

Metaprogrammed Adapters

by Stephen C. Dewhurst

In this installment of Common Knowledge, we'll look at the use of some simple template metaprogramming in the design of a fairly unusual STL iterator adapter. However, its implementation illustrates techniques that can and should be commonly applied. And who knows? Even this adapter may be commonly used some day.

Supersequences and Multin

Perhaps the most brilliantly simple aspect of the architecture of the STL is that generic algorithms are designed to work with sequences rather than containers. Among other benefits, this provides the flexibility to work with any valid subsequence of a more complete sequence. For example, we could choose to sort only a portion of a container's content, or examine only every third element of a complete sequence, and so on.

```
vector<int> data;
//...
const int n = data.size()/4;
bubblesort( data.begin()+n, data.end()-n );
```

What if the sequence of interest is not contained within a single container or other source? A commonplace and effective approach is to copy the values of sequences of interest into a single sequence, and work with the new, combined sequence. A less common and, admittedly, often less effective approach would be to design an iterator adapter that can combine two sequences into a supersequence without copying. Let's call this adapter a `MultiIn`, since it will be adapting two input sequences to behave as a single input sequence.

Let's assume that we'd like to iterate through a supersequence defined by, potentially, two different types of input iterator. As with the `MultiOut` iterator adapter we designed in the previous installment of this column, we arbitrarily define the value type to be that of the first template argument. Note that, unlike the corresponding case in `MultiOut`, there really is no opportunity for conversion between the value types of `In1` and `In2`. This is because the sequence element to which an input iterator refers may be read, and this is accomplished by returning a reference to the sequence element when the iterator is dereferenced with `operator *`.

```
template <typename In1, typename In2>
class MultiIn : public std::iterator<
    std::input_iterator_tag, // category is input
    typename std::iterator_traits<In1>::value_type,
```

```
ptrdiff_t> { //...
```

What about the iterator category? Wouldn't it be nice if the `MultiIn` were a forward iterator if possible? It would.¹

```
template <typename In1, typename In2>
class MultiIn : public std::iterator<
    typename BothFor<In1, In2>::Result, // category depends...
    typename std::iterator_traits<In1>::value_type,
    ptrdiff_t> { //...
```

Compile Time Questions and Template Metaprogramming

Much of what is considered “advanced” generic programming has the undeserved reputation of being difficult.² However, there are many simple generic programming techniques that can be used to make our code more flexible without increasing its complexity. Consider the situation when you'd like to say, “If the following condition is true, then the type is A, otherwise it's B.” This is easy to accomplish with class template partial specialization.

```
template <bool cond, typename A, typename B>
struct Select {
    typedef A Result;
};
template <typename A, typename B>
struct Select<false, A, B> {
    typedef B Result;
};
```

For example, we can select the type of a variable based on compile-time arithmetic.³

```
const Select<__LINE__ <= CHAR_MAX, char, long>::Result
    curLine = __LINE__;
```

An expansion of the `Select` template evaluates a condition at compile time, then instantiates one of two versions of the template depending on the Boolean result of the expression. It's a compile time if-statement that says, “If the condition is true, the nested `Result` type is A, otherwise it's B.”

We can use simple building blocks like this to perform what is often referred to as “template metaprogramming”: using templates to perform calculation, control flow, and type selection at compile time. In the case of `MultiIn`, we want to determine whether both `In1` and `In2` are forward iterators, and if so, to set the iterator category for `MultiIn` to be `std::forward_iterator_tag` rather than `std::input_iterator_tag`.

```
template <typename In>
struct IsFor {
    enum { isfor = Conversion<
        typename std::iterator_traits<In>::iterator_category,
```

```

        std::forward_iterator_tag>::exists };
typedef Select<isfor,
        std::forward_iterator_tag,
        std::input_iterator_tag>::Result Result;
};
template <typename In1, typename In2>
struct BothFor {
    enum { result = IsFor<In1>::isfor && IsFor<In2>::isfor };
    typedef Select<result,
        std::forward_iterator_tag,
        std::input_iterator_tag>::Result Result;
};

```

Implementing Multilin

The most straightforward way to represent a pair of sequences is as a pair of sequences. However, it is not necessarily possible to distinguish the end of one sequence from the end of another. For example, a sequence obtained from a node-based container like `std::list` might implement the end value of any sequence that is not part of a larger sequence as a null pointer to a node. A more effective approach is to maintain a flag to indicate in which sequence the iterator is currently resident. We also don't care about the end value of the second sequence, unless we're constructing an end value for the `MultiIn` itself, but we must record an end value for the first sequence in order to recognize when it's time to proceed to the second.

```

template <typename In1, typename In2>
class MultiIn : public std::iterator<
    typename BothFor<In1, In2>::Result,
    typename std::iterator_traits<In1>::value_type,
    ptrdiff_t> {
    //...
    MultiIn( In1 b1, In1 e1, In2 b2 )
        : c1_(b1), e1_(e1), c2_(b2), first_(b1!=e1) {}
    MultiIn( In1 e1, In2 e2 )
        : c1_(e1), e1_(e1), c2_(e2), first_(false) {}
    MultiIn()
        {}
    //...
    In1 c1_; // first
    In1 e1_; // end of first

```

```

    In2 c2_; // second
    bool first_; // in first sequence?
};

```

The first constructor is used to create a `MultiIn` from two input sequences, whereas the second constructor is used to create an end value for the `MultiIn` sequence itself. This is a rather clunky and anti-intuitive interface. We'll look at this problem and try to improve the situation a bit later.

Implementing increment is straightforward.⁴

```

MultiIn &operator ++() {
    if( first_ ) {
        if( ++c1_ == e1_ )
            first_ = false;
    }
    else
        ++c2_;
    return *this;
}

MultiIn operator ++( int ) {
    MultiIn tmp( *this );
    ++*this;
    return tmp;
}

```

The implementation of the dereference operators is a little more nuanced. A first attempt might simply return the `value_type` of the sequence.

```

typedef typename std::iterator_traits<In1>
    ::value_type value_type;
value_type &operator *() const {
    if( first_ )
        return *c1_;
    else
        return *c2_;
}

value_type *operator ->() const
    { return &operator *(); }

```

However, the return value for these operators may have to vary depending on whether one of `In1` or `In2` is a “pure” input iterator, like an `istream_iterator`, rather than an input iterator that is also a forward iterator and can therefore be written, like a `list::iterator`. In that case, the return value of `In1::operator *` or `In2::operator *` will be `const value_type &`, rather than simply

`value_type &`, and this should be reflected in the return value of `MultiIn::operator *`. We can accomplish this with a little more compile time type selection.⁵

```
typedef typename
    Select<BothFor<In1,In2>::result,
        value_type &, const value_type &>
    ::Result R;
R operator *();
typedef typename
    Select<BothFor<In1,In2>::result,
        value_type *, const value_type *>
    ::Result P;
P operator ->();
```

Compile Time Conditionals and Selective Instantiation

Our first problem occurs with the implementation of the equality operators. The implementation of `operator ==` is fundamentally different depending on whether `In1` and `In2` are the same type or not. If they are the same type, then the following implementation is legal.

```
bool operator ==( const MultiIn &that ) const {
    // if In1 is same type as In2
    In1 i1( first_ ? c1_ : c2_ );
    In1 i2( that.first_ ? that.c1_ : that.c2_ );
    return i1 == i2;
}
```

If the types are different, a somewhat more complex implementation is required.

```
bool operator ==( const MultiIn &that ) const {
    // if In1 is different type from In2
    if( first_ )
        if( that.first_ )
            return c1_ == that.c1_;
        else
            return false;
    else
        if( that.first_ )
            return false;
        else
            return c2_ == that.c2_;
```

```
}
```

Note that the first version of `operator ==` is illegal if `In1` is different from `In2`. The second version will compile, but is incorrect for the case where `In1` is the same as `In2`, because it is optimized with the assumption that `In1` is a different type from `In2`, and can therefore avoid a direct comparison of the iterators in two of the four cases. It could be modified to handle the case where `In1` and `In2` are the same type, but that would penalize instantiations where the types differ. It's the moral equivalent of a type-based conditional executed at runtime, and is just not done. To avoid the runtime type-based conditional, we'll execute the conditional code at compile time.

```
struct Yes {};  
struct No {};  
template <typename A, typename B>  
struct IsSame {  
    enum { is_same = false };  
    typedef No Result;  
};  
template <typename A>  
struct IsSame<A,A> {  
    enum { is_same = true };  
    typedef Yes Result;  
};
```

Here we have another use of class template partial specialization for implementing a compile time conditional.⁶ We'll use the conditional to select the appropriate implementation of `operator ==` through the use of overload resolution.

```
bool compare( const MultiIn &that, Yes ) const {  
    // if In1 is same type as In2  
    //...  
}  
bool compare( const MultiIn &that, No ) const {  
    // if In1 is different type from In2  
    //...  
}  
bool operator ==( const MultiIn &that ) const  
    { return compare( that, IsSame<In1,In2>::Result() ); }
```

This a version of the same trick we used in an earlier installment of this column to choose among sorting algorithms based on iterator category.⁷ In this case, we're selecting the implementation of `operator ==` based on whether or not `In1` and `In2` are the same type. Note that only one implementation of `compare` will be called and instantiated. The uncalled version of `compare` will not be instantiated, and no compile time error will result.

Full Implementation Specialization

At this point, we recognize that many other aspects of the implementation of `MultiIn` can be improved if they are special-cased according to whether `In1` and `In2` are the same type. We could continue in the same vein as we did with `operator ==`, special-casing each operation. However, if most of the implementation will differ, it makes more sense to use partial specialization on the `MultiIn` template as a whole.

The unspecialized version handles the general case where `In1` differs from `In2`.

```
template <typename In1, typename In2>
class MultiIn : public std::iterator<
    typename BothFor<In1, In2>::Result,
    typename std::iterator_traits<In1>::value_type,
    ptrdiff_t> {
public:
    MultiIn( In1 b1, In1 e1, In2 b2 );
    MultiIn( In1 e1, In2 e2 );
    MultiIn();
    //...
    R operator *() const;
    P operator ->() const;
    MultiIn &operator ++();
    MultiIn operator ++(int);
    bool operator ==( const MultiIn &that ) const;
    bool operator !=( const MultiIn &that ) const;
private:
    In1 c1_, e1_;
    In2 c2_;
    bool first_;
};
```

The specialization handles the case where `In1` and `In2` are the same.

```
template <class In>
class MultiIn<In, In> : public std::iterator<
    typename IsFor<In>::Result,
    typename std::iterator_traits<In>::value_type,
    ptrdiff_t> {
public:
    typedef typename std::iterator_traits<In>
```

```

        ::value_type value_type;
MultiIn( In b1, In e1, In b2 )
    : c_(b1), e1_(e1), b2_(b2), first_(b1 != e1)
    { if( !first_ ) c_ = b2; }
MultiIn( In e1, In e2 )
    : c_(e2), e1_(e1), b2_(e2), first_(false) {}
MultiIn()
    {}
//...
R operator *() const;
P operator ->() const;
MultiIn &operator ++();
MultiIn operator ++(int);
bool operator ==( const MultiIn &that ) const;
bool operator !=( const MultiIn &that ) const;
private:
    In c_, e1_, b2_;
    bool first_;
};

```

Another advantage of specialization is that it allows us to change the actual content of the template rather than simply select among different versions of operations. In this case, choosing a different set of data members allows us to simplify a number of other operations in addition to `operator ==`.

```

value_type &operator *() const
    { return *c_; }

MultiIn &operator ++() {
    ++c_;
    if( first_ && c_ == e1_ ) {
        c_ = b2_;
        first_ = false;
    }
    return *this;
}

bool operator ==( const MultiIn &that ) const
    { return c_ == that.c_; }

```

Cascading Helpers

As we noted earlier, the interface for creating `MultiIns` is less than ideal.

```
MultiIn<deque<int>::iterator, list<int>::iterator>
    mibegin( aDeque.begin(), aDeque.end(), aList.begin() );
MultiIn<deque<int>::iterator, list<int>::iterator>
    miend( aDeque.end(), aList.end() );
```

To simplify things for our users, we'll provide helper functions to deal with common cases. The following helpers can be used to generate `MultiIns` for the beginning and ending of supersequences extracted from pairs of STL-compliant containers.

```
template <class C1, class C2>
MultiIn<typename C1::iterator,typename C2::iterator>
multiIn( C1 &c1, C2 &c2 ) {
    typedef typename C1::iterator I1;
    typedef typename C2::iterator I2;
    return MultiIn<I1,I2>( c1.begin(), c1.end(), c2.begin() );
}
```

```
template <class C1, class C2>
MultiIn<typename C1::iterator,typename C2::iterator>
multiInEnd( C1 &c1, C2 &c2 ) {
    typedef typename C1::iterator I1;
    typedef typename C2::iterator I2;
    return MultiIn<I1,I2>( c1.end(), c2.end() );
}
```

Since a `MultiIn` is an input iterator, and perhaps a forward iterator as well, it can be cascaded to provide traversal over an unbounded number of sequences. Helpers are a help here as well.

```
template <class C1, class C2, class C3>
MultiIn< typename C1::iterator,
        MultiIn< typename C2::iterator,
                typename C3::iterator> >
multiIn( C1 &c1, C2 &c2, C3 &c3 ) {
    typedef typename C1::iterator I1;
    typedef typename C2::iterator I2;
    typedef typename C3::iterator I3;

    return MultiIn< I1, MultiIn<I2,I3> >
```

```
    (
        c1.begin(),
        c1.end(),
        multiIn( c2, c3 )
    );
}
```

Manipulating Supersequences with MultiIn

Now we can, finally, do some serious work with supersequences. For example, we can bubblesort a supersequence constructed from two deques and a list.

```
deque<E> d1;
list<E> l1;
deque<E> d2;
//...
bubblesort( multiIn( d1, l1, d2 ), multiInEnd( d1, l1, d2 ) );
copy( multiIn( d1, l1, d2 ), multiInEnd( d1, l1, d2 ),
      ostream_iterator<E>( cout, " " ) );
```

At the end of the sort operation, we'll find the original sequences of objects in their respective containers, but their values will have been configured into sorted order. Note that this is a different operation with a potentially very different result from concatenating the component sequences into a single sequence before sorting.

```
cout << "\nSTART: ";
copy(d1.begin(), d1.end(), ostream_iterator<E>( cout, " " ) );
cout << "\nMIDDLE: ";
copy(l1.begin(), l1.end(), ostream_iterator<E>( cout, " " ) );
cout << "\nEND: ";
copy(d2.begin(), d2.end(), ostream_iterator<E>( cout, " " ) );
cout << endl;
```

Uncommonly Useful Techniques

`MultiIn` is not likely to be as commonly useful as other iterator adapters, but the techniques used in its implementation could and should be commonly used. Often, a little template metaprogramming can go a long way toward making components both flexible and efficient. In the case of our implementation of `MultiIn`, we were able to examine the characteristics of its initializing iterators at compile time, and configure the implementation appropriately. We achieved maximal flexibility and efficiency using compile time metaprogramming, based on information available in the precise context of each `MultiIn`'s instantiation.

end of article

¹ We are now entering into the realm of class template partial specialization, which is, tragically and inexcusably, not universally supported. It would also be nice if `MultiIn` supported the other iterator categories and a proper `difference_type` rather than assume it's `ptrdiff_t`. This is certainly possible, but we don't have the space here to examine such an involved implementation.

² This misconception is largely due to the efforts of Andrei Alexandrescu and his *Modern C++ Design* (Addison Wesley, 2001). Joke. The implementation of `Select` is adapted from Chapter 2 of *Modern C++ Design*, where the curious reader may find an implementation of the `Conversion` template used later in this section.

³ This is one of the few examples where formatting is directly related to the correctness of a program. This code has a bug.

⁴ Note that, for brevity and presumably for clarity, I've omitted the enclosing class template definitions for template member functions.

⁵ However, the reader may very likely find that the compiler is not up to the task! In fact, the code presented here should really be written more simply:

```
typedef typename
    Select<BothFor<In1, In2>::result,
        value_type, const value_type>
    ::Result R;

R &operator *();
R *operator ->();
```

But some otherwise reliable compilers balked at this version. This is one of the problems of being on the bleeding edge, but we can expect most platforms will be able to translate code like this in the “near” future.

⁶ The functionality of the `IsSame` template is subsumed by that of the `Conversion` template, but the `IsSame` implementation is instructive in that it shows how simple a metaprogramming solution can be.

⁷ S. C. Dewhurst, *Conventional Generic Algorithms*, CUJ, December 2001.