Doc. No. X3J16/92-0051
ISO/WG21/N0128
Date: May 26, 1992
Project: Programming Language C++
Reply to: Chuck Allison
allison@decus.org
(801) 240-4510

Bitsets for C++

Many vendors currently supply bit-handling classes to compensate for the deficiencies of the bitwise operators of the C programming language. This paper proposes two such classes for the C++ standard library.

Class `bits` represents a fixed-length collection of bits. It extends C bitwise semantics by allowing easy access to bits, by allowing an arbitrary number of bits in a bitset, and by adding new functionality. It is defined as a template class, with the number of bits in the collection as the template parameter. It is highly suitable for interface with the host operating system, and is designed for efficiency (it can be stack-based).

Class `bit_string` can be used when the number of bits is not known at compile time, or if a dynamic-length bitset is desired.

Neither of these classes attempts to provide complete semantics for mathematical sets, but member functions exist that provide the equivalent of union, intersection, and symmetric difference as well as insert, remove and test for membership.

Class `bits`

```
typedef unsigned long _Block;   // any unsigned integral type

#include <xmsg>
class bits_msg : public xmsg {};

class string;

template<int nbits>
class bits
{
public:
    // Exception class
    class failure : public bits_msg {};

    // Constructors
    bits();
    bits(_Block);
    bits(const bits&);
    bits(const string&) throw failure;

    // Conversions
    _Block block(unsigned = 0) const;
```

```
    operator string();

    // Assignment
    bits& operator=(const bits&);

    // Equality
    int operator==(const bits&) const;
    int operator!=(const bits&) const;

    // Basic operations
    bits& set(unsigned) throw failure;
    bits& set();
    bits& reset(unsigned) throw failure;
    bits& reset();
    bits& toggle(unsigned) throw failure;
    bits& toggle();
    bits operator~() const;
    int test(unsigned) const throw failure;
    int any() const;
    int none() const;
    unsigned count() const;
    unsigned size() const;

    // Bitwise operators
    bits& operator<<=(unsigned);
    bits operator<<(unsigned) const;
    bits& operator>>=(unsigned);
    bits operator>>(unsigned) const;

    bits operator&(const bits&) const;
    bits operator|(const bits&) const;
    bits operator^(const bits&) const;

    bits& operator&=(const bits&);
    bits& operator|=(const bits&);
    bits& operator^=(const bits&);
};
```

Member Function Descriptions

`bits()`
: initializes all bits to zero.

`bits(_Block n)`
: initializes the object with the bits of `n`. If `nbits > sizeof(_Block)`, the extra bits are set to zero.

`bits(const bits&)`
: standard copy constructor.

`bits(const string& s) throw failure`
: each character of `s` is interpreted as a bit (a string of 1's and 0's is expected). In typical fashion, the last character of `s` is considered to be bit-0. Throws `bits::failure` if a character other than '1' or '0' is encountered.

`_Block block(unsigned n = 0) const`
: the `n`th block of `*this` is returned (defaults to the block that includes bit-0). This is useful when the bits represent flags in a word passed to the operating system.

`operator string()`
returns a temporary string of 1's and 0's representing the contents of `*this`. As with C unsigned's, the last character is bit-0.

`bits& operator=(const bits&)`
standard assignment operator.

`int operator==(const bits&) const`
`int operator!=(const bits&) const`
obvious.

`bits& set(unsigned n) throw failure`
sets the `n`th bit. Throws `bits::failure` if `n` is not in `[0,nbits-1]`.

`bits& set()`
sets all bits.

`bits& reset(unsigned) throw failure`
resets the `n`th bit. Throws `bits::failure` if `n` is not in `[0,nbits-1]`.

`bits& reset()`
resets all bits.

`bits& toggle(unsigned) throw failure`
toggles the `n`th bit. Throws `bits::failure` if `n` is not in `[0,nbits-1]`.

`bits& toggle()`
toggles all bits.

`bits operator~() const`
returns the results as if all the bits of `*this` were toggled.

`int test(unsigned n) const throw failure`
returns 1 if bit-`n` is set, 0 otherwise. Throws `bits::failure` if `n` is not in `[0,nbits-1]`.

`int any() const`
returns 0 if all bits are 0; 1 otherwise.

`int none() const`
returns 1 if all bits are 0; 0 otherwise.

`bits operator<<(unsigned n) const`
returns the results as if `*this` were shifted left `n` bit positions. If `n > nbits`, then all bits are reset. Shifting "left" by `n` means that bit-`n` receives the value of bit-0, bit-(`n`+1) receives bit-1, etc.

`bits operator>>(unsigned n) const`
returns the results as if `*this` were shifted right `n` bit positions. If `n > nbits`, then all bits are reset. Shifting "right" by `n` means that bit-0 receives the value of bit-`n`, bit-1 receives bit-(`n`+1), etc.

`bits& operator<<=(unsigned)`
`bits& operator>>=(unsigned)`
assigment versions of the above.

`bits operator&(const bits& b) const`
returns the result of the bitwise-and of `*this` and `b`.

`bits operator|(const bits& b) const`
returns the result of the bitwise-or of `*this` and `b`.

`bits operator^(const bits& b) const`
returns the result of the bitwise-xor of `*this` and `b`.

```
bits& operator&=(const bits&)
bits& operator|=(const bits&)
bits& operator^=(const bits&)
```
assignment versions of the above.

`int count() const`
returns the number of bits that are set.

`int size() const`
returns "`nbits`".

Notes for Class `bits`

1. Having an expression as the template parameter has the following effects:

   - a `bits`-object can be stack-based
   - objects of different sizes (i.e., with a different number of bits) are different types (so such objects cannot be mixed in a common expression)
   - no **friend** functions are allowed (because they become template functions, which are not allowed to have expression parameters)

2. Since objects of different sizes are different types, the class `bits_msg` is necessary to allow a single **catch** clause to handle an exception thrown by any `bits` object.

3. The forced "friendlessness" of this class has the following effects:

   - There can be no binary stream operators. To accommodate I/O, conversions to and from class `string` are included. This allows statements such as the following:

     ```
     string s("101110011");
     bits<24> b(s);
     cout << s << endl;
     ```

     Since this class is an "extension" of unsigned integer as far as bitwise operations are concerned, a string of bits represents a number. Following standard numerical convention, bit-0 is considered the rightmost bit, so the output of the above statements would be:

     ```
     000000000000000101110011
     ```

   - We can evaluate the expression `b | 5` but not `5 | b`.

Because of this lack of symmetry, there is some disagreement on whether we should include non-assignment versions of the "and", "or", or "exclusive-or" bitwise operators. This author thinks that they would be heavily used and appreciated in practice, asymmetry notwithstanding.

4. To be consistent with current C bitwise operations, the statements:

```
bits<8> b;
b |= 5;
cout << b << endl;
```

result in

```
00000101
```

that is, the bits of 5 are or'ed (instead of bit-5 being turned on).

Class `bit_string`

This abstraction represents a *string* of bits. It seems reasonable, therefore, for it to behave like a string, i.e., bit-0 is the leftmost, just like character-0 is the leftmost in character strings. I have proceeded under this assumption. This makes converting to and from unsigneds a little counter-intuitive, but the **string**-ness (or "**array**-ness") is the foundation of this abstraction. If you want an abstraction compatible with numeric conventions (i.e., where bit-0 is rightmost), use the class `bits` instead.

Most of the functionality is identical to class `bits`. The main differences are in the constructors, and the additional functions `append()`, `concat()`, `size(unsigned)`, and `trim()`. For binary operations between two bit strings, the shorter is considered to be zero-padded *on the right* so as to have the same length as the longer for the operation.

```
typedef unsigned long _Block;    // any unsigned integral type

#include <xmsg>

class string;

class bit_string
{
public:
    // Exception class
    class failure : public xmsg {};

    // Constructors
    bit_string();
    bit_string(_Block, unsigned = 1);
    bit_string(const string&) throw failure;
    ~bit_string();

    // Copy/Assign
    bit_string(const bit_string&);
    bit_string& operator=(const bit_string&);

    // Conversions
    _Block block(unsigned n = 0) const;
    operator string() const;
```

```
    // Basic operations
    bit_string& set(unsigned) throw failure;
    bit_string& set();
    bit_string& reset(unsigned) throw failure;
    bit_string& reset();
    bit_string& toggle(unsigned) throw failure;
    bit_string& toggle();
    int test(unsigned) const throw failure;
    int any() const;
    int none() const;
    bit_string operator~() const;
    unsigned count() const;
    unsigned size() const;
    unsigned size(unsigned);
    unsigned trim();
    bit_string& append(const bit_string&);
    friend bit_string concat(const bit_string&,const bit_string&);

    // Bitwise operations
    bit_string& operator<<=(unsigned);
    friend bit_string operator<<(const bit_string&, unsigned);
    bit_string& operator>>=(unsigned);
    friend bit_string operator>>(const bit_string&, unsigned);

    friend bit_string operator&(const bit_string&, const bit_string&);
    friend bit_string operator|(const bit_string&, const bit_string&);
    friend bit_string operator^(const bit_string&, const bit_string&);
    bit_string& operator&=(const bit_string&);
    bit_string& operator|=(const bit_string&);
    bit_string& operator^=(const bit_string&);

    // Equality
    int operator==(const bit_string& b) const;
    int operator!=(const bit_string& b) const;
};
```

Member Function Descriptions

`bit_string()`

creates an empty bit string.

`bit_string(_Block n, unsigned nbits = 0)`

creates a bit string of `length() >= nbits`, initialized with the bits of `n`. No significant bits are lost, i.e., `this->length() == max(nbits, N+1)`, where `N` is the bit position of the highest-order 1-bit. If `n == 0`, then it creates the bit string "0". Pads with zeroes in the higher order bits if necessary to fill out `nbits` bits. The common usage of this constructor would be to initialize a `bit_string` of a certain length with zeroes, e.g.,

```
bit_string x(0,16);
```

Care should be taken when using a non-zero value for initialization, since the bits will be "reversed". For example, the numeric bit pattern for the number 21537 is 101001000100001, but the output from the following code

```
bit_string x(21537);
cout << x << endl;
```

is 100001000100101.

`bit_string(const string& s) throw failure`
: each character of `s` is interpreted as a bit (a string of 1's and 0's is expected). The first (left-most) character of `s` is considered to be bit-0. `this->length() == strlen(s)`. Throws `bit_string::failure` if any character in the string is not a '1' or '0'.

`_Block block(unsigned n = 0) const`
: the `n`th block of `*this` is returned (defaults to the one that includes bit-0). The bits are reversed so that bit-0 from `*this` (the "left-most") is the same as bit-0 in the returned unsigned integer (the "rightmost"). Inclusion of this function in the class is questionable. Applications requiring integer conversion should probably be using class `bits`.

`operator string()`
: returns a temporary string of 1's and 0's representing the contents of `*this`. The first (left-most) character is considered bit-0.

`int operator==(const bit_string&) const`
`int operator!=(const bit_string&) const`
: bit strings are equal iff they have the same `size()` and bit pattern.

`bit_string& set(unsigned n)`
: sets the `n`th bit.

`bit_string& set()`
: sets all bits.

`bit_string& reset(unsigned)`
: resets the `n`th bit.

`bit_string& reset()`
: resets all bits.

`bit_string& toggle(unsigned)`
: toggles the `n`th bit.

`bit_string& toggle()`
: toggles all bits.

`bit_string operator~() const`
: returns the result as if all the bits of `*this` were toggled.

`int test(unsigned n) const`
: returns 1 if bit-`n` is set, 0 otherwise.

`int any() const`
: returns 0 if all bits are 0; 1 otherwise.

`int none() const`
: returns 1 if all bits are 0; 0 otherwise.

`bit_string operator<<(unsigned n) const`
: returns the result as if `*this` were shifted left `n` bit positions. If `n > nbits`, then all bits are reset. Shifting "left" by `n` means that bit-0 receives the value of bit-`n`, bit-1 receives bit-(`n`+1), etc (NOTE: because of the left-to-right ordering of the bits in a `bit_string`, this is "backwards" from the bitwise semantics of unsigneds, but is visually correct).

`bit_string operator>>(unsigned n) const`
returns the result as if `*this` were shifted right left `n` bit positions. If `n > nbits`, then all bits are reset. Shifting "right" by `n` means that bit-`n` receives the value of bit-0, bit-`(n+1)` receives bit-1, etc (NOTE: because of the left-to-right ordering of the bits in a `bit_string`, this is "backwards" from the bitwise semantics of unsigneds, but is visually correct).

`bit_string& operator<<=(unsigned)`
`bit_string& operator>>=(unsigned)`
assigment versions of the above.

`bit_string operator&(const bit_string& b) const`
returns the result of the bitwise-and of `*this` and `b`. The length of the result is the length of the longer of the two operands. Behaves as if, prior to the operation, the shorter operand is extended with high-order zero bits until its length equals that of the longer operand.

`bit_string operator|(const bit_string& b) const`
returns the result of the bitwise-or of `*this` and `b`. The length of the result is the length of the longer of the two operands. Behaves as if, prior to the operation, the shorter operand is extended with high-order zero bits until its length equals that of the longer operand.

`bit_string operator^(const bit_string& b) const`
returns the result of the bitwise-xor of `*this` and `b`. The length of the result is the length of the longer of the two operands. Behaves as if, prior to the operation, the shorter operand is extended with high-order zero bits until its length equals that of the longer operand.

`bit_string& operator&=(const bit_string&)`
`bit_string& operator|=(const bit_string&)`
`bit_string& operator^=(const bit_string&)`
assigment versions of the above. Semantics are such that `x &= y` is equivalent to `x = x & y`, etc.

`unsigned count() const`
returns the number of bits that are set.

`unsigned size() const`
returns the number of bits in the bit string.

`unsigned size(unsigned n)`
truncates or zero-pads as needed so that `this->length == n`. Returns `n`.

`unsigned trim()`
Truncates high-order zero bits. If `N` is the highest-numbered 1-bit, then `this->length()` becomes `N+1`. Returns the new length `(N+1)`.

`bit_string& append(const bit_string& b)`
extends `*this` with the bits of `b`. `this->length() == oldthis->length() + b.length()`.

`bit_string concat(const bit_string& b1, const bit_string& b2)`
returns the result as if `b2` were appended to `b1`.

Notes for Class `bit_string`

1) To accommodate I/O, conversions to and from class `string` are included. This allows statements such as the following:

```
string s("101110011");
bit_string b(s);
cout << s << endl;
```

The output of the above statements would be:

```
101110011
```

2) The statements:

```
bit_string(0,8) b;
b |= 5;
cout << b << endl;
```

result in

```
10100000
```

that is, the bits of 5 are converted to the bit_string "101", which is then extended to "10100000", and in turn and-ed with 0. Again, the bits are reversed because bit-0 is considered left-most in a `bit_string`. Using a non-zero value for initialization is of little value.

3) I really don't care about the names `append()` vs. `concat()`. I've chosen to make them different, but will agree if others want the names the same. However, it seems like stretching things a bit to overload `operator+()` here (this is reasonable for character strings, but might be confusing since we're mixing features of strings with binary numbers here).

4) As in class `bits`, note the absence of `operator[]()`.

5) I could have added `operator<<(ostream&, const bit_string&)` and `operator>>(istream&, bit_string&)` for I/O, but we already have string conversion, so I decided against adding yet one more feature missing in class `bits`.