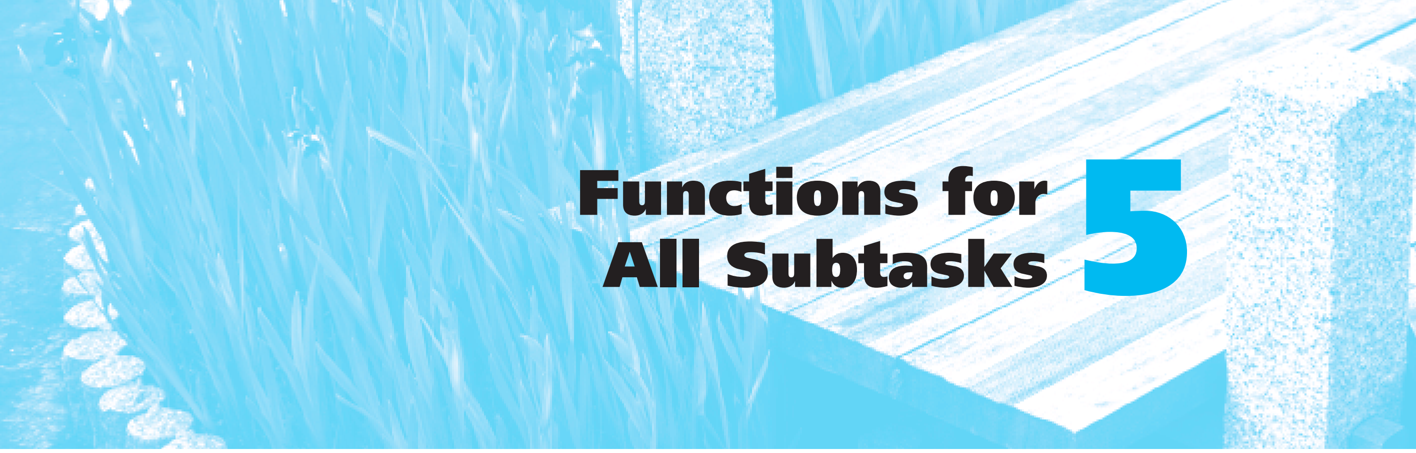


Functions for All Subtasks 5

Everything is possible.

COMMON MAXIM

INTRODUCTION

The top-down design strategy discussed in Chapter 4 is an effective way to design an algorithm for a program. You divide the program's task into subtasks and then implement the algorithms for these subtasks as functions. Thus far, we have seen how to define functions that start with the values of some arguments and return a single value as the result of the function call. A subtask that computes a single value is a very important kind of subtask, but it is not the only kind. In this chapter we will complete our description of C++ functions and present techniques for designing functions that perform other kinds of subtasks.

PREREQUISITES

You should read Chapters 2 through 4 before reading this chapter.

5.1 *void* FUNCTIONS

void functions return no value

Subtasks are implemented as functions in C++. The functions discussed in Chapter 4 always return a single value, but there are other forms of subtasks. A subtask might produce several values or it might produce no values at all. In C++, a function must either return a single value or return no values at all. As we will see later in this chapter, a subtask that produces several different values is usually (and perhaps paradoxically) implemented as a function that returns no value. For the moment, however, let us avoid that complication and focus on subtasks that intuitively produce no values at all, and let us see how these subtasks are implemented. A function that returns no value is called a *void* function. For example, one typical subtask for a program is to output the results of some calculation. This subtask produces output on the screen, but it produces no values for the rest of the program to use. This kind of subtask would be implemented as a *void* function.

Definitions of *void* Functions

In C++ a *void* function is defined in almost the same way as a function that returns a value. For example, the following is a *void* function that outputs the result of a calculation that converts a temperature expressed in Fahrenheit degrees to a temperature expressed in Celsius degrees. The actual calculation would be done elsewhere in the program. This *void* function implements only the subtask for outputting the results of the calculation. For now, we do not need to worry about how the calculation will be performed.

```
void show_results(double f_degrees, double c_degrees)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << f_degrees
         << " degrees Fahrenheit is equivalent to\n"
         << c_degrees << " degrees Celsius.\n";
    return;
}
```

Function definition

As this function definition illustrates, there are only two differences between a function definition for a *void* function and the function definitions we discussed in Chapter 4. One difference is that we use the keyword *void* where we would normally specify the type of the value to be returned. This tells the compiler that this function will not return any value. The name *void* is used as a way of saying "no value is returned by this function." The second difference is that the *return* statement does not contain an expression for a value to be returned, because, after all, there is no value returned. The syntax is summarized in Display 5.1.

Function call

A *void* function call is an executable statement. For example, our function `show_results` might be called as follows:

```
show_results(32.5, 0.3);
```

If this statement were executed in a program, it would cause the following to appear on the screen:

```
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

Notice that the function call ends with a semicolon, which tells the compiler that the function call is an executable statement.

When a *void* function is called, the arguments are substituted for the formal parameters and the statements in the function body are executed. For example, a call to the *void* function `show_results`, which we gave earlier in this section, will cause some output to be written to the screen. One way to think of a call to a *void* function is to imagine that the body of the function definition is copied into the program in place of the function call. When the function is called, the arguments are substituted for the formal parameters, and then it is just as if the body of the function were lines in the program.

DISPLAY 5.1 Syntax for a *void* Function Definition

void Function Declaration

```
void Function_Name(Parameter_List);
Function_Declaration_Comment
```

void Function Definition

```
void Function_Name(Parameter_List)
{
    Declaration_1
    Declaration_2
        . . .
    Declaration_Last
    Executable_Statement_1
    Executable_Statement_2
        . . .
    Executable_Statement_Last
{
```

function header

You may intermix the declarations with the executable statements

body

May (or may not) include one or more `return` statements.

Functions with no arguments

It is perfectly legal, and sometimes useful, to have a function with no arguments. In that case, there simply are no formal parameters listed in the function declaration and no arguments are used when the function is called. For example, the *void* function `initialize_screen`, defined next, simply sends a new line command to the screen:

```
void initialize_screen()
{
    using namespace std;
    cout << endl;
    return;
}
```

If your program includes the following call to this function as its first executable statement, then the output from the previously run program will be separated from the output for your program:

```
initialize_screen();
```

Be sure to notice that even when there are no parameters to a function, you still must include the parentheses in the function declaration and in a call to the function. The next programming example shows these two sample *void* functions in a complete program.

PROGRAMMING EXAMPLE Converting Temperatures

The program in Display 5.2 takes a Fahrenheit temperature as input and outputs the equivalent Celsius temperature. A Fahrenheit temperature F can be converted to an equivalent Celsius temperature C as follows:

```
C = (5/9)(F - 32)
```

The function `celsius` shown in Display 5.2 uses this formula to do the temperature conversion.

return Statements in *void* Functions

void functions and *return* statements

Both *void* functions and functions that return a value can have *return* statements. In the case of a function that returns a value, the *return* statement specifies the value returned. In the case of a *void* function, the *return* statement simply ends the function call. As we saw in the previous chapter, every function that returns a value must end by executing a *return* statement. However, a *void* function need not contain a *return* statement. If it does not contain a *return* statement, it will end after executing the code in the function body. It is as if there were an implicit *return* statement just before the final closing brace } at the end of the function body. For example, the functions `initialize_screen` and `show_results` in Display 5.2 would perform exactly the same if we omitted the *return* statements from their function definitions.

The fact that there is an implicit *return* statement before the final closing brace in a function body does not mean that you never need a *return* statement in a *void* function. For example, the function definition in Display 5.3 might be used as part of a restaurant management program. That function outputs instructions for dividing a given amount of ice cream among the people at a table. If there are no people at the table (that is, if `number` equals `0`), then the *return* statement within the *if* statement terminates the function call and avoids a division by zero. If `number` is not `0`, then the function call ends when the last `cout` statement is executed at the end of the function body.

The `main` part of a program is a function

By now you may have guessed that the `main` part of a program is actually the definition of a function called `main`. When the program is run, the function `main` is automatically called and it, in turn, may call other functions. Although it may seem that the *return* statement in the `main` part of a program should be optional, officially it is not. Technically, the `main` part of a program is a function that returns a value of type *int*, so it requires a *return* statement. However, the function `main` is used as if it were a *void* function. Treating the `main` part of your program as a function that returns an integer may sound

DISPLAY 5.2 *void* **Functions** *(part 1 of 2)*

```
//Program to convert a Fahrenheit temperature to a Celsius temperature.
#include <iostream>

void initialize_screen( );
//Separates current output from
//the output of the previously run program.

double celsius(double fahrenheit);
//Converts a Fahrenheit temperature
//to a Celsius temperature.

void show_results(double f_degrees, double c_degrees);
//Displays output. Assumes that c_degrees
//Celsius is equivalent to f_degrees Fahrenheit.

int main( )
{
    using namespace std;
    double f_temperature, c_temperature;

    initialize_screen( );
    cout << "I will convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << "Enter a temperature in Fahrenheit: ";
    cin >> f_temperature;

    c_temperature = celsius(f_temperature);

    show_results(f_temperature, c_temperature);
    return 0;
}

//Definition uses iostream:
void initialize_screen( )
{
    using namespace std;
    cout << endl;
    return;        <-- This return is optional.
}
double celsius(double fahrenheit)
{
    return ((5.0/9.0)*(fahrenheit - 32));
}
//Definition uses iostream:
void show_results(double f_degrees, double c_degrees)
{
```

(continued)

DISPLAY 5.2 ***void* Functions** *(part 2 of 2)*

```
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << f_degrees
         << " degrees Fahrenheit is equivalent to\n"
         << c_degrees << " degrees Celsius.\n";
    return;                                    This return is optional.
}
```

Sample Dialogue

```
I will convert a Fahrenheit temperature to Celsius.
Enter a temperature in Fahrenheit: 32.5
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

DISPLAY 5.3 **Use of *return* in a *void* Function**

Function Declaration

```
void ice_cream_division(int number, double total_weight);
//Outputs instructions for dividing total_weight ounces of
//ice cream among number customers.
//If number is 0, nothing is done.
```

Function Definition

```
//Definition uses iostream:
void ice_cream_division(int number, double total_weight)
{
    using namespace std;
    double portion;
                                      If number is 0, then the
    if (number == 0)                  function execution ends here.
        return;
    portion = total_weight/number;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Each one receives "
         << portion << " ounces of ice cream." << endl;
}
```

crazy, but that's the tradition. It might be best to continue to think of the `main` part of the program as just "the main part of the program" and not worry about this minor detail.[1]

SELF-TEST EXERCISES

1. What is the output of the following program?

```
#include <iostream>
void friendly();
void shy(int audience_count);
int main()
{
    using namespace std;
    friendly();
    shy(6);
    cout << "One more time:\n";
    shy(2);
    friendly();
    cout << "End of program.\n";
    return 0;
}

void friendly()
{
    using namespace std;
    cout << "Hello\n";
}

void shy(int audience_count)
{
    using namespace std;
    if (audience_count < 5)
        return;
    cout << "Goodbye\n";
}
```

2. Are you required to have a *return* statement in a *void* function definition?

3. Suppose you omitted the *return* statement in the function definition for `initialize_screen` in Display 5.2. What effect would it have on the program? Would the program compile? Would it run? Would the program behave any differently? What about the *return* statement in the function

[1] The C++ Standard says that you can omit the *return* `0` in the main part, but many compilers still require it.

definition for `show_results` in that same program? What effect would it have on the program if you omitted the *`return`* statement in the definition of `show_results`? What about the *`return`* statement in the function definition for `celsius` in that same program? What effect would it have on the program if you omitted the *`return`* statement in the definition of `celsius`?

4. Write a definition for a *`void`* function that has three arguments of type *`int`* and that outputs to the screen the product of these three arguments. Put the definition in a complete program that reads in three numbers and then calls this function.

5. Does your compiler allow *`void`* `main()` and *`int`* `main()`? What warnings are issued if you have *`int`* `main()` and do not supply a *`return`* `0;` statement? To find out, write several small test programs and perhaps ask your instructor or a local guru.

6. Is a call to a *`void`* function used as a statement or is it used as an expression?

5.2 CALL-BY-REFERENCE PARAMETERS

When a function is called, its arguments are substituted for the formal parameters in the function definition, or to state it less formally, the arguments are "plugged in" for the formal parameters. There are different mechanisms used for this substitution process. The mechanism we used in Chapter 4, and thus far in this chapter, is known as the ***call-by-value*** mechanism. The second main mechanism for substituting arguments is known as the ***call-by-reference*** mechanism.

A First View of Call-by-Reference

The call-by-value mechanism that we used until now is not sufficient for certain subtasks. For example, one common subtask is to obtain one or more input values from the user. Look back at the program in Display 5.2. Its tasks are divided into four subtasks: initialize the screen, obtain the Fahrenheit temperature, compute the corresponding Celsius temperature, and output the results. Three of these four subtasks are implemented as the functions `initialize_screen, celsius,` and `show_results`. However, the subtask of obtaining the input is implemented as the following four lines of code (rather than as a function call):

```
cout << "I will convert a Fahrenheit temperature"
     << " to Celsius.\n"
     << "Enter a temperature in Fahrenheit: ";
cin >> f_temperature;
```

The subtask of obtaining the input should be accomplished by a function call. To do this with a function call, we will use a call-by-reference parameter.

A function for obtaining input should set the values of one or more variables to values typed in at the keyboard, so the function call should have one or more variables as arguments and should change the values of these argument variables. With the call-by-value formal parameters that we have used until now, an argument in a function call can be a variable, but the function takes only the value of the variable and does not change the variable in any way. With a call-by-value formal parameter only the value of the argument is substituted for the formal parameter. For an input function, we want the variable (not the value of the variable) to be substituted for the formal parameter. The call-by-reference mechanism works in just this way. With a **call-by-reference** formal parameter (also called simply a **reference** parameter), the corresponding argument in a function call must be a variable and this argument variable is substituted for the formal parameter. It is as if the argument variable were literally copied into the body of the function definition in place of the formal parameter. After the argument is substituted in, the code in the function body is executed and this code can change the value of the argument variable.

A call-by-reference parameter must be marked in some way so that the compiler will know it from a call-by-value parameter. The way that you indicate a call-by-reference parameter is to attach the ampersand sign, &, to the end of the type name in the formal parameter list in both the function declaration and the header of the function definition. For example, the following function definition has one formal parameter, `f_variable`, and that formal parameter is a call-by-reference parameter:

```
void get_input (double& f_variable)
{
    using namespace std;
    cout << "I will convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << "Enter a temperature in Fahrenheit: ";
    cin >> f_variable;
}
```

In a program that contains this function definition, the following function call sets the variable `f_temperature` equal to a value read from the keyboard:

```
get_input(f_temperature);
```

Using this function definition, we could easily rewrite the program shown in Display 5.2 so that the subtask of reading the input is accomplished by this function call. However, rather than rewrite an old program, let's look at a completely new program.

Display 5.4 demonstrates call-by-reference parameters. The program doesn't do very much. It just reads in two numbers and writes the same numbers out, but in the reverse order. The parameters in the functions `get_numbers` and `swap_values` are call-by-reference parameters. The input is performed by the function call

```
get_numbers(first_num, second_num);
```

DISPLAY 5.4 Call-by-Reference Parameters

```
//Program to demonstrate call-by-reference parameters.
#include <iostream>

void get_numbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swap_values(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void show_results(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.
int main( )
{
    int first_num = 0, second_num = 0;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

//Uses iostream:
void get_numbers (int& input1, int& input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}
void swap_values(int& variable1, int& variable2)
{
    int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

//Uses iostream:
void show_results(int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Sample Dialogue

```
Enter two integers: 5 10
In reverse order the numbers are: 10 5
```

The values of the variables `first_num` and `second_num` are set by this function call. After that, the following function call reverses the values in the two variables `first_num` and `second_num`:

```
swap_values(first_num, second_num);
```

In the next few subsections we describe the call-by-reference mechanism in more detail and also explain the particular functions used in Display 5.4.

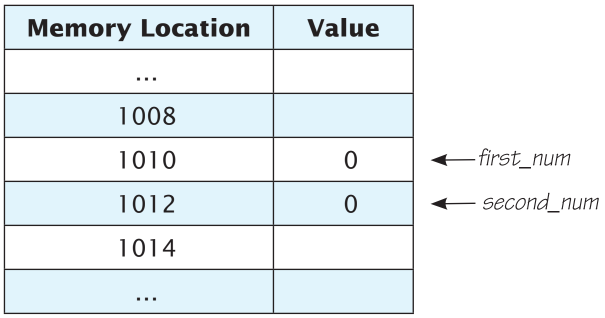
Call-by-Reference in Detail

In most situations, the call-by-reference mechanism works as if the name of the variable given as the function argument were literally substituted for the call-by-reference formal parameter. However, the process is a bit more subtle than that. In some situations, this subtlety is important, so we need to examine more details of this call-by-reference substitution process.

Recall that program variables are implemented as memory locations. The compiler assigns one memory location to each variable. For example, when the program in Display 5.4 is compiled, the variable `first_num` might be assigned location 1010, and the variable `second_num` might be assigned 1012. For purposes of this example, consider these variables to be stored at these memory locations. In other words, after executing the line

```
int first_num = 0, second_num = 0;
```

the value 0 will be stored at memory locations 1010 and 1012. The arrows in the diagram below point to the memory locations referenced by the variables.



Memory Location	Value
...	
1008	
1010	0
1012	0
1014	
...	

Next, consider the following function declaration from Display 5.4:

```
void get_numbers(int& input1, int& input2);
```

The call-by-reference formal parameters `input1` and `input2` are place holders for the actual arguments used in a function call.

Call-by-Reference

To make a formal parameter a **call-by-reference** parameter, append the **ampersand sign &** to its type name. The corresponding argument in a call to the function should then be a variable, not a constant or other expression. When the function is called, the corresponding variable argument (not its value) will be substituted for the formal parameter. Any change made to the formal parameter in the function body will be made to the argument variable when the function is called. The exact details of the substitution mechanisms are given in the text of this chapter.

EXAMPLE (OF CALL-BY-REFERENCE PARAMETERS IN A FUNCTION DECLARATION):

```
void get_data(int& first_in, double& second_in);
```

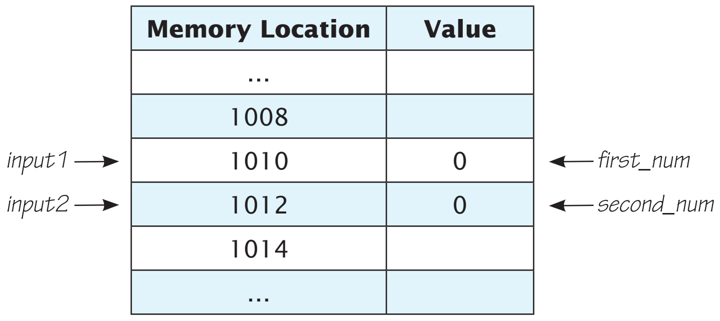
Now consider a function call like the following from the same display:

```
get_numbers(first_num, second_num);
```

When the function call is executed, the function is not given values stored in `first_num` and `second_num`. Instead, it is given the memory locations associated with each name. In this example, the locations are

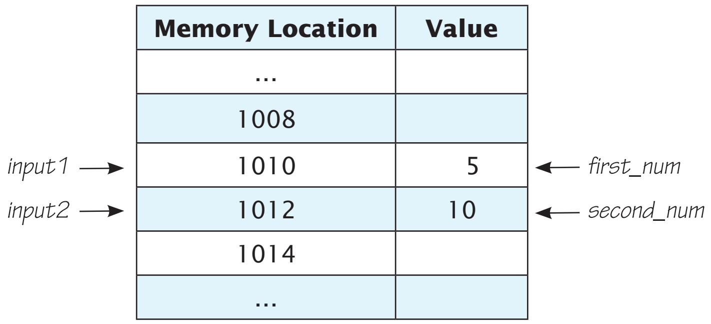
```
1010
1012
```

which are the locations assigned to the argument variables `first_num` and `second_num`, in that order. It is these memory locations that are associated with the formal parameters. The first memory location is associated with the first formal parameter, the second memory location is associated with the second formal parameter, and so forth. In our example `input1` is the first parameter, so it gets the same memory location as `first_num`. The second parameter is `input2` and it gets the same memory location as `second_num`. Diagrammatically, the correspondence is



Memory Location	Value
...	
1008	
1010	0
1012	0
1014	
...	

When the function statements are executed, whatever the function body says to do to a formal parameter is actually done to the variable in the memory location associated with that formal parameter. In this case, the instructions in the body of the function `get_numbers` say that a value should be stored in the formal parameter `input1` using a `cin` statement, and so that value is stored in the variable in memory location 1010 (which happens to be where the variable `first_num` is stored). Similarly, the instructions in the body of the function `get_numbers` say that a value should then be stored in the formal parameter `input2` using a `cin` statement, and so that value is stored in the variable in memory location 1012 (which happens to be where the variable `second_num` is stored). Thus, whatever the function instructs the computer to do to `input1` and `input2` is actually done to the variables `first_num` and `second_num`. For example, if the user enters `5` and `10` as in Display 5.4, then the result is



When the function `get_numbers` exits, the variables `input1` and `input2` go out of scope and are lost. This means we can no longer retrieve the data values at 1010 and 1012 through the variables `input1` and `input2`. However, the data still exists in memory location 1010 and 1012 and is accessible through the variables `first_num` and `second_num` within the scope of the `main` function. These details of how the call-by-reference mechanism works in this function call to `get_numbers` are described in Display 5.5.

It may seem that there is an extra level of detail, or at least an extra level of verbiage. If `first_num` is the variable with memory location 1010, why do we insist on saying "the variable at memory location 1010" instead of simply saying "`first_num`"? This extra level of detail is needed if the arguments and formal parameters contain some confusing coincidence of names. For example, the function `get_numbers` has formal parameters named `input1` and `input2`. Suppose you want to change the program in Display 5.4 so that it uses the function `get_numbers` with arguments that are also named `input1` and `input2`, and suppose that you want to do something less than obvious. Suppose you want the first number typed in to be stored in a variable named `input2`, and the second

DISPLAY 5.5 Behavior of Call-by-Reference Arguments *(part 1 of 2)*

Anatomy of a Function Call from Display 5.4 Using Call-by-Reference Arguments

0 Assume the variables `first_num` and `second_num` have been assigned the following memory address by the compiler:

```
first_num  ──────► 1010
second_num ──────► 1012
```

(We do not know what addresses are assigned and the results will not depend on the actual addresses, but this will make the process very concrete and thus perhaps easier to follow.)

1 In the program in Display 5.4, the following function call begins executing:

```
get_numbers(first_num, second_num);
```

2 The function is told to use the memory location of the variable `first_num` in place of the formal parameter `input1` and the memory location of the `second_num` in place of the formal parameter `input2`. The effect is the same as if the function definition were rewritten to the following (which is not legal C++ code, but does have a clear meaning to us):

```
void get_numbers(int& <the variable at memory location 1010>,
                 int& <the variable at memory location 1012>)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> <the variable at memory location 1010>
        >> <the variable at memory location 1012>;
}
```

Anatomy of the Function Call in Display 5.4 (*concluded*)

Since the variables in locations 1010 and 1012 are `first_num` and `second_num`, the effect is thus the same as if the function definition were rewritten to the following:

```
void get_numbers(int& first_num, int& second_num)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num
        >> second_num;
}
```

3 The body of the function is executed. The effect is the same as if the following were executed:

(continued)

DISPLAY 5.5 Behavior of Call-by-Reference Arguments *(part 2 of 2)*

```
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num
        >> second_num;
}
```

4 When the `cin` statement is executed, the values of the variables `first_num` and `second_num` are set to the values typed in at the keyboard. (If the dialogue is as shown in Display 5.4, then the value of `first_num` is set to 5 and the value of `second_num` is set to 10.)

5 When the function call ends, the variables `first_num` and `second_num` retain the values that they were given by the `cin` statement in the function body. (If the dialogue is as shown in Display 5.4, then the value of `first_num` is 5 and the value of `second_num` is 10 at the end of the function call.)

number typed in to be stored in the variable named `input1`—perhaps because the second number will be processed first, or because it is the more important number. Now, let's suppose that the variables `input1` and `input2`, which are declared in the `main` part of your program, have been assigned memory locations 1014 and 1016. The function call could be as follows:

```
int input1, input 2;
get_numbers(input2, input1);
```

Notice the order of the arguments

In this case if you say "`input1`," we do not know whether you mean the variable named `input1` that is declared in the `main` part of your program or the formal parameter `input1`. However, if the variable `input1` declared in the `main` part of your program is assigned memory location 1014, the phrase "the variable at memory location 1014" is unambiguous. Let's go over the details of the substitution mechanisms in this case.

In this call the argument corresponding to the formal parameter `input1` is the variable `input2`, and the argument corresponding to the formal parameter `input2` is the variable `input1`. This can be confusing to us, but it produces no problem at all for the computer, since the computer never does actually "substitute `input2` for `input1`" or "substitute `input1` for `input2`." The computer simply deals with memory locations. The computer substitutes "the variable at memory location 1016" for the formal parameter `input1`, and "the variable at memory location 1014" for the formal parameter `input2`.

PROGRAMMING EXAMPLE The swap_values Function

The function `swap_values` defined in Display 5.4 interchanges the values stored in two variables. The description of the function is given by the following function declaration and accompanying comment:

```
void swap_values(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.
```

To see how the function is supposed to work, assume that the variable `first_num` has the value `5` and the variable `second_num` has the value `10` and consider the function call:

```
swap_values(first_num, second_num);
```

After this function call, the value of `first_num` will be **10** and the value of `second_num` will be `5`.

As shown in Display 5.4, the definition of the function `swap_values` uses a local variable called `temp`. This local variable is needed. You might be tempted to think the function definition could be simplified to the following:

```
void swap_values(int& variable1, int& variable2)
{
    variable1 = variable2;
    variable2 = variable1;
}
```

This does not work!

To see that this alternative definition cannot work, consider what would happen with this definition and the function call

```
swap_values(first_num, second_num);
```

The variables `first_num` and `second_num` are substituted for the formal parameters `variable1` and `variable2` so that, with this incorrect function definition, the function call is equivalent to the following:

```
first_num = second_num;
second_num = first_num;
```

This code does not produce the desired result. The value of `first_num` is set equal to the value of `second_num`, just as it should be. But then, the value of `second_num` is set equal to the changed value of `first_num`, which is now the original value of `second_num`. Thus the value of `second_num` is not changed at all. (If this is unclear, go through the steps with specific values for the variables `first_num` and `second_num`.) What the function needs to do is to save the original value of `first_num` so that value is not lost. This is what the local variable `temp` in the correct function definition is used for. That correct definition is the one in Display 5.4. When that correct version is used and the function is called with the arguments `first_num` and `second_num`, the function call is equivalent to the following code, which works correctly:

```
temp = first_num;
first_num = second_num;
second_num = temp;
```

Parameters and Arguments

All the different terms that have to do with parameters and arguments can be confusing. However, if you keep a few simple points in mind, you will be able to easily handle these terms.

1. The **formal parameters** for a function are listed in the function declaration and are used in the body of the function definition. A formal parameter (of any sort) is a kind of blank or place holder that is filled in with something when the function is called.
2. An **argument** is something that is used to fill in a formal parameter. When you write down a function call, the arguments are listed in parentheses after the function name. When the function call is executed, the arguments are "plugged in" for the formal parameters.
3. The terms *call-by-value* and *call-by-reference* refer to the mechanism that is used in the "plugging in" process. In the **call-by-value** method, only the value of the argument is used. In this call-by-value mechanism, the formal parameter is a local variable that is initialized to the value of the corresponding argument. In the **call-by-reference** mechanism, the argument is a variable and the entire variable is used. In the call-by-reference mechanism, the argument variable is substituted for the formal parameter so that any change that is made to the formal parameter is actually made to the argument variable.

Mixed Parameter Lists

Whether a formal parameter is a call-by-value parameter or a call-by-reference parameter is determined by whether there is an ampersand attached to its type specification. If the ampersand is present, then the formal parameter is a call-by-reference parameter. If there is no ampersand associated with the formal parameter, then it is a call-by-value parameter.

Mixing call-by-reference and call-by-value

It is perfectly legitimate to mix call-by-value and call-by-reference formal parameters in the same function. For example, the first and last of the formal parameters in the following function declaration are call-by-reference formal parameters and the middle one is a call-by-value parameter:

```
void good_stuff(int& par1, int par2, double& par3);
```

Call-by-reference parameters are not restricted to *void* functions. You can also use them in functions that return a value. Thus, a function with a call-by-reference parameter could both change the value of a variable given as an argument and return a value.

PROGRAMMING TIP What Kind of Parameter to Use

VideoNote
Call by Reference and Call by Value

Display 5.6 illustrates the differences between how the compiler treats call-by-value and call-by-reference formal parameters. The parameters `par1_value` and `par2_ref` are both assigned a value inside the body of the function definition. But since they are different kinds of parameters, the effect is different in the two cases.

`par1_value` is a call-by-value parameter, so it is a local variable. When the function is called as follows

```
do_stuff(n1, n2);
```

the local variable `par1_value` is initialized to the value of `n1`. That is, the local variable `par1_value` is initialized to `1` and the variable `n1` is then ignored by the function. As you can see from the sample dialogue, the formal parameter `par1_value` (which is a local variable) is set to `111` in the function body and this value is output to the screen. However, the value of the argument `n1` is not changed. As shown in the sample dialogue, `n1` has retained its value of `1`.

DISPLAY 5.6 Comparing Argument Mechanisms *(part 1 of 2)*

```
//Illustrates the difference between a call-by-value
//parameter and a call-by-reference parameter.
#include <iostream>

void do_stuff(int par1_value, int& par2_ref);
//par1_value is a call-by-value formal parameter and
//par2_ref is a call-by-reference formal parameter.

int main( )
{
    using namespace std;
    int n1, n2;

    n1 = 1;
    n2 = 2;
    do_stuff(n1, n2);
    cout << "n1 after function call = " << n1 << endl;
    cout << "n2 after function call = " << n2 << endl;
    return 0;
}
void do_stuff(int par1_value, int& par2_ref)
{
    using namespace std;
```

(continued)

DISPLAY 5.6 Comparing Argument Mechanisms *(part 2 of 2)*

```
        par1_value = 111;
        cout << "par1_value in function call = "
             << par1_value << endl;
        par2_ref = 222;
        cout << "par2_ref in function call = "
             << par2_ref << endl;
    }
```

Sample Dialogue

```
par1_value in function call = 111
par2_ref in function call = 222
n1 after function call = 1
n2 after function call = 222
```

On the other hand, `par2_ref` is a call-by-reference parameter. When the function is called, the variable argument `n2` (not just its value) is substituted for the formal parameter `par2_ref`. So that when the following code is executed:

```
par2_ref = 222;
```

it is the same as if the following were executed:

```
n2 = 222;
```

Thus, the value of the variable `n2` is changed when the function body is executed, so as the dialogue shows, the value of `n2` is changed from `2` to `222` by the function call.

If you keep in mind the lesson of Display 5.6, it is easy to decide which parameter mechanism to use. If you want a function to change the value of a variable, then the corresponding formal parameter must be a call-by-reference formal parameter and must be marked with the ampersand sign, `&`. In all other cases, you can use a call-by-value formal parameter. ■

PITFALL Inadvertent Local Variables

If you want a function to change the value of a variable, the corresponding formal parameter must be a call-by-reference parameter and must have the ampersand, `&`, attached to its type. If you carelessly omit the ampersand, the function will have a call-by-value parameter where you meant to have a call-by-reference parameter, and when the program is run, you will discover that the function call does not change the value of the corresponding argument. This is because a formal call-by-value parameter is a local variable, so if it has its value changed in the function, then as with any local variable, that change has no effect outside of the function body. This is a logic error that can be very difficult to see because it looks right.

For example, the program in Display 5.7 is identical to the program in Display 5.4, except that the ampersands were mistakenly omitted from the function `swap_values`. As a result, the formal parameters `variable1` and `variable2` are local variables. The argument variables `first_num` and `second_num` are never substituted in for `variable1` and `variable2`; `variable1` and `variable2` are instead initialized to the values of `first_num` and `second_num`. Then, the values of `variable1` and `variable2` are interchanged, but the values of `first_num` and `second_num` are left unchanged. The omission of two ampersands has made the program completely wrong, yet it looks almost identical to the correct program and will compile and run without any error messages. ■

DISPLAY 5.7 Inadvertent Local Variable

```cpp
//Program to demonstrate call-by-reference parameters.
#include <iostream>

void get_numbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swap_values(int variable1, int variable2);
//Interchanges the values of variable1 and variable2.

void show_results(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main( )
{
    int first_num, second_num;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

void swap_values(int variable1, int variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
        <The definitions of get_numbers and
            show_results are the same as in Display 5.4.>
```

forgot the & here

forgot the & here

inadvertent local variables

Sample Dialogue

```
Enter two integers: 5 10
In reverse order the numbers are: 5 10
```

SELF-TEST EXERCISES

7. What is the output of the following program?

```
#include <iostream>
void figure_me_out(int& x, int y, int& z);
int main()
{
    using namespace std;
    int a, b, c;
    a = 10;
    b = 20;
    c = 30;
    figure_me_out(a, b, c);
    cout << a << " " << b << " " << c;
    return 0;
}

void figure_me_out(int& x, int y, int& z)
{
    using namespace std;
    cout << x << " " << y << " " << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << " " << y << " " << z << endl;
}
```

8. What would be the output of the program in Display 5.4 if you omit the ampersands, &, from the first parameter in the function declaration and function heading of swap_values? The ampersand is not removed from the second parameter.

9. What would be the output of the program in Display 5.6 if you change the function declaration for the function do_stuff to the following and you change the function header to match, so that the formal parameter par2_ref is changed to a call-by-value parameter:

```
void do_stuff(int par1_value, int par2_ref);
```

10. Write a *void* function definition for a function called zero_both that has two reference parameters, both of which are variables of type *int*, and sets the values of both variables to 0.

11. Write a *void* function definition for a function called add_tax. The function add_tax has two formal parameters: tax_rate, which is the amount of sales tax expressed as a percentage, and cost, which is the cost of an item before tax. The function changes the value of cost so that it includes sales tax.

12. Can a function that returns a value have a call-by-reference parameter? May a function have both call-by-value and call-by-reference parameters?

5.3 USING PROCEDURAL ABSTRACTION

My memory is so bad, that many times I forget my own name!

MIGUEL DE CERVANTES SAAVEDRA, *Don Quixote*

Recall that the principle of procedural abstraction says that functions should be designed so that they can be used as black boxes. For a programmer to use a function effectively, all the programmer should need to know is the function declaration and the accompanying comment that says what the function accomplishes. The programmer should not need to know any of the details contained in the function body. In this section we discuss a number of topics that deal with this principle in more detail.

Functions Calling Functions

A function body may contain a call to another function. The situation for these sorts of function calls is exactly the same as it would be if the function call had occurred in the `main` function of the program; the only restriction is that the function declaration should appear before the function is used. If you set up your programs as we have been doing, this will happen automatically, since all function declarations come before the `main` function and all function definitions come after the `main` function. Although you may include a function call within the definition of another function, you cannot place the definition of one function within the body of another function definition.

Display 5.8 shows an enhanced version of the program shown in Display 5.4. The program in Display 5.4 always reversed the values of the variables `first_num` and `second_num`. The program in Display 5.8 reverses these variables only some of the time. The program in Display 5.8 uses the function `order` to reorder the values in these variables so as to ensure that

```
first_num <= second_num
```

If this condition is already true, then nothing is done to the variables `first_num` and `second_num`. If, however, `first_num` is greater than `second_num`, then the function `swap_values` is called to interchange the values of these two variables. This testing for order and exchanging of variable values all takes place within the body of the function `order`. Thus, the function `swap_values` is called within the body of the function `order`. This presents no special problems. Using the principle of procedural abstraction, we think of the function `swap_values` as performing an action (namely, interchanging the values of two variables); this action is the same no matter where it occurs.

DISPLAY 5.8 Function Calling Another Function *(part 1 of 2)*

```
//Program to demonstrate a function calling another function.
#include <iostream>

void get_input(int& input1, int& input2);
//Reads two integers from the keyboard.

void swap_values(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void order(int& n1, int& n2);
//Orders the numbers in the variables n1 and n2
//so that after the function call n1 <= n2.

void give_results(int output1, int output2);
//Outputs the values in output1 and output2.
//Assumes that output1 <= output2


int main( )
{
    int first_num, second_num;

    get_input(first_num, second_num);
    order(first_num, second_num);
    give_results(first_num, second_num);
    return 0;
}


//Uses iostream:
void get_input(int& input1, int& input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1 >> input2;
}

void swap_values(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

```

(continued)

DISPLAY 5.8 Function Calling Another Function *(part 2 of 2)*

```
void order(int& n1, int& n2)
{
    if (n1 > n2)
        swap_values(n1, n2);
}

//Uses iostream:
void give_results(int output1, int output2)
{
    using namespace std;
    cout << "In increasing order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

These function definitions can be in any order.

Sample Dialogue

```
Enter two integers: 10 5
In increasing order the numbers are: 5 10
```

Preconditions and Postconditions

One good way to write a function declaration comment is to break it down into two kinds of information, called a precondition and a postcondition. The **precondition** states what is assumed to be true when the function is called. The function should not be used and cannot be expected to perform correctly unless the precondition holds. The **postcondition** describes the effect of the function call; that is, the postcondition tells what will be true after the function is executed in a situation in which the precondition holds. For a function that returns a value, the postcondition will describe the value returned by the function. For a function that changes the value of some argument variables, the postcondition will describe all the changes made to the values of the arguments.

For example, the function declaration comment for the function `swap_values` shown in Display 5.8 can be put into this format as follows:

```
void swap_values(int& variable1, int& variable2);
//Precondition: variable1 and variable2 have been given
//values.
//Postcondition: The values of variable1 and variable2
//have been interchanged.
```

The comment for the function `celsius` from Display 5.2 can be put into this format as follows:

```
double celsius(double fahrenheit);
//Precondition: fahrenheit is a temperature expressed
```

```
//in degrees Fahrenheit.
//Postcondition: Returns the equivalent temperature
//expressed in degrees Celsius.
```

When the only postcondition is a description of the value returned, programmers often omit the word postcondition. A common and acceptable alternative form for the previous function declaration comments is the following:

```
//Precondition: fahrenheit is a temperature expressed
//in degrees Fahrenheit.
//Returns the equivalent temperature expressed in
//degrees Celsius.
```

Another example of preconditions and postconditions is given by the following function declaration:

```
void post_interest(double& balance, double rate);
//Precondition: balance is a nonnegative savings
//account balance.rate is the interest rate
//expressed as a percent, such as 5 for 5%.
//Postcondition: The value of balance has been
//increased by rate percent.
```

You do not need to know the definition of the function `post_interest` in order to use this function, so we have given only the function declaration and accompanying comment.

Preconditions and postconditions are more than a way to summarize a function's actions. They should be the first step in designing and writing a function. When you design a program, you should specify what each function does before you start designing how the function will do it. In particular, the function declaration comments and the function declaration should be designed and written down before starting to design the function body. If you later discover that your specification cannot be realized in a reasonable way, you may need to back up and rethink what the function should do, but by clearly specifying what you think the function should do, you will minimize both design errors and wasted time writing code that does not fit the task at hand.

Some programmers prefer not to use the words precondition and postcondition in their function comments. However, whether you use the words or not, your function comment should always contain the precondition and postcondition information.

CASE STUDY Supermarket Pricing

This case study solves a very simple programming task. It may seem that it contains more detail than is needed for such a simple task. However, if you see the design elements in the context of a simple task, you can concentrate on learning them without the distraction of any side issues. Once you learn the techniques that are illustrated in this simple case study, you can apply these same techniques to much more complicated programming tasks.

Problem Definition

We have been commissioned by the Quick-Shop supermarket chain to write a program that will determine the retail price of an item given suitable input. Their pricing policy is that any item that is expected to sell in one week or less is marked up 5 percent, and any item that is expected to stay on the shelf for more than one week is marked up 10 percent over the wholesale price. Be sure to notice that the low markup of 5 percent is used for up to 7 days and that at 8 days the markup changes to 10 percent. It is important to be precise about exactly when a program should change from one form of calculation to a different one.

As always, we should be sure we have a clear statement of the input required and the output produced by the program.

Input

The input will consist of the wholesale price of an item and the expected number of days until the item is sold.

Output

The output will give the retail price of the item.

Analysis of the Problem

Like many simple programming tasks, this one breaks down into three main subtasks:

1. Input the data.
2. Compute the retail price of the item.
3. Output the results.

These three subtasks will be implemented by three functions. The three functions are described by their function declarations and accompanying comments, which are given below. Note that only those items that are changed by the functions are call-by-reference parameters. The remaining formal parameters are call-by-value parameters.

```
void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//turnover is the expected number of days
//until sale of the item.
//Returns the retail price of the item.
```

```
void give_output(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item;
//turnover is the expected time until sale of the item;
//price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price have
//been written to the screen.
```

Now that we have the function headings, it is trivial to write the `main` part of our program:

```
int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}
```

Even though we have not yet written the function bodies and have no idea of how the functions work, we can write the above code that uses the functions. That is what is meant by the principle of procedural abstraction. The functions are treated like black boxes.

Algorithm Design

The implementations of the functions `get_input` and `give_output` are straightforward. They simply consist of a few `cin` and `cout` statements. The algorithm for the function `price` is given by the following pseudocode:

```
if turnover ≤ 7 days then
    return (cost +5% of cost);
else
    return (cost +10% of cost);
```

Coding

There are three constants used in this program: a low markup figure of 5 percent, a high markup figure of 10 percent, and an expected shelf stay of 7 days as the threshold above which the high markup is used. Since these constants might need to be changed to update the program should the company decide to change its pricing policy, we declare global named constants at the start of our program for each of these three numbers. The declarations with the *const* modifier are the following:

```
const double LOW_MARKUP = 0.05; //5%
const double HIGH_MARKUP = 0.10; //10%
const int THRESHOLD = 7; //Use HIGH_MARKUP if do not
                         //expect to sell in 7 days or less
```

The body of the function `price` is a straightforward translation of our algorithm from pseudocode to C++ code:

```
{
    if (turnover <= THRESHOLD)
        return ( cost + (LOW_MARKUP * cost) );
    else
        return ( cost + (HIGH_MARKUP * cost) );
}
```

The complete program is shown in Display 5.9.

DISPLAY 5.9 Supermarket Pricing *(part 1 of 2)*

```
//Determines the retail price of an item according to
//the pricing policies of the Quick-Shop supermarket chain.
#include <iostream>

const double LOW_MARKUP = 0.05; //5%
const double HIGH_MARKUP = 0.10; //10%
const int THRESHOLD = 7;//Use HIGH_MARKUP if not expected
                        //to sell in 7 days or less.

void introduction();
//Postcondition: Description of program is written on the screen.

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//turnover is the expected number of days until sale of the item.
//Returns the retail price of the item.

void give_output(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item; turnover is the
//expected time until sale of the item; price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price have been
//written to the screen.

int main( )
{
    double wholesale_cost, retail_price;
    int shelf_time;
    introduction( );
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}

//Uses iostream:
void introduction( )
```

(continued)

DISPLAY 5.9 Supermarket Pricing *(part 2 of 2)*

```
{
    using namespace std;
    cout<< "This program determines the retail price for\n"
        << "an item at a Quick-Shop supermarket store.\n";
}

//Uses iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}

//Uses iostream:
void give_output(double cost, int turnover, double price)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Wholesale cost = $" << cost << endl
         << "Expected time until sold = "
         << turnover << " days" << endl
         << "Retail price = $" << price << endl;
}

//Uses defined constants LOW_MARKUP, HIGH_MARKUP, and THRESHOLD:
double price(double cost, int turnover)
{
    if (turnover <= THRESHOLD)
        return ( cost + (LOW_MARKUP * cost) );
    else
        return ( cost + (HIGH_MARKUP * cost) );

}
```

Sample Dialogue

```
This program determines the retail price for an item at a Quick-Shop
supermarket store. Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $1.27
```

Program Testing

An important technique in testing a program is to test all kinds of input. There is no precise definition of what we mean by a "kind" of input, but in practice, it is often easy to decide what kinds of input data a program deals with. In the case of our supermarket program, there are two main kinds of input: input that uses the low markup of 5 percent and input that uses the high markup of 10 percent. Thus, we should test at least one case in which the item is expected to remain on the shelf for less than 7 days and at least one case in which the item is expected to remain on the shelf for more than 7 days.

Test all kinds of input

Another testing strategy is to test boundary values. Unfortunately, boundary value is another vague concept. An input (test) value is a boundary value if it is a value at which the program changes behavior. For example, in our supermarket program, the program's behavior changes at an expected shelf stay of 7 days. Thus, 7 is a boundary value; the program behaves differently for a number of days that is less than or equal to 7 than it does for a number of days that is greater than 7. Hence, we should test the program on at least one case in which the item is expected to remain on the shelf for exactly 7 days. Normally, you should also test input that is one step away from the boundary value as well, since you can easily be off by one in deciding where the boundary is. Hence, we should test our program on input for an item that is expected to remain on the shelf for 6 days, an item that is expected to remain on the shelf for 7 days, and an item that is expected to remain on the shelf for 8 days. (This is in addition to the test inputs described in the previous paragraph, which should be well below and well above 7 days.)

Test boundary values

SELF-TEST EXERCISES

13. Can a function definition appear inside the body of another function definition?

14. Can a function definition contain a call to another function?

15. Rewrite the function declaration comment for the function order shown in Display 5.8 so that it is expressed in terms of preconditions and postconditions.

16. Give a precondition and a postcondition for the predefined function `sqrt`, which returns the square root of its argument.

5.4 TESTING AND DEBUGGING FUNCTIONS

"I beheld the wretch—the miserable monster whom I had created."

MARY WOLLSTONECRAFT SHELLEY, *Frankenstein*

Stubs and Drivers

Each function should be designed, coded, and tested as a separate unit from the rest of the program. This is the essence of the top-down design strategy. When you treat each function as a separate unit, you transform one big task into a series of smaller, more manageable tasks. But how do you test a function outside of the program for which it is intended? You write a special program to do the testing. For example, Display 5.10 shows a program to test the function `get_input`, which was used in the program in Display 5.9.

DISPLAY 5.10 Driver Program *(part 1 of 2)*

```
//Driver program for the function get_input.
#include <iostream>

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

int main( )
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
             << wholesale_cost << endl;
        cout << "Days until sold is now "
             << shelf_time << endl;

        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}
```

(continued)

DISPLAY 5.10 Driver Program *(part 2 of 2)*

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}
```

Sample Dialogue

```
Enter the wholesale cost of item: $123.45
Enter the expected number of days until sold: 67
Wholesale cost is now $123.45
Days until sold is now 67
Test again? (Type y for yes or n for no): y

Enter the wholesale cost of item: $9.05
Enter the expected number of days until sold: 3
Wholesale cost is now $9.05
Days until sold is now 3
Test again? (Type y for yes or n for no): n
```

Programs like this one are called **driver** programs. These driver programs are temporary tools and can be quite minimal. They need not have fancy input routines. They need not perform all the calculations the final program will perform. All they need do is obtain reasonable values for the function arguments in as simple a way as possible—typically from the user—then execute the function and show the result. A loop, as in the program shown in Display 5.10, will allow you to retest the function on different arguments without having to rerun the program.

If you test each function separately, you will find most of the mistakes in your program. Moreover, you will find out which functions contain the mistakes. If you were to test only the entire program, you would probably find out if there were a mistake but may have no idea where the mistake is. Even worse, you may think you know where the mistake is but be wrong.

Once you have fully tested a function, you can use it in the driver program for some other function. Each function should be tested in a program in which it is the only untested function. However, it's fine to use a fully tested function when testing some other function. If a bug is found, you know the bug is in the untested function. For example, after fully testing the function `get_input` with the driver program in Display 5.10, you can use `get_input` as the input routine in driver programs to test the remaining functions.

It is sometimes impossible or inconvenient to test a function without using some other function that has not yet been written or has not yet been tested. In this case, you can use a simplified version of the missing or untested function. These simplified functions are called **stubs.** These stubs will not necessarily perform the correct calculation, but they will deliver values that suffice for testing, and they are simple enough that you can have confidence in their performance. For example, the program in Display 5.11 is designed to test the function `give_output` from Display 5.9 as well as the basic layout of the program. This program uses the function `get_input`, which we already

DISPLAY 5.11 Program with a Stub *(part 1 of 2)*

```
//Determines the retail price of an item according to
//the pricing policies of the Quick-Shop supermarket chain.
#include <iostream>

void introduction( );
//Postcondition: Description of program is written on the screen.

void get_input(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: The value of cost has been set to the
//wholesale cost of one item. The value of turnover has been
//set to the expected number of days until the item is sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//turnover is the expected number of days until sale of the item.
//Returns the retail price of the item.

void give_output(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item; turnover is the
//expected time until sale of the item; price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price have been
//written to the screen.

int main( )
{
    double wholesale_cost, retail_price;
    int shelf_time;

    introduction( );
        get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}
```

(continued)

DISPLAY 5.11 Program with a Stub *(part 2 of 2)*

fully tested function

```
//Uses iostream:
void introduction( )
{
    using namespace std;
    cout << "This program determines the retail price for\n"
         << "an item at a Quick-Shop supermarket store.\n";
}
```

fully tested function

```
//Uses iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}
```

function being tested

```
//Uses iostream:
void give_output(double cost, int turnover, double price)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Wholesale cost = $" << cost << endl
         << "Expected time until sold = "
         << turnover << " days" << endl
         << "Retail price= $" << price << endl;
}
```

stub

```
//This is only a stub:
double price(double cost, int turnover)
{
    return 9.99; //Not correct, but good enough for some testing.
}
```

Sample Dialogue

```
This program determines the retail price for
an item at a Quick-Shop supermarket store.
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $9.99
```

fully tested using the driver program shown in Display 5.10. This program also includes the function `initialize_screen`, which we assume has been tested in a driver program of its own, even though we have not bothered to show that simple driver program. Since we have not yet tested the function `price`, we have used a stub to stand in for it. Notice that we could use this program before we have even written the function `price`. This way we can test the basic program layout before we fill in the details of all the function definitions.

Using a program outline with stubs allows you to test and then "flesh out" the basic program outline, rather than write a completely new program to test each function. For this reason, a program outline with stubs is usually the most efficient method of testing. A common approach is to use driver programs to test some basic functions, like the input and output functions, and then use a program with stubs to test the remaining functions. The stubs are replaced by functions one at a time: One stub is replaced by a complete function and tested; once that function is fully tested, another stub is replaced by a full function definition, and so forth until the final program is produced.

The Fundamental Rule for Testing Functions

Every function should be tested in a program in which every other function in that program has already been fully tested and debugged.

SELF-TEST EXERCISES

17. What is the fundamental rule for testing functions? Why is this a good way to test functions?

18. What is a driver program?

19. Write a driver program for the function introduction shown in Display 5.11.

20. Write a driver program for the function `add_tax` from Self-Test Exercise 11.

21. What is a stub?

22. Write a stub for the function whose function declaration is given next. Do not write a whole program, only the stub that would go in a program. (*Hint:* It will be very short.)

```
double rain_prob(double pressure, double humidity,
                 double temp);
//Precondition: pressure is the barometric
```

```
    //pressure in inches of mercury,
    //humidity is the relative humidity as a percent, and
    //temp is the temperature in degrees Fahrenheit.
    //Returns the probability of rain, which is a number
    //between 0 and 1.
    //0 means no chance of rain. 1 means rain is 100%
    //certain.
```

5.5 GENERAL DEBUGGING TECHNIQUES

VideoNote
Debugging

Careful testing through the use of stubs and drivers can detect a large number of bugs that may exist in a program. However, examination of the code and the output of test cases may be insufficient to track down many logic errors. In this case, there are a number of general debugging techniques that you may employ.

Keep an Open Mind

Examine the system as a whole and don't assume that the bug occurs in one particular place. If the program is giving incorrect output values, then you should examine the source code, different test cases for the input and output values, and the logic behind the algorithm itself. For example, consider the code to determine price for the supermarket example in Display 5.9. If the wrong price is displayed, the error might simply be that the input values were different from those you were expecting in the test case, leading to an apparently incorrect program.

Some novice programmers will "randomly" change portions of the code hoping that it will fix the error. Avoid this technique at all costs! Sometimes this approach will work for the first few simple programs that you write. However, it will almost certainly fail for larger programs and will often introduce new errors to the program. Make sure that you understand what logical impact a change to the code will make before committing the modification.

Finally, if allowed by your instructor, you could show the program to someone else. A fresh set of eyes can sometimes quickly pinpoint an error that you have been missing. Taking a break and returning to the problem a few hours later or the next day can also sometimes help in discovering an error.

Check Common Errors

One of the first mistakes you should look for are common errors that are easy to make, as described throughout the textbook in the Pitfall and Programming Tip sections. Examples of sources for common errors include (1) uninitialized variables, (2) off-by-one errors, (3) exceeding a data boundary, (4) automatic type conversion, and (5) using = instead of ==.

Localize the Error

Determining the precise cause and location of a bug is one of the first steps to fixing the error. Examining the input and output behavior for different test cases is one way to localize the error. A related technique is to add `cout` statements to strategic locations in the program that print out the values for critical variables. The `cout` statements also serve to show what code the program is executing. This is the strategy of tracing variables that was described in Chapter 3 for loops, but it can be used even when there are no loops present in the code.

For example, consider the code in Display 5.12 that is intended to convert a temperature from Fahrenheit to Celsius using the formula

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

When this program is executed with an input of 100 degrees Fahrenheit, the output is "Temperature in Celsius is 0". This is obviously incorrect, as the correct answer is 37.8 degrees Celsius.

To track down the error we can print out the value of critical variables. In this case, something appears to be wrong with the conversion formula, so we try a two-step approach. In the first step we compute (Fahrenheit – 32) and in the second step we compute (5 / 9) and then output both values. This

DISPLAY 5.12 Temperature Conversion Program with a Bug

```
#include <iostream>
using namespace std;

int main()
{
    double fahrenheit;
    double celsius;

    cout << "Enter temperature in Fahrenheit." << endl;
    cin >> fahrenheit;
    celsius = (5 / 9) * (fahrenheit - 32);
    cout << "Temperature in Celsius is " << celsius << endl;

    return 0;
}
```

Sample Dialogue

```
Enter temperature in Fahrenheit.
100
Temperature in Celsius is 0
```

is illustrated in Display 5.13. We have also commented out the original line of code by placing // at the beginning of the line. This tells the compiler to ignore the original line of code but still leave it in the program for our reference. If we ever wish to restore the code, we simply remove the // instead of having to type the line in again if it was deleted.

By examining the result of the cout statements we have now identified the precise location of the bug. In this case, the conversion factor is not computed correctly. Since we are setting the conversion factor to 5 / 9,

DISPLAY 5.13 Debugging with cout Statements

```
#include <iostream>
using namespace std;

int main()
{
    double fahrenheit;
    double celsius;

    cout << "Enter temperature in Fahrenheit." << endl;
    cin >> fahrenheit;

    // Comment out original line of code but leave it
    // in the program for our reference
    // celsius = (5 / 9) * (fahrenheit - 32);

    // Add cout statements to verify (5 / 9) and (fahrenheit - 32)
    // are computed correctly
    double conversionFactor = 5 / 9;
    double tempFahrenheit = (fahrenheit - 32);

    cout << "fahrenheit - 32 = " << tempFahrenheit << endl;
    cout << "conversionFactor = " << conversionFactor << endl;
    celsius = conversionFactor * tempFahrenheit;
    cout << "Temperature in Celsius is " << celsius << endl;

    return 0;
}
```

code that is commented out

debugging with cout statements

Sample Dialogue

```
Enter temperature in Fahrenheit.
100
fahrenheit - 32 = 68
conversionFactor = 0
Temperature in Celsius is 0
```

this instructs the compiler to compute the division of two integers, which results in zero. The simple fix is to perform floating-point division instead of integer division by changing one of the operands to a floating-point type, for example:

```
double conversionFactor = 5.0 / 9;
```

Once the bug has been identified we can now remove or comment out the debug code and return to a corrected version of the original program by modifying the line that computes the formula to the following:

```
celsius = (5.0 / 9) * (fahrenheit - 32);
```

Adding debugging code and introducing `cout` statements is a simple technique that works in almost any programming environment. However, it can sometimes be tedious to add a large number of `cout` statements to a program. Moreover, the output of the `cout` statements may be long or difficult to interpret, and the introduction of debugging code might even introduce new errors. Many compilers and integrated developing environments include a separate program, a **debugger,** that allows the programmer to stop execution of the program at a specific line of code called a breakpoint and step through the execution of the code one line at a time. As the debugger steps through the code, the programmer can inspect the contents of variables and even manually change the values stored in those variables. No `cout` statements are necessary to view the values of critical variables. The interface, commands, and capabilities of debuggers vary among C++ compilers, so check your user manual or check with your instructor for help on how to use these features.

The `assert` Macro

In Section 5.3 we discussed the concept of preconditions and postconditions for subroutines. The `assert` macro is a tool to ensure that the expected conditions are true at the location of the `assert` statement. If the condition is not met, then the program will display an error message and abort. To use `assert`, first include the definition of `assert` in your program with the following include statement:

```
#include <cassert>
```

To use `assert`, add the following line of code at the location where you would like to enforce the assertion with a boolean expression that should evaluate to true:

```
assert(boolean_expression);
```

The `assert` statement is a macro, which is a construct similar to a function. As an example, consider a subroutine that uses Newton's method to calculate the square root of a number *n*:

$$sqrt_{i+1} = \frac{1}{2}\left(sqrt_i + \frac{n}{sqrt_i}\right)$$

Here $\text{sqrt}_0 = 1$ and sqrt_i approaches the square root of n as i approaches infinity. A subroutine that implements this algorithm requires that n be a positive number and that the number of iterations we will repeat the calculation is also a positive number. We can guarantee this condition by adding `assert` to the subroutine as shown below:

```
// Approximates the square root of n using Newton's
// Iteration.
// Precondition: n is positive, num_iterations is positive
// Postcondition: returns the square root of n
double newton_sqroot(double n, int num_iterations)
{
    double answer = 1;
    int i = 0;

    assert((n > 0) && (num_iterations> 0));
    while (i <num_iterations)
    {
        answer = 0.5 * (answer + n / answer);
        i++;
    }
    return answer;
}
```

If we try to execute this subroutine with any negative parameters, then the program will abort and display the assertion that failed. The `assert` statement can be used in a similar manner for any assertion that you would like to enforce and is an excellent technique for defensive programming.

If you are going to distribute your program, you might not want the executable program to include the `assert` statements, since users could then get error messages that they might not understand. If you have added many `assert` statements to your code, it can be tedious to remove them all. Fortunately, you can disable all `assert` macros by adding the following line to the beginning of your program, before the `include` statement for `<cassert>` as follows:

```
#define NDEBUG
#include <cassert>
```

If you later change your program and need to debug it again, you can turn the `assert` statements back on by deleting the line `#define NDEBUG` (or commenting it out).

SELF-TEST EXERCISES

23. If computing the statement: `x = (x * y / z);` how can you use the `assert` macro to avoid division by zero?

24. What is a debugger?

25. What general techniques can you use to determine the source of an error?

CHAPTER SUMMARY

- All subtasks in a program can be implemented as functions, either as functions that return a value or as *void* functions.
- A **formal parameter** is a kind of place holder that is filled in with a function **argument** when the function is called. There are two methods of performing this substitution, call-by-value and call-by-reference.
- In the **call-by-value** substitution mechanism, the value of an argument is substituted for its corresponding formal parameter. In the **call-by-reference** substitution mechanism, the argument should be a variable and the entire variable is substituted for the corresponding argument.
- The way to indicate a call-by-reference parameter in a function definition is to attach the ampersand sign, &, to the type of the formal parameter.
- An argument corresponding to a call-by-value parameter cannot be changed by a function call. An argument corresponding to a call-by-reference parameter can be changed by a function call. If you want a function to change the value of a variable, then you must use a call-by-reference parameter.
- A good way to write a function declaration comment is to use a precondition and a postcondition. The **precondition** states what is assumed to be true when the function is called. The **postcondition** describes the effect of the function call; that is, the postcondition tells what will be true after the function is executed in a situation in which the precondition holds.
- Every function should be tested in a program in which every other function in that program has already been fully tested and debugged.
- A **driver program** is a program that does nothing but test a function.
- A simplified version of a function is called a **stub.** A stub is used in place of a function definition that has not yet been tested (or possibly not even written) so that the rest of the program can be tested.
- A debugger, strategic placement of `cout` statements, and the `assert` macro are tools that can help you debug a program.

Answers to Self-Test Exercises

1.
```
Hello
Goodbye
One more time:
Hello
End of program.
```

2. No, a *void* function definition need not contain a *return* statement. A *void* function definition may contain a *return* statement, but one is not required.

3. Omitting the *return* statement in the function definition for `initialize_screen` in Display 5.2 would have absolutely no effect on how the program behaves. The program will compile, run, and behave exactly the same. Similarly, omitting the *return* statement in the function definition for `show_results` also will have no effect on how the program behaves. However, if you omit the *return* statement in the function definition for `celsius`, that will be a serious error that will keep the program from running. The difference is that the functions `initialize_screen` and `show_results` are *void* functions, but `celsius` is not a *void* function.

4.
```
#include <iostream>

void product_out(int n1, int n2, int n3);
int main()
{
    using namespace std;
    int num1, num2, num3;
    cout << "Enter three integers: ";
    cin >> num1 >> num2 >> num3;
    product_out(num1, num2, num3);
    return 0;
}

void product_out(int n1, int n2, int n3)
{
    using namespace std;
    cout << "The product of the three numbers "
         << n1 << ", " << n2 << ", and "
         << n3 << " is " << (n1 * n2 * n3) << endl;
}
```

5. These answers are system dependent.

6. A call to a *void* function followed by a semicolon is a statement. A call to a function that returns a value is an expression.

7.
```
10 20 30
1 2 3
1 20 3
```

8.
```
Enter two integers: 5 10
In reverse order the numbers are: 5 5
```
different

9.
```
par1_value in function call = 111
par2_ref in function call = 222
n1 after function call = 1
n2 after function call = 2
```
different

10.
```
void zero_both(int& n1, int& n2)
{
    n1 = 0;
    n2 = 0;
}
```

11.
```
void add_tax(double tax_rate, double& cost)
{
    cost = cost + ( tax_rate/100.0 ) * cost;
}
```

The division by 100 is to convert a percent to a fraction. For example, 10% is 10/100.0 or 1/10th of the cost.

12. Yes, a function that returns a value can have a call-by-reference parameter. Yes, a function can have a combination of call-by-value and call-by-reference parameters.

13. No, a function definition cannot appear inside the body of another function definition.

14. Yes, a function definition can contain a call to another function.

15.
```
void order(int& n1, int& n2);
//Precondition: The variables n1 and n2 have values.
//Postcondition: The values in n1 and n2 have been
//ordered so that n1 <= n2.
```

16.
```
double sqrt(double n);
//Precondition: n >= 0.
//Returns the squareroot of n.
```

You can rewrite the second comment line to the following if you prefer, but the previous version is the usual form used for a function that returns a value:

```
//Postcondition: Returns the squareroot of n.
```

17. The fundamental rule for testing functions is that every function should be tested in a program in which every other function in that program has already been fully tested and debugged. This is a good way to test a function because if you follow this rule, then when you find a bug, you will know which function contains the bug.

18. A driver program is a program written for the sole purpose of testing a function.

19.
```
#include <iostream>

void introduction();
//Postcondition: Description of program is written on
//the screen.
int main()
{
    using namespace std;
    introduction();
    cout << "End of test.\n";
    return 0;
}
//Uses iostream:
void introduction()
{
    using namespace std;
    cout << "This program determines the retail price for\n"
         << "an item at a Quick-Shop supermarket store.\n";
}
```

20.
```
//Driver program for the function add_tax.
#include <iostream>

void add_tax(double tax_rate, double& cost);
//Precondition: tax_rate is the amount of sales tax as
//a percentage and cost is the cost of an item before
//tax.
//Postcondition: cost has been changed to the cost of
//the item after adding sales tax.

int main()
{
    using namespace std;
    double cost, tax_rate;
    char ans;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        cout << "Enter cost and tax rate:\n";
```

```
            cin >> cost >> tax_rate;
            add_tax(tax_rate, cost);

            cout << "After call to add_tax\n"
                 << "tax_rate is " << tax_rate << endl
                 << "cost is " << cost << endl;

            cout << "Test again?"
                 << " (Type y for yes or n for no): ";
            cin >> ans;
            cout << endl;
        } while (ans == 'y' || ans == 'Y');

        return 0;
    }

    void add_tax(double tax_rate, double& cost)
    {
        cost = cost + ( tax_rate/100.0 )* cost;
    }
```

21. A stub is a simplified version of a function that is used in place of the function so that other functions can be tested.

22.
```
//THIS IS JUST A STUB.
double rain_prob(double pressure, double humidity, double temp)
{
    return 0.25; //Not correct, but good enough for some testing.
}
```

23. `assert(z != 0).`

24. A debugger is a tool that allows the programmer to set breakpoints, step through the code line by line, and inspect or modify the value of variables.

25. Keeping an open mind, adding `cout` statements to narrow down the cause of the error, using a debugger, searching for common errors, and devising a variety of tests are a few techniques that you can use to debug a program.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write a function that computes the average and standard deviation of four scores. The standard deviation is defined to be the square root of the average of the four values: $(s_i - a)^2$, where a is average of the four scores s_1, s_2, s_3, and s_4. The function will have six parameters and will call two other

functions. Embed the function in a driver program that allows you to test the function again and again until you tell the program you are finished.

2. Write a program that reads in a length in feet and inches and outputs the equivalent length in meters and centimeters. Use at least three functions: one for input, one or more for calculating, and one for output. Include a loop that lets the user repeat this computation for new input values until the user says he or she wants to end the program. There are 0.3048 meters in a foot, 100 centimeters in a meter, and 12 inches in a foot.

3. Write a program like that of the previous exercise that converts from meters and centimeters into feet and inches. Use functions for the subtasks.

4. (You should do the previous two Practice Programs before doing this one.) Write a program that combines the functions in the previous two Practice Programs. The program asks the user if he or she wants to convert from feet and inches to meters and centimeters or from meters and centimeters to feet and inches. The program then performs the desired conversion. Have the user respond by typing the integer 1 for one type of conversion and 2 for the other conversion. The program reads the user's answer and then executes an *`if-else`* statement. Each branch of the *`if-else`* statement will be a function call. The two functions called in the *`if-else`* statement will have function definitions that are very similar to the programs for the previous two Practice Programs. Thus, they will be function definitions that call other functions in their function bodies. Include a loop that lets the user repeat this computation for new input values until the user says he or she wants to end the program.

5. Write a program that reads in a weight in pounds and ounces and outputs the equivalent weight in kilograms and grams. Use at least three functions: one for input, one or more for calculating, and one for output. Include a loop that lets the user repeat this computation for new input values until the user says he or she wants to end the program. There are 2.2046 pounds in a kilogram, 1000 grams in a kilogram, and 16 ounces in a pound.

VideoNote
Solution to Practice Program 5.5

6. Write a program like that of the previous exercise that converts from kilograms and grams into pounds and ounces. Use functions for the subtasks.

7. (You should do the previous two Practice Programs before doing this one.) Write a program that combines the functions of the previous two Practice Programs. The program asks the user if he or she wants to convert from pounds and ounces to kilograms and grams or from kilograms and grams to pounds and ounces. The program then performs the desired conversion. Have the user respond by typing the integer 1 for one type of conversion and 2 for the other. The program reads the user's answer and then executes an `if-else` statement. Each branch of the `if-else` statement

will be a function call. The two functions called in the `if-else` statement will have function definitions that are very similar to the programs for the previous two Practice Programs. Thus, they will be function definitions that call other functions in their function bodies. Include a loop that lets the user repeat this computation for new input values until the user says he or she wants to end the program.

8. (You need to do Practice Programs 4 and 7 before doing this one.) Write a program that combines the functions of Practice Programs 4 and 7. The program asks the user if he or she wants to convert lengths or weights. If the user chooses lengths, then the program asks the user if he or she wants to convert from feet and inches to meters and centimeters or from meters and centimeters to feet and inches. If the user chooses weights, a similar question about pounds, ounces, kilograms, and grams is asked. The program then performs the desired conversion. Have the user respond by typing the integer 1 for one type of conversion and 2 for the other. The program reads the user's answer and then executes an `if-else` statement. Each branch of the `if-else` statement will be a function call. The two functions called in the `if-else` statement will have function definitions that are very similar to the programs for Practice Programs 4 and 7. Thus, these functions will be function definitions that call other functions in their function bodies; however, they will be very easy to write by adapting the programs you wrote for Practice Programs 4 and 7.

   Notice that your program will have `if-else` statements embedded inside of `if-else` statements, but only in an indirect way. The outer `if-else` statement will include two function calls as its two branches. These two function calls will each in turn include an `if-else` statement, but you need not think about that. They are just function calls and the details are in a black box that you create when you define these functions. If you try to create a four-way branch, you are probably on the wrong track. You should only need to think about two-way branches (even though the entire program does ultimately branch into four cases). Include a loop that lets the user repeat this computation for new input values until the user says he or she wants to end the program.

9. The area of an arbitrary triangle can be computed using the formula

   $$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

   where a, b, and c are the lengths of the sides, and s is the semiperimeter.

   $$s = (a + b + c)/2$$

   Write a *`void`* function that computes the area and perimeter *(not the semiperimeter)* of a triangle based on the length of the sides. The function should use five parameters—three value parameters that provide the lengths of the edges and two reference parameters that store the computed area and perimeter. Make your function robust. Note that not all combinations of *a*, *b*, and *c* produce a triangle. Your function should produce correct results for legal data and reasonable results for illegal combinations.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Write a program that converts from 24-hour notation to 12-hour notation. For example, it should convert 14:25 to 2:25 PM. The input is given as two integers. There should be at least three functions, one for input, one to do the conversion, and one for output. Record the AM/PM information as a value of type char, 'A' for AM and 'P' for PM. Thus, the function for doing the conversions will have a call-by-reference formal parameter of type `char` to record whether it is AM or PM. (The function will have other parameters as well.) Include a loop that lets the user repeat this computation for new input values again and again until the user says he or she wants to end the program.

2. Write a program that requests the current time and a waiting time as two integers for the number of hours and the number of minutes to wait. The program then outputs what the time will be after the waiting period. Use 24-hour notation for the times. Include a loop that lets the user repeat this calculation for additional input values until the user says she or he wants to end the program.

3. Modify your program for Programming Project 2 so that it uses 12-hour notation, such as 3:45 PM.

4. Write a program that tells what coins to give out for any amount of change from 1 cent to 99 cents. For example, if the amount is 86 cents, the output would be something like the following:

   ```
   86 cents can be given as
   3 quarter(s) 1 dime(s) and 1 penny(pennies)
   ```

   Use coin denominations of 25 cents (quarters), 10 cents (dimes), and 1 cent (pennies). Do not use nickel and half-dollar coins. Your program will use the following function (among others):

   ```
   void compute_coins(int coin_value, int& num, int& amount_left);
   //Precondition: 0 < coin_value < 100; 0 <= amount_left < 100.
   //Postcondition: num has been set equal to the maximum number
   //of coins of denomination coin_value cents that can be obtained
   //from amount_left. Additionally, amount_left has been decreased
   //by the value of the coins, that is, decreased by
   //num * coin_value.
   ```

For example, suppose the value of the variable `amount_left` is `86`. Then, after the following call, the value of number will be `3` and the value of `amount_left` will be `11` (because if you take 3 quarters from 86 cents, that leaves 11 cents):

```
compute_coins(25, number, amount_left);
```

Include a loop that lets the user repeat this computation for new input values until the user says he or she wants to end the program. (*Hint:* Use integer division and the % operator to implement this function.)

5. In cold weather, meteorologists report an index called the windchill factor, that takes into account the wind speed and the temperature. The index provides a measure of the chilling effect of wind at a given air temperature. Windchill may be approximated by the formula:

$$W = 13.12 + 0.6215 * t - 11.37 * v^{0.16} + 0.3965 * t * v^{0.016}$$

where
v = wind speed in m/sec
t = temperature in degrees Celsius: $t <= 10$
W = windchill index (in degrees Celsius)

Write a function that returns the windchill index. Your code should ensure that the restriction on the temperature is not violated. Look up some weather reports in back issues of a newspaper in your university library and compare the windchill index you calculate with the result reported in the newspaper.

VideoNote
Solution to Programming Project 5.6

6. In the land of Puzzlevania, Aaron, Bob, and Charlie had an argument over which one of them was the greatest puzzler of all time. To end the argument once and for all, they agreed on a duel to the death. Aaron is a poor shooter and only hits his target with a probability of 1/3. Bob is a bit better and hits his target with a probability of 1/2. Charlie is an expert marksman and never misses. A hit means a kill and the person hit drops out of the duel.

To compensate for the inequities in their marksmanship skills, it is decided that the contestants would fire in turns starting with Aaron, followed by Bob, and then by Charlie. The cycle would repeat until there was one man standing. And that man would be remembered as the greatest puzzler of all time.

a. Write a function to simulate a single shot. It should use the following declaration:

```
void shoot(bool& targetAlive, double accuracy);
```

This would simulate someone shooting at `targetAlive` with the given accuracy by generating a random number between 0 and 1. If the random number is less than `accuracy`, then the target is hit and `targetAlive` should be set to false. Chapter 4 illustrates how to generate random numbers.

For example, if Bob is shooting at Charlie, this could be invoked as:

```
shoot(charlieAlive, 0.5);
```

Here, `charlieAlive` is a Boolean variable that indicates if Charlie is alive. Test your function using a driver program before moving on to step b.

b. An obvious strategy is for each man to shoot at the most accurate shooter still alive on the grounds that this shooter is the deadliest and has the best chance of hitting back. Write a second function named `startDuel` that uses the `shoot` function to simulate an entire duel using this strategy. It should loop until only one contestant is left, invoking the `shoot` function with the proper target and probability of hitting the target according to who is shooting. The function should return a variable that indicates who won the duel.

c. In your main function, invoke the `startDuel` function 1000 times in a loop, keeping track of how many times each contestant wins. Output the probability that each contestant will win when everyone uses the strategy of shooting at the most accurate shooter left alive.

d. A counterintuitive strategy is for Aaron to intentionally miss on his first shot. Thereafter, everyone uses the strategy of shooting at the most accurate shooter left alive. This strategy means that Aaron is guaranteed to live past the first round, since Bob and Charlie will fire at each other. Modify the program to accommodate this new strategy and output the probability of winning for each contestant.

7. Write a program that inputs a date (for example, July 4, 2008) and outputs the day of the week that corresponds to that date. The following algorithm is from http://en.wikipedia.org/wiki/Calculating_the_day_of_the_week. The implementation will require several functions.

```
bool isLeapYear(int year);
```

This function should return `true` if `year` is a leap year and `false` if it is not. Here is pseudocode to determine a leap year:

```
leap_year = (year divisible by 400) or (year divisible by 4 and
     year not divisible by 100))
int getCenturyValue(int year);
```

This function should take the first two digits of the year (that is, the century), divide by 4, and save the remainder. Subtract the remainder from 3 and return this value multiplied by 2. For example, the year 2008 becomes: (20/4) = 5 with a remainder of 0. 3 – 0 = 3. Return 3 * 2 = 6.

```
int getYearValue(int year);
```

This function computes a value based on the years since the beginning of the century. First, extract the last two digits of the year. For example, 08 is extracted for 2008. Next, factor in leap years. Divide the value from the previous step by 4 and discard the remainder. Add the two results together and return this value. For example, from 2008 we extract 08. Then (8/4) = 2 with a remainder of 0. Return 2 + 8 = 10.

```
int getMonthVa0lue(int month, int year);
```

This function should return a value based on the table below and will require invoking the `isLeapYear` function.

Month	**Return Value**
`January`	`0 (6 if year is a leap year)`
`February`	`3 (2 if year is a leap year)`
`March`	`3`
`April`	`6`
`May`	`1`
`June`	`4`
`July`	`6`
`August`	`2`
`September`	`5`
`October`	`0`
`November`	`3`
`December`	`5`

Finally, to compute the day of the week, compute the sum of the date's day plus the values returned by `getMonthValue, getYearValue`, and `getCenturyValue.` Divide the sum by 7 and compute the remainder. A remainder of 0 corresponds to Sunday, 1 corresponds to Monday, etc., up to 6, which corresponds to Saturday. For example, the date July 4, 2008 should be computed as (day of month) + (getMonthValue) + (getYearValue) + (getCenturyValue) = 4 + 6 + 10 + 6 = 26. 26/7 = 3 with a remainder of 5. The fifth day of the week corresponds to Friday.

Your program should allow the user to enter any date and output the corresponding day of the week in English.

This program should include a void function named getInput that prompts the user for the date and returns the month, day, and year using pass-by-reference parameters. You may choose to have the user enter the date's month as either a number (1–12) or a month name.

8. Complete the previous Programming Project and create a top-level function named dayOfWeek with the header:

   ```
   int dayOfWeek(int month, int day, int year);
   ```

   The function should encapsulate the necessary logic to return the day of the week of the specified date as an int (Sunday = 0, Monday = 1, etc.) You should add validation code to the function that tests if any of the inputs are invalid. If so, the function should return –1 as the day of the week. In your main function write a test driver that checks if dayOfWeek is returning the correct values. Your set of test cases should include at least two cases with invalid inputs.