Objective Viewpoint

# An Overview of The Standard Template Library

*by G. Bowden Wise*

One practical limitation of C++ has been the lack of a common set of generic data structures, called **containers**, for use in programs. C++ programmers have had to ``roll their own'' data structures whenever they needed them. Thousands of C++ programmers everywhere have been re-inventing the wheel time and time again. Those of you who have written a *generic* data structure, such as a binary tree, know how tedious and error-prone this undertaking can be.

The reason for this shortcoming is that, until recently, C++ lacked sufficient language mechanisms to enable developers to implement container classes that were type safe, flexible, and efficient for general use. As a result, Stroustrup and others of the Extensions Working Group (WG) of the ANSI C++ Standards Committee, embarked on a project to incorporate **templates** into the C++ language [6]. The first official proposal for templates appeared in the ARM [1] in 1990 and templates are now an official language construct as outlined in the April 1995 ANSI draft [2]. The Extensions WG was also responsible for the incorporation of exceptions into the language [4].

Without templates, there were only two approaches for creating generic components in C++: implement the data structure as a C `struct` with `void` pointers to hold the data; or, implement the data structure using a C++ **class** hierarchy and use **inheritance** and **virtual function**s. However, neither of these approaches could provide the efficiency required for a component in a standard library that would be put to general use.

The Standard Template Library (STL) provides C++ programmers with a library of common data structures -- linked lists, vectors, deques, sets, and maps -- and a set of fundamental algorithms that operate on them. C++ programmers no longer need to implement these basic data structures and can be assured that the data structures provided by the STL perform efficiently and correctly.

## Orthogonal Component Structure

The STL would not have been possible without the use of C++ templates; class and function templates are used throughout the STL. Templates provide not only the efficiency needed for a generic component library, but also make the library extensible. Templates allow the STL to work with built-in types and user-defined types in a seamless way.

Templates are still a recent addition to C++ and many compiler vendors do not provide all of the features as suggested in the April 1995 ANSI draft. The STL depends heavily on many of these advanced features and, in some cases, relies on workarounds to accommodate the current generation of compilers.

There are six components in the STL organization. Three components, in particular, can be considered the core components of the library:

- **Containers** are data structures that manage a set of memory locations.
- **Algorithms** are computational procedures.
- **Iterators** provide a mechanism for traversing and examining the elements in a container.

An STL data structure or container, unlike traditional ones, does not contain many member functions. STL containers contain a minimal set of operations for creating, copying, and destroying the container along with operations for adding and removing elements. You will not find container member functions for examining the elements in a container or sorting them. Instead, algorithms have been decoupled from the container and can only interact with a container via traversal by an iterator. This orthogonal component structure is illustrated in Figure 1.

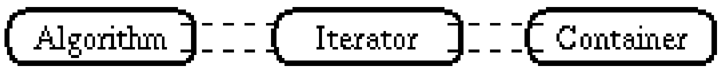

**Figure 1:** Containers, algorithms, and iterators form an orthogonal component space in the STL.

This orthogonal structure is what brings the STL its power, flexibility, and extensibility. Developers that implement new algorithms utilizing one of the STL iterators are guaranteed that their algorithms will work with existing container types as well as those that have not yet been developed.

The remaining three components of the STL are also fundamental to the library and contribute to its flexibility and portability:

- **Function objects** encapsulate a function as an object.
- **Adaptors** provide an existing component with a different interface.
- **Allocators** encapsulate the memory model of the machine.

Let's take a look at each of the STL components in a little more detail.

## Containers

**Containers** are data structures that manage a collection of elements and are responsible for the allocation and deallocation of those elements. Containers typically have constructors and destructors along with operations for inserting and deleting elements.

Elements are stored in containers as whole objects; no pointers are used to access the elements in a container. This results in containers that are type safe and efficient. The STL provides two categories of containers:

**Sequence containers** store elements in sequential order. These containers group a finite set of elements in a linear arrangement. The STL includes class templates for `vector`s, `list`s, and `deque`s.

Here is an example of a `vector` of `int`s:

```
   vector<int> l;
   for (int i=0; i < 100; i++)
      l.push_back (i);
```

**Associative containers** store elements based on a key value. Implemented as red-black trees, they provide efficient retrieval of elements based on their key. The STL provides class templates for `map`s, `set`s, `multimap`s, and `multiset`s. `map`s and `multimap`s allow arbitrary data to be associated with each key. Also, `map`s and `set`s only allow elements with unique keys, whereas `multimap`s and `multiset`s may contain elements with duplicate keys. Associative arrays are easily created using `map`s:

```
   map<string,string,less<string> > cap;
   cap["Arkansas"]="Little Rock";
   cap["New York"]="Albany";
```

## Iterators

Containers, by themselves, do not provide access to their elements. Instead, **iterators** are used to traverse the elements within a container. Iterators are very similar to smart pointers and have increment and dereferencing operations. By generalizing access to containers through iterators, the STL makes it possible to interact with containers in a uniform manner. In addition, as we saw in Figure 1, iterators are the glue that connects algorithms to containers.

Iterators are the cornerstone of the STL design and give the STL its flexibility. Instead of developing algorithms for a specific container, they are developed for a specific **iterator category**. This strategy makes it possible to use the same algorithm with a variety of different containers.

Each category forms a set of requirements that must be met by concrete iterator types within that category. Requirements for a given iterator category are specified by a set of of valid expressions for iterators in that category as well as precise semantics describing their usage. In addition, iterators in the STL must satisfy complexity requirements. These requirements ensure that algorithms written in terms of iterators will work correctly and efficiently.

The STL provides a hierarchy of iterator categories as shown in Figure 2. Iterators at the top of the hierarchy are the most general; those at the bottom are the most restricted and have fewer requirements. An iterator satisfies all of the requirements of the iterator below it.

Each container specifies which iterator category it belongs to. By using a novel language technique, the STL selects the right algorithm at compile time depending on the iterator category.

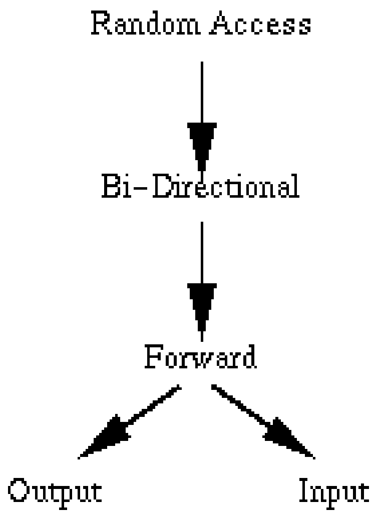

**Figure 2:** Iterators in the STL form a hierarchy.

Iterators are used in a similar manner as pointers. Just like pointers, for any iterator type, there is guaranteed to be an iterator value that points **past** the last element of a corresponding container. Each STL container provides member functions, `begin()` and `end()`, that return iterator values for the first and end (one past the last) elements.

To iterate through the list of `int`s we created above, we write:

```
list::iterator iter;
for (iter  = l.begin();
     iter != l.end();
     iter++)    {
   cout << *l << endl;
}
```

Note that each time through the loop we dereference the iterator to obtain the value the iterator is pointing at. The increment operator is used to advance the iterator through the list.

The STL also provides `const` iterators so that iterators may be used with `const` containers:

```
template <class T>
void PrintList (const list<T>& l)  {
   list::const_iterator iter;
   for (iter  = l.begin();
        iter != l.end();
        iter++)  {
      cout << *l << endl;
   }
}
```

## Stream Iterators

The C++ Standard Library provides `iostream`s to facilitate the reading and writing of data to and from input/output streams. The STL provides two iterator templates so that algorithms may work directly with I/O streams:

- `istream_iterator`s for reading data from an input stream.
- `ostream_iterator`s for writing data to an output stream.

For example, we could read data into our `list` from the standard input as follows:

```
istream_iterator<int, ptrdiff_t> in(cin);
istream_iterator<int, ptrdiff_t> eos;
copy (in, eos, back_inserter(l));
```

and later write the list to the standard output:

```
ostream_iterator out(cout, ", ");
copy (l.begin(), l.end(), out):
```

## Algorithms

STL algorithms are decoupled from the particular containers they operate on and are instead parameterized by iterator types. This means that all containers within the same iterator category (see Figure 2) can utilize the same algorithms.

Because algorithms are written to work on iterators rather than components, the software development effort is drastically reduced. Instead of writing a search routine for each kind of container, one need only write one for each iterator of interest. Since different components can be accessed by the same iterators, just a few versions of the search routine must be implemented.

For example, here is the HP implementation [3 ] of the STL `count` algorithm, which counts the number of elements in a container with a particular value

```
template <class InputIterator,
          class T,
          class Size>
void count(InputIterator first,
           InputIterator last,
           const T& value,
           Size& n) {
    while (first != last)
        if (*first++ == value) ++n;
}
```

An algorithm which uses `InputIterator`s can be used with any container that works with `InputIterator`s or any of the iterator categories above it in the hierarchy (Figure 2).

To count the number of elements equal to `10` in our `list`, we can write:

```
int countTen = 0;
count(l.begin(), l.end(), 10, countTen);
```

The semantics of the `InputIterator first` and `InputIterator last` in the STL `count` algorithm look remarkably like regular C pointers. In fact all STL algorithms will also work with regular C/C++ pointers. For example, we can use the same count algorithm to to count values in a regular `int` array:

```
int a[100];
// put some values into array a
int countTen = 0;
count(&a[0], &a[100], 10, countTen);
```

Most C/C++ programmers are so familiar with the pointer programming paradigm that they quickly become familiar with the STL algorithms.

The pointer-like semantics of STL algorithms guarantee that there is an efficient implementation for them, often resulting in code that is nearly as efficient as hand-written assembly code.

The STL provides many fundamental algorithms in seven categories:

- **Non-Mutating Sequence Operations.** Algorithms like `count` and `search` which do not modify the iterator or its associated container.
- **Mutating Sequence Operations.** Algorithms like `copy`, `reverse`, or `swap` which may modify a container or its iterator.
- **Searching and Sorting.** Sorting, searching, and merging algorithms, such as `stable_sort`, `binary_search`, and `merge`.
- **Set Operations.** Mathematical set operations, such as `set_union` and `set_intersection`. These algorithms only work on sorted containers.
- **Heap Operations.** Heaps are a very useful and efficient data structure that is often used to implement priority queues. The STL provides facilities for making heaps and using them.
- **Numeric Operations.** The STL provides a few numerical routines to show how the STL might be used to provide a template-based numeric library. The STL provides algorithms for: `accumulate`, `inner_product`, `partial_sum`, and `adjacent_difference`.
- **Miscellaneous Operations.** This final category is for algorithms, like `min` and `next_permutation` that don't quite fit in the above categories.

## Function Objects

In C, you can make a routine more generic by passing a pointer to a helper function that can be used by the routine as part of its algorithm. This is how the C library quisksort algorithm, `qsort`, is implemented. To sort an array of integers, we define a comparison function

```
static int intcmp(int* i,int *j){
   return(*i - *j);
}
```

and pass a pointer to this function when calling `qsort`:

```
qsort(a,10,sizeof(int),intcmp);
```

The problem with passing pointers to functions is that they are less efficient than template functions that can be expanded inline.

**Function objects** are class objects which have a function call operator ( `operator()`) defined. They contain no data members or constructors/destructors (except the default ones provided by the compiler). Function objects can be passed to template functions in much the same way as pointers to functions are passed to C functions. However, because there is no pointer indirection and because they can often be expanded inline, they are much more efficient.

Suppose we wish to remove from our `list`, all of the `int`s that are a multiple of `3`. We can use the STL `remove_if` function and supply our own predicate, called `three_mult`, as a function object.

The actual checking of the multiple goes in the body of `operator()`, which returns a `bool`:

```
struct three_mult {
   bool operator() (int& v) {
      return (v % 3 == 0);
   }
};
```

A new list, `m`, is created as the predicate is applied to each element of the list by `remove_copy_if`:

```
remove_copy_if (l.begin(),
                l.end(),
                back_inserter(m),
                three_mult());
```

The STL provides many function objects that are used by STL algorithms, but may also be used by users in their code. The STL provides function objects for arithmetic operators (e.g., `times`, `divides`, and `modulus`), comparison operators (e.g., `greater`, `less`, and `less_equal`), and logical operators (e.g., `logical_and`, `logical_or`, and `logical_not`).

## Adaptors

**Adaptors** are template classes that provide an existing class with a new interface. Adaptors can be used to create new interfaces for containers or iterators.

### Container Adaptors

Often it is desirable to create a specialized container from an existing, more general container. The operations of the new container are implemented using the underlying operations of the existing containers. **Container adaptors** are used to create a new container by mapping the interface of an existing container to that of the new container. This allows new containers to be created without much additional effort, since most of the functionality is already provided by the existing container.

The STL provides three container adaptors: `stack<Container>`, `queue<Container>`, and `deque<Container>`. The following program demonstrates the use of a `stack` that has been implemented as a `vector`:

```
#include <vector.h>
#include <stack.h>
void
main()  {
   stack <vector<int> > st;

   for (int i=0; i < 10; i++)
      st.push (i);

   while ( !st.empty() ) {
      cout << st.top() << " ";
      st.pop();
   }
   cout << endl;
}
```

## Iterator Adaptors

Adaptors can also be used to extend the functionality of an existing iterator. The STL provides three iterator adaptors:

- **Reverse iterators.** Random access iterators and bi-directional iterators can be traversed forwards and backwards. By applying the `reverse_iterator` adaptor to either a random access iterator or a bi-directional iterator an iterator is obtained that will traverse the same container in the reverse direction.

- **Insert iterators.** Iterators behave a lot like pointers; if you assign a new value to a container through its iterator, you will overwrite any value that is already at the location specified by the iterator. Sometimes, it is desirable to be able to **insert** a new element at that location in the container.

  The STL provides three different **insert iterators** that instead of overwriting the value at that location, actually insert a value there. Each type of adaptor performs the insertion into the container in a different place.

  - `back_insert_iterator`s add new elements to the end of the container. Because the algorithm uses the `back_insert` function to do the insertions it can only be used with `vector`s, `list`s, and `deque`s.

  - `front_insert_iterator`s add new elements to the front of a container. Because `push_front` is used for insertions, it can only be used on `list`s, and `deque`s.

  - `insert_iterator`s add new elements to the container at the location specified when the iterator is constructed. This iterator can be used to insert new elements anywhere within a container, not just at the front or the end. Because it uses `insert` to add the elements, it can be used with any of the STL containers.

- **Raw storage iterators.** Raw storage iterators allow algorithms to use raw, uninitialized memory during their execution. The `raw_storage_iterator` is used by several internal algorithms for partitioning and merging elements in a container.

## Allocators

Any package provided by the Standard C++ Library must be portable to many different machine architectures. Portability is one of C++'s strengths and the standards process helps to ensure that all compiler vendors have a common base to address this need.

One of the main problems to address in portability is the memory model of the machine. The memory model contains information about pointer types, the type of the difference between two pointers, the size of objects and also which primitives are used to allocate and deallocate raw memory. The STL encapsulates this information in a special class called an **allocator**. Allocators separate the STL from the dependencies of the underlying memory model of the machine architecture.

Each container is given an allocator when it is constructed. Whenever a container inserts or removes an element, it uses its allocator to allocate and deallocate the memory for the object. The container does not know anything about the memory model of the machine, it relies on the allocator for all of its memory needs.

C++ programmers who develop applications for Microsoft Windows are all too familiar with the multitude of memory models in the DOS and 16-bit Windows environments. However, allocators allow Windows programmers to easily mix models when using STL containers. It is (almost!) easy to have a `list` in far memory and a `vector` in near memory, simply by including the correct allocator. To create a `list` in far memory, we can write:

```
#include <faralloc.h>
#include <list.h>
list <int, far_allocator> fl;
```

I say `almost' because the current generation of compilers does not yet support default template arguments that are in fact templates themselves. The STL uses default template arguments to hide the allocator semantics from users. When you specify or declare a container without an allocator, you get the default allocator. As a workaround to this problem, the HP implementation relies on the C++ preprocessor to include the correct allocator for a container. To use a far `list`, we write:

```
#include <faralloc.h>
#define Allocator far_allocator
#include <list.h>
list<int> farList;
```

One problem with this workaround is that now it is impossible to use more than one allocator for a given container type in the same source file, since you are limited to reading in a header file just once. However, you may still use mixed-model containers across different source files. This problem will go away as compiler vendors update their compilers to include the STL.

## Try It!

I hope this article has given you a look under the hood of the STL. In fact, if you have not yet experimented with the STL I urge you to try it! You can obtain a public domain version of the STL from HP Labs via `ftp` from `butler.hpl.hp.com` [3]. I have been using this implementation with Borland's C++ compiler (Version 4.5). See the `stl.faq` [3] file for information about other platforms. If you are using the STL, consider sharing your experiences with others in this column! Write me for details at `wiseb@cs.rpi.edu`.

## References

**1**

ELLIS, M. A., AND STROUSTRUP, B. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass., 1990.

**2**

KOENIG, A. E. Working paper for draft proposed international standard for information systems -- programming language c++. Document Number: X3J16/95-0087, WG31/N0785, April 28 1995.

**3**

HEWLETT-PACKARD LABORATORIES. Public domain implementation of the standard template library. Available from ftp://butler.hpl.hp.com/stl/, October 31 1995.

**4**

LAJOIE, J. A five year retrospective. *C++ Report 7*, 8 (October 1995), 15--21,43.

**5**

STEPANOV, A., AND LEE, M. The standard template library (ansi/iso document). Available from ftp://butler.hpl.hp.com/stl/stl.zip/, October 31 1995.

**6**

STROUSTRUP, B. A perspective on iso c++. *C++ Report 7*, 8 (October 1995), 23--28.