Variable Scope

The concept of *variable scope* is most important when you write functions. Variable scope determines which functions recognize certain variables. If a function recognizes a variable, the variable is *visible* to that function. Variable scope protects variables in one function from other functions that might overwrite them. If a function doesn't need access to a variable, that function shouldn't be able to see or change the variable. In other words, the variable should not be "visible" to that particular function.

This chapter introduces you to

- Global and local variables
- Passing arguments
- Automatic and static variables
- Passing parameters

The previous chapter introduced the concept of using a different function for each task. This concept is much more useful when you learn about local and global variable scope.

Global Versus Local Variables

If you have programmed only in BASIC, the concept of local and global variables might be new to you. In many interpreted versions of BASIC, all variables are *global*, meaning the entire program knows each variable and has the capability to change any of them. If you use a variable called SALES at the top of the program, even the last line in the program can use SALES. (If you don't know BASIC, don't despair—there will be one less habit you have to break!)

Global variables can be dangerous. Parts of a program can inadvertently change a variable that shouldn't be changed. For example, suppose you are writing a program that keeps track of a grocery store's inventory. You might keep track of sales percentages, discounts, retail prices, wholesale prices, produce prices, dairy prices, delivered prices, price changes, sales tax percentages, holiday markups, post-holiday markdowns, and so on.

The huge number of prices in such a system is confusing. When writing a program to keep track of every price, it would be easy to mistakenly call both the dairy prices `d_prices` and the delivered prices `d_prices`. Either C++ will not enable you to do this (you can't define the same variable twice) or you will overwrite a value used for something else. Whatever happens, keeping track of all these different—but similarly named—prices makes this program confusing to write.

Global variables are visible across many program functions.

Global variables can be dangerous because code can inadvertently overwrite a variable initialized elsewhere in the program. It is better to make every variable *local* in your programs. Then, only functions that should be able to change the variables can do so.

Local variables are visible only in the block where they are defined.

Local variables can be seen (and changed) only from the function in which they are defined. Therefore, if a function defines a variable as local, that variable's scope is protected. The variable cannot be used, changed, or erased by any other function without special programming that you learn about shortly.

If you use only one function, `main()`, the concept of local and global is academic. You know from Chapter 16, "Writing C++ Functions," however, that single-function programs are not recommended. It is best to write modular, structured programs made up

of many smaller functions. Therefore, you should know how to define variables as local to only those functions that use them.

Defining Variable Scope

When you first learned about variables in Chapter 4, "Variables and Literals," you learned you can define variables in two places:

- Before they are used inside a function
- Before a function name, such as `main()`

All examples in this book have declared variables with the first method. You have yet to see an example of the second method. Because most these programs have consisted entirely of a single `main()` function, there has been no reason to differentiate the two methods. It is only after you start using several functions in one program that these two variable definition methods become critical.

The following rules, specific to local and global variables, are important:

- A variable is local *if and only if* you define it after the opening brace of a block, usually at the top of a function.
- A variable is global *if and only if* you define it outside a function.

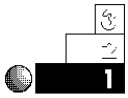
All variables you have seen so far have been local. They have all been defined immediately after the opening braces of `main()`. Therefore, they have been local to `main()`, and only `main()` can use them. Other functions have no idea these variables even exist because they belong to `main()` only. When the function (or block) ends, all its local variables are destroyed.

TIP: All local variables disappear (lose their definition) when their block ends.

Global variables are visible from their definition through the remainder of the program.

Global variables are visible ("known") from their point of definition to the end of the program. If you define a global variable, *any* line throughout the rest of the program—no matter how many functions and code lines follow it—is able to use that global variable.

Examples



1. The following section of code defines two local variables, `i` and `j`.

```
main()
{
   int i, j;                    // Local because they're
                                // defined after the brace.
   // Rest of main() goes here.
}
```

   These variables are visible to `main()` and not to any other function that might follow or be called by `main()`.

2. The following section of code defines two global variables, `g` and `h`.

```
#include <iostream.h>
int g, h;                    // Global because they're
                             // defined before a function.
main()
{
   // main()'s code goes here.
}
```

   It doesn't matter whether your `#include` lines go before or after global variable declarations.

3. Global variables can appear before any function. In the following program, `main()` uses no variables. However, both of the two functions after `main()` can use `sales` and `profit` because these variables are global.

```
// Filename: C17GLO.CPP
// Program that contains two global variables.
#include <iostream.h>
do_fun();
third_fun();  // Prototype discussed later.
main()
{
   cout << "No variables defined in main() \n\n";
   do_fun();                 // Call the first function.
```

```
   return 0;
}

float sales, profit;              // Two global variables.
do_fun()
{
   sales = 20000.00;         // This variable is visible
                             // from this point down.
   profit = 5000.00;         // As is this one. They are
                             // both global.

   cout << "The sales in the second function are " <<
           sales << "\n";
   cout << "The profit in the second function is " <<
           profit << "\n\n";

   third_fun();           // Call the third function to
                          // show that globals are visible.
   return 0;
}

third_fun()
{
   cout << "In the third function: \n";
   cout << "The sales in the third function are " <<
           sales << "\n";
   cout << "The profit in the third function is " <<
           profit << "\n";
   // If sales and profit were local, they would not be
   // visible by more than one function.
   return 0;
}
```

Notice that the `main()` function can never use `sales` and `profit` because they are not visible to `main()`—even though they are global. Remember, global variables are visible only from their point of definition downward in the program. Statements that appear before global variable definitions cannot use those variables. Here is the result of running this program.

```
No variables defined in main()

The sales in the second function are 20000
The profit in the second function is 5000

In the third function:
The sales in the third function are 20000
The profit in the third function is 5000
```

TIP: Declare all global variables at the top of your programs. Even though you can define them later (between any two functions), you can find them faster if you declare them at the top.

4. The following program uses both local and global variables. It should now be obvious to you that `j` and `p` are local and `i` and `z` are global.

```
// Filename: C17GLLO.CPP
// Program with both local and global variables.
// Local Variables       Global Variables
//     j, p                    i, z
#include <iostream.h>
pr_again();  // Prototype

int i = 0;                // Global variable because it's
                                // defined outside main().
main()
{
   float p ;                        // Local to main() only.
   p = 9.0;                // Puts value in global variable.
   cout << i << ", " << p << "\n";      // Prints global i
                                          //  and local p.
   pr_again();                     // Calls next function.
   return 0;                            // Returns to DOS.

 }
```

```
float z = 9.0;              // Global variable because it's
                              //  defined before a function.
pr_again()
{
   int j = 5;                   // Local to only pr_again().
   cout << j << ", " << z;       // This can't print p!.
   cout << ", " << i << "\n";
   return 0;                              // Return to main().
 }
```

Even though `j` is defined in a function that `main()` calls, `main()` cannot use `j` because `j` is local to `pr_again()`. When `pr_again()` finishes, `j` is no longer defined. The variable `z` is global from its point of definition down. This is why `main()` cannot print `z`. Also, the function `pr_again()` cannot print `p` because `p` is local to `main()` only.

Make sure you can recognize local and global variables before you continue. A little study here makes the rest of this chapter easy to understand.

5. Two variables can have the same name, as long as they are local to two different functions. They are distinct variables, even though they are named identically.

   The following short program uses two variables, both named `age`. They have two different values, and they are considered to be two different variables. The first `age` is local to `main()`, and the second `age` is local to `get_age()`.

```
// Filename: C17LOC1.CPP
// Two different local variables with the same name.
#include <iostream.h>
get_age();  // Prototype
main()
{
   int age;
   cout << "What is your age? ";
   cin >> age;

   get_age();                  // Call the second function.
   cout << "main()'s age is still " << age << "\n";
```

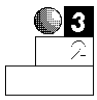
```
   return 0;
}

get_age()
{
   int age;                  // A different age. This one
                                //  is local to get_age().
   cout << "What is your age again? ";
   cin >> age;
   return 0;
}
```

Variables local to `main()` cannot be used in another function that `main()` calls.

The output of this program follows. Study this output carefully. Notice that `main()`'s last `cout` does not print the newly changed `age`. Rather, it prints the `age` known to `main()`—the `age` that is *local* to `main()`. Even though they are named the same, `main()`'s `age` has nothing to do with `get_age()`'s `age`. They might as well have two different variable names.

```
What is your age? 28
What is your age again? 56
main()'s age is still 28
```

You should be careful when naming variables. Having two variables with the same name is misleading. It would be easy to become confused while changing this program later. If these variables truly have to be separate, name them differently, such as `old_age` and `new_age`, or `ag1` and `ag2`. This helps you remember that they are different.



6. There are a few times when overlapping local variable names does not add confusion, but be careful about overdoing it. Programmers often use the same variable name as the counter variable in a `for` loop. For example, the two local variables in the following program have the same name.

```
// Filename: C17LOC2.CPP
// Using two local variables with the same name
```

```
// as counting variables.
#include <iostream.h>
do_fun();  // Prototype
main()
{
   int ctr;                                  // Loop counter.
   for (ctr=0; ctr<=10; ctr++)
      { cout << "main()'s ctr is " << ctr << "\n"; }
   do_fun();                        // Call second function.

   return 0;
}

do_fun()
{
   int ctr;
   for (ctr=10; ctr>=0; ctr--)
      { cout << "do_fun()'s ctr is " << ctr << "\n"; }
   return 0;                                 // Return to main().
}
```

Although this is a nonsense program that simply prints `0` through `10` and then prints `10` through `0`, it shows that using `ctr` for both function names is not a problem. These variables do not hold important data that must be processed; rather, they are `for` loop-counting variables. Calling them both `ctr` leads to little confusion because their use is limited to controlling `for` loops. Because a `for` loop initializes and increments variables, the one function never relies on the other function's `ctr` to do anything.

7. Be careful about creating local variables with the same name in the same function. If you define a local variable early in a function and then define another local variable with the same name inside a new block, C++ uses only the innermost variable, until its block ends.

   The following example helps clarify this confusing problem. The program contains one function with three local variables. See if you can find these three variables.

```
// Filename: C17MULI.CPP
// Program with multiple local variables called i.
#include <iostream.h>
main()
{
   int i;                                  // Outer i
   i = 10;

   { int i;                                // New block's i
     i = 20;                    // Outer i still holds a 10.
     cout << i << " " << i  << "\n";       // Prints 20 20.

     { int i;     // Another new block and local variable.
       i = 30;                          // Innermost i only.
       cout << i << " " << i <<
               " " << i << "\n";         // Prints 30 30 30.
     }                 // Innermost i is now gone forever.

    }       // Second i is gone forever (its block ended).

   cout << i << " " << i << " " <<
           i << "\n";                    // Prints 10 10 10.
   return 0;
}                  // main() ends and so do its variables.
```

All local variables are local to the block in which they are defined. This program has three blocks, each one nested within another. Because you can define local variables immediately after an opening brace of a block, there are three distinct `i` variables in this program.

The local `i` disappears completely when its block ends (when the closing brace is reached). C++ always prints the variable that it interprets as the most local—the one that resides within the innermost block.

Use Global Variables Sparingly

You might be asking yourself, "Why do I have to understand global and local variables?" At this point, that is an understandable

question, especially if you have been programming mostly in BASIC. Here is the bottom line: Global variables can be *dangerous.* Code can inadvertently overwrite a variable that was initialized in another place in the program. It is better to have every variable in your program be *local to the function that has to access it.*

Read the last sentence again. Even though you now know how to make variables global, you should avoid doing so! Try to never use another global variable. It might seem easier to use global variables when you write programs having more than one function: If you make every variable used by every function global, you never have to worry whether one is visible or not to any given function. On the other hand, a function can accidentally change a global variable when that was not your intention. If you keep variables local only to functions that need them, you protect their values, and you also keep your programs fully modular.

The Need for Passing Variables

You just learned the difference between local and global variables. You saw that by making your variables local, you protect their values because the function that sees the variable is the only one that can modify it.

What do you do, however, if you have a local variable you want to use in *two or more* functions? In other words, you might need a variable to be both added from the keyboard in one function and printed in another function. If the variable is local only to the first function, how can the second one access it?

You have two solutions if more than one function has to share a variable. One, you can declare the variable globally. This is not a good idea because you want only those two functions to have access to the variable, but all functions have access to it when it's global. The other alternative—and the better one by far—is to *pass* the local variable from one function to another. This has a big advantage: The variable is only known to those two functions. The rest of the program still has no access to it.

CAUTION: Never pass a global variable to a function. There is no reason to pass global variables anyway because they are already visible to all functions.

You pass an argument when you pass one local variable to another function.

When you pass a local variable from one function to another, you *pass an argument* from the first function to the next. You can pass more than one argument (variable) at a time, if you want several local variables to be sent from one function to another. The receiving function *receives a parameter* (variable) from the function that sends it. You shouldn't worry too much about what you call them—either arguments or parameters. The important thing to remember is that you are sending local variables from one function to another.

NOTE: You have already passed arguments to parameters when you passed data to the `cout` operator. The literals, variables, and expressions in the `cout` parentheses are arguments. The built-in `cout` function receives these values (called parameters on the receiving end) and displays them.

A little more terminology is needed before you see some examples. When a function passes an argument, it is called the *calling function.* The function that receives the argument (called a parameter when it is received) is called the *receiving function.* Figure 17.1 explains these terms.

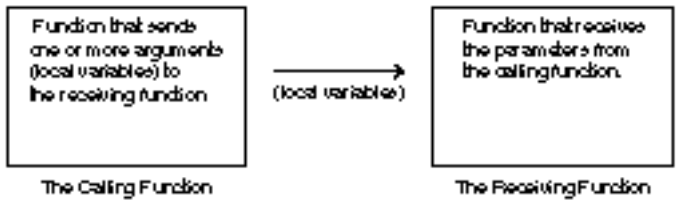


Figure 17.1. The calling and receiving functions.

If a function name has empty parentheses, nothing is being passed to it.

To pass a local variable from one function to another, you must place the local variable in parentheses in both the calling function and the receiving function. For example, the local and global

examples presented earlier did not pass local variables from `main()` to `do_fun()`. If a function name has empty parentheses, nothing is being passed to it. Given this, the following line passes two variables, `total` and `discount`, to a function called `do_fun()`.

```
do_fun(total, discount);
```

It is sometimes said that a variable or function is *defined*. This has nothing to do with the `#define` preprocessor directive, which defines literals. You define variables with statements such as the following:

```
int i, j;
int m=9;
float x;
char ara[] = "Tulsa";
```

These statements tell the program that you need these variables to be reserved. A function is defined when the C++ compiler reads the first statement in the function that describes the name and when it reads any variables that might have been passed to that function as well. Never follow a function definition with a semicolon, but always follow the statement that calls a function with a semicolon.

NOTE: To some C++ purists, a variable is only declared when you write `int i;` and only truly defined when you assign it a value, such as `i=7;`. They say that the variable is both declared and defined when you declare the variable and assign it a value at the same time, such as `int i=7;`.

The following program contains two function definitions, `main()` and `pr_it()`.

To practice passing a variable to a function, declare `i` *as an integer variable and make it equal to five. The passing (or calling) function is* `main()`*, and the receiving function is* `pr_it()`*. Pass the* `i` *variable to the* `pr_it()` *function, then go back to* `main()`*.*

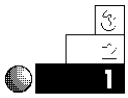
```
main()                    // The main() function definition.
{
  int i=5;                // Defines an integer variable.
  pr_it(i);               // Calls the pr_it().
                          // function and passes it i.
  return 0;               // Returns to the operating system.
}

pr_it(int i)              // The pr_it() function definition.
{
   cout << i << "\n";     // Calls the cout operator.
   return 0;              // Returns to main().
}
```

Because a passed parameter is treated like a local variable in the receiving function, the `cout` in `pr_it()` prints a `5`, even though the `main()` function initialized this variable.

When you pass arguments to a function, the receiving function is not aware of the data types of the incoming variables. Therefore, you must include each parameter's data type in front of the parameter's name. In the previous example, the definition of `pr_it()` (the first line of the function) contains the type, `int`, of the incoming variable `i`. Notice that the `main()` calling function does not have to indicate the variable type. In this example, `main()` already knows the type of variable `i` (an integer); only `pr_it()` has to know that `i` is an integer.

TIP: Always declare the parameter types in the receiving function. Precede each parameter in the function's parentheses with `int`, `float`, or whatever each passed variable's data type is.

Examples



1. Here is a `main()` function that contains three local variables. `main()` passes one of these variables to the first function and two of them to the second function.

```
// Filename: C17LOC3.CPP
// Pass three local variables to functions.
#include <iostream.h>
#include <iomanip.h>
pr_init(char initial);  // Prototypes discussed later.
pr_other(int age, float salary);

main()
{
   char initial;           // Three variables local to
                           // main().
   int age;
   float salary;

   // Fill these variables in main().
   cout << "What is your initial? ";
   cin >> initial;
   cout << "What is your age? ";
   cin >> age;
   cout << "What is your salary? ";
   cin >> salary;

   pr_init(initial);               // Call pr_init() and
                                      // pass it initial.
   pr_other(age, salary);          // Call pr_other() and
                                // pass it age and salary.

   return 0;
}

pr_init(char initial)         // Never put a semicolon in
                               // the function definition.
{
   cout << "Your initial is " << initial << "\n";
   return 0;                           // Return to main().
}

pr_other(int age, float salary) // Must type both parameters.
{
   cout << "You look young for " << age << "\n";
   cout << "And " << setprecision(2) << salary <<
```

```
            " is a LOT of money!";
   return 0;                              // Return to main().
}
```

2. A receiving function can contain its own local variables. As long as the names are not the same, these local variables do not conflict with the passed ones. In the following program, the second function receives a passed variable from `main()` and defines its own local variable called `price_per`.

```
// Filename: C17LOC4.CPP
// Second function has its own local variable.
#include <iostream.h>
#include <iomanip.h>
compute_sale(int gallons); // Prototypes discussed later.

main()
{
   int gallons;

   cout << "Richard's Paint Service \n";
   cout << "How many gallons of paint did you buy? ";
   cin >> gallons;           // Get gallons in main().

   compute_sale(gallons);    // Compute total in function.
   return 0;
}

compute_sale(int gallons)
{
   float price_per = 12.45;    // Local to compute_sale().

   cout << "The total is " << setprecision(2) <<
           (price_per*(float)gallons) << "\n";
         // Had to type cast gallons because it was integer.
   return 0;                              // Return to main().
}
```

3. The following sample code lines test your skill at recognizing calling functions and receiving functions. Being able to recognize the difference is half the battle of understanding them.

```
do_it()
```

The preceding fragment must be the first line of a new function because it does not end with a semicolon.

```
do_it2(sales);
```

This line calls a function called `do_it2()`. The calling function passes the variable called `sales` to `do_it2()`.

```
pr_it(float total)
```

The preceding line is the first line of a function that receives a floating-point variable from another function that called it. All receiving functions must specify the type of each variable being passed.

```
pr_them(float total, int number)
```

This is the first line of a function that receives two variables—one is a floating-point variable and the other is an integer. This line cannot be calling the function `pr_them` because there is no semicolon at the end of the line.

Automatic Versus Static Variables

The terms *automatic* and *static* describe what happens to local variables when a function returns to the calling procedure. By default, all local variables are automatic, meaning that they are erased when their function ends. You can designate a variable as automatic by prefixing its definition with the term `auto`. The `auto` keyword is optional with local variables because they are automatic be default.

The two statements after `main()`'s opening brace declare automatic local variables:

```
main()
{
   int i;
   auto float x;
   // Rest of main() goes here.
```

Because `auto` is the default, you did not have to include the term `auto` with `x`.

NOTE: C++ programmers rarely use the `auto` keyword with local variables because they are automatic by default.

Automatic variables are local and disappear when their function ends.

The opposite of an automatic variable is a static variable. All global variables are static and, as mentioned, all static variables retain their values. Therefore, if a local variable is static, it too retains its value when its function ends—in case the function is called a second time. To declare a variable as static, place the `static` keyword in front of the variable when you define it. The following code section defines three variables, `i`, `j`, and `k`. The variable `i` is automatic, but `j` and `k` are static.

```
my_fun()            // Start of new function definition.
{
   int i;
   static j=25;    // Both j and k are static variables.
   static k=30;
```

If local variables are static, their values remain in case the function is called again.

Always assign an initial value to a static variable when you declare it, as shown here in the last two lines. This initial value is placed in the static variable only the first time `my_fun()` executes. If you don't assign a static variable an initial value, C++ initializes it to zero.

TIP: Static variables are good to use when you write functions that keep track of a count or add to a total. If the counting or totaling variables were local and automatic, their values would disappear when the function finished—destroying the totals.

Automatic and Static Rules for Local Variables

Local automatic variables disappear when their block ends. All local variables are automatic by default. You can prefix a variable (when you define it) with the `auto` keyword, or you can omit it; the variable is still automatic and its value is destroyed when its local block ends.

Local static variables do not lose their values when their function ends. They remain local to that function. When the function is called after the first time, the static variable's value is still in place. You declare a static variable by placing the `static` keyword before the variable's definition.

Examples

1. Consider this program:

```
// Filename: C17STA1.CPP
// Tries to use a static variable
// without a static declaration.
#include <iostream.h>
triple_it(int ctr);

main()
{
   int ctr;                    // Used in the for loop to
                               // call a function 25 times.
   for (ctr=1; ctr<=25; ctr++)
     { triple_it(ctr); }        // Pass ctr to a function
                                    // called triple_it().
   return 0;
}

triple_it(int ctr)
{
   int total=0, ans;           // Local automatic variables.
```

```
    // Triples whatever value is passed to it
    // and adds the total.

    ans = ctr * 3;                  // Triple number passed.
    total += ans;  // Add triple numbers as this is called.

    cout << "The number " << ctr << " multiplied by 3 is "
         << ans << "\n";

    if (total > 300)
      { cout << "The total of triple numbers is over 300 \n"; }
    return 0;
}
```

This is a nonsense program that doesn't do much, yet you might sense something is wrong. The program passes numbers from 1 to 25 to the function called `triple_it`. The function triples the number and prints it.

The variable called `total` is initially set to 0. The idea here is to add each tripled number and print a message when the total is larger than 300. However, the `cout` never executes. For each of the 25 times that this subroutine is called, `total` is reset to 0. The `total` variable is an automatic variable, with its value erased and initialized every time its procedure is called. The next example corrects this.

2. If you want `total` to retain its value after the procedure ends, you must make it static. Because local variables are automatic by default, you have to include the `static` keyword to override this default. Then the value of the `total` variable is retained each time the subroutine is called.

   The following corrects the mistake in the previous program.

```
// Filename: C17STA2.CPP
// Uses a static variable with the static declaration.
#include <iostream.h>
triple_it(int ctr);

main()
```

```
{
   int ctr;                    // Used in the for loop to
                              //  call a function 25 times.
   for (ctr=1; ctr<=25; ctr++)
     { triple_it(ctr); }        // Pass ctr to a function
                                    // called triple_it().

   return 0;
}

triple_it(int ctr)
{
   static int total=0;               // Local and static
   int ans;                         // Local and automatic
   // total is set to 0 only the first time this
   // function is called.

   // Triples whatever value is passed to it and adds
   // the total.

   ans = ctr * 3;                  // Triple number passed.
   total += ans;  // Add triple numbers as this is called.

   cout << "The number " << ctr << " multiplied by 3 is "
          << ans << "\n";

   if (total > 300)
     { cout << "The total of triple numbers is over 300 \n"; }
   return 0;
}
```

This program's output follows. Notice that the function's `cout` is triggered, even though `total` is a local variable. Because `total` is static, its value is not erased when the function finishes. When `main()` calls the function a second time, `total`'s previous value (at the time you left the routine) is still there.

```
The number 1 multiplied by 3 is 3
The number 2 multiplied by 3 is 6
The number 3 multiplied by 3 is 9
The number 4 multiplied by 3 is 12
```

```
The number 5 multiplied by 3 is 15
The number 6 multiplied by 3 is 18
The number 7 multiplied by 3 is 21
The number 8 multiplied by 3 is 24
The number 9 multiplied by 3 is 27
The number 10 multiplied by 3 is 30
The number 11 multiplied by 3 is 33
The number 12 multiplied by 3 is 36
The number 13 multiplied by 3 is 39
The number 14 multiplied by 3 is 42
The number 15 multiplied by 3 is 45
The number 16 multiplied by 3 is 48
The number 17 multiplied by 3 is 51
The number 18 multiplied by 3 is 54
The number 19 multiplied by 3 is 57
The number 20 multiplied by 3 is 60
The number 21 multiplied by 3 is 63
The number 22 multiplied by 3 is 66
The number 23 multiplied by 3 is 69
The number 24 multiplied by 3 is 72
The number 25 multiplied by 3 is 75
```

This does not mean that local static variables become global. The main program cannot refer, use, print, or change `total` because it is local to the second function. Static simply means that the local variable's value is still there if the program calls the function again.

Three Issues of Parameter Passing

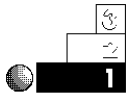
To have a complete understanding of programs with several functions, you have to learn three additional concepts:

- Passing arguments (variables) by value (also called "by copy")
- Passing arguments (variables) by address (also called "by reference")
- Returning values from functions

The first two concepts deal with the way local variables are passed and received. The third concept describes how receiving functions send values back to the calling functions. Chapter 18, "Passing Values," concludes this discussion by explaining these three methods for passing parameters and returning values.

Review Questions

The answers to the review questions are in Appendix B.



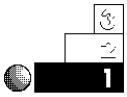
1. True or false: A function should always include a `return` statement as its last command, even though `return` is not required.
2. When a local variable is passed, is it called an argument or a parameter?

3. True or false: A function that is passed variables from another function cannot also have its own local variables.
4. What must appear inside the receiving function's parentheses, other than the variables passed to it?
5. If a function keeps track of a total or count every time it is called, should the counting or totaling variable be automatic or static?

6. When would you pass a global variable to a function? (Be careful—this might be a trick question!)
7. How many arguments are there in the following statement?

```
printf("The rain has fallen %d inches.", rainf);
```

Review Exercises



1. Write a program that asks, in `main()`, for the age of the user's dog. Write a second function called `people()` that computes the dog's age in human years (by multiplying the dog's age by seven).

2. Write a function that counts the number of times it is called. Name the function `count_it()`. Do not pass it anything. In the body of `count_it()`, print the following message:

   ```
   The number of times this function has been called is: ##
   ```

   where `##` is the number. (*Hint:* Because the variable must be local, make it static and initialize it to zero when you first define it.)

3. The following program contains several problems. Some of these problems produce errors. One problem is not an error, but a bad location for a variable declaration. (*Hint:* Find all the global variables.) See if you can spot some of the problems, and rewrite the program so it works better.

```
// Filename: C17BAD.CPP
// Program with bad uses of variable declarations.
#include <iostream.h>
#define NUM 10
do_var_fun();  // Prototypes discussed later.

char city[] = "Miami";
int count;

main()
{
   int abc;

   count = NUM;
   abc = 5;
   do_var_fun();

   cout << abc << " " << count << " " << pgm_var << " "
        << xyz;
   return 0;
}

int pgm_var = 7;

do_var_fun()
```

```
{
   char xyz = 'A';

   xyz = 'b';
   cout << xyz << " " << pgm_var << " " abc << " " << city;
   return 0;
}
```

Summary

Parameter passing is necessary because local variables are better than global. Local variables are protected in their own routines, but sometimes they must be shared with other routines. If local data are to remain in those variables (in case the function is called again in the same program), the variables should be static because otherwise their automatic values disappear.

Most the information in this chapter becomes more obvious as you use functions in your own programs. Chapter 18, "Passing Values," covers the actual passing of parameters in more detail and shows you two different ways to pass them.