# Programming in C++ using STL

Rex Jaeschke

Programming in C++ using STL



Edition: 2.0



**The training materials associated with this book are available for license. Interested parties should contact the author at the address below.**

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

 

# Preface

The Standard Template Library (STL) is a collection of template classes and algorithms that were designed to work together to provide a wide variety of functionality. The STL was developed by Alexander Stepanov and Meng Lee (at Hewlett-Packard) along with David R. Musser (at Rensselaer Polytechnic Institute). In an effort to gain widespread distribution and subsequent use of STL, HP placed its implementation in the public domain and successfully lobbied for STL's inclusion in the C++ standard.

The STL is somewhat different to most existing class libraries in that its algorithms operate on generic data. That is, they can handle almost any type of object yet they don't know anything about any one type. For example, the list class permits a list to be constructed from objects of an infinite number of types yet this class has no I/O operators defined, since knowing how to do I/O on objects of some type typically requires knowledge about that type.

## Reader Assumptions

A working knowledge of data structures is necessary to exploit fully the STL. If you have never taken a formal course in data structures or you are a bit rusty on the topic, it's a good idea to get an introductory text on that topic. A good text on the STL itself is "STL Tutorial and Reference Guide", second edition, by Musser, Derge, and Saini; Addison Wesley, 2001, ISBN 0-201-37923-6.

I assume that you know C++. Although experience with templates would be useful, it is not necessary.

## Limitations

This book introduces the STL. However, a very small percentage of library facilities are mentioned or covered in any detail. The STL contains so many functions that whole books have been written about that subject alone.

## Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training for some 14 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and focused. Many books use GUI and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other is; I simply know that my approach works well, and has formed the basis of my successful seminar business.

## Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory, within which each program has its own subdirectory.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory, within which each program has its own subdirectory.[1]

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

## Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C++ seminars provided useful feedback and located numerous typographical errors.

*Rex Jaeschke, September 2009*

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

 

# 1. Overview

In this chapter, we introduce the Standard Template Library and show how to use its main components.

## 1.1 Introduction

There are six main kinds of components in STL: containers, generic algorithms, iterators, function objects, adaptors, and allocators, and each is introduced in the following sections.

## 1.2 Containers

A *container* is an object that contains a collection of other objects. There are two kinds of containers: *sequence containers*, whose elements are organized in a linear fashion (as with vectors, lists, and deques), and *sorted associative containers* whose elements can be accessed by some sort of key (as with sets and maps).

The following example (see directory ov01) demonstrates the creation of some vectors and the use of several container-class member functions:

```
#include <iostream>
#include <vector>
using namespace std;

template<class T>
ostream& operator<<(ostream& os, const vector<T>& v);

int main()
{
/*1*/   vector<int> vi(5, 9);

/*2*/   cout << "vi: " << vi << ", size = " << vi.size() << '\n';

/*3*/   vi[1] = 1;
/*4*/   vi[3] = 3;

        cout << "vi: " << vi << ", size = " << vi.size() << '\n';
```

```
/*5*/   vector<double> vd;

        cout << "vd: " << vd << ", size = " << vd.size() << '\n';
/*6*/   cout << "vd is" << (vd.empty() ? " " : " not ") << "empty\n";

/*7*/   vd.push_back(1.5);
        vd.push_back(2.3);

        cout << "vd: " << vd << ", size = " << vd.size() << '\n';
        cout << "vd is" << (vd.empty() ? " " : " not ") << "empty\n";

        return 0;
}

template<class T>
ostream& operator<<(ostream& os, const vector<T>& v)
{
        unsigned long int vectorLength = v.size();

        os << '[';
        for (unsigned long int i = 0; i < vectorLength; ++i)
        {
/*8*/           os << ' ' << v[i];
        }
        os << ']';

        return os;
}
```

The output produced is:

```
vi: [ 9 9 9 9 9], size = 5
vi: [ 9 1 9 3 9], size = 5
vd: [], size = 0
vd is empty
vd: [ 1.5 2.3], size = 2
vd is not empty
```

In case 1, vector `vi` contains 5 `int`s each with value 9, whereas in case 5, `vd` is an empty vector of `double`s. Since the elements of a vector can be accessed randomly, `operator[]` is defined; we use it in cases 3, 4, and 8. In case 2, the member function `size` returns the number of elements currently stored in the given vector while in case 6, `empty` indicates whether the vector has zero elements. In case 7, function `push_back` adds a new element to the end of the vector.

 

For the most part, a member function exists for each container class for which that operation makes sense and for which that operation can be implemented in a reasonably efficient manner. For example, `size` and `push_back` exist for deques and lists as well as for vectors.

The following example (see directory ov02) demonstrates the creation of a map called `ages` in which the names of people are inserted along with their corresponding ages:

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
/*1*/   map<string, unsigned int> ages;

/*2*/   ages["John"]  = 40;
        ages["Mary"]  = 39;
        ages["Simon"] = 12;
        ages["Jane"]  = 14;

        string name;
        while (true)
        {
                cout << "Enter a name (or 'exit' to end): ";
                cin >> name;
                if (name == "exit")
                {
                        break;
                }

/*3*/           if (ages.find(name) != ages.end())
                {
                        cout << "The age of " << name << " is "
                               << ages[name] << ".\n";
                }
                else
                {
                        cout << "Person " << name << " is unknown.\n";
                }
        }

        return 0;
}
```

# 4. Iterators

In this chapter, we take a more in-depth look at iterators.

## 4.1 Introduction

An *iterator* is an object that behaves like a pointer in that an iterator contains a value that somehow leads to an element in a container. In fact, an iterator behaves so much like a pointer that *every* template function that accepts an iterator argument also accepts a pointer instead.

For any given iterator *i*, the expression `*i` is well defined for an element of any class, enumeration, or built-in type. And for elements of class type, `i->m` is well defined for any member *m*. An iterator can be mutable or constant; that is, it can allow or disallow, respectively, modification of the element to which it points. If it disallows modification, the expressions `*i` and `i->m` cannot be used in the context of a modifiable lvalue.

An iterator can point to any element in a container. It can also point one element beyond the end of that container; however, if iterator *i* points to this non-existent element, the value of the expression `*i` is undefined.

Many of STL's algorithms operate on a range of elements. A range is specified using a pair of iterators that designate the start and end positions for the operation. Ranges are written using the notation [*start*,*end*), which means that the position start is included whereas position end is not. Therefore, [*i*,*i*) indicates an empty range. Range [*i*,*j*) is valid only if *j* is reachable from *i*; that is, we must be able to go from *i* to *j* by applying a finite sequence of `++i` operations. If an algorithm is given an invalid range, the behavior is undefined.

There are five categories of standard iterators: input, output, forward, bidirectional, and random access. Each is discussed in one of the sections following.

## 4.2 Input Iterators

These iterators are used to access elements from an existing sequence. They must have the following defined: `operator==`, `operator!=`, prefix and postfix `operator++`, and unary `operator*`; however, `operator*` only has to be able to retrieve an element's value, not be able to change it. It is not possible to make a copy of an input iterator and to use it later on to start traversing from that position; input iterators can be used only by single-pass algorithms.

Here is an example of how the algorithms `accumulate` and `find` might be implemented and used:

```
template<class InputIterator, class T>
T accumulate(InputIterator start, InputIterator end, T value)
{
        while (start != end)
        {
                value = value + *start;
                ++start;
        }
        return value;
}

template<class InputIterator, class T>
InputIterator find(InputIterator start, InputIterator end, const T& value)
{
        while (start != end)
        {
                if (*start == value)
                {
                        break;
                }
                ++start;
        }
        return start;
}

int values[] = {9, 8, 7, 6, 5, 4, 3};
vector<int> vi1(&values[0], &values[7]);

int sum1 = accumulate(vi1.begin(), vi1.end(), 0);

vector<int>::iterator x1 = find(vi1.begin(), vi1.end(), 20);
if (x1 == vi1.end())
{
        // No such element
}
```

As we can see, both functions take a range specified using two input iterator arguments. Since vector `vi1` is a non-`const` object, the `begin` and `end` functions called are those with the following signatures:

```
iterator begin();
iterator end();
```

that is, they each return a mutable iterator. And if we want to define a variable of this iterator type, we must give it the type `vector<int>::iterator`. Consider what happens if we define a `const` vector `vi2`:

 

```
const vector<int> vi2(&values[0], &values[7]);

int sum2 = accumulate(vi2.begin(), vi2.end(), 0);

vector<int>::const_iterator x2 = find(vi2.begin(), vi2.end(), 20);
```

Now the `begin` and `end` functions called are those with the following signatures:

```
const_iterator begin() const;
const_iterator end() const;
```

that is, they each return a constant iterator. Now we must use the type `vector<int>::const_iterator`. Note carefully that the name `const_iterator` can be misleading; it is not the iterator that is constant, but rather, the iterator points to a constant object. So while we cannot change the value of the object it points to, we can change the value of the iterator itself (using `operator++`, for example).

## 4.3 Output Iterators

These iterators are used to generate a new sequence. They must have the following defined: prefix and postfix `operator++`, and unary `operator*`; however, `operator*` can only be used to change an element's value, not to retrieve it. As such, a constant iterator cannot be used as an output iterator. It is not possible to make a copy of an output iterator and to use it later on to start traversing from that position; output iterators can be used only by single-pass algorithms.

Here is an example of how the algorithm `copy` might be implemented:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator start, InputIterator end, OutputIterator dest)
{
        while (start != end)
        {
                *dest++ = *start++;
        }
        return dest;
}
```

In this example, we retrieve via the input iterator and store through the output iterator. And we compare for inequality using the input operator.

The beauty of iterators is that they allow us to write generic algorithms that deal with different container types at the same time. For example:

```
int values[] = {0, 1, 2};
vector<int> vi(&values[0], &values[3]);
list<int> li(6, 4);

copy(vi.begin(), vi.end(), li.begin());
```

# 6. Adaptors

In this chapter, we look at ways in which containers, iterators, and function objects can be adapted to provide new storage mechanisms and extra operations.

## 6.1 Introduction

An *adaptor* is a component that can be used to change the interface of another component. There are three kinds of adaptors: container adaptors, iterator adaptors, and function adaptors. Each is discussed in the following sections.

## 6.2 Container Adaptors

An example of a container adaptor is a stack adaptor which can be used to provide a stack-like interface to a vector, list, or deque, by restricting all operations other than those permitted for a stack. Likewise, a queue adaptor can be used to adapt a vector or list as a queue, and a priority queue adaptor can be used to adapt a vector or deque as a priority queue. Here are the relevant parts of these adaptor containers' definitions:

```
template <class T, class Container = deque<T> >
class queue
{
public:
        typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;
        typedef typename Container container_type;
protected:
        Container c;
public:
        // ...
};
```

Since the default type of the underlying container is a deque, the following definitions are equivalent:

```
queue<int> q;
queue<int, deque<int> > q;
```

The priority class queue not only needs an underlying container type for its internal representation, it also needs a function object to figure out the ordering. By default, `less` is used such that the highest priority elements are placed at the front of the queue.

```
template <class T, class Container = vector<T>, class Compare =
less<Container::value_type> >
class priority_queue
{
public:
        typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;
        typedef typename Container container_type;
protected:
        Container c;
        Compare comp;
public:
        // ...
};
```

Like `queue`, class `stack` also defaults to using a deque as its underlying container type:

```
template <class T, class Container = deque<T> >
class stack
{
public:
        typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;
        typedef typename Container container_type;
protected:
        Container c;
public:
        // ...
};
```

## 6.3 Iterator Adaptors

An *iterator adaptor* can be used to change the interface of an iterator component. There are two kinds of iterator adaptors: reverse iterators and insert iterators.

Any random access iterator can have applied to it a `reverse_iterator` adaptor. A bidirectional iterator can have applied to it a `reverse_bidirectional_iterator`. However, we normally don't need to use these adaptors directly, since the standard container classes define the iterator types `reverse_iterator` and `const_reverse_iterator`, and member functions `rbegin` and `rend` that produce them. For example:

 

```
int values[] = {1, 2, 3, 4, 5};
vector<int> v(&values[0], &values[5]);

vector<int>::reverse_iterator i;

for (i = v.rbegin(); i != v.rend(); ++i)
{
        //
}
```

Note that the reverse iterator is incremented from the end to the start; that is, the sense of direction is reversed. In particular,

```
sort(v.rbegin(), v.rend());
```

sorts the elements in v in descending order. It really does them in ascending order, but since it is traversing from the end backwards, ascending in that direction is really descending from the point of view of the other direction.

Ordinarily, algorithms that change the contents of a container do so by overwriting the values of one or more elements. By using an *insert iterator*, we can change this behavior, allowing new elements to be inserted instead. The `back_inserter` iterator causes the container's `push_back` function to be called; `front_inserter` iterator causes the container's `push_front` function to be called; and `inserter` iterator causes the container's `insert` function to be called. Here's an example of their use (see directory ad01) along with the output produced:

```
int values[] = {3, 5, 7};
vector<int> vi(&values[0], &values[3]);
list<int> li(3, 9);

cout << "li: " << li << '\n';
copy(vi.begin(), vi.end(), back_inserter(li));
cout << "li: " << li << '\n';

li:  9 9 9
li:  9 9 9 3 5 7
```

As we can see, the three new elements were appended to the end of `li` in the same order they existed in `vi`.

```
copy(vi.begin(), vi.end(), front_inserter(li));
cout << "li: " << li << '\n';

li:  7 5 3 9 9 9 3 5 7
```

Here the three new elements were added to the front of `li` one at a time, in the same order they existed in `vi`; that is, 3 was added to the front, then 5 was added to the front, and finally, 7 was added.