

Segmented Iterators and Hierarchical Algorithms

Matthew H. Austern

Silicon Graphics Computer Systems
austern@sgi.com

Abstract. Many data structures are naturally segmented. Generic algorithms that ignore that feature, and that treat every data structure as a uniform range of elements, are unnecessarily inefficient. A new kind of iterator abstraction, in which segmentation is explicit, makes it possible to write hierarchical algorithms that exploit segmentation.

Keywords: Standard Template Library, iterators, multidimensional data structures

1 Introduction

A defining characteristic of generic programming is “Lifting of a concrete algorithm to as general a level as possible without losing efficiency; *i.e.*, the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.” The best known example of a generic library is the C++ Standard Template Library, or STL [1–3], a collection of algorithms and data structures dealing with one-dimensional ranges of elements.

For an important class of data structures, those which exhibit segmentation, the STL does not make it possible to write algorithms that satisfy the goal of abstraction without loss of efficiency. A generic algorithm written within the framework of the STL can’t exploit segmentation. The difficulty is a limitation of the STL’s central abstraction: iterators.

2 Iterators and Algorithms

The STL is mainly concerned with algorithms on one-dimensional ranges. As a simple example of such an algorithm, consider the operation of assigning a value to the elements of an array. In C, we can write this operation as follows:

```
void fill1(char* first, char* last, char value)
{
    for ( ; first != last; ++first)
        *first = value;
}
```

The argument `first` is a pointer to the beginning of the array: the element that `first` points to, `*first`, is the first element of the array. The argument `last` is a pointer just past the end of the array. That is, `fill1` performs the assignments `*first = value`, `*(first + 1) = value`, and so on up to but not including `last`.

The arguments `first` and `last` form a *range*, `[first, last)`, that contains the elements from `first` up to but not including `last`. The range `[first, last)` contains `last - first` elements.

The function `fill1` is similar to `memset`, from the standard C library; like `memset`, it can only be used for assigning a value to an array of type `char`. In C++ [4, 5] it is possible to write a more general function, `fill`:

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last,
          T value)
{
    for ( ; first != last; ++first)
        *first = value;
}
```

This is an example of a generic algorithm written in the STL framework. Like `fill1`, `fill` steps through the range `[first, last)` from beginning to end. For each iteration it tests whether there are any remaining elements in the range, and, if there are, it performs the assignment `*first = value` and then increments `first`. The difference between `fill1` and `fill` is how the arguments `first` and `last` are declared: in `fill1` they are pointers of type `char*`, but in `fill`, a function template, they may be of any type `ForwardIterator` for which the operations in `fill` are well-defined. The type `ForwardIterator` need not be a pointer type; it need only conform to a pointer-like interface. It must support the following operations:

- *Copying and assignment*: Objects of type `ForwardIterator` can be copied, and one object of type `ForwardIterator` can be assigned to another.
- *Dereference*: An object of type `ForwardIterator` can be dereferenced. If `i` is an object of type `ForwardIterator`, then the expression `*i` returns a reference to some C++ object. The type of that object is known as `ForwardIterator`'s *value type*.
- *Comparison for equality*: Objects of type `ForwardIterator` can be compared for equality. If `i` and `j` are values of type `ForwardIterator`, then the expressions `i == j` and `i != j` are well-defined. Two dereferenceable iterators `i` and `j` are equal if and only if `*i` and `*j` refer to the same object.
- *Increment*: Objects of type `ForwardIterator` can be incremented. If `i` is an object of type `ForwardIterator`, then `++i` modifies `i` so that it points to the next element.

These requirements are known as the `Forward Iterator` requirements, or the `Forward Iterator` concept, and a type that conforms to them is known as a `Forward`

Iterator. Pointers are Forward Iterators, for example, and it is also possible to define Forward Iterators that step through singly or doubly lists, segmented arrays, and many other data structures. Since operators in C++ can be overloaded, a user-defined iterator type can ensure that expressions like `*i` and `++i` have the appropriate behavior.

The Forward Iterator requirements are part of the STL, and so are algorithms, like `fill`, that operate on Forward Iterators. The STL also defines several other iterator concepts. A Bidirectional Iterator type, for example, is a type that conforms to all of the Forward Iterator and that provides additional functionality: a Bidirectional Iterator `i` can be decremented, using the expression `--i`, as well as incremented. Similarly, a Random Access Iterator type conforms to all of the Bidirectional Iterator requirements and also provides more general arithmetic operations. Random Access Iterators provide such operations as `i += n`, `i[n]`, and `i - j`.

Pointers are Random Access Iterators, which means that they are Bidirectional Iterators and Forward Iterators as well.

Dispatching algorithms and iterator traits The reason for the different iterator concepts is that they support different kinds of algorithms. Forward Iterators are sufficient for an algorithm like `fill`, which steps through a range from one element to the next, but they are not sufficient for an algorithm like Shell sort, which requires arbitrary-sized steps. If Shell sort were implemented as a generic STL algorithm, it would have to use Random Access Iterators. An algorithm written to use Forward Iterators can be called with arguments that are Random Access Iterators (every Random Access Iterator is also a Forward Iterator), but not the other way around.

It's obvious what kind of iterator an algorithm like Shell sort ought to use, but sometimes the choice is less obvious. Consider, for example, the problem of reversing the elements in a range. It is possible to write a reverse algorithm that uses Forward Iterators, or one that uses Bidirectional Iterators. The version that uses Forward Iterators is more general, but the version that uses Bidirectional Iterators is faster when it can be called at all.

Both generality and efficiency are important, so the STL introduces the notion of *dispatching algorithms*: algorithms that select one of several methods depending on the kind of iterator that they are invoked with. The dispatch mechanism uses a traits class and a set of iterator tag classes, and it incurs no runtime performance penalty.

The iterator tags are a set of five empty classes, each of which is a placeholder corresponding to an iterator concept.¹ The traits class is a template, `iterator_traits`, whose nested types supply information about an iterator.

For any iterator type `Iter`, `iterator_traits<Iter>::value_type` is `Iter`'s value type and `iterator_traits<Iter>::iterator_category` is the iterator

¹ The five iterator concepts are Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, and Random Access Iterator. This paper does not discuss Input Iterator and Output Iterator

tag type corresponding to the most specific concept that `Iter` conforms to. For example, `iterator_traits<char*>::value_type` is `char` and `iterator_traits<char*>::iterator_category` is the category tag for Random Access Iterators, `random_access_iterator_tag`. The traits mechanism is quite general, and has many different uses.

Given this machinery, writing a dispatching algorithm is simple. The algorithm, using `iterator_traits`, calls an overloaded helper function with the iterator's category tag as an argument. The compiler selects the appropriate helper function via ordinary overload resolution. For example, here is the skeleton of `reverse`:

```
template <class ForwardIter>
void reverse(ForwardIter first, ForwardIter last,
             forward_iterator_tag) {
    ...
}

template <class BidirectionalIter>
void reverse(BidirectionalIter first, BidirectionalIter last,
             bidirectional_iterator_tag) {
    ...
}

template <class RandomAccessIter>
void reverse(RandomAccessIter first, RandomAccessIter last,
             random_access_iterator_tag) {
    ...
}

template <class Iter>
inline void reverse(Iter first, Iter last) {
    typedef typename iterator_traits<Iter>::iterator_category
        category;
    reverse(first, last, category());
}
```

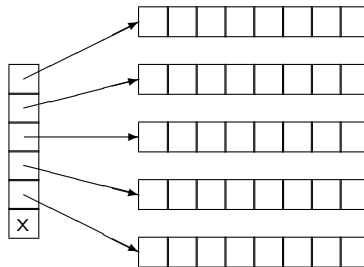
3 Segmented Data Structures

The STL defines five different iterator concepts, but all of them have one thing in common: each of them represents a position in a uniform one-dimensional range. Given a Forward Iterator `i`, for example, the next iterator in the range (if there is a next) is always `++i`. As far as an algorithm that uses Forward Iterators is concerned, every increment operation is the same. Similarly, given a range `[first, last)` of Random Access Iterators, every iterator in the range is equal to `first + n` for some integer `n`. Again, the range is uniform and one-dimensional; every position within it is characterized by a single measure of distance.

Many data structures can be viewed as linear ranges (any finite collection can be arranged in *some* order), but there are data structures for which a different view is also appropriate. Often, the most natural index for an element isn't a single offset but a more complicated multidimensional description. One example is a segmented array, or a vector of vectors.

```
template <class T> struct seg_array
{
    typedef T value_type;
    typedef vector<T> node;
    vector<node*> nodes
    ...
};
```

Here is a simple pictorial representation:



Elements in the segmented array are contained in nodes, each of which is a vector; each element may be characterized by a node and an index within the node. The outer vector consists of pointers to nodes, and, to make it easier to detect the end of the array, the last entry in the outer vector is a null pointer. (Represented in the picture by an “x.”)

This isn't an artificial example: segmented data structures are common. Within the STL, for example, the `deque` container is typically implemented as a segmented array, and hash tables [6] are typically implemented as arrays of buckets. Other examples of segmentation include two-dimensional matrices, graphs, files in record-based file systems, and, on modern NUMA (non-uniform memory architecture) multiprocessor machines, even ordinary memory.

The most natural way of assigning a value to all of the elements in a `seg_array` is a nested loop: loop through all of the nodes, and, for each node, assign a value to all of the node's elements.

```
vector<vector<T>*>::iterator i;
for (i = nodes.begin(); i != nodes.end(); ++i) {
    vector<T>::iterator j;
    for (j = (**i).begin(); j != (**i).end(); ++j)
        *j = value;
}
```

Instead of writing a new function to assign a value to every element in a `seg_array`, it is preferable to reuse the existing generic algorithm `fill`. It is merely necessary to define a Forward Iterator that steps through a `seg_array`, which, in turn, means defining a sense in which the elements of a `seg_array` can be viewed as a simple linear range. There is a straightforward linearization: an element's successor is the next element in the same node, or, if no such element exists, the first element in the next node. This definition can be used to write an iterator for `seg_array`, `seg_array_iter`:

```
template <class T> struct seg_array_iter
{
    vector<T>::iterator      cur;
    vector<vector<T>*>::iterator node;

    T& operator*() const { return *cur; }

    seg_array_iter& operator++() {
        if (++cur == (**node).end()) {
            ++node;
            cur = *node ? (**node).begin() : 0;
        }
    }
    ...
};
```

Using a `seg_array_iter` with `fill` is simple, but it changes the algorithm. Every time a `seg_array_iter` is incremented, it checks to see if it has arrived at the end of a node. Instead of a nested loop, the combination of `seg_array_iter` and `fill` is more like this:

```
vector<vector<T>*>::iterator node = nodes.begin();
vector<T>::iterator cur = (**node).begin();
while (node != nodes.end()) {
    *cur = value;
    ++cur;
    if (cur == (**node).end()) {
        ++node;
        if (node != nodes.end())
            cur = (**node).begin();
    }
}
```

As expected, the loop overhead is higher. On a 150 MHz MIPSTM R5000, for example, it is 20% slower than the straightforward nested loop.

In this instance, the STL fails to achieve the goal of generic programming: the STL's generic algorithm `fill` is not as efficient as the original algorithm written for `seg_array`. The original algorithm uses a simple and easily optimized nested

loop, while the generic algorithm expands into a complicated loop with multiple tests in every iteration.

The difficulty, fundamentally, is that, while the `seg_array_iter` class manages segmentation (it keeps track of a node and a position within the node), it does not expose the segmentation as part of its interface; segmentation is not part of the **Forward Iterator** requirements. The solution is a new iterator concept, one that explicitly supports segmentation.

4 Segmented Iterators²

The discussion of `fill` in Section 3 is not completely general, because it only addresses the special case of assigning a new value to every element of a segmented array. It does not consider the traversal of incomplete segments.

More generally `fill` operates on a range `[first, last)`, where `first` is not necessarily at the beginning of a segment and `last` is not necessarily at the end of one. The general case is illustrated in the following diagram:



An algorithm that operates on a range `[first, last)` in a segmented data structure can be implemented using nested loops. The outer loop is from `first`'s segment up to and including `last`'s segment. There are three kinds of inner loops: a loop through the first segment (from `first` to the end of the segment), a loop through the last segment (from the beginning of the segment up to but not including `last`), and loops through every segment other than the first and the last (from the beginning of the segment to the end). One final special case is when `first` and `last` point into the same segment.

In pseudocode, a segmented version of `fill` might be written as follows:

```
hierarchical_fill(first, last, value)
{
    sfirst = Segment(first);
    slast  = Segment(last);

    if (Segment(first) == Segment(last))
        fill(Local(first), Local(last), value);
    else {
        fill(Local(first), End(sfirst), value);
        ++sfirst;
    }
}
```

² Other names that have been used for the same concept include “NUMA iterator,” “bucket iterator,” and “cluster iterator.”

```

    while (sfirst != slast) {
        fill(Begin(sfirst), End(sfirst), value);
        ++sfirst;
    }
    fill(Begin(slast), Local(last), value);
}
}

```

A segmented iterator can be viewed as an ordinary **Forward Iterator**; ***first** is an element in a data structure, and **first** can be incremented some finite number of times to obtain **last**. Additionally, though, a segmented iterator can be decomposed into two pieces: a *segment iterator* that points to a particular segment, and a *local iterator* that points to a location within that segment. Furthermore, it must be possible to inspect a segment iterator and obtain local iterators that point to the beginning and the end of the segment.³

A segmented iterator and its associated local iterator are both iterators, both have the same value type, and both can be used to traverse a segment within a segmented container. The difference is that the full segmented iterator can traverse a range that includes more than one segment, while a local iterator is valid only within a single segment. Incrementing a local iterator can thus be a very fast operation; it need not incur the overhead of checking for the end of a segment.

As an example, the `seg_array_iter` class from Section 3 is a segmented iterator. Its associated segment iterator type points to a `vector<T>` (incrementing a segment iterator moves from one vector to the next), and its associated local iterator type is an iterator that traverses a single vector; that is, it is `vector<T>::iterator`.

The distinction between segmented and nonsegmented iterators is orthogonal to the distinction between **Forward Iterators**, **Bidirectional Iterators**, and **Random Access Iterators**. The `seg_array_iter` class could easily be implemented as a **Random Access Iterator**; each segment has the same number of elements, so the operation `i += n` requires nothing more than integer arithmetic. A hash table where the buckets are implemented as linked lists could also provide segmented iterators, but, in contrast to `seg_array`, they could be at most **Bidirectional Iterators**. A segmented iterator type can be no stronger than the weaker of its associated segment iterator and local iterator types.

A segmented iterator has the same associated types as any other iterator (a value type, for example), and, additionally, it has an associated segment iterator type and local iterator type. Writing a generic algorithm that uses segmented iterators requires the ability to name those types. Additionally, a fully generic algorithm ought to be usable with both segmented and nonsegmented iterators. This is the same problem as with the generic algorithm `reverse`, in Section 2,

³ STL iterators can be *past-the-end*. An iterator like `last` doesn't necessarily point to anything; it might, for example, point beyond the end of an array. Note that it must be possible to obtain a valid segment iterator even from a past-the-end segmented iterator.

and the solution is also the same: a traits class, which makes it possible to write `fill` as a dispatching algorithm.

The `segmented_iterator_traits` class identifies whether or not an iterator is segmented, and, for segmented iterators, it defines the segment and local iterator types. Finally, it is convenient for the traits class to contain the functions for decomposing a segmented iterator into its segment iterator and local iterator.⁴

The general definition of `segmented_iterator_traits` contains nothing but a flag identifying the iterator as nonsegmented:

```
template <class Iterator>
struct segmented_iterator_traits {
    typedef false_type is_segmented_iterator;
};
```

The traits class is then specialized for every segmented iterator type. For `seg_array_iterator`, for example, the specialization is as follows:

```
template <class T>
struct segmented_iterator_traits<seg_array_iter<T> >
{
    typedef true_type                is_segmented_iterator;
    typedef seg_array_iter<T>        iterator;
    typedef vector<vector<T>*>::iterator segment_iterator;
    typedef vector<T>::iterator      local_iterator;

    static segment_iterator segment(iterator);
    static local_iterator   local(iterator);

    static iterator compose(segment_iterator, local_iterator);

    static local_iterator begin(segment_iterator);
    static local_iterator end(segment_iterator);
};
```

Every such specialization for a segmented iterator type has the same interface: the types `segment_iterator` and `local_iterator`, and the functions `segment`, `local`, `compose`, `begin`, and `end`. This interface can be used by generic algorithms such as `fill`. Here is a full implementation:

```
template <class SegIter, class T>
void fill(SegIter first, SegIter last, const T& x, true_type)
{
    typedef segmented_iterator_traits<SegIter> traits;
```

⁴ Putting these functions into the traits class is slightly more convenient than putting them into the iterator class, because it makes it easier to reuse existing components as segmented iterators.

```

typename traits::segment_iterator sfirst
    = traits::segment(first);
typename traits::segment_iterator slast
    = traits::segment(last);

if (sfirst == slast)
    fill(traits::local(first), traits::local(last), x);
else {
    fill(traits::local(first), traits::end(sfirst), x);
    for (++sfirst ; sfirst != slast ; ++sfirst)
        fill(traits::begin(sfirst), traits::end(sfirst), x);
    fill(traits::begin(sfirst), traits::local(last), x);
}
}

template <class ForwardIter, class T>
void fill(ForwardIter first, ForwardIter last, const T& x,
         false_type)
{
    for ( ; first != last; ++first)
        *first = x;
}

template <class Iter, class T>
inline void fill(Iter first, Iter last, const T& x)
{
    typedef segmented_iterator_traits<Iter> Traits;
    fill(first, last, x,
         typename Traits::is_segmented_iterator());
}

```

The main function, `fill`, uses the traits mechanism to select either a segmented or a nonsegmented version. The segmented version, written using nested loops, uses the traits mechanism to decompose a segmented iterator into its segment and local iterators. The inner loop is written as in invocation of `fill` on a single segment.

Multiple levels of segmentation can be used with no extra effort. Nothing in this discussion, or this implementation of `fill`, assumes that the local iterator is nonsegmented.

5 Conclusions

The most important innovation of the STL is the *iterator* abstraction, which represents a position in a one-dimensional range. Iterators are provided by containers and used by generic algorithms, and they make it possible to decouple algorithms from the data structures that they operate on.

Ordinary STL iterators describe a uniform range with no additional substructure. Even if a data structure has additional features, such as segmentation, it is impossible to access those features through the iterator interface.

Segmented iterators, an extension of ordinary STL iterators, make it possible for generic algorithms to treat a segmented data structure as something other than a uniform range of elements. This allows existing algorithms to operate more efficiently on segmented data structures, and provides a natural decomposition for parallel computation. Segmented iterators enable new kinds of generic algorithms that explicitly depend on segmentation.

Acknowledgments

This work is a collaborative effort by many members of the Silicon Graphics compiler group. I also wish to thank Ullrich Köthe and Dietmar Kühl for helpful suggestions.

References

1. A. A. Stepanov and M. Lee, “The Standard Template Library.” Hewlett-Packard technical report HPL-95-11(R.1), 1995.
2. D. R. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
3. M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.
4. B. Stroustrup, *The C++ Programming Language*, Third Edition. Addison-Wesley, 1997.
5. International Organization for Standardization (ISO), 1 rue de Varembe, Case postale 56, CH-1211 Genève 20, Switzerland, *ISO/IEC Final Draft International Standard 14882: Programming Language C++*, 1998.
6. J. Barreiro, R. Fraley, and D. R. Musser, “Hash tables for the Standard Template Library.” Technical Report X3J16/94-0218 and WG21/N0605, International Organization for Standardization, February 1995.