DIVIDING YOUR PROGRAM into manageable chunks of code is an idea that's fundamental to programming in every language. A function is a basic building block in all C++ programs. So far, you've made use of some of the functions from the standard library, but the only functions you've written yourself are the ones called main(). This chapter is all about defining your own functions with names that you choose.

In this chapter you will learn:

• What a function is, and why you should segment your programs into functions

• How to declare and define functions

• How arguments are passed to a function, and how a value is returned

• What pass-by-value means

• How specifying a parameter as a pointer affects the pass-by-value mechanism

• How using const as a qualifier for a parameter type affects the operation of a function

• What pass-by-reference means, and how you can declare a reference in your program

• How to return a value from a function

• What an inline function is

• The effect of declaring a variable as static within a function

Segmenting Your Programs

All the programs you have written so far have consisted of just one function, main(). As you know, all C++ programs must have a function called main()—it's where program execution starts. However, C++ allows you to include as many other functions in your programs as you need, and I've already used quite a few functions from the standard library in my examples. Defining and using your own functions is just as easy. An illustration of the overall structure of an arbitrary program that has been implemented as several functions is shown in Figure 8-1.
Execution starts here

```cpp
int main() {
    do_this();
    do_that();
    return 0;
}
```

Figure 8-1. Functions calling functions

The sequence in which the functions shown in Figure 8-1 execute is indicated by the numbers on the arrows. Generally, when you call a function at a given point in your program, the code that the function contains executes; when it finishes, program execution continues immediately after the point where the function was called. Any function can call other functions—that's what `main()` does, after all—that may in turn call other functions, so a single function call could result in several functions being executed.

When one function calls another, which calls another, and so on, you have a situation where several functions are still in action, and each function is waiting for the function that it called to return. In Figure 8-1, `main()` calls `do_that()`, which calls `calc_that()`, so all three functions are in progress simultaneously. While the `calc_that()` function is executing, the `do_that()` function is waiting for it to return, and `main()` is waiting for `do_that()` to return.

For a function to execute, the code for it must reside somewhere in memory. Now, it's all very well saying that one function "waits for another to return," but in order for that to happen, something must keep track of from where in memory calls were made, and where functions must return. This information is all recorded and maintained automatically in the **call stack**. The call stack contains information on all the outstanding function calls at any given time, as well as details of the arguments passed to each function. The debugging facilities that come with most C++ development systems usually provide ways for you to view the call stack while your program executes.