Doc. No. X3J16/91-0142
ISO/WG21/N0075
Date: November 25, 1991
Project: Programming Language C++
Reply to: Chuck Allison
allison@decus.org
(801) 240-4510

Proposal for a Bitset Class

Research by the Library Working Group has revealed that a class supporting sets of integral types is found in many of the popular third-party C++ class libraries available today. This paper proposes such a class for inclusion into the standard.

A bitset can be seen as either an array of bits or a set of integers. As a bit array, it allows the typical operations of setting, resetting, testing and toggling a bit. The first three operations are equivalent to adding, removing and testing for membership, respectively, an integer with respect to a set. Other basic set operations include union, intersection, difference, complement and symmetric difference (exclusive-or). Most of these features are available in languages such as Pascal and Modula-2. This proposal formulates them in a way fitting the philosophy of C++ (e.g., bit-wise operators are overloaded when it makes sense to do so).

Common usage suggests that dynamic-length bitsets are seldom needed. This proposal calls for the number of bits to be supported by an object to be supplied to the constructor. Once fixed, the capacity of a bitset does not change (so assignment may possibly truncate). Bitsets of different sizes can appear in the same operation. Whether truncation or zero-padding is used depends upon the operation:

```
size(x & y) == min(size(x), size(y))      // Truncates
size(x | y) == max(size(x), size(y))      // Pads
size(x ^ y) == max(size(x), size(y))      // Pads
size(x - y) == size(x)                    // May truncate
```

where "size" means the capacity (i.e., number of bits supported). This of course only applies to these non-destructive operators.

Exceptions have not yet been considered.

Constructors / Destructors

```
Bitset(unsigned);                 // Number of bits / size of set
Bitset(const Bitset&);
~Bitset();
```

Assignment

```
Bitset& operator=(const Bitset&);      // Truncates long right operands
```

Bit Operations

```
Bitset& set(unsigned);                 // Set a bit
Bitset& set(unsigned, unsigned);       // Set a range of bits
Bitset& set();                         // Set all bits

Bitset& reset(unsigned);               // Reset a bit
Bitset& reset(unsigned, unsigned);     // Reset a range of bits
Bitset& reset();                       // Reset all bits

Bitset& toggle(unsigned);              // Toggle a bit
Bitset& toggle(unsigned, unsigned);    // Toggle a range
Bitset& toggle();                      // Toggle all
    (NOTE: "compl" would be nice here, but is now a keyword)

int test(unsigned b) const;            // Test if a bit is set
```

Set Operations

```
// Intersection
Bitset& operator&=(const Bitset&);
Bitset& set_intersect(const Bitset&);  // Same as &=
friend Bitset operator&(const Bitset&, const Bitset&);

// Union
Bitset& operator|=(const Bitset&);
Bitset& set_union(const Bitset&);      // Same as |=
friend Bitset operator|(const Bitset&, const Bitset&);

// Difference
Bitset& operator-=(const Bitset&);
Bitset& set_diff(const Bitset&);       // Same as -=
friend Bitset operator-(const Bitset&, const Bitset&);

// Symmetric Difference (XOR)
Bitset& operator^=(const Bitset&);
Bitset& set_sym_diff(const Bitset&);   // Same as ^=
friend Bitset operator^(const Bitset&, const Bitset&);

// Complement (non-destructive; toggle() is destructive)
Bitset operator~() const;

// Test for empty set
int isempty() const;
operator int() const;                  // Inverse of isempty()
```

```
// Count (number of members in set)
int count() const;

// Size of universe (number of bits supplied to constructor)
int size();

// Subset operations
int subsetof(const Bitset&) const;      // Same as <=
friend int operator<=(const Bitset&, const Bitset&);
friend int operator>=(const Bitset&, const Bitset&);
friend int operator<(const Bitset&, const Bitset&);
friend int operator>(const Bitset&, const Bitset&);

// Equality
int equal(const Bitset&) const;
int operator==(const Bitset&) const;
int not_equal(const Bitset& b) const;
int operator!=(const Bitset& b) const;

// Inserter
friend ostream& operator<<(ostream&, const Bitset&);
```

It appears unreasonable to impose operator[]() for subscripting (how does one return a reference to an arbitrary bit position?). It could be used as an rvalue (1 if bit was set, 0 if not), but why have an asymmetrical subscript operator? Use set() or test() instead. The member functions operator<() and operator>() are convenient, but not necessary. The range versions of set(), reset() and toggle can also be omitted, if you think this class is too large.

A sample implementation follows.

```
// bitset.h:    Bit arrays / sets

class ostream;

class Bitset
{
public:
    Bitset(unsigned nb);
    Bitset(const Bitset&);
    ~Bitset() {delete [numwords(nbits)] bits;}

    // Bit operations
    Bitset& set(unsigned);
    Bitset& set(unsigned, unsigned);
    Bitset& set();
    Bitset& reset(unsigned);
    Bitset& reset(unsigned, unsigned);
    Bitset& reset();
    Bitset& toggle(unsigned);
    Bitset& toggle(unsigned, unsigned);
    Bitset& toggle();
    int test(unsigned b) const;

    // Set operations
    Bitset& operator&=(const Bitset& b) {return set_intersect(b);}
    Bitset& set_intersect(const Bitset&);
    Bitset& operator|=(const Bitset& b) {return set_union(b);}
    Bitset& set_union(const Bitset&);
    Bitset& operator^=(const Bitset& b) {return set_sym_diff(b);}
    Bitset& set_sym_diff(const Bitset&);
    Bitset& operator-=(const Bitset& b) {return set_diff(b);}
    Bitset& set_diff(const Bitset&);
    Bitset operator~() const;
    operator int() const {return !isempty();}
    int isempty() const;
    int count() const;
    int size() const {return nbits;}
    int subsetof(const Bitset&) const;

    // Equality
    int operator==(const Bitset& b) const {return equal(b);}
    int equal(const Bitset&) const;
    int operator!=(const Bitset& b) const {return !equal(b);};
    int not_equal(const Bitset& b) const {return !equal(b);};

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

    // Friends
    friend ostream& operator<<(ostream&, const Bitset&);
    friend Bitset operator&(const Bitset&, const Bitset&);
    friend Bitset operator|(const Bitset&, const Bitset&);
    friend Bitset operator^(const Bitset&, const Bitset&);
    friend Bitset operator-(const Bitset&, const Bitset&);
    friend int operator<=(const Bitset& x, const Bitset& y)
      {return x.subsetof(y);}
    friend int operator>=(const Bitset& x, const Bitset& y)
      {return y.subsetof(x);}
    friend int operator<(const Bitset& x, const Bitset& y)
      {return x != y && x.subsetof(y);}
    friend int operator>(const Bitset& x, const Bitset& y)
      {return x != y && y.subsetof(x);}

private:
    unsigned *bits;
    unsigned nbits;

    enum {WRDSIZ = 16};
    static unsigned word(unsigned b) {return b / WRDSIZ;}
    static unsigned offset(unsigned b) {return WRDSIZ - b%WRDSIZ - 1;}
    static unsigned mask(unsigned b) {return 1u << offset(b);}
    static unsigned numwords(unsigned nb) {return (nb+WRDSIZ-1) / WRDSIZ;}
```

```
    void cleanup();
    void _reset(unsigned b) {bits[word(b)] &= ~mask(b);}  // Caveat usor
};
```

```
// bitset.cpp

#include <iostream.h>
#include "bitset.h"

Bitset::Bitset(unsigned nb)
{
    nbits = nb;
    bits = new unsigned[numwords(nb)];
    reset();
}

ostream& operator<<(ostream& os, const Bitset& bs)
{
    for (int i = 0; i < bs.nbits; ++i)
        os << bs.test(i);
    return os;
}

Bitset& Bitset::set()
{
    unsigned nw = numwords(nbits);
    while (nw--)
        bits[nw] = ~0u;
    cleanup();
    return *this;
}

Bitset& Bitset::set(unsigned b)
{
    if (b < nbits)
        bits[word(b)] |= mask(b);
    return *this;
}

Bitset& Bitset::set(unsigned start, unsigned stop)
{
    for (int i = start; i < nbits && i <= stop; ++i)
        set(i);
    return *this;
}

Bitset& Bitset::reset()
{
    unsigned nw = numwords(nbits);
    while (nw--)
        bits[nw] = 0u;
    return *this;
}

Bitset& Bitset::reset(unsigned b)
{
    if (b < nbits)
        _reset(b);
    return *this;
}

Bitset& Bitset::reset(unsigned start, unsigned stop)
{
    for (int i = start; i < nbits && i <= stop; ++i)
        reset(i);
    return *this;
}

Bitset& Bitset::toggle()
{
    unsigned nw = numwords(nbits);
    while (nw--)
        bits[nw] = ~bits[nw];
    cleanup();
    return *this;
}
```

```
Bitset& Bitset::toggle(unsigned b)
{
    if (b < nbits)
        bits[word(b)] ^= mask(b);
    return *this;
}

Bitset& Bitset::toggle(unsigned start, unsigned stop)
{
    for (int i = start; i < nbits && i <= stop; ++i)
        toggle(i);
    return *this;
}

int Bitset::test(unsigned b) const
{
    if (b < nbits)
        return (bits[word(b)] & mask(b)) != 0;
    else
        return 0;
}

Bitset& Bitset::set_intersect(const Bitset& b)
{
    unsigned nw = numwords(nbits);
    unsigned nw2 = numwords(b.nbits);
    for (int i = 0; i < nw && i < nw2; ++i)
        bits[i] &= b.bits[i];
    return *this;
}

Bitset& Bitset::set_union(const Bitset& b)
{
    unsigned nw = numwords(nbits);
    unsigned nw2 = numwords(b.nbits);
    for (int i = 0; i < nw && i < nw2; ++i)
        bits[i] |= b.bits[i];
    cleanup();
    return *this;
}

Bitset& Bitset::set_sym_diff(const Bitset& b)
{
    unsigned nw = numwords(nbits);
    unsigned nw2 = numwords(b.nbits);
    for (int i = 0; i < nw && i < nw2; ++i)
        bits[i] ^= b.bits[i];
    cleanup();
    return *this;
}

Bitset& Bitset::set_diff(const Bitset& b)
{
    unsigned nw = numwords(nbits);
    unsigned nw2 = numwords(b.nbits);
    for (int i = 0; i < nw && i < nw2; ++i)
        bits[i] &= ~b.bits[i];
    return *this;
}

Bitset& Bitset::operator=(const Bitset& b)
{
    unsigned nw = numwords(nbits);
    unsigned nw2 = numwords(b.nbits);
    for (int i = 0; i < nw && i < nw2; ++i)
        bits[i] = b.bits[i];
    cleanup();
    return *this;
}
```

```
Bitset::Bitset(const Bitset& b)
{
    nbits = b.nbits;
    bits = new unsigned[numwords(nbits)];
    for (int i = 0; i < numwords(nbits); ++i)
          bits[i] = b.bits[i];
}

int Bitset::equal(const Bitset& b) const
{
    // Must be the same size to be equal
    if (nbits == b.nbits)
    {
        for (int i = 0; i < numwords(nbits); ++i)
            if (bits[i] != b.bits[i])
                return 0;
        return 1;
    }
    return 0;
}

Bitset operator&(const Bitset& x, const Bitset& y)
{
    if (x.nbits < y.nbits)
    {
        Bitset b(x);
        return b &= y;
    }
    else
    {
        Bitset b(y);
        return b &= x;
    }
}

Bitset operator|(const Bitset& x, const Bitset& y)
{
    if (x.nbits > y.nbits)
    {
        Bitset b(x);
        return b |= y;
    }
    else
    {
        Bitset b(y);
        return b |= x;
    }
}

Bitset operator^(const Bitset& x, const Bitset& y)
{
    if (x.nbits > y.nbits)
    {
        Bitset b(x);
        return b ^= y;
    }
    else
    {
        Bitset b(y);
        return b ^= x;
    }
}

Bitset operator-(const Bitset& x, const Bitset& y)
{
    Bitset b(x);
    return b -= y;
}
```

```
Bitset Bitset::operator~() const
{
    Bitset b(*this);
    b.toggle();
    b.cleanup();
    return b;
}

int Bitset::isempty() const
{
    for (int i = 0; i < numwords(nbits); ++i)
        if (bits[i])
            return 0;
    return 1;
}

int Bitset::subsetof(const Bitset& b) const
{
    Bitset r(*this);
    r &= b;
    return equal(r);
}

int Bitset::count() const
{
    int sum = 0;
    for (int i = 0; i < nbits; ++i)
        if (test(i))
            ++sum;
    return sum;
}

void Bitset::cleanup()
{
    // Make sure unused bits don't get set
    for (int i = nbits; i < numwords(nbits)*WRDSIZ; ++i)
        _reset(i);
}
```

```
          .pp:            Test the Bitset class

        <iostream.h>
      .e "bitset.h"

   .n()

    Bitset x(12), y(18);

    x.set(0,5);
    cout << x << endl;
    cout << "count: " << x.count() << ", size: " << x.size() << endl;
    y.set(0,8);
    cout << y << endl;
    cout << (x <= y) << endl;
    x.toggle();
    cout << ~x << endl;
    cout << x << endl;
    cout << (x <= y) << endl;
    cout << (x == y) << endl;
    cout << (x != y) << endl;

    cout << (x & y) << endl;
    cout << (x | y) << endl;
    cout << (x ^ y) << endl;
    cout << (x - y) << endl;
    cout << (y - x) << endl;

    if (x)
        cout << x << endl;
    else
        cout << "didn't print" << endl;
    x.reset();
    if (x)
        cout << x << endl;
    else
        cout << "didn't print" << endl;
}
```