

Running Circles Round You, Logically

by Stephen C. Dewhurst

It's hard times in small-town New England. Town meeting's coming up, state and local tax revenues are way down, and things don't look good for the school budget. High school programming courses are being cut. What's a concerned C++ hacker to do? Pitch in and teach, of course! Things have changed a lot since my day, and the College Board now offers an advanced placement examination (actually two) for computer science. Amazing.

However, amazement gives way to, well, amazement on close reading of the description of the subset of C++ used in the examination (which is, perforce, the subset that will be taught). In the section that describes that portion of C++ not included in the subset, we learn that the “operators: comma, `?:`, bitwise, `sizeof`” are “not essential.”¹ Not essential! Just to give you an idea how the youth of America is being corrupted, inheritance is not discussed either. Or reference variables. Or `typedef`. Or implicit conversions. But the comment about bitwise operators stings.

In this installment of Common Knowledge, I hope to strike blow for bitwise operators. I intend to show that, far from being outmoded antiquities, they can be alloyed with the most recent enhancements of C++ to the benefit of both. We're going to meld the standard template library with the bitwise exclusive-or operator. And like it.

Illogical Irritants

Isn't it irritating to see code like this?

```
bool r;  
if( a < b )  
    r = true;  
else  
    r = false;
```

Instead of this?

```
bool r = a<b;
```

Do you have to count to eight when presented with the following?

```
int ctr = 0;  
for( int i = 0; i < 8; ++i )  
    if( bits & 1<<(8+i) )  
        if( ctr++ ) {
```

```
        cerr << "Too many options selected";
        break;
    }
```

Instead of this?

```
    if( bits&0xFF00 & (bits&0xFF00)-1 )
        cerr << "Too many options selected";
```

What ever happened to Boolean logic? Why, when I was a boy, machines had only two registers, and believe-you-me we were happy to have them! And memory was scarce, but we got by because we knew the virtue of thrift, and had low expectations. And not only that, we had to walk miles to the computer center through snow that was...oh, excuse me.

In any case, we employed a number of simple coding tricks that could be used profitably in these constrained circumstances. One of my favorites is the “exclusive-or trick” for swapping values without a temporary.

```
void swap( int &a, int &b ) {
    a ^= b;
    b ^= a;
    a ^= b;
}
```

A moment’s reflection shows that the values $a \oplus b$ and b can be used to derive a , and that $a \oplus b$ and a can be used to derive b . And of course, inasmuch as pointers to data are really just integers with an attitude, we have the following:

```
template <typename T>
void swap( T *&a, T *&b ) {
    typedef ptrdiff_t I;
    (I &)a ^= (I &)b;
    (I &)b ^= (I &)a;
    (I &)a ^= (I &)b;
}
```

At an intermediate stage in the swap, we have a value that is essentially a two-way pointer that is just waiting to be decoded.² Let’s use this observation in the design of an STL-like container with a charmingly ambivalent attitude toward order and direction.

A List, Naturally

Our list container will be composed of the nodes that form the list, a list handle, and iterators over the list.

```
template <typename T> class ListEl;
template <typename T> class List;
template <typename T> class Iter;
```

The list nodes contain a single pointer that is the encoding of both the previous and next elements in the sequence of nodes.

```
struct ListElBase {
    ListElBase () : ptr(0) {}
    ListElBase *ptr;
    typedef std::ptrdiff_t Bits;
};

template <typename T>
class ListEl : public ListElBase {
    ListEl( const T &v ) : el( v ) {}
    T el;
    friend class List<T>;
    friend class Iter<T>;
};
```

The two-way pointer can be decoded with the address of the previous node to produce the address of the following node, or with the address of the following node to produce the previous node. That is, since the value of the node's pointer is $\text{previous}^{\wedge}\text{next}$, $\text{previous}^{\wedge}\text{next}^{\wedge}\text{previous} == \text{next}$, and $\text{previous}^{\wedge}\text{next}^{\wedge}\text{next} == \text{previous}$.

The list container itself is not very impressive, and we'll keep things to a manageable minimum here for reasons of space.³

```
template <typename T>
class List {
public:
    typedef Iter<T> iterator;
    typedef T value_type;
    typedef std::ptrdiff_t difference_type;
    typedef std::size_t size_type;
    //...
    List();
    List( const List & );
    List &operator =( const List & );
    bool empty() const;
    size_type size() const;
    size_type max_size() const;
    void swap( List & );
    void push_back( const T &v );
```

```
void push_front( const T &v );
void pop_front();
void pop_back();
iterator begin();
iterator end();
void reverse();
//...
};
```

Most list implementations employ either a dummy node to indicate the start and end of the list, or a pair of pointers to the first and last elements on the list. Here, we'll use both.

```
template <class T>
class List {
    //...
private:
    typedef ListElBase::Bits Bits;
    typedef ListEl<T> *Ptr;
    ListElBase clasp; // dummy node
    ListEl<T> *hd, *tl;
    void push( const T &value, Ptr &head, Ptr &tail );
    void pop( Ptr &head, Ptr &tail );
    friend class Iter<T>;
};
```

The implementation is a circular list; that is, the next element after the last element is the first element again, and vice versa. The circular list is rooted in a dummy first/last node of type `ListElBase`. The dummy node is of the base class type rather than `ListEl<T>` in order to avoid the necessity of instantiating an unused member of type `T` in the dummy node. Not only would this be wasteful, it would also preclude our instantiating a list with element types that did not support default construction. The curious `clasp` identifier comes from my visualizing this structure as a bead necklace, which is held together by a clasp that differs in structure from the beads. The `hd` and `tl` pointers simply refer to the first and last elements of the list, respectively, or to `clasp` if there are no beads in the necklace.

You Want Me To Push *Where*?

The push and pop operations are not too difficult, once you get the hang of dereferencing a two-way pointer. Let's look first at pushing.

```
template <typename T>
void List<T>::push( const T &v, Ptr &hd, Ptr &tl ) {
    ListEl<T> *newEl = new ListEl<T>( v );
```

```
newEl->ptr = Ptr( Bits(&clasp)^Bits(hd) );
hd->ptr = Ptr( Bits(hd->ptr)^Bits(newEl)^Bits(&clasp) );
clasp.ptr = Ptr( Bits(clasp.ptr)^Bits(newEl)^Bits(hd) );
if( empty() )
    tl = newEl;
hd = newEl;
}
```

The new list element is inserted between the clasp and the head of the list, so its pointer will refer simultaneously to these nodes ($\&\text{clasp}^{\wedge}\text{hd}$). The pointers in the clasp and (former) head node are fixed up to take the new node into account. For example, we reset the value of the (former) head node's pointer from referring to clasp and the node following the head node ($\&\text{clasp}^{\wedge}\&\text{next}$) to referring to the new head node and next ($\&\text{clasp}^{\wedge}\&\text{next}^{\wedge}\text{newEl}^{\wedge}\&\text{clasp} == \text{newEl}^{\wedge}\&\text{next}$).

Oops! I guess we should have called that operation `push_front`, to be conformant with other STL-like containers. That's easily fixed, and while we're at it, we'll implement `push_back`.

```
template <typename T>
void List<T>::push_front( const T &v )
    { push( v, hd, tl ); }
```

```
template <typename T>
void List<T>::push_back( const T &v )
    { push( v, tl, hd ); }
```

The use of the two-way pointers adds a pleasing note of symmetry to the circular list data structure. It doesn't really matter which is the head or the tail of the list, since traversal can proceed in either direction. This allows us to change a `push_front` into a `push_back` simply by labeling head as tail and vice versa. Implementation of the "pop" operations is similar.

```
template <typename T>
void List<T>::pop( Ptr &hd, Ptr &tl ) {
    Ptr n = Ptr( Bits(hd->ptr)^Bits(&clasp) );
    clasp.ptr = Ptr( Bits(tl)^Bits(n) );
    n->ptr = Ptr( Bits(n->ptr)^Bits(hd)^Bits(&clasp) );
    if( tl == hd )
        tl = n;
    delete hd;
    hd = n;
}
```

```
template <class T>
void List<T>::pop_front()
```

```
    { pop( hd, tl ); }

template <class T>
void List<T>::pop_back()
    { pop( tl, hd ); }
```

Fickle Iterators

We could use the two-way pointer implementation of the list nodes to produce a bidirectional list that uses half the pointer space of a more traditional implementation. Boring. Let's instead give our list container a forward iterator. A weird one.

```
template <class T>
class Iter
    : public std::iterator<std::forward_iterator_tag,
        T, std::ptrdiff_t> {
public:
    Iter( ListEl<T> *start, ListEl<T> *previous );
    Iter() {}
    Iter &operator ++();
    Iter operator ++(int);
    T &operator *();
    T *operator ->();
    bool operator ==( const Iter &that ) const;
    bool operator !=( const Iter &that ) const;
    void reverse();
private:
    typedef ListEl<T>::Bits Bits;
    typedef List<T>::Ptr Ptr;
    ListEl<T> *prev, *curr;
};
```

The iterator is implemented with a (plain old) pointer to the current element, and a pointer to the previous element. That second address is necessary to decode the two-way pointer in the list element to which the “current” pointer refers.⁴

```
template <typename T>
Iter<T> &Iter<T>::operator ++() {
    ListEl<T> *tmp = curr;
    curr = (ListEl<T> *) (Bits(curr->ptr) ^ Bits(prev));
```

```
    prev = tmp;
    return *this;
}
```

Dereference and equality operations are concerned only with the current node.

```
template <typename T>
T *Iter<T>::operator ->()
    { return &curr->el; }

template <typename T>
bool Iter<T>::operator ==( const Iter &that ) const
    { return curr == that.curr; }
```

What happens if we change the previous pointer in an iterator to refer to the node following the current node?

```
template <typename T>
void Iter<T>::reverse()
    { prev = Ptr( Bits(curr->ptr) ^ Bits(prev)); }
```

That's right, the iterator will now move in the other direction. Note that it is not a bidirectional iterator, since it cannot “back up” with a `--` operator. It can still move only in a forward direction, but it can change its mind with respect to what that direction is. This is the major reason that the iterator equality operators do not examine the `prev` pointer. If we compare two iterators into the same sequence, we are probably interested only in whether they refer to the same list element, and not with what direction they happen to be moving.

This brings up a rather unusual property of the end value of a list sequence produced by the `List` container. There are two of them, and which one applies depends on the direction of motion. (This is rather like the low-level values `+0` and `-0`, found on some computer architectures, which compare equal but are produced by different operations.)

```
template <typename T>
Iter<T> List<T>::begin()
    { return Iter<T>(hd, Ptr(&clasp)); }

template <typename T>
Iter<T> List<T>::end()
    { return Iter<T>( Ptr(&clasp), tl ); }
```

The list returns the end value that specifies not only that it is an end value, but that the previous position is the last element, or tail, of the list. Reversing the iterator returned by `List::end` will traverse the list “backwards.” The other reasonable value for `List::end` is `Iter<T>(Ptr(&clasp), hd)`, which compares equal to the other value of `List::end`, but when reversed will traverse the list elements in the “forward” direction.

It's fairly difficult to take advantage of this flexibility within the current STL framework, which has the

notions of forward and bidirectional iterators, but not fickle-forward iterators. But the capability can be useful. For example, the standard reverse algorithm requires a bidirectional sequence.

```
template <typename Bi>
void reverse( Bi first, Bi last ) {
    while( true )
        if( first == last || first == --last )
            return;
        else
            iter_swap( first++, last );
}
```

With a fickle-forward iterator, we can reverse a `List` with the same linear complexity.

```
template <class FF>
void ffReverse( FF first, FF last ) {
    last.reverse();
    while( true )
        if( first == last || first == ++last )
            return;
        else
            iter_swap( first++, last );
}
//...
List<std::string> names;
//...
ffReverse( names.begin(), names.end() );
```

Symmetric Lists

Of course, the standard STL `list` container has a linear reverse algorithm implemented as a member. So does `List`, but it's a constant time algorithm.

```
names.reverse();
```

Because of the symmetric structure of the circular list implementation—and more importantly—the symmetric structure of the nodes' two-way pointers, all `reverse` has to do is swap the head and tail of the list. A quick change to the implementation will handle that.

```
template <class T>
class List {
    //...
private:
```

```
//...
ListElBase clasp;
ListEl<T> *&hd, *&tl, *p1, *p2;
};
```

The head and tail pointers now refer indirectly to pointers to the actual head and tail nodes of the list. This allows us to use the uniform names `hd` and `tl` for these concepts while swapping the actual head and tail nodes easily.⁵ Reversing a `List`, as well as the direction of generated (but not existing) iterators, the type of end value returned by `List::end`, and the `List`'s insertion and deletion operations is accomplished by swapping `p1` and `p2`.

```
template <typename T>
void List<T>::reverse() {
    // We should use std::swap, but...
    (Bits &)p1 ^= (Bits &)p2;
    (Bits &)p2 ^= (Bits &)p1;
    (Bits &)p1 ^= (Bits &)p2;
}
```

It's Logical, After All

This column often deals with what was formerly common knowledge among competent C++ programmers that has not been communicated to newer C++ programmers. This poverty of history has consequences. Ignorance of the ban on cosmic hierarchies leads to dynamic casting, exceptions, and failure. Ignorance of problems associated with casting leads to code that cannot adapt itself automatically to remote maintenance. And ignorance of the simple and basic bitwise operators in C++ can lead to bloated and inefficient code. This ignorance also comes at a cost to the C++ apprentice, who does not experience the spare elegance of many of these techniques, and who will be a poorer programmer and person for the lack. It is certainly not the case that these established techniques should be used precisely as they were when they were first developed, but knowledge of both the new and the old allows them to be leveraged off each other in new, interesting, and useful ways. Such a combination can often put our newer techniques in a different light, and lead to a deeper understanding of them.

end of article

¹ <http://www.collegeboard.com/ap/students/compsci/subset04.html>

² Of course, we can encode any number of binary values in this way. However, the usefulness of the technique is diminished with multiple encoding, since $n-1$ “clear” values (or partial encodings of the $n-1$ clear values) are needed to decode an encoding of n values.

³ This is the reason for the waffle-words “STL-like,” rather than “STL-compliant” in the description of the container. We could produce an STL-compliant list container without too much difficulty, but the implementation would be much too long for the space we have available.

⁴ STL wonks may well argue that this implies that our list is not a “node-based” container, since remove operations (like our pop operations) in all node based containers affect only iterators to the removed elements. However, removal of the previous element will affect an iterator to the following element as well. Some picky readers might

also point out that I could avoid the temporary in the implementation of `operator ++` through the use of `^`. As an exercise, these readers are invited to rewrite the entire implementation of `List`, substituting `^` and `^=` for assignment, and eliminating all temporaries.

⁵ We used reference data members here so as to avoid having to rewrite any of our earlier source code. However, the use of reference or constant data members is generally a bad idea (just try to write `List::operator =`), and a pointer to a pointer would have been a better choice.