# C++ Reference Material
# STL Iterators and Iterator Adaptors

An STL iterator may be thought of as a generalized array pointer, which usually (but not always) points to a container component and is used to provide access to the value in that component for various purposes. This page contains a discussion of what is meant by an STL "range", and five tables:

1. A table showing the five iterator types and the operations each provides.
2. A table showing the type of iterator provided by each (first-class) container with its `iterator` typedef. (The container adaptor classes—stack, queue and priority_queue— do not support iterators of any kind.)
3. A table of iterator adaptors used with streams for input and output.
4. A table of iterator adaptors used for insertion.
5. A table of useful iterator functions.

## What is meant by an STL *range*?

One of the first ideas a programmer needs to grasp when dealing with the STL is the idea of an STL *range*, as given by a pair of iterators.

Probably the best analogy for an STL range is the notion of a half-open interval of integer values in mathematics, for which the notation is `[a,b)`, where the relationship of `a` to `b` is understood to be `a < b`. This notation means the collection of all values from `a` to `b`, including `a` but *not* including `b`. Similar notation may have any of the four possible combinations of round and square brackets, and the meaning is that the value adjacent to a square bracket *is* in the set, but the value adjacent to a round bracket is *not* in the set. In all cases, all values strictly between the two values `a` and `b` *are* in the set.

In the context of the STL we use a similar notation to mean a similar thing. For example, we might use `[iterBegin,iterEnd)` to refer to the collection of all iterator values from `iterBegin` to `iterEnd`, including `iterBegin` but not including `iterEnd`. This is a rocky shoal upon which many an STL programmer has foundered (particularly beginners).

## The five iterator types

Any given iterator is a class object, and there are (conceptually) five types of iterators, as shown in the table below. These iterators are listed in order from "most powerful" (random access iterators) to "least powerful" (output iterators).

| Iterator Type | Behavioral Description | Operations Supported |
|---|---|---|
| random access<br>(most powerful) | Store and retrieve values<br>Move forward and backward<br>Access values randomly | `* = ++ -> == != --`<br>`+ - [] < > <= >= += -=` |
| bidirectional | Store and retrieve values<br>Move forward and backward | `* = ++ -> == != --` |
| forward | Store and retrieve values<br>Move forward only | `* = ++ -> == !=` |
| input | Retrieve but not store values<br>Move forward only | `* = ++ -> == !=` |
| output<br>(least powerful) | Store but not retrieve values<br>Move forward only | `* = ++` |

Some notes on the above table:

- In the iterator descriptions we could just as easily have used the terms "read" and "write" where we used "retrieve" and "store". The "reading and writing" terminology is particularly appropriate if the iterator in question is being used on an "input stream" or "output stream", either of which may often be regarded as a "container" in the STL sense.
- Element access for reading and writing is performed via the `*` dereference operator, in a manner precisely analogous to the way in which this operator is employed in the "usual" pointer context.
- In general, if a certain kind of iterator is required for a certain kind of access in a given context, a more powerful iterator can also be used for the same kind of access in the same context.

## Container classes and their associated iterator types

The table below shows, first of all, that the container adaptors have *no* iterators. Each of the other classes (the three sequential containers and the four associative containers) has a default iterator of the type shown. Any such iterator is obtained by declaring an object in a manner analogous to the declaration

```
vector<int>::iterator iter;
```

which supplies an iterator `iter` capable of pointing at the elements in a vector container of values of type `int`. Each of the seven container classes that have iterators has a typedef name called `iterator` which is an alias for the class (i.e., the type) of iterator that "goes with" that particular container class of values. There is more to the story, of course, but that gives the general idea.

| **Container Class** | **Iterator Type** | **Container Category** |
|---|---|---|
| vector | random access | sequential |
| deque | random access | |
| list | bidirectional | |
| set | bidirectional | associative |
| multiset | bidirectional | |
| map | bidirectional | |
| multimap | bidirectional | |
| stack | none | adaptor |
| queue | none | |
| priority_queue | none | |

## Iterator adaptors for streams (stream iterators)

Sometimes a class will have the functionality you seek but not the right interface for accessing that functionality. For example, the STL `copy()` algorithm requires a pair of input iterators as its first two parameters to give the range of values to be copied. An istream object can act as a source of such data values but it does not have any iterators that the copy algorithm can use.

Similarly the third parameter of the `copy()` algorithm is an output iterator that directs the copied values to their proper destination. That destination could be an output stream but output streams do not directly provide any output iterators.

An *adaptor class* is one that acts like a "translator" by "adapting" the messages you want to send to produce messages that the other class object wants to receive.

There is an iterator adaptor class called `istream_iterator` that provides the interface that the `copy()` algorithm expects for input and translates requests from this algorithm into appropriate `istream` operations. And there is another one called `ostream_iterator` that provides the interface that the `copy()` algorithm expects for output and translates requests from the algorithm into appropriate `ostream` operations.

| Stream | Iterator Adaptor |
|---|---|
| istream | istream_iterator<br>istreambuf_iterator |
| ostream | ostream_iterator<br>ostreambuf_iterator |

**stream_iterators.cpp** | **Windows_executable** | **program_output_(text)**

Illustrates the use of both input and output stream iterators with standard input and output and files, and uses the STL `copy()` and `transform()` algorithms.

## Iterator adaptors for insertion (insert iterators)

Inserters (also called "insert iterators") are "iterator adaptors" that permit algorithms (the `copy` algorithm, for example) to operate in insert mode rather than overwrite mode, which is the default. Thus they solve the problem that crops up when an algorithm tries to write elements to a destination container not already big enough to hold them, by making the destination grow as needed. There are three kinds of inserters, as shown in the table below:

1. The back_inserter(), which can be used if the recipient container supports the push_back() member function.
2. The front_inserter(), which can be used if the recipient container supports the push_front() member function.
3. The inserter(), which can be used if the recipient container supports the insert() member function.

| |
|---|
| `back_inserter(container_supporting_push_back)`<br>Used to permit an algorithm to operate in "insert mode" at the "back" of a container that supports the push_back() member function. |
| `front_inserter(container_supporting_push_front)`<br>Used to permit an algorithm to operate in "insert mode" at the "front" of a container that supports the push_front() member function. |
| `inserter(container_supporting_insert, insert_start_location)`<br>Used to permit an algorithm to operate in "insert mode" at any "interior" point of a container that supports the insert() member function. |

**back_inserters.cpp** | **Windows_executable** | **program_output (text)**

Illustrates the use of a back_inserter() to insert several values into an empty container.

**front_inserters.cpp** | **Windows_executable** | **program_output (text)**

Illustrates the use of front_inserter() to insert several values into an empty container.

**general_inserters.cpp** | **Windows_executable** | **program_output (text)**

Illustrates the use of a general inserter() to insert several values into the interior of a container.

## Useful iterator functions

Among other things, the functions shown in the table below help us to overcome the difficulties caused by the fact that

- Only random-access iterators permit a integer value to be added to or subtracted from an iterator. The advance() function can be used to achieve the same effect when the iterator is not of the the random access variety, and can also be used with random access iterators.
- Only random access iterators permit one iterator to be subtracted from another. The distance() function can be used to achieve the same effect for non-random-access iterators, and can also be used with random access iterators.

| |
|---|
| `advance(inIter, num)`<br>Moves inIter \|num\| positions forward if num > 0 and backward if num < 0. |
| `distance(inIterBegin, inIterEnd)`<br>Returns the number of values (or positions) in the range `[inIterBegin, inIterEnd)`. Note, however, that the return value must be cast to an `int` (or `unsigned int`), since its type is actually more complicated. |

**advance.cpp** | **Windows_executable** | **program_output (text)**

Illustrates the use of the advance() function to move an iterator a given number of positions.

**distance.cpp** | **Windows_executable** | **program_output (text)**

Illustrates the use of the distance() function to compute the number of positions between two iterators.