CHAPTER 3

Modules

TOPICS

3.1 Introduction to Modules

CONCEPT: **A module is a group of statements that exist for the purpose of performing a specific task within a program.**

In Chapter 1 you learned that a program is a set of instructions that a computer follows to perform a task. Then, in Chapter 2 you saw a simple program that performs the task of calculating an employee's pay. Recall that the program multiplied the number of hours that the employee worked by the employee's hourly pay rate. A more realistic payroll program, however, would do much more than this. In a real-world application, the overall task of calculating an employee's pay would consist of several subtasks, such as the following:

- Getting the employee's hourly pay rate
- Getting the number of hours worked
- Calculating the employee's gross pay
- Calculating overtime pay
- Calculating withholdings for taxes and benefits
- Calculating the net pay
- Printing the paycheck

Most programs perform tasks that are large enough to be broken down into several subtasks. For this reason, programmers usually break down their programs into modules. A *module* is a group of statements that exist within a program for the purpose of performing a specific task. Instead of writing a large program as one long sequence of statements, it can

be written as several small modules, each one performing a specific part of the task. These small modules can then be executed in the desired order to perform the overall task.

This approach is sometimes called *divide and conquer* because a large task is divided into several smaller tasks that are easily performed. Figure 3-1 illustrates this idea by comparing two programs: one that uses a long, complex sequence of statements to perform a task, and another that divides a task into smaller tasks, each of which is performed by a separate module.

When using modules in a program, you generally isolate each task within the program in its own module. For example, a realistic pay calculating program might have the following modules:

- A module that gets the employee's hourly pay rate
- A module that gets the number of hours worked
- A module that calculates the employee's gross pay
- A module that calculates the overtime pay
- A module that calculates the withholdings for taxes and benefits
- A module that calculates the net pay
- A module that prints the paycheck

Although every modern programming language allows you to create modules, they are not always referred to as modules. Modules are commonly called *procedures, subroutines, subprograms, methods,* and *functions*. (A function is a special type of module that we will discuss in Chapter 6.)

Benefits of Using Modules

A program benefits in the following ways when it is modularized:

Simpler Code

A program's code tends to be simpler and easier to understand when it is modularized. Several small modules are much easier to read than one long sequence of statements.

Code Reuse

Modules also reduce the duplication of code within a program. If a specific operation is performed in several places in a program, a module can be written once to perform that operation, and then be executed any time it is needed. This benefit of using modules is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

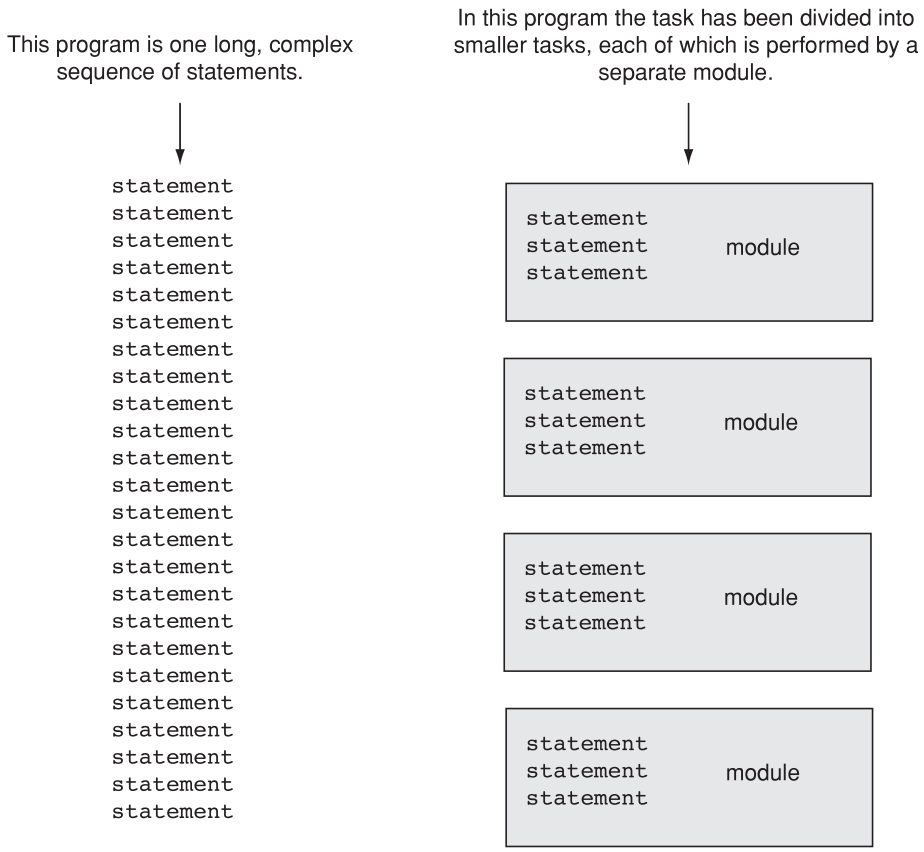
Better Testing

When each task within a program is contained in its own module, testing and debugging become simpler. Programmers can test each module in a program individually, to determine whether it correctly performs its operation. This makes it easier to isolate and fix errors.

Easier Maintenance

Most programs must be periodically modified to correct logic errors, improve performance, and provide a better experience for the user. Such modifications are known as *maintenance*. A modularized program is easier to maintain than an unmodularized program because its code tends to be simpler, smaller, and easier to understand.

Figure 3-1 Using modules to divide and conquer a large task



Faster Development

Suppose a programmer or a team of programmers is developing multiple programs. They discover that each of the programs performs several common tasks, such as asking for a username and a password, displaying the current time, and so on. It doesn't make sense to write the code for these tasks multiple times. Instead, modules can be written for the commonly needed tasks, and those modules can be incorporated into each program that needs them.

Easier Facilitation of Teamwork

Modules also make it easier for programmers to work in teams. When a program is developed as a set of modules that each perform an individual task, then different programmers can be assigned the job of writing different modules.

Checkpoint

3.1 What is a module?

3.2 What is meant by the phrase "divide and conquer"?

3.3 How do modules help you reuse code in a program?

3.4 How can modules make the development of multiple programs faster?

3.5 How can modules make it easier for programs to be developed by teams of programmers?

3.2 Defining and Calling a Module

CONCEPT: **The code for a module is known as a module definition. To execute the module, you write a statement that calls it.**

Module Names

Before we discuss the process of creating and using modules, we should mention a few things about module names. Just as you name the variables that you use in a program, you also name the modules. A module's name should be descriptive enough so that anyone reading your code can reasonably guess what the module does.

Because modules perform actions, most programmers prefer to use verbs in module names. For example, a module that calculates gross pay might be named `calculateGrossPay`. This name would make it evident to anyone reading the code that the module calculates something. What does it calculate? The gross pay, of course. Other examples of good module names would be `getHours`, `getPayRate`, `calculateOvertime`, `printCheck`, and so on. Each module name describes what the module does.

When naming a module, most languages require that you follow the same rules that you follow when naming variables. This means that module names cannot contain spaces, cannot typically contain punctuation characters, and usually cannot begin with a number. These are only general rules, however. The specific rules for naming a module will vary slightly with each programming language. (Recall that we discussed the common variable naming rules in Chapter 2.)


VideoNote
Defining and Calling a Module

Defining and Calling a Module

To create a module you write its *definition*. In most languages, a module definition has two parts: a header and a body. The *header* indicates the starting point of the module, and the *body* is a list of statements that belong to the module. Here is the general format that we will follow when we write a module definition in pseudocode:

```
Module name()
   statement
   statement
   etc.
End Module
```

These statements are the body of the module.

The first line is the module header. In our pseudocode the header begins with the word `Module`, followed by the name of the module, followed by a set of parentheses. It is a common practice in most programming languages to put a set of parentheses after a

module name. Later in this chapter, you will see the actual purpose of the parentheses, but for now, just remember that they come after the module name.

Beginning at the line after the module header, one or more statements will appear. These statements are the module's body, and are performed any time the module is executed. The last line of the definition, after the body, reads `End Module`. This line marks the end of the module definition.

Let's look at an example. Keep in mind that this is not a complete program. We will show the entire pseudocode program in a moment.

```
Module showMessage()
  Display "Hello world."
End Module
```

This pseudocode defines a module named `showMessage`. As its name implies, the purpose of this module is to show a message on the screen. The body of the `showMessage` module contains one statement: a `Display` statement that displays the message "Hello world."

Notice in the previous example that the statement in the body of the module is indented. Indenting the statements in the body of a module is not usually required,[1] but it makes your code much easier to read. By indenting the statements inside a module, you visually set them apart. As a result, you can tell at a glance which statements are inside the module. This practice is a programming style convention that virtually all programmers follow.

Calling a Module

A module definition specifies what a module does, but it does not cause the module to execute. To execute a module, we must *call* it. In pseudocode we will use the word `Call` to call a module. This is how we would call the `showMessage` module:

```
Call showMessage()
```

When a module is called, the computer jumps to that module and executes the statements in the module's body. Then, when the end of the module is reached, the computer jumps back to the part of the program that called the module, and the program resumes execution at that point.

To fully demonstrate how module calling works, we will look at Program 3-1.

Program 3-1

```
1  Module main()
2     Display "I have a message for you."
3     Call showMessage()
4     Display "That's all, folks!"
5  End Module
6
```

[1] The Python language requires you to indent the statements inside a module.

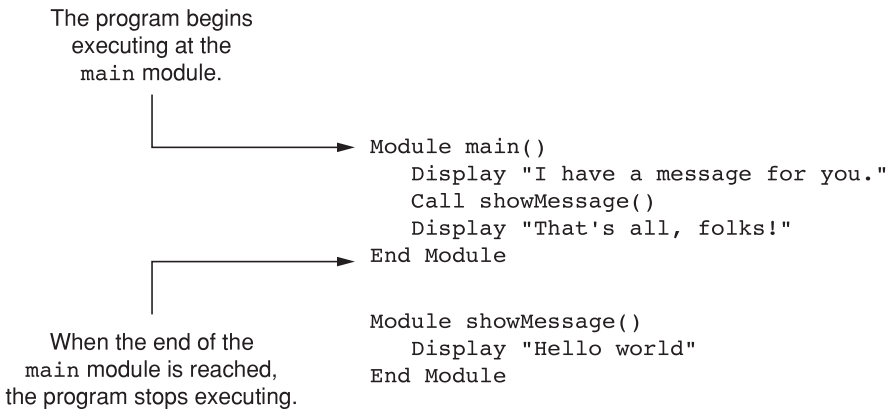
```
7  Module showMessage()
8     Display "Hello world"
9  End Module
```

Program Output

```
I have a message for you.
Hello world
That's all, folks!
```

First, notice that Program 3-1 has two modules: a module named `main` appears in lines 1 through 5, and the `showMessage` module appears in lines 7 through 9. Many programming languages require that programs have a *main module*. The main module is the program's starting point, and it generally calls other modules. When the end of the main module is reached, the program stops executing. In this book, any time you see a pseudocode program with a module named `main`, we are using that module as the program's starting point. Likewise, when the end of the `main` module is reached, the program will stop executing. This is shown in Figure 3-2.

Figure 3-2 The `main` module



NOTE: Many languages, including Java, C, and C++, require that the main module actually be named `main`, as we have shown in Program 3-1.

Let's step through the program. When the program runs, the `main` module starts and the statement in line 2 displays "I have a message for you." Then, line 3 calls the `showMessage` module. As shown in Figure 3-3, the computer jumps to the `showMessage` module and executes the statements in its body. There is only one statement in the body of the `showMessage` module: the `Display` statement in line 8. This statement displays "Hello world" and then the module ends. As shown in Figure 3-4, the computer jumps back to the part of the program that called

showMessage, and resumes execution from that point. In this case, the program resumes execution at line 4, which displays "That's all, folks!" The main module ends at line 5, so the program stops executing.

Figure 3-3 Calling the showMessage module

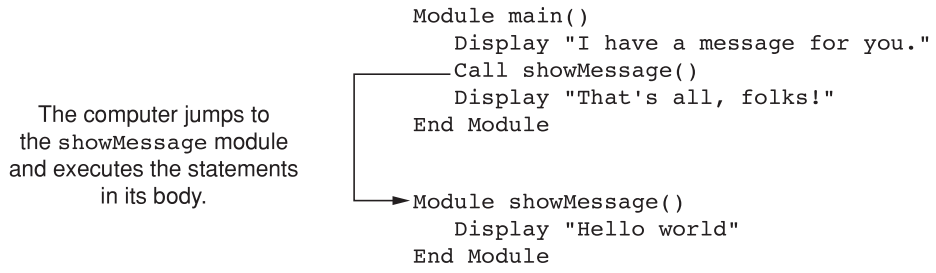
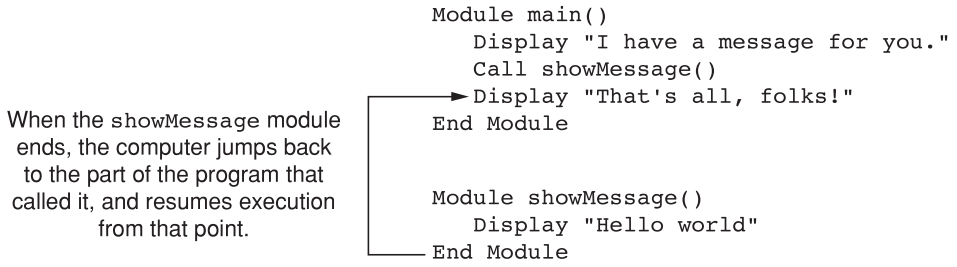


Figure 3-4 The showMessage module returns



When the computer encounters a module call, such as the one in line 3 of Program 3-1, it has to perform some operations "behind the scenes" so it will know where to return after the module ends. First, the computer saves the memory address of the location that it should return to. This is typically the statement that appears immediately after the module call. This memory location is known as the *return point*. Then, the computer jumps to the module and executes the statements in its body. When the module ends, the computer jumps back to the return point and resumes execution.

NOTE: When a program calls a module, programmers commonly say that the *control* of the program transfers to that module. This simply means that the module takes control of the program's execution.

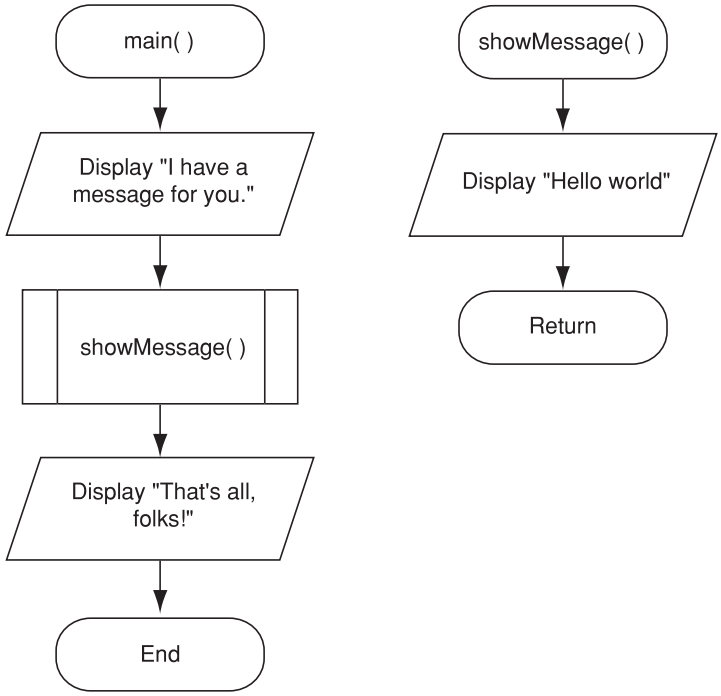
Flowcharting a Program with Modules

In a flowchart, a module call is shown with a rectangle that has vertical bars at each side, as shown in Figure 3-5. The name of the module that is being called is written on the symbol. The example shown in Figure 3-5 shows how we would represent a call to the showMessage module.

Figure 3-5 Module call symbol

Programmers typically draw a separate flowchart for each module in a program. For example, Figure 3-6 shows how Program 3-1 would be flowcharted. Notice that the figure shows two flowcharts: one for the `main` module and another for the `showMessage` module.

Figure 3-6 Flowchart for Program 3-1



When drawing a flowchart for a module, the starting terminal symbol usually shows the name of the module. The ending terminal symbol in the `main` module reads `End` because it marks the end of the program's execution. The ending terminal symbol for all other modules reads `Return` because it marks the point where the computer returns to the part of the program that called the module.

Top-Down Design

In this section, we have discussed and demonstrated how modules work. You've seen how the computer jumps to a module when it is called, and returns to the part of the program that called the module when the module ends. It is important that you understand these mechanical aspects of modules.

Just as important as understanding how modules work is understanding how to design a modularized program. Programmers commonly use a technique known as *top-down design* to break down an algorithm into modules. The process of top-down design is performed in the following manner:

- The overall task that the program is to perform is broken down into a series of subtasks.
- Each of the subtasks is examined to determine whether it can be further broken down into more subtasks. This step is repeated until no more subtasks can be identified.
- Once all of the subtasks have been identified, they are written in code.

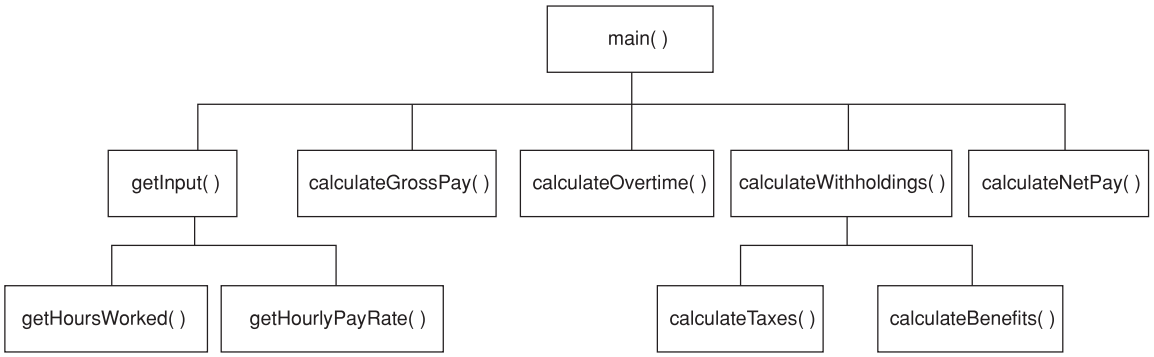
This process is called top-down design because the programmer begins by looking at the topmost level of tasks that must be performed, and then breaks down those tasks into lower levels of subtasks.

NOTE: The top-down design process is sometimes called *stepwise refinement*.

Hierarchy Charts

Flowcharts are good tools for graphically depicting the flow of logic inside a module, but they do not give a visual representation of the relationships between modules. Programmers commonly use *hierarchy charts* for this purpose. A hierarchy chart, which is also known as a *structure chart*, shows boxes that represent each module in a program. The boxes are connected in a way that illustrates their relationship to one another. Figure 3-7 shows an example of a hierarchy chart for a pay calculating program.

Figure 3-7 A hierarchy chart



The chart shown in Figure 3-7 shows the `main` module as the topmost module in the hierarchy. The main module calls five other modules: `getInput`, `calculateGrossPay`, `calculateOvertime`, `calculateWithholdings`, and `calculateNetPay`. The `getInput` module calls two additional modules: `getHoursWorked` and `getHourlyPayRate`. The `calculateWithholdings` module also calls two modules: `calculateTaxes` and `calculateBenefits`.

Notice that the hierarchy chart does not show the steps that are taken inside a module. Because hierarchy charts do not reveal any details about how modules work, they do not replace flowcharts or pseudocode.

In the Spotlight:

Defining and Calling Modules

Professional Appliance Service, Inc. offers maintenance and repair services for household appliances. The owner wants to give each of the company's service technicians a small handheld computer that displays step-by-step instructions for many of the repairs that they perform. To see how this might work, the owner has asked you to develop a program that displays the following instructions for disassembling an ACME laundry dryer:

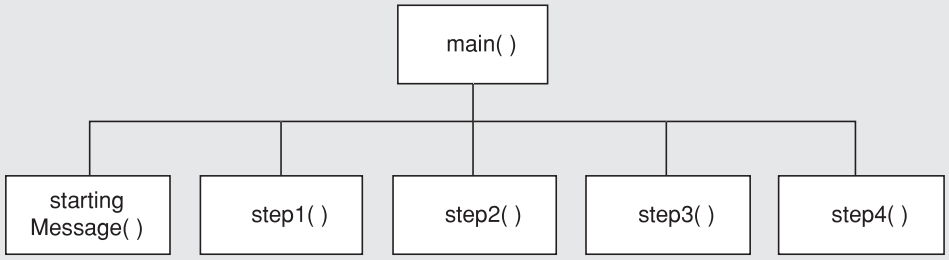
Step 1. Unplug the dryer and move it away from the wall.
Step 2. Remove the six screws from the back of the dryer.
Step 3. Remove the dryer's back panel.
Step 4. Pull the top of the dryer straight up.

During your interview with the owner, you determine that that the program should display the steps one at a time. You decide that after each step is displayed, the user will be asked to press a key to see the next step. Here is the algorithm for the program:

1. Display a starting message, explaining what the program does.
2. Ask the user to press a key to see Step 1.
3. Display the instructions for Step 1.
4. Ask the user to press a key to see the next step.
5. Display the instructions for Step 2.
6. Ask the user to press a key to see the next step.
7. Display the instructions for Step 3.
8. Ask the user to press a key to see the next step.
9. Display the instructions for Step 4.

This algorithm lists the top level of tasks that the program needs to perform, and becomes the basis of the program's `main` module. Figure 3-8 shows the program's structure in a hierarchy chart.

Figure 3-8 Hierarchy chart for the program



As you can see from the hierarchy chart, the `main` module will call several other modules. Here are summaries of those modules:

- `startingMessage`—This module will display the starting message that tells the technician what the program does.
- `step1`—This module will display the instructions for Step 1.
- `step2`—This module will display the instructions for Step 2.
- `step3`—This module will display the instructions for Step 3.
- `step4`—This module will display the instructions for Step 4.

Between calls to these modules, the `main` module will instruct the user to press a key to see the next step in the instructions. Program 3-2 shows the pseudocode for the program. Figure 3-9 shows the flowchart for the `main` module, and Figure 3-10 shows the flowcharts for the `startingMessage`, `step1`, `step2`, `step3`, and `step4` modules.

Program 3-2

```
 1  Module main()
 2     // Display the starting message.
 3     Call startingMessage()
 4     Display "Press a key to see Step 1."
 5     Input
 6
 7     // Display Step 1.
 8     Call step1()
 9     Display "Press a key to see Step 2."
10     Input
11
12     // Display Step 2.
13     Call step2()
14     Display "Press a key to see Step 3."
15     Input
16
17     // Display Step 3.
18     Call step3()
19     Display "Press a key to see Step 4."
20     Input
21
22     // Display Step 4.
23     Call step4()
24  End Module
25
26  // The startingMessage module displays
27  // the program's starting message.
28  Module startingMessage()
29     Display "This program tells you how to"
30     Display "disassemble an ACME laundry dryer."
31     Display "There are 4 steps in the process."
32  End Module
33
34  // The step1 module displays the instructions
35  // for Step 1.
```

```
36  Module step1()
37     Display "Step 1: Unplug the dryer and"
38     Display "move it away from the wall."
39  End Module
40
41  // The step2 module displays the instructions
42  // for Step 2.
43  Module step2()
44     Display "Step 2: Remove the six screws"
45     Display "from the back of the dryer."
46  End Module
47
48  // The step3 module displays the instructions
49  // for Step 3.
50  Module step3()
51     Display "Step 3: Remove the dryer's"
52     Display "back panel."
53  End Module
54
55  // The step4 module displays the instructions
56  // for Step 4.
57  Module step4()
58     Display "Step 4: Pull the top of the"
59     Display "dryer straight up."
60  End Module
```

Program Output (with Input Shown in Bold)

```
This program tells you how to
disassemble an ACME laundry dryer.
There are 4 steps in the process.
Press a key to see Step 1.
[Enter]
Step 1: Unplug the dryer and
move it away from the wall.
Press a key to see Step 2.
[Enter]
Step 2: Remove the six screws
from the back of the dryer.
Press a key to see Step 3.
[Enter]
Step 3: Remove the dryer's
back panel.
Press a key to see Step 4.
[Enter]
Step 4: Pull the top of the
dryer straight up.
```

NOTE: Lines 5, 10, 15, and 20 show an `Input` statement with no variable specified. In our pseudocode, this is the way we will read a keystroke from the keyboard without saving the character that was pressed. Most programming languages provide a way to do this.

Figure 3-9 Flowchart for the `main` module in Program 3-2

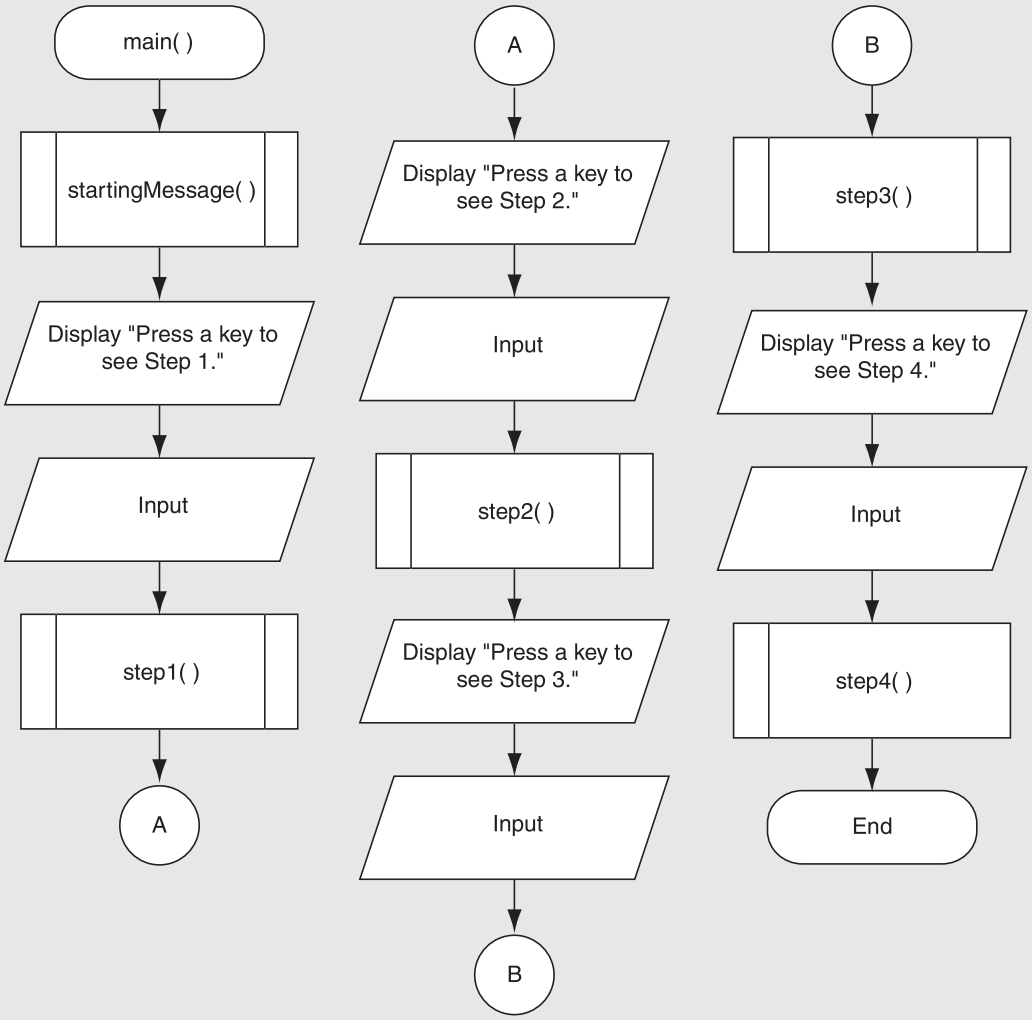
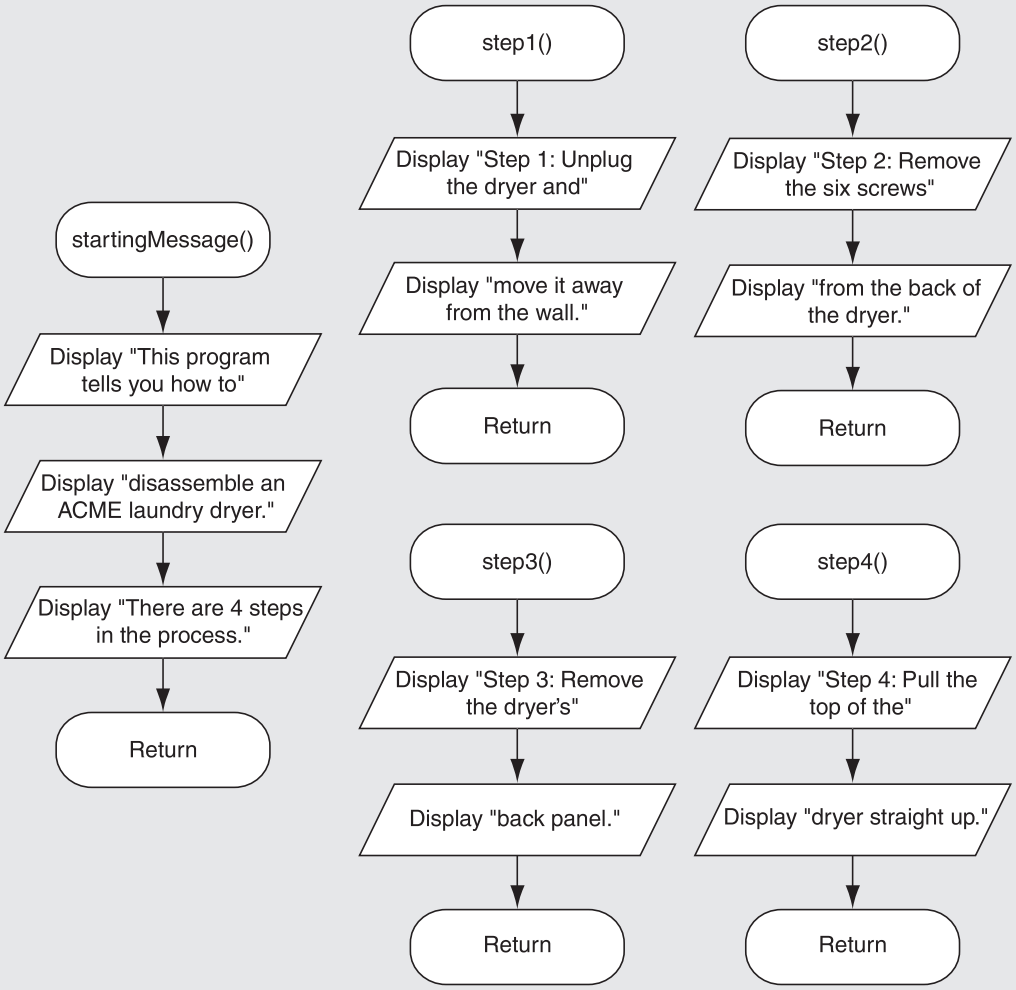


Figure 3-10 Flowcharts for the other modules in Program 3-2



Checkpoint

3.6 In most languages, a module definition has what two parts?

3.7 What does the phrase "calling a module" mean?

3.8 When a module is executing, what happens when the end of the module is reached?

3.9 Describe the steps involved in the top-down design process.

3.3 Local Variables

CONCEPT: **A local variable is declared inside a module and cannot be accessed by statements that are outside the module. Different modules can have local variables with the same names because the modules cannot see each other's local variables.**

In most programming languages, a variable that is declared inside a module is called a *local variable*. A local variable belongs to the module in which it is declared, and only statements inside that module can access the variable. (The term *local* is meant to indicate that the variable can be used only locally, within the module in which it is declared.)

An error will occur if a statement in one module tries to access a local variable that belongs to another module. For example, look at the pseudocode in Program 3-3.

Program 3-3

```
1  Module main()
2     Call getName()
3     Display "Hello ", name      <-- This will cause an error!
4  End Module
5
6  Module getName()
7     Declare String name         <-- This is local variable.
8     Display "Enter your name."
9     Input name
10 End Module
```

The `name` variable is declared in line 7, inside the `getName` module. Because it is declared inside the `getName` module, it is a local variable belonging to that module. Line 8 prompts the user to enter his or her name, and the `Input` statement in line 9 stores the user's input in the `name` variable.

The `main` module calls the `getName` module in line 2. Then, the `Display` statement in line 3 tries to access the `name` variable. This results in an error because the `name` variable is local to the `getName` module, and statements in the `main` module cannot access it.

Scope and Local Variables

Programmers commonly use the term *scope* to describe the part of a program in which a variable may be accessed. A variable is visible only to statements inside the variable's scope.

A local variable's scope usually begins at the variable's declaration and ends at the end of the module in which the variable is declared. The variable cannot be accessed by statements that are outside this region. This means that a local variable cannot be accessed by code that is outside the module, or inside the module but before the

variable's declaration. For example, look at the following code. It has an error because the `Input` statement tries to store a value in the `name` variable, but the statement is outside the variable's scope. Moving the variable declaration to a line before the `Input` statement will fix this error.

```
Module getName()
   Display "Enter your name."
   Input name                    <-- This statement will cause an error because
   Declare String name               the name variable has not been declared yet.
End Module
```

Duplicate Variable Names

In most programming languages, you cannot have two variables with the same name in the same scope. For example, look at the following module:

```
Module getTwoAges()
   Declare Integer age
   Display "Enter your age."
   Input age
   Declare Integer age           <-- This will cause an error!
   Display "Enter your pet's age."   A variable named age has already
   Input age                         been declared.
End Module
```

This module declares two local variables named `age`. The second variable declaration will cause an error because a variable named `age` has already been declared in the module. Renaming one of the variables will fix this error.

TIP: You cannot have two variables with the same name in the same module because the compiler or interpreter would not know which variable to use when a statement tries to access one of them. All variables that exist within the same scope must have unique names.

Although you cannot have two local variables with the same name in the same module, it is usually okay for a local variable in one module to have the same name as a local variable in a different module. For example, look at Program 3-4.

Program 3-4

```
1  Module main()
2     Call showSquare()
3     Call showHalf()
4  End Module
5
6  Module showSquare()
7     Declare Real number
8     Declare Real square
9
```

```
10     Display "Enter a number."
11     Input number
12     Set square = number^2
13     Display "The square of that number is ", square
14  End Module
15
16  Module showHalf()
17     Declare Real number
18     Declare Real half
19
20     Display "Enter a number."
21     Input number
22     Set half = number / 2
23     Display "Half of that number is ", half
24  End Module
```

Program Output (with Input Shown in Bold)

```
Enter a number.
5 [Enter]
The square of that number is 25
Enter a number.
20 [Enter]
Half of that number is 10
```

Program 3-4 has three modules: `main`, `showSquare`, and `showHalf`. Notice that the `showSquare` module has a local variable named `number` (declared in line 7) and the `showHalf` module also has a local variable named `number` (declared in line 17). It is legal for the two modules to have a local variable with the same name because the variables are not in the same scope. The scope of the two `number` variables is shown in Figure 3-11.

Figure 3-11 Scope of the two number variables

```
Module main()
   Call showSquare()
   Call showHalf()
End Module

Module showSquare()
   Declare Real number
   Declare Real square

   Display "Enter a number."
   Input number
   Set square = number^2
   Display "The square of that number is ", square
End Module

Module showHalf()
   Declare Real number
   Declare Real half

   Display "Enter a number."
   Input number
   Set half = number / 2
   Display "Half of that number is ", half
End Module
```

Scope of the `number` variable in the `showSquare` module.

Scope of the `number` variable in the `showHalf` module.

Checkpoint

3.10 What is a local variable? How is access to a local variable restricted?

3.11 What is a variable's scope?

3.12 Is it usually permissible to have more than one variable with the same name in the same scope? Why or why not?

3.13 Is it usually permissible for a local variable in one module to have the same name as a local variable in a different module?

3.4 Passing Arguments to Modules

CONCEPT: **An argument is any piece of data that is passed into a module when the module is called. A parameter is a variable that receives an argument that is passed into a module.**

VideoNote
Passing Arguments to a Module

Sometimes it is useful not only to call a module, but also to send one or more pieces of data into the module. Pieces of data that are sent into a module are known as *arguments*. The module can use its arguments in calculations or other operations.

If you want a module to receive arguments when it is called, you must equip the module with one or more parameter variables. A *parameter variable,* often simply called a *parameter,* is a special variable that receives an argument when a module is called. Here is an example of a pseudocode module that has a parameter variable:

```
Module doubleNumber(Integer value)
   Declare Integer result
   Set result = value * 2
   Display result
End Module
```

This module's name is `doubleNumber`. Its purpose is to accept an integer number as an argument and display the value of that number doubled. Look at the module header and notice the words `Integer value` that appear inside the parentheses. This is the declaration of a parameter variable. The parameter variable's name is `value` and its data type is `Integer`. The purpose of this variable is to receive an `Integer` argument when the module is called. Program 3-5 demonstrates the module in a complete program.

Program 3-5

```
1  Module main()
2     Call doubleNumber(4)
3  End Module
4
```

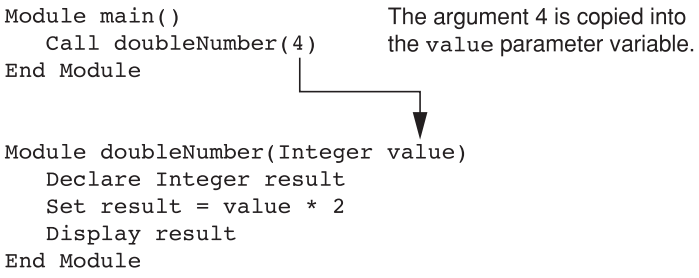
```
5  Module doubleNumber(Integer value)
6     Declare Integer result
7     Set result = value * 2
8     Display result
9  End Module
```

Program Output

```
8
```

When this program runs, the `main` module will begin executing. The statement in line 2 calls the `doubleNumber` module. Notice that the number 4 appears inside the parentheses. This is an argument that is being passed to the `doubleNumber` module. When this statement executes, the `doubleNumber` module will be called with the number 4 copied into the `value` parameter variable. This is shown in Figure 3-12.

Figure 3-12 The argument 4 is copied into the `value` parameter variable



Let's step through the `doubleNumber` module. As we do, remember that the `value` parameter variable will contain the number that was passed into it as an argument. In this program, that number is 4.

Line 6 declares a local `Integer` variable named `result`. Then, line 7 assigns the value of the expression `value * 2` to `result`. Because the `value` variable contains 4, this line assigns 8 to `result`. Line 8 displays the contents of the `result` variable. The module ends at line 9.

For example, if we had called the module as follows:

```
Call doubleNumber(5)
```

the module would have displayed 10.

We can also pass the contents of a variable as an argument. For example, look at Program 3-6. The `main` module declares an `Integer` variable named `number` in line 2. Lines 3 and 4 prompt the user to enter a number, and line 5 reads the user's input into the `number` variable. Notice that in line 6 `number` is passed as an argument to the

doubleNumber module, which causes the number variable's contents to be copied into the value parameter variable. This is shown in Figure 3-13.

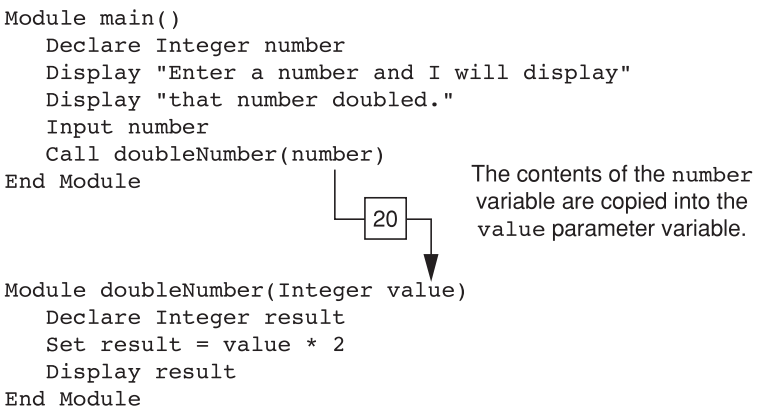
Program 3-6

```
1  Module main()
2     Declare Integer number
3     Display "Enter a number and I will display"
4     Display "that number doubled."
5     Input number
6     Call doubleNumber(number)
7  End Module
8
9  Module doubleNumber(Integer value)
10    Declare Integer result
11    Set result = value * 2
12    Display result
13 End Module
```

Program Output (with Input Shown in Bold)

```
Enter a number and I will display
that number doubled.
20 [Enter]
40
```

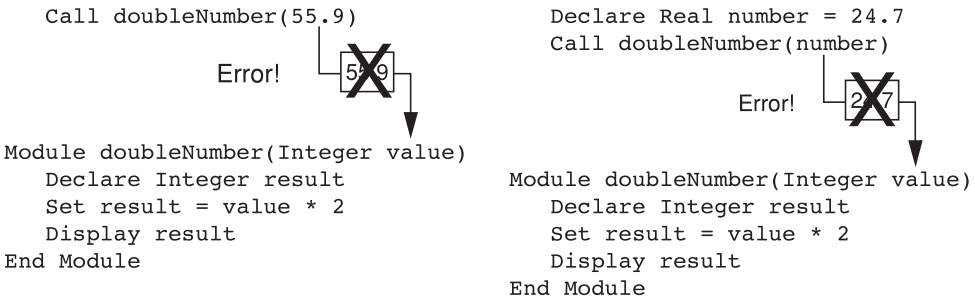
Figure 3-13 The contents of the number variable passed as an argument



Argument and Parameter Compatibility

When you pass an argument to a module, most programming languages require that the argument and the receiving parameter variable be of the same data type. If you try to pass an argument of one type into a parameter variable of another type, an error usually occurs. For example, Figure 3-14 shows that you cannot pass a real number or a Real variable into an Integer parameter.

Figure 3-14 Arguments and parameter variables must be of the same type



NOTE: Some languages allow you to pass an argument into a parameter variable of a different type as long as no data will be lost. For example, some languages allow you to pass integer arguments into real parameters because real variables can hold whole numbers. If you pass a real argument, such as 24.7, into an integer parameter, the fractional part of the number would be lost.

Parameter Variable Scope

Earlier in this chapter, you learned that a variable's scope is the part of the program in which the variable may be accessed. A variable is visible only to statements inside the variable's scope. A parameter variable's scope is usually the entire module in which the parameter is declared. No statement outside the module can access the parameter variable.

Passing Multiple Arguments

Most languages allow you to write modules that accept multiple arguments. Program 3-7 shows a pseudocode module named `showSum` that accepts two `Integer` arguments. The module adds the two arguments and displays their sum.

Program 3-7

```
1  Module main()
2     Display "The sum of 12 and 45 is:"
3     Call showSum(12, 45)
4  End Module
5
6  Module showSum(Integer num1, Integer num2)
7     Declare Integer result
8     Set result = num1 + num2
9     Display result
10 End Module
```

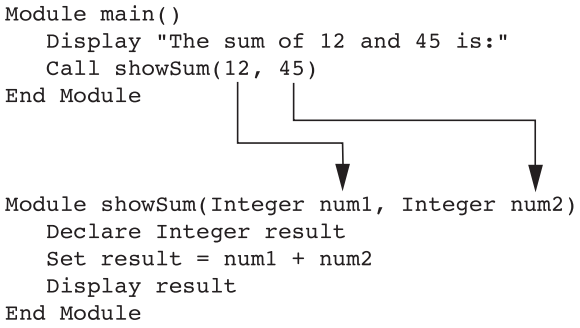
Program Output

```
The sum of 12 and 45 is:
57
```

Notice that two parameter variables, `num1` and `num2`, are declared inside the parentheses in the module header. This is often referred to as a *parameter list*. Also notice that a comma separates the declarations.

The statement in line 3 calls the `showSum` module and passes two arguments: 12 and 45. The arguments are passed into the parameter variables in the order that they appear in the module call. In other words, the first argument is passed into the first parameter variable, and the second argument is passed into the second parameter variable. So, this statement causes 12 to be passed into the `num1` parameter and 45 to be passed into the `num2` parameter, as shown in Figure 3-15.

Figure 3-15 Two arguments passed into two parameters



Suppose we were to reverse the order in which the arguments are listed in the module call, as shown here:

```
Call showSum(45, 12)
```

This would cause 45 to be passed into the `num1` parameter and 12 to be passed into the `num2` parameter. The following pseudocode code shows one more example. This time we are passing variables as arguments.

```
Declare Integer value1 = 2
Declare Integer value2 = 3
Call showSum(value1, value2)
```

When the `showSum` method executes as a result of this code, the `num1` parameter will contain 2 and the `num2` parameter will contain 3.

In the Spotlight:

Passing an Argument to a Module

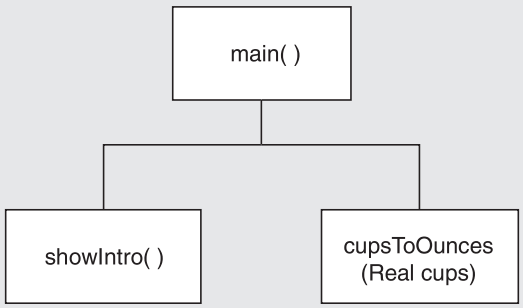
Your friend Michael runs a catering company. Some of the ingredients that his recipes require are measured in cups. When he goes to the grocery store to buy those ingredients, however, they are sold only by the fluid ounce. He has asked you to write a simple program that converts cups to fluid ounces.

You design the following algorithm:

1. Display an introductory screen that explains what the program does.
2. Get the number of cups.
3. Convert the number of cups to fluid ounces and display the result.

This algorithm lists the top level of tasks that the program needs to perform, and becomes the basis of the program's `main` module. Figure 3-16 shows the program's structure in a hierarchy chart.

Figure 3-16 Hierarchy chart for the program



As shown in the hierarchy chart, the `main` module will call two other modules.

Here are summaries of those modules:

- `showIntro`—This module will display a message on the screen that explains what the program does.
- `cupsToOunces`—This module will accept the number of cups as an argument and calculate and display the equivalent number of fluid ounces.

In addition to calling these modules, the `main` module will ask the user to enter the number of cups. This value will be passed to the `cupsToOunces` module. Program 3-8 shows the pseudocode for the program, and Figure 3-17 shows a flowchart.

Program 3-8

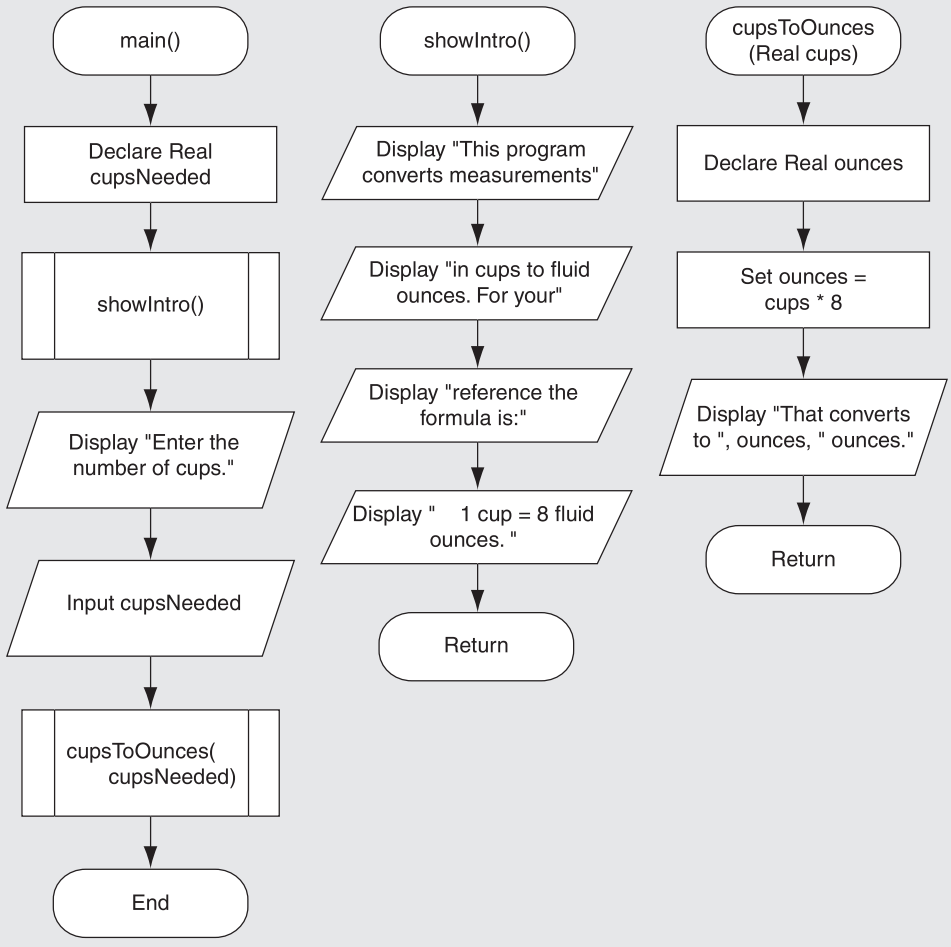
```
1  Module main()
2     // Declare a variable for the
3     // number of cups needed.
4     Declare Real cupsNeeded
5
6     // Display an intro message.
7     Call showIntro()
8
9     // Get the number of cups.
10    Display "Enter the number of cups."
11    Input cupsNeeded
12
13    // Convert cups to ounces.
14    Call cupsToOunces(cupsNeeded)
15 End Module
16
17 // The showIntro module displays an
18 // introductory screen.
19 Module showIntro()
20    Display "This program converts measurements"
21    Display "in cups to fluid ounces. For your"
22    Display "reference the formula is:"
23    Display "    1 cup = 8 fluid ounces."
24 End Module
25
26 // The cupsToOunces module accepts a number
27 // of cups and displays the equivalent number
```

```
// of ounces.
Module cupsToOunces(Real cups)
   // Declare variables.
   Declare Real ounces

   // Convert cups to ounces.
   Set ounces = cups * 8

   // Display the result.
   Display "That converts to ",
           ounces, " ounces."
End Module
```

Program Output (with Input Shown in Bold)

```
This program converts measurements
in cups to fluid ounces. For your
reference the formula is:
    1 cup = 8 fluid ounces.
Enter the number of cups.
2 [Enter]
That converts to 16 ounces.
```

Figure 3-17 Flowchart for Program 3-8



Passing Arguments by Value and by Reference

Many programming languages provide two different ways to pass arguments: by value and by reference. Before studying these techniques in detail, we should mention that different languages have their own way of doing each. In this book, we will teach you the fundamental concepts behind these techniques, and show you how to model them in pseudocode. When you begin to use these techniques in an actual language, you will need to learn the details of how they are carried out in that language.

Passing Arguments by Value

All of the example programs that we have looked at so far pass arguments by value. Arguments and parameter variables are separate items in memory. Passing an argument *by value* means that only a copy of the argument's value is passed into the parameter variable. If the contents of the parameter variable are changed inside the module, it has no effect on the argument in the calling part of the program. For example, look at Program 3-9.

Program 3-9

```
1  Module main()
2     Declare Integer number = 99
3
4     // Display the value stored in number.
5     Display "The number is ", number
6
7     // Call the changeMe module, passing
8     // the number variable as an argument.
9     Call changeMe(number)
10
11    // Display the value of number again.
12    Display "The number is ", number
13 End Module
14
15 Module changeMe(Integer myValue)
16    Display "I am changing the value."
17
18    // Set the myValue parameter variable
19    // to 0.
20    Set myValue = 0
21
22    // Display the value in myValue.
23    Display "Now the number is ", myValue
24 End Module
```

Program Output

```
The number is 99
I am changing the value.
Now the number is 0
The number is 99
```

The `main` module declares a local variable named `number` in line 2, and initializes it to the value 99. As a result, the `Display` statement in line 5 displays "The number is 99." The `number` variable's value is then passed as an argument to the `changeMe` module in

line 9. This means that in the `changeMe` module the value 99 will be copied into the `myValue` parameter variable.

Inside the `changeMe` module, in line 20, the `myValue` parameter variable is set to 0. As a result, the `Display` statement in line 23 displays "Now the number is 0." The module ends, and control of the program returns to the `main` module.

The next statement to execute is the `Display` statement in line 12. This statement displays "The number is 99." Even though the `myValue` parameter variable was changed in the `changeMe` method, the argument (the `number` variable in `main`) was not modified.

Passing an argument is a way that one module can communicate with another module. When the argument is passed by value, the communication channel works in only one direction: the calling module can communicate with the called module. The called module, however, cannot use the argument to communicate with the calling module.

Passing Arguments by Reference

Passing an argument *by reference* means that the argument is passed into a special type of parameter known as a *reference variable*. When a reference variable is used as a parameter in a module, it allows the module to modify the argument in the calling part of the program.

A reference variable acts as an alias for the variable that was passed into it as an argument. It is called a reference variable because it references the other variable. Anything that you do to the reference variable is actually done to the variable it references.

Reference variables are useful for establishing two-way communication between modules. When a module calls another module and passes a variable by reference, communication between the modules can take place in the following ways:

- The calling module can communicate with the called module by passing an argument.
- The called module can communicate with the calling module by modifying the value of the argument via the reference variable.

In pseudocode we will declare that a parameter is a reference variable by writing the word `Ref` before the parameter variable's name in the module header. For example, look at the following pseudocode module:

```
Module setToZero(Integer Ref value)
   Set value = 0
End Module
```

The word `Ref` indicates that `value` is a reference variable. The module stores 0 in the `value` parameter. Because `value` is a reference variable, this action is actually performed on the variable that was passed to the module as an argument. Program 3-10 demonstrates this module.

Program 3-10

```
1  Module main()
2     // Declare and initialize some variables.
3     Declare Integer x = 99
```

```
 4     Declare Integer y = 100
 5     Declare Integer z = 101
 6
 7     // Display the values in those variables.
 8     Display "x is set to ", x
 9     Display "y is set to ", y
10     Display "z is set to ", z
11
12     // Pass each variable to setToZero.
13     Call setToZero(x)
14     Call setToZero(y)
15     Call setToZero(z)
16
17     // Display the values now.
18     Display "---------------"
19     Display "x is set to ", x
20     Display "y is set to ", y
21     Display "z is set to ", z
22  End Module
23
24  Module setToZero(Integer Ref value)
25     Set value = 0
26  End Module
```

Program Output

```
x is set to 99
y is set to 100
z is set to 101
---------------
x is set to 0
y is set to 0
z is set to 0
```

In the `main` module the variable `x` is initialized with 99, the variable `y` is initialized with 100, and the variable `z` is initialized with 101. Then, in lines 13 through 15 those variables are passed as arguments to the `setToZero` module. Each time `setToZero` is called, the variable that is passed as an argument is set to 0. This is shown when the values of the variables are displayed in lines 19 through 21.

NOTE: In an actual program you should never use variable names like `x`, `y`, and `z`. This particular program is meant for demonstration purposes, however, and these simple names are adequate.

NOTE: Normally, only variables may be passed by reference. If you attempt to pass a non-variable argument into a reference variable parameter, an error will result. Using the `setToZero` module as an example, the following statement will generate an error:

```
// This is an error!
setToZero(5);
```

WARNING! Be careful when using reference variables as parameters. Any time you allow a module to alter a variable that's outside the module, you are creating potential debugging problems. Reference variables should only be used as parameters when the situation requires them.

In the Spotlight:

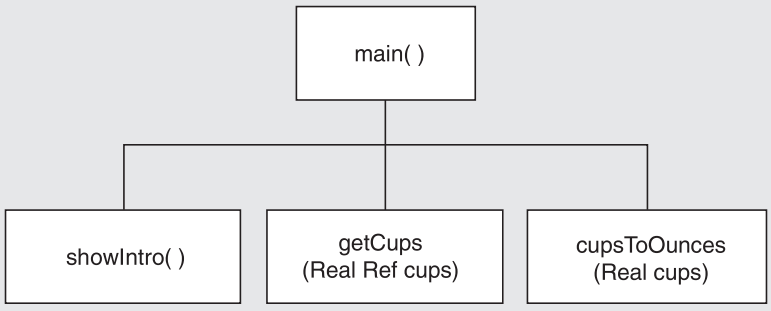
Passing an Argument by Reference

In the previous *In the Spotlight* case study, we developed a program that your friend Michael can use in his catering business. The program does exactly what Michael wants it to do: it converts cups to fluid ounces. After studying the program that we initially wrote, however, you believe that you can improve the design. As shown in the following pseudocode, the `main` module contains the code that reads the user's input. This code should really be treated as a separate subtask, and put in its own module. If this change is made, the program will be like the new hierarchy chart shown in Figure 3-18.

```
Module main()
   // Declare a variable for the
   // number of cups needed.
   Declare Real cupsNeeded

   // Display an intro message.
   Call showIntro()

   // Get the number of cups.
   Display "Enter the number of cups."
   Input cupsNeeded

   // Convert cups to ounces.
   Call cupsToOunces(cupsNeeded)
End Module
```

This code can be put in its own module.

Figure 3-18 Revised hierarchy chart



This version of the hierarchy chart shows a new module: `getCups`. Here is the pseudocode for the `getCups` module:

```
Module getCups(Real Ref cups)
   Display "Enter the number of cups."
   Input cups
End Module
```

The `getCups` module has a parameter, `cups`, which is a reference variable. The module prompts the user to enter the number of cups and then stores the user's input in the `cups` parameter. When the `main` module calls `getCups`, it will pass the local variable `cupsNeeded` as an argument. Because it will be passed by reference, it will contain the user's input when the module returns. Program 3-11 shows the revised pseudocode for the program, and Figure 3-19 shows a flowchart.

NOTE: In this case study, we improved the design of an existing program without changing the behavior of the program. In a nutshell, we "cleaned up" the design. Programmers call this *refactoring*.

Program 3-11

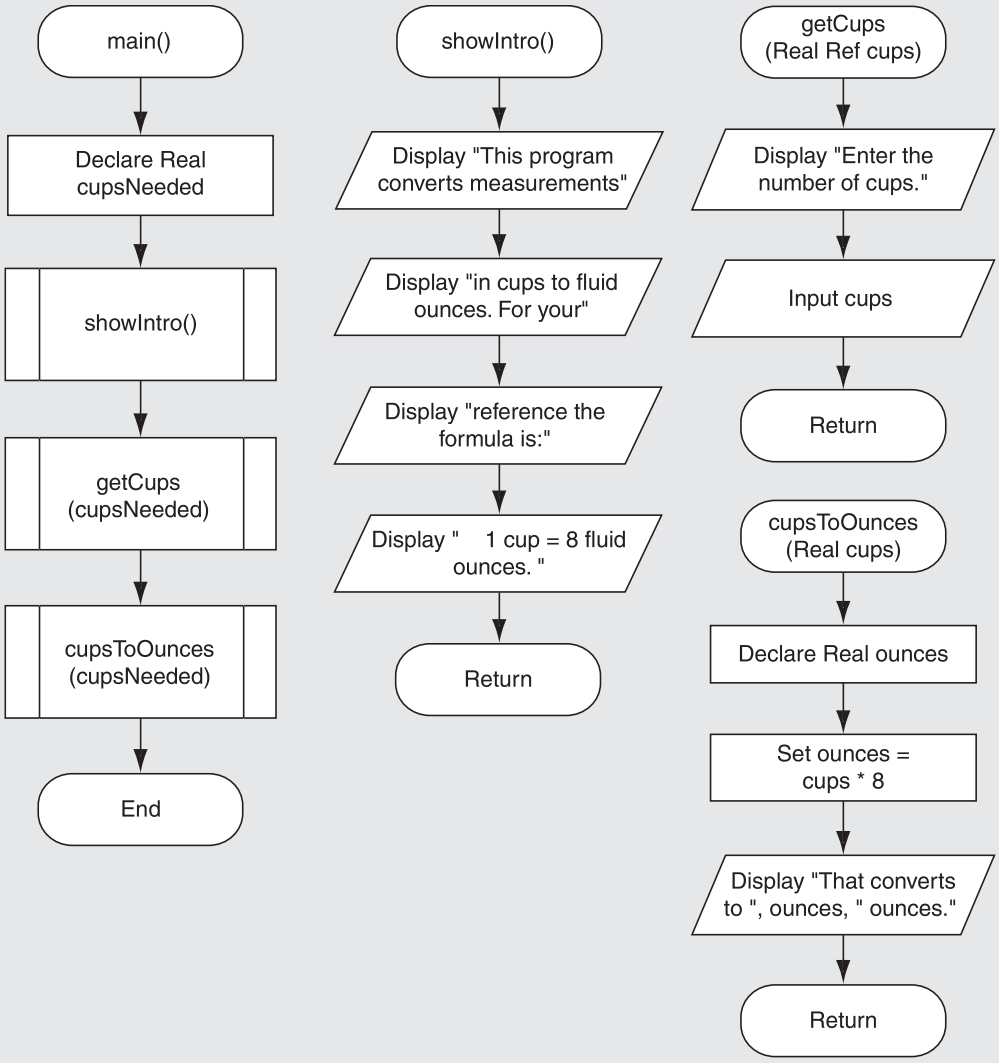
```
 1  Module main()
 2     // Declare a variable for the
 3     // number of cups needed.
 4     Declare Real cupsNeeded
 5
 6     // Display an intro message.
 7     Call showIntro()
 8
 9     // Get the number of cups.
10     Call getCups(cupsNeeded)
11
12     // Convert cups to ounces.
13     Call cupsToOunces(cupsNeeded)
14  End Module
15
16  // The showIntro module displays an
17  // introductory screen.
18  Module showIntro()
19     Display "This program converts measurements"
20     Display "in cups to fluid ounces. For your"
21     Display "reference the formula is:"
22     Display "    1 cup = 8 fluid ounces."
23  End Module
24
25  // The getCups module gets the number of cups
26  // and stores it in the reference variable cups.
27  Module getCups(Real Ref cups)
28     Display "Enter the number of cups."
29     Input cups
30  End Module
31
32  // The cupsToOunces module accepts a number
33  // of cups and displays the equivalent number
34  // of ounces.
35  Module cupsToOunces(Real cups)
36     // Declare variables.
37     Declare Real ounces
38
39     // Convert cups to ounces.
```

```
40      Set ounces = cups * 8
41
42      // Display the result.
43      Display "That converts to ",
44              ounces, " ounces."
45  End Module
```

Program Output (with Input Shown in Bold)

```
This program converts measurements
in cups to fluid ounces. For your
reference the formula is:
    1 cup = 8 fluid ounces.
Enter the number of cups.
2 [Enter]
That converts to 16 ounces.
```

Figure 3-19 Flowchart for Program 3-11



Checkpoint

3.14 What are the pieces of data that are passed into a module called?

3.15 What are the variables that receive pieces of data in a module called?

3.16 Does it usually matter whether an argument's data type is different from the data type of the parameter that it is being passed to?

3.17 Typically, what is a parameter variable's scope?

3.18 Explain the difference between passing by value and passing by reference.

3.5 Global Variables and Global Constants

CONCEPT: **A global variable is accessible to all the modules in a program.**

Global Variables

A *global variable* is a variable that is visible to every module in the program. A global variable's scope is the entire program, so all of the modules in the program can access a global variable. In most programming languages, you create a global variable by writing its declaration statement outside of all the modules, usually at the top of the program. Program 3-12 shows how you can declare a global variable in pseudocode.

Program 3-12

```
1  // The following declares a global Integer variable.
2  Declare Integer number
3
4  // The main module
5  Module main()
6     // Get a number from the user and store it
7     // in the global variable number.
8     Display "Enter a number."
9     Input number
10
11    // Call the showNumber module.
12    Call showNumber()
13 End Module
14
15 // The showNumber module displays the contents
16 // of the global variable number.
17 Module showNumber()
18    Display "The number you entered is ", number
19 End Module
```

Program Output (with Input Shown in Bold)

```
Enter a number.
22 [Enter]
The number you entered is 22
```

Line 2 declares an `Integer` variable named `number`. Because the declaration does not appear inside a module, the `number` variable is a global variable. All of the modules that are defined in the program have access to the variable. When the `Input` statement in line 9 (inside the `main` module) executes, the value entered by the user is stored in the global variable `number`. When the `Display` statement in line 18 (inside the `showNumber` module) executes, it is the value of the same global variable that is displayed.

Most programmers agree that you should restrict the use of global variables, or not use them at all. The reasons are as follows:

- Global variables make debugging difficult. Any statement in a program can change the value of a global variable. If you find that the wrong value is being stored in a global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.
- Modules that use global variables are usually dependent on those variables. If you want to use such a module in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- Global variables make a program hard to understand. A global variable can be modified by any statement in the program. If you are to understand any part of the program that uses a global variable, you have to be aware of all the other parts of the program that access the global variable.

In most cases, you should declare variables locally and pass them as arguments to the modules that need to access them.

Global Constants

Although you should try to avoid the use of global variables, it is permissible to use global constants in a program. A *global constant* is a named constant that is available to every module in the program. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about many of the potential hazards that are associated with the use of global variables.

Global constants are typically used to represent unchanging values that are needed throughout a program. For example, suppose a banking program uses a named constant to represent an interest rate. If the interest rate is used in several modules, it is easier to create a global constant, rather than a local named constant in each module. This also simplifies maintenance. If the interest rate changes, only the declaration of the global constant has to be changed, instead of several local declarations.

In the Spotlight:

Using Global Constants

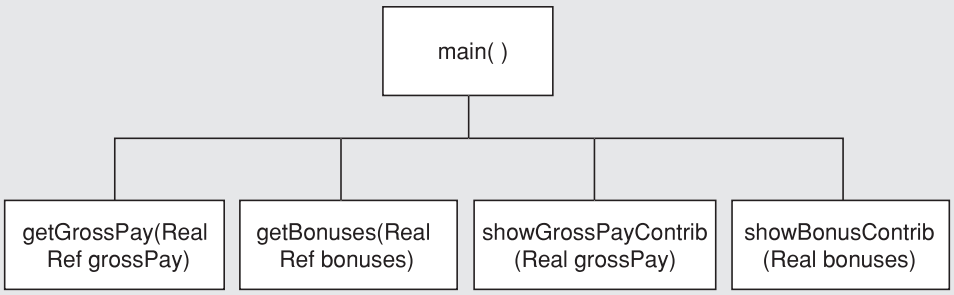
Marilyn works for Integrated Systems, Inc., a software company that has a reputation for providing excellent fringe benefits. One of its benefits is a quarterly bonus that is paid to all employees. Another benefit is a retirement plan for each employee. The company contributes 5 percent of each employee's gross pay and bonuses to his or her retirement plan. Marilyn wants to design a program that will calculate the company's contribution to an employee's retirement account for a year. She wants the program to show the amount of contribution for the employee's gross pay and for the bonuses separately.

Here is an algorithm for the program:

1. Get the employee's annual gross pay.
2. Get the amount of bonuses paid to the employee.
3. Calculate and display the contribution for the gross pay.
4. Calculate and display the contribution for the bonuses.

Figure 3-20 shows a hierarchy chart for the program. The pseudocode for the program is shown in Program 3-13, and a set of flowcharts is shown in Figure 3-21.

Figure 3-20 Hierarchy chart



Program 3-13

```
1  // Global constant for the rate of contribution.
2  Constant Real CONTRIBUTION_RATE = 0.05
3
4  // main module
5  Module main()
6     // Local variables
7     Declare Real annualGrossPay
8     Declare Real totalBonuses
9
10    // Get the annual gross pay.
11    Call getGrossPay(annualGrossPay)
12
13    // Get the total of the bonuses.
14    Call getBonuses(totalBonuses)
15
16    // Display the contribution for
17    // the gross pay.
18    Call showGrossPayContrib(annualGrossPay)
19
20    // Display the contribution for
21    // the bonuses.
22    Call showBonusContrib(totalBonuses)
23 End Module
24
25 // The getGrossPay module gets the
26 // gross pay and stores it in the
27 // grossPay reference variable.
28 Module getGrossPay(Real Ref grossPay)
29    Display "Enter the total gross pay."
```

```
30     Input grossPay
31  End Module
32
33  // The getBonuses module gets the
34  // amount of bonuses and stores it
35  // in the bonuses reference variable.
36  Module getBonuses(Real Ref bonuses)
37     Display "Enter the amount of bonuses."
38     Input bonuses
39  End Module
40
41  // The showGrossPayContrib module
42  // accepts the gross pay as an argument
43  // and displays the retirement contribution
44  // for gross pay.
45  Module showGrossPayContrib(Real grossPay)
46     Declare Real contrib
47     Set contrib = grossPay * CONTRIBUTION_RATE
48     Display "The contribution for the gross pay"
49     Display "is $", contrib
50  End Module
51
52  // The showBonusContrib module accepts
53  // the bonus amount as an argument and
54  // displays the retirement contribution
55  // for bonuses.
56  Module showBonusContrib(Real bonuses)
57     Declare Real contrib
58     Set contrib = bonuses * CONTRIBUTION_RATE
59     Display "The contribution for the bonuses"
60     Display "is $", contrib
61  End Module
```

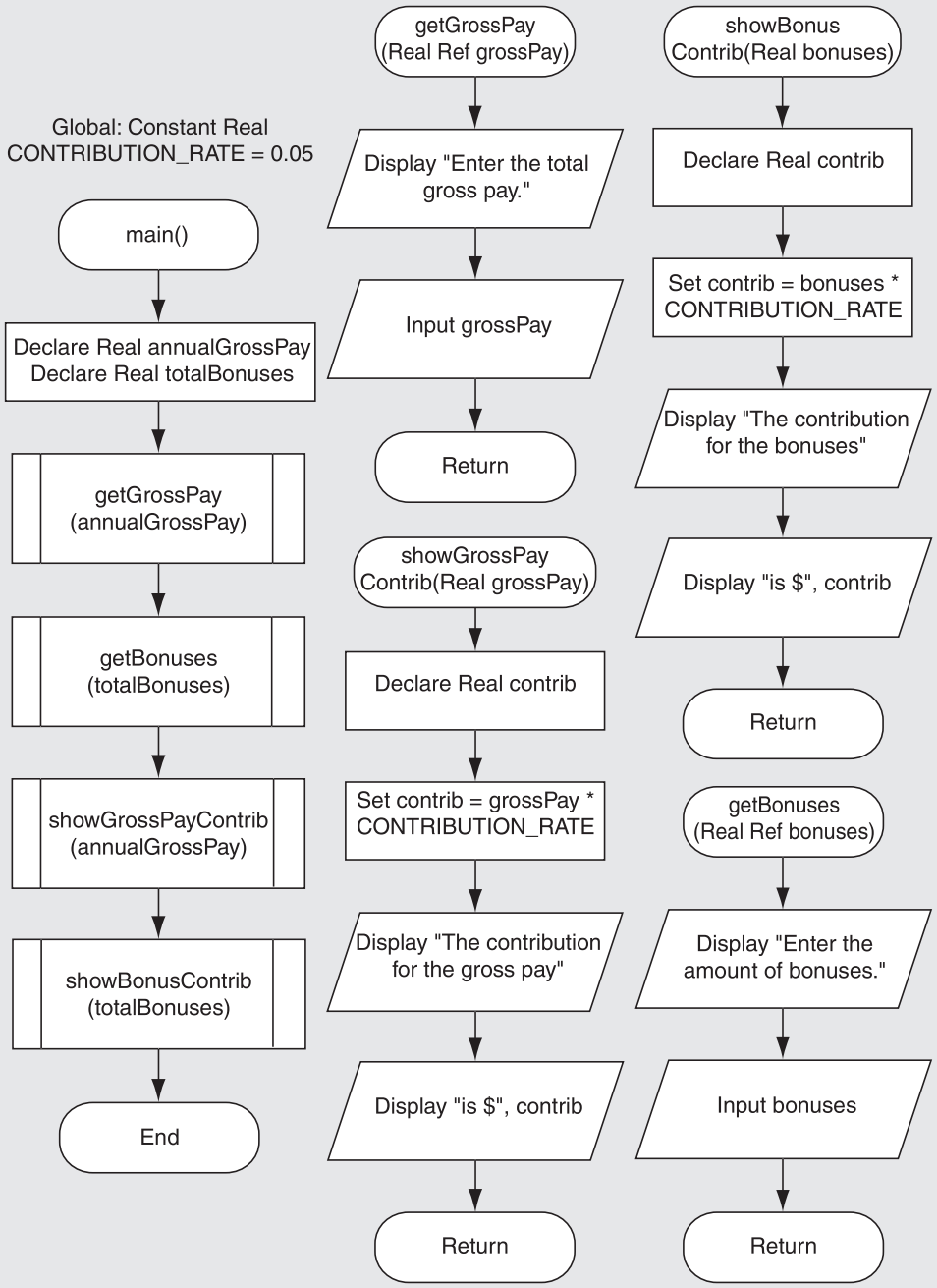
Program Output (with Input Shown in Bold)

```
Enter the total gross pay.
80000.00 [Enter]
Enter the amount of bonuses.
20000.00 [Enter]
The contribution for the gross pay
is $4000
The contribution for the bonuses
is $1000
```

A global constant named `CONTRIBUTION_RATE` is declared in line 2, and initialized with the value 0.05. The constant is used in the calculation in line 47 (in the `showGrossPayContrib` module) and again in line 58 (in the `showBonusContrib` module). Marilyn decided to use this global constant to represent the 5 percent contribution rate for two reasons:

- It makes the program easier to read. When you look at the calculations in lines 47 and 58, it is apparent what is happening.
- Occasionally the contribution rate changes. When this happens, it will be easy to update the program by changing the declaration statement in line 2.

Figure 3-21 Flowchart for Program 3-13



Checkpoint

3.19 What is the scope of a global variable?

3.20 Give one good reason that you should not use global variables in a program.

3.21 What is a global constant? Is it permissible to use global constants in a program?