Passing Values

C++ passes variables between functions using two different methods. The one you use depends on how you want the passed variables to be changed. This chapter explores these two methods. The concepts discussed here are not new to the C++ language. Other programming languages, such as Pascal, FORTRAN, and QBasic, pass parameters using similar techniques. A computer language must have the capability to pass information between functions before it can be called truly structured.

This chapter introduces you to the following:

- Passing variables by value
- Passing arrays by address
- Passing nonarrays by address

Pay close attention because most of the programs in the remainder of the book rely on the methods described in this chapter.

Passing by Value (by Copy)

The two wordings "passing by value" and "passing by copy" mean the same thing in computer terms. Some textbooks and C++ programmers state that arguments are passed *by value,* and some state that they are passed *by copy*. Both of these phrases describe one

of the two methods by which arguments are passed to receiving functions. (The other method is called "by address," or "by reference." This method is covered later in the chapter.)

When you pass by value, a copy of the variable's value is passed to the receiving function.

When an argument (local variable) is passed by value, a copy of the variable's value is sent to—and is assigned to—the receiving function's parameter. If more than one variable is passed by value, a copy of each of their values is sent to—and is assigned to—the receiving function's parameters.

Figure 18.1 shows the *passing by copy* in action. The value of `i`—not the variable—is passed to the called function, which receives it as a variable `i`. There are two variables called `i`, not one. The first is local to `main()`, and the second is local to `pr_it()`. They both have the same names, but because they are local to their respective functions, there is no conflict. The variable does not have to be called `i` in both functions, and because the value of `i` is sent to the receiving function, it does not matter what the receiving function calls the variable that receives this value.

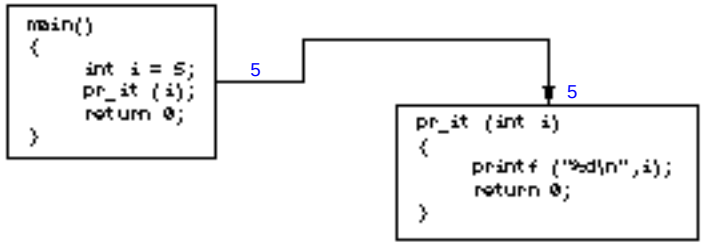


Figure 18.1. Passing the variable `i` by value.

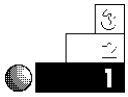
In this case, when passing and receiving variables between functions, it is wisest to retain the same names. Even though they are not the same variables, they hold the same value. In this example, the value `5` is passed from `main()`'s `i` to `pr_it()`'s `i`.

Because a copy of `i`'s value (and not the variable itself) is passed to the receiving function, if `pr_it()` changed `i`, it would be changing only its copy of `i` and not `main()`'s `i`. This fact truly separates functions and variables. You now have the technique for passing a copy of a variable to a receiving function, with the receiving function being unable to modify the calling function's variable.

All C++'s nonarray variables you have seen so far are passed by value. You do not have to do anything special to pass variables by value, except to pass them in the calling function's argument list and receive them in the receiving function's parameter list.

NOTE: The default method for passing parameters is by value, as just described, unless you pass arrays. Arrays are always passed by the other method, by address, described later in the chapter.

Examples



1. The following program asks users for their weight. It then passes that weight to a function that calculates the equivalent weight on the moon. Notice the second function uses the passed value, and calculates with it. After `weight` is passed to the second function, that function can treat `weight` as though it were a local variable.

Identify the program and include the necessary input/output file.

You want to calculate the user's weight on the moon. Because you have to hold the user's weight somewhere, declare the variable `weight` *as an integer. You also need a function that does the calculations, so create a function called* `moon()`.

Ask the user how much he or she weighs. Put the user's answer in `weight`. *Now pass the user's weight to the* `moon()` *function, which divides the weight by six to determine the equivalent weight on the moon. Display the user's weight on the moon.*

You have finished, so leave the `moon()` *function, then leave the* `main()` *function.*

```
// Filename: C18PASS1.CPP
// Calculate the user's weight in a second function.
#include <iostream.h>
moon(int weight);  // Prototypes discussed later.
```

```
main()
{
   int weight;                    // main()'s local weight.
   cout << "How many pounds do you weigh? ";
   cin >> weight;

   moon(weight);          // Call the moon() function and
                          //   pass it the weight.
   return 0;              // Return to the operating system.
}

moon(int weight)           // Declare the passed parameter.
{
   // Moon weights are 1/6th earth's weights
   weight /= 6;                  // Divide the weight by six.

   cout << "You weigh only " << weight <<
           " pounds on the moon!";
   return 0;                               // Return to main().
}
```

The output of this program follows:

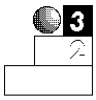
```
How many pounds do you weigh? 120
You weigh only 20 pounds on the moon!
```

2. You can rename passed variables in the receiving function. They are distinct from the passing function's variable. The following is the same program as in Example 1, except the receiving function calls the passed variable `earth_weight`. A new variable, called `moon_weight`, is local to the called function and is used for the moon's equivalent weight.


```
// Filename: C18PASS2.CPP
// Calculate the user's weight in a second function.
#include <iostream.h>
moon(int earth_weight);

main()
```

```
{
   int weight;                    // main()'s local weight.
   cout << "How many pounds do you weigh? ";
   cin >> weight;

   moon(weight);          // Call the moon() function and
                          // pass it the weight.
   return 0;              // Return to the operating system.
}

moon(int earth_weight)    // Declare the passed parameter.
{
   int moon_weight;               // Local to this function.

   // Moon's weights are 1/6th of earth's weights.
   moon_weight = earth_weight / 6;  // Divide weight by six.

   cout << "You only weigh " << moon_weight <<
           " pounds on the moon!";
   return 0;                                // Return to main().
}
```

The resulting output is identical to that of the previous program. Renaming the passed variable changes nothing.



3. The next example passes three variables—of three different types—to the called function. In the receiving function's parameter list, each of these variable types must be declared.

   This program prompts users for three values in the `main()` function. The `main()` function then passes these variables to the receiving function, which calculates and prints values related to those passed variables. When the called function modifies a variable passed to the function, notice again that this does not affect the calling function's variable. When variables are passed by value, the value—not the variable—is passed.

```
// Filename: C18PASS3.CPP
// Get grade information for a student.
#include <iostream.h>
#include <iomanip.h>
check_grade(char lgrade, float average, int tests);
```

```
main()
{
   char lgrade;   // Letter grade.
   int  tests;    // Number of tests not yet taken.
   float average; // Student's average based on 4.0 scale.

   cout << "What letter grade do you want? ";
   cin >> lgrade;
   cout << "What is your current test average? ";
   cin >> average;
   cout << "How many tests do you have left? ";
   cin >> tests;

   check_grade(lgrade, average, tests);  // Calls function
                  //  and passes three variables by value.
   return 0;
}

check_grade(char lgrade, float average, int tests)
{
   switch (tests)
   {
     case (0): { cout << "You will get your current grade "
                      << "of " << lgrade;
                 break; }
     case (1): { cout << "You still have time to bring " <<
                         "up your average";
                 cout << "of " << setprecision(1) <<
                         average << "up. Study hard!";
                 break; }
     default:  { cout << "Relax. You still have plenty of "
                      <<  "time.";
                 break; }
   }
   return 0;
}
```

Passing by Address (by Reference)

When you pass by address, the address of the variable is passed to the receiving function.

The two phrases "by address" and "by reference" mean the same thing. The previous section described passing arguments by value (or by copy). This section teaches you how to pass arguments by address.

When you pass an argument (local variable) *by address*, the variable's address is sent to—and is assigned to—the receiving function's parameter. (If you pass more than one variable by address, each of their addresses is sent to—and is assigned to—the receiving function's parameters.)

Variable Addresses

All variables in memory (RAM) are stored at memory addresses—see Figure 18.2. If you want more information on the internal representation of memory, refer to Appendix A, "Memory Addressing, Binary, and Hexadecimal Review."

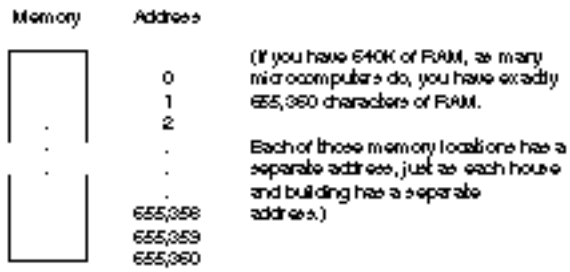


Figure 18.2. Memory addresses.

When you tell C++ to define a variable (such as `int i;`), you are requesting C++ to find an unused place in memory and assign that place (or memory address) to `i`. When your program uses the variable called `i`, C++ goes to `i`'s address and uses whatever is there.

If you define five variables as follows,

```
int i;
float x=9.8;
char ara[2] = {'A', 'B'};
int j=8, k=3;
```

C++ might arbitrarily place them in memory at the addresses shown in Figure 18.3.

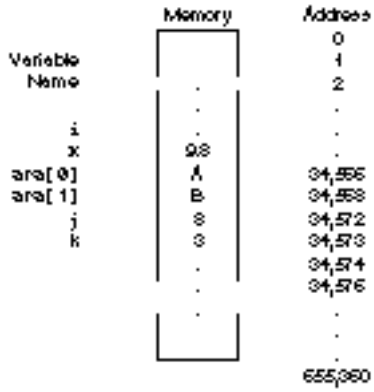


Figure 18.3. Storing variables in memory.

You don't know what is contained in the variable called `i` because you haven't put anything in it yet. Before you use `i`, you should initialize it with a value. (All variables—except character variables—usually use more than 1 byte of memory.)

Sample Program

All C++ arrays are passed by address.

The address of the variable, not its value, is copied to the receiving function when you pass a variable by address. In C++, *all arrays are automatically passed by address.* (Actually, a copy of their address is passed, but you will understand this better when you learn more about arrays and pointers.) The following important rule holds true for programs that pass by address:

Every time you pass a variable by address, if the receiving function changes the variable, it is changed also in the calling function.

Therefore, if you pass an array to a function and the function changes the array, those changes are still with the array when it returns to the calling function. Unlike passing by value, passing by address gives you the ability to change a variable in the *called* function and to keep those changes in effect in the *calling* function. The following sample program helps to illustrate this concept.

```
// Filename: C18ADD1.CPP
// Passing by address example.
#include <iostream.h>
#include <string.h>
change_it(char c[4]);   // Prototype discussed later.
main()
{
   char name[4]="ABC";

   change_it(name);          // Passes by address because
                             // it is an array.
   cout << name << "\n";     // Called function can
                             // change array.
   return 0;
}

change_it(char c[4])         // You must tell the function
                             // that c is an array.
{
   cout << c << "\n";        // Print as it is passed.
   strcpy(c, "USA");         // Change the array, both
                             // here and in main().
   return 0;
}
```

Here is the output from this program:

```
ABC
USA
```

At this point, you should have no trouble understanding that the array is passed from `main()` to the function called `change_it()`. Even though `change_it()` calls the array `c`, it refers to the same array passed by the `main()` function (`name`).

Figure 18.4 shows how the array is passed. Although the address of the array—and not its value—is passed from `name` to `c`, both arrays are the same.

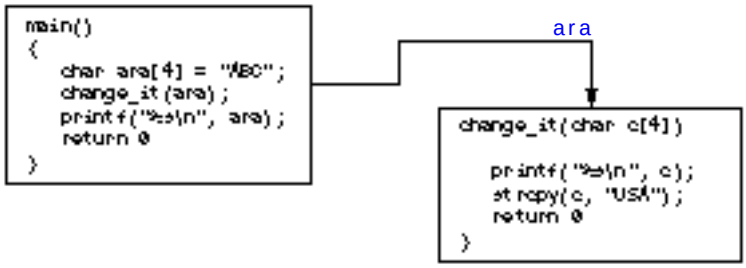


Figure 18.4. Passing an array by address.

Before going any further, a few additional comments are in order. Because the address of `name` is passed to the function—even though the array is called `c` in the receiving function—it is still the same array as `name`. Figure 18.5 shows how C++ accomplishes this task at the memory-address level.

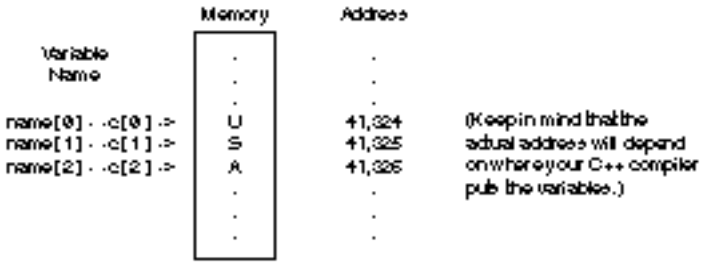
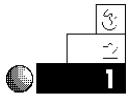


Figure 18.5. The array being passed is the same array in both functions.

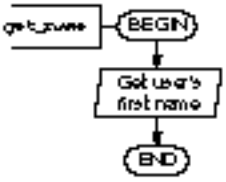
The variable array is referred to as `name` in `main()` and as `c` in `change_it()`. Because the address of `name` is copied to the receiving function, the variable is changed no matter what it is called in either

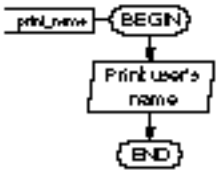
function. Because `change_it()` changes the array, the array is changed also in `main()`.

Examples



1. You can now use a function to fill an array with user input. The following function asks users for their first name in the function called `get_name()`. As users type the name in the array, it is also entered in `main()`'s array. The `main()` function then passes the array to `pr_name()`, where it is printed. (If arrays were passed by value, this program would not work. Only the array value would be passed to the called functions.)






```
// Filename: C18ADD2.CPP
// Get a name in an array, then print it using
// separate functions.
#include <iostream.h>
get_name(char name[25]);   // Prototypes discussed later.
print_name(char name[25]);

main()
{
   char name[25];
   get_name(name);           // Get the user's name.
   print_name(name);         // Print the user's name.
   return 0;
}

get_name(char name[25])      // Pass the array by address.
{
   cout << "What is your first name? ";
   cin >> name;
   return 0;
}

print_name(char name[25])
{
   cout << "\n\n Here you are, " << name;
   return 0;
}
```

When you pass an array, be sure to specify the array's type in the receiving function's parameter list. If the previous program declared the passed array with

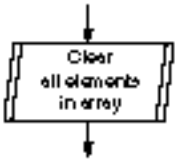
```
get_name(char name)
```

the function `get_name()` would interpret this as a single character variable, *not* a character array. You never have to put the array size in brackets. The following statement also works as the first line of `get_name()`.

```
get_name(char name[])
```

Most C++ programmers put the array size in the brackets to clarify the array size, even though the size is not needed.

2. Many programmers pass character arrays to functions to erase them. Here is a function called `clear_it()`. It expects two parameters: a character array and the total number of elements declared for that array. The array is passed by address (as are all arrays) and the number of elements, `num_els`, is passed by value (as are all nonarrays). When the function finishes, the array is cleared (all its elements are reset to null zero). Subsequent functions that use it can then have an empty array.




```
clear_it(char ara[10], int num_els)
{
   int ctr;
   for (ctr=0; ctr<num_els; ctr++)
     { ara[ctr] = '\0'; }
   return 0;
}
```

The brackets after `ara` do not have to contain a number, as described in the previous example. The `10` in this example is simply a placeholder for the brackets. Any value (or no value) would work as well.

Passing Nonarrays by Address

You can pass nonarrays by address as well.

You now should see the difference between passing variables by address and by value. Arrays can be passed by address, and nonarrays can be passed by value. You can override the *by value* default for nonarrays. This is helpful sometimes, but it is not always recommended because the called function can damage values in the called function.

If you want a nonarray variable changed in a receiving function and also want the changes kept in the calling function, you must override the default and pass the variable by address. (You should understand this section better after you learn how arrays and pointers relate.) To pass a nonarray by address, you must precede the argument in the receiving function with an ampersand (`&`).

This might sound strange to you (and it is, at this point). Few C++ programmers override the default of passing by address. When you learn about pointers later, you should have little need to do so. Most C++ programmers don't like to clutter their code with these extra ampersands, but it's nice to know you can override the default if necessary.

The following examples demonstrate how to pass nonarray variables by address.

Examples

1. The following program passes a variable by address from `main()` to a function. The function changes it and returns to `main()`. Because the variable is passed by address, `main()` recognizes the new value.

```
// Filename: C18ADD3.CPP
// Demonstrate passing nonarrays by address.
#include <iostream.h>
do_fun(int &amt);   // Prototypes discussed later.

main()
{
   int amt;
```

```
   amt = 100;                  // Assign a value in main().
   cout << "In main(), amt is " << amt << "\n";

   do_fun(amt);          // Pass amt by address
   cout << "After return, amt is " << amt << " in main()\n";
   return 0;
}

do_fun(int &amt)               // Inform function of
                               // passing by address.
{
   amt = 85;                   // Assign new value to amt.
   cout << "In do_fun(), amt is " << amt << "\n";
   return 0;
}
```

The output from this program follows:

```
In main(), amt is 100
In do_fun(), amt is 85
After return, amt is 85 in main()
```

Notice that `amt` changed in the called function. Because it was passed by address, it is changed also in the calling function.

2. You can use a function to get the user's keyboard values. The `main()` function recognizes those values as long as you pass them by address. The following program calculates the cubic feet in a swimming pool. In one function, it requests the width, length, and depth. In another function, it calculates the cubic feet of water. Finally, in a third function, it prints the answer. The `main()` function is clearly a controlling function, passing variables between these functions by address.


```
// Filename: C18POOL.CPP
// Calculates the cubic feet in a swimming pool.
#include <iostream.h>
get_values(int &length, int &width, int &depth);
calc_cubic(int &length, int &width, int &depth, int &cubic);
print_cubic(int &cubic);
```

```
main()
{
   int length, width, depth, cubic;

   get_values(length, width, depth);
   calc_cubic(length, width, depth, cubic);
   print_cubic(cubic);

   return 0;
}

get_values(int &length, int &width, int &depth)
{
   cout << "What is the pool's length? ";
   cin >> length;
   cout << "What is the pool's width? ";
   cin >> width;
   cout << "What is the pool's average depth? ";
   cin >> depth;
   return 0;
}

calc_cubic(int &length, int &width, int &depth, int &cubic)
{
   cubic = (length) * (width) * (depth);
   return 0;
}

print_cubic(int &cubic)
{
   cout << "\nThe pool has " << cubic << " cubic feet\n";
   return 0;
}
```

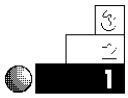
The output follows:

```
What is the pool's length? 16
What is the pool's width? 32
What is the pool's average depth? 6
The pool has 3072 cubic feet
```

All variables in a function must be preceded with an ampersand if they are to be passed by address.

Review Questions

The answers to the review questions are in Appendix B.



1. What type of variable is automatically passed by address?
2. What type of variable is automatically passed by value?
3. True or false: If a variable is passed by value, it is passed also by copy.

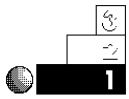
4. If a variable is passed to a function by value and the function changes the variable, is it changed in the calling function?
5. If a variable is passed to a function by address and the function changes the variable, is it changed in the calling function?

6. What is wrong with the following function?

```
do_fun(x, y, z)
{
   cout << "The variables are " << x << y << z;
   return 0;
}
```

7. Suppose you pass a nonarray variable and an array to a function at the same time. What is the default?
   a. Both are passed by address.
   b. Both are passed by value.
   c. One is passed by address and the other is passed by value.

Review Exercises



1. Write a `main()` function and a second function that `main()` calls. Ask users for their annual income in `main()`. Pass the income to the second function and print a congratulatory message if the user makes more than $50,000 or an encouragement message if the user makes less.

2. Write a three-function program, consisting of the following functions:

```
main()
fun1()
fun2()
```

   Declare a 10-element character array in `main()`, fill it with the letters `A` through `J` in `fun1()`, then print that array backwards in `fun2()`.

3. Write a program whose `main()` function passes a number to a function called `print_aster()`. The `print_aster()` function prints that many asterisks on a line, across the screen. If `print_aster()` is passed a number greater than 80, display an error because most screens cannot print more than 80 characters on the same line. When execution is finished, return control to `main()` and then return to the operating system.

4. Write a function that is passed two integer values by address. The function should declare a third local variable. Use the third variable as an intermediate variable and swap the values of both passed integers. For example, suppose the calling function passes your function `old_pay` and `new_pay` as in

```
swap_it(old_pay, new_pay);
```

   The `swap_it()` function reverses the two values so, when control returns to the calling function, the values of `old_pay` and `new_pay` are swapped.

Summary

You now have a complete understanding of the various methods for passing data to functions. Because you will be using local variables as much as possible, you have to know how to pass local variables between functions but also keep the variables away from functions that don't need them.

You can pass data in two ways: by value and by address. When you pass data by value, which is the default method for nonarrays, only a copy of the variable's contents are passed. If the called function modifies its parameters, those variables are not modified in the calling function. When you pass data by address, as is done with arrays and nonarray variables preceded by an ampersand, the receiving function can change the data in both functions.

Whenever you pass values, you must ensure that they match in number and type. If you don't match them, you could have problems. For example, suppose you pass an array and a floating-point variable, but in the receiving function, you receive a floating-point variable followed by an array. The data does not reach the receiving function properly because the parameter data types do not match the variables being passed. Chapter 19, "Function Return Values and Prototypes," shows you how to protect against such disasters by prototyping all your functions.