for Section 1.6**

*Self Check Questions 1.25, 1.27, and 1.29 refer to material that is optional.*

1.25 The most widely accepted standard for representation of floating point numbers is the ________.

1.26 Which of the following is *not* a floating point number?

   a. 6
   b. 0.0
   c. -0.53
   d. 125,467,987.8792

1.27 Which of the following is *not* a rational number?

   a. $\sqrt{2}$
   b. $\frac{567}{32}$
   c. $\frac{1}{3}$
   d. 7.623623623623623

1.28 The following data will be used in a program. Create appropriate variable names and declare these variables with the data type needed for their use:

   a. the number of batteries needed to operate a flashlight
   b. the price of filling up a car’s gas tank
   c. the area of a circle, given the radius

1.29 T F Since integers can only be positive numbers, it is best to declare all numbers as floating point numbers.

1.30 T F Declaring a variable as type `Float` is efficient because the `Float` data type uses about half as much memory as the `Integer` type.
### Key Terms

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>irrational number</td>
</tr>
<tr>
<td>-</td>
<td>Java program code</td>
</tr>
<tr>
<td>*</td>
<td>magnitude</td>
</tr>
<tr>
<td>/</td>
<td>maximum unsigned integer</td>
</tr>
<tr>
<td>^</td>
<td>named constant</td>
</tr>
<tr>
<td>%</td>
<td>normalization</td>
</tr>
<tr>
<td></td>
<td>alphanumeric data</td>
</tr>
<tr>
<td>annotate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arithmetic operators</td>
</tr>
<tr>
<td>assignment statement</td>
<td></td>
</tr>
<tr>
<td>binary system</td>
<td></td>
</tr>
<tr>
<td>bit</td>
<td></td>
</tr>
<tr>
<td>C++ program code</td>
<td></td>
</tr>
<tr>
<td>character</td>
<td></td>
</tr>
<tr>
<td>character string</td>
<td></td>
</tr>
<tr>
<td>concatenation</td>
<td></td>
</tr>
<tr>
<td>constant (in program)</td>
<td></td>
</tr>
<tr>
<td>cyclic process</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td></td>
</tr>
<tr>
<td>data types</td>
<td></td>
</tr>
<tr>
<td>declare (variables)</td>
<td></td>
</tr>
<tr>
<td>floating point number</td>
<td></td>
</tr>
<tr>
<td>hierarchy of operations</td>
<td></td>
</tr>
<tr>
<td>IEEE Standard</td>
<td></td>
</tr>
<tr>
<td>initialize</td>
<td></td>
</tr>
<tr>
<td>input (to program)</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>irrational number</td>
</tr>
<tr>
<td></td>
<td>Java program code</td>
</tr>
<tr>
<td></td>
<td>magnitude</td>
</tr>
<tr>
<td></td>
<td>maximum unsigned integer</td>
</tr>
<tr>
<td></td>
<td>named constant</td>
</tr>
<tr>
<td></td>
<td>normalization</td>
</tr>
<tr>
<td></td>
<td>alphanumeric data</td>
</tr>
<tr>
<td></td>
<td>annotate</td>
</tr>
<tr>
<td></td>
<td>arithmetic operators</td>
</tr>
<tr>
<td></td>
<td>assignment statement</td>
</tr>
<tr>
<td></td>
<td>binary system</td>
</tr>
<tr>
<td></td>
<td>bit</td>
</tr>
<tr>
<td></td>
<td>C++ program code</td>
</tr>
<tr>
<td></td>
<td>character</td>
</tr>
<tr>
<td></td>
<td>character string</td>
</tr>
<tr>
<td></td>
<td>concatenation</td>
</tr>
<tr>
<td></td>
<td>constant (in program)</td>
</tr>
<tr>
<td></td>
<td>cyclic process</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>data types</td>
</tr>
<tr>
<td></td>
<td>declare (variables)</td>
</tr>
<tr>
<td></td>
<td>floating point number</td>
</tr>
<tr>
<td></td>
<td>hierarchy of operations</td>
</tr>
<tr>
<td></td>
<td>IEEE Standard</td>
</tr>
<tr>
<td></td>
<td>initialize</td>
</tr>
<tr>
<td></td>
<td>input (to program)</td>
</tr>
<tr>
<td></td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>irrational number</td>
</tr>
<tr>
<td></td>
<td>Java program code</td>
</tr>
<tr>
<td></td>
<td>magnitude</td>
</tr>
<tr>
<td></td>
<td>maximum unsigned integer</td>
</tr>
<tr>
<td></td>
<td>named constant</td>
</tr>
<tr>
<td></td>
<td>normalization</td>
</tr>
<tr>
<td></td>
<td>alphanumeric data</td>
</tr>
<tr>
<td></td>
<td>annotate</td>
</tr>
<tr>
<td></td>
<td>arithmetic operators</td>
</tr>
<tr>
<td></td>
<td>assignment statement</td>
</tr>
<tr>
<td></td>
<td>binary system</td>
</tr>
<tr>
<td></td>
<td>bit</td>
</tr>
<tr>
<td></td>
<td>C++ program code</td>
</tr>
<tr>
<td></td>
<td>character</td>
</tr>
<tr>
<td></td>
<td>character string</td>
</tr>
<tr>
<td></td>
<td>concatenation</td>
</tr>
<tr>
<td></td>
<td>constant (in program)</td>
</tr>
<tr>
<td></td>
<td>cyclic process</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>data types</td>
</tr>
<tr>
<td></td>
<td>declare (variables)</td>
</tr>
<tr>
<td></td>
<td>floating point number</td>
</tr>
<tr>
<td></td>
<td>hierarchy of operations</td>
</tr>
<tr>
<td></td>
<td>IEEE Standard</td>
</tr>
<tr>
<td></td>
<td>initialize</td>
</tr>
<tr>
<td></td>
<td>input (to program)</td>
</tr>
<tr>
<td></td>
<td>integer</td>
</tr>
</tbody>
</table>

### Chapter Summary

**In this chapter, we discussed the following topics:**

1. The nature of programs both in everyday life and for the computer as follows:
   - A general problem-solving strategy—understand the problem, devise a plan, carry out the plan, review the results
   - The program development cycle—analyze the problem, design a program, code the program, test the program

2. The following basic building blocks of a computer program:
   - Input statements transmit data to the program from an outside source
   - Processing manipulates data to obtain the desired results
   - Output statements display the results on the screen, printer, or another device
3. Pseudocode statements used to perform input, processing, and output as follows:
   - Input, Set (assignment), and Write
4. The use of input prompts in a program
5. The use of variables in a program such as the following:
   - Variable names
   - Constants and named constants
   - How the computer processes variables
6. Basic arithmetic operations (addition, subtraction, multiplication, division, modulus, and exponentiation) as follows:
   - How arithmetic operators are represented in a programming language
   - The order in which arithmetic operations are performed (hierarchy of operations)
7. Character and String data types as follows:
   - The definition of a character string
   - Declaring Character and String variables
   - The concatenation operator (+) for joining strings
8. The Integer data type:
   - Declaring integer variables
   - How decimal numbers are represented in the binary system
   - How unsigned integers are represented (positive and zero) in a computer
   - The sign-and-magnitude method of representing signed integers
   - The range of integers that can be represented in a computer
9. The Floating Point data type:
   - How Floating Point numbers differ from Integers
   - Rational and Irrational numbers
   - How Floating Point numbers are represented in a computer and the IEEE Standard
   - Declaring Floating Point variables
   - The limits of computer accuracy

Review Exercises

Fill in the Blank

1. A computer __________ is a list of instructions to be executed by the computer to accomplish a certain task.
2. The general process of designing a suitable computer program to solve a given problem is known as the __________.
3. The three basic building blocks of a program are input, __________, and output.
4. The term __________ refers to the numbers, text, and other symbols that are manipulated by a program.
5. The first of the following statements
   Write "Enter your weight in pounds:"
   Input Weight
   provides a(n) ____________ for the input.

6. The Input statement in Exercise 5 assigns the number entered by the user
to the variable ____________.

7. The two basic types of numeric data are ____________ data and ____________ data.

8. In most programming languages variables that represent numbers must be
   ____________, or defined, prior to their use.

9. The maximum unsigned number that can be represented by a computer
   that uses 8 bits to store a single unsigned integer is ____________.

10. The standard representation for storing integers in computers today is
    ____________.

11. The __________ is the most widely accepted standard for representation of
    floating point numbers.

12. Roughly speaking, a(n) ____________ is any symbol that can be typed at the
    keyboard.

13. A character ____________ is any sequence of characters.

Multiple Choice

14. Which of the following is the first step in the general problem-solving
    strategy?
    a. Devise a plan to solve the problem
    b. Make sure that you completely understand the problem
    c. Make a list of possible solutions to the problem
    d. Make a list of what is needed to review the results

15. After coding a computer program, you should do which of the following?
    a. Analyze the problem that led to that program
    b. Devise a plan for using the code to solve the given problem
    c. Run the program to see if it works
    d. Move on to the next problem

16. Which of the following is not an integer?
    a. 4
    b. 28,754,901
    c. –17
    d. 3.0

17. Which of the following is not a floating point number?
    a. 236,895.34766
    b. –236,895.34766
    c. 0
    d. 6/18
18. Which of the following is not a rational number?
   a. \( \sqrt{3} \)
   b. 0.873
   c. \( \frac{1}{3} \)
   d. \( \frac{22}{5} \)

True or False
19. T  F In everyday life, a program is a plan of action to attain a certain end.
20. T  F Problem solving is a cyclic process, often requiring you to return to a previous step before you find a satisfactory solution.
21. T  F After devising and carrying out a plan of action to solve a problem, you should review your results to see if the plan has worked.
22. T  F As you develop a computer program, you should code the program before designing it.
23. T  F A variable may be considered the name for a certain storage location in the computer's memory.
24. T  F The smallest number that can be represented in sign-and-magnitude integer format is always 0.
25. T  F A rational number is any number that can be represented as an integer divided by another integer.
26. T  F A number stored as an integer takes up less space in the computer's memory than a floating point number.
27. T  F If the value of the variable \( \text{MyAge} \) is 3, then the statement \( \text{Set MyAge} = 4 \) assigns the value 7 to \( \text{MyAge} \).

Short Answer
28. Suppose \( X = 3 \) and \( Y = 4 \). Give the value of each of the following expressions:
   a. \( X * Y ^ 2 / 12 \)
   b. \( ((X + Y) * 2 - (Y - X) * 4) ^ 2 \)
29. What are the two possible values (depending on the programming language that is being used) of the expression \( 7/2 \)?
30. Suppose \( X = 3 \) and \( Y = 4 \). If all parentheses were omitted from the expression in Exercise 28b, what would be its value?
31. Suppose \( X = 14 \). Give the value of each of the following expressions:
   a. \( X \% 5 \)
   b. \( X \% 7 \)
32. Suppose \( X = 12, Y = 6, \) and \( Z = 5 \). Give the value of each of the following expressions:
   a. \( X \% (Z + Y) \)
   b. \( X \% (Y + Z) \)
33. What is the decimal value of each of the following binary numbers?
   a. $1000_2$
   b. $0110_2$

34. What is the decimal value of each of the following binary numbers?
   a. $0000_2$
   b. $1111_2$

35. If Name1 = "John" and Name2 = "Smith", what string results from each of the following operations?
   a. Name1 + Name2
   b. Name2 + " ," + Name1

36. Write a pair of statements that prompts for and inputs the user's age.

37. Write a pair of statements that prompts for and inputs an item's price.

38. Write a series of statements that does the following:
   - Inputs the user's age (including a suitable prompt).
   - Subtracts 5 from the number entered by the user.
   - Displays the message “You don’t look a day over” followed by the number computed in the previous step.

39. Write a series of statements that does the following:
   - Inputs the price of an item, in dollars, from the user (including a suitable prompt).
   - Divides the number entered by the user by 1.62.
   - Displays the message “That’s only”, followed by a number computed in a previous step, followed by the message “in British pounds.”

40. Suppose that Number1 = 15 and Number2 = 12 are both of Integer type. Give the two possible values (depending on the programming language in use) of Number1/Number2.

41. If Name1 = "Marcy", Text1 = "is now", Text2 = " years old.", and Age = 24, what will be output after the following operation:
   Name1 + Text1 + Age + Text2

42. If Character1 and Character2 are single characters, is Character1 + Character2 also a single character?

Exercises 43–50 refer to the following program:

Set Number1 = 4
Set Number1 = Number1 + 1
Set Number2 = 3
Set Number2 = Number1 * Number2
Write Number2

43. List the variables in the program.

44. What are the program's
   a. Input statements?
   b. Assignment statements?
   c. Output statements?
45. What number is displayed by this program?
46. Replace each of the first and third statements of this program by a statement that inputs a number from the user.
47. Write a statement that supplies a suitable input prompt for the Input statements in Exercise 46.
48. Suppose we want to precede the last statement in this program by a statement that displays the following message:
The result of the computation is:
Write such a statement.
49. What are the possible data types for Number1 and Number2 in this program?
50. Write a statement that declares the variables used in this program.

Programming Problems

1. Write a list of instructions (like those that appear in Section 1.1) so that someone can perform each of the following tasks:
   a. Do one load of laundry
   b. Use an ATM machine to withdraw money
   c. Make a sandwich (you choose the type of sandwich)

In Programming Problems 2–6, write a program (like the programs in this chapter) to solve the given problem. Include appropriate input prompts and annotate output.

2. Write a program that computes and displays a 15 percent tip when the price of a meal is input by the user. (Hint: the tip is computed by multiplying the price of the meal by 0.15.) You will need the following variables:
   MealPrice (a Float)  Tip (a Float)

3. Write a program that converts a temperature input in degrees Celsius into degrees Fahrenheit and displays both temperatures. You will need the following variables:
   Celsius (a Float)  Fahrenheit (a Float)
   You will need the following formula:
   Fahrenheit = (9/5)*Celsius + 32

4. Write a program that computes and displays the batting average for a baseball player when the user inputs the number of hits and at-bats for that player. Batting average is computed by dividing the number of hits by the number of at-bats.) You will need the following variables:
   Hits (an Integer)  AtBats (an Integer)  BatAvg (a Float)

5. Write a program to compute and display an investment’s total interest and final value when the user inputs the original amount invested, rate of
interest, (as a decimal), and number of years invested. You will need the following variables:

- **Interest** (a Float)
- **Principal** (a Float)
- **Rate** (a Float)
- **FinalValue** (a Float)
- **Time** (an Integer)

You will need the following formulas:

- \( \text{Interest} = \text{Principal} \times \text{Rate} \times \text{Time} \)
- \( \text{FinalValue} = \text{Principal} + \text{Interest} \)

6. Write a program that inputs the first name, middle initial (without the period), and last name of a user and displays that person's name with the first name first, middle initial followed by a period, and last name last. You will need the following variables:

- **FirstName** (a String)
- **MiddleInitial** (a String)
- **LastName** (a String)