

Conventional Generic Algorithms

by Stephen C. Dewhurst

One thread that runs through the individual installments of this column is recognition of the pervasive and beneficial presence of idiom in mid-level and low-level design with C++. This adherence to idiom is nowhere more important than in the design of STL-compliant generic algorithms. In this installment of Common Knowledge, we'll look at some of the more common idiomatic "rules" for implementing these algorithms.

A Not So Simple Generic Algorithm

Consider a generic algorithm for swapping the values of two objects of the same type.

swap:

```
take two things
put the value of the first thing in a temp
put the value of the second thing in the first thing
put the value of the temp in the second thing
```

This representation is a basic outline of the implementation of the swap algorithm, but is imprecise about its lower-level meaning. Consider two potential implementations.

```
template <class T>
void swap1( T &a, T &b ) {
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}

template <class T>
void swap2( T &a, T &b ) {
    T tmp( a );
    a = b;
    b = tmp;
}
```

These implementations look similar, but can exhibit very different behavior and ranges of applicability. If we are interested in swapping the values of two integers, either version will succeed, and will most probably result in the same object code being generated. However, the implementations will start to show divergent behavior with more complex, user-defined types.

```
std::string a( "hello" ), b( "Goodbye" );
swap1( a, b ); // works, slow
swap2( a, b ); // works, faster
```

The first version of `swap` forces the default initialization of a `string` temporary that is then overwritten by an assignment. This is not an error, necessarily, but it is probably less efficient than simply initializing the temporary with the desired value, as the second `swap` implementation does. Things can get more complex.

```
struct R {
    //...
};
struct S {
    S();
    S &operator =( const S & );
    //...
};
struct T {
    T( int );
    T( const T & );
    T &operator =( const T & );
    //...
};
R r1, r2;
S s1, s2;
T t1(1), t2(2);
swap1( r1, r2 ); // might work...
swap2( r1, r2 ); // might work...
swap1( s1, s2 ); // works
swap2( s1, s2 ); // might work...
swap1( t1, t2 ); // error!
swap2( t1, t2 ); // works
```

The first implementation requires the presence of a default constructor, whereas the second requires a copy constructor. The implementation of a generic algorithm imposes a set of implicit requirements on its arguments. The requirements are both syntactic (is an assignment operator defined for this type?) and semantic (does assignment copy a value or make a telephone call?)

A moral relativist might conclude that either implementation could be correct, because each has areas of applicability not addressed by the other. That's wrong. The second implementation is correct, the first implementation is incorrect, and the reason is idiomatic. User-defined types in C++ that allow copying implement the copy operation idiom. This generally-understood-but-unwritten idiom says that any type that allows copying implements the operations using a copy constructor and assignment operator. Additionally, these operators have a de facto standard interface and meaning.¹ The second implementation of swap is written in the context of the copy operation idiom and the first is not. We shall see that the idiomatic context of the standard template library provides a framework for extending the STL with new generic algorithms.

An Unconventional Generic Algorithm

Let's look at an occasionally useful generic algorithm for sorting.²

```
template <class T>
void slowSort( T a[], size_t n ) {
    for( int i = 0; i < n; ++i )
        for( int j = i; j < n; ++j )
            if( a[j] < a[i] )
                swap( a[j], a[i] );
}
//...
int scores[] = { 82, 78, 93, 55, 98, 81 };
int n = sizeof(scores)/sizeof(scores[0]);
extern States states[50];
bool operator <( const State &, const State & );
//...
slowSort( scores, n );
slowSort( states, 50 );
```

Unfortunately, the requirements imposed by this algorithm on its arguments are both implicit and unconventional. In order to know precisely how to use the algorithm, a programmer must either read the implementation in detail (never a good idea), or depend on the presence of accurate documentation accompanying the declaration (possible, but unlikely).

An STL Generic Algorithm

Here's the same algorithm implemented in a more conventional style.

```
template <class Iter>
void slowsort( Iter s, Iter e ) {
    for( Iter i(s); i != e; ++i )
        for( Iter j(i); j != e; ++j )
```

```

        if( *j < *i )
            std::iter_swap( i, j );
    }
    //...
    slowsort( scores, scores+n );
    slowsort( states, states+50 );

```

STL-compliant generic algorithms are the subset of generic algorithms that “play by the STL rules.” These rules are essentially a set of normative expectations about conformance issues.

Algorithms, Iterators, and Documentation

A generic algorithm implicitly specifies requirements on its iterators through the syntax used to manipulate them and by the results expected by these operations.

```

template <class I, class T>
int myCount( I first, I last, const T &value ) {
    int n = 0;
    for( I i = first; i < last; ++i )
        if( *i == value )
            ++n;
    return n;
}

```

In this version of `count`, similar to the standard `std::count` algorithm, we traverse a sequence, maintaining a count of how many times a particular value occurs in a sequence. It doesn’t get much simpler than that, but we’ve still managed to go wrong in several important ways.

First, we’re making unnecessary demands on the qualifications of the iterator. This becomes obvious if we try to use `myCount` with a sequence drawn from a `list` or a `file`.

```

list<int> ilist;
//...
cout << "5 count: "
    << myCount( ilist.begin(), ilist.end(), 5 ) // error!
    << endl;
cout << "5 count: "
    << myCount (
        istream_iterator<int>(cin),
        istream_iterator<int>(), 5 ) // error!
    << endl;

```

As written, the algorithm requires a random access iterator, even though the algorithm’s logical structure requires only an input iterator. The culprit is the use of `operator <` in the loop condition--which is

available only with random access iterators--when the use of `operator !=` would have served just as well.

Another problem concerns the naming of the iterator type. Since the algorithm requires only an input iterator, it should be documented as such.³

```
template <typename InputIterator, typename T>
int myCount(
    InputIterator first,
    InputIterator last,
    const T &value );
```

The standard employs the sesquipedalian spellings `InputIterator`, `OutputIterator`, `ForwardIterator`, `BidirectionalIterator`, and the truly inspired `RandomAccessIterator` for these iterator capability classes. Stroustrup proposes a more compact de facto standard as `In`, `Out`, `For`, `Bi`, and `Ran`.⁴ I don't know about you, but I'll string along with Bjarne on this one. Whatever your decision, for readability it's best to employ one of these two conventions; don't make up your own.

```
template <typename In, typename T>
int myCount( In first, In last, const T &value ) {
    int n = 0;
    for( In i( first ); i != last; ++i )
        if( *i == value )
            ++n;
    return n;
}
```

Comparators and Predicates

Let's revisit our properly documented sorting algorithm.

```
template <typename For>
void slowsort( For s, For e ) {
    for( For i(s); i != e; ++i )
        for( For j(i); j != e; ++j )
            if( *j < *i )
                std::iter_swap( i, j );
}
```

This version of the algorithm is effective, but it's inflexible because it assumes that an `operator <` is available for the element type of the sequence. Earlier, we wanted to sort an array of `States`, and were forced to declare a non-member `operator <` to provide the comparison. Another problem is that it is difficult to change the sorting criterion. For example, we may want to sort `States` by name or by population, or we may want to sort integers in descending order or by their number of prime factors.

STL algorithms that employ a comparator conventionally allow the comparator to be supplied explicitly.

```
template <typename For, typename Comp>
void slowsort( For s, For e, Comp c ) {
    for( For i(s); i != e; ++i )
        for( For j(i); j != e; ++j )
            if( c( *j, *i ) )
                std::iter_swap( i, j );
}
```

The same is true for algorithms that employ predicates. The `std::for_each` algorithm performs an operation on each element of a sequence, then returns a copy of the operation. It would be more flexible to specify which elements of the sequence should be so manipulated.⁵

```
template <typename For, typename Op, typename Pred>
Op myForeach( For b, For e, Op op, Pred p ) {
    while( b != e ) {
        if( p( *b ) )
            op( *b );
        ++b;
    }
    return op;
}

template <typename T>
struct True : public std::unary_function<T,bool> {
    bool operator()( const T & ) const
        { return true; }
};
```

Generally, an implementation will provide two versions of an algorithm. One will use the “normal” default comparator or predicate (“less than” and “always true” respectively), and the other will allow the comparator or predicate to be supplied explicitly.

Compile Time Queries

We still have a problem with our version of the count algorithm.

```
template <typename In, typename T>
int myCount( In first, In last, const T &value );
//...
TerabyteContainer<int> c;
cout << "5 count: "
```

```
<< myCount( c.begin(), c.end(), 5 ) // oops!
<< endl;
```

What if the result is too big to be represented in an `int`? The best (or perhaps worst) that can happen is we'll silently truncate the result or return a large negative number. It's much better to ask the iterator what its `difference_type` is.⁶ For reasons that we will explore in a future installment of this column, we actually do not ask the iterator directly, but rather ask an ancillary traits class.⁷

```
template <typename In, typename T>
typename std::iterator_traits<In>::difference_type
myCount( In first, In last, const T &value ){
    typename std::iterator_traits<In>::difference_type n = 0;
    for( In i = first; i != last; ++i )
        if( *i == value )
            ++n;
    return n;
}
```

Note that we had to use the `typename` keyword to let the compiler know that the nested name `difference_type` within `iterator_traits<In>` was a type name. Otherwise, because the content of `iterator_traits<In>` is unknown at the time the `myCount` template is parsed, the compiler would have assumed that `difference_type` was not a type name, and the parse would have failed. Yes, if you're relatively new to templates and the STL, the syntax is challenging. Never fear. After a while it will start to look normal (although it's not guaranteed that you will still be regarded as normal by society in general by that point).

Now we can ask `TerabyteContainer<int>::iterator` what the return type of `myCount` should be, rather than assuming an `int` will be adequate. All STL iterators are capable of answering, through the corresponding instantiation of `iterator_traits`, the following questions about themselves:

1. What is the type of a temporary variable that can hold a value of the element type to which the iterator refers? `value_type`
2. What is the type of a pointer to the element? `pointer`
3. What is the type of a reference to the element? `reference`
4. What category of iterator is it; random access, bidirectional, forward, input, or output? `iterator_category`
5. What type is used to represent the distance between two iterators; the length of a sequence? `difference_type`

The first item, `value_type`, looks a little unusual. Why would we be interested in the type of a temporary rather than the type of the sequence element? Consider the following algorithm for finding the minimum value within a non-empty sequence.

```
template <typename In>
typename iterator_traits<In>::value_type
min_value( In first, In last ) {
```

```

    typename iterator_traits<In>::value_type min = *first;
    while( ++first != last )
        if( *first < min )
            min = *first; // requires non-const!
    return min;
}

```

If we want to determine the minimum value contained within a sequence of constant elements, `value_type` must be assignable. That is, if the element type is non-constant, `value_type` is the same as the element type. If the element type is constant, `value_type` strips the `const` qualifier from it.

```

int const g[] = { 9, 8, 1 };
int const minimum = min_value( g, g+3 ); // works!

```

Compile Time Conditionals

It is also often convenient to determine the capabilities of an iterator at compile time. For example, even though I have a strong preference for the `slowsort` algorithm, I recognize it may not be the best choice for sorting a 100,000-element vector. What we'd like to do is select the proper sorting algorithm at compile time so as not to generate any unnecessary code or incur any runtime cost.

An iterator's `iterator_category` tells us whether the iterator is input, output, forward, bidirectional, or random access. The `iterator_category` is one of five standard types arranged in a hierarchy.

```

struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag
    : public input_iterator_tag {};
struct bidirectional_iterator_tag
    : public forward_iterator_tag {};
struct random_access_iterator_tag
    : public bidirectional_iterator_tag {};

```

The is-a relationships within the hierarchy largely mirror the capabilities of the different iterator categories.⁸ For example, a random access iterator is also a bidirectional iterator, a forward iterator, an input iterator, and an output iterator. A forward iterator is also an input and an output iterator. Therefore, we can ask an iterator for its category by accessing its corresponding iterator traits, and then determine if that category fulfills the requirements of a particular iterator category by using the is-a relationships within the iterator tag hierarchy.

```

void informUs( forward_iterator_tag )
    { cout << "it's some kind of forward" << endl; }
void informUs( ... )
    { cout << "it's not forward" << endl; }
template <typename Iter>

```

```

void informUsAbout( Iter ) {
    typedef typename
        iterator_traits<Iter>::iterator_category Category;
    if( typeid(Category) ==
        typeid(bidirectional_iterator_tag) )
        cout << "it's exactly a bidirectional" << endl;
    else
        informUs( Category() );
}
//...
vector<State>::iterator i;
istream_iterator<int> j;
list<double>::iterator k;
vector<float> v;
informUsAbout( i ); // some kind of forward
informUsAbout( j ); // not forward
informUsAbout( k ); // exactly bidirectional
informUsAbout( back_inserter(v) ); // not forward

```

Cute, but a more practical use of these compile-time queries is to perform compile-time algorithm selection based on the category of iterator.

```

template <typename Ran>
inline void
mySortImpl( Ran b, Ran e, std::random_access_iterator_tag )
    { sort( b, e ); }

template <typename For>
inline void
mySortImpl( For b, For e, std::forward_iterator_tag )
    { slowsort( b, e ); }

template <typename For>
inline void mySort( For b, For e ) {
    typedef typename
        std::iterator_traits<For>::iterator_category Cat;
    mySortImpl( b, e, Cat() );
}

```

The `mySort` algorithm uses a combination of the ability to ask a question of an iterator, overloading, and inlining to achieve optimal algorithm selection with no runtime cost. This is a common technique used by many generic algorithms.

```
list<State> states;
deque<int> cards;
//...
mySort( states.begin(), states.end() ); // uses slowsort
mySort( cards.begin(), cards.end() ); // uses std::sort
```

Convention Rules!

Writing effective generic algorithms in the context of the STL requires adherence to a number of conventions. This column has examined several of the more common ones. In summary, STL-compliant generic algorithms obey the following conventions:

1. They use standard naming conventions to document proper use.
2. They are written to the least powerful iterator for which there is an efficient implementation.
3. They don't hard-code predicates or comparators, unless a more flexible alternative is also provided.
4. They may ask compile-time questions of iterators.
5. Type-based control decisions are made at compile time, not runtime.

end of article

¹ Of course, moral rectitude must also permit some degree of relativism. Even as fundamental an idiom as the copy operation idiom has exceptions. See S.C. Dewhurst, "Split Idioms," *C/C++ Users Journal* 19(6), June 2001 for one example.

² Yes, it's sometimes a better option than more sophisticated sorting algorithms with better asymptotic complexity. It's left as an exercise to the reader to figure out when and why.

³ This declaration illustrates an emerging useful convention. The language allows the keywords `typename` and `class` to be used interchangeably in this context, where either is used to indicate that the following template argument is a type name. However, some authors and programmers now use `class` to indicate that the argument must be of class type, and `typename` to indicate that the argument may be any type name, class or otherwise.

⁴ Bjarne Stroustrup, *The C++ Programming Language*, 3rd edition, Addison Wesley, 1997, p. 511.

⁵ Of course, the predicate that makes the selection could be embedded in the operation itself, but it is usually both clearer and more efficient to separate explicitly these two aspects of processing a sequence.

⁶ This is instead known, for reasons that are beyond my ability to forgive, as `distance_type` in "one major implementation." Does this cause problems? Yes it does.

⁷ For more on traits, see Andrei Alexandrescu, "Traits: The else-if-then of types" *C++ Report*, April 2000.

⁸ Note the curious fact that the tag type for forward iterators is derived only from the tag type for input iterators, rather than both the input and output tags. In *The C++ Programming Language*, 3rd edition, p. 554, Bjarne Stroustrup has the following to say: "Looking at the operations supported by input and forward iterators...we would expect `forward_iterator_tag` to be derived from `output_iterator_tag` as well as from

`input_iterator_tag`. The reasons that it is not are obscure and probably invalid. However, I have yet to see an example in which the derivation would have simplified real code.” In *Generic Programming and the STL*, p. 181, Matt Austern says, “[The inheritance hierarchy] has no deep significance. It is nothing more than a minor convenience...Inheritance from `output_iterator_tag` is left out because it is unnecessary for that purpose.”