3.3 Floating-point arithmetic

Concepts

Real and floating-point numbers
Significant digits and bits
IEEE binary floating-point standard
Intel 8087 family coprocessors
8087 emulation by software
Borland C++ **`float`**, **`double`**, and **`long double`** types
Nonnormalized numbers, infinities and *NaN*s
Arithmetic and rounding
The need for arithmetic testing
Input/output

Real and floating-point numbers

Using decimal notation you can represent a nonzero real number in the form $x = \pm 10^E \times d_0.d_1d_2\ldots$, where the *exponent* E is an integer; the *significand* $d_0.d_1d_2\ldots$ is an infinite sequence of digits d_k in the range $0 \le d_k < 10$, with *leading digit* $d_0 \neq 0$. The value of x is

$$x = \pm 10^E \sum_{k=0}^{\infty} 10^{-k} d_k .$$

For computation, you approximate x by truncating or rounding the significand to n *significant* digits:

$$x \approx \pm 10^E \times d_0.d_1\ldots d_{n-1} = \pm 10^E \sum_{k=0}^{n-1} 10^{-k} d_k .$$

You choose n small enough for convenient computation, but large enough for the required precision. Approximations like this, and the number zero, are called *floating-point* numbers. If you want to record floating-point numbers on a physical medium, you

must also limit the exponent to a specific range $E_{min} \le E \le E_{max}$. ($E_{min}$ is generally negative and E_{max} positive.)

It's instructive to contrast the sets of real and floating-point numbers available for reasoning and calculating. Consider, for example, floating-point numbers with three significant digits and exponents E in the range $-8 \le E \le 8$. The two smallest positive floating-point numbers in this system are 1.00×10^{-8} and 1.01×10^{-8}; the two largest are 9.98×10^{8} and 9.99×10^{8}. The negative floating-point numbers mirror these. In contrast, real numbers can have arbitrarily large or small magnitudes. Moreover, the real numbers form a *continuum*: there are no gaps. But the floating-point numbers form a *finite discrete* set: nonzero floating-point numbers are bounded away from zero, those nearest zero are separated by at least $0.01 \times 10^{-8} = 10^{-10}$, and those largest in magnitude can be separated by as much as $0.01 \times 10^{8} = 10^{6}$. When you use floating-point numbers, be careful what you mean by precision, because attainable precision depends on the magnitude of the numbers you're using.

With a floating-point system like the one just considered, you may be tempted to use numbers like $u = 0.12 \times 10^{-8}$. But u isn't admissible, because its leading digit is zero. Numbers like u convey less precision (fewer significant digits) than those that really belong to the system. Sometimes numbers like u are in fact useful; then they're called *nonnormalized* floating-point numbers.

Computers use binary notation, not decimal. Using binary notation for the significand, a nonzero real number has the form $x = \pm 2^{E} \times b_0.b_1b_2\ldots$, where the *exponent* E is an integer; the *significand* $b_0.b_1b_2\ldots$ is an infinite sequence of bits b_k in the range $0 \le b_k \le 1$, with *leading bit* $b_0 \ne 0$. The value of x is

$$x = \pm 2^{E} \sum_{k=0}^{\infty} 2^{-k} b_k .$$

For computation, you obtain a binary floating-point system by restricting the significand to n significant bits and the exponent to a range $E_{min} \le E \le E_{max}$.

IEEE binary floating-point standard

In the early days of computing, many different binary floating-point systems were used for numerical computation, as well as some decimal systems and some others. That was awkward because results computed by a given algorithm would vary when transferred to a different computer or programming language.

The American National Standards Institute (ANSI) has approved a floating-point arithmetic standard. Languages and hardware that adhere to this standard produce the same results for the same arithmetic problem. The standard was prepared by a committee of the Institute for Electrical and Electronics Engineers (IEEE), and is described in its document [21].

The Intel 8087 numeric coprocessor family supports the IEEE standard. These components provide the basis for most numerical computation by IBM-compatible personal computers, the target machines for Borland C++. If no coprocessor is installed, PC software often emulates one. For example, normally compiled Borland C++ programs will use an 8087 family coprocessor for numerical computations if one is installed; if not, a rather involved software system will perform the computations as though an 8087 were in use. The only major difference is speed: the emulated computations may take as much as ten times as long to complete.

The IEEE standard includes definitions of three floating-point data types that are implemented by the 8087 coprocessors and Borland C++: **`float`**, **`double`**, and **`long double`**. (Their IEEE names are *single*, *double*, and *extended*.) They differ in the number of bytes allocated to store a number, and consequently in the number of bits allocated to exponent and significand. These and other relevant data are displayed in Figure 4.2.1. The following discussion explains the entries in that figure. Many entries and a few other related data are given with full precision by mnemonic macro constants in Borland C++ header file **`Float.H`**. (These values may disagree occasionally with Borland C++ manuals. The **`Float.H`** constants are correct.)

For example, a **`double`** variable requires 8 bytes—64 bits—of storage. As shown in Figure 4.2.2, the first bit is the sign (0 for + and 1 for −). The next 11 bits store the exponent, and the final 52 store the significand. The exponent bits will accommodate up to $2^{11} = 2048$ different exponents E. They are apportioned between positive and negative E values by storing the sum e of E and the *bias* 1023. The inequality $0 \leq$

$e = E + 1023 < 2048$ implies $-1023 \le E < 1025$. But one of these bit patterns, $e = 1024$, is reserved for special use with infinities and *NaN*s (discussed later). Thus the valid **`double`** exponents lie in the range $-1023 \le E \le 1023$.

	`float`	**`double`**	**`long double`**
Bytes for number	4	8	10
Bits for significand	23 (+1)	52 (+1)	63 (+1)
Bits for exponent	8	11	16
Exponent bias	127	1023	16383
Significant digits	7	15	19
Largest positive number $\approx$	3.4×10^{38}	1.8×10^{308}	1.2×10^{4932}
Smallest positive number $\approx$	1.2×10^{-38}	2.2×10^{-308}	3.4×10^{-4932}
$\varepsilon \approx$	1.2×10^{-7}	2.2×10^{-16}	1.1×10^{-19}

Figure 4.2.1 Types **`float`**, **`double`**, and **`long double`**

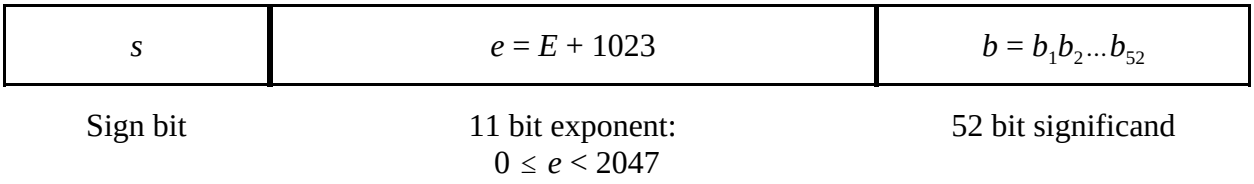

Figure 4.2.2 A **`double`** variable

The largest positive **`double`** value is binary $1.11...1 \times 2^{1023} \approx 2 \times 2^{1023} = 2^{1024} \approx 1.8 \times 10^{308}$. The smallest positive value is binary $1.00...01 \times 2^{-1023} \approx 2^{-1023} \approx 2.2 \times 10^{-308}$. (You'll see later that the binary value $1.00...0 \times 2^{-1023}$ isn't allowed.)

A valid nonzero floating-point number is normalized: the leading bit b_0 of the significand $b_0.b_1b_2...$ is always 1. Therefore, it's not necessary to store b_0. The

remaining 52 bits of a **double** variable are bits $b_1 b_2 \ldots b_{52}$ of the significand. A **double** value has 53 significant bits.

Under this scheme, the value $1.00\ldots \times 2^{-1023}$ would be stored with $s = 0$, $e = -1023 + 1023 = 0$, and $f = 00\ldots0 = 0$ (using the Figure 4.2.2 notation). All bits would be zero. But IEEE reserves that particular bit pattern, which would otherwise have represented the smallest positive value, to represent zero. The corresponding pattern with $s = 1$ represents -0. The IEEE standard has some rules and suggestions regarding the use of -0; they're beyond the scope of this book.

The largest integer that can be represented as a **double** value without round-off is binary $1.11\ldots1 \times 2^{52} = 2^{53} - 1 \approx 9.0 \times 10^{15}$, a 16-digit number. Some but not all 16-digit numbers are representable without round-off. This means that **double** values have 15 significant digits in general, but sometimes 16.

The first floating-point number larger than 1 is $1 + \varepsilon =$ binary $1.00\ldots01 \times 2^0$, which gives $\varepsilon = 2^{-52} \approx 2.2 \times 10^{-16}$. The number ε, tabulated in Figure 4.2.1, is sometimes used in numerical analysis calculations as a threshold: numbers δ with magnitude $< \varepsilon$ are regarded as zero, because **double** arithmetic yields $1 + \delta = 1$.

The following table shows how all possible bit patterns of **double** variables are interpreted. Figure 4.2.2 showed a sign bit s, followed by an 11-bit exponent $e = E + 1023$, then a 52 bit significand $b = b_1 b_2 \ldots b_{52}$.

e	b	*Interpretation*	
0	0	$(-1)^s\, 0$	(Zero values)
0	$\neq 0$	$(-1)^s\, 0.b \times 2^{-1022}$	(Nonnormalized numbers)
255	0	$(-1)^s \infty$	(Infinite values)
255	$\neq 0$	*NaN*	(*N*ot *a N*umber)
All others		$(-1)^s\, 1.b \times 2^E$	(Normal **double** values)

The IEEE standard requires that nonnormalized numbers be interpreted as shown, but discourages their use. The infinite values and *NaN*s are used in reporting and handling error situations like attempts to divide by zero or take the square root of a negative number. Their use is described in somewhat more detail later, in Section 3.7.

All entries in the **`double`** column of Figure 4.2.1 have been explained. You can mimic this discussion for the **`float`** and **`long double`** types.

Arithmetic and rounding

Following the IEEE standard, an 8087-family coprocessor implements the standard arithmetic operations:

$x + y$	$-x$	$x < y$	$x \times y$
$x - y$	$\lvert x \rvert$	round(x)	x / y .

Further, it will extract the exponent or significand from a floating-point number, and multiply by a specified power of 2 (by manipulating the exponent). Borland C++ either uses the coprocessor to perform these operations or else invokes its own routines, which emulate the coprocessor.

In order to ensure correctness of all bits of the result of a floating-point operation, the coprocessor performs every computation with some extra bits, then rounds the result. Four rounding methods are provided:

(0) round to nearest
(1) round up
(2) round down,
(3) round toward zero (chop).

If a result computed with extra bits is exactly halfway between its nearest floating-point neighbors, and the coprocessor is rounding to nearest, it selects the neighbor with least significant bit zero. About half the time this rounds up, and half the time down, so some round-off errors may cancel.

The coprocessor provides a mechanism for controlling the rounding method: you store a coded *control word* in a coprocessor register. In turn, Borland C++ provides Library function

```
unsigned _control87(unsigned NewCW, unsigned Mask)
```

to set the control word. It will set the control word bits that correspond to the 1 bits in **`Mask`**. You must place the new values in the corresponding bits of **`NewCW`**. Any

remaining `NewCW` bits are ignored. The function returns the modified control word. You'll find the `_control87` prototype in Borland C++ header file `Float.H`, along with many mnemonic macro constants that you can use for `Mask` and `NewCW`. For example, the constant `MCW_RC` is the mask for the rounding control bits, and the constant `RC_NEAR` contains the bits to be set for rounding to nearest. To select that rounding method, execute `_control87(RC_NEAR,MCW_RC)`. Rounding to nearest is the coprocessor's default method. To select a different method, use `Float.H` constant `RC_DOWN`, `RC_UP`, or `RC_CHOP`.

You'll probably never have to deal with rounding problems, unless you have to troubleshoot a disagreement between results of a program when run under Borland C++ and under another C system, or perhaps modify and extend features of the Borland C++ floating-point library functions. So your lack of information about rounding will probably not be serious. But this information gap is more extensive. To what extent *does* the emulator produce the same results as the coprocessor?

Further, how do you *know* that the coprocessor really produces accurate results? The Intel literature [22] is comprehensive enough to let you control the coprocessor. (This requires low-level C++ or assembly language techniques.) But it doesn't document the underlying algorithms or any definitive tests. A software package called *Paranoia* has been developed to test floating-point arithmetic, and is available in a C version [25]. The results of a comprehensive test should be published in an accessible source. Unfortunately, that's beyond the scope of this book.

Input/output

C++ provides several methods for inputting floating-point values. They're familiar, so it's not necessary to discuss them in detail here. What's really involved in converting from a decimal floating-point input to binary notation? How do you do it by hand? Consider, as an example, the input 0.123. Write it as a ratio of two integers, first in decimal, then in binary:

$$0.123 = \frac{123}{1000} = \frac{1111011}{1111101000}.$$

Now do long division, in binary:

```
                     0.000111.....
1111101000 | 1111011.00000000
           |  111110.1000
           ---------------------
             | 111100 10000
             |  11111 01000
             -------------------
               | 11101 010000
               |  1111 101000
               -----------------
                   ⋮
```

Finally, normalize: 0.123 = binary 0.000111... = binary $1.11\ldots \times 2^{-4}$.

Formatting floating-point *output* is often unexpectedly difficult. C++ provides comprehensive facilities for this task—too many to consider here. Unfortunately, the facilities provided by any language always seem to fall just short of what's required. For instance, it was necessary to include a special formatting routine **`Show`** in the MSP **`Scalar`** module. Output of numerical data is discussed in great detail in Section 3.7 of the author's earlier book [47] on Borland C.