

Output Iterator Adapters

by Stephen C. Dewhurst

In the last installment of this column, we examined how to design generic algorithms for flexibility and efficiency. In this installment of Common Knowledge, we'll look at an indirect mechanism for algorithm customization; the design of output iterator adapters. In effect, a properly designed iterator adapter can be used to customize a generic algorithm as effectively as production of an entirely new algorithm.

Standard Adapters

Let's start with a look at two singularly pleasant and useful iterator adapters drawn from the C++ standard library. Consider the problem of copying a sequence into a container. The best procedure is generally to leverage an existing generic algorithm, since this approach has the advantage of correctness, efficiency, reuse, and self-documentation.

```
extern vector<State> &src;
//...
list<State> res;
copy( src.begin(), src.end(), res.begin() ); // boom.
```

The problem is that the `copy` algorithm assumes the result sequence is large enough to hold the source sequence; no checking is (or, generally, can be) done.

```
template <typename In, typename Out>
Out copy( In b, In e, Out r ) {
    while( b != e ) {
        *r = *b; // problem area
        ++b;
        ++r;
    }
    return r;
}
```

We could preallocate a result sequence of the appropriate length, but this is potentially wasteful and definitely socially gauche.¹

```
list<State> res2( src.size() );
copy( src.begin(), src.end(), res2.begin() ); // often works...
```

Another alternative would be to dispense with `copy`, and just hand code the operation using `list`'s `push_back` operation.

```
for( vector<State>::iterator i( src.begin() );
    i != src.end(); ++i )
    res.push_back( *i );
```

What we'd really like is the effect of a `push_back` while retaining the ability to use a standard generic algorithm. Note that there is nothing in the definition or implementation of the `copy` algorithm that requires an assignment operator to perform an assignment. All that's required is that the assignment syntax be supported and that the semantics do something reasonable. ("Reasonable," as we shall see throughout this column, is a delightfully flexible term.) The standard approach to this copy problem is to employ a `back_insert_iterator`.

```
copy( src.begin(), src.end(), back_inserter( res ) );
```

A `back_insert_iterator` is initialized with a container that supports the `push_back` operation, and defines assignment to perform a `push_back` into the container. The syntax of use of the resulting iterator is impeccable; it just changes the meaning of assignment to something more appropriate. Let's look at a typical implementation.

```
template <class Cont>
class back_insert_iterator
    : public std::iterator<std::output_iterator_tag,
        void,void,void,void> {
public:
    explicit back_insert_iterator( Cont &c )
        : c_(&c){}
    back_insert_iterator &
    operator =( const typename Cont::value_type &v ) {
        c_->push_back( v );
        return *this; }
    back_insert_iterator &operator *()
        { return *this; }
    back_insert_iterator &operator ++()
        { return *this; }
    back_insert_iterator operator ++(int)
        { return *this; }
private:
    Cont *c_;
};
```

Like all good, STL-compliant iterators that are implemented as classes, `back_insert_iterator` is derived from the standard `iterator` base class. This ensures that the iterator defines the proper set of

type names, and can be interrogated at compile time as to its properties.² The only property *typically* of interest to users of a `back_insert_iterator` is the iterator category, defined appropriately as the standard `output_iterator_tag` type. Most of the operations are no-ops, except assignment, which is defined to perform the `push_back`. This means our “problem area” line in the `copy` algorithm above, written as `*r = *b;`, will be translated as

```
(r.operator *()).operator =( *b );
```

The `operator *` just returns its object, and `operator =` inlines the `push_back`, so we’re left with

```
r.c_>push_back( *b );
```

Simple and effective. The syntax required for instantiating and initializing a `back_insert_iterator` is tedious enough for us to employ a helper function.

```
template <class Cont>
back_insert_iterator<Cont>
back_inserter( Cont &c )
{ return back_insert_iterator<Cont>( c ); }
```

In effect, we use the compiler’s ability to perform template argument deduction for function templates to instantiate the class template for us.

```
copy( src.begin(), src.end(),
      back_inserter(res) ); // with helper
copy( src.begin(), src.end(),
      back_insert_iterator< list<State> >(res) ); // without
```

Raw Deal

Consider a class type that has non-encapsulated resources, like a financial bond deal with an associated pricing algorithm.³

```
class Bond {
public:
    Bond &operator =( const Bond &that ) {
        if( this != &that ) {
            //...
            delete m_; // problem area
            m_ = that.m_>clone();
        }
        return *this;
    }
    //...
private:
    //...
```

```

    PVModel *m_;
};

```

The line indicated as a problem area is perfectly correct. An assignment operation is similar (but not identical!) to a destruction of the target of the assignment followed by its initialization; it's perfectly proper for a bond's assignment to delete its existing pricing model before attaching a new one. But what if we engage in foul trickery?

```

extern Bond x;
Bond *buf = static_cast<Bond *>
    (malloc( sizeof(Bond) )); // raw storage...
Bond &rx = *buf; // foul trickery...
rx = x; // probable error!

```

An assignment operator must assume that it is working with two objects, but in this case we're dealing with one object (the right argument) and a block of uninitialized storage (the left argument). As a result, we may attempt to delete a bad pointer value. Code like the above is actually fairly hard to produce in isolation, except as an act of malice aforethought. However, it's not uncommon to use the standard library to leverage an error like this into existence.

```

extern Bond a[10];
Bond *ary = static_cast<Bond *>(malloc( 10*sizeof(Bond) ));
copy( a, a+10, ary ); // oops!

```

In this case, the clever author of this code is attempting to circumvent the cost or unavailability of a default initialization of an array of Bonds by copying into uninitialized storage. What is really needed here is not assignment, but copy construction. Well, why not? A `raw_storage_iterator` reinterprets assignment to do just that.

```

template <typename Out, typename T>
class raw_storage_iterator
    : public std::iterator<std::output_iterator_tag,
        void,void,void,void> {
public:
    explicit raw_storage_iterator( Out cur )
        : cur_(cur) {}
    raw_storage_iterator &operator *()
        { return *this; }
    raw_storage_iterator &operator =( const T& element );
    raw_storage_iterator &operator ++()
        { ++cur_; return *this; }
    raw_storage_iterator operator ++( int ) {
        raw_storage_iterator tmp( *this );
        ++*this;
    }
};

```

```

        return tmp;
    }
private:
    Out cur_;
};

```

As with a `back_insert_iterator`, we are primarily concerned with the semantics of assignment, but we must also implement the increment operations on the underlying iterator. I understand that some readers may be upset to see that the dereference operator returns a `raw_storage_iterator` rather than an object of type `T`. However, a `raw_storage_iterator` is an output iterator, and only usage that employs both dereference and assignment must have a useful meaning. In other words, if `o` is an output iterator, and `v` is a compatible value, then only `*o = v` is required to mean anything useful.⁴

It is impossible to call a constructor directly, or to take its address, so assignment uses the placement form of `new` to trick the compiler into invoking a copy constructor.

```

template <class Out, class T>
raw_storage_iterator<Out,T> &
raw_storage_iterator<Out,T>::operator =( const T &v ) {
    T *elem = &*cur_; // cur_ is initializing iterator
    new ( elem ) T(v); // placement and copy constructor
    return *this;
}

```

Now we can use a `raw_storage_iterator` to copy into uninitialized storage without untoward side effects. Note that no helper function is provided. A `raw_storage_iterator` is typically used deep in the implementation of some other feature that is written and maintained by troll-like programmers who actually prefer the arcane template instantiation syntax.

```

Bond *ary = static_cast<Bond *>(malloc( 10*sizeof(Bond) ));
copy( a, a+10,
      raw_storage_iterator<Bond *, Bond>( ary ) );

```

Which Way? Both!

Now that we've seen a couple of useful iterator adapters, let's engage in a somewhat less serious design problem. Suppose we'd like to write to several different output streams simultaneously, rather than in sequence. For example, we might want to copy the same sequence to both a vector and to an output stream, or to a vector and the end of a list, and so on. Let's design a `MultiOut` iterator adapter that allows us to do just that.

```

Bond a[12];
list<Bond> lb;
vector<Bond> vb( 6 );
Bond *b = static_cast<Bond *>(malloc( sizeof(a) ));

```

```
//...
copy( a, a+6,
      multiOut( vb.begin(),
                ostream_iterator<int>(cout, " ") ) );
copy( a, a+6,
      multiOut( vb.begin(), back_inserter(lb) ) );
copy( a, a+6,
      multiOut( lb.begin(),
                raw_storage_iterator<Bond *, Bond>(b) ) );
```

Our first task is to make sure that our new iterator type properly defines its iterator traits, and identifies itself as an output iterator. As before, we'll employ the standard iterator base class. A `MultiOut` adapter is instantiated with two output iterator types.

```
template <typename Out1, typename Out2>
class MultiOut : public std::iterator<
    std::output_iterator_tag,
    // this is an output iterator
    typename std::iterator_traits<Out1>::value_type,
    // use Out1's value
    void>;
    // difference_type not used
```

Note that, in specifying the second argument to the iterator base class, we arbitrarily chose the `value_type` of the first template argument to be the `value_type` of the `MultiOut`. This will allow us to instantiate a `MultiOut` with different value types, provided that the `value_type` of `Out1` can be converted implicitly to that of `Out2`.

```
int a[] = { 1,2,3,4,5,6 };
list<int> li;
vector<double> vd;
deque<int> di( 6 );
copy( a, a+6,
      multiOut( di.begin(), back_inserter(vd) ) );
```

We initialize a `MultiOut` with two output iterators, and provide a helper function to perform the instantiation.

```
template <typename Out1, typename Out2>
class MultiOut : public iterator< /*...*/ > {
public:
    MultiOut( Out1 a, Out2 b ) : _a( a ), _b( b ) {}
    //...
```

```

private:
    Out1 _a;
    Out2 _b;
};

template <typename Out1, typename Out2>
MultiOut<Out1,Out2> multiOut( Out1 a, Out2 b )
    { return MultiOut<Out1,Out2>( a, b ); }

```

We provide the standard output operator operations of increment and “dereferenced assignment” to perform those operations on each subsidiary iterator in sequence.⁵

```

template <typename Out1, typename Out2>
class MultiOut : public iterator< /*...*/ > {
public:
    //...
    MultiOut &operator ++()
        { ++_a; ++_b; return *this; }
    MultiOut operator ++(int)
        { MultiOut tmp( *this ); ++*this; return tmp; }
    MultiOut &operator =( const value_type &v )
        { *_a = v; *_b = v; return *this; }
    MultiOut &operator *()
        { return *this; }
    //...
};

```

That’s all there is to it. However, it might also be nice to write simultaneously to three, four, or more output streams. The simplest way to achieve this is to recognize that a `MultiOut` is, itself, an output iterator. We can then provide additional helper functions to cascade `MultiOuts` for us.

```

template <typename Out1, typename Out2>
struct M2{ typedef MultiOut<Out1,Out2> T; };

template <typename Out1, typename Out2, typename Out3>
struct M3{ typedef MultiOut<Out1,M2<Out2,Out3>::T> T; };

template <class Out1, class Out2, class Out3, class Out4>
struct M4{ typedef MultiOut<Out1,M3<Out2,Out3,Out4>::T> T; };

template <class Out1, class Out2, class Out3>

```

```

M3<Out1,Out2,Out3>::T
multiOut( Out1 a, Out2 b, Out3 c ) {
    return M3<Out1,Out2,Out3>::T( a, multiOut(b,c) );
}

template <typename Out1, typename Out2,
          typename Out3, typename Out4>
M4<Out1,Out2,Out3,Out4>::T
multiOut( Out1 a, Out2 b, Out3 c, Out4 d ) {
    return M4<Out1,Out2,Out3,Out4>::T( a, multiOut(b,c,d) );
}

```

This gives us the convenience of broadcasting to an unbounded (here, four) number of output sinks. This allows us to solve the common problem of writing simultaneously to the back of a vector, the front and back of a list, and the front of a deque.

```

copy( a, a+6,
      multiOut( back_inserter(vd), back_inserter( li ),
                front_inserter(li), front_inserter(di)) );

```

Standard Whining

Unfortunately, that last piece of code may not compile on all platforms. In fact, if we had been a little less circumspect with some of our earlier uses of `MultiOut`, we would have had the same portability difficulties. For example, our earlier example of copying to a vector and the end of a list is portable.

```

copy( a, a+6,
      multiOut( vb.begin(), back_inserter(li) ) );

```

However, if we reverse the arguments to the `multiOut` helper, the code may not compile on many platforms.

```

copy( a, a+6,
      multiOut( back_inserter(li), vb.begin() ) );

```

The problem is that the `back_insert_iterator` template, as we defined it above, has no `value_type`, or, more precisely, its `value_type` is defined to be `void`. For its part, the `MultiOut` template (arbitrarily) chooses the `value_type` of its first template argument to construct the argument used by its definition of `operator =, const value_type &`. If the first template argument neglects to define a `value_type`, then `MultiOut` will attempt to define an illegal argument type of `const void &`.

The reason `back_insert_iterator` may not define a `value_type` is that it is precisely an output iterator, without also being a more general iterator. (On the contrary, the iterator returned by `vb.begin()` is an output iterator, but it is also an input, forward, bidirectional, and random access iterator as well.) The reason output iterators are allowed to get away with this are a little obscure (at least, they are to me).⁶

The standard says in section 24.1 that “All iterators `i` support the expression `*i`, resulting in a value of some class, enumeration, or built-in type `T`, called the *value type* of the iterator.” This is clearly not the case for most implementations of pure output iterators, like `back_insert_iterator` and `raw_storage_iterator`, where operator `*` returns a proxy object for use in “dereferenced assignment,” typically a reference to the iterator itself. In section 24.1.2 we see that operator `*` may not necessarily be used by itself on an output iterator, but only in the context of dereferenced assignment. This restriction may be used, I suppose, to exempt the earlier statement of section 24.1. In my opinion, a more useful definition of `value_type` is, quite naturally, the type of a value in the sequence. This is a concept that is well defined for all iterator types, and, as we’ve seen with the implementation of `MultiOut`, it renders the resulting iterator more useful. Most importantly, it would help us to avoid the problems of using constructs that may or may not support a particular feature; this leads to complexity and unportability.

The particular problem with `value_type` in the context of output iterators is that it often exists simply by accident, as an artifact of a particular implementation. For example, consider the declaration of assignment that we used in our implementation of `back_insert_iterator` above.

```
back_insert_iterator<Cont>&
    operator =( const typename Cont::value_type &v );
```

Another perfectly rational implementation might have employed a typedef.

```
typedef typename Cont::value_type value_type;
back_insert_iterator &
    operator =( const value_type &v );
```

This second version of `back_insert_iterator` defines a `value_type` within the template, and can, therefore, be used as the first argument to the `MultiOut` template. This is the kind of undocumented and unstable state of affairs that makes our lives miserable when porting to a new environment or when accepting a new version of an existing library. It would be much better, I think, for the standard to require that every output iterator define a `value_type`.

```
template <class Cont>
class back_insert_iterator
    : public std::iterator<std::output_iterator_tag,
        typename Cont::value_type,
        void>;
```

Stepping Back, Going Forward

STL generic algorithms can be customized by a variety of mechanisms. As we have seen in the previous installment of Common Knowledge, a generic algorithm can be customized through compile time queries of its iterator’s capabilities and by allowing customization of its predicates and comparators. In this installment, we saw how to modify the semantics of a generic algorithm indirectly, by modifying the semantics of its output iterators through the use of output iterator adapters. In effect, a properly designed generic algorithm is often only a high-level suggestion of an algorithm’s intent, whose concrete semantics are determined by many different components. We’ll continue in this vein in the next installment of this column, where we’ll use some simple template metaprogramming to create an unusual input iterator adapter.

end of article

¹ Note that it also requires that the `State` class provide a default constructor, which is both costly and restrictive, since many useful classes do not offer defaults. Another alternative would be to dispense with the `copy` algorithm and employ an alternate form of the `list` constructor, as in `list<State> res2(src.begin(), src.end());` This will probably work on most compilers, depending on the implementation of `vector<State>::iterator`. However, the moderately different `list<State> res3(res2.begin(), res2.end());` will likely fail on compilers that do not support template member functions.

² See the previous installment of this column, “Conventional Generic Algorithms,” *C/C++ Users’ Journal*, October 2001. Note that some authors, including the venerable Dr. Stroustrup (*The C++ Programming Language*, 3rd edition, Addison Wesley, 1997, pp. 562-3), will on occasion derive an iterator from `std::iterator_traits` rather than `std::iterator`. Although either will have the same practical effect at present, I disagree with this approach. First, public inheritance from the `iterator` base class is an explicit statement that you are creating a new iterator type; there is no such implication when deriving from `iterator_traits`. Second, `iterator` and `iterator_traits` are separate entities, and there is no guarantee that they will always be interchangeable in this regard. For example, in the future, `iterator` may define additional information that is not present in `iterator_traits`.

³ For those who are keeping score, this is an instance of the Strategy pattern.

⁴ The best and most complete discussion of this and many other fine points of the STL is to be found in Matt Austern’s *Generic Programming and the STL*, Addison Wesley, 1999.

⁵ Note that the compiler will correctly (in this case) provide the destructor, copy constructor, and assignment operator for `MultiOut`.

⁶ It is also the case that output iterators do not have to define a `difference_type` or support the equality operators. The reasons behind this indecision are, I think, much more technically defensible than those surrounding the decision not to require a definition of `value_type`.