The Integer class.

The **Integer** class provides multiple precision integer arithmetic facilities. Some representation details are discussed in the Representation section.

Integers may be up to **b * ((1 << b) - 1)** bits long, where **b** is the number of bits per short (typically 1048560 bits when **b = 16**). The implementation assumes that a **long** is at least twice as long as a **short**. This assumption hides beneath almost all primitive operations, and would be very difficult to change. It also relies on correct behavior of **unsigned** arithmetic operations.

Some of the arithmetic algorithms are very loosely based on those provided in the MIT Scheme **bignum.c** release, which is Copyright © 1987 Massachusetts Institute of Technology. Their use here falls within the provisions described in the Scheme release.

Integers may be constructed in the following ways:

Integer x;	Declares an uninitialized Integer.
Integer x = 2; Integer y(2);	Set x and y to the Integer value 2;
Integer u(x); Integer v = x;	Set u and v to the same value as x.

Integers may be coerced back into longs via the **long** coercion operator. If the Integer cannot fit into a long, this returns MINLONG or MAXLONG (depending on the sign) where MINLONG is the most negative, and MAXLONG is the most positive representable long. The member function **fits_in_long()** may be used to test this. **Integers** may also be coerced into **doubles**, with potential loss of precision. **+/-HUGE** is returned if the Integer cannot fit into a double. **fits_in_double()** may be used to test this.

All of the usual arithmetic operators are provided (**+**, **-**, *****, **/**, **%**, **+=**, **++**, **-=**, **--**, ***=**, **/=**, **%=**, **==**, **!=**, **<**, **<=**, **>**, **>=**). All

operators support special versions for mixed arguments of Integers and regular C++ longs in order to avoid useless coercions, as well as to allow automatic promotion of shorts and ints to longs, so that they may be applied without additional Integer coercion operators. The only operators that behave differently than the corresponding int or long operators are **++** and **--**. Because C++ does not distinguish prefix from postfix application, these are declared as **void** operators, so that no confusion can result from applying them as postfix. Thus, for Integers x and y, **++x; y = x;** is correct, but **y = ++x;** and **y = x++;** are not.

Bitwise operators (**~**, **&**, **|**, **^**, **<<**, **>>**, **&=**, **|=**, **^=**, **<<=**, **>>=**) are also provided. However, these operate on sign-magnitude, rather than two's complement representations. The sign of the result is arbitrarily taken as the sign of the first argument. For example, **Integer(-3) & Integer(5)** returns **Integer(-1)**, not -3, as it would using two's complement. Also, **~**, the complement operator, complements only those bits needed for the representation. Bit operators are also provided in the BitSet and BitString classes. One of these classes should be used instead of Integers when the results of bit manipulations are not interpreted numerically.

The following utility functions are also provided. (All arguments are Integers unless otherwise noted).

void divide(x, y, q, r);	Sets q to the quotient and r to the remainder of x and y. (q and r are returned by reference).
Integer pow(Integer x, Integer p)	returns x raised to the power p.
Integer lpow(long x, long p)	returns x raised to the power p.
Integer gcd(x, y)	returns the greatest common divisor of x and y.
Integer lcm(x, y)	returns the least common multiple of x and y.
Integer abs(x);	returns the absolute value of x.
void x.negate();	negates x.

Integer sqr(x)	returns x * x;
Integer sqrt(x)	returns the floor of the square root of x.
long lg(x);	returns the floor of the base 2 logarithm of abs(x)
int sign(x)	returns -1 if x is negative, 0 if zero, else +1. Using **if (sign(x) == 0)** is a generally faster method of testing for zero than using relational operators.
int even(x)	returns true if x is an even number
int odd(x)	returns true if x is an odd number.
void setbit(Integer& x, long b)	sets the b'th bit (counting right-to-left from zero) of x to 1.
void clearbit(Integer& x, long b)	sets the b'th bit of x to 0.
int testbit(Integer x, long b)	returns true if the b'th bit of x is 1.
Integer atol(char* asciinumber, int base = 10);	converts the base base char* string into its Integer form.
void Integer::printon(ostream& s, int base = 10, int width = 0);	prints the ascii string value of **(*this)** as a base **base** number, in field width at least **width**.
ostream << x;	prints x in base ten format.
istream >> x;	reads x as a base ten number.

int compare(Integer x, Integer y)	returns a negative number if x<y, zero if x==y, or positive if x>y.
int ucompare(Integer x, Integer y)	like compare, but performs unsigned comparison.
add(x, y, z)	A faster way to say z = x + y.
sub(x, y, z)	A faster way to say z = x - y.
mul(x, y, z)	A faster way to say z = x * y.
div(x, y, z)	A faster way to say z = x / y.
mod(x, y, z)	A faster way to say z = x % y.
and(x, y, z)	A faster way to say z = x & y.
or(x, y, z)	A faster way to say z = x | y.
xor(x, y, z)	A faster way to say z = x ^ y.
lshift(x, y, z)	A faster way to say z = x << y.
rshift(x, y, z)	A faster way to say z = x >> y.
pow(x, y, z)	A faster way to say z = pow(x, y).
complement(x, z)	A faster way to say z = ~x.
negate(x, z)	A faster way to say z = -x.