Program Modules, Subprograms, and Functions

In Chapter 2, we introduced the idea of modular programming. We discussed how to design and code a program as a set of interrelated modules. In subsequent chapters, especially in the Focus on Problem Solving sections, we used this technique to help simplify the design of relatively complicated programs. In this chapter, we will discuss more advanced aspects of this topic, including the concepts of arguments and parameters, functions, and recursion.

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

- Use a data flow diagram to indicate the data being transmitted among the program's modules [Section 7.1]
- Use arguments and parameters to pass data between modules [Section 7.1]
- Use value and reference parameters [Section 7.2]
- Define and specify the scope of a variable [Section 7.2]
- Use some functions commonly built into a programming language [Section 7.3]
- Create your own functions [Section 7.3]
- Create recursive functions to solve certain programming problems [Section 7.4]
As you know, a basic problem-solving strategy involves breaking a problem into modules and submodules and solving each subproblem separately. When we use this technique, we often use the data generated by one module in another module. For example, suppose you want to send your mother a bouquet of flowers and as luck would have it, you have just received an email from Ye Olde Flower Shop offering you a discount on a bouquet if you order by phone within the next few days. So the problem of how to send a gift of flowers to your mom has the following simple solution:

1. Locate a suitable bouquet and the store’s phone number from the Ye Olde Flower Shop Web site.
2. Call Ye Olde Flower Shop.
3. Place your order.

To solve this problem, the data collected in Step 1 (the phone number and bouquet name) are used in Steps 2 and 3, respectively. In the language of programming, we say that the phone number and bouquet name are exported from the Locate_Information module and imported by the Call_Shop and Place_Order modules, respectively.

Here’s another example of passing data among modules. Consider the steps you take to file your federal income tax return. You must do the following:

1. Gather data about your income and (possibly) your expenses.
2. Fill out the forms.
3. Mail or email your return to the IRS.

As you fill out the main form (Form 1040), you may discover that you also have to complete two other documents. You might be required to submit Schedule A (deductions) and Schedule B (interest and dividends). So a refinement of Step 2 might look as follows:

2. Fill out Form 1040.
   a. Fill out Schedule A.
   b. Fill out Schedule B.

The Form_1040 module imports all data from the Gather_Data module and exports the following:

- The expense data (if necessary) to the Schedule_A module, which imports the data and exports the total deductions to Form_1040
- The interest and dividend income data (if necessary) to the Schedule_B module, which imports the data and exports the total interest and total dividend income to Form_1040

In the everyday world, you transfer data from one module to another by looking at it and recording it. In programming, this action is accomplished by using parameters and arguments.
7.1 Data Flow Diagrams and Parameters

In this section, we will describe how data is transmitted between program submodules, or subprograms. We will discuss subprogram parameters, which allow the program to transmit information between modules, and data flow diagrams, which keep track of the data transmitted to and from each subprogram.

Most subprograms manipulate data. If a data item in the main program is needed in a certain subprogram, its value must be passed to, or imported by that subprogram. Conversely, if a data item processed by a subprogram is needed in the main program, it must be returned to, or exported to that module. We say that we pass a value to a subprogram and that subprogram may or may not return a value to the calling program. To illustrate these concepts, let’s consider the following programming problem.

A Big Sale: The Sale Price Computation Program

Katrina Katz owns a small pet supply store. She wants you to design a program that will compute and display the sale price of a discounted store item when she inputs the original price and its percentage discount.

We will briefly discuss the analysis and design for this problem and then use it to introduce the concepts of data flow diagrams and arguments and parameters. (A similar problem was presented in Chapter 2 to illustrate the notion of top-down modular design.)

Problem Analysis

Our program must input the original price (OriginalPrice) of an item and its percentage discount (DiscountRate). Then it must compute the sale price (SalePrice) of the item and display this amount. To arrive at the sale price, first we compute the amount of the discount using the following formula:

\[ \text{AmountSaved} = \text{OriginalPrice} \times \text{DiscountRate}/100 \]

Note that dividing by 100 converts the discount rate, entered as a percentage, to a decimal. We need the decimal amount because AmountSaved represents currency. This also means that all variables will be declared as Float. Then we compute the sale price using the following:

\[ \text{SalePrice} = \text{OriginalPrice} - \text{AmountSaved} \]

Program Design

Our modular design for this program consists of a Main module and four submodules, constructed as follows:

1We will use the word subprogram to describe the code that implements a program submodule; later we will consider a specific type of subprogram—the function. Be aware that the words subprogram and function may have other meanings in some programming languages.
Main Module
Call Welcome_Message module
Call Input_Data module
Call Compute_Results module
Call Output_Results module

Welcome_Message Module
Display a brief description of the program

Input_Data Module
Prompt for and input the original price, OriginalPrice
Prompt for and input the percentage discounted, DiscountRate

Compute_Results Module
Set AmountSaved = OriginalPrice * DiscountRate/100
Set SalePrice = OriginalPrice – AmountSaved

Output_Results Module
Write “The original price of the item is $ “ + OriginalPrice
Write “The discount is: ” + DiscountRate + “%”
Write “The sale price of the item is $ “ + SalePrice

Imported and Exported Data
The following occurs in the Sale Price Computation program:

- The Welcome_Message module does not import or export any data. This means that no data is passed to it and it does not return any data to the Main module.
- The Input_Data module inputs data, OriginalPrice and DiscountRate, from the user and then exports (sends or returns) these values to the Main module so that they can be used by another module.
- The Compute_Results module imports the values of OriginalPrice and DiscountRate from the Main module and exports the value of SalePrice to the Main module. Alternatively, we can say that the values of OriginalPrice and DiscountRate are passed to the Compute_Results module and the value of SalePrice is returned to the Main module. The value of AmountSaved is neither imported by, nor exported from, the Compute_Results module. It is only needed and used within the Compute_Results module.
- The Output_Results module displays the values of OriginalPrice, DiscountRate, and SalePrice, so it needs to import these values from the Main module. It does not export any data to another program module.

Data Flow Diagrams
In the process of designing a program like this, we can keep track of the data passed among the various modules by using a data flow diagram. This is a hierarchy chart that shows the data imported by and exported from each program mod-
ule. For example, the data flow diagram for the sale price computation program design is given in Figure 7.1. The arrows indicate the direction in which the data is passed.

Figure 7.1 Data flow diagram for the Sale Price Computation program

An Introduction to Arguments and Parameters

When data is passed between a calling module and a submodule, programming languages use parameters to transfer the data. In this discussion about parameters, we will use the Sale Price Computation program as an example. In our design of this program, the Output_Results module imports and displays the values of OriginalPrice, DiscountRate, and SalePrice. At the time that the Output_Results module is called, these variables will have values that were assigned within other subprograms and then transmitted to the main program. For example, Katrina Katz may have decided to put doghouses on sale. The original price of a doghouse might have been $150.00 and the discount may have been 30 percent; therefore, the sale price would end up being $105.00 and the values of these variables would be as follows:

\[
\text{OriginalPrice} = 150.00, \quad \text{DiscountRate} = 30.0, \quad \text{SalePrice} = 105.00
\]

These values must be sent from one module to another.

We will use the following pseudocode to call the Output_Results subprogram, and at the same time, pass the values of OriginalPrice, DiscountRate, and SalePrice to this subprogram as follows:

\[
\text{Call Output\_Results(OriginalPrice,DiscountRate,SalePrice)}
\]

Notice that we place the relevant variable names in parentheses, separated by commas. This pseudocode transfers control to the Output_Results subprogram and simultaneously transmits the values of the indicated variables to this subprogram.
If we use this pseudocode for the `Call` statement, then the first line (the **header**) in the definition of the `Output_Results` subprogram must contain a corresponding list of three variables as follows:

```plaintext
Subprogram Output_Results(OldPrice, Rate, NewPrice)
```

Notice that the names of the variables listed in the `Call` statement and the subprogram header don't need to be the same. In fact, in most cases it is better if they are not the same for reasons that will be explained later in the chapter. However, the names of the variables listed in the `Call` statement and those in the subprogram header must agree in number, in type, and in the order in which they are given in the subprogram's list. In this case, we are sending (or passing) three values (stored in variables named `OriginalPrice`, `DiscountRate`, and `SalePrice`) so there must be three variables listed in the `Call` statement and in the subprogram header. Also, if a certain variable in the `Call` statement is of one type, then the corresponding variable in the subprogram header cannot be of a different type. For example, if the fourth variable in a `Call` statement is of type `Integer`, then the fourth variable in the subprogram header must be of type `Integer` as well. In this case, our three variables (`OriginalPrice`, `DiscountRate`, and `SalePrice`) are all `Floats` so the three variables in the `Call` statement (`OldPrice`, `Rate`, `NewPrice`) must also be `Floats`.

The *order* in which the variables are sent to the calling submodule is extremely important. In this example, the calling submodule header is written as follows:

```plaintext
Subprogram Output_Results(OldPrice, Rate, NewPrice)
```

The value of the first variable that is sent over will be passed to the variable `OldPrice`, the value of the second variable will be passed to `Rate`, and the value of the third variable will be passed to `NewPrice`. We will take a closer look at what this means.

The items listed in parentheses in the `Call` statement are known as **arguments**, whereas those appearing in the subprogram header are known as parameters. Arguments (appearing in the `Call`) may be constants, variables, or more general expressions, but parameters (appearing in the subprogram header) must be variables. This means you can *send* data to a subprogram that is stored as a variable or is a constant or an expression. These values will be passed into the variables listed as parameters in the subprogram.

**How Data Is Transferred between Modules**

When a subprogram is called, the current values of the *arguments* are assigned to the corresponding *parameters*. This correspondence is made solely on the basis of the order of appearance in the two lists. For example, when the subprogram with the header

```plaintext
Subprogram Output_Results(OldPrice, Rate, NewPrice)
```

is called by the statement

```plaintext
Call Output_Results(OriginalPrice, DiscountRate, SalePrice)
```
the following occurs:

- The value of the first argument (**OriginalPrice**) is assigned to the first parameter (**OldPrice**).
- The value of the second argument (**DiscountRate**) is assigned to the second parameter (**Rate**).
- The value of the last argument (**SalePrice**) is assigned to the last parameter (**NewPrice**).

Examples 7.1 and 7.2 illustrate this concept.

**Example 7.1 Passing Data between Modules**

Suppose that the values of **OriginalPrice**, **DiscountRate**, and **SalePrice** are respectively, 200.00, 20.0, and 160.00. After the call to **Output_Results**, the variables within the **Output_Results** subprogram have the following values:

- **OldPrice** = 200.00
- **Rate** = 20.0
- **NewPrice** = 160.00

Pictorially, we can represent the way the values of the arguments in the call are transmitted to the parameters in the subprogram header as follows:

```
Call Output_Results( OriginalPrice, DiscountRate, SalePrice)

Subprogram Output_Results( OldPrice, Rate, NewPrice)
```

**Argument and Parameter Names**

Be aware that, while it is not required that you use different names for an argument and its corresponding parameter, it is normally better to do so. For example, the header for the **Output_Results** subprogram could be the following:

```
Subprogram Output_Results(OriginalPrice,DiscountRate,SalePrice)
```

Regardless of the names used for arguments and parameters, be sure that you use the name listed in the subprogram header to refer to a given parameter when you write the statements in that subprogram description.

**Example 7.2 More about Passing Data between Modules**

The pseudocode for the **Output_Results** subprogram (with parameters **OldPrice**, **Rate**, and **NewPrice**) is as follows:

```
Subprogram Output_Results(OldPrice,Rate,NewPrice)
    Write "The original price of the item is $ " + OldPrice
    Write "The discount is: " + Rate + "%"
    Write "The sale price of the item is $ " + NewPrice
End Subprogram
```
On the other hand, the following pseudocode would work just as well:

```
Subprogram Output_Results(OriginalPrice, DiscountRate, SalePrice)
    Write "The original price of the item is $ " + OriginalPrice
    Write "The discount is: " + DiscountRate + "%"
    Write "The sale price of the item is $ " + SalePrice
End Subprogram
```

Remember that the arguments that appear in a call to a subprogram may be constants, variables, or expressions. If variables appear in the call, in many programming languages they must be declared in the module that contains the Call statement. The parameters that appear in the subprogram itself, however, are declared in that subprogram’s header.

Example 7.3 uses a Call statement that contains String values, which are sent to a subprogram with two parameters. It demonstrates the importance of sending values to subprograms in proper order.

**Example 7.3  A Horse of a Different Color**

Katrina Katz, our pet supply store owner, wants a subprogram that will display the type and color of a pet ordered by a customer. The following program segment has a Call statement that sends two arguments to the subprogram. The arguments are the color and type of the pet. In this Call statement, the arguments passed to the subprogram are actual values; they are not variables. However, the parameters in the subprogram header are variables. In this example, they take on the values of yellow and duck. The subprogram simply displays the information sent in the Call as follows:

```
Call Animal("yellow","duck")
Subprogram Animal(Color, Beast)
    Write "The pet you are buying is a " + Color + " " + Beast
End Subprogram
```

In this case, the display would be as follows:

```
The pet you are buying is a yellow duck
```

However, if the expressions in the calling statement were reversed, as shown below, the result would be significantly different.

```
Call Animal("duck","yellow")
Subprogram Animal(Color, Beast)
    Write "The pet you are buying is a " + Color + " " + Beast
End Subprogram
```

If this program segment were coded and run, the display would not be what the programmer intended; rather it would look as follows:

```
The pet you are buying is a duck yellow
```

Example 7.3 demonstrates how an improper Call statement, where the arguments are sent to the subprogram in the wrong order, can lead to a silly result. However,
if your parameters were of different data types and you passed values in the wrong order, the consequences could be a lot worse. Attempting to store a String value in an Integer or Float variable will result in an error that may even cause the program to stop running.

## The Benefits of Using Arguments and Parameters

Sometimes people who are new to programming wonder why we don't just name the arguments in the main program the same as the parameters in the subprograms. In the sale price computation program, the variables in the Call statement are named `OriginalPrice`, `DiscountRate`, and `SalePrice`. But in the subprogram, `Output_Results`, these variables are named `OldPrice`, `Rate`, and `NewPrice`. We said that this is not necessary—the subprogram would work just as well if its parameters had the same names as the arguments passed to them. However, it is usually preferable to name the parameters of a subprogram independently of the rest of the program.

Using arguments and corresponding parameters to pass data among program modules is an important feature of the programming process for the following reasons:

- It enhances the usefulness of subprograms. They can be designed and coded independently of the main program and even used in several different programs, if desired. Only the structure of the subprogram is important; not the naming of its variables.
- It makes it easier for different programmers to design and code different subprograms. The programmer of a particular subprogram only needs to know what kinds of variables are transmitted to or from that module. He or she doesn't need to be concerned about how these variables are named or used in the main program or in another subprogram.
- It makes it easier to test and debug a subprogram independently of the main program.

We will return to the Sale Price Computation program in Section 7.2, but before completing this section, Example 7.4 provides a simple, yet complete demonstration of the use of subprogram parameters.

## Assigning Data Types to Parameters

Earlier in this section we discussed the importance of passing arguments to the subprogram in the correct order and said that an error would occur if an argument of one type was passed to a parameter of a different type. The values that are passed from the calling program or subprogram are declared in the program or subprogram where they originate. But we have not yet shown how to declare the type of a parameter in pseudocode. In this book, we will use the following pseudocode to assign a type to a parameter:

```
Subprogram Subprogram_Name(String Var1, Integer Var2, Float Var3)
```
Therefore, if Katrina Katz wanted you to write a subprogram that would display the number, color, and type of pet that a customer was buying, the pseudocode for the first line of this subprogram would look as follows:

```
Subprogram Animal(Integer Number, String Color, String Beast)
```

Example 7.4 also demonstrates how to declare the data type of parameters in a subprogram.

**Example 7.4 Assigning Types to Parameters**

This program displays a sequence of asterisks (***** before and after a message (which is stored as a character string) entered by the user. It contains one subprogram (named Surround_And_Display), which imports the input string from the main program and displays that string enclosed within asterisks. In the following pseudocode, we will explicitly declare the necessary program variables:

```
Main
  Declare Message As String
  Write "Enter a short message: 
  Input Message
  Call Surround_And_Display(Message)
End Program
Subprogram Surround_And_Display(String Words)
  Write "***** " + Words + " *****"
End Subprogram
```

**What Happened?**

As usual, execution begins with the first statement in the main program. Thus, when code corresponding to this pseudocode is run, the user is prompted to enter a message, which is assigned to the variable Message. Then, the statement

```
Call Surround_And_Display(Message)
```

calls the subprogram, transferring control to it and assigning the value of the argument Message to the parameter Words. Notice that, in the subprogram header, the data type of the parameter Words is indicated. In this program, the subprogram's Write statement

```
Write "***** " + Words + " *****"
```

displays the value of Message preceded and followed by five asterisks. Then control returns to the next statement in the main program, which is End Program.

**Self Check for Section 7.1**

7.1 Draw a data flow diagram for the following problem: A salesperson's commission is 15 percent of total sales. Input the total sales and compute and display the commission.
Self Check Questions 7.2–7.4 refer to the following pseudocode:

Main
    Declare Name As String
    Declare Age As Integer
    Write "Enter your name: "
    Input Name
    Write "Enter your age as a whole number: "
    Input Age
    Call Voting_Age(String Name, Integer Age)
End Program
Subprogram Voting_Age(String Voter, Integer VoterAge)
    If VoterAge >= 18 Then
        Write Voter + ", you are eligible to vote."
    Else
        Write "Sorry, " + Voter + ", you are too young."
    End Subprogram

7.2 Identify the arguments in the program segment.
7.3 Identify the parameters in the program segment.
7.4 What, if anything, would change in the output if the variables in the Subprogram were changed from Voter and VoterAge to Person and PersonAge?
7.5 Write a program that inputs a name and uses the following subprogram to display the input name three times:
Subprogram Display_Name(String Name)
    Write Name
End Subprogram

7.6 What is the output when code corresponding to the following pseudocode is run?
Main
    Declare Num1, Num2, Num3 As Integer
    Set Num1 = 1
    Set Num2 = 2
    Set Num3 = 3
    Call Display(Num3, Num2, Num1)
End Program
Subprogram Display(Integer Num3, Integer Num2, Integer Num1)
    Write Num3 + " * " + Num2 + " * " + Num1
End Subprogram

7.7 Suppose the subprogram header in Self Check 7.6 were changed to the following:
Subprogram Display(Integer A, Integer B, Integer C)
    a. What changes (if any) should be made to the subprogram body (description)?
    b. What changes (if any) should be made to the main program for the output to be the same as that of Self Check 7.6?
7.2 More about Subprograms

In Section 7.1, we introduced the concepts of subprogram arguments and parameters. In this section, we will delve more deeply into this subject.

Value and Reference Parameters

Previously we used arguments and parameters for the sole purpose of passing data from the main program to a subprogram. Here we will consider the reverse process—exporting data from a subprogram to the main program. This can occur automatically when control is transferred to the main program at the end of the subprogram, but there is an important difference in the way various programming languages implement this action. We illustrate this difference in Example 7.5.

Example 7.5 Changing the Value … but Where?

Consider the following pseudocode:

```plaintext
Main
    Declare NumberOne As Integer
    Set NumberOne = 1
    Call Change_Value(NumberOne)
    Write NumberOne
End Program
Subprogram Change_Value(Integer Number)
    Set Number = 2
End Subprogram
```

What Happened?

The output of code corresponding to this pseudocode will differ, depending on the programming language used. In all languages, the variable NumberOne is set equal to 1 in the main program and this value is transmitted to the parameter Number of the subprogram when the latter is called. Then, the value of Number is changed to 2 in the subprogram and the subprogram ends. What happens next however, depends on the language as follows:

- In some programming languages when control is returned to the main program, the new value of Number is assigned to the corresponding variable, NumberOne, so the output in this case is 2.
- In other programming languages when control is returned to the main program, unless otherwise indicated, changes to the parameter Number do not affect the corresponding variable, NumberOne, in the main program. Hence, the output in this case is 1.

In this book, we will follow the lead of most programming languages and distinguish between the following types of subprogram parameters:

- **Value parameters** have this property: Changes to their values in the subprogram do not affect the value of the corresponding (argument) variables.
in the calling module. These parameters can only be used to import data into a subprogram.

- **Reference parameters** have this property: Changes in their values *do* affect the corresponding arguments in the calling module. They can be used to both **import data** into and **export data** from a subprogram.

**The Inside Story: Pass by Value, Pass by Reference Revealed**

The distinction between value parameters and reference parameters is extremely important because it can have a significant impact on how a program works. When you understand what is happening inside the computer, you will also understand how and when to use each type of parameter.

You already know that when a variable is declared, the computer sets aside a specific location in memory for the contents of that variable. When a variable is passed by value to a submodule, that submodule receives only a copy of that variable. In other words, a separate storage location is created and the value of the variable is stored there also. So now the value of that variable exists in two places. It is in the location where it was stored originally and it is in a second location. The module that sends that variable to a submodule has access to the original location only; the submodule has access to the second location only—the copy. If the submodule does calculations that change the value of that variable, the changes are made only to the copy. When the submodule finishes its work, any changes that have been made to that variable remain in the copied location. The original variable in the main (or calling) module retains its original value.

On the other hand, when a variable is passed by reference, the submodule receives the actual storage location in the computer’s memory where the value of that variable is stored. This means that if the submodule does anything that changes the value of that variable, the value is also changed in the main (or calling) module.

**The Value of Value Parameters**

You may wonder why, if a subprogram changes the value of a parameter, the corresponding argument shouldn’t change in the calling module. It may seem as though value parameters don’t serve any purpose, but there is a good reason why many programming languages support them. Value parameters enhance the independence of subprograms from the main program and each other, and this is a key ingredient in the modular programming method. In particular, the use of value parameters prevents the code in one subprogram from inadvertently affecting the action of the main program or another subprogram through unwanted changes in its variables.

**How to Tell the Difference between Value and Reference Parameters**

Every programming language that distinguishes between value and reference parameters has a way to indicate in the subprogram header whether a parameter...
is of value or reference type. In this book, we will place the symbols As Ref after the parameter name to indicate that it is of reference type, and therefore, changes to it will affect the corresponding argument in the calling module. If these words do not appear, then the parameter is of value type. For example, in the header

```plaintext
Subprogram Switch(Integer Number1, Integer Number2 As Ref)
```

Number1 is a value parameter and Number2 is a reference parameter. Examples 7.6 and 7.7 further illustrate and build on this usage.

### Example 7.6 Passing by Reference, Passing by Value

Consider the following pseudocode:

```plaintext
1  Main
2    Declare MyNumber As Integer
3    Declare YourNumber As Integer
4    Set MyNumber = 156
5    Set YourNumber = 293
6    Call Switch(MyNumber, YourNumber)
7    Write MyNumber + " " + YourNumber
8  End Program
9  Subprogram Switch(Integer Number1, Integer Number2 As Ref)
10   Set Number1 = 293
11   Set Number2 = 156
12  End Subprogram
```

#### What Happened?

- Line 1 tells us that we begin this segment in the Main module. On line 4, the integer variable MyNumber is set equal to the value of 156. On line 5, the integer variable YourNumber is set equal to the value of 293.
- Line 6 transfers control to the subprogram Switch.
- Now we must jump to line 9 to see what is happening. The value of MyNumber is passed into Number1, the first parameter of Switch. The value of YourNumber is passed into Number2, the second parameter of Switch. At this point, the Switch parameters, Number1 and Number2, have been assigned values. Number1 = 156 and Number2 = 293. However, line 9 also adds that Number1 is a value parameter and Number2 is a reference parameter.
- Line 10 changes the value of Number1 to 293 and line 11 changes the value of Number2 to 156.
- Line 12 ends the subprogram, Switch, and control is transferred back to the main program on line 7. Note that control returns to the line directly after the call statement, which in this case, is line 7.
- Line 7 simply writes the values of MyNumber and YourNumber to the screen, separated by a space. However, because Number1 is a value parameter, MyNumber retains the original value (156) it had when the subprogram was called. On the other hand, because Number2 is a reference parameter,
YourNumber changes to the current value of Number2 (156). Thus, the output of code corresponding to this pseudocode is as follows:

156 156

We didn’t actually switch the numbers, did we? To actually switch the values of MyNumber and YourNumber, both Number1 and Number2 must be specified as reference parameters.

**Example 7.7 The Sale Price Computation Program Revisited**

For another example of the use of value and reference parameters, let’s return to the Sale Price Computation program described at the beginning of Section 7.1. For the sake of convenience, we repeat it here:

Katrina Katz owns a small pet supply store. She wants you to design a program that will compute and display the sale price of a discounted store item when she inputs the original price and its percentage discount. Note the following:

- The Input_Data module inputs the pre-sale price and the percentage discount (OriginalPrice and DiscountRate) from the user. These quantities must be exported to the main program; therefore, they must be *reference* parameters. In other words, we want the values of OriginalPrice and DiscountRate that are obtained in the Input_Data module to be retained when they are exported to the Main module to be used in the next step (computing the sale price).

- The Compute_Results module imports these quantities from the main program, computes the sale price (SalePrice) of the item, and exports the latter to the main program. Each time Katrina runs the program, new values for OriginalPrice and DiscountRate will be sent to this module. Since these values are not exported out of this module, they are *value* parameters. However, the SalePrice that is computed here must be exported to the Main module and then used in the Output_Results module. Therefore, this parameter must be a *reference* parameter.

- The Output_Results module imports OriginalPrice, DiscountRate, and SalePrice from the main program and displays their values. All parameters here are *value* parameters because Output_Results does not export any data to the main program.

The pseudocode for the entire program follows:

```
Main
    Declare OriginalPrice As Float
    Declare DiscountRate As Float
    Declare SalePrice As Float
    Call Welcome_Message
    Call Input_Data(OriginalPrice, DiscountRate)
    Call Compute_Results(OriginalPrice, DiscountRate, SalePrice)
    Call Output_Results(OriginalPrice, DiscountRate, SalePrice)
End Program
```
Subprogram Welcome_Message
  Write "This program is a sale price calculator."
  Write "When you enter the original price of an item and how much"
  Write "it has been discounted, the program will display the"
  Write "original price, the discount rate, and the new sale price."
End Subprogram

Subprogram Input_Data(Float Price As Ref,Float Rate As Ref)
  Write "Enter the price of an item: "
  Input Price
  Write "Enter the percentage it is discounted: "
  Input Rate
End Subprogram

Subprogram Compute_Results(OrigPrice,DiscRate,Sale As Ref)
  Declare AmountSaved As Float
  Set AmountSaved = OrigPrice * DiscRate/100
  Set Sale = OrigPrice – AmountSaved
End Subprogram

Subprogram Output_Results(Float OldPrice,Float Rate,Float NewPrice)
  Write "The original price of the item is $ " + OldPrice
  Write "The discount is: " + Rate + "%"
  Write "The sale price of the item is $ " + NewPrice
End Subprogram

What Happened?

- The program starts and control transfers to the Welcome_Message subprogram, which displays a welcome message and transfers control back to the main program.
- Control then transfers to the Input_Data subprogram. Because the Input_Data arguments are undefined (have not been assigned values), the parameters Price and Rate are initially undefined as well. However, they are assigned values by the Input statements in this subprogram, and since both are reference parameters, they are exported to the main program and assigned to OriginalPrice and DiscountRate, respectively.
- Control transfers to the Compute_Results subprogram, in which the value of Sale is computed and returned to the main program in the variable SalePrice.
- Control transfers to the Output_Results subprogram. The values of OriginalPrice, DiscountRate, and SalePrice are assigned to OldPrice, Rate, and NewPrice, respectively, and displayed.
- Control returns to the main program and execution terminates.

To summarize the preceding discussion, when a subprogram is called, the kind of parameter (value or reference) determines the way in which memory is allocated.
for the value of that parameter. If the parameter is of value type, then the following is true:

- A new (temporary) storage location is set up in memory to hold the value of that parameter while the subprogram executes.
- The value of the corresponding argument is copied into this location.
- Whenever the value of the parameter is modified by the subprogram, only the contents of this temporary storage location are changed, so the corresponding argument is unaffected.

If the parameter is of reference type, then the following is true:

- It is assigned the same storage location as the corresponding argument, which in this case must be a variable.
- Whenever the value of the parameter is modified by the subprogram, the contents of this common storage location are changed, so the value of the argument is modified as well.

**Two Helpful Functions: ToUpper() and ToLower()**

Before we take another look at the important difference between passing by value and passing by reference, we introduce two functions that exist in most programming languages: ToUpper() and ToLower().

As you have gone through the exercises in this text, you may have noticed that often the prompt asking a user to enter information is case sensitive. This means that, for example, when the user is asked to indicate by entering “y” or “yes” that he wants to continue, the program will read an entry of “Y” or “YES” as a “no.” In the early days of computer technology, people knew that they must always type in exactly what the computer requested. But today, computers are far more user friendly. Browser search engines forgive misspellings and, unless specifically indicated, it is assumed that you can respond to computer-generated questions without regard to the case (upper or lower). This latter capability is due, in large part, to built-in functions that automatically change a user’s response to either all uppercase or all lowercase. Then the program can continue to evaluate the response and proceed as needed. These two functions are described briefly here and will be used to demonstrate the important distinction between passing parameters by value and by reference.

When a string value or variable is placed inside the parentheses of the **ToUpper()** function, all characters in that string are converted to uppercase. Similarly, when a string value or variable is placed inside the parentheses of the **ToLower()** function, all characters in the string are converted to lowercase. Example 7.8 demonstrates how this works.

There are also functions available in most languages to read only the first letter of a word so that an entry of “yes” would be interpreted as a “y” but we do not need to include those in this small example.
Example 7.8 Using the ToUpper() and ToLower() functions

The following program segment shows two uses for the ToUpper() and ToLower() functions. The user is asked to enter a “Y” for “yes” but many users simply type “y”. The ToUpper() function converts the response to uppercase. The body of the little game in this program will display a box formed of any word the user enters. The Length_Of() function is used to find how many characters are in a word entered by the user and this number is used in the loop that draws the box. The ToLower() function changes all characters in any word entered to all lowercase. The pseudocode for this example is as follows:

1 Declare Response As Character
2 Declare Word As String
3 Declare Box As String
4 Declare Count As Integer
5 Declare X As Integer
6 Write "Do you want to draw a word-box? Enter 'Y' or 'N'"
7 Input Response
8 While ToUpper(Response) == "Y"
9 Write "Enter any word: "
10 Input Word
11 Set X = Length_Of(Word)
12 Set Box = ToLower(Word)
13 Set Count = 1
14 While Count <= X
15 Write Box
16 Set Count = Count + 1
17 End While(Count)
18 Write "Create a new box? Enter 'Y' or 'N'"
19 Input Response
20 End While(Response)

What Happened?

There are several lines of interest in this program. Line 8 uses the ToUpper() function to convert any Response to uppercase. Since ToUpper("Y") yields the same result as ToUpper("y"), the outer loop will be entered regardless of which case the user types. Any other entry is the equivalent to “no.”

Line 12 uses the ToLower() function slightly differently. Here, the result of the function is assigned to a new variable, Box, which is used later in the program. Both uses, as well as the other built-in functions we have discussed, are acceptable.

Here is what this program segment would display, assuming the user enters ‘Y’ for the first input, ‘HelpMe’ for the second input, and ‘N’ for the third:

helpme
helpme
helpme
helpme
helpme
helpme
Example 7.9 illustrates what can happen when one is not careful about passing parameters by value by reference.

**Example 7.9 Pass Those Variables Carefully!**

Natalie and Nicholas are co-presidents of the Gamers at College club (GAC). They have created a Web site and they want the site to be secure. Nick suggests that each member should have a secret login name and Natalie offers to write a program to achieve this. Unfortunately, Natalie did not study this chapter carefully and does not understand the difference between value parameters and reference parameters. She writes the following pseudocode:

```plaintext
1 Main
2 Declare Response As String
3 Declare First As String
4 Declare Last As String
5 Write "Do you want to start? Enter 'yes' or 'no':"
6 Input Response
7 Set Response = ToLower(Response)
8 While Response == "yes"
9   Write "Enter this member's first name:"
10  Input First
11  Write "Enter this member's last name:"
12  Input Last
13  Call Secret_Login(First, Last)
14  Write "Member name: " + First + " * " + Last
15  Write "Enter another member?"
16  Input Response
17  Set Response = ToLower(Response)
18 End While
19 End Program
20 Subprogram Secret_Login(Part1 As Ref, Part2 As Ref)
21 Declare Login As String
22 Declare Temp As String
23 Set Temp = Part1
24 Set Part1 = ToLower(Part2) + "**"
25 Set Part2 = ToLower(Temp)
26 Set Login = Part1 + Part2
27 Write "Your secret login is: " + Login
28 End Subprogram
```

What Happened?

Nick is not impressed with the results of the program and tells Natalie to determine what went wrong. Natalie runs the program twice, entering the names Mary Lamb and Jack Sprat. When she sees the following display she realizes what happened:

```
Your secret login is: lamb**mary
Member name: lamb** mary
```
Your secret login is: sprat**jack
Member name: sprat** jack

Luckily, it takes very little effort to fix the program. Do you know how to fix it? Natalie sent reference parameters to the subprogram Secret_Login. The value of the first name (First) was sent into Part1 but all changes to Part1 affected First. The same is true for Last; since it was sent into Secret_Login as a reference parameter, its value changed as changes were made to Part2 in the subprogram. However, if line 20 were changed to

```
Subprogram Secret_Login(Part1, Part2)
```

then the display, given the entries of Mary Lamb and Jack Sprat, would look as follows:

```
Your secret login is: lamb**mary
Member name: Mary Lamb
Your secret login is: sprat**jack
Member name: Jack Sprat
```

The Scope of a Variable

When a variable is input, processed, or output in a program module, we say that it has been referenced in that module. In certain situations, a variable that is declared in one program module cannot be referenced in another module. Trying to do so will result in an undefined variable error message when the program is compiled. The part of the program in which a given variable can be referenced is called the scope of that variable.

In many programming languages, the scope of a variable declared in a certain program module consists of that module together with all its submodules. We will follow this practice in this book. For example, in the sale price computation program given above:

- The variables OriginalPrice, DiscountRate, and SalePrice are declared in the main program. Because all other program modules are subprograms of the main program, we will consider the scope of these variables to be the entire program; they can be referenced in any module.
- The variable AmountSaved is declared in the Compute_Results subprogram, so its scope is limited to this subprogram; it cannot be used in any other program module.

Global and Local Variables

Notice that, in this book, variables declared in the main program have a scope that is the entire program. Such variables are called global variables. On the other hand, in some programming languages, a variable is only global if it is declared outside of, or prior to, all program modules including the main program.

In the sale price computation program, the variables OriginalPrice, DiscountRate, and SalePrice are global variables. Variables that are declared in a
particular subprogram, such as AmountSaved in the Compute_Results subprogram, are said to be local to that module. **Local variables** have the following properties:

- When the value of a local variable changes in a subprogram, the value of a variable with the same name outside that subprogram remains unchanged.
- When the value of a variable changes elsewhere in a program, a local variable with the same name remains unchanged in its subprogram.

Sometimes local and global variables come into conflict. For example, suppose a variable named MyName is declared in the main program and another variable, also called MyName, is declared in a subprogram. We say that MyName has a *multiple declaration* in the program. The MyName declared in the main program is a global variable and thus may be assigned a value in any subprogram. But the MyName declared in the subprogram is local to that subprogram, so changes to its value do not affect a variable with the same name outside the subprogram. To resolve this conflict, *the local declaration takes precedence*. This means that the value of the main program variable, MyName, is not changed when the subprogram MyName is assigned a new value. Example 7.10 illustrates this situation.

Note that it is always a bad idea to give the same name to two variables, except perhaps for counters. In fact, some languages don’t support global variables at all.

**Example 7.10 Keeping Track of the Value of MyNumber**

Consider the following pseudocode:

```
Main
    Declare MyNumber As Integer
    Set MyNumber = 7654
    Call Any_Sub
    Write MyNumber
End Program
Subprogram Any_Sub
    Declare MyNumber As Integer
    Declare YourNumber As Integer
    Set MyNumber = 2
    Set YourNumber = MyNumber * 3
    Write YourNumber
End Subprogram
```

**What Happened?**

- What will be displayed after this program segment is coded and run? The main program begins by assigning MyNumber the value 7654. Then control is transferred to the subprogram. In the subprogram, MyNumber is a local variable and is set to 2, locally. The value assigned to it inside Any_Sub does not affect the value of the main program’s variable, MyNumber.
- However, the value of the local variable, MyNumber, is valid within the subprogram. Therefore, YourNumber will be set equal to 2*3 since MyNumber,
within the confines of the subprogram, Any_Sub, has the value of 2. The Write statement in Any_Sub will display the number 6.

- When control is returned to the main program, MyNumber still maintains its global value and the Write statement displays the number 7654.

The computer treats local variables in the same way as value parameters. Whenever a local variable is declared, even if it has the same name as a previously declared variable, it’s allocated a new storage location in memory. Thus, in Example 7.10, from the computer’s point of view, the main program MyNumber and the subprogram MyNumber are treated as two different variables. Changing one has no effect on the other.

Use Parameters, Not Global Variables, to Pass Data among Modules

In many programming languages, it’s possible to pass data among modules by making use of global variables. By its nature, a global variable can be imported by or exported from every program module. Nevertheless, it is considered poor programming practice to use global variables for this purpose because doing so diminishes the independence of program modules. The proper way to pass data among program modules is through the use of arguments and parameters, as described in Section 7.1 and earlier in this section.

We can use the property of local variables to our advantage, as demonstrated by the use of counters in Example 7.11.

Example 7.11 Using Counters Locally

Counters are often used in programs and subprograms. Luckily, the value of a counter in a subprogram does not affect the value of a counter in the main program or in another subprogram, as shown by the following pseudocode, which calculates the weekly gross pay (pre-tax) of employees in a small business:

```plaintext
1  Main
2    Declare Name As String
3    Declare NumEmployees As Integer
4    Declare Count As Integer
5    Write "How many employees do you have?"
6    Input NumEmployees
7    For(Count = 1; Count <= NumEmployees; Count++)
8      Write "Enter this employee's name: 
9        Input Name
10     Call Pay_Employee(Name)
11  End For
12 End Program
13 Subprogram Pay_Employee(EmpName)
14    Declare Rate As Float
15    Declare Hours As Float
```
16   Declare Sum As Float
17   Declare Pay As Float
18   Declare Count As Integer
19   Set Sum = 0
20   Write "Enter the pay rate for " + Name
21   Input Rate
22   For (Count = 1; Count <= 7; Count++)
23      Write "Enter hours worked for day " + Count
24      Input Hours
25      Set Sum = Sum + Hours
26   End For
27   Set Pay = Sum * Rate
28   Write "Gross pay this week for " + EmpName
29   Write "is $ " + Pay
30   End Subprogram

What Happened?
This program finds the weekly gross pay for employees. Initially, the user enters the number of employees (NumEmployees) and the For loop on lines 7–11 gets an employee's name (Name) and sends that name to the subprogram (Pay_Employee(EmpName)) for each employee. The subprogram uses a For loop to get the number of hours worked by that employee for seven days (one week). At the end of each time this subprogram is called, the counter, Count, has the value of 8. However, the variable Count in the main program must maintain its original value. Because Count is declared in Main and also declared locally in Pay_Employees, the two values remain separate and changes to the value of Count in the subprogram have no effect on the value of Count in Main.

Self Check for Section 7.2

The following program is used in Self Checks 7.8–7.10

Main
   Declare X, Y, Z As Integer
   Set X = 1
   Set Y = 2
   Set Z = 3
   Call Display(Z,Y,X)
   Write X + " " + Y + " " + Z
End Program
Subprogram Display(Integer Num1,Integer Num2,Integer Num3 As Ref)
   Write Num1 + " " + Num2 + " " + Num3
   Set Num1 = 4
   Set Num2 = 5
   Set Num3 = 6
   Write Num1 + " " + Num2 + " " + Num3
End Subprogram
7.8 What is the output of code corresponding to this pseudocode?

7.9 Suppose that all occurrences of Num1, Num2, and Num3 in this program were changed to X, Y, and Z, respectively. What is the output of code for the modified program?

7.10 Determine the output of code for the given program if its subprogram header were changed as follows:
   a. Subprogram Display(Integer Num1, Integer Num2, Integer Num3)
   b. Subprogram Display(Integer Num1 As Ref, Integer Num2 As Ref, Integer Num3 As Ref)

7.11 What is displayed after the following program segment is coded and run? Assume all variables have been declared as String variables.
   ```
   Set MyName = "Marty"
   Set PetName = "JoJo"
   Write ToUpper(MyName) + " and " + ToLower(PetName)
   ```

7.12 What is the output of code corresponding to the following pseudocode? Assume that variables declared in the main program are global variables.
   a. Main
      ```
      Declare X As Integer
      Set X = 0
      Call Simple()
      Write X
      End Program
      Subprogram Simple()
      Set X = 1
      End Subprogram
      ```
   b. Main
      ```
      Declare X As Integer
      Set X = 0
      Write X
      Call Simple()
      End Program
      Subprogram Simple()
      Set X = 1
      End Subprogram
      ```

7.3 Functions

A function, as we have seen, is a special type of subprogram—one whose name can be assigned a value. In this section, we will discuss built-in functions, which are supplied by the programming language, and user-defined functions, which are program modules created by the programmer.

**Built-in Functions**

Programming languages typically provide a wide assortment of built-in functions. These are often referred to as a library. The code for these functions is supplied
in separate modules, and doesn’t need to be included within your program. In this
text, you have already seen the following examples of built-in functions:

- Sqrt(X) computes the square root of the number X (see Chapter 3).
- Int(X) computes the integer obtained by discarding the fractional part of
  the number X (see Chapter 4).
- Ceiling(X) computes the integer obtained by rounding the number X up
  to the next integer (see Chapter 4).
- Floor(X) computes the integer obtained by discarding the fractional part
  of the number X (see Chapter 4).
- Random generates a random integer (whole number) from 0.0 to 1.0,
  including 0.0 but not 1.0. (see Chapter 5).
- Length_Of(S) computes the length of the string S (see Chapter 5).
- ToUpper(S) changes the value of all characters in a string, S, to uppercase
  (see Section 7.2).
- ToLower(S) changes the value of all characters in a string, S, to lowercase
  (see Section 7.2).

**Built-in functions** can be viewed as subprograms, which normally contain one or
more parameters and return (export) at least one value. As with any subprogram,
the arguments in the call to a built-in function may be constants, variables, or
expressions of the appropriate type. However, built-in functions (including the
ones listed above) differ from the subprograms discussed in Sections 7.1 and 7.2
in the following ways:

1. The header and definition of a built-in function do not appear in the pro-
   gram that calls that function.
2. When a built-in function is called, the function name is assigned a value
   (of the type specified for that function).
3. A built-in function is called by using the function name anywhere in the
   program that a constant of its type is allowed.

For example, the Sqrt() function is of type Float—it is assigned a Float value
when it is called. Sqrt() may be used (called) anywhere in a program that a con-
stant of type Float is allowed. Thus, all of the following are valid calls to the Sqrt()
function (assuming of course that Num is a variable of numeric type and the value
of the argument is not negative):

- Set X = Sqrt(10)
- Write Sqrt(2*(Num+1))
- Call Display(Sqrt(Num))

Here are a few more functions that are commonly built into programming lan-
guages. While the exact name and form of any of the functions given here may dif-
fer from those used in a particular programming language, most common
programming languages have built-in functions that perform the following tasks:

- Abs(X) computes and returns the absolute value of the real number X. The
  absolute value of a given number is the number obtained by ignoring the
  sign, if any, of the given number. This function is of type Float.
- Round(X) rounds the real number X to the nearest whole number and returns that number. It is of type Integer.
- Str(X) converts the number X into the corresponding string and returns that string. It is of type String.
- Val(S,N) converts the string S, if it represents a number, into a number of the appropriate type (Integer or Float) and sets N = 1. If S does not represent a number, this function sets both Val and N equal to 0. Val is of type Float and N is an Integer parameter of reference type.

Example 7.12 provides examples of using built-in functions.

**Example 7.12 Built-in Functions**

The following examples show the result of applying the indicated function to the specified argument:

- \( \text{Abs}(10) \) returns 10
- \( \text{Abs}(-10) \) returns 10
- \( \text{Round}(10.5) \) returns 11
- \( \text{Round}(100 \times 10.443)/100 \) returns 10.44
- \( \text{Str}(31.5) \) returns "31.5"
- \( \text{Str}(-100) \) returns ",-100"
- \( \text{Val}("31.5","N) \) returns the number 31.5 and N = 1
- \( \text{Val}("abc","N) \) returns the number 0 and N = 0

**When a Number Is Not a Number**

Although the string "31.5" and the number 31.5 may look similar, from a programming point of view they are quite different as follows:

- The number 31.5 is stored in memory as the binary equivalent of 31.5. Moreover, because it's of numeric type, it can be added to, subtracted from, divided by, or multiplied by another number.
- The string "31.5" is stored in memory by placing the ASCII codes for "3", "1", ".”, and “5” in consecutive storage locations. Because "31.5" is a string, we cannot perform arithmetic operations on it, but we can concatenate it with another string.

The Val() function is very important in programming because it allows the programmer to convert a string to a numeric value, if appropriate, or signify to the program that the string cannot be converted to a numeric value. There are many programming situations that require this functionality.

Example 7.13 provides an application of the Val() function to the process of data validation. It demonstrates the technique of inputting numbers as strings, which prevents a program crash if the characters entered by the user do not form a valid number.
Example 7.13  The Valuable Val() Function

The following pseudocode prompts the user to enter an integer. It receives the input as a string, checks whether that string represents an integer, and repeats the prompt if it does not.

1  Declare InputString As String
2  Declare N As Integer
3  Declare Number As Integer
4  Repeat
5    Write "Enter an integer: 
6    Input InputString
7    Set Number = Val(InputString,N)
8  Until (N != 0) AND (Number == Int(Number))

What Happened?

The Repeat-Until loop on lines 4–8 validates the data input with the aid of the Val and Int functions. If the string entered by the user is an integer, then both conditions in the Until statement on line 8 are true and the loop is exited. If the input string is not a number, then the first condition is false; if it is a number but not an integer, then the second condition is false. In either of these two cases, the loop is reentered and the user is again asked to input an integer.

Accessing Built-in Functions

Although you don’t need to include the code for a built-in function in the program you are writing, the file that contains that code may have to be linked to your program in order for the program to run. This is usually done by inserting a statement at the beginning of the program that tells the compiler, when it translates the program into machine language, where to locate the function’s code and instructs the compiler to link that code to your program. These files are stored in libraries which are called, normally in the beginning of the program. In programming, a library is a collection of precompiled routines that a program can use.

Following is an example of the beginning of a C++ program. These are calls to C++ libraries that contain functions (and other information) needed for this specific program. These calls are identified, in C++, as preprocessor directives.

```cpp
/** preprocessor directives
#include <iostream>    /** Header for stream I/O (input/output)
#include <string>      /** Header for string type
#include <vector>      /** Header for vector class
#include <cstdlib>     /** Header for standard C library
#include <cctype>      /** Header for CType library functions
```
User-Defined Functions

Programming languages also allow you to create your own function subprograms, which are called user-defined functions. In some languages all subprograms are functions; in others functions are one special type of subprogram. In this book, we consider both subprograms and functions. The difference between a subprogram that is a function and one that is not (sometimes referred to as a procedure) is twofold:

1. A function’s name may be assigned a value (of a specified type) in the code that defines it.
2. A function is called by placing its name (and desired arguments) anywhere in the program that a constant of the function’s specified type is allowed.

In this book, we will begin a function’s header with the word Function (instead of Subprogram). Example 7.14 illustrates these points.

Example 7.14 A Cubic Function

The following program defines and calls a function Cube that imports a number, Side from the main program and returns its cube, \( \text{Side}^3 \), to the main program, where it is displayed:

```
1 Main
2   Declare LittleBox As Float
3   Set LittleBox = Cube(10)
4   Write LittleBox
5 End Program
6 Function Cube(Side) As Float
7   Set Cube = Side^3
8 End Function
```

What Happened?

Line 1 begins the main program. Line 2 declares a Float variable named LittleBox, but no value is assigned to LittleBox. The value of LittleBox is set on the next line and needs a little more explanation.

The call to the function is made on line 3 where the variable, LittleBox, is given the value of what will happen to the number 10 after it has been acted upon by the Cube() function as follows:

```
Set LittleBox = Cube(10)
```

When this statement is executed, control transfers to line 6, to the function subprogram named Cube(). Notice that this function has one parameter, Side. Line 3

\(^3\)Here the word user refers to the programmer (the user of the programming language), not the person executing the program.
passes the value of 10 to the parameter, Side. Notice also that the function header declares the function type—the type for Cube()—as Float. Then, the statement

Set Cube = Side^3

assigns the value 1000 (i.e., $10^3$) to the function. When the function ends, this value is returned to the main program, assigned to LittleBox, and displayed.

**When to Use a Function**

If the programming language you are using contains both functions and non-function subprograms, then either one may be used to implement a given program submodule. Which one you decide to use in a particular instance is a matter of style. Here is a guideline for deciding whether or not to use a function to implement a particular submodule: If the submodule in question computes and returns a single value to the calling module, then implement it with a function. Notice that the function in Example 7.14 satisfies this criterion.

To illustrate this point, and to provide another example of the use of a user-defined function, let’s reexamine the Sale Price Computation program discussed in Sections 7.1 and 7.2. To facilitate our discussion, we will repeat the description for the problem.

Katrina Katz owns a small pet supply store. She has asked you to design a program that will compute and display the sale price of any discounted item in her store when she inputs the original price and the percentage it is discounted.

The data flow diagram in Figure 7.2 makes it clear that the only submodule that returns a single value to the Main module is Compute_Results(). Therefore, following the guideline stated above, we would use subprograms to implement the Welcome_Message, Input_Data(), and Output_Results() modules (as in Sections 7.1 and 7.2), but use a function to implement the Compute_Results() module. The pseudocode for the modified Main program and the Compute_Results() subpro-

---

**Figure 7.2** Data flow diagram for the Sale Price Computation program
Program (which now becomes the NewPrice() function) is given below. The pseudocode for the other three modules is identical to that given in Section 7.2.

```plaintext
Main
Declare OriginalPrice As Float
Declare DiscountRate As Float
Declare SalePrice As Float
Call Welcome_Message
Call Input_Data(OriginalPrice,DiscountRate)
Set SalePrice = NewPrice(OriginalPrice,DiscountRate)
Call Output_Results(OriginalPrice,DiscountRate,SalePrice)
End Program
Function NewPrice(OriginalPrice,DiscountRate) As Float
Declare AmountSaved As Float
Set AmountSaved = OriginalPrice*DiscountRate/100
Set NewPrice = OriginalPrice – AmountSaved
End Function
```

Before we end this section, we will look at Example 7.15, which shows a program that combines the use of parallel arrays and a user-defined function in a longer program.

**Example 7.15 Getting Good Mileage Out of a Function**

Penelope Pinchpenny is concerned about the amount of money she spends on gasoline. She asks you to write a program that will allow her to compare the miles per gallon she uses over ten different driving trips. She wants to compare highway miles with city miles, trips over flat distances and those on curvy hilly roads, and the like. After some careful thought, you come up with the following design:

The output required is a table that identifies the specific type of road trip, the total miles of each trip, and the miles per gallon used for each trip. Parallel arrays are ideal to use in this situation to store this information so you decide you will need the following:

- A String array: TripName[10]
- Two Float arrays: TripMiles[10] and TripMPG[10]

After Penelope enters information for each trip, you need to calculate the miles per gallon. You decide to create a function named Answer() to do this. You will need to send two pieces of information to the function—the number of miles traveled and the number of gallons of gas used. The result of the Answer() function must be stored in the appropriate element in the TripsMPG[] array. Note that we use the symbol \t to indicate a tab stop in the display.

To save space, we assume the following variables have been declared in the beginning of Main, as well as the three arrays previously mentioned: Count As Integer, K As Integer, Name As String, Miles As Float, Gallons As Float

```plaintext
1  Main
2    Set Count = 0
3    While Count < 10
4        Write "Enter a description of this trip: "
```
5    Input Name
  6    Set TripName[Count] = Name
  7    Write "How many miles did you drive? 
  8    Input Miles
  9    Set TripMiles[Count] = Miles
 10   Write "How many gallons of gas did you use on this trip?"
 11   Input Gallons
 12   Set TripMPG[Count] = Answer(Miles, Gallons)
 13   Set Count = Count + 1
 14  End While(Count)
 15  Set K = 0
 16  Write "Trip Name \t Miles Traveled \t MPG"
 17  While K < 10
 18    Write TripName[K] + "\t" + TripMiles[K] + "\t" + TripMPG[K]
 19    Set K = K + 1
 20  End While(K)
 21  End Program
 22  Function Answer(Num1, Num2) As Float
 23    Set Answer = Num1/Num2
 24  End Function

What Happened?

Line 1 begins the main program. The first While loop on lines 3–14 do most of the work. Penelope enters a brief description of a given trip on line 5 and this value is stored in the first element of the TripName[] array. Next, she enters the miles traveled on line 8 and this value is stored in the first element of the TripMiles[] array. Then she enters the number of gallons used on line 11. Line 12 calls the function, Answer() and sends it the present values of Miles and Gallons. These values are sent to the arguments Num1 and Num2 in the function on line 22. The function simply divides the number of miles by the gallons used to find the miles per gallon result. That result is stored in the first element of the TripMPG[] array on line 12. The counter, Count, is then incremented and Penelope is prompted for the next set of data.

When all the data has been entered, the second While loop displays the table. Notice the \t symbol helps format the information.

Getting the Most Out of Functions

You may wonder why we gave the function a generic name like Answer() and why we called the arguments Num1 and Num2 instead of names that indicate their significance. You might be tempted to name the function something that shows its purpose more clearly, like mpg(numMiles, numGals). However, by giving the function a generic name, it can be reused in other programs. For example, this particular function could easily be used in a program to determine the cost of a pound of potatoes if you knew the price of a 10-pound bag or the exam average of a student if you knew the sum of the exam scores and the number of exams. In fact, you will be asked to do this in Self Check Question 7.18.
Self Check for Section 7.3

7.13 Determine the value of the following expressions:
   a. \( \sqrt{4} \)
   b. \( \text{Int}(3.9) \)

7.14 Determine the value of the following expressions:
   a. \( \text{Abs}(0) \)
   b. \( \text{Round}(3.9) \)
   c. \( \text{Str}(0.1) \)
   d. \( \text{Val}("-32",N) \)

7.15 What is the output of code corresponding to the following pseudocode?
   ```plaintext
   Main
   Write F(1,2)
   Write G(-1)
   End Program
   Function F(X,Y) As Integer
   Set F = 5 * X + Y
   End Function
   Function G(X) As Integer
   Set G = X * X
   End Function
   ```

7.16 a. Write a function named Area as follows:
   ```plaintext
   Function Area(L,W) As Float
   that computes the area of a rectangle of length \( L \) and width \( W \).
   ```
   b. Write an assignment statement that calls this function, using any values you want for the length and width.

Self Check Questions 7.17 and 7.18 refer to Example 7.15:

7.17 Add pseudocode to Example 7.15 to include another parallel array to store the number of gallons used on each trip and display this with the rest of the information.

7.18 Write a program that will allow the user to enter three exam scores and, using the Answer(Num1, Num2) function, computes and displays the exam average.

7.4 Recursion

When a subprogram calls itself, the process is called recursion, and the subprogram is said to be recursive. Some programming languages allow recursion; others do not. Recursive algorithms are algorithms that make use of recursive subprograms. Sometimes they can provide quick and simple solutions to complex problems.
The Recursive Process

To illustrate the concept of recursion, let’s consider a simple programming problem: Given a positive integer, \(N\), we will write a function, \(\text{Sum}(N)\), which computes the sum of the first \(N\) positive integers. To solve this problem, we can follow the strategy given in Chapter 4. We will set up a counter-controlled loop in which each successive integer is added to a running total. The pseudocode is as follows:

```
Function Sum(N) As Integer
    Set Total = 0
    For (K = 1; K <= N; K++)
        Set Total = Total + K
    End For
    Set Sum = Total
End Function
```

This pseudocode provides a non-recursive solution to the problem. To solve this problem recursively, we ask the question, “If we know the sum of the first \(N-1\) positive integers, how would we find the sum of the first \(N\) positive integers?” The answer is to add the integer \(N\) to the sum of the first \(N-1\) positive integers as follows:

\[ \text{Sum}(N) = (1 + 2 + \ldots + N - 1) + N \]

or

\[ \text{Sum}(N) = \text{Sum}(N-1) + N \]

For example, if \(N = 4\), then

\[ \text{Sum}(4) = (1 + 2 + 3) + 4 \quad \text{or} \quad \text{Sum}(4) = \text{Sum}(3) + 4 \]

This is the key to the recursive solution of the problem. We have effectively replaced the original problem with a lesser problem. Now we must find the sum of the first \(N-1\) positive integers; that is, we must find \(\text{Sum}(N-1)\). But a similar argument gives us

\[ \text{Sum}(N-1) = \text{Sum}(N-2) + N - 1 \]

and we can continue in this manner until we reach the following:

\[ \text{Sum}(2) = \text{Sum}(1) + 2. \]

Notice that we can’t write \(\text{Sum}(1) = \text{Sum}(0) + 1\), because \(\text{Sum}(0)\) makes no sense—we can’t sum the first zero positive integers! However, since \(\text{Sum}(1)\) is the sum of the first positive integer, we have the following:

\[ \text{Sum}(1) = 1 \]

Thus, we can define the function \(\text{Sum}(N)\) by the following pair of formulas:

- If \(N = 1\), \(\text{Sum}(N) = 1\).
- If \(N > 1\), \(\text{Sum}(N) = \text{Sum}(N-1) + N\).

Then we can use these formulas to compute \(\text{Sum}(N)\) for any value of \(N\). To illustrate the technique, let \(N = 4\); that is, we will find \(\text{Sum}(4)\), the sum of the first four
positive integers. To do so, we apply the formulas above with, successively, \( N = 4 \), \( N = 3 \), \( N = 2 \), and \( N = 1 \). The computation goes as follows:

\[
\begin{align*}
N = 4: & \quad \text{Sum}(4) = \text{Sum}(3) + 4 \\
N = 3: & \quad \text{Sum}(3) = \text{Sum}(2) + 3 \\
\end{align*}
\]

Now, substituting this expression for \( \text{Sum}(3) \) into the line above it gives us the following:

\[
\begin{align*}
N = 4: & \quad \text{Sum}(4) = [\text{Sum}(2) + 3] + 4 = \text{Sum}(2) + 7 \\
N = 2: & \quad \text{Sum}(2) = \text{Sum}(1) + 2 \\
\end{align*}
\]

Substituting this expression for \( \text{Sum}(2) \) into the line above it gives us the following:

\[
\begin{align*}
N = 4: & \quad \text{Sum}(4) = [\text{Sum}(1) + 2] + 7 = \text{Sum}(1) + 9 \\
N = 1: & \quad \text{Sum}(1) = 1 \\
\end{align*}
\]

Substituting this value for \( \text{Sum}(1) \) into the line above it gives us the following:

\[
\begin{align*}
N = 4: & \quad \text{Sum}(4) = 1 + 9 = 10 \\
\end{align*}
\]

Compare this to the straightforward way of finding \( \text{Sum}(4) \) as follows:

\[
\begin{align*}
N = 4: & \quad \text{Sum}(4) = 1 + 2 + 3 + 4 = 10 \\
\end{align*}
\]

The recursive solution shown here may seem awkward and even confusing, but remember that you are looking at it from a human perspective. When a recursive solution is carried out in a program, it may require only a small amount of code that can be executed very quickly. Example 7.16 provides an illustration of this.

**Example 7.16 A Recursive Solution**

The following pseudocode provides a recursive solution to the problem of summing the first \( N \) positive integers, where \( N \) is a given positive integer:

```plaintext
1  Function Sum(N) As Integer
2    If N == 1 Then
3       Set Sum = 1
4    Else
5       Set Sum = Sum(N–1) + N
6    End If
7  End Function
```

**What Happened?**

To help explain how this pseudocode works, suppose that this function is called by the statement

\[
\text{Set Total} = \text{Sum}(4)
\]

where \( \text{Total} \) is an \texttt{Integer} variable that has previously been declared. When this statement is executed, control transfers to the function \( \text{Sum()} \). \( N \) is set equal to 4 (line 1). Then execution proceeds as follows:

- **First call to the function:** Because \( N = 4 \), lines 2 and 3 are not executed and control goes to the \texttt{Else} clause on line 4. Then line 5 is executed. The right side of the assignment statement
  \[
  \text{Set Sum} = \text{Sum}(N-1) + N
  \]
is evaluated. At this point, since \( N = 4 \) and \( (N-1) = 3 \), this gives us the following for the right side:

\[
\text{Sum}(3) + 4.
\]

But, before this expression can be assigned to the variable \text{Sum}, the function \text{Sum}(3) causes a call to the function \text{Sum()} with \( N = 3 \).

- **Second call to the function:** This is actually where the function calls itself. On this second call, lines 2 and 3 are still skipped because \( N \) does not equal 1. However, the \texttt{Else} clause is executed now with \( N = 3 \). At this point, \( (N-1) = 2 \), so when the right side of the assignment statement is evaluated this time, we get

\[
\text{Sum}(2) + 3.
\]

which causes the function to be called again, this time with \( N = 2 \).

- **Third call to the function:** Lines 2 and 3 are skipped again during this call and the \texttt{Else} clause is executed with \( N = 2 \). The right side of the assignment statement is evaluated, which gives us the following:

\[
\text{Sum}(1) + 2.
\]

Now the function calls itself one more time, with \( N = 1 \).

- **Fourth (and last) call to the function:** Since \( N = 1 \), this time the \texttt{Then} clause on lines 2 and 3 is executed and \text{Sum} is set equal to 1. In this case, the function does not call itself and execution of this call to the function is completed.

- **Control now returns to the assignment statement on line 5 as**

\[
\text{Set Sum} = \text{Sum}(1) + 2
\]

in the *third call to the function* (where the last call to the function was made). Here \text{Sum}(1) is replaced by 1 and \text{Sum} (on the left side) takes on the value 3. Execution of the third call is now complete.

- **Control now returns to the assignment statement as**

\[
\text{Set Sum} = \text{Sum}(2) + 3
\]

in the *second call to the function*. Here \text{Sum}(2) is replaced by 3 and \text{Sum} (on the left side) takes on the value 6. Execution of the second call is now complete.

- **Finally control returns to the assignment statement as**

\[
\text{Set Sum} = \text{Sum}(3) + 4
\]

in the *first call to the function*. Here \text{Sum}(3) is replaced by 6 and \text{Sum} (on the left side) takes on the value 10. Execution of the first call is now complete and \text{Total} (in the initial calling statement) is set equal to 10.

Table 7.1 summarizes the action of the function \text{Sum()} if it is called from the main program with \( N = 4 \).
Chapter 7 • Program Modules, Subprograms, and Functions

Table 7.1 Calling Sum(N) with N = 4

<table>
<thead>
<tr>
<th>If Execution Is Here</th>
<th>Value of N</th>
<th>Value of Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of execution of first call to Sum()</td>
<td>4</td>
<td>Undefined</td>
</tr>
<tr>
<td>Start of execution of second call to Sum()</td>
<td>3</td>
<td>Undefined</td>
</tr>
<tr>
<td>Start of execution of third call to Sum()</td>
<td>2</td>
<td>Undefined</td>
</tr>
<tr>
<td>Start of execution of fourth call to Sum()</td>
<td>1</td>
<td>Undefined</td>
</tr>
<tr>
<td>End of execution of fourth call to Sum()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>End of execution of third call to Sum()</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>End of execution of second call to Sum()</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>End of execution of first call to Sum()</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

Example 7.17 provides another example of recursion.

**Example 7.17 Using Recursion to Do Exponentiation**

We will write a recursive function that finds the Nth power, \(X^N\), of the number \(X\), where \(N\) is a given positive integer. We name this function \(\text{Power}(\cdot)\). To apply recursion, we have to express \(X^N\) in terms of \(X^{N-1}\):

\[X^N = X \times X^{N-1}\]

Thus, if we call our function \(\text{Power}(X,N)\), where \(X\) represents the base and \(N\) represents the exponent, we have the following:

\[
\text{If } N > 1, \quad \text{Power}(X,N) = X \times \text{Power}(X,N-1) \\
\text{If } N = 1, \quad \text{Power}(X,N) = X \quad (\text{since } X^1 = X)
\]

For example, when \(X = 2\) and \(N = 5\):

\[2^5 = 2 \times 2 \times 2 \times 2 \times 2 = 2 \times (2 \times 2 \times 2 \times 2) = 2 \times 2^4\]

Therefore, we can replace the problem of finding the \(N\)th power of \(X\) with that of finding the \((N-1)\)th power of \(X\). This is the key to a recursive solution. Adapting the pseudocode of Example 7.16 to the current situation, we have the following:

1. Function \(\text{Power}(X,N)\) As Float
2. If \(N == 1\) Then
3. Set Power = X
4. Else
5. Set Power = \(\text{Power}(X,N-1) \times X\)
6. End If
7. End Function

Let’s trace the execution of this function when it is called by the following statement:

\[\text{Set } Answer = \text{Power}(5, 3)\]
What Happened?

This will assign the value $5^3 = 125$ to the variable Answer. The assignment statement transfers control to the function Power(), where X is set equal to 5 and N is set equal to 3. This is the first call to the recursive function, which calls itself a total of three times, as follows:

- The first call to the function begins on line 1: However, since $N = 3$, lines 2 and 3 are skipped and the Else clause on line 5 is executed. The right side of the assignment statement,
  
  \[
  \text{Set } \text{Power} = \text{Power}(X,N-1) \times X
  \]

  is evaluated. The right side yields the following:
  
  \[
  \text{Power}(5,2) \times 5
  \]
  
  and causes a second call to the function Power() with $X = 5$ and $N = 2$.

- The second call to the function is where the function calls itself the first time: The Else clause on line 5 is now executed again with $N = 2$ and the right side of the assignment statement is evaluated, which yields the following:
  
  \[
  \text{Power}(5,1) \times 5
  \]

  and causes Power() to be called again with $X = 5$ and $N = 1$.

- The function calls itself once more. On this third (and last) call to the function: Since $N = 1$ in this function call, the Then clause on line 3 is executed and Power is set equal to 5. The function does not call itself here, and execution of the third call to the Power() function is complete.

- Control now returns to the right side of the assignment statement on line 5 as follows:
  
  \[
  \text{Set } \text{Power} = \text{Power}(5,1) \times 5
  \]

  in the second call to the function (where the third call was made). Here, Power(5,1) is replaced by 5 and Power (on the left side) takes on the value 25. Execution of the second call is now complete.

- Control now returns to the right side of the assignment statement (still line 5) as follows:
  
  \[
  \text{Set } \text{Power} = \text{Power}(5,2) \times 5
  \]

  in the first call to the function. Here, Power(5,2) is replaced by 25 and Power (on the left side) takes on the value 125. Execution of the first call is now complete and Answer is set equal to 125.

Self Check for Section 7.4

Use the following information for Self Check Questions 7.19–7.22:

The factorial of a positive integer $N$ is denoted by $N!$ and is read “$N$ factorial.” It is defined to be the product of the first $N$ positive integers as follows: $N! = 1 \times 2 \ldots \times N$

In particular, if $N = 1$, then $N! = 1$. Also, $2! = 1 \times 2 = 2$, $3! = 1 \times 2 \times 3 = 6$, and so on. “Eight factorial” is written $8!$ and $8! = 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$. 
Chapter 7  •  Program Modules, Subprograms, and Functions

7.19 Express \( N! \) in terms of \((N-1)!\).

7.20 Write a recursive function, \( \text{Factorial}(N) \), which computes and returns \( N! \).

7.21 Trace the action of the function \( \text{Factorial}(N) \), similarly to that of Example 7.16, if it is called by the following statement:

\[
\text{Set Answer = Factorial(3)}
\]

7.22 Use a \textit{For} loop to write a non-recursive function, \( \text{Fac}(N) \), which computes and returns \( N! \).

7.5 \textbf{Focus on Problem Solving: A Grade Management Program\textsuperscript{4}}

In this section, we will develop a program that not only makes use of the concept of subprogram parameters covered in this chapter, but also draws upon material from prior chapters and from the material on sequential data files in Chapter 8. In order to understand this program fully, it is necessary to read the material in Chapter 8 first. This is a menu-driven program that creates and modifies a sequential file with the aid of arrays.

\textbf{Problem Statement}

One of your instructors, Professor Allknowing, would like to have a program that creates an electronic grade sheet so that he can monitor the scores of his students. This program should allow the professor to enter the names of his students and their scores on three tests. It should also compute the average test score for each student and display this information when Professor Allknowing wants to review it.

\textbf{Problem Analysis}

This program needs a file, \( \text{grades} \), to hold the required information—name, test scores, and test average—for each student in the class (see Chapter 8). Thus, this file must have records of the following form:

\[
\text{name, test 1 score, test 2 score, test 3 score, average}
\]

To create the file \( \text{grades} \), we must input the student names and test scores (although not necessarily all at once) from the user and compute the test average for each student. This necessitates the use of the following program variables—\( \text{Name, Score1, Score2, Score3, and Average} \)—where

\[
\text{Average} = \frac{(\text{Score1} + \text{Score2} + \text{Score3})}{3}
\]

The output of this program is the display of the contents of the \( \text{grades} \) file. To create and display the file, we will use the techniques described in Chapter 8.

\textbf{Program Design}

To determine the basic tasks that this program must perform, imagine that you are the instructor. The first thing you would have to do, probably at the beginning of

\textsuperscript{4}This section uses material from Chapter 8 extensively. It is strongly recommended that you complete Chapter 8 before beginning Section 7.5.
the semester, is to create the file `grades` and enter the names of all students into this file. Then, after each test is given, the scores of all students on that test would be entered into the file. Moreover, at any time, you might want to view the contents of the file. Thus, the program’s basic tasks are as follows:

1. Create the `grades` file with the names of the students in the class.
2. Enter scores for all students on a particular test into the file.
3. Compute the average test score for each student after all scores have been entered and enter these averages into the file.
4. Display the contents of the `grades` file.

In light of the nature of these tasks, we will design a menu-driven program with options corresponding to the four tasks listed above, as well as an option to quit the program. This necessitates another basic task, as follows, which must be performed prior to the others:

0. Allow the user to select an option from the menu.

In task 1 we write data to `grades`. In tasks 2 and 3 we read each student’s name from `grades` and then write the corresponding test score or average to this file. In task 4 we read and display the entire `grades` file. Thus, before executing task 2, 3, or 4, it would simplify the process if we loaded `grades` into arrays. Following this line of reasoning we will place tasks 2, 3, and 4 in a submodule called `Retrieve_Grade_Sheet`. This submodule will perform certain housekeeping tasks, such as opening `grades` for input, loading it into arrays, and closing it when done. This leads to the following program modularization:

```plaintext
Main module
   Select_From_Menu() module
   Create_Grade_Sheet module
   Retrieve_Grade_Sheet() module
      Enter_Test_Scores() module
      Compute_Averages() module
      Display_Grade_Sheet() module
```

The hierarchy chart shown in Figure 7.3 shows the relationships among these modules. A rough pseudocode outline for each module appears below.

### Main Module

The Main module, after displaying a welcome message, must call the `Select_From_Menu()` module to determine which of its other submodules to call. Here is its rough pseudocode:

```plaintext
Main
   Display a welcome message
   Call Select_From_Menu() module
   Call either Create_Grade_Sheet or Retrieve_Grade_Sheet() module
End Program
```
**Select From Menu() Module**

This module displays the menu of options and inputs the user's selection. A rough pseudocode outline is as follows:

- Display menu options
  - Create Grade Sheet
  - Enter Test Scores
  - Compute Test Averages
  - Display Grade Sheet
  - Quit

  Input user selection, Choice

This submodule exports (returns) the value of Choice to the Main module.

**Create Grade Sheet Module**

This module creates the file grades used by this program; it inputs the student names and initializes (for each student) all three test scores and the test average to 0. (See Chapter 8 for information on creating a sequential file.) Its pseudocode is as follows:

```
Open "grades" For Output As DataFile
Write "Enter student name; enter * when done."
Input Name
While Name != "*"
  Write DataFile, Name, 0, 0, 0
  Write "Enter student name; enter * when done."
```
Input **Name**
End While
Close **DataFile**

No data is imported or exported by this submodule.

**Retrieve Grade Sheet() Module**

This module opens the existing grades file (for input) and loads it into parallel arrays. There are three parallel arrays—one for student names, one for student averages, and a two-dimensional array for the three test scores. Then there is a call to either the Enter_Test_Scores() module, the Compute_Averages() module, or the Display_Grade_Sheet() module. The module that is called depends on the user's menu selection, Choice. Finally, the file is closed when control returns from the called module. (Reading a sequential file and loading it into arrays are discussed in Chapter 8.) The pseudocode for this module is as follows:

```
Open "grades" For Input As DataFile
Declare Names[40], Averages[40], Scores[3,40]
Declare Count As Integer
Set Count = 0
While NOT EOF(DataFile)
    Read DataFile, Names[Count], Scores[0,Count], Scores[1,Count], Scores[2,Count], Averages[Count]
    Set Count = Count + 1
End While
/* Decrement Count by 1 so it now holds the value of the highest subscript in the arrays */
Set Count = Count – 1
/* Depending on value of Choice Call Enter_Test_Scores, Compute_Averages, or Display_Grade_Sheet module */
Close DataFile
```

This submodule needs to import the value of **Choice** from the Main module and export the value of **Count** (which is one less than the number of students in the class, or, as we know, the highest subscript in the arrays) as well as possibly the arrays Names, Scores, and/or Averages to the submodule it calls.

**Enter_Test_Scores() Module**

This module determines which test score (1, 2, or 3) is to be entered, then successively displays each student name, and inputs the corresponding score. The pseudocode must account for the fact that the first test score, TestNum = 1, of the first student is stored in Scores as Scores[0,0], the third test score of the fifth student is stored as Scores[2,4], and so forth. (Two-dimensional arrays are discussed in Chapter 6.)

```
Write "Enter the test number:"
Input **TestNum**
Write "When a name is displayed, enter the test score."
```
Chapter 7  •  Program Modules, Subprograms, and Functions

Program Modules, Subprograms, and Functions

For (K = 0; K <= Count; K++)
Write Names[K]
Input Scores[TestNum - 1,K]
End For

This submodule imports the value of Count and the array Names from the Retrieve_Grade_Sheet() module and imports and exports the array Scores.

Compute_Averages() Module

This module computes the test average for each student by summing the three test scores and dividing by three.

For (K = 0; K <= Count; K++)
Set Sum = Scores[0,K] + Scores[1,K] + Scores[2,K]
Set Averages[K] = Sum/3
End For

This submodule imports the value of Count and the array Scores from the Retrieve_Grade_Sheet() module and exports the array Averages to this module.

Display_Grade_Sheet() Module

This module displays the contents of the grades file which, when Display_Grade_Sheet() is called, has already been loaded into arrays as follows:

For (K = 0; K <= Count; K++)
End For

This submodule imports the value of Count and the arrays Names, Scores, and Averages from the Retrieve_Grade_Sheet() module. It does not export any data.

We can now turn the hierarchy chart of Figure 7.3 into the data flow diagram shown in Figure 7.4. Refer to the discussion at the end of each submodule's pseudocode to see what data is imported and exported from each submodule.

Next, we will refine the pseudocode for several of the program modules, making use of the subprogram arguments/parameters notation introduced in Sections 7.1 and 7.2. Recall that variables exported from a subprogram must be indicated as reference (i.e., As Ref) variables. We will also indicate which subprogram variables are local by placing the appropriate Declare statements in the program modules. The following is a rundown on the refinements:

- In the Main module (the main program), we place the calls to its submodules in a loop so that more than one task may be accomplished in each program run.
- In the Select_From_Menu() module (subprogram), we validate the user's selection to ensure that it is in the proper range.
- In the Retrieve_Grade_Sheet() subprogram, we explicitly specify which of its three submodules is called.
The refined pseudocode for the entire program follows:

```plaintext
Main
    Declare Choice As Integer
    Display a welcome message
    Repeat
        Call Select_From_Menu(Choice)
        If Choice == 1 Then
            Call Create_Grade_Sheet
        End If
        If (Choice == 2) OR (Choice == 3) OR (Choice == 4) Then
            Call Retrieve_Grade_Sheet(Choice)
        End If
    Until Choice == 0
End Program
```
Subprogram Select_From_Menu(Choice As Ref)
    Repeat
        Write "0 – Quit the program"
        Write "1 – Create grade sheet"
        Write "2 – Enter test scores"
        Write "3 – Compute test averages"
        Write "4 – Display grade sheet"
        Input Choice
    Until (Choice == Int(Choice)) AND (Choice >= 0) AND (Choice <= 4)
End Subprogram

Subprogram Create_Grade_Sheet
    Declare Name As String
    Open "grades" For Output As DataFile
    Write "Enter student name; enter * when done."
    Input Name
    While Name != "**"
        Write DataFile, Name, 0, 0, 0, 0
        Write "Enter student name; enter * when done."
        Input Name
    End While
    Close DataFile
End Subprogram

Subprogram Retrieve_Grade_Sheet(Choice)
    Open "grades" For Input As DataFile
    Declare Names[40], Averages[40], Scores[3,40]
    Declare Count As Integer
    Set Count = 0
    While NOT EOF(DataFile)
        Read DataFile, Names[Count], Scores[0,Count], Scores[1,Count], Scores[2,Count], Averages[Count]
        Set Count = Count + 1
    End While
    Set Count = Count – 1
    Select Case Of Choice
        Case 2:
            Call Enter_Test_Scores(Count, Names, Scores)
            Break
        Case 3:
            Call Compute_Averages(Count, Scores, Averages)
            Break
        Case 4:
            Call Display_Grade_Sheet(Count, Names, Scores, Averages)
            Break
        Default
            Write "Invalid entry"
    End Case
    Close DataFile
End Subprogram
Subprogram Enter_Test_Scores(Count, Names, Scores As Ref)
Declare TestNum, K As Integer
Do
    Write "Enter the test number:"
    Input TestNum
While (TestNum < 1) OR (TestNum > 3)
Write "When a name is displayed, enter the test score."
For (K = 0; K <= Count; K++)
    Write Names[K]
    Input Scores[TestNum - 1,K]
End For
End Subprogram

Subprogram Compute_Averages(Count, Scores, Averages As Ref)
Declare Sum, K As Integer
For (K = 0; K <= Count; K++)
    Set Sum = Scores[0,K] + Scores[1,K] + Scores[2,K]
    Set Averages[K] = Sum/3
End For
End Subprogram

Subprogram Display_Grade_Sheet(Count, Names, Scores, Averages)
Declare K As Integer
Write "Each student's name is displayed followed by "
Write "the 3 test scores and the student's test average."
// The tab stop (\t) is used to format columns in the display
Write "Student Name \t Exam 1 \t Exam 2 \t Exam 3 \t Average"
For (K = 0; K <= Count; K++)
End For
End Subprogram

Program Code
The program code is now written using the design as a guide. At this stage, header comments and step comments are inserted into each module to provide internal documentation for the program. Here are several other points concerning the coding that are specific to this program:

- The main menu and the grade sheet should be displayed on blank screens. Recall that this is accomplished by using the programming language's clear screen statement. The clear screen statement should be the first statement in the Repeat-Until loop in the main program and the first statement in the Display_Grade_Sheet subprogram.
- To display the contents of the grades file in a neat, readable fashion, the output produced by the Display_Grade_Sheet module must be formatted. The names, test scores, and averages should line up in columns. This can be accomplished using the programming language's special print formatting statements if the tab stops are insufficient.
Another way to improve the output’s readability is to insert a blank line after displaying every few lines of student information. This can be done with the aid of the Int() function by changing the For loop in the Display_Grade_Sheet() subprogram as follows:

```
For (K = 0; K <= Count; K++)
    Write Names[K] + "\t" + Scores[0,K] + "\t" + Scores[1,K] + "\t"
    + Scores[2,K] + "\t" + Averages[K]
If (Count + 1)/3 == Int((Count + 1)/3) Then
    Write
End If
End For
```

This will put a blank line after every third record. When (Count + 1) equals any multiple of 3, (Count + 1)/3 will be an integer. Otherwise, (Count + 1)/3 will not be an integer. Also, recall that in the pseudocode in this book, the statement consisting of simply the word Write moves the cursor to the beginning of the next line. (Each programming language has a way to code a new line statement.) Thus, this pseudocode causes a line to be skipped after every three lines of output. Can you see why we start at Count + 1 instead of simply Count? Since the first value of Count is 0, Int(0) is an integer and a blank line would be displayed after the first record.

There is another, more subtle, problem with the display of the grade sheet. Immediately after it is displayed, control returns to the Retrieve_Grade_Sheet() subprogram and then to the main program. In the latter, the clear screen statement (see the first bullet above) will immediately remove the grade sheet from the screen, probably before it can be read. To delay returning from the Display_Grade_Sheet() subprogram so the user will have time to read the display, place code corresponding to the following pseudocode statements at the end of the Display_Grade_Sheet() module. The corresponding text will appear under the grade sheet:

```
Write "To return to the main menu, press the Enter key."
Input Response
```

The variable, Response, must be of Character or String type. Then execution will pause until the user presses Enter and the grade sheet can be viewed leisurely.

**Program Test**

The best way to test this program is to imagine that you are the instructor of a very small class (say, two or three students), and use the program to create the grade sheet, enter scores for each of the three tests, and compute the test score averages. After performing each of these tasks, display the grade sheet to check that the operation has been executed successively. In the course of doing this testing, you should also enter invalid values for the menu option (in the Select_From_Menu() submodule) and the test number (in the Enter_Test_Scores() submodule).
As you begin to test a program with data you create, you should think carefully about what data you enter. Most programmers write programs for other users, not just themselves. So think about what types of data a non-programmer might enter by mistake and be sure your program can deal with it. In this example, think about how your program would handle numbers entered where strings are to be input and string data entered where numbers are requested. Normally, in the first case, where a number is entered when a string is requested, the consequence is minor. Professor Allknowing might end up with a student named 96, but otherwise there would be no harm done. The program would simply accept the number as if it were a string of characters. On the other hand, the consequences of entering a string when a number is needed would be more serious. The Val(S,N) function can be used to test for this. Recall from Example 7.3 that this function converts the string S, if it represents a number, into a number of the appropriate type (Integer or Float) and sets N = 1. If S does not represent a number, this function sets both Val and N equal to 0.

As you enter test data, try these options to be sure your program is coded to deal with them. Always test for a division by zero error. It is also a good idea to test numerical input with both positive and negative numbers. The more “bad” data you use to test a program, the better your program will be!

Self Check for Section 7.5

Self Check Questions 7.23 and 7.24 refer to the Grade Management program of this section.

7.23 Suppose we want to create a subprogram, Display_Student_Record(), as a submodule of the Retrieve_Grade_Sheet module. This subprogram should input a student name from the user and display that student’s name, test scores, and average.
   a. What data must be imported to and exported from this subprogram?
   b. Write pseudocode for this subprogram.

7.24 Suppose we want to create a subprogram, Change_Test_Score(), as a submodule of the Retrieve_Grade_Sheet() module. This subprogram should input a student name, test number (1, 2, or 3), and test score from the user and change the appropriate test score to the one input and correct the test average for that student.
   a. What data must be imported to and exported from this subprogram?
   b. Write the header for this subprogram.
Key Terms

argument  pass by value
built-in functions preprocessor directive
data flow diagram recursion
export data recursive algorithm
function reference parameter
global variable return a value
header (of subprogram) scope (of a variable)
import data subprocess
library (of functions) 
local variable 
parameter 
pass a value 
pass by reference 

Chapter Summary

In this chapter, we have discussed the following topics:

1. Parameters and arguments:
   - Data transmitted from one program module to another is *exported* from the former and *imported* by the latter.
   - A *data flow diagram* shows the relationships among the program modules and indicates the data imported and exported by each module.
   - To pass data from a module to a submodule, the *Call* statement (in the former) contains *arguments* and the subprogram header (in the latter) contains *parameters*.
   - The number and type of arguments in a *Call* statement must be the same as the number and type of parameters in the corresponding header; the data is passed from an argument to a parameter based solely on its position in the argument and parameter list.

2. Value and reference parameters:
   - A change in the value of a *value parameter* in its subprogram does *not* affect the value of the corresponding argument, but a change in the value of a *reference parameter* in its subprogram *does* change the value of the corresponding argument.
   - Value parameters are used to import data into a subprogram.
   - Reference parameters are used to export (or to import and export) data from a subprogram.
   - To indicate that a variable in a subprogram’s parameter list is of reference type, this book follows its name by the symbols *As Ref.*
   - The *ToUpper()* and *ToLower()* functions are used to make more user-friendly programs.
3. Scope of a variable:
   - The scope of a variable is the part of the program in which that variable can be referenced (used).
   - The scope of a global variable is the entire program.
   - The scope of local variable is the subprogram in which it is declared and any subprograms within that subprogram.
   - If the same variable is declared locally and globally, it is treated as if it were two different variables, and the local declaration takes precedence.

4. Functions:
   - A function is a subprogram whose name returns a value to the calling subprogram. The function name may appear anywhere in the program that a constant of that type is valid.
   - Built-in functions are supplied with the programming language software; in this chapter, we introduced the following built-in functions, as well as ToUpper(S) and ToLower(S):
     - Abs(X) returns the absolute value of a number X.
     - Round(X) rounds X to the nearest whole number.
     - Str(X) converts X from a number to its string equivalent.
     - Val(S,N) converts the string S into a number.
   - User-defined functions are functions created by the programmer; they are subprograms whose name may be assigned a value, and are called in the same way as built-in functions.

5. Recursion:
   - A recursive subprogram is a subprogram that calls itself.
   - To create a recursive function, F(N), which computes an expression that depends on a positive integer N, you must do the following:
     - Express F(N) in terms of F(N–1).
     - Determine the value of F(1).
     - In the function definition, use an If-Then-Else statement that sets F equal to F(1) if N = 1, and which involves F(N–1) if N > 1.

Review Exercises

Fill in the Blank

1. If a data item is transmitted from a subprogram to the main program, it is said to be returned to or ____________ to the main program.

2. If a data item is transmitted from the main program to a subprogram, it is said to be passed to or ____________ by the subprogram.

3. A diagram that shows the data transmitted between program modules is called a(n) ____________.

4. A chart that shows the data that is imported to, processed by, and exported from each program module is called a(n) ____________.

5. The part of the program in which a given variable can be referenced is called the ____________ of that variable.
6. The scope of a(n) _________ variable is the entire program.
7. A(n) _________ is a type of subprogram whose name may be assigned a value.
8. A(n) _________ function is one that is supplied by the programming language; its code does not appear in the program that uses it.
9. When a subprogram calls itself, the process is called _________.
10. If \( N = 2 \) and \( \text{Sum}(N) \) is a function with \( \text{Sum}(1) = 5 \), then the statement
    
    \[
    \text{Set } \text{Sum} = \text{Sum}(N-1) + N
    \]
    
    assigns the value _________ to \( \text{Sum} \).

**True or False**

11. T  F Changes to a value parameter in a subprogram affect the corresponding argument in the calling module.
12. T  F Changes to a reference parameter in a subprogram affect the corresponding argument in the calling module.
13. T  F The \( \text{ToUpper()} \) and \( \text{ToLower()} \) functions can take a variable of any data type.
14. T  F The statement: \( \text{Display ToUpper(“Yes”)} \) is not allowed because “Y” is already uppercase.

**Short Answer**

Exercises 15–20 refer to the following program:

```
Main
    Declare X As Integer
    Set X = 1
    Call Display(2*X,X,5)
End Program
```

Subprogram Display(Integer Num1,Integer Num2,Integer Num3)
    Write Num1 + " " + Num2 + " " + Num3
End Subprogram

15. What data is passed from the main program to the subprogram?
16. What data is imported by the subprogram Display?
17. Draw a data flow diagram for this program.
18. If code corresponding to this pseudocode is run but \( X \) is initially set to 4, what is the output of this program?
19. If code corresponding to this pseudocode is run, what is the output of this program?
20. Suppose that in the subprogram, all occurrences of \( \text{Num1} \), \( \text{Num2} \), and \( \text{Num3} \) were replaced by \( A \), \( B \), and \( C \), respectively. If code corresponding to the resulting pseudocode were run, what would be the output now?
21. Write a subprogram called \( \text{Input_Data()} \) that inputs two numbers from the user and exports their values to the main program.
22. Write a subprogram called Flip() that imports two variables from the main program (into parameters \( X \) and \( Y \)), interchanges their values, and exports the results to the main program.

Exercises 23–28 refer to the following program. Assume, as we have been doing in this book, that variables declared in the main program are global.

**Main**

- Declare \( X \) As Integer
- Declare \( Y \) As Integer
- Set \( X = 1 \)
- Set \( Y = 2 \)
- Call Sub(\( X, Y \))
- Write \( X \)
- Write \( Y \)

**End Program**

**Subprogram** Sub(Integer Num1,Integer Num2 As Ref)

- Declare \( X \) As Integer
- Set Num1 = 3
- Set Num2 = 4
- Set \( X = 5 \)
- Write \( X \)

**End Subprogram**

23. What is the scope of the following variables?

- a. \( X \), declared in the main program?
- b. \( X \), declared in the subprogram?

24. List the local and global variables in this program.

25. List the value and reference parameters in the subprogram Sub.

26. What is the output of this program if code corresponding to this pseudocode is run?

27. Suppose the subprogram header is changed to the following:

   **Subprogram** Sub(Integer Num1 As Ref,Integer Num2)

   If code corresponding to the new pseudocode is run, what is the output now?

28. Suppose the subprogram header is changed to the following:

   **Subprogram** Sub(Integer Num1,Integer Num2)

   If code corresponding to the new pseudocode is run, what is the output now?

In Exercises 29–35, give the value returned by the built-in function. (These functions were introduced in Sections 7.2 and 7.3.)

29. a. \( \text{Abs}(0) \)
    
   b. \( \text{Abs}(-1.5) \)

30. a. \( \text{Round}(3.8) \)
    
   b. \( \text{Round}(	ext{Abs}(-1.4)) \)
31. a. `Str(10.5)`
   b. `Val("ten",N)`

32. a. `Str(Val("-1.5",N))`
   b. `Val(Str(87.6),N)`

33. a. `ToUpper("N")`
   b. `ToUpper("Nancy Newley")`

34. a. `ToLower("N")`
   b. `Length_Of(ToLower(Name)),` where `Name` is a String variable and `Name = "Nancy Newley"`

35. Given the Float variables: `Charge = -87.23` and `Cost = 456.87`
   a. `Val(Str(Cost),N)`
   b. `Abs(Round(Charge))`

36. Refer to Example 7.15. Add the following functionality to this program:
    Allow the user to enter the cost of a gallon of gas on each trip and use a
    function, `Cost()` to calculate the cost of purchasing gas for that trip.
    Display this information in the table with the rest of the information. You
    will need to add another parallel array, `TripCost[]`, to hold this informa-
    tion.

37. Suppose a program contains the following function:

   ```
   Function F(X) As Float
       Set F = X + 1
   End Function
   ```

   What is displayed when the statement
   ```
   Write F(3)
   ```
   in the main program is executed?

38. Given the function of Exercise 37, what is displayed when the statement:
   ```
   Write F(F(0))
   ```
   in the main program is executed?

39. Suppose a program contains the following function:

   ```
   Function G(X,Y) As Float
       Set G = X + Y
   End Function
   ```

   What is displayed when the statement
   ```
   Write G(4,5)
   ```
   in the main program is executed?

40. Given the functions of Exercises 37 and 39, what is displayed when the
    statement
    ```
    Write G(1,F(1))
    ```
    in the main program is executed?
41. Write a function
   Function Average(Num1,Num2) As Float
   that finds the mean, \((\text{Num1} + \text{Num2})/2\), of the numbers \text{Num1} and \text{Num2}.

42. Write a Main module (main program) that inputs two numbers from the user, calls the Average() function of Exercise 41 to find the mean of these numbers, and displays the result.

Exercises 43–46 refer to the following program:

Main
   Declare K As Integer
   Input K
   Set Result = F(K)
   Write Result
End Program
Function F(N) As Integer
   If N == 1 Then
      Set F = 1
   Else
      Set F = N * F(N-1)
      Set N = N -1
   End If
End Function

43. What is the output of this program if \(K = 1\)?
44. If \(K = 3\), how many times is the function \(F()\) called?
45. What is the output of this program if \(K = 3\)?
46. Write a non-recursive function (using a For loop) that has the same effect as \(F()\).
47. Write a recursive function \(\text{Mult}(M,N)\) that multiplies the positive integers \(M\) and \(N\) by using the fact that
   \[ M \times N = M + M + \ldots + M \text{ (}N\text{ times)} \]
48. Write a non-recursive function (using a loop) that has the same effect as the recursive function \(\text{Mult}(M,N)\) of Exercise 47.

Programming Problems
For each of the following Programming Problems, use pseudocode to design a suitable program to solve it. In your program, make use of subprograms with parameters and arguments.

1. Input a list of positive numbers (terminated by 0) into an array, find the largest number in the array, and output the result. Use a subprogram to input the numbers, a function to find the largest number, and a subprogram to output the result.
2. Input a list of positive numbers (terminated by 0) into an array, find the mean (average) of the numbers in the array, and output the result. Use a subprogram to input the numbers, a function to find the mean, and a subprogram to output the result.

3. Develop a menu-driven program that inputs a number \( X \), and at the user's option, finds and displays the area \( (A) \) of one of the following:
   - A square with side \( X \), \( A = X^2 \)
   - A circle with radius \( X \), \( A = 3.14 \times X^2 \)
   - An equilateral triangle with side \( X \), \( A = \sqrt{3}/4 \times X^2 \)

4. The factorial of a positive integer \( N \), denoted by \( N! \), is defined by the following:
   \[
   N! = 1 \times 2 \times \ldots \times N \quad (\text{Note: } 0! = 1)
   \]
   Using subprograms and functions, create a recursive program to compute \( N! \). The user should input a positive integer and a subprogram should check that the input is correct (a positive integer). Then use recursion to compute the factorial. Create a subprogram that will call itself to do the multiplication until \( N = 1 \). Then display the result in the main program.

5. Compute and display the income tax due in the state of East Euphoria on a taxable income entered by the user, according to the following table:

<table>
<thead>
<tr>
<th>Taxable Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>From $0</td>
<td>To $50,000</td>
</tr>
<tr>
<td>$50,000</td>
<td>To $100,000</td>
</tr>
<tr>
<td>$100,000</td>
<td>...</td>
</tr>
</tbody>
</table>