Floating-Point Numbers

Now that you have seen the complete line of C++ integer types, let's look at the floating-point types, which compose the second major group of fundamental C++ types. These numbers let you represent numbers with fractional parts, such as the gas mileage of an M1 tank (0.56 MPG). They also provide a much greater range in values. If a number is too large to be represented as type `long`—for example, the number of bacterial cells in a human body (estimated to be greater than 100,000,000,000,000)—you can use one of the floating-point types.

With floating-point types, you can represent numbers such as 2.5 and 3.14159 and 122442.32—that is, numbers with fractional parts. A computer stores such values in two parts. One part represents a value, and the other part scales that value up or down. Here's an analogy. Consider the two numbers 34.1245 and 34124.5. They're identical except for scale. You can represent the first one as 0.341245 (the base value) and 100 (the scaling factor). You can represent the second as 0.341245 (the same base value) and 100,000 (a bigger scaling factor). The scaling factor serves to move the decimal point, hence the term *floating-point*. C++ uses a similar method to represent floating-point numbers internally, except it's based on binary numbers, so the scaling is by factors of 2 instead of by factors of 10. Fortunately, you don't have to know much about the internal representation. The main points are that floating-point numbers let you represent fractional, very large, and very small values, and they have internal representations much different from those of integers.

Writing Floating-Point Numbers

C++ has two ways of writing floating-point numbers. The first is to use the standard decimal-point notation you've been using much of your life:

```
12.34             // floating-point
939001.32         // floating-point
0.00023           // floating-point
8.0               // still floating-point
```

Even if the fractional part is 0, as in 8.0, the decimal point ensures that the number is represented in floating-point format and not as an integer. (The C++ Standard does allow for implementations to represent different locales—for example, providing a mechanism for using the European method of using a comma instead of a period for the decimal point. However, these choices govern how the numbers can appear in input and output, not in code.)

The second method for representing floating-point values is called E notation, and it looks like this: `3.45E6`. This means that the value 3.45 is multiplied by 1,000,000; the E6 means 10 to the 6th power, which is 1 followed by 6 zeros. Thus `3.45E6` means 3,450,000. The 6 is called an *exponent*, and the 3.45 is termed the *mantissa*. Here are more examples:

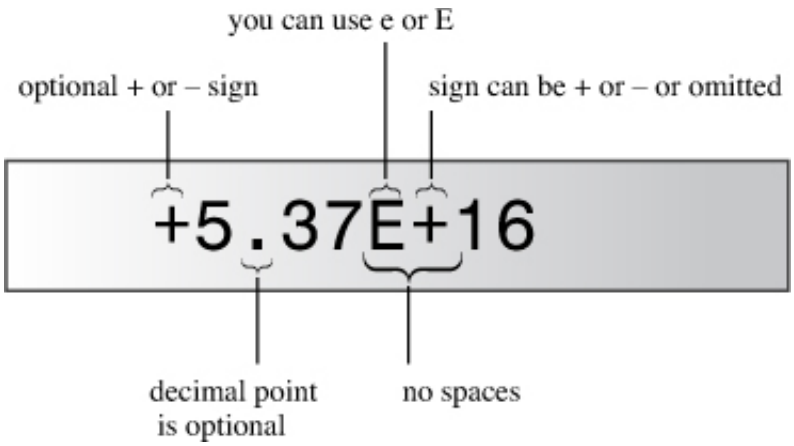
```
2.52e+8           // can use E or e, + is optional
8.33E-4           // exponent can be negative
7E5               // same as 7.0E+05
-18.32e13         // can have + or - sign in front
```

```
1.69e12             // 2010 Brazilian public debt in reais
5.98E24             // mass of earth in kilograms
9.11e-31            // mass of an electron in kilograms
```

As you might have noticed, E notation is most useful for very large and very small numbers.

E notation guarantees that a number is stored in floating-point format, even if no decimal point is used. Note that you can use either `E` or `e`, and the exponent can have a positive or negative sign (see Figure 3.3). However, you can't have spaces in the number, so, for example, `7.2 E6` is invalid.

Figure 3.3. E notation.



To use a negative exponent means to divide by a power of 10 instead of to multiply by a power of 10. So 8.33E-4 means 8.33 / 10^4, or 0.000833. Similarly, the electron mass 9.11e-31 kg means 0.000000000000000000000000000000911 kg. Take your choice. (Incidentally, note that 911 is the usual emergency telephone number in the United States and that telephone messages are carried by electrons. Coincidence or scientific conspiracy? You be the judge.) Note that –8.33E4 means –83300. A sign in front applies to the number value, and a sign in the exponent applies to the scaling.

Note

The form `d.dddE+n` means move the decimal point `n` places to the right, and the form `d.dddE-n` means move the decimal point `n` places to the left. This moveable decimal point is the origin of the term "floating-point."

Floating-Point Types

Like ANSI C, C++ has three floating-point types: `float`, `double`, and `long double`. These types are described in terms of the number of significant figures they can represent and the minimum allowable range of exponents. *Significant figures* are the meaningful digits in a number. For example, writing the height of Mt. Shasta in California as 14,179 feet uses five significant figures, for it specifies the height to the nearest foot. But writing the height of Mt. Shasta as about 14,000 feet tall uses two significant figures, for the result is rounded to the nearest thousand feet; in this case, the remaining three digits are just placeholders. The number of significant figures doesn't depend on

the location of the decimal point. For example, you can write the height as 14.179 thousand feet. Again, this uses five significant digits because the value is accurate to the fifth digit.

In effect, the C and C++ requirements for significant digits amount to `float` being at least 32 bits, `double` being at least 48 bits and certainly no smaller than `float`, and `long double` being at least as big as `double`. All three can be the same size. Typically, however, `float` is 32 bits, `double` is 64 bits, and `long double` is 80, 96, or 128 bits. Also the range in exponents for all three types is at least –37 to +37. You can look in the `cfloat` or `float.h` header files to find the limits for your system. (`cfloat` is the C++ version of the C `float.h` file.) Here, for example, are some annotated entries from the `float.h` file for Borland C++Builder:

```
// the following are the minimum number of significant digits
#define DBL_DIG 15         // double
#define FLT_DIG 6          // float
#define LDBL_DIG 18        // long double

// the following are the number of bits used to represent the mantissa
#define DBL_MANT_DIG     53
#define FLT_MANT_DIG     24
#define LDBL_MANT_DIG    64

// the following are the maximum and minimum exponent values
#define DBL_MAX_10_EXP   +308
#define FLT_MAX_10_EXP   +38
#define LDBL_MAX_10_EXP  +4932

#define DBL_MIN_10_EXP   -307
#define FLT_MIN_10_EXP   -37
#define LDBL_MIN_10_EXP  -4931
```

Listing 3.8 examines types `float` and `double` and how they can differ in the precision to which they represent numbers (that's the significant figure aspect). The program previews an `ostream` method called `setf()` from Chapter 17, "Input, Output, and Files." This particular call forces output to stay in fixed-point notation so that you can better see the precision. It prevents the program from switching to E notation for large values and causes the program to display six digits to the right of the decimal. The arguments `ios_base::fixed` and `ios_base::floatfield` are constants provided by including `iostream`.

Listing 3.8. `floatnum.cpp`

```
// floatnum.cpp -- floating-point types
#include <iostream>
int main()
{
    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield); // fixed-point
    float tub = 10.0 / 3.0;     // good to about 6 places
    double mint = 10.0 / 3.0;   // good to about 15 places
    const float million = 1.0e6;

    cout << "tub = " << tub;
```

```
    cout << ", a million tubs = " << million * tub;
    cout << ",\nand ten million tubs = ";
    cout << 10 * million * tub << endl;

    cout << "mint = " << mint << " and a million mints = ";
    cout << million * mint << endl;
    return 0;
}
```

Here is the output from the program in Listing 3.8:

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

Program Notes

Normally `cout` drops trailing zeros. For example, it would display 3333333.250000 as 3333333.25. The call to `cout.setf()` overrides that behavior, at least in new implementations. The main thing to note in Listing 3.8 is how `float` has less precision than `double`. Both `tub` and `mint` are initialized to 10.0 / 3.0. That should evaluate to 3.3333333333333333...(etc.). Because `cout` prints six figures to the right of the decimal, you can see that both `tub` and `mint` are accurate that far. But after the program multiplies each number by a million, you see that `tub` diverges from the proper value after the seventh three. `tub` is good to seven significant figures. (This system guarantees six significant figures for `float`, but that's the worst-case scenario.) The type `double` variable, however, shows 13 threes, so it's good to at least 13 significant figures. Because the system guarantees 15, this shouldn't surprise you. Also note that multiplying a million `tubs` by 10 doesn't quite result in the correct answer; this again points out the limitations of `float` precision.

The `ostream` class to which `cout` belongs has class member functions that give you precise control over how the output is formatted—field widths, places to the right of the decimal point, decimal form or E form, and so on. Chapter 17 outlines those choices. This book's examples keep it simple and usually just use the `<<` operator. Occasionally, this practice displays more digits than necessary, but that causes only aesthetic harm. If you do mind, you can skim Chapter 17 to see how to use the formatting methods. Don't, however, expect to fully follow the explanations at this point.

Reading Include Files

The include directives found at the top of C++ source files often take on the air of a magical incantation; novice C++ programmers learn, through reading and experience, which header files add particular functionalities, and they include them solely to make their programs work. Don't rely on the include files only as a source of mystic and arcane knowledge; feel free to open them up and read them. They are text files, so you can read them easily. All the files you include in your programs exist on your computer or in a place where your computer can use them. Find the includes you use and see what they contain. You'll quickly see that the source and header files you use are an excellent source of knowledge and information—in some cases, the best documentation available. Later, as you progress into more complex inclusions and begin to use other, nonstandard libraries in your applications, this habit will serve you well.

Floating-Point Constants

When you write a floating-point constant in a program, in which floating-point type does the program store it? By default, floating-point constants such as 8.24 and 2.4E8 are type `double`. If you want a constant to be type `float`, you use an `f` or `F` suffix. For type `long double`, you use an `l` or `L` suffix. (Because the lowercase `l` looks a lot like the digit 1, the uppercase `L` is a better choice.) Here are some samples:

```
1.234f          // a float constant
2.45E20F        // a float constant
2.345324E28     // a double constant
2.2L            // a long double constant
```

Advantages and Disadvantages of Floating-Point Numbers

Floating-point numbers have two advantages over integers. First, they can represent values between integers. Second, because of the scaling factor, they can represent a much greater range of values. On the other hand, floating point operations usually are slightly slower than integer operations, and you can lose precision. Listing 3.9 illustrates the last point.

Listing 3.9. `fltadd.cpp`

```
// fltadd.cpp -- precision problems with float
#include <iostream>
int main()
{
    using namespace std;
    float a = 2.34E+22f;
    float b = a + 1.0f;

    cout << "a = " << a << endl;
    cout << "b - a = " << b - a << endl;
    return 0;
}
```

The program in Listing 3.9 takes a number, adds 1, and then subtracts the original number. That should result in a value of 1. Does it? Here is the output from the program in Listing 3.9 for one system:

```
a = 2.34e+022
b - a = 0
```

The problem is that 2.34E+22 represents a number with 23 digits to the left of the decimal. By adding 1, you are attempting to add 1 to the 23rd digit in that number. But type `float` can represent only the first 6 or 7 digits in a number, so trying to change the 23rd digit has no effect on the value.

Classifying Data Types

C++ brings some order to its basic types by classifying them into families. Types `signed char`, `short`, `int`, and `long` are termed *signed integer* types. C++11 adds `long long` to that list. The unsigned versions are termed *unsigned integer* types. The `bool`, `char`, `wchar_t`, signed integer, and unsigned integer types together are termed *integral* types or *integer* types. C++11 adds `char16_t` and `char32_t` to that list. The `float`, `double`, and `long double` types are termed *floating-point* types. Integer and floating-point types are collectively termed *arithmetic* types.