3.2 Integer arithmetic

Concepts

Integer types
+, –, and * operators
Modular arithmetic
Order and absolute value
Integer division
Additional functions
Formatting output
Arbitrarily long integers

Although this chapter is mostly concerned with real and complex numbers, it starts by considering the integers. There are three reasons for this. First, integers, after all, form the basis of the real number system. Second, all computations involve counting, hence integer operations, to some extent. Third, some features of integer arithmetic yield results that aren't easy to predict unless you've taken the time at least once to study them in detail. This section surveys the various integer types available to C++ programmers, stressing their similarities. It considers the standard integer operations and functions supplied with C++, and a few more that you may find useful. It concludes by mentioning some aspects of integer arithmetic that are not supported by C++ or this book's MSP software package.

C++ integer types

Borland C++ provides six integer types, which you can distinguish by their ranges of values:

Type	*Full type name*	*Range*	*Macro*
`char`	`signed char`	$-2^7 \ldots 2^7 - 1 = 127$	
`int`	`signed int`	$-2^{15} \ldots 2^{15} - 1 = 32767$	= `MAXINT`
`long`	`signed long int`	$-2^{31} \ldots 2^{31} - 1 = 2147483647$	= `MAXLONG`
	`unsigned char`	$0 \ldots 2^8 - 1 = 255$	
	`unsigned int`	$0 \ldots 2^{16} - 1 = 65535$	
	`unsigned long`	$0 \ldots 2^{32} - 1 = 4294967295$	

The two macro constants are provided for your convenience in Borland C++ header file `Values.H`. (The ranges in this table are not C++ standards, but depend on the machine for which the language is implemented.) The storage required for an integer depends on the type. For Borland C++ the requirements are

`char` types: 1 byte
`int` types: 2 bytes
`long` types: 4 bytes.

`int` arithmetic is generally faster than `long` arithmetic, but because of the architecture of the Intel 8086-family machines on which it runs, Borland C++ performs `char` arithmetic no faster than `int`. Most C++ programs, including virtually all in this book, use `int` arithmetic, resorting to `long` variables only when extremely large sets must be counted—for example, the number of bytes in the text files containing this chapter.

Addition, subtraction, and multiplication

The standard integer arithmetic operations are provided for each C++ integer type. Addition, subtraction, and multiplication are really *modular* arithmetic, with these moduli:

`char` types: mod 2^8
`int` types: mod 2^{16}
`long` types: mod 2^{32}.

To perform an arithmetic operation mod M, you carry out the operation as usual, obtaining an intermediate value v; then you divide v by M, return the remainder r as the result, and discard the quotient q. The following equations hold among these quantities:

$$\frac{v}{M} = q + \frac{r}{M} \qquad v = qM+r \ .$$

r is adjusted to lie in the appropriate range: $-\frac{1}{2}M \le r < \frac{1}{2}M$ for the **signed** types, and $0 \le r < M$ for the **unsigned** ones.

For example, the program fragment

```
int I = MAXINT;
I = 2*I;
cout << "I = " << I;
```

outputs **I = -2**. (Since **MAXINT** $= 2^{15} - 1$, the intermediate value v is $2(2^{15} - 1) = 2^{16} - 2 = M - 2$, so the quotient is 1 and the remainder is -2, the output.) Here's an analogous example with an unsigned type: the fragment

```
unsigned long u = MAXLONG + 2;
u = 2*u;
cout << "u = " << u;
```

outputs **u = 2**. (Since **MAXLONG** $= 2^{31} - 1$, the intermediate value v is $2(2^{31} + 1) = 2^{32} + 2 = M + 2$, so the quotient is 1 and the remainder is 2, the output.) You can construct similar fragments for the other four integer types. (You'll have to output **char** values using **printf** format string **"%d"**; otherwise **char** output defaults to characters, not numerals.)

These examples should convince you that there's no such thing as overflow in integer arithmetic. But if the numbers get too large for the selected integer type, you may get unexpected results!

Order and absolute value

When you study arithmetic mod M, you generally regard the range of values as circular, like the examples of Figure 4.1.1 with $M = 12$. (Modular arithmetic is taught in grade school as *clock* arithmetic!) It makes no intrinsic sense to say that one value is larger than another. But the order relation $<$ for the infinite set of integers must be imitated in modular arithmetic for each of the C++ types. In fact, given two instances j and k of an integer type, you determine whether $j < k$ by cutting the circle at the

natural place, flattening out the scale, then checking as usual to see whether $j < k$. The natural cut depends on whether the type is unsigned or signed, as in Figure 3.2.1.

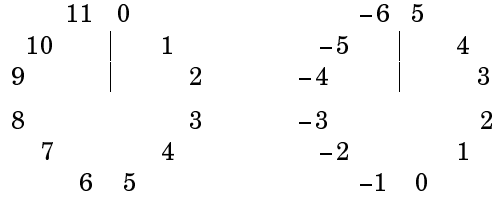

Figure 3.2.1 Circles for unsigned and signed modular arithmetic

This seems a straightforward way to interpret the order relation $j < k$, but you should realize that, because of the circular nature of modular arithmetic and the violence done by the cut, some of the familiar order properties fail to hold in integer arithmetic. For example, the statement *if* $i < j$ *and* $j < k$ *then* $i < k$ is still true, but the statement *if* $i < j$ *then* $2i < 2j$ is false when applied to type **int** arithmetic. (Try $i = 1$ and $j = 16384$: $2i = 2$ but $2j = -32768$. Following the analysis presented under the previous heading, the modulus is $M = 2^{16}$ and $j = \frac{1}{4}M$, so the intermediate product of 2 by j is $v = \frac{1}{2}M = M - \frac{1}{2}M$; the quotient and remainder are $q = 1$ and $-\frac{1}{2}M$. The latter is the final result $2j$ in type **int**.)

The absolute value function is involved with the order relation: $|j|$ is j or $-j$, whichever is ≥ 0. This operation is implemented by the MSP **abs** template in **General.H**, described in Section 3.1. It works with operands of any type for which operators **>** and **–** are defined. Thus it works with both **unsigned** and **signed** integer types. The Borland C++ Library routines **abs** and **labs**, on the other hand, are only defined for the signed types. If you misapplied one of *those* to a parameter

```
#include <IOStream.H>
#include   <StdLib.H>

void main() {
  unsigned j = 32769;
  cout << j << ' ' << abs(j); }
```

Output

```
32769 32767
```

Figure 3.2.2 Improper use of **abs**

of an unsigned type, you'd get an incorrect result. For example, consider the program in Figure 3.2.2: before executing **abs**, C++ automatically converts **j** to the **int** value –32767 with the same bit pattern in memory. *Beware!*

Integer division

When you divide an integer x by a nonzero divisor d, you get an integer quotient q and an integer remainder r such that

$$\frac{x}{d} = q + \frac{r}{d} \qquad x = qd + r$$

$$|r| < |d| \qquad \text{sign}(r) = \text{sign}(x) \ .$$

Since x and r have the same sign, q is also the result of truncating the exact (real) quotient x/d toward 0. The Intel 8086-family CPUs are designed to implement this familiar process, so Borland C++ integer division reflects these same features. For any integer type, you can obtain the quotient and remainder separately by executing

```
q = x/d;  r = x % d;
```

For the **int** type, you can get them both at once by executing the C++ Library function

```
div_t div(int x, int d)
```

The **div_t** type is defined with **div** in header file **StdLib.H** as

```
typedef struct {long quot; long rem; } div_t;
```

An analogous function **ldiv** and type **ldiv_t** are provided for dividing **long** integers.

Additional integer operations

As mentioned in Section 3.1, the MSP **General** module implements the function **Sign(x)** whose value is –1, 0, or 1 depending on whether **x** is negative, zero, or positive. Defined by a template, it works with any argument **x** for which operations **x**

`> 0` and `x == 0` are defined. Borland C++ Library header file `StdLib.H` includes similar templates for `max(x,y)` and `min(x,y).`

You can invoke the Borland C++ `Math.H` Library function

```
double pow(double b, double p)
```

with `int` arguments to compute an integer power b^p. It returns a `double` value; if that's in the `int` or `long` range, you can assign it immediately to a variable of that type, and it will be rounded correctly. More details of the function `pow` are discussed in Section 3.4. Two more functions commonly used for integer calculations are implemented in the MSP `ElemFunc` module:

```
long double Factorial(int    n);       // n!
double      Binomial (double x,        // Binomial coefficient
                      int    m);       // x  over  m .
```

The `Factorial` function returns a `long double` value because `n!` gets so large; it overflows when `n` > 1054. `ElemFunc` features are discussed in detail in Section 3.10.

Formatting output

Formatting numerical output is often unexpectedly difficult. C++ provides comprehensive facilities for this task—too many to consider here. For integer output the questions aren't really complicated. Even so, MSP needed a new function `Digits`, already discussed in Section 3.1, to compute the minimum length of a numeral that represents a given integer. This subject is discussed in great detail in Section 3.7 of the author's earlier book [47] on Borland C.

Integers of arbitrary length

Current developments are creating many applications for an integer data type that's not supported by Borland C++, nor included in the MSP software of this book: *arbitrarily long integers*. These are normally stored as linked lists of integers or linked

lists of digits. Devising efficient algorithms for their arithmetic is a major software engineering project. Some applications for this kind of programming are quite new, such as encryption methods for secure communication. Others originated decades or centuries ago—they stem from classical problems that require exact calculations with integer coefficients in problems involving polynomials and linear equations. Also, integers of arbitrary length form the numerators and denominators of rational numbers, a form of scalar that's coming into frequent use now to avoid round-off error. These kinds of applications are increasing in importance because the hardware necessary to perform such calculations in reasonable time is now becoming affordable. One of the first steps in developing MSP beyond the scope of this book should be to implement the arithmetic of integers of arbitrary length.