```
double totalWeight = count * avgWeight;
cout << "totalWeight=" << totalWeight << endl;
return 0;
}
```

Here a variable of type `int` is multiplied by a variable of type `float` to yield a result of type `double`. This program compiles without error; the compiler considers it normal that you want to multiply (or perform any other arithmetic operation on) numbers of different types.

Not all languages are this relaxed. Some don't permit mixed expressions, and would flag the line that performs the arithmetic in MIXED as an error. Such languages assume that when you mix types you're making a mistake, and they try to save you from yourself. C++ and C, however, assume that you must have a good reason for doing what you're doing, and they help carry out your intentions. This is one reason for the popularity of C++ and C. They give you more freedom. Of course, with more freedom, there's also more opportunity for you to make a mistake.

Automatic Conversions

Let's consider what happens when the compiler confronts such mixed-type expressions as the one in MIXED. Types are considered "higher" or "lower," based roughly on the order shown in Table 2.4.

TABLE 2.4 Order of Data Types

Data Type	Order
`long double`	Highest
`double`	
`float`	
`long`	
`int`	
`short`	
`char`	Lowest

The arithmetic operators such as `+` and `*` like to operate on two operands of the same type. When two operands of different types are encountered in the same expression, the lower-type variable is converted to the type of the higher-type variable. Thus in MIXED, the `int` value of `count` is converted to type `float` and stored in a temporary variable before being multiplied by the `float` variable `avgWeight`. The result (still of type `float`) is then converted to `double` so that it can be assigned to the `double` variable `totalWeight`. This process is shown in Figure 2.9.

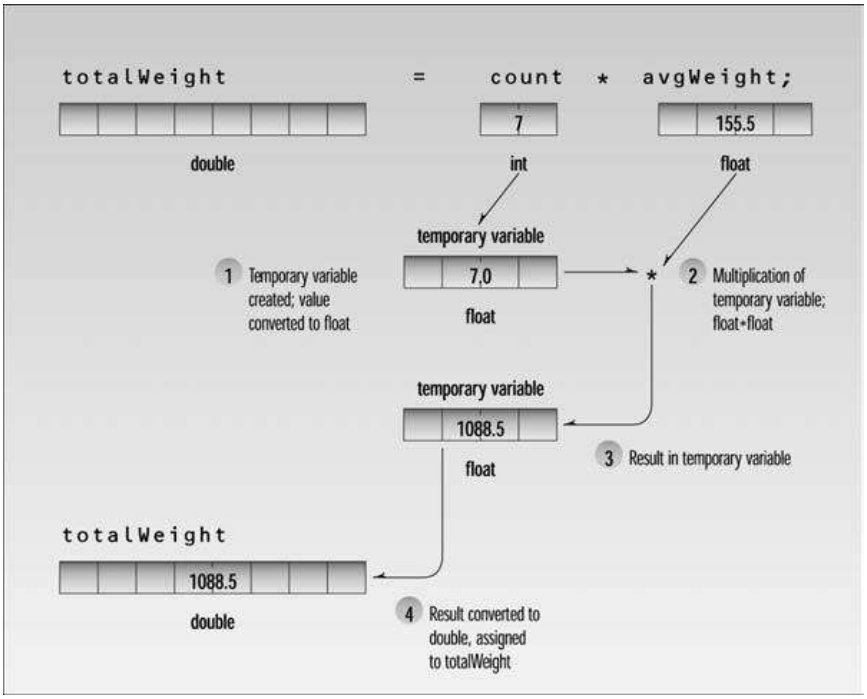

FIGURE 2.9
Data conversion.

These conversions take place invisibly, and ordinarily you don't need to think too much about them; C++ automatically does what you want. However, sometimes the compiler isn't so happy about conversions, as we'll see in a moment. Also, when we start to use objects, we will in effect be defining our own data types. We may want to use these new data types in mixed expressions, just as we use normal variables in mixed expressions. When this is the case, we must be careful to create our own conversion routines to change objects of one type into objects of another. The compiler won't do it for us, as it does here with the built-in data types.

Arithmetic Operators

As you have probably gathered by this time, C++ uses the four normal arithmetic operators `+`, `-`, `*`, and `/` for addition, subtraction, multiplication, and division. These operators work on all the data types, both integer and floating-point. They are used in much the same way that they are used in other languages, and are closely analogous to their use in algebra. However, there are some other arithmetic operators whose use is not so obvious.

The Remainder Operator

There is a fifth arithmetic operator that works only with integer variables (types `char`, `short`, `int`, and `long`). It's called the remainder operator, and is represented by the percent symbol (%). This operator (also called the modulus operator) finds the remainder when one number is divided by another. The REMAIND program demonstrates the effect.

```
// remaind.cpp
// demonstrates remainder operator
#include <iostream>
using namespace std;

int main()
   {
   cout <<  6 % 8 << endl    // 6
        <<  7 % 8 << endl    // 7
        <<  8 % 8 << endl    // 0
        <<  9 % 8 << endl    // 1
        << 10 % 8 << endl;   // 2
   return 0;
   }
```

Here the numbers 6–10 are divided by 8, using the remainder operator. The answers are 6, 7, 0, 1, and 2—the remainders of these divisions. The remainder operator is used in a wide variety of situations. We'll show examples as we go along.

A note about precedence: In the expression

```
cout << 6 % 8
```

the remainder operator is evaluated first because it has higher precedence than the << operator. If it did not, we would need to put parentheses around `6%8` to ensure it was evaluated before being acted on by <<.

Arithmetic Assignment Operators

C++ offers several ways to shorten and clarify your code. One of these is the arithmetic assignment operator. This operator helps to give C++ listings their distinctive appearance.

The following kind of statement is common in most languages.

```
total = total + item;  // adds "item" to "total"
```

In this situation you add something to an existing value (or you perform some other arithmetic operation on it). But the syntax of this statement offends those for whom brevity is important, because the name `total` appears twice. So C++ offers a condensed approach: the arithmetic assignment operator, which combines an arithmetic operator and an assignment operator and

eliminates the repeated operand. Here's a statement that has exactly the same effect as the preceding one.

```
total += item;    // adds "item" to "total"
```

Figure 2.10 emphasizes the equivalence of the two forms.

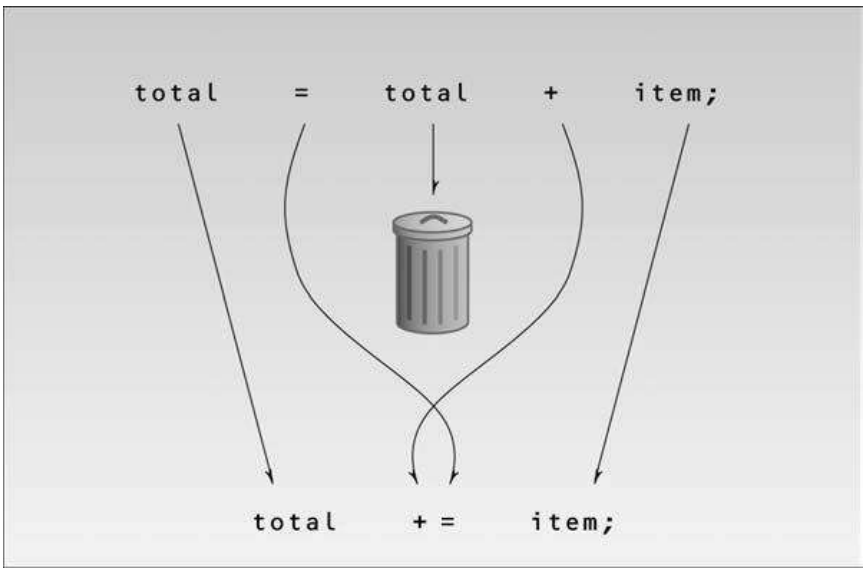

FIGURE 2.10
Arithmetic assignment operator.

There are arithmetic assignment operators corresponding to all the arithmetic operations: +=, -=, *=, /=, and %= (and some other operators as well). The following example shows the arithmetic assignment operators in use:

```
// assign.cpp
// demonstrates arithmetic assignment operators
#include <iostream>
using namespace std;

int main()
   {
   int ans = 27;

   ans += 10;              //same as: ans = ans + 10;
   cout << ans << ", ";
   ans -= 7;               //same as: ans = ans - 7;
   cout << ans << ", ";
   ans *= 2;               //same as: ans = ans * 2;
   cout << ans << ", ";
```

```
ans /= 3;                //same as: ans = ans / 3;
cout << ans << ", ";
ans %= 3;                //same as: ans = ans % 3;
cout << ans << endl;
return 0;
}
```

Here's the output from this program:

```
37, 30, 60, 20, 2
```

You don't need to use arithmetic assignment operators in your code, but they are a common feature of the language; they'll appear in numerous examples in this book.

Increment Operators

Here's an even more specialized operator. You often need to add 1 to the value of an existing variable. You can do this the "normal" way:

```
count = count + 1;   // adds 1 to "count"
```

Or you can use an arithmetic assignment operator:

```
count += 1;   // adds 1 to "count"
```

But there's an even more condensed approach:

```
++count;   // adds 1 to "count"
```

The ++ operator increments (adds 1 to) its argument.

Prefix and Postfix

As if this weren't weird enough, the increment operator can be used in two ways: as a prefix, meaning that the operator precedes the variable; and as a postfix, meaning that the operator follows the variable. What's the difference? Often a variable is incremented within a statement that performs some other operation on it. For example

```
totalWeight = avgWeight * ++count;
```

The question here is this: Is the multiplication performed before or after `count` is incremented? In this case `count` is incremented first. How do we know that? Because prefix notation is used: `++count`. If we had used postfix notation, `count++`, the multiplication would have been performed first, then `count` would have been incremented. This is shown in Figure 2.11.

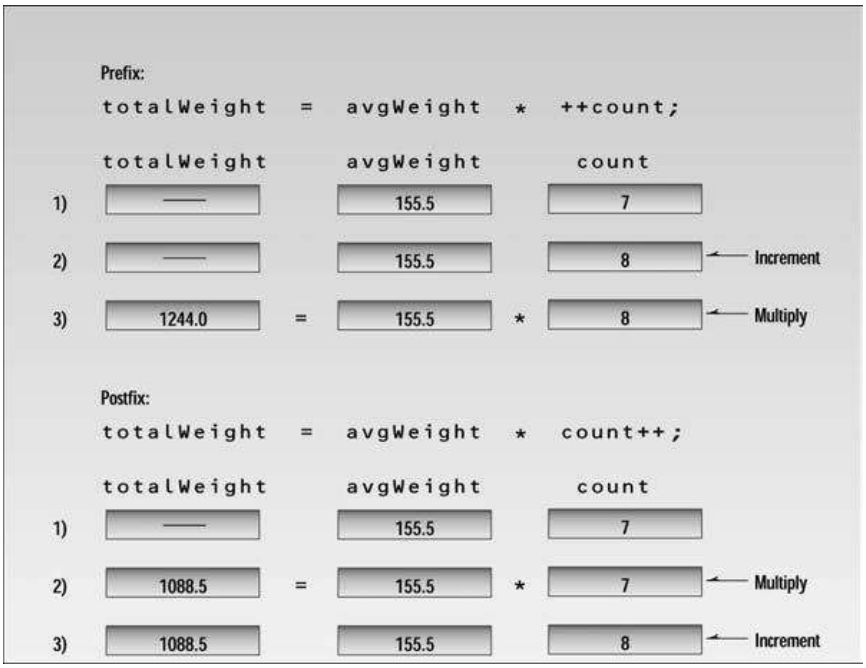

FIGURE 2.11
The increment operator.

Here's an example that shows both the prefix and postfix versions of the increment operator:

```
// increm.cpp
// demonstrates the increment operator
#include <iostream>
using namespace std;

int main()
   {
   int count = 10;

   cout << "count=" << count << endl;    //displays 10
   cout << "count=" << ++count << endl; //displays 11 (prefix)
   cout << "count=" << count << endl;    //displays 11
   cout << "count=" << count++ << endl; //displays 11 (postfix)
   cout << "count=" << count << endl;    //displays 12
   return 0;
   }
```

Here's the program's output:

```
count=10
count=11
```

```
count=11
count=11
count=12
```

The first time `count` is incremented, the prefix `++` operator is used. This causes the increment to happen at the beginning of the statement evaluation, before the output operation has been carried out. When the value of the expression `++count` is displayed, it has already been incremented, and `<<` sees the value 11. The second time `count` is incremented, the postfix `++` operator is used. When the expression `count++` is displayed, it retains its unincremented value of 11. Following the completion of this statement, the increment takes effect, so that in the last statement of the program we see that `count` has acquired the value 12.

The Decrement (--) Operator

The decrement operator, `--`, behaves very much like the increment operator, except that it subtracts 1 from its operand. It too can be used in both prefix and postfix forms.

Library Functions

Many activities in C++ are carried out by library functions. These functions perform file access, mathematical computations, and data conversion, among other things. We don't want to dig too deeply into library functions before we explain how functions work (see Chapter 5), but you can use simple library functions without a thorough understanding of their operation.

The next example, SQRT, uses the library function `sqrt()` to calculate the square root of a number entered by the user.

```
// sqrt.cpp
// demonstrates sqrt() library function
#include <iostream>              //for cout, etc.
#include <cmath>                 //for sqrt()
using namespace std;

int main()
   {
   double number, answer;        //sqrt() requires type double

   cout << "Enter a number: ";
   cin >> number;                //get the number
   answer = sqrt(number);        //find square root
   cout << "Square root is "
    << answer << endl;           //display it
   return 0;
   }
```

The program first obtains a number from the user. This number is then used as an argument to the `sqrt()` function, in the statement

```
answer = sqrt(number);
```

An argument is the input to the function; it is placed inside the parentheses following the function name. The function then processes the argument and returns a value; this is the output from the function. In this case the return value is the square root of the original number. Returning a value means that the function expression takes on this value, which can then be assigned to another variable—in this case `answer`. The program then displays this value. Here's some output from the program:

```
Enter a number: 1000
Square root is 31.622777
```

Multiplying 31.622777 by itself on your pocket calculator will verify that this answer is pretty close.

The arguments to a function, and their return values, must be the correct data type. You can find what these data types are by looking at the description of the library function in your compiler's help file, which describes each of the hundreds of library functions. For `sqrt()`, the description specifies both an argument and a return value of type `double`, so we use variables of this type in the program.

Header Files

As with `cout` and other such objects, you must `#include` a header file that contains the declaration of any library functions you use. In the documentation for the `sqrt()` function, you'll see that the specified header file is CMATH. In SQRT the preprocessor directive

```
#include <cmath>
```

takes care of incorporating this header file into our source file.

If you don't include the appropriate header file when you use a library function, you'll get an error message like this from the compiler: *'sqrt' unidentified identifier*.

Library Files

We mentioned earlier that various files containing library functions and objects will be linked to your program to create an executable file. These files contain the actual machine-executable code for the functions. Such library files often have the extension .LIB. The `sqrt()` function is found in such a file. It is automatically extracted from the file by the linker, and the proper connections are made so that it can be called (that is, invoked or accessed) from the SQRT program. Your compiler takes care of all these details for you, so ordinarily you don't need to worry about the process. However, you should understand what these files are for.

Header Files and Library Files

The relationship between library files and header files can be confusing, so let's review it. To use a library function like `sqrt()`, you must link the library file that contains it to your program. The appropriate functions from the library file are then connected to your program by the linker.

However, that's not the end of the story. The functions in your source file need to know the names and types of the functions and other elements in the library file. They are given this information in a header file. Each header file contains information for a particular group of functions. The functions themselves are grouped together in a library file, but the information about them is scattered throughout a number of header files. The IOSTREAM header file contains information for various I/O functions and objects, including `cout`, while the CMATH header file contains information for mathematics functions such as `sqrt()`. If you were using string functions such as `strcpy()`, you would include STRING.H, and so on.

Figure 2.12 shows the relationship of header files and library files to the other files used in program development.

The use of header files is common in C++. Whenever you use a library function or a predefined object or operator, you will need to use a header file that contains appropriate declarations.

Two Ways to Use `#include`

You can use `#include` in two ways. The angle brackets `<` and `>` surrounding the filenames IOSTREAM and CMATH in the SQRT example indicate that the compiler should begin searching for these files in the standard INCLUDE directory. This directory, which is traditionally called INCLUDE, holds the header files supplied by the compiler manufacturer for the system.

Instead of angle brackets around the filename, you can also use quotation marks, as in

```
#include "myheader.h"
```

Quotation marks instruct the compiler to begin its search for the header file in the current directory; this is usually the directory that contains the source file. You normally use quotation marks for header files you write yourself (a situation we'll explore in Chapter 13, "Multifile Programs"). Quotation marks or angle brackets work in any case, but making the appropriate choice speeds up the compilation process slightly by giving the compiler a hint about where to find the file.

Appendix C, "Microsoft Visual C++," and Appendix D, "Borland C++Builder," explain how to handle header files with specific compilers.

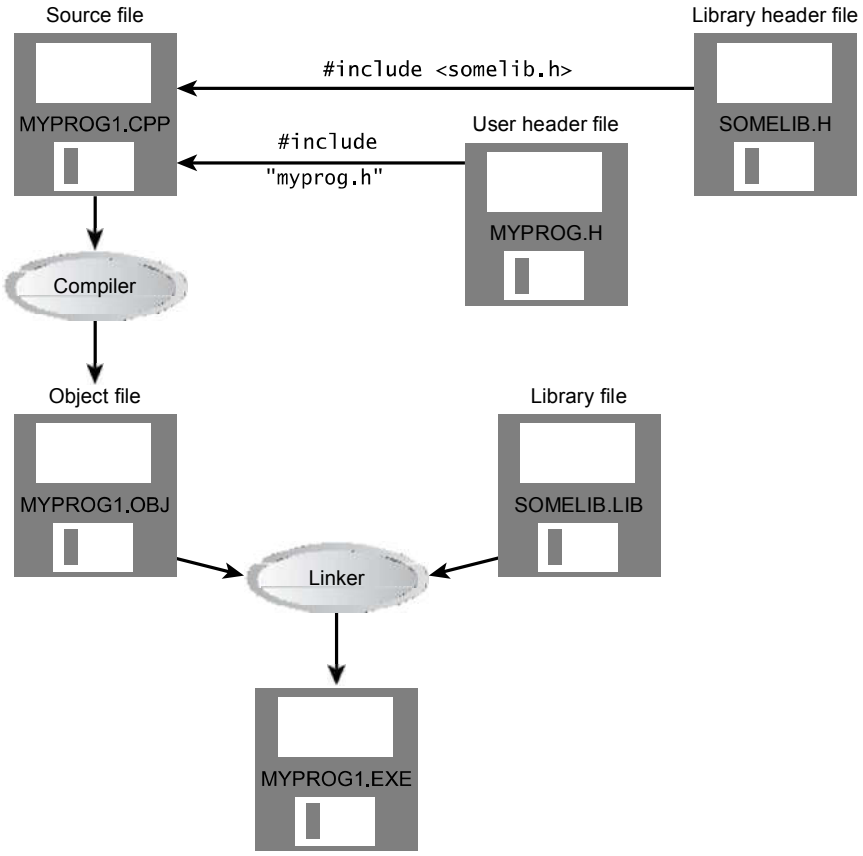

FIGURE 2.12
Header and library files.

Summary

In this chapter we've learned that a major building block of C++ programs is the function. A function named `main()` is always the first one executed when a program is executed.

A function is composed of statements, which tell the computer to do something. Each statement ends with a semicolon. A statement may contain one or more expressions, which are sequences of variables and operators that usually evaluate to a specific value.

Output is most commonly handled in C++ with the `cout` object and `<<` insertion operator, which together cause variables or constants to be sent to the standard output device—usually the screen. Input is handled with `cin` and the extraction operator `>>`, which cause values to be received from the standard input device—usually the keyboard.