**Definition of STL**

The **Standard Template Library** (**STL**) is a software **library** for the C++ programming language that influenced many parts of the C++ **Standard Library**. It provides four components called algorithms, containers, functional, and iterators..

**Introduction to the Standard Template Library**

STL provides many of the basic algorithms and data structures of computer science. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template. We should make sure that you understand how templates work in C++ before you use the STL. Components of STL are:

**Containers and algorithms:**

Like many class libraries, the STL includes *container* classes: classes whose purpose is to contain other objects. The STL includes vector, list, deque, set, multiset, map, multimap, hash set, hash multiset, hash map and hash multimap.

Each of these classes is a template, and can be instantiated to contain any type of object. You can, for example, use a vector<int> in much the same way as you would use an ordinary C array, except that vector liminates the chore of managing dynamic memory allocation by hand.

**vector<int> v(3); // Declare a vector of 3 elements.**

**v[0] = 7;**

**v[1] = v[0] + 3;**

**v[2] = v[0] + v[1]; // v[0] == 7, v[1] == 10, v[2] == 17**

The STL also includes a large collection of *algorithms* that manipulate the data stored in containers. You can reverse the order of elements in a vector, for example, by using the [reverse](https://www.sgi.com/tech/stl/reverse.html) algorithm.

**Reverse (v.begin(), v.end()); // v[0] == 17, v[1] == 10, v[2] == 7**

There are two important points to notice about this call to reverse. First, it is a global function, not a member function. Second, it takes two arguments rather than one: it operates on a *range* of elements, rather than on a container. In this particular case the range happens to be the entire container v.

The reason for both of these facts is the same: reverse, like other STL algorithms, is decoupled from the STL container classes. This means that reversecan be used not only to reverse elements in vectors, but also to reverse elements in lists, and even elements in C arrays. The following program is also valid.

**double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };**

**reverse(A, A + 6);**

**for (int i = 0; i < 6; ++i)**

**cout << "A[" << i << "] = " << A[i];**

This example uses a *range*, just like the example of reversing a vector: the first argument to reverse is a pointer to the beginning of the range, and the second argument points one element past the end of the range. This range is denoted [A, A + 6); the asymmetrical notation is a reminder that the two endpoints are different, that the first is the beginning of the range and the second is *one past* the end of the range.

**Iterators:**

The concept of an iterator is fundamental to understanding the C++ Standard Template Library (STL) because iterators provide a means for accessing data stored in container classes such a vector, map, list, etc.

You can think of an iterator as pointing to an item that is part of a larger container of items. For instance, all containers support a function called begin, which will return an iterator pointing to the beginning of the container (the first element) and function, end, that returns an iterator corresponding to having reached the end of the container. In fact, you can access the element by "dereferencing" the iterator with a \*, just as you would dereference a pointer.

To request an iterator appropriate for a particular STL templated class, you use the syntax

**std::class\_name<template\_parameters>::iterator name**

where name is the name of the iterator variable you wish to create and the class\_name is the name of the STL container you are using, and the template\_parameters are the parameters to the template used to declare objects that will work with this iterator. Note that because the STL classes are part of the std namespace, you will need to either prefix every container class type with "std::", as in the example, or include "using namespace std;" at the top of your program.

For instance, if you had an STL vector storing integers, you could create an iterator for it as follows:

**std::vector<int> myIntVector;**

**std::vector<int>::iterator myIntVectorIterator;**

Different operations and containers support different types of iterator behavior. In fact, there are several different classes of iterators, each with slightly different properties. First, iterators are distinguished by whether you can use them for reading or writing data in the container. Some types of iterators allow for both reading and writing behavior, though not necessarily at the same time.

Some of the most important are the forward, backward and the bidirectional iterators. Both of these iterators can be used as either input or output iterators, meaning you can use them for either writing or reading. The forward iterator only allows movement one way -- from the front of the container to the back. To move from one element to the next, the increment operator, ++, can be used.

For instance, if you want to access the elements of an STL vector, it's best to use an iterator instead of the traditional C-style code. The strategy is fairly straightforward: call the container's begin function to get an iterator, use ++ to step through the objects in the container, access each object with the \* operator ("\*iterator") similar to the way you would access an object by dereferencing a pointer, and stop iterating when the iterator equals the container's end iterator. You can compare iterators using != to check for inequality, == to check for equality. (This only works when the iterators are operating on the same container!)

**The STL approach (use this)**

**using namespace std;**

**vector<int> myIntVector;**

**vector<int>::iterator myIntVectorIterator;**

**// Add some elements to myIntVector**

**myIntVector.push\_back(1);**

**myIntVector.push\_back(4);**

**myIntVector.push\_back(8);**

**for(myIntVectorIterator = myIntVector.begin();**

**myIntVectorIterator != myIntVector.end();**

**myIntVectorIterator++)**

**{**

**cout<<\*myIntVectorIterator<<" ";**

**//Should output 1 4 8**

**}**

As you might imagine, you can use the decrement operator, --, when working with a bidirectional iterator or a backward operator.

Iterators are often handy for specifying a particular range of things to operate on. For instance, the range item.begin(), item.end() is the entire container, but smaller slices can be used. This is particularly easy with one other, extremely general class of iterator, the random access iterator, which is functionally equivalent to a pointer in C or C++ in the sense that you can not only increment or decrement but also move an arbitrary distance in constant time (for instance, jump multiple elements down a vector).

For instance, the iterators associated with vectors are random access iterators so you could use arithmetic of the form

**iterator + n**

where n is an integer. The result will be the element corresponding to the nth item after the item pointed to be the current iterator. This can be a problem if you happen to exceed the bounds of your iterator by stepping forward (or backward) by too many elements.

The following code demonstrates both the use of random access iterators and exceeding the bounds of the array (don't run it!):

**vector<int> myIntVector;**

**vector<int>::iterator myIntVectorIterator;**

**myIntVectorIterator = myIntVector.begin() + 2;**

You can also use the standard arithmetic shortcuts for addition and subtraction, += and -=, with random access iterators. Moreover, with random access iterators you can use <, >, <=, and >= to compare iterator positions within the container.

Iterators are also useful for some functions that belong to container classes that require operating on a range of values. A simple but useful example is the erase function. The vector template supports this function, which takes a range as specified by two iterators -- every element in the range is erased. For instance, to erase an entire vector:

**vector<int>::iterator myIntVectorIterator;**

**myIntVector.erase(myIntVectorIterator.begin(), myIntVectorIterator.end());**

which would delete all elements in the vector. If you only wanted to delete the first two elements, you could use **myIntVector.erase(myIntVectorIterator.begin(), myIntVectorIterator.begin()+2);**

Note that various container class support different types of iterators -- the vector class, which has served as our model for iterators, supports a random access iterator, the most general kind. Another container, the list container (to be discussed later), only supports bidirectional iterators.

So why use iterators? First, they're a flexible way to access the data in containers that don't have obvious means of accessing all of the data.They're also quite flexible -- if you change the underlying container, it's easy to change the associated iterator so long as you only use features associated with the iterator supported by both classes. Finally, the STL algorithms defined in <algorithm> use iterators.

**Summary**

The STL provides iterators as a convenient abstraction for accessing many different types of containers.

Iterators for templated classes are generated inside the class scope with the syntax

**class\_name<parameters>::iterator**

Iterators can be thought of as limited pointers (or, in the case of random access iterators, as nearly equivalent to pointers)

Iterators do not provide bounds checking; it is possible to overstep the bounds of a container, resulting in segmentation faults

Different containers support different iterators, so it is not always possible to change the underlying container type without making changes to your code

Iterators can be invalidated if the underlying container (the container being iterated over) is changed significantly

In the example of reversing a C array, the arguments to reverse are clearly of type double\*. What are the arguments to reverse if you are reversing a vector, though, or a list? That is, what exactly does reverse declare its arguments to be, and what exactly do v.begin() and v.end() return?

The answer is that the arguments to reverse are *iterators*, which are a generalization of pointers. Pointers themselves are iterators, which is why it is possible to reverse the elements of a C array. Similarly, vector declares the nested types iterator and const\_iterator. In the example above, the type returned by v.begin() and v.end() is vector<int>::iterator. There are also some iterators, such as istream\_iterator and ostream\_iterator, that aren't associated with containers at all.

Iterators are the mechanism that makes it possible to decouple algorithms from containers: algorithms are templates, and are parameterized by the type of iterator, so they are not restricted to a single type of container. Consider, for example, how to write an algorithm that performs linear search through a range. This is the STL's [find](https://www.sgi.com/tech/stl/find.html) algorithm.

**template <class InputIterator, class T>**

**InputIterator find(InputIterator first, InputIterator last, const T& value)**

**{**

**while (first != last && \*first != value) ++first;**

**return first;**

**}**

Find takes three arguments: two iterators that define a range, and a value to search for in that range. It examines each iterator in the range [first, last), proceeding from the beginning to the end, and stops either when it finds an iterator that points to value or when it reaches the end of the range.

First and last are declared to be of type InputIterator, and InputIterator is a template parameter. That is, there isn't actually any type calledInputIterator: when you call find, the compiler substitutes the actual type of the arguments for the formal type parameters InputIterator and T. If the first two arguments to find are of type int\* and the third is of type int, then it is as if you had called the following function.

**int\* find(int\* first, int\* last, const int& value)**

**{**

**while (first != last && \*first != value) ++first;**

**return first;**

**}**

**Concepts and Modeling:** One very important question to ask about any template function, not just about STL algorithms, is what the set of types is that may correctly be substituted for the formal template parameters. Clearly, for example, int\* or double\* may be substituted for find's formal template parameter InputIterator. Equally clearly, int or double may not: find uses the expression \*first, and the dereference operator makes no sense for an object of type int or of type double. The basic answer, then, is that find implicitly defines a set of requirements on types, and that it may be instantiated with any type that satisfies those requirements. Whatever type is substituted for InputIterator must provide certain operations: it must be possible to compare two objects of that type for equality, it must be possible to increment an object of that type, it must be possible to dereference an object of that type to obtain the object that it points to, and so on.

Find isn't the only STL algorithm that has such a set of requirements; the arguments to for\_each and [count](https://www.sgi.com/tech/stl/count.html), and other algorithms, must satisfy the same requirements. These requirements are sufficiently important that we give them a name: we call such a set of type requirements a *concept*, and we call this particular concept [Input Iterator](https://www.sgi.com/tech/stl/InputIterator.html). We say that a type *conforms to a concept*, or that it *is a model of a concept*, if it satisfies all of those requirements. We say that int\* is a model of Input Iterator because int\* provides all of the operations that are specified by the Input Iterator requirements.

Concepts are not a part of the C++ language; there is no way to declare a concept in a program, or to declare that a particular type is a model of a concept. Nevertheless, concepts are an extremely important part of the STL. Using concepts makes it possible to write programs that cleanly separate interface from implementation: the author of find only has to consider the interface specified by the concept Input Iterator, rather than the implementation of every possible type that conforms to that concept. Similarly, if you want to use find, you need only to ensure that the arguments you pass to it are models ofInput Iterator. This is the reason why find and reverse can be used with lists, vectors, C arrays, and many other types: programming in terms of concepts, rather than in terms of specific types, makes it possible to reuse software components and to combine components together.

**Refinement:** Input Iterator is, in fact, a rather weak concept: that is, it imposes very few requirements. An Input Iterator must support a subset of pointer arithmetic (it must be possible to increment an Input Iterator using prefix and postfix operator++), but need not support all operations of pointer arithmetic. This is sufficient for [find](https://www.sgi.com/tech/stl/find.html), but some other algorithms require that their arguments satisfy additional requirements. [Reverse](https://www.sgi.com/tech/stl/reverse.html), for example, must be able to decrement its arguments as well as increment them; it uses the expression --last. In terms of concepts, we say that reverse's arguments must be models of[BidirectionalHYPERLINK "https://www.sgi.com/tech/stl/BidirectionalIterator.html" Iterator](https://www.sgi.com/tech/stl/BidirectionalIterator.html) rather than Input Iterator.

The Bidirectional Iterator concept is very similar to the Input Iterator concept: it simply imposes some additional requirements. The types that are models of Bidirectional Iterator are a subset of the types that are models of Input Iterator: every type that is a model of Bidirectional Iterator is also a model of Input Iterator. Int\*, for example, is both a model of Bidirectional Iterator and a model of Input Iterator, but istream\_iterator, is only a model of Input Iterator: it does not conform to the more stringent Bidirectional Iterator requirements.

We describe the relationship between Input Iterator and Bidirectional Iterator by saying that Bidirectional Iterator is a *refinement* of Input Iterator. Refinement of concepts is very much like inheritance of C++ classes; the main reason we use a different word, instead of just calling it "inheritance", is to emphasize that refinement applies to concepts rather than to actual types.

There are actually three more iterator concepts in addition to the two that we have already discussed: the five iterator concepts are [Output Iterator](https://www.sgi.com/tech/stl/OutputIterator.html), [Input Iterator](https://www.sgi.com/tech/stl/InputIterator.html), [Forward Iterator](https://www.sgi.com/tech/stl/ForwardIterator.html), [Bidirectional Iterator](https://www.sgi.com/tech/stl/BidirectionalIterator.html), and [Random Access Iterator](https://www.sgi.com/tech/stl/RandomAccessIterator.html); Forward Iterator is a refinement of Input Iterator, Bidirectional Iterator is a refinement of Forward Iterator, and Random Access Iterator is a refinement of Bidirectional Iterator. ([Output Iterator](https://www.sgi.com/tech/stl/OutputIterator.html) is related to the other four concepts, but it is not part of the hierarchy of refinement: it is not a refinement of any of the other iterator concepts, and none of the other iterator concepts are refinements of it.) The [*Iterator Overview*](https://www.sgi.com/tech/stl/Iterators.html) has more information about iterators in general.

Container classes, like iterators, are organized into a hierarchy of concepts. All containers are models of the concept [Container](https://www.sgi.com/tech/stl/Container.html); more refined concepts, such as [Sequence](https://www.sgi.com/tech/stl/Sequence.html) and [Associative Container](https://www.sgi.com/tech/stl/AssociativeContainer.html), describe specific types of containers.

**Other parts of the STL:** If you understand algorithms, iterators, and containers, then you understand almost everything there is to know about the STL. The STL does, however, include several other types of components.

First, the STL includes several *utilities*: very basic concepts and functions that are used in many different parts of the library. The concept [Assignable](https://www.sgi.com/tech/stl/Assignable.html), for example, describes types that have assignment operators and copy constructors; almost all STL classes are models of Assignable, and almost all STL algorithms require their arguments to be models of Assignable.

Second, the STL includes some low-level mechanisms for allocating and deallocating memory. [*Allocators*](https://www.sgi.com/tech/stl/Allocators.html) are very specialized, and you can safely ignore them for almost all purposes.

Finally, the STL includes a large collection of [*function objects*](https://www.sgi.com/tech/stl/functors.html), also known as *functors*. Just as iterators are a generalization of pointers, function objects are a generalization of functions: a function object is anything that you can call using the ordinary function call syntax. There are several different concepts relating to function objects, including [Unary Function](https://www.sgi.com/tech/stl/UnaryFunction.html) (a function object that takes a single argument, *i.e.* one that is called as f(x)) and [Binary HYPERLINK "https://www.sgi.com/tech/stl/BinaryFunction.html"Function](https://www.sgi.com/tech/stl/BinaryFunction.html)(a function object that takes two arguments, *i.e.* one that is called as f(x, y)). Function objects are an important part of generic programming because they allow abstraction not only over the types of objects, but also over the operations that are being performed.

**Vector Class**

**Vector**: Vectors are sequence containers representing arrays that can change in size.  
Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.  
Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.  
Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual [capacity](http://www.cplusplus.com/vector::capacity) greater than the storage strictly needed to contain its elements (i.e., its [size](http://www.cplusplus.com/vector::size)). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of [size](http://www.cplusplus.com/vector::size) so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (as explained below in  push\_back).  
Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.  
Compared to the other dynamic sequence containers (deques, [lists](http://www.cplusplus.com/list) and forward\_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its [end](http://www.cplusplus.com/vector::end). For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than [lists](http://www.cplusplus.com/list) and forward\_lists.

**Library: #include<vector>**

**Syntax: vector <int>numbers;**

**Container properties:**

***Sequence:*** Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

***Dynamic array:*** Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.

***Allocator-aware:*** The container uses an allocator object to dynamically handle its storage needs.

**Template parameters:**

**T:** The type of the element to be stored in the vector  
Only if T [is guaranteed to not throw while moving](http://www.cplusplus.com/is_nothrow_move_constructible), implementations can optimize to move elements instead of copying them during reallocations.  
Aliased as member type vector::value\_type.

**Alloc:** The type of the object that allocates and deallocates memory for the vector.Type of the allocator object used to define the storage allocation model. By default, the [allocator](http://www.cplusplus.com/allocator) class template is used, which defines the simplest memory allocation model and is value-independent.This argument is optional and the default value is **std::allocator***<Type>.*  
Aliased as member type vector::allocator\_type.

**Member Types:**

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| value\_type | The first template parameter (T) |  |
| allocator\_type | The second template parameter (Alloc) | defaults to:[allocator](http://www.cplusplus.com/allocator)<value\_type> |
| reference | allocator\_type::reference | for the default allocator:value\_type& |
| const\_reference | allocator\_type::const\_reference | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type& |
| pointer | allocator\_type::pointer | for the default allocator:value\_type\* |
| const\_pointer | allocator\_type::const\_pointer | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type\* |
| iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to value\_type | convertible to const\_iterator |
| const\_iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to const value\_type |  |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | a signed integral type, identical to:iterator\_traits<iterator>::difference\_type | usually the same as ptrdiff\_t |
| size\_type | an unsigned integral type that can represent any non-negative value of difference\_type | usually the same as size\_t |

**Member Functions:**

[**(HYPERLINK "http://www.cplusplus.com/reference/vector/vector/vector/"constructorHYPERLINK "http://www.cplusplus.com/reference/vector/vector/vector/")**](http://www.cplusplus.com/reference/vector/vector/vector/): Construct vector (public member function

[**(HYPERLINK "http://www.cplusplus.com/reference/vector/vector/~vector/"destructorHYPERLINK "http://www.cplusplus.com/reference/vector/vector/~vector/")**](http://www.cplusplus.com/reference/vector/vector/~vector/) : Vector destructor (public member function )

**Operator:** Assign content (public member function )

**Iterators:**

[**begin**](http://www.cplusplus.com/reference/vector/vector/begin/): Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/vector/vector/end/) :Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/vector/vector/rbegin/): Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/vector/vector/rend/): Return reverse iterator to reverse end (public member function )

[**rend**](http://www.cplusplus.com/reference/vector/vector/rend/) :Return reverse iterator to reverse end (public member function )

[**cbeginHYPERLINK "http://www.cplusplus.com/reference/vector/vector/cbegin/"**](http://www.cplusplus.com/reference/vector/vector/cbegin/) Return const\_iterator to beginning (public member function )

[**cendHYPERLINK "http://www.cplusplus.com/reference/vector/vector/cend/"**](http://www.cplusplus.com/reference/vector/vector/cend/) :Return const\_iterator to end (public member function )

[**crbeginHYPERLINK "http://www.cplusplus.com/reference/vector/vector/crbegin/"**](http://www.cplusplus.com/reference/vector/vector/crbegin/) :Return const\_reverse\_iterator to reverse beginning (public member function )

[**crendHYPERLINK "http://www.cplusplus.com/reference/vector/vector/crend/"**](http://www.cplusplus.com/reference/vector/vector/crend/) :Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**size**](http://www.cplusplus.com/reference/vector/vector/size/) :Return size (public member function )

[**max\_size**](http://www.cplusplus.com/reference/vector/vector/max_size/): Return maximum size (public member function )

[**resize**](http://www.cplusplus.com/reference/vector/vector/resize/): Change size (public member function )

[**capacity**](http://www.cplusplus.com/reference/vector/vector/capacity/): Return size of allocated storage capacity (public member function )

[**empty**](http://www.cplusplus.com/reference/vector/vector/empty/): Test whether vector is empty (public member function )

[**reserve**](http://www.cplusplus.com/reference/vector/vector/reserve/): Request a change in capacity (public member function )

[**shrink\_to\_HYPERLINK "http://www.cplusplus.com/reference/vector/vector/shrink\_to\_fit/"fitHYPERLINK "http://www.cplusplus.com/reference/vector/vector/shrink\_to\_fit/"**](http://www.cplusplus.com/reference/vector/vector/shrink_to_fit/): Shrink to fit (public member function )

**Element access**:

[**operator[HYPERLINK "http://www.cplusplus.com/reference/vector/vector/operator%5b%5d/"]**](http://www.cplusplus.com/reference/vector/vector/operator%5b%5d/):Access element (public member function )

[**at**](http://www.cplusplus.com/reference/vector/vector/at/): Access element (public member function )

[**front**](http://www.cplusplus.com/reference/vector/vector/front/): Access first element (public member function )

[**back**](http://www.cplusplus.com/reference/vector/vector/back/): Access last element (public member function )

[**dataHYPERLINK "http://www.cplusplus.com/reference/vector/vector/data/":HYPERLINK "http://www.cplusplus.com/reference/vector/vector/data/"**](http://www.cplusplus.com/reference/vector/vector/data/) Access data (public member function )

**Modifiers**:

[**assign**](http://www.cplusplus.com/reference/vector/vector/assign/): Assign vector content (public member function )

[**push\_back**](http://www.cplusplus.com/reference/vector/vector/push_back/): Add element at the end (public member function )

[**pop\_back**](http://www.cplusplus.com/reference/vector/vector/pop_back/): Delete last element (public member function )

[**insert**](http://www.cplusplus.com/reference/vector/vector/insert/): Insert elements (public member function )

[**erase**](http://www.cplusplus.com/reference/vector/vector/erase/) :Erase elements (public member function )

[**swap**](http://www.cplusplus.com/reference/vector/vector/swap/) Swap content (public member function )

[**clear**](http://www.cplusplus.com/reference/vector/vector/clear/) :Clear content (public member function )

[**emplaceHYPERLINK "http://www.cplusplus.com/reference/vector/vector/emplace/"**](http://www.cplusplus.com/reference/vector/vector/emplace/) :Construct and insert element (public member function )

[**emplace\_HYPERLINK "http://www.cplusplus.com/reference/vector/vector/emplace\_back/"backHYPERLINK "http://www.cplusplus.com/reference/vector/vector/emplace\_back/"**](http://www.cplusplus.com/reference/vector/vector/emplace_back/): Construct and insert element at the end (public member function )

**Allocator**:

[**get\_allocator**](http://www.cplusplus.com/reference/vector/vector/get_allocator/): Get allocator (public member function )

**Non Member Function overloads:**

[**relationalHYPERLINK "http://www.cplusplus.com/reference/vector/vector/operators/" operators**](http://www.cplusplus.com/reference/vector/vector/operators/): Relational operators for vector (function template )

[**swap**](http://www.cplusplus.com/reference/vector/vector/swap-free/) :Exchange contents of vectors (function template )

**Templates specialization:**

[**vectorHYPERLINK "http://www.cplusplus.com/reference/vector/vector-bool/"<HYPERLINK "http://www.cplusplus.com/reference/vector/vector-bool/"boolHYPERLINK "http://www.cplusplus.com/reference/vector/vector-bool/">**](http://www.cplusplus.com/reference/vector/vector-bool/): Vector of bool (class template specialization )

When you are not sure what type of sequence container to use, use vector.

**template**

**<**

**class Type,**

**class Allocator = allocator<Type>**

**>**

**class vector**

[Remarks](javascript:void(0))

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires linear time. The performance of the deque Class container is superior with respect to insertions and deletions at the beginning and end of a sequence. The [list Class](https://msdn.microsoft.com/en-us/library/802d66bt.aspx) container is superior with respect to insertions and deletions at any location within a sequence.

Vector reallocation occurs when a member function must increase the number of elements contained in the vector beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the sequence become invalid. If no reallocation happens, only iterators and references before the insertion/deletion point remain valid.

The [vectorHYPERLINK "https://msdn.microsoft.com/en-us/library/t0723a54.aspx"<HYPERLINK "https://msdn.microsoft.com/en-us/library/t0723a54.aspx"boolHYPERLINK "https://msdn.microsoft.com/en-us/library/t0723a54.aspx">HYPERLINK "https://msdn.microsoft.com/en-us/library/t0723a54.aspx" Class](https://msdn.microsoft.com/en-us/library/t0723a54.aspx) is a full specialization of the class template vector for elements of type bool with an allocator for the underlying type used by the specialization.

The [vectorHYPERLINK "https://msdn.microsoft.com/en-us/library/t248k97e.aspx"<HYPERLINK "https://msdn.microsoft.com/en-us/library/t248k97e.aspx"boolHYPERLINK "https://msdn.microsoft.com/en-us/library/t248k97e.aspx">HYPERLINK "https://msdn.microsoft.com/en-us/library/t248k97e.aspx" reference Class](https://msdn.microsoft.com/en-us/library/t248k97e.aspx) is a nested class whose objects are able to provide references to elements (single bits) within a vector<bool> object.

[Members](javascript:void(0)) [Constructors](javascript:void(0))

|  |  |
| --- | --- |
| [vector](https://msdn.microsoft.com/en-us/library/zzw4bwhd.aspx) | Constructs a vector of a specific size or with elements of a specific value or with a specific **allocator** or as a copy of some other vector. |

[Typedefs](javascript:void(0))

|  |  |
| --- | --- |
| [allocator\_type](https://msdn.microsoft.com/en-us/library/b5hh368e.aspx) | The type of the **allocator** used by the vector. |
| [const\_iterator](https://msdn.microsoft.com/en-us/library/352sf8za.aspx) | A type that represents a random-access read-only iterator. |
| [const\_pointer](https://msdn.microsoft.com/en-us/library/hkha4ta7.aspx) | A type that represents a read-only pointer to an element in a vector. |
| [const\_reference](https://msdn.microsoft.com/en-us/library/943xt47h.aspx) | A reference to a **const** element stored in a vector for reading and performing **const** operations. |
| [const\_reverse\_iterator](https://msdn.microsoft.com/en-us/library/h83bcbad.aspx) | A read-only reverse iterator. |
| [difference\_type](https://msdn.microsoft.com/en-us/library/e19sha3d.aspx) | A type that represents the difference between the addresses of two elements in a vector. |
| [iterator](https://msdn.microsoft.com/en-us/library/3xyxtaft.aspx) | A type that provides a random-access iterator that can read or modify any element in a vector. |
| [pointer](https://msdn.microsoft.com/en-us/library/7e4tx21z.aspx) | A type that represents a pointer to an element in a vector. |
| [reference](https://msdn.microsoft.com/en-us/library/6b1d6hf3.aspx) | A type that represents a reference to an element stored in a vector. |
| [reverse\_iterator](https://msdn.microsoft.com/en-us/library/47cxz14f.aspx) | A type that represents a reverse\_iterator. |
| [size\_type](https://msdn.microsoft.com/en-us/library/5ctzdcb6.aspx) | A type that represents the number of elements in a vector. |
| [value\_type](https://msdn.microsoft.com/en-us/library/0yf66c82.aspx) | A type that represents the data type stored in a vector. |

[Member Functions](javascript:void(0))

|  |  |
| --- | --- |
| [assign](https://msdn.microsoft.com/en-us/library/azbhc96f.aspx) | Replaces the specified elements with copies of the new elements. |
| [at](https://msdn.microsoft.com/en-us/library/c6kkh778.aspx) | Returns a reference to the element at a specified location in the vector and throws out\_of\_range  exception if the index is less than zero or greater than or equal to **size()**. |
| [back](https://msdn.microsoft.com/en-us/library/0532x4xk.aspx) | Returns a reference to the last element of the vector. |
| [begin](https://msdn.microsoft.com/en-us/library/1sc2e041.aspx) | Returns a random-access iterator to the first element in the vector. |
| [capacity](https://msdn.microsoft.com/en-us/library/azcztabt.aspx) | Returns the number of elements that the vector could contain without allocating more storage. |
| [cbegin](https://msdn.microsoft.com/en-us/library/dd647617.aspx) | Returns a random-access const iterator to the first element in the vector. |
| [cend](https://msdn.microsoft.com/en-us/library/dd647622.aspx) | Returns a random-access const iterator that points to one pastj the end of the vector. |
| [crbegin](https://msdn.microsoft.com/en-us/library/dd647615.aspx) | Returns a const reverse\_iterator to the last element in a vector. |
| [crend](https://msdn.microsoft.com/en-us/library/dd647621.aspx) | Returns a const reverse\_iterator that points to one before the first element in the vector. |
| [clear](https://msdn.microsoft.com/en-us/library/fs5a18ce.aspx) | Erases the elements of the vector. |
| [data](https://msdn.microsoft.com/en-us/library/dd647618.aspx) | Returns a pointer to the first element in the vector. |
| [emplace](https://msdn.microsoft.com/en-us/library/dd647616.aspx) | Inserts an element constructed in place into the vector at a specified position. |
| [emplace\_back](https://msdn.microsoft.com/en-us/library/dd647620.aspx) | Adds an element constructed in place to the end of the vector. |
| [empty](https://msdn.microsoft.com/en-us/library/86968s02.aspx) | Tests if the vector contains any elements. |
| [end](https://msdn.microsoft.com/en-us/library/txa4wa2y.aspx) | Returns a random-access iterator that points to one past the last element of the vector. |
| [erase](https://msdn.microsoft.com/en-us/library/ceh559x2.aspx) | Removes an element or a range of elements in a vector from specified positions. |
| [front](https://msdn.microsoft.com/en-us/library/0z70c7a5.aspx) | Returns a reference to the first element in a vector. |
| [get\_allocator](https://msdn.microsoft.com/en-us/library/15efhsex.aspx) | Returns the **allocator** used by the vector. |
| [insert](https://msdn.microsoft.com/en-us/library/s5bta5ha.aspx) | Inserts an element or a number of elements into the vector at a specified position. |
| [max\_size](https://msdn.microsoft.com/en-us/library/k3k7ekc0.aspx) | Returns the maximum length of the vector. |
| [pop\_back](https://msdn.microsoft.com/en-us/library/59ykakh8.aspx) | Deletes the element at the end of the vector. |
| [push\_back](https://msdn.microsoft.com/en-us/library/7fthz5xd.aspx) | Adds an element to the end of the vector. |
| [rbegin](https://msdn.microsoft.com/en-us/library/eh974a8w.aspx) | Returns an iterator to the first element in a reversed vector. |
| [rend](https://msdn.microsoft.com/en-us/library/c1xese40.aspx) | Returns a reverse\_iterator that points to one before the first element. |
| [reserve](https://msdn.microsoft.com/en-us/library/f7yseh4d.aspx) | Reserves a minimum length of storage for a vector object. |
| [resize](https://msdn.microsoft.com/en-us/library/wezs0zy6.aspx) | Specifies a new size for a vector. |
| [shrink\_to\_fit](https://msdn.microsoft.com/en-us/library/dd647619.aspx) | Discards excess capacity. |
| [size](https://msdn.microsoft.com/en-us/library/3y41k4hb.aspx) | Returns the number of elements in the vector. |
| [swap](https://msdn.microsoft.com/en-us/library/8762zzx6.aspx) | Exchanges the elements of two vectors. |

[Operators](javascript:void(0))

|  |  |
| --- | --- |
| operator[] | Returns a reference to the vector element at a specified position. |
| [operator=](https://msdn.microsoft.com/en-us/library/dd673471.aspx) | Replaces the elements of the vector with a copy of another vector. |

Vectors are more powerful than arrays because the number of functions that are available for accessing and modifying vectors. Unfortunately, the [] operator still does not provide bounds checking. There is an alternative way of accessing the vector, using the function at, which does provide bounds checking at an additional cost. Let's take a look at several functions provided by the vector class:

unsigned int size(); Returns the number of elements in a vector

push\_back(*type* element); Adds an element to the end of a vector

bool empty(); Returns true if the vector is empty

void clear(); Erases all elements of the vector

*type* at(int n); Returns the element at index n, with bounds checking

also, there are several basic operators defined for the vector class:

= Assignment replaces a vector's contents with the contents of another

== An element by element comparison of two vectors

[] Random access to an element of a vector (usage is similar to that

of the operator with arrays.) Keep in mind that it does not provide

bounds checking.

Let's take a look at an example program using the vector class:

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**int main()**

**{**

**vector <int> example; //Vector to store integers**

**example.push\_back(3); //Add 3 onto the vector**

**example.push\_back(10); //Add 10 to the end**

**example.push\_back(33); //Add 33 to the end**

**for(int x=0; x<example.size(); x++)**

**{**

**cout<<example[x]<<" "; //Should output: 3 10 33**

**}**

**if(!example.empty()) //Checks if empty**

**example.clear(); //Clears vector**

**vector <int> another\_vector; //Creates another vector to store integers**

**another\_vector.push\_back(10); //Adds to end of vector**

**example.push\_back(10); //Same**

**if(example==another\_vector) //To show testing equality**

**{**

**example.push\_back(20);**

**}**

**for(int y=0; y<example.size(); y++)**

**{**

**cout<<example[y]<<" "; //Should output 10 20**

**}**

**return 0;**

**}**

The following example demonstrates various techniques involving a vector and C++ Standard Library algorithms, notably shuffling, sorting, finding the largest element, and erasing from a vector using the erase-remove idiom.

**#include <iostream>**

**#include <vector>**

**#include <array>**

**#include <algorithm> // sort, max\_element, random\_shuffle, remove\_if, lower\_bound**

**#include <functional> // greater**

**#include <iterator> //begin, end, cbegin, cend, distance**

**// used here for convenience, use judiciously in real programs.**

**using namespace std;**

**using namespace std::placeholders;**

**auto main(int, char\*\*)**

**-> int**

**{**

**std::array<int,4> arr{ 1, 2, 3, 4 };**

**// initialize a vector from an array**

**vector<int> numbers( cbegin(arr), cend(arr) );**

**// insert more numbers into the vector**

**numbers.push\_back(5);**

**numbers.push\_back(6);**

**numbers.push\_back(7);**

**numbers.push\_back(8);**

**// the vector currently holds { 1, 2, 3, 4, 5, 6, 7, 8 }**

**// randomly shuffle the elements**

**random\_shuffle( begin(numbers), end(numbers) );**

**// locate the largest element, O(n)**

**auto largest = max\_element( cbegin(numbers), cend(numbers) );**

**cout << "The largest number is " << \*largest << "\n";**

**cout << "It is located at index " << distance(largest, cbegin(numbers)) << "\n";**

**// sort the elements**

**sort( begin(numbers), end(numbers) );**

**// find the position of the number 5 in the vector**

**auto five = lower\_bound( cbegin(numbers), cend(numbers), 5 );**

**cout << "The number 5 is located at index " << distance(five, cbegin(numbers)) << "\n";**

**// erase all the elements greater than 4**

**numbers.erase( remove\_if(begin(numbers), end(numbers),**

**bind(greater<>{}, \_1, 4) ), end(numbers) );**

**// print all the remaining numbers**

**for(const auto& element : numbers)**

**cout << element << " ";**

**return 0;**

**}**

The output will be the following:

**The largest number is 8**

**It is located at index 6 (implementation-dependent)**

**The number 5 is located at index 4**

**1 2 3 4  
  
Summary of Vector Benefits**: Vectors are somewhat easier to use than regular arrays. At the very least, they get around having to be resized constantly using new and delete. Furthermore, their immense flexibility - support for any datatype and support for automatic resizing when adding elements - and the other helpful included functions give them clear advantages to arrays.Another argument for using vectors are that they help avoid memory leaks--you don't have to remember to free vectors, or worry about how to handle freeing a vector in the case of an exception. This simplifies program flow and helps you write tighter code. Finally, if you use the at() function to access the vector, you get bounds checking at the cost of a slight performance penalty.