Classes for Bit manipulation

libg++ provides several different classes supporting the use and manipulation of collections of bits in different ways.

- Class **Integer** provides ``integer'' semantics. It supports manipulation of bits in ways that are often useful when treating bit arrays as numerical (integer) quantities. This class is described elsewhere.

- Class **BitSet** provides ``set'' semantics. It supports operations useful when treating collections of bits as representing potentially infinite sets of integers.

- Class **BitSet32** supports fixed-length BitSets holding exactly 32 bits.

- Class **BitSet256** supports fixed-length BitSets holding exactly 256 bits.

- Class **BitString** provides ``string'' (or ``vector'') semantics. It supports operations useful when treating collections of bits as strings of zeros and ones.

These classes also differ in the following ways:

- BitSets are logically infinite. Their space is dynamically altered to adjust to the smallest number of consecutive bits actually required to represent the sets. Integers also have this property. BitStrings are logically finite, but their sizes are internally dynamically managed to maintain proper length. This means that, for example, BitStrings are concatenatable while BitSets and Integers are not.

- BitSet32 and BitSet256 have precisely the same properties as BitSets, except that they use constant fixed length bit vectors.

- While all classes support basic unary and binary operations **~**, **&**, **|**, **^**, **-**, the semantics differ. BitSets perform

bit operations that precisely mirror those for infinite sets. For example, complementing an empty BitSet returns one representing an infinite number of set bits. Operations on BitStrings and Integers operate only on those bits actually present in the representation. For BitStrings and Integers, the the **&** operation returns a BitString with a length equal to the minimum length of the operands, and **|**, **^** return one with length of the maximum.

- Only BitStrings support substring extraction and bit pattern matching.

BitSet

BitSets are objects that contain logically infinite sets of nonnegative integers. Representational details are discussed in the Representation chapter. Because they are logically infinite, all BitSets possess a trailing, infinitely replicated 0 or 1 bit, called the ``virtual bit'', and indicated via 0* or 1*.

BitSet32 and BitSet256 have they same properties, except they are of fixed length, and thus have no virtual bit.

BitSets may be constructed as follows:

BitSet a;	declares an empty BitSet.
BitSet a = atoBitSet("001000");	sets a to the BitSet 0010*, reading left-to-right. The ``0*'' indicates that the set ends with an infinite number of zero (clear) bits.
BitSet a = atoBitSet("00101*");	sets a to the BitSet 00101*, where ``1*'' means that the set ends with an infinite number of one (set) bits.
BitSet a = longtoBitSet((long)23);	sets a to the BitSet 111010*, the binary representation of decimal 23.

BitSet a = utoBitSet((unsigned)23); sets a to the BitSet 111010*, the binary representation of decimal 23.

The following functions and operators are provided (Assume the declaration of BitSets a = 0011010*, b = 101101*, throughout, as examples).

~a	returns the complement of a, or 1100101* in this case.
a.complement()	sets a to ~a.
a & b; a &= b;	returns a intersected with b, or 0011010*.
a | b; a |= b;	returns a unioned with b, or 1011111*.
a - b; a -= b;	returns the set difference of a and b, or 000010*.
a ^ b; a ^= b;	returns the symmetric difference of a and b, or 1000101*.
a.empty()	returns true if a is an empty set.
a == b;	returns true if a and b contain the same set.
a <= b;	returns true if a is a subset of b.
a < b;	returns true if a is a proper subset of b;
a != b; a >= b; a > b;	are the converses of the above.
a.set(7)	sets the 7th (counting from 0) bit of a, setting a to 001111010*

a.clear(2)	clears the 2nd bit bit of a, setting a to 00011110*
a.clear()	clears all bits of a;
a.set()	sets all bits of a;
a.invert(0)	complements the 0th bit of a, setting a to 10011110*
a.set(0,1)	sets the 0th through 1st bits of a, setting a to 110111110* The two-argument versions of clear and invert are similar.
a.test(3)	returns true if the 3rd bit of a is set.
a.test(3, 5)	returns true if any of bits 3 through 5 are set.
int i = a[3]; a[3] = 0;	The subscript operator allows bits to be inspected and changed via standard subscript semantics, using a friend class BitSetBit. The use of the subscript operator a[i] rather than a.test(i) requires somewhat greater overhead.
a.first(1) or **a.first()**	returns the index of the first set bit of a (2 in this case), or -1 if no bits are set.
a.first(0)	returns the index of the first clear bit of a (0 in this case), or -1 if no bits are clear.
a.next(2, 1) or a.next(2)	returns the index of the next bit after position 2 that is set (3 in this case) or -1. **first** and **next** may be used as iterators, as in **for (int i = a.first(); i >= 0; i = a.next(i))...**.

a.last(1)	returns the index of the rightmost set bit, or -1 if there or no set bits or all set bits.
a.previous(3, 0)	returns the index of the previous clear bit before position 3.
a.count(1)	returns the number of set bits in a, or -1 if there are an infinite number.
a.virtual_bit()	returns the trailing (infinitely replicated) bit of a.
a = atoBitSet("ababX", 'a', 'b', 'X');	converts the char* string into a bitset, with 'a' denoting false, 'b' denoting true, and 'X' denoting infinite replication.
a.printon(cout, '-', '.', 0)	prints **a** to **cout** represented with '-' for falses, '.' for trues, and no replication marker.
cout << a	prints **a** to **cout** (representing falses by 'f', trues by 't', and using '*' as the replication marker).
diff(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.
complement(x, z)	A faster way to say z = ~x.

BitString

BitStrings are objects that contain arbitrary-length strings of zeroes and ones. BitStrings possess some features that make them behave like sets, and others that behave as strings. They are useful in applications (such as signature-based algorithms) where both capabilities are needed. Representational details are discussed in the Representation chapter. Most capabilities are exact analogs of those supported in the BitSet and String classes. A BitSubString is used with substring operations along the same lines as the String SubString class. A BitPattern class is used for masked bit pattern searching.

Only a default constructor is supported. The declaration **BitString a;** initializes a to be an empty BitString. BitStrings may often be initialized via **atoBitString** and **longtoBitString**.

Set operations (**~**, **complement**, **&**, **&=**, **|**, **|=**, **-**, **^**, **^=**) behave just as the BitSet versions, except that there is no ``virtual bit'': complementing complements only those bits in the BitString, and all binary operations across unequal length BitStrings assume a virtual bit of zero. The **&** operation returns a BitString with a length equal to the minimum length of the operands, and **|**, **^** return one with length of the maximum.

Set-based relational operations (**==**, **!=**, **<=**, **<**, **>=**, **>**) follow the same rules. A string-like lexicographic comparison function, **lcompare**, tests the lexicographic relation between two BitStrings. For example, lcompare(1100, 0101) returns 1, since the first BitString starts with 1 and the second with 0.

Individual bit setting, testing, and iterator operations (**set**, **clear**, **invert**, **test**, **first**, **next**, **last**, **previous**) are also like those for BitSets. BitStrings are automatically expanded when setting bits at positions greater than their current length.

The string-based capabilities are just as those for class String. BitStrings may be concatenated (**+**, **+=**), searched (**index**, **contains**, **matches**), and extracted into BitSubStrings (**before**, **at**, **after**) which may be assigned and otherwise manipulated. Other string-based utility functions (**reverse**, **common_prefix**, **common_suffix**) are also provided. These have the same capabilities and descriptions as those for Strings.

String-oriented operations can also be performed with a mask via class BitPattern. BitPatterns consist of two BitStrings, a pattern and a mask. On searching and matching, bits in the pattern that correspond to 0 bits in the mask are ignored. (The mask may be shorter than the pattern, in which case trailing mask bits are assumed to be 0). The pattern and mask are both public variables, and may be individually subjected to other bit operations.

Converting to char* and printing (**atoBitString**, **atoBitPattern**, **printon**, **ostream <<**) are also as in BitSets, except that no virtual bit is used, and an 'X' in a BitPattern means that the pattern bit is masked out.

The following features are unique to BitStrings.

Assume declarations of BitString a = atoBitString("01010110") and b = atoBitSTring("1101").

a = b + c;	Sets a to the concatenation of b and c;
a = b + 0; a = b + 1;	sets a to b, appended with a zero (one).
a += b;	appends b to a;
a += 0; a += 1;	appends a zero (one) to a.
a << 2; a <<= 2	return a with 2 zeros prepended, setting a to 0001010110. (Note the necessary confusion of << and >> operators. For consistency with the integer versions, << shifts low bits to high, even though they are printed low bits first.)
a >> 3; a >>= 3	return a with the first 3 bits deleted, setting a to 10110.
a.left_trim(0)	deletes all 0 bits on the left of a, setting a to 1010110.

a.right_trim(0)	deletes all trailing 0 bits of a, setting a to 0101011.
cat(x, y, z)	A faster way to say z = x + y.
diff(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.
complement(x, z)	A faster way to say z = ~x.