7

Functions: C++'s Programming Modules

In this chapter you'll learn about the following:

- Function basics
- Function prototypes
- Passing function arguments by value
- Designing functions to process arrays
- Using `const` pointer parameters
- Designing functions to process text strings
- Designing functions to process structures
- Designing functions to process objects of the `string` class
- Functions that call themselves (recursion)
- Pointers to functions

Fun is where you find it. Look closely, and you can find it in functions. C++ comes with a large library of useful functions (the standard ANSI C library plus several C++ classes), but real programming pleasure comes with writing your own functions. (On the other hand, real programming productivity can come with learning more about what you can do with the STL and the BOOST C++ libraries.) This chapter and Chapter 8, "Adventures in Functions," examine how to define functions, convey information to them, and retrieve information from them. After reviewing how functions work, this chapter concentrates on how to use functions in conjunction with arrays, strings, and structures. Finally, it touches on recursion and pointers to functions. If you've paid your C dues, you'll find much of this chapter familiar. But don't be lulled into a false sense of expertise. C++ has made several additions to what C functions can do, and Chapter 8 deals primarily with those. Meanwhile, let's attend to the fundamentals.

Function Review

Let's review what you've already seen about functions. To use a C++ function, you must do the following:

- Provide a function definition
- Provide a function prototype
- Call the function

If you're using a library function, the function has already been defined and compiled for you. Also you can and should use a standard library header file to provide the prototype. All that's left to do is call the function properly. The examples so far in this book have done that several times. For example, the standard C library includes the `strlen()` function for finding the length of the string. The associated standard header file `cstring` contains the function prototype for `strlen()` and several other string-related functions. This advance work allows you to use the `strlen()` function in programs without further worries.

But when you create your own functions, you have to handle all three aspects—defining, prototyping, and calling—yourself. Listing 7.1 shows these steps in a short example.

Listing 7.1 **`calling.cpp`**

```
// calling.cpp -- defining, prototyping, and calling a function
#include <iostream>

void simple();    // function prototype

int main()
{
    using namespace std;
    cout << "main() will call the simple() function:\n";
    simple();     // function call
        cout << "main() is finished with the simple() function.\n";
    // cin.get();
    return 0;
}

// function definition
void simple()
{
    using namespace std;
    cout << "I'm but a simple function.\n";
}
```

Here's the output of the program in Listing 7.1:

```
main() will call the simple() function:
I'm but a simple function.
main() is finished with the simple() function.
```

Program execution in `main()` halts as control transfers to the `simple()` function. When `simple()` finishes, program execution in `main()` resumes. This example places a `using` directive inside each function definition because each function uses `cout`. Alternatively, the program could have a single `using` directive placed above the function definitions or otherwise use `std::cout`.

Let's take a more detailed look at these steps now.

Defining a Function

You can group functions into two categories: those that don't have return values and those that do. Functions without return values are termed type `void` functions and have the following general form:

```
void functionName(parameterList)
{
    statement(s)
    return;          // optional
}
```

Here *`parameterList`* specifies the types and number of arguments (parameters) passed to the function. This chapter more fully investigates this list later. The optional return statement marks the end of the function. Otherwise, the function terminates at the closing brace. Type `void` functions correspond to Pascal procedures, FORTRAN subroutines, and modern BASIC subprogram procedures. Typically, you use a `void` function to perform some sort of action. For example, a function to print *Cheers!* a given number (`n`) of times could look like this:

```
void cheers(int n)           // no return value
{

    for (int i = 0; i < n; i++)
        std::cout << "Cheers! ";
    std::cout << std::endl;
}
```

The `int n` parameter list means that `cheers()` expects to have an `int` value passed to it as an argument when you call this function.

A function with a return value produces a value that it returns to the function that called it. In other words, if the function returns the square root of 9.0 (`sqrt(9.0)`), the function call has the value `3.0`. Such a function is declared as having the same type as the value it returns. Here is the general form:

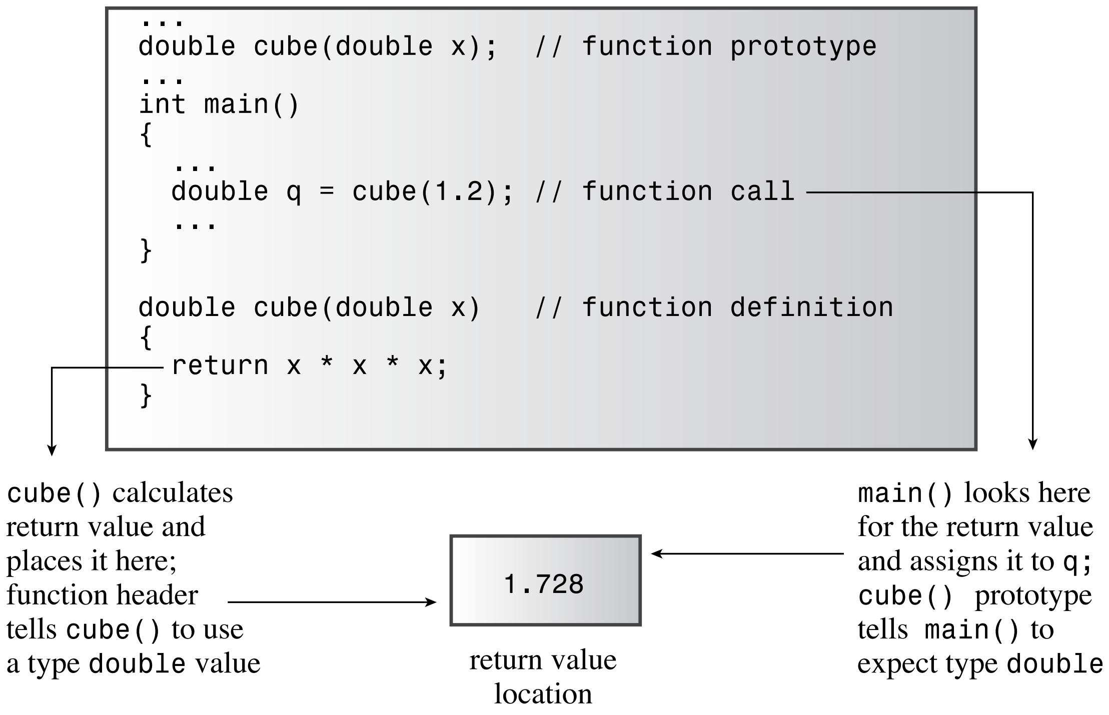
```
typeName functionName(parameterList)
{
      statements
      return value;   // value is type cast to type typeName
}
```

Functions with return values require that you use a return statement so that the value is returned to the calling function. The value itself can be a constant, a variable, or a more general expression. The only requirement is that the expression reduces to a value that has, or is convertible to, the *`typeName`* type. (If the declared return type is, say, `double`, and the function returns an `int` expression, the `int` value is type cast to type `double`.) The function then returns the final value to the function that called it. C++ does place a restriction on what types you can use for a return value: The return value cannot be an array. Everything else is possible—integers, floating-point numbers, pointers, and even structures and objects! (Interestingly, even though a C++ function can't return an array directly, it can return an array that's part of a structure or object.)

As a programmer, you don't need to know how a function returns a value, but knowing the method might clarify the concept for you. (Also it gives you something to talk about with your friends and family.) Typically, a function returns a value by copying the return value to a specified CPU register or memory location. Then the calling program examines that location. Both the returning function and the calling function have to agree on the type of data at that location. The function prototype tells the calling program what to expect, and the function definition tells the called program what to return (see Figure 7.1). Providing the same information in the prototype as in the definition might seem like extra work, but it makes good sense. Certainly, if you want a courier to pick up something from your desk at the office, you enhance the odds of the task being done right if you provide a description of what you want both to the courier and to someone at the office.

A function terminates after executing a return statement. If a function has more than one return statement—for example, as alternatives to different `if else` selections—the function terminates after it executes the first return statement it reaches. For instance, in the following example, the `else` isn't needed, but it does help the casual reader understand the intent:

```
int bigger(int a, int b)
{
    if (a > b )
        return a;  // if a > b, function terminates here
    else
        return b;  // otherwise, function terminates here
}
```


```
...
double cube(double x);  // function prototype
...
int main()
{
  ...
  double q = cube(1.2); // function call
  ...
}

double cube(double x)   // function definition
{
  return x * x * x;
}
```
cube() calculates return value and places it here; function header tells cube() to use a type double value
1.728
return value location
main() looks here for the return value and assigns it to q; cube() prototype tells main() to expect type double

Figure 7.1 A typical return value mechanism.

(Usually, having multiple return statements in a function is considered potentially confusing, and some compilers might issue a warning. However, the code here is simple enough to understand.)

Functions with return values are much like functions in Pascal, FORTRAN, and BASIC. They return a value to the calling program, which can then assign that value to a variable, display the value, or otherwise use it. Here's a simple example that returns the cube of a type `double` value:

```
double cube(double x)    // x times x times x
{
    return x * x * x; // a type double value
}
```

For example, the function call `cube(1.2)` returns the value `1.728`. Note that this return statement uses an expression. The function computes the value of the expression (`1.728`, in this case) and returns the value.

Prototyping and Calling a Function

By now you are familiar with making function calls, but you may be less comfortable with function prototyping because that's often been hidden in the `include` files. Listing 7.2 shows the `cheers()` and `cube()` functions used in a program; notice the function prototypes.

Listing 7.2 **protos.cpp**

```
// protos.cpp -- using prototypes and function calls
#include <iostream>
void cheers(int);       // prototype: no return value
double cube(double x);  // prototype: returns a double
int main()
{
    using namespace std;
    cheers(5);          // function call
    cout << "Give me a number: ";
    double side;
    cin >> side;
    double volume = cube(side);    // function call
    cout << "A " << side <<"-foot cube has a volume of ";
    cout << volume << " cubic feet.\n";
    cheers(cube(2));    // prototype protection at work
    return 0;
}

void cheers(int n)
{
    using namespace std;
    for (int i = 0; i < n; i++)
        cout << "Cheers! ";
    cout << endl;
}

double cube(double x)
{
    return x * x * x;
}
```

The program in Listing 7.2 places a `using` directive in only those functions that use the members of the `std` namespace. Here's a sample run:

```
Cheers! Cheers! Cheers! Cheers! Cheers!
Give me a number: 5
A 5-foot cube has a volume of 125 cubic feet.
Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers!
```

Note that `main()` calls the type `void` function `cheers()` by using the function name and arguments followed by a semicolon: `cheers(5);`. This is an example of a function call statement. But because `cube()` has a return value, `main()` can use it as part of an assignment statement:

```
double volume = cube(side);
```

But I said earlier that you should concentrate on the prototypes. What should you know about prototypes? First, you should understand why C++ requires prototypes. Then because C++ requires prototypes, you should know the proper syntax. Finally, you should appreciate what the prototype does for you. Let's look at these points in turn, using Listing 7.2 as a basis for discussion.

Why Prototypes?

A prototype describes the function interface to the compiler. That is, it tells the compiler what type of return value, if any, the function has, and it tells the compiler the number and type of function arguments. Consider how the prototype affects this function call from Listing 7.2:

```
double volume = cube(side);
```

First, the prototype tells the compiler that `cube()` should have one type `double` argument. If the program fails to provide the argument, prototyping allows the compiler to catch the error. Second, when the `cube()` function finishes its calculation, it places its return value at some specified location—perhaps in a CPU register, perhaps in memory. Then the calling function, `main()` in this case, retrieves the value from that location. Because the prototype states that `cube()` is type `double`, the compiler knows how many bytes to retrieve and how to interpret them. Without that information, the compiler could only guess, and that is something compilers won't do.

Still, you might wonder, why does the compiler need a prototype? Can't it just look further in the file and see how the functions are defined? One problem with that approach is that it is not very efficient. The compiler would have to put compiling `main()` on hold while searching the rest of the file. An even more serious problem is the fact that the function might not even be in the file. C++ allows you to spread a program over several files, which you can compile independently and then combine later. In such a case, the compiler might not have access to the function code when it's compiling `main()`. The same is true if the function is part of a library. The only way to avoid using a function prototype is to place the function definition before its first use. That is not always possible. Also the C++ programming style is to put `main()` first because it generally provides the structure for the whole program.

Prototype Syntax

A function prototype is a statement, so it must have a terminating semicolon. The simplest way to get a prototype is to copy the function header from the function definition and add a semicolon. That's what the program in Listing 7.2 does for `cube()`:

```
double cube(double x); // add ; to header to get prototype
```

However, the function prototype does not require that you provide names for the variables; a list of types is enough. The program in Listing 7.2 prototypes `cheers()` by using only the argument type:

```
void cheers(int); // okay to drop variable names in prototype
```

In general, you can either include or exclude variable names in the argument lists for prototypes. The variable names in the prototype just act as placeholders, so if you do use names, they don't have to match the names in the function definition.

C++ Versus ANSI C Prototyping

ANSI C borrowed prototyping from C++, but the two languages do have some differences. The most important is that ANSI C, to preserve compatibility with classic C, made prototyping optional, whereas C++ makes prototyping mandatory. For example, consider the following function declaration:

```
void say_hi();
```

In C++, leaving the parentheses empty is the same as using the keyword `void` within the parentheses. It means the function has no arguments. In ANSI C, leaving the parentheses empty means that you are declining to state what the arguments are. That is, it means you're forgoing prototyping the argument list. The C++ equivalent for not identifying the argument list is to use an ellipsis:

```
void say_bye(...);   // C++ abdication of responsibility
```

Normally this use of an ellipsis is needed only for interfacing with C functions having a variable number of arguments, such as `printf()`.

What Prototypes Do for You

You've seen that prototypes help the compiler. But what do they do for you? They greatly reduce the chances of program errors. In particular, prototypes ensure the following:

- The compiler correctly handles the function return value.
- The compiler checks that you use the correct number of function arguments.
- The compiler checks that you use the correct type of arguments. If you don't, it converts the arguments to the correct type, if possible.

We've already discussed how to correctly handle the return value. Let's look now at what happens when you use the wrong number of arguments. For example, suppose you make the following call:

```
double z = cube();
```

A compiler that doesn't use function prototyping lets this go by. When the function is called, it looks where the call to `cube()` should have placed a number and uses whatever value happens to be there. This is how C worked before ANSI C borrowed prototyping from C++. Because prototyping is optional for ANSI C, this is how some C programs still work. But in C++ prototyping is not optional, so you are guaranteed protection from that sort of error.

Next, suppose you provide an argument but it is the wrong type. In C, this could create weird errors. For example, if a function expects a type `int` value (assume that's 16 bits) and you pass a `double` (assume that's 64 bits), the function looks at just the first 16 bits of the 64 and tries to interpret them as an `int` value. However, C++ automatically converts the value you pass to the type specified in the prototype, provided that both are arithmetic types. For example, Listing 7.2 manages to get two type mismatches in one statement:

```
cheers(cube(2));
```

First, the program passes the `int` value of `2` to `cube()`, which expects type `double`. The compiler, noting that the `cube()` prototype specifies a type `double` argument, converts `2` to `2.0`, a type `double` value. Then `cube()` returns a type `double` value (`8.0`) to be used as an argument to `cheers()`. Again, the compiler checks the prototypes and notes that `cheers()` requires an `int`. It converts the return value to the integer `8`. In general, prototyping produces automatic type casts to the expected types. (Function overloading, discussed in Chapter 8, can create ambiguous situations, however, that prevent some automatic type casts.)

Automatic type conversion doesn't head off all possible errors. For example, if you pass a value of `8.33E27` to a function that expects an `int`, such a large value cannot be converted correctly to a mere `int`. Some compilers warn you of possible data loss when there is an automatic conversion from a larger type to a smaller.

Also prototyping results in type conversion only when it makes sense. It won't, for example, convert an integer to a structure or pointer.

Prototyping takes place during compile time and is termed *static type checking*. Static type checking, as you've just seen, catches many errors that are much more difficult to catch during runtime.

Function Arguments and Passing by Value

It's time to take a closer look at function arguments. C++ normally passes arguments *by value*. That means the numeric value of the argument is passed to the function, where it is assigned to a new variable. For example, Listing 7.2 has this function call:

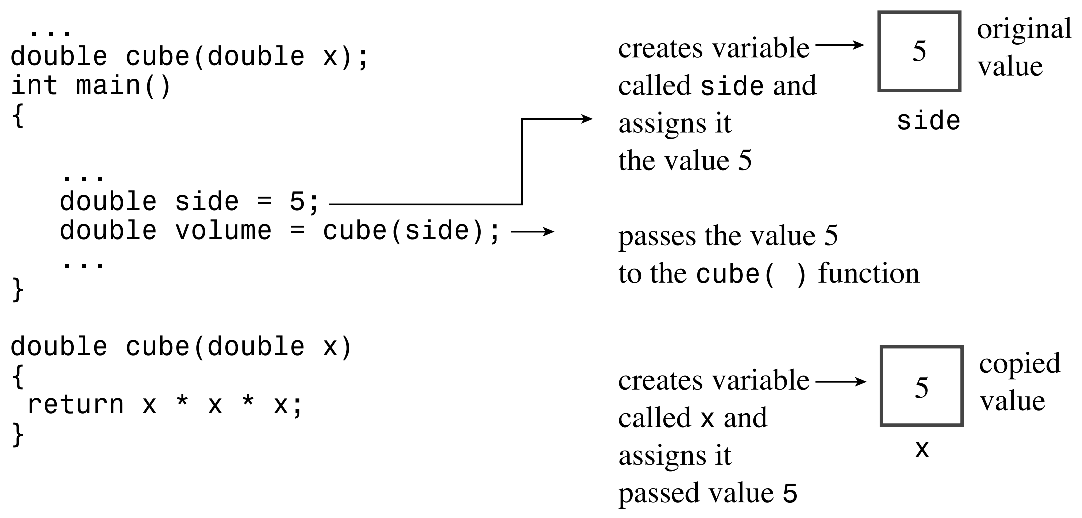
```
double volume = cube(side);
```

Here `side` is a variable that, in the sample run, had the value `5`. The function header for `cube()`, recall, was this:

```
double cube(double x)
```

When this function is called, it creates a new type `double` variable called `x` and initializes it with the value `5`. This insulates data in `main()` from actions that take place in `cube()` because `cube()` works with a copy of `side` rather than with the original data.

You'll see an example of this protection soon. A variable that's used to receive passed values is called a *formal argument* or *formal parameter*. The value passed to the function is called the *actual argument* or *actual parameter*. To simplify matters a bit, the C++ Standard uses the word *argument* by itself to denote an actual argument or parameter and the word *parameter* by itself to denote a formal argument or parameter. Using this terminology, argument passing initializes the parameter to the argument (see Figure 7.2).



```
 ...
double cube(double x);
int main()
{
   ...
   double side = 5;
   double volume = cube(side);
   ...
}

double cube(double x)
{
 return x * x * x;
}
```

creates variable called side and assigns it the value 5
5
original value
side
passes the value 5 to the cube( ) function
creates variable called x and assigns it passed value 5
5
copied value
x

Figure 7.2 Passing by value.

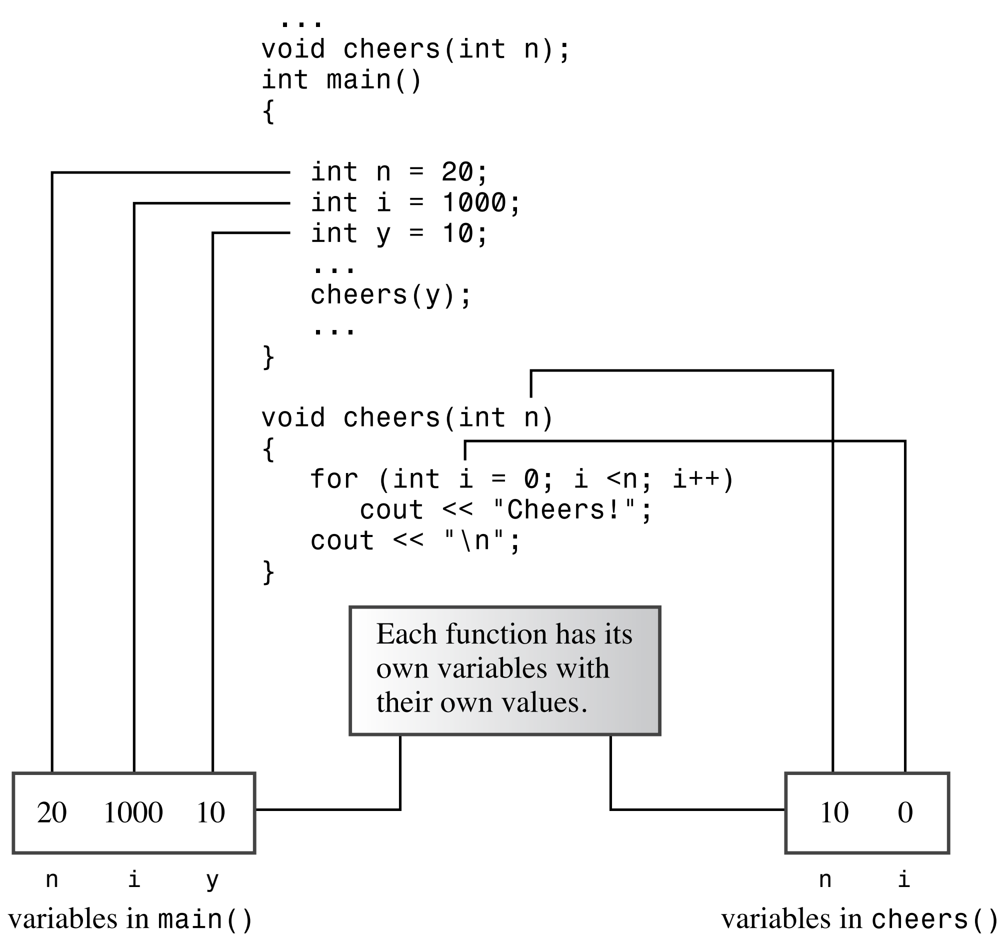
Variables, including parameters, declared within a function are private to the function. When a function is called, the computer allocates the memory needed for these variables. When the function terminates, the computer frees the memory that was used for those variables. (Some C++ literature refers to this allocating and freeing of memory as *creating and destroying variables*. That does make it sound much more exciting.) Such variables are called *local variables* because they are localized to the function. As mentioned previously, this helps preserve data integrity. It also means that if you declare a variable called `x` in `main()` and another variable called `x` in some other function, these are two distinct, unrelated variables, much as the Albany in California is distinct from the Albany in New York (see Figure 7.3). Such variables are also termed *automatic variables* because they are allocated and deallocated automatically during program execution.

Multiple Arguments

A function can have more than one argument. In the function call, you just separate the arguments with commas:

```
n_chars('R', 25);
```

This passes two arguments to the function `n_chars()`, which will be defined shortly.



```
 ...
void cheers(int n);
int main()
{
   int n = 20;
   int i = 1000;
   int y = 10;
   ...
   cheers(y);
   ...
}

void cheers(int n)
{
   for (int i = 0; i <n; i++)
      cout << "Cheers!";
   cout << "\n";
}
```

Figure 7.3 Local variables.

Similarly, when you define the function, you use a comma-separated list of parameter declarations in the function header:

```
void n_chars(char c, int n)  // two arguments
```

This function header states that the function `n_chars()` takes one type `char` argument and one type `int` argument. The parameters `c` and `n` are initialized with the values passed to the function. If a function has two parameters of the same type, you have to give the type of each parameter separately. You can't combine declarations the way you can when you declare regular variables:

```
void fifi(float a, float b)  // declare each variable separately
void fufu(float a, b)        // NOT acceptable
```

As with other functions, you just add a semicolon to get a prototype:

```
void n_chars(char c, int n); // prototype, style 1
```

As with single arguments, you don't have to use the same variable names in the prototype as in the definition, and you can omit the variable names in the prototype:

```
void n_chars(char, int);     // prototype, style 2
```

However, providing variable names can make the prototype more understandable, particularly if two parameters are the same type. Then the names can remind you which argument is which:

```
double melon_density(double weight, double volume);
```

Listing 7.3 shows an example of a function with two arguments. It also illustrates how changing the value of a formal parameter in a function has no effect on the data in the calling program.

Listing 7.3 **twoarg.cpp**

```
// twoarg.cpp -- a function with 2 arguments
#include <iostream>
using namespace std;
void n_chars(char, int);
int main()
{
    int times;
    char ch;

    cout << "Enter a character: ";
    cin >> ch;
    while (ch != 'q')        // q to quit
    {
        cout << "Enter an integer: ";
        cin >> times;
        n_chars(ch, times); // function with two arguments
        cout << "\nEnter another character or press the"
                " q-key to quit: ";
           cin >> ch;
    }
    cout << "The value of times is " << times << ".\n";
    cout << "Bye\n";
    return 0;
}

void n_chars(char c, int n) // displays c n times
{
    while (n-- > 0)         // continue until n reaches 0
        cout << c;
}
```

The program in Listing 7.3 illustrates placing a `using` directive above the function definitions rather than within the functions. Here is a sample run:

```
Enter a character: W
Enter an integer: 50
```

```
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
Enter another character or press the q-key to quit: a
Enter an integer: 20
aaaaaaaaaaaaaaaaaaaa
Enter another character or press the q-key to quit: q
The value of times is 20.
Bye
```

Program Notes

The `main()` function in Listing 7.3 uses a `while` loop to provide repeated input (and to keep your loop skills fresh). Note that it uses `cin >> ch` rather than `cin.get(ch)` or `ch = cin.get()` to read a character. There's a good reason for this. Recall that the two `cin.get()` functions read all input characters, including spaces and newlines, whereas `cin >>` skips spaces and newlines. When you respond to the program prompt, you have to press Enter at the end of each line, thus generating a newline character. The `cin >> ch` approach conveniently skips over these newlines, but the `cin.get()` siblings read the newline following each number entered as the next character to display. You can program around this nuisance, but it's simpler to use `cin` as the program in Listing 7.3 does.

The `n_chars()` function takes two arguments: a character `c` and an integer `n`. It then uses a loop to display the character the number of times the integer specifies:

```
while (n-- > 0)         // continue until n reaches 0
      cout << c;
```

Notice that the program keeps count by decrementing the `n` variable, where `n` is the formal parameter from the argument list. This variable is assigned the value of the `times` variable in `main()`. The `while` loop then decreases `n` to `0`, but, as the sample run demonstrates, changing the value of `n` has no effect on `times`. Even if you use the name `n` instead of `times` in `main()`, the value of `n` in `main()` is unaffected by changes in the value of `n` in `n_chars()`.

Another Two-Argument Function

Let's create a more ambitious function—one that performs a nontrivial calculation. Also the function illustrates the use of local variables other than the function's formal arguments.

Many states in the United States now sponsor a lottery with some form of Lotto game. Lotto lets you pick a certain number of choices from a card. For example, you might get to pick six numbers from a card having 51 numbers. Then the Lotto managers pick six numbers at random. If your choice exactly matches theirs, you win a few million dollars or so. Our function will calculate the probability that you have a winning pick. (Yes, a function that successfully predicts the winning picks themselves would be more useful, but C++, although powerful, has yet to implement psychic faculties.)

First, you need a formula. If you have to pick six values out of 51, mathematics says that you have one chance in `R` of winning, where the following formula gives `R`:

$$R = \frac{51 \quad 50 \quad 49 \quad 48 \quad 47 \quad 46}{6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1}$$

For six choices, the denominator is the product of the first six integers, or 6 factorial. The numerator is also the product of six consecutive numbers, this time starting with 51 and going down. More generally, if you pick `picks` values out of `numbers` numbers, the denominator is `picks` factorial and the numerator is the product of `picks` integers, starting with the value `numbers` and working down. You can use a `for` loop to make that calculation:

```
long double result = 1.0;
for (n = numbers, p = picks; p > 0; n--, p--)
    result = result * n / p ;
```

Rather than multiply all the numerator terms first, the loop begins by multiplying `1.0` by the first numerator term and then dividing by the first denominator term. Then in the next cycle, the loop multiplies and divides by the second numerator and denominator terms. This keeps the running product smaller than if you did all the multiplication first. For example, compare

```
(10 * 9) / (2 * 1)
```

with

```
(10 / 2) * (9 / 1)
```

The first evaluates to 90 / 2 and then to 45, whereas the second evaluates to 5   9 and then to 45. Both give the same answer, but the first method produces a larger intermediate value (90) than does the second. The more factors you have, the bigger the difference gets. For large numbers, this strategy of alternating multiplication with division can keep the calculation from overflowing the maximum possible floating-point value.

Listing 7.4 incorporates this formula into a `probability()` function. Because the number of picks and the total number of choices should be positive values, the program uses the `unsigned int` type (`unsigned`, for short) for those quantities. Multiplying several integers can produce pretty large results, so `lotto.cpp` uses the `long double` type for the function's return value. Also terms such as 49 / 6 produce a truncation error for integer types.

Note

Some C++ implementations don't support type `long double`. If your implementation falls into that category, try ordinary `double` instead.

Listing 7.4 **lotto.cpp**

```
// lotto.cpp -- probability of winning
#include <iostream>
// Note: some implementations require double instead of long double
long double probability(unsigned numbers, unsigned picks);
int main()
{
    using namespace std;
    double total, choices;
    cout << "Enter the total number of choices on the game card and\n"
            "the number of picks allowed:\n";
    while ((cin >> total >> choices) && choices <= total)
    {
        cout << "You have one chance in ";
        cout << probability(total, choices);      // compute the odds
        cout << " of winning.\n";
        cout << "Next two numbers (q to quit): ";
    }
    cout << "bye\n";
    return 0;
}

// the following function calculates the probability of picking picks
// numbers correctly from numbers choices
long double probability(unsigned numbers, unsigned picks)
{
    long double result = 1.0;  // here come some local variables
    long double n;
    unsigned p;

    for (n = numbers, p = picks; p > 0; n--, p--)
        result = result * n / p ;
    return result;
}
```

Here's a sample run of the program in Listing 7.4:

```
Enter the total number of choices on the game card and
the number of picks allowed:
49 6
You have one chance in 1.39838e+007 of winning.
Next two numbers (q to quit): 51 6
You have one chance in 1.80095e+007 of winning.
Next two numbers (q to quit): 38 6
You have one chance in 2.76068e+006 of winning.
Next two numbers (q to quit): q
bye
```

Notice that increasing the number of choices on the game card greatly increases the odds against winning.

Program Notes

The `probability()` function in Listing 7.4 illustrates two kinds of local variables you can have in a function. First, there are the formal parameters (`numbers` and `picks`), which are declared in the function header before the opening brace. Then come the other local variables (`result`, `n`, and `p`). They are declared in between the braces bounding the function definition. The main difference between the formal parameters and the other local variables is that the formal parameters get their values from the function that calls `probability()`, whereas the other variables get values from within the function.

Functions and Arrays

So far the sample functions in this book have been simple, using only the basic types for arguments and return values. But functions can be the key to handling more involved types, such as arrays and structures. Let's take a look now at how arrays and functions get along with each other.

Suppose you use an array to keep track of how many cookies each person has eaten at a family picnic. (Each array index corresponds to a person, and the value of the element corresponds to the number of cookies that person has eaten.) Now you want the total. That's easy to find; you just use a loop to add all the array elements. But adding array elements is such a common task that it makes sense to design a function to do the job. Then you won't have to write a new loop every time you have to sum an array.

Let's consider what the function interface involves. Because the function calculates a sum, it should return the answer. If you keep your cookies intact, you can use a function with a type `int` return value. So that the function knows what array to sum, you want to pass the array name as an argument. And to make the function general so that it is not restricted to an array of a particular size, you pass the size of the array. The only new ingredient here is that you have to declare that one of the formal arguments is an array name. Let's see what that and the rest of the function header look like:

```
int sum_arr(int arr[], int n) // arr = array name, n = size
```

This looks plausible. The brackets seem to indicate that `arr` is an array, and the fact that the brackets are empty seems to indicate that you can use the function with an array of any size. But things are not always as they seem: `arr` is not really an array; it's a pointer! The good news is that you can write the rest of the function just as if `arr` were an array. First, let's use an example to check that this approach works, and then let's look into why it works.

Listing 7.5 illustrates using a pointer as if it were an array name. The program initializes the array to some values and uses the `sum_arr()` function to calculate the sum. Note that the `sum_arr()` function uses `arr` as if it were an array name.

Listing 7.5 **arrfun1.cpp**

```
// arrfun1.cpp -- functions with an array argument
#include <iostream>
const int ArSize = 8;
int sum_arr(int arr[], int n);        // prototype
int main()
{
    using namespace std;
    int cookies[ArSize] = {1,2,4,8,16,32,64,128};
// some systems require preceding int with static to
// enable array initialization

    int sum = sum_arr(cookies, ArSize);
    cout << "Total cookies eaten: " << sum <<  "\n";
    return 0;
}

// return the sum of an integer array
int sum_arr(int arr[], int n)
{
    int total = 0;

    for (int i = 0; i < n; i++)
        total = total + arr[i];
    return total;
}
```

Here is the output of the program in Listing 7.5:

```
Total cookies eaten: 255
```

As you can see, the program works. Now let's look at why it works.

How Pointers Enable Array-Processing Functions

The key to the program in Listing 7.5 is that C++, like C, in most contexts treats the name of an array as if it were a pointer. Recall from Chapter 4, "Compound Types," that C++ interprets an array name as the address of its first element:

```
cookies == &cookies[0]  // array name is address of first element
```

(There are a few exceptions to this rule. First, the array declaration uses the array name to label the storage. Second, applying `sizeof` to an array name yields the size of the whole array, in bytes. Third, as mentioned in Chapter 4, applying the address operator `&` to an array name returns the address of the whole array; for example, `&cookies` would be the address of a 32-byte block of memory if `int` is 4 bytes.)