Custom Iterators for the STL

Christopher Baus and Thomas Becker
Zephyr Associates Inc.
christopher@baus.net
thomas@styleadvisor.com

Abstract. We discuss several kinds of custom iterators for use with the STL that are substantially different from the iterators that come with the STL. We present class templates that implement these custom iterators in a generic manner.

Contents

Introduction

We will assume that the reader possesses a basic working knowledge of the C++ Standard Template Library (STL).

The iterators that come with the STL typically represent ranges within an STL container. In other words, the range of data items that is represented by a pair of these iterators is present in memory in the form of an STL container object. We have found many situations where a program needs to iterate through sequentially structured data, but this data is not or should not be present in memory in the form of a range in a container. The STL allows the client programmer to write custom iterators that gather or generate their data in ways other than pulling it sequentially out of a container. Writing such an iterator for a specific situation is a rather elementary task. However, it is possible to identify unifying concepts among these custom iterators and implement them generically.

One purpose of this article is to present several generic custom iterator class templates. Another, perhaps more important purpose is to demonstrate how understanding and applying the generic programming paradigm has helped us to unleash the full power of the STL.

STL Iterators and Custom Iterators

To put things in perspective, we begin by describing how our own use and understanding of the

STL has evolved over time. When we first decided to use the STL as a standard in our C++ code, we were not totally oblivious to the concept of generic programming. However, we did not have much of a feeling for how much it was going to be worth. To us, the STL was primarily a library of container classes that seemed to have been designed much more thoughtfully than other, proprietary libraries. Accordingly, we viewed the use of iterators mainly as a clever way to access the elements of a container and to pass them to algorithms: iterators were the glue between containers and algorithms. On this level, genericity of algorithms merely meant that one and the same algorithm could be used for ranges of elements stemming from containers of different types.

It did not take long for us to become quite enthralled with the elegance and ease of use afforded by the container-iterator-algorithm trinity. Therefore, the idea of letting our own classes expose iterators came quite naturally. More precisely, whenever a class contains sequentially organized data that it wishes to expose through its interface, it should do so by exposing STL-style iterators. One reason for doing this is that these iterators may be used to pass the data to custom or STL algorithms. Another, equally valid reason is ease of use and uniformity of style. As an example, consider the time series class that lies at the heart of every financial analytics application. This class represents the total historical return data of a financial instrument. The interface of this class tends to be on the fat side, and its internals are quite complex. At its core, however, is a simple sequence of doubles, typically implemented as an STL vector, or, if it is legacy code, as a C-style dynamically allocated array. In either case, access to the raw numbers is most conveniently provided through the use of iterators. Here, it is clear that the iterators will be passed to algorithms, e.g., in order to calculate statistics such as compound return or standard deviation.

Another example is a class that represents a path name. If it is expected that clients will want to iterate through the components of the path, it is convenient to do so using iterators. In this case, it is rather unlikely that the iterators will be passed to an algorithm. Using them is simply a matter of convenience, and also a way of increasing the readability and maintainability of the code through uniformity of style: when iterating, use iterators.

All these user-defined iterators have one thing in common with the iterators that the STL containers expose: they are essentially pointers. They refer to a data item that exists in a certain memory location and is part of a range of data items. Moreover, they know how to get to the next item in that sequence, and possibly also the previous one. The only possible difference between such an iterator and an actual pointer is that operations such as dereferencing or advancing may involve actions like retrieving the data within a node, or stepping from one node to the next. However, the STL would not be a very good example of generic programming if there were anything that required iterators to resemble pointers in this way. The requirements on iterators are, of course, much weaker than that. For a complete discussion of these requirements, see e.g. Chapter 7 of Matt Austern's book ([1]). In essence, all one has to observe when writing an iterator class is that, depending on the category that the iterator will belong to, certain operators such as dereferencing and incrementing must be implemented, and, again depending on the iterator category, certain plausibility conditions have to be met. For example, if you take a random access iterator that is not at its end position, derefence it, increment it, decrement it, and dereference it again, the two dereferenced values must be equal under operator==(). It is clear that none of this places any restrictions on the origin of the data that the iterator retrieves. Interestingly, at the very beginning of his book on the STL ([1], p.4), Matt Austern gives an example of an iterator that gathers data from a place other than a range in memory. His iterator line_iterator retrieves lines from an input stream. The operator++() reads the next line and stores it, and the operator*() returns that stored line.

The next step on our path to generic programming enlightenment was the realization that such "non-pointer-like" iterators came in handy in many situations. In fact, in some parts of our software,

they are now the rule rather than the exception. To see examples, consider an application that crunches numbers and then presents the results to the user in charts such as lines, scatter plots, bar charts, and so on. When the charting code draws these charts, it must essentially iterate through sequences of numbers and use them as coordinates in the appropriate manner. If you subscribe to the STL way of handling sequential data, you will of course let the number cruncher store its results in STL containers. The charting code retrieves these results through iterators: when iterating, use iterators. If the numbers that are stored represent the result of lengthy calculations, all is well. Now suppose the number cruncher has already stored two equally long sequences of numbers, and you need to expose to the client a third sequence representing the pointwise difference between these two sequences. (The excess return of your favorite mutual fund over the S&P500 comes to mind.) The naive way to do this is to store the sequence of differences in a third container and expose iterators to this container. It is clear that this is a horrible thing to do in terms of time vs. space trade-off. Taking the difference between two doubles is an operation of almost negligible complexity. Of course, we want to calculate this difference on demand rather than storing the entire sequence of differences. Do we have to abandon the STL iterator idiom to accommodate our efficiency requirements? Of course not, because an iterator does not have to iterate through an actual, physical range of data. It can gather or generate its data on the fly. For the particular purpose of the difference of two existing sequences, it is quite easy to see how to write an appropriate iterator class. This class holds two iterators, one into each of the two ranges. Operations such as advancing the iterator are implemented by performing the respective operation on both iterators. Dereferencing the iterator returns the difference between the dereferenced values of the two iterators. There are of course some immediate questions as to the kind of iterator we obtain here: const or non-const, what category? We will address these issues shortly. For now, we have one more step to climb on the stairway to generic programming heaven. If we were to write a custom iterator from scratch for every situation such as the one described above, we would be missing a level of genericity. It is possible to identify several commonly occurring concepts and implement them generically as class templates.

We will present five different custom iterator concepts here: combining iterators, processing iterators, dereferencing iterators, iterators to members, and generating iterators. Needless to say, we do not claim that these are comprehensive in any way. They are all based on class templates that were originally developed in the trenches as the need arose while working toward deadlines. One purpose of this article is to take a step back and apply some finishing strokes to get the genericity and the STL compliance right. The reader should not view these concepts as the ultimate solution to the problem of finding generic custom iterators. Rather, they should serve as an inspiration to be open-minded and creative about iterators, and more generally, as a reminder of rule number one when working with the STL: "Think Generic."

Combining Iterators

The concept of a combining iterator provides a generalization of the difference iterator that we discussed earlier. Roughly speaking, a combining iterator holds several iterators and a functional. Moving the combining iterator is implemented as moving the member iterators in parallel. Dereferencing the combining iterator is implemented as dereferencing the member iterators and applying the functional to the values thus obtained.

There are several design questions to be addressed. First of all, should we require that the individual iterators held by the combining iterator are all of the same type? Obviously, such a requirement places a restriction on the generality of the concept. On the other hand, having this requirement allows us to hold the member iterators in a collection rather than in individual member variables.

Thus, we can have one class that allows the number of member iterators to be specified at runtime, upon construction. To cover the more general case, where the individual iterators can have different types, we would have to write one combining iterator class with two member iterators, another one with three member iterators, and so on (unless, of course, we require the member iterators to be derived from a common base class, clearly a no-no in the world of STL iterators). In view of all this, it is clear that there is no easy answer to this design question. We have opted for the requirement that all member iterators must have the same type, simply because that was good enough for our needs. It will not be good enough for everybody's needs. (We have not tried to apply template-metaprogramming techniques in the design of the combining iterator. Using these techniques may result in a more general and more widely applicable solution.)

The second design question is whether we should make provisions for the combining iterator to be a non-const iterator. Whenever we have used a combining iterator, the point was to gather data from different places, process it through a functional, and return a value. Theoretically, one could conceive of an iterator that simultaneously modifies data in several places through several iterators. However, we have never seen a situation where it would make sense to use a combining iterator for that purpose. In our experience, combining iterators are hunters and gatherers whose job it is to get data, not to set it. Therefore, our combining iterators are meant to be const iterators. This is reflected by the typedefs inside our combining iterator and by the result type of its operator*():

```
template<
  typename _It, // type of member iterators
  typename _Func // type of functional
  >
class combining_iterator
{
public:
  typedef typename _Func::result_type value_type;
  typedef const typename _Func::result_type* pointer;
  typedef const typename _Func::result_type& reference;
  // ...
  value_type operator*() const;
  // ...
};
```

As you can see, we let the return value of the combining iterator's operator*() be the result type of the functional that the combining iterator uses. The functional, in turn, is forced to return its result by value and not by reference. If you use a functional that returns a reference, you get a compile-time error from the reference typedef. Thereby, we force the return value of operator*() to be a non-lvalue, and hence our combining iterator to be a const iterator. Normally, operator*() of an iterator returns the iterator's reference type. For a const iterator, that reference type is a const reference. The combining iterator is one of those rare cases where the appropriate return type for operator*() is the iterator's value type. As we said before, we do not see this as a real restriction. It has the advantage that we now know how to properly typedef the combining iterator's value_type, pointer, and reference.

The third question is, what is the iterator category of the combining iterator? The short answer is, it is the same as the category of the member iterators. That is what we set as the default by placing an appropriate typedef inside the combining iterator class. The catch is that this is true only if the functional that gets applied upon dereferencing has no state, i.e., is a function that deterministically returns equal values for equal arguments. If the functional holds state, then it is entirely possible that the member iterators of the combining iterator are random access iterators, and yet only one simple pass through the data is allowed, thus downgrading the combining iterator to an input iterator. Obviously, it must be the responsibility of the client to set the iterator category correctly in

that case. This can be done by providing a specialization or partial specialization, as the case may be, of the STL's iterator_traits for the combining iterator in question. With the basic design issues settled, we may now look at the complete implementation of the combining iterator class template:

```
template<
  typename _It, // type of member iterators
  typename _Func // type of functional
  >
class combining_iterator
{
public:
  typedef typename
    std::iterator_traits<_It>::iterator_category
    iterator_category;
  typedef typename _Func::result_type value_type;
  typedef typename
    std::iterator_traits<_It>::difference_type
    difference_type;
  typedef const typename _Func::result_type* pointer;
  typedef const typename _Func::result_type& reference;

  // Construct an uninitialized combining iterator.
  combining_iterator(){}

  // Construct a combining iterator from a collection of
  // iterators.
  template
  combining_iterator(
    _IteratorCollectionIterator it1,
    _IteratorCollectionIterator it2,
    _Func(func)
    ) :
    m_vect_its(it1, it2), m_func(func) {}

  // The operators * and [] apply the functional to the
  // collection of values of the embedded iterators.
  value_type operator*() const;
  value_type operator[](difference_type n) const;

  // All other operators behave the standard way.
  combining_iterator& operator++();
  const combining_iterator operator++(int);
  combining_iterator& operator--();
  const combining_iterator operator--(int);
  combining_iterator& operator+=(difference_type n);
  combining_iterator operator+(difference_type n) const;
  combining_iterator& operator-=(difference_type n);
  combining_iterator operator-(difference_type n) const;
  difference_type operator-(
    const combining_iterator& rhs
    ) const;

  bool operator==(
    const combining_iterator<_It, _Func>& rhs
    ) const
  { return m_vect_its == rhs.m_vect_its; }
  bool operator!=(
    const combining_iterator<_It, _Func>& rhs
    ) const
  { return m_vect_its != rhs.m_vect_its; }

private:
  std::vector<_It> m_vect_its;
  _Func m_func;
```

```
} ;

template
inline combining_iterator<_It, _Func>::value_type
combining_iterator<_It, _Func>::operator*() const
{ return m_func(m_vect_its.begin(), m_vect_its.end()); }

template
inline combining_iterator<_It, _Func>::value_type
combining_iterator<_It, _Func>::operator[](
  difference_type n
  ) const
{ return *(*this + n); }

template
inline combining_iterator<_It, _Func>&
combining_iterator<_It, _Func>::operator++()
{
  std::vector<_It>::iterator it = m_vect_its.begin();
  while( it != m_vect_its.end() ) { ++*it; ++it; }
  return *this;
}

template
inline const combining_iterator<_It, _Func>
combining_iterator<_It, _Func>::operator++(int)
{
  combining_iterator<_It, _Func> tmp = *this;
  ++*this;
  return tmp;
}

template
inline combining_iterator<_It, _Func>&
combining_iterator<_It, _Func>::operator--()
{
  std::vector<_It>::iterator it = m_vect_its.begin();
  while( it != m_vect_its.end() ) { --*it; ++it; }
  return *this;
}

template
inline const combining_iterator<_It, _Func>
combining_iterator<_It, _Func>::operator--(int)
{
  combining_iterator<_It, _Func> tmp = *this;
  --*this;
  return tmp;
}

template
inline combining_iterator<_It, _Func>&
combining_iterator<_It,
_Func>::operator+=(difference_type n)
{
  std::vector<_It>::iterator it = m_vect_its.begin();
  while( it != m_vect_its.end() ) { *it += n; ++it; }
  return *this;
}

template
inline combining_iterator<_It, _Func>
combining_iterator<_It, _Func>::operator+(
  difference_type n
```

```
  ) const
{
  combining_iterator<_It, _Func> Tmp(*this);
  Tmp += n;
  return Tmp;
}

template
inline combining_iterator<_It, _Func>&
combining_iterator<_It, _Func>::operator-
=(difference_type n)
{
  std::vector<_It>::iterator it = m_vect_its.begin();
  while( it != m_vect_its.end() ) { *it -= n; ++it; }
  return *this;
}

template
inline combining_iterator<_It, _Func>
combining_iterator<_It, _Func>::operator-(
  difference_type n
  ) const
{
  combining_iterator<_It, _Func> Tmp(*this);
  Tmp -= n;
  return Tmp;
}

template
inline combining_iterator<_It, _Func>::difference_type
combining_iterator<_It, _Func>::operator-(
  const combining_iterator<_It, _Func>& rhs
  ) const
{
  if(0 < m_vect_its.size() && 0 < rhs.m_vect_its.size())
    return m_vect_its[0] - rhs.m_vect_its[0];
  else
    return 0;
}

template
struct combining_iterator_traits
{
  typedef typename
    std::vector<_It>::const_iterator functional_arg_type;
};
```

There are a few points about this code that merit discussion. First of all, notice that we have one single class that implements all operators for moving the iterator and for taking differences. If the template argument _It is anything less than a random access iterator, then some of these operator definitions do not compile, because the type _It does not provide them. There is not really a problem with that, because member functions that are never called should not be compiled. Of course, we have just now entered the shady realm of compiler dependency hell, and there is no guarantee as to what your compiler will do. Also, there is one serious restriction here: you cannot do an explicit template instantiation with our combining iterator class template unless the underlying iterator type is a random access iterator. That is because an explicit template instantiation will necessarily try to compile all member functions of the class, some of which are not compilable if the underlying iterator type is not a random access iterator.

Note that the type of the functional that the combining iterator uses must provide its return type through the typedef result_type. For a functional with less than three arguments, this should be done

by deriving from std::unary_function or std::binary_function. Otherwise, you must explicitly put a line such as

```
typedef int result_type;
```

into the declaration of your functional.

Another thing to note about the functional is that its argument type is somewhat unintuitive. Conceptually, the functional operates on the current values of the member iterators of the combining iterator. Technically, when the functional is called inside the combining iterator's operator*(), it gets passed two iterators. These two iterators delimit a range that contains the member iterators at their current position. Think of this as an STL-ish way of passing arguments to a functional: rather than passing an argument list, you give a range consisting of iterators, and the actual argument list is obtained by iterating over this range of iterators and dereferencing each.

It is time for an example. Let us use the combining iterator class to write the difference operator that we discussed before. Suppose we have two vectors of doubles of equal length:

```
std::vector vect_of_doubles_1(42);
std::vector vect_of_doubles_2(42);
// Do something that sets values in vect_of_doubles_1
// and vect_of_doubles_2
```

We wish to produce an iterator that can be used to parallel-iterate through these two vectors and produce their difference. First, we need the functional. Think generic: when writing the functional, we should not specify the exact nature of our two ranges of doubles. Therefore, we let the type of iterator that is suitable for use with these ranges be a template argument to our functional. Also, instead of just doubles, we should allow any type for which subtraction is defined:

```
template
class func_diff : public
  std::binary_function<
    combining_iterator_traits<_It>::functional_arg_type,
    combining_iterator_traits<_It>::functional_arg_type,
    _T>
{
public:
  _T operator()(
    combining_iterator_traits<_It>::functional_arg_type
it1,
    combining_iterator_traits<_It>::functional_arg_type
it2
    ) const
  { return **it1 - **(it1 + 1); }
};
```

As you can see, the type of the arguments passed to the functional is provided by the helper type combining_iterator_traits for convenience. In order to write the functional, you have to understand what that type is: it is an iterator into the collection of member iterators of the combining iterator. Dereferencing such an iterator once takes you to a member iterator, dereferencing it twice takes you to the current value of a member iterator. Our combining iterator class guarantees that functional_arg_type is always a random access iterator. Therefore, the expression **(it1 + 1) in our functional is fine.

Besides writing the functional, the only other thing about using combining iterators that is somewhat less than intuitive is constructing a combining iterator. What you must do is throw the

prospective member iterators of your combining iterator into a collection, then pass to the constructor iterators to the begin and end position of that collection. Here is the code to print out the sequence of differences between our two vectors of doubles:

```
// Typedefs for functional and difference iterator
typedef func_diff<
  std::vector::const_iterator,
  double
  >
  func_double_vect_diff;
typedef combining_iterator<
  std::vector::const_iterator,
  func_double_vect_diff
  >
  double_vect_diff_iterator;

// Construct a difference iterator at the begin position.
std::vector::const_iterator>
vect_tmp(2);
vect_tmp[0] = vect_of_doubles_1.begin();
vect_tmp[1] = vect_of_doubles_2.begin();
double_vect_diff_iterator diff_it_run(
  vect_tmp.begin(),
  vect_tmp.end(),
  func_double_vect_diff()
  );

// Construct a difference iterator at the end position.
vect_tmp[0] = vect_of_doubles_1.end();
vect_tmp[1] = vect_of_doubles_2.end();
double_vect_diff_iterator diff_it_end(
  vect_tmp.begin(),
  vect_tmp.end(),
  func_double_vect_diff()
  );

// Iterate through sequence of differences.
while( diff_it_run != diff_it_end )
{
  cout << *diff_it_run;
  ++diff_it_run;
}
```

Following the STL's philosophy of keeping range and consistency checks to an absolute minimum, we let the integrity and consistency of the combining iterator be the client's responsibility. If, for example, the size of the two vectors in the above example is different, then the running iterator will never equal the end iterator, and the iteration will go off into la-la land.

When working with iterators and algorithms, one can often get away without ever declaring variables to hold iterators. If, for example, we wanted to know if the number 42 occurs our first vector of doubles, we could simply write

```
if( vect_of_doubles1.end() ==  std::find(
  vect_of_doubles1.begin(),
  vect_of_doubles1.end(),
  42.0
  ) );
```

To be able to work with combining iterators in this way, we provide a generating function for combining iterators:

```
template
inline combining_iterator<
  std::iterator_traits<_It>::value_type,
  _Func
  >
make_combining_iterator(_It it1, _It it2, _Func func)
{
  return combining_iterator<
    std::iterator_traits<_It>::value_type,
    _Func
    >
  (
    it1,
    it2,
    func
    );
}
```

Assuming that we have an algorithm standard_deviation to calculate the standard deviation of a sequence, we can now calculate the standard deviation of the difference between our two collection of doubles as follows:

```
std::vector::const_iterator>
  vect_begin_its(2);
vect_begin_its[0] = vect_of_doubles_1.begin();
vect_begin_its[1] = vect_of_doubles_2.begin();

std::vector::const_iterator>
  vect_end_its(2);
vect_end_its[0] = vect_of_doubles_1.end();
vect_end_its[1] = vect_of_doubles_2.end();

double std_dev = standard_deviation(
  make_combining_iterator(
    vect_begin_its.begin(),
    vect_begin_its.end(),
    func_double_vect_diff()
    ),
  make_combining_iterator(
    vect_end_its.begin(),
    vect_end_its.end(),
    func_double_vect_diff()
    )
  );
```

This way, we never have to deal with the actual type of the combining iterator. All we need to know explicitly is the type of the functional.

Processing Iterators

It makes perfect sense to use the combining iterator that we discussed in the previous section with just one member iterator. In that case, the combining iterator pulls its data from just one range, but it still applies its functional upon dereferencing. In this special case, the combining iterator is a somewhat wasteful solution: putting the one member iterator in an STL vector wastes a few bytes of space, and getting to it inside the functional takes a gratuitous level of indirection. Moreover, calling the constructor and writing the functional is more complicated than necessary. Therefore, we have provided a special implementation for this special case which we call a processing iterator. A processing iterator wraps a single iterator. The functionality that this wrapper adds to the plain

iterator is calling a functional upon dereferencing. A simpler version of the processing iterator has been described by the second author in [2].

We said before that the general combining iterator is a const iterator by nature. We have found that in the special case of the processing iterator, it does make sense to use it as a non-const iterator. The dereferencing iterators and the iterator-to-member of the next two sections will provide examples. Technically, this means that we allow the functional used by the processing iterator to return a reference. The only disadvantage is that it is now no longer possible to give meaningful defaults for the processing iterator's value_type, pointer, and reference. It must be left to the client to write a specialization or partial specialization of iterator_traits to get these right. If the functional returns a value, say an integer, then the types must be as follows:

```
typedef int value_type;
typedef const int* pointer;
typedef const int& reference;
```

If, on the other hand, the functional returns a reference, say a reference to an integer, then these types must be:

```
typedef int value_type;
typedef int* pointer;
typedef int& reference;
```

Here is the declaration of the processing iterator, with some obvious stuff omitted:

```
template<
  typename _It, // type of wrapped iterator
  typename _Func // type of functional
  >
class processing_iterator
{

public:
  typedef typename
    std::iterator_traits<_It>::iterator_category
    iterator_category;
  typedef typename
    std::iterator_traits<_It>::difference_type
    difference_type;
  typedef _Func::result_type func_result_type;

  processing_iterator() {}
  processing_iterator(_It from, _Func(func)) :
    m_it(from), m_func(func){}

  // The operators * and [] apply the functional to the
  //value of the embedded iterator.
  func_result_type operator*() const;
  func_result_type operator[](difference_type n) const;

private:
  _It m_it;
  _Func m_func;

} ;

template
inline processing_iterator<_It, _Func>::func_result_type
processing_iterator<_It, _Func>::operator*() const
{ return m_func(*m_it); }
```

```
template
inline processing_iterator<_It, _Func>::func_result_type
processing_iterator<_It, _Func>::operator[](
  difference_type n
  ) const
{ return m_func(m_it[n]); }
```

Note how writing the functional for a particular purpose is much more intuitive here than it was for the combining iterator: here, the functional takes the result of dereferencing the wrapped iterator as its argument. This allows you to use predefined STL functionals with processing iterators. Here's an example that shows how to print out a range of integers with 42 subtracted everywhere:

```
std::vector vectInts;
// Do stuff to put ints in the vector

// Define processing iterators at the begin and end
position of the vector
//
typedef std::binder2nd > subtract_int;
processing_iterator<
  std::vector::const_iterator,
  subtract_int
  >
  prit_run(
    vectInts.begin(),
    std::bind2nd(std::minus(),42)
    );

processing_iterator<
  std::vector::const_iterator,
  subtract_int
  >
  prit_end(
    vectInts.end(),
    std::bind2nd(std::minus(),42)
    );

while( prit_run != prit_end )
{
  std::cout << *prit_run << _T("\n") <<  std::flush;
  ++prit_run;
}
```

As with the combining iterator, there is a generating function that can save you from having to declare variables of type processing_iterator:

```
template
inline processing_iterator<_It, _Func>
make_processing_iterator(_It it, _Func func)
{ return processing_iterator<_It, _Func>(it, func); }
```

If , for example, you want to pass the two iterators of the example above to some function, you do not have to think about their types at all:

```
foo(
  make_processing_iterator(
    vectInts.begin(),
    std::bind2nd(std::minus(),42)
    ),
  make_processing_iterator(
    vectInts.end(),
    std::bind2nd(std::minus(),42)
```

```
    )
  );
```

Dereferencing Iterators

It is often convenient or necessary to work with STL containers that hold pointers to objects, or iterators into other containers that hold the actual objects. This situation typically arises when containers are used as in-memory indices for fast lookup of large objects. These "index containers" are not well suited to be used with algorithms, because typically, the algorithms should not be applied to the pointers that the containers hold, but to the objects pointed to. In some cases, such as the algorithm find, there is a corresponding if-version such as find_if, and the act of dereferencing the pointers can be placed into the functional that find_if takes. But this is not always the case. Moreover, we feel that this is not really the generically correct solution. Processing logic that has nothing to do with the specific algorithm should be placed into the iterator. Therefore, it is useful to have iterators that perform an extra dereferencing inside their operator()*. This can be achieved with a processing iterator as described in the previous section together with a suitable functional. For containers that hold plain pointers, the appropriate generic functional is:

```
template<
  typename _T // type pointed to
  >
class func_pointer_deref :
  public std::unary_function<_T*, _T&>
{
public:
  _T& operator()(_T* arg) const
  { return *arg; }
};
```

If, for example, vect_of_pointers is an STL vector holding pointers to objects of type fat_class, then the following iterator is a const iterator at the begin position of vect_of_pointers that makes it look as if vect_of_pointers were holding the actual objects rather than their addresses:

```
processing_iterator<
  std::vector::const_iterator,
  func_pointer_deref
  >
  const_deref_it(
    vect_of_pointers.begin(),
    func_pointer_deref()
    );
```

For containers that hold iterators rather than plain pointers, the appropriate generic functional for the dereferencing iterator is a little more complex:

```
template<
  typename _Ret, // result type of operator*
  typename _It // type of iterator
  >
class func_iterator_deref :
  public std::unary_function{
public:
  _Ret operator() (const _It& arg) const
  { return *arg; }
};
```

Here, we have to explicitly specify the result type of the iterator's operator*(), because this result type cannot be inferred with certainty from the type of the iterator.

Iterators to Members

Suppose you have a container that contains STL pairs. Now you wish to apply an algorithm that operates on the first component of each pair rather than the entire pair. As before in the case of containers holding pointers, you may be able to use an if- version of your algorithm such as find_if. You may then perform the passage from the pair to its first component inside the functional that the if-algorithm takes. Again, however, there may not be an if-version of your algorithm, and besides, we feel that this logic belongs into the iterator anyway. This can be achieved using a processing iterator with the appropriate instance of the following generic functional:

```
template<
  typename _C, // type of entire object
  typename _R  // type of member
  >
class func_to_member :
  public std::unary_function<_C&, _R&>
{
public:
  func_to_member(_R _C::* memptr) : m_memptr(memptr) {}

  _R& operator() (_C& arg) const
  { return arg.*m_memptr; }

private:
  _R _C::* m_memptr;

};
```

When a processing iterator is constructed with an instance of this functional, a collection of objects of type _C will appear as if it consisted only of the members whose address was passed to the constructor of the functional. If, for example, vect_of_int_pairs is an STL vector holding pairs of integers, then the following iterator is an iterator at the begin position of vect_of_int_pairs that makes things look as if vect_of_int_pairs were holding just the first components of the pairs:

```
typedef func_to_member, int>
  func_pair_ints;
processing_iterator<
    std::vector >::iterator,
    func_pair_ints
    >
  it_to_mem_run(
    vect_of_pairs.begin(),
    func_pair_ints(&std::pair::first)
    );
```

As usual in a situation like this, a suitable generating function for the functional may come in handy when iterators are used directly as function arguments:

```
template
func_to_member<_C, _R> make_func_to_member(
  _R _C::* memptr
  )
{ return func_to_member<_C, _R>(memptr); }
```

Generating Iterators

The iterators that we have discussed thus far all focus on modifying and processing data that was obtained one or more given iterators. The generating iterator that we present in this section, by contrast, creates its data by just calling a functional that gets passed to the iterator upon construction. Austern's line iterator ([1], p. 4) that we mentioned earlier comes to mind as a concrete example. Koenig's book [3] also contains examples of generating iterators.

There are several possible levels of generality for the generating iterator. What we present here is an input iterator, which allows only one pass of forward iteration. Upon construction, the iterator gets passed a functional, a seed value, and an integer that serves as a position index. When the iterator gets dereferenced at its initial position, the seed value is returned. Each time the iterator is advanced, the position index is incremented, the functional gets called, and the return value is stored. The next call to operator*() returns the stored value. The functional takes as its arguments the previously stored value and the position index. Two generating iterators are considered equal under operator==() if and only if they have the same position index. That makes it possible to define ranges with generating iterators despite the fact that there need not be a physical range of data. Here is the complete code for the generating iterator:

```
template
class generating_iterator
{
public:
  typedef std::input_iterator_tag iterator_category;
  typedef int difference_type;
  typedef typename _IncFunc::result_type value_type;
  typedef const value_type& reference;
  typedef const value_type* pointer;

  // Default constructor
  generating_iterator() : m_index(0){}

  // Constructor for begin iterator
  generating_iterator(
    value_type seed,
    _IncFunc inc_func,
    unsigned int index = 0
    ) :
  m_inc_func(inc_func),
  m_current_value(seed),
  m_index(index)
  {}

  // Constructor for end iterator
  generating_iterator(unsigned int index) :
    m_index(index) {}

  reference operator*() const;

  const generating_iterator& operator++();
  generating_iterator operator++(int);

  bool operator==(
    const generating_iterator<_IncFunc>& rhs
    ) const
  { return m_index == rhs.m_index; }
  bool operator!=(
    const generating_iterator<_IncFunc>& rhs
```

```
    ) const
  { return m_index != rhs.m_index; }

private:
  value_type m_current_value;
  unsigned int m_index;
  _IncFunc m_inc_func;

};

template
inline generating_iterator<_IncFunc>::reference
generating_iterator<_IncFunc>::operator*() const
{ return m_current_value; }

template
inline const generating_iterator<_IncFunc>&
generating_iterator<_IncFunc>::operator++()
{
  m_current_value = m_inc_func(m_current_value, m_index);
  ++m_index;
  return *this;
}

template
inline generating_iterator<_IncFunc>
generating_iterator<_IncFunc>::operator++(int)
{
  generating_iterator<_IncFunc> tmp = *this;
  ++*this;
  return tmp;
}
```

One possible use of generating iterators is to generate mathematically defined sequences such as the Fibonacci numbers and pass them to functions as STL-style ranges. Note that when the functional gets called inside the generating iterator's operator++(), only the previously stored value is passed as an argument. If, as is the case with the Fibonacci numbers, older values are needed as well, then this can of course be achieved by adding state to the functional. Passing the previously stored value as an argument is merely a matter of convenience.

As another application, we now show how generating iterators can be used to make given ranges appear as if they had been rearranged, possibly dropping or repeating some elements in the process. Suppose, for example, that you are given a range delimited by two random access iterators it1 and it2 of some type ItType, and suppose you wish to iterate through this range beginning at it1 and then advancing in steps of three. To this end, we define a functional that advances iterators in leaps while safeguarding against overrun:

```
template
class advance_n : public std::binary_function<
  const _It&,
  unsigned int,
  _It
  >
{
public:

  advance_n() : m_inc(0), m_max_pos(0) {}
  advance_n(unsigned int inc, unsigned max_pos) :
    m_inc(inc), m_max_pos(max_pos)
  {}
```

```
  _It operator()(
    const _It& it,
    unsigned int pos
    ) const
  {
    if(pos == m_max_pos)
      return it;
    unsigned int inc = m_inc * (pos + 1) < m_max_pos ?
      m_inc : m_max_pos - m_inc * pos;
    return it + inc;
  }

private:
  unsigned int m_inc;
  unsigned int m_max_pos;

};
```

We then define a generating iterator for running through the range delimited by it1 and it2 in steps of three, with safeguard against overrun. Upon dereferencing, this generating iterator will return, as it progresses, the values it1, it1+3, it1+6, and so on.

```
generating_iterator > it_run(
  it1,
  advance_n(3, it2 - it1)
  );
```

The appropriate end iterator is simply:

```
generating_iterator >
  it_end(it2 - it1);
```

Suppose the result type of the dereferencing operator of our iterator type _It is some type named ItDerefType. Using the above generating iterators with the dereferencing iterator as discussed earlier, we obtain a pair of iterators that can serve as drop-ins for the original iterators, except that they make the sequence look as if two out of each three elements had been dropped:

```
processing_iterator<
 generating_iterator >,
 func_iterator_deref
 >
 three_step_it1(
   it_run,
   func_iterator_deref()
   );

processing_iterator<
 generating_iterator >,
 func_iterator_deref
 >
 three_step_it2(
   it_end,
   func_iterator_deref()
   );
```

Alexander Stepanov [4] has indicated that the STL may eventually include iterator adaptors called *strides* that would achieve exactly what we're doing in this example.

Conclusion

We have demonstrated how custom STL-style iterators can provide a uniform, generic way to present sequentially structured data. Using iterators in this way enhances the power of the STL insofar as algorithms can now be applied to a much wider range of data. The STL no longer appears as just a collection of algorithms and containers. Instead, algorithms, iterators, and containers truly become the three pillars of the STL.

References

[1] Matthew H. Austern, Generic Programming and the STL, Addison Wesley 1998

[2] Thomas Becker, Smart iterators and the STL, C/C++ Users Journal 16/9, 39-46 (1998).

[3] Andrew Koenig and Barbara E. Moo (Editor), Ruminations on C++: A Decade of Programming Insight and Experience, Addison Wesley 1996

[4] An interview with A. Stepanov, STLport.org

[5] Alexander Stepanov and Meng Lee, The Standard Template Library, Hewlett Packard Laboratories, Nov 1995