Display 4.1 **Formal Parameter Used as a Local Variable** (part 2 of 2)

Sample Dialogue

```
Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5 46
For 5 hours and 46 minutes, your bill is $3450.00
```

Call-by-value parameters are local variables just like the variables you declare within the body of a function. However, you should not add a variable declaration for the formal parameters. Listing the formal parameter `minutesWorked` in the function heading also serves as the variable declaration. The following is the *wrong way* to start the function definition for `fee` because it declares `minutesWorked` twice:

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;
    int minutesWorked;
        ...
```

Do not do this when **`minutesWorked`** *is a parameter!*

MyProgrammingLab™

Self-Test Exercises

1. Carefully describe the call-by-value parameter mechanism.

2. The following function is supposed to take as arguments a length expressed in feet and inches and to return the total number of inches in that many feet and inches. For example, `totalInches(1, 2)` is supposed to return `14`, because 1 foot and 2 inches is the same as 14 inches. Will the following function perform correctly? If not, why not?

```
double totalInches(int feet, int inches)
{
    inches = 12*feet + inches;
    return inches;
}
```

A First Look at Call-by-Reference Parameters

The call-by-value mechanism that we used until now is not sufficient for all tasks you might want a function to perform. For example, one common task for a function is to obtain an input value from the user and set the value of an argument variable to this input value. With the call-by-value formal parameters that we have used until now, a corresponding 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, you 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, the corresponding argument in a function call must be a variable, and this argument variable is substituted for the formal parameter. It is almost 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 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. This is done in both the function declaration (function prototype) and the header of the function definition. For example, the following function definition has one formal parameter, receiver, which is a call-by-reference parameter:

ampersand, &

```
void getInput(double& receiver)
{
    cout << "Enter input number:\n";
    cin >> receiver;
}
```

In a program that contains this function definition, the following function call will set the double variable inputNumber equal to a value read from the keyboard:

```
getInput(inputNumber);
```

C++ allows you to place the ampersand either with the type name or with the parameter name, so you will sometimes see

```
void getInput(double &receiver);
```

which is equivalent to

```
void getInput(double& receiver);
```

Display 4.2 demonstrates call-by-reference parameters. The program reads in two numbers and writes the same numbers out, but in the reverse order.

The parameters in the functions getNumbers and swapValues are call-by-reference parameters. The input is performed by the function call

```
getNumbers(firstNum, secondNum);
```

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

```
swapValues(firstNum, secondNum);
```

Display 4.2 **Call-by-Reference Parameters**

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

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

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

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

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}

void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

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

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

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Sample Dialogue

```
Enter two integers: 5 6
In reverse order the numbers are: 6 5
```

The next few subsections describe the call-by-reference mechanism in more detail and also explain the particular functions used in Display 4.2.

Call-by-Reference Parameters

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

```
void getData(int& firstInput, double& secondInput);
```

Call-by-Reference Mechanism 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.

address

Program variables are implemented as memory locations. Each memory location has a unique **address** that is a number. The compiler assigns one memory location to each variable. For example, when the program in Display 4.2 is compiled, the variable `firstNum` might be assigned location `1010`, and the variable `secondNum` might be assigned `1012`. For all practical purposes, these memory locations are the variables.

For example, consider the following function declaration from Display 4.2:

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

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

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

```
getNumbers(firstNum, secondNum);
```

When the function call is executed, the function is not given the argument names `firstNum` and `secondNum`. Instead, it is given a list of the memory locations associated with each name. In this example, the list consists of the locations

```
1010
1012
```

which are the locations assigned to the argument variables `firstNum` and `secondNum`, *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. Diagrammatically, in this case, the correspondence is

```
firstNum  ⟶ 1010 ⟶ input1
secondNum ⟶ 1012 ⟶ input2
```

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 `getNumbers` 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 the variable `firstNum`). Similarly, the instructions in the body of the function `getNumbers` say that another 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 the variable `secondNum`). Thus, whatever the function instructs the computer to do to `input1` and `input2` is actually done to the variables `firstNum` and `secondNum`.

It may seem that there is an extra level of detail, or at least an extra level of verbiage. If `firstNum` is the variable with memory location `1010`, why do we insist on saying "the variable at memory location `1010`" instead of simply saying "`firstNum`?" This extra level of detail is needed if the arguments and formal parameters contain some confusing coincidence of names. For example, the function `getNumbers` has formal parameters named `input1` and `input2`. Suppose you want to change the program in Display 4.2 so that it uses the function `getNumbers` 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 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, input2;
getNumbers(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` function 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`.

Constant Reference Parameters

We place this subsection here for reference value. If you are reading this book in order, you may as well skip this section. The topic is explained in more detail later in the book.

If you place a `const` before a call-by-reference parameter's type, you get a call-by-reference parameter that cannot be changed. For the types we have seen so far, this has no advantages. However, it will turn out to be an aid to efficiency with array and class type parameters. We will discuss these constant parameters when we discuss arrays and when we discuss classes.

EXAMPLE: The `swapValues` Function

The function `swapValues` defined in Display 4.2 interchanges the values stored in two variables. The description of the function is given by the following function declaration and accompanying comment:

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

To see how the function is supposed to work, assume that the variable `firstNum` has the value `5` and the variable `secondNum` has the value `6` and consider the following function call:

```
swapValues(firstNum, secondNum);
```

After this function call, the value of `firstNum` will be `6` and the value of `secondNum` will be `5`.

As shown in Display 4.2, the definition of the function `swapValues` 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 swapValues(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

```
swapValues(firstNum, secondNum);
```

(continued)

EXAMPLE: (continued)

The variables `firstNum` and `secondNum` would be substituted for the formal parameters `variable1` and `variable2` so that with this incorrect function definition, the function call would be equivalent to the following:

```
firstNum = secondNum;
secondNum = firstNum;
```

This code does not produce the desired result. The value of `firstNum` is set equal to the value of `secondNum`, just as it should be. But then, the value of `secondNum` is set equal to the changed value of `firstNum`, which is now the original value of `secondNum`. Thus, the value of `secondNum` is not changed at all. (If this is unclear, go through the steps with specific values for the variables `firstNum` and `secondNum`.) What the function needs to do is save the original value of `firstNum` 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 4.2. When that correct version is used and the function is called with the arguments `firstNum` and `secondNum`, the function call is equivalent to the following code, which works correctly:

```
temp = firstNum;
firstNum = secondNum;
secondNum = temp;
```

TIP: Think of Actions, Not Code

Although we can explain how a function call works in terms of substituting code for the function call, that is not the way you should normally think about a function call. You should instead think of a function call as an action. For example, consider the function `swapValues` in Display 4.2 and an invocation such as

```
swapValues(firstNum, secondNum);
```

It is easier and clearer to think of this function call as the action of swapping the values of its two arguments. It is much less clear to think of it as the code

```
temp = firstNum;
firstNum = secondNum;
secondNum = temp;
```

■

MyProgrammingLab™

Self-Test Exercises

3. What is the output of the following program?

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