C++ Iterators

Iterators are just like pointers used to access the container elements.

Important Points:

- Iterators are used to traverse from one element to another element, a process is known as **iterating through the container**.
- The main advantage of an iterator is to provide a common interface for all the containers type.
- Iterators make the **algorithm independent** of the type of the container used.
- Iterators provide a generic approach to navigate through the elements of a container.

Syntax

```
<ContainerType> :: iterator;
<ContainerType> :: const_iterator;
```

Operations Performed on the Iterators:

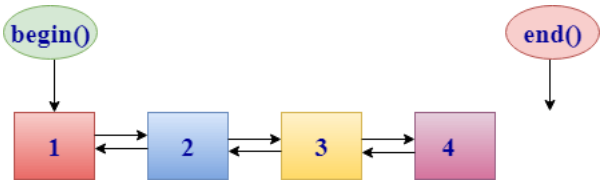
- **Operator (*) :** The '*' operator returns the element of the current position pointed by the iterator.
- **Operator (++) :** The '++' operator increments the iterator by one. Therefore, an iterator points to the next element of the container.
- **Operator (==) and Operator (!=) :** Both these operators determine whether the two iterators point to the same position or not.
- **Operator (=) :** The '=' operator assigns the iterator.

Difference b/w Iterators & Pointers

Iterators can be smart pointers which allow to iterate over the complex data structures. A Container provides its iterator type. Therefore, we can say that the iterators have the common interface with different container type.

The container classes provide two basic member functions that allow to iterate or move through the elements of a container:

- **begin()**: The begin() function returns an iterator pointing to the first element of the container.
- **end()**: The end() function returns an iterator pointing to the past-the-last element of the container.



Let's see a simple example:

```
#include <iostream>
#include<iterator>
#include<vector>
using namespace std;
int main()
{
```

```
  std::vector<int> v{1,2,3,4,5};
  vector<int>::iterator itr;
  for(itr=v.begin();itr!=v.end();itr++)
  {
     std::cout << *itr <<" ";
  }
   return 0;
}
```

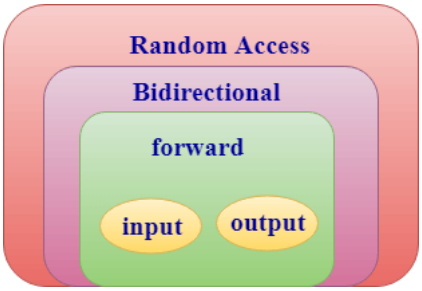
Output:

```
1 2 3 4 5
```

Iterator Categories

An iterator can be categorized in the following ways:

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator



Input Iterator: An input iterator is an iterator used to access the elements from the container, but it does not modify the value of a container.

Operators used for an input iterator are:

- Increment operator(++)
- Equal operator(==)
- Not equal operator(!=)
- Dereference operator(*)

Output Iterator: An output iterator is an iterator used to modify the value of a container, but it does not read the value from a container. Therefore, we can say that an output iterator is a **write-only iterator**.

Operators used for an output iterator are:

- Increment operator(++)
- Assignment operator(=)

Forward Iterator: A forward iterator is an iterator used to read and write to a container. It is a multi-pass iterator.

Operators used for a Forward iterator are:

- Increment operator(++)
- Assignment operator(=)
- Equal operator(=)

- Not equal operator(!=)

Bidirectional iterator: A bidirectional iterator is an iterator supports all the features of a forward iterator plus it adds one more feature, i.e., decrement operator(--). We can move backward by decrementing an iterator.

Operators used for a Bidirectional iterator are:

- Increment operator(++)
- Assignment operator(=)
- Equal operator(=)
- Not equal operator(!=)
- Decrement operator(--)

Random Access Iterator: A Random Access iterator is an iterator provides random access of an element at an arbitrary location. It has all the features of a bidirectional iterator plus it adds one more feature, i.e., pointer addition and pointer subtraction to provide random access to an element.

Providers Of Iterators

Iterator categories	Provider
Input iterator	istream
Output iterator	ostream
Forward iterator	
Bidirectional iterator	List, set, multiset, map, multimap
Random access iterator	Vector, deque, array

Iterators and their Characteristics

Iterator	Access method	Direction of movement	I/O capability
Input	Linear	Forward only	Read-only
Output	Linear	Forward only	Write-only
Forward	Linear	Forward only	Read/Write
Bidirectional	Linear	Forward & backward	Read/Write
Random	Random	Forward & backward	Read/Write

Disadvantages of iterator

- If we want to move from one data structure to another at the same time, iterators won't work.
- If we want to update the structure which is being iterated, an iterator won?t allow us to do because of the way it stores the position.
- If we want to backtrack while processing through a list, the iterator will not work in this case.

Advantages of iterator

Following are the advantages of an iterator:

- **Ease in programming:** It is convenient to use iterators rather than using a subscript operator[] to access the elements of a container. If we use subscript operator[] to access the elements, then we need to keep the track of the number of elements added at the runtime, but this would not happen in the case of an iterator.

Let's see a simple example:

```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()
{
   vector<int> v{1,2,3,4,5};
   vector<int>::iterator itr;
   for(int i=0;i<5;i++)          // Traversal without using an iterator.
   {
      cout<<v[i]<<" ";
   }
   cout<<'\n';
   for(itr=v.begin();itr!=v.end();itr++)  // Traversal by using an iterator.
   {
      cout<<*itr<<" ";
   }
   v.push_back(10);
   cout<<'\n';
    for(int i=0;i<6;i++)
   {
      cout<<v[i]<<" ";
   }
   cout<<'\n';
    for(itr=v.begin();itr!=v.end();itr++)
   {
      cout<<*itr<<" ";
   }
    return 0;
}
```

Output:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 10
1 2 3 4 5 10
```

In the above example, we observe that if we traverse the elements of a vector without using an iterator, then we need to keep track of the number of elements added in the container.

- **Code Reusability:** A code can be reused if we use iterators. In the above example, if we replace vector with the list, and then the subscript operator[] would not work to access the elements as the list does not support the random access. However, we use iterators to access the elements, then we can also access the list elements.
- **Dynamic Processing:** C++ iterators provide the facility to add or delete the data dynamically.

Let's see a simple example:

```
#include <iostream>
#include<vector>
```

```
#include<iterator>
using namespace std;
int main()
{
   vector<int> v{1,2,3,4,5};  // vector declaration
   vector<int>::iterator itr;
   v.insert(v.begin()+1,10);
   for(itr=v.begin();itr!=v.end();itr++)
   {
      cout<<*itr<<" ";
   }
   return 0;
}
```

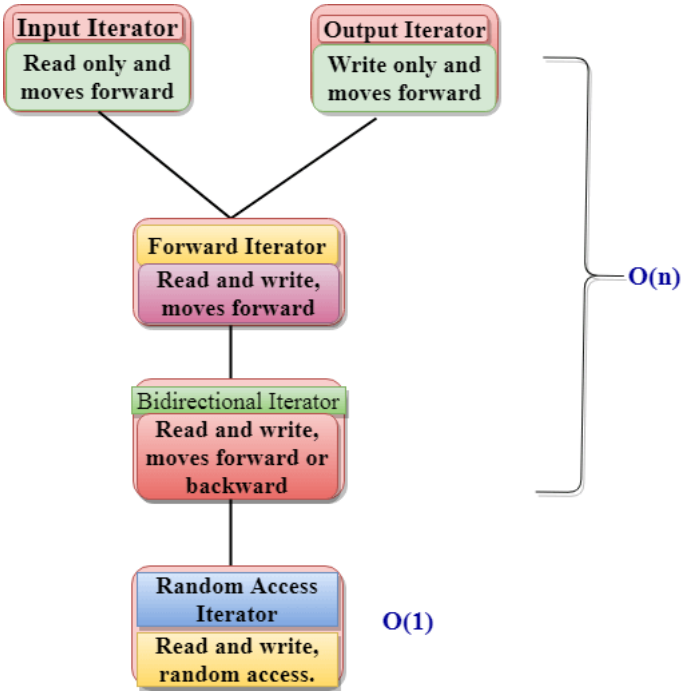
Output:

```
1 10 2 3 4 5
```

In the above example, we insert a new element at the second position by using insert() function and all other elements are shifted by one.

Difference b/w Random Access Iterator and Other Iterators

The most important difference between the Random access iterator and other iterators is that **random access iterator requires '1' step to access an element while other iterators require 'n' steps**.



← Prev Next →

C++ Bidirectional iterator

- A Bidirectional iterator supports all the features of a forward iterator, and it also supports the two **decrement operators** (prefix and postfix).
- Bidirectional iterators are the iterators used to access the elements in both the directions, i.e., **towards the end and towards the beginning**.
- A **random access iterator** is also a valid bidirectional iterator.
- Many containers implement the bidirectional iterator such as list, set, multiset, map, multimap.
- C++ provides two non-const iterators that move in both the directions are iterator and reverse iterator.
- C++ Bidirectional iterator has the same features like the forward iterator, with the only difference is that the bidirectional iterator can also be decremented.

Properties Of Bidirectional Iterator

Suppose x and y are the two iterators:

Property	Expressions
A Bidirectional iterator is a default-constructible, copy-assignable and destructible.	A x; A y(x); y=x;
It can be compared by using equality or inequality operator.	x==y x!=y
It can be dereferenced means we can retrieve the value by using a **dereference operator(*)**.	*x
A mutable iterator can be dereferenced as an lvalue.	*x = t
A Bidirectional iterator can be incremented.	x++ ++x
A Bidirectional iterator can also be decremented.	x-- --x

In the above table, **'A' is of bidirectional type, x** and **y** are the objects of an iterator type, and 't' is an object pointed by the iterator.

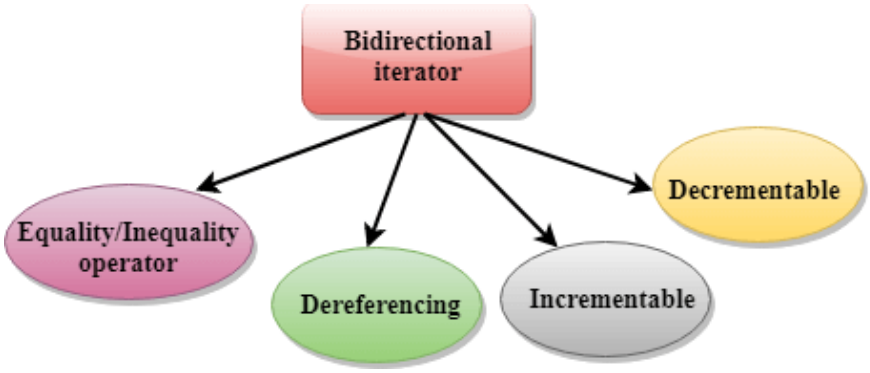
Let's see a simple example:

```
#include <iostream>
#include<iterator>
#include<vector>
using namespace std;
int main()
{
  vector<int> v{1,2,3,4,5};                  // vector declaration
  vector<int> ::iterator itr;                 // iterator declaration
  vector<int> :: reverse_iterator ritr;       // reverse iterator declaration
  for(itr = v.begin();itr!=v.end();itr++)
  {
     cout<<*itr<<" ";
  }
  cout<<'\n';
  for(ritr = v.rbegin();ritr!= v.rend();ritr++)
  {
     cout<<*ritr<<" ";
  }
   return 0;
}
```

Output:

```
1 2 3 4 5
5 4 3 2 1
```

Features of the Bidirectional iterator



- **Equality/Inequality operator**: A bidirectional iterator can be compared by using an **equality** or **inequality operator**. The two iterators are equal only when both the iterators point to the same position.

Suppose 'A' and 'B' are the two iterators:

```
A==B;
A!=B;
```

- **Dereferencing**: A bidirectional iterator can also be dereferenced both as an **lvalue** and **rvalue**.

Suppose 'A' is an iterator and 't' is an integer variable:

```
*A = t;
 t = *A
```

- **Incrementable**: A bidirectional iterator can be incremented by using an **operator++()** function.

```
A++;
++A;
```

- **Decrementable: A bidirectional iterator can also be decremented by using an Operator --() function.**

```
A--;
--A;
```

Limitations Of Bidirectional Iterator:

- **Relational operator**: An equality or inequality operator can be used with the bidirectional iterator, but the other iterators cannot be applied on the bidirectional iterator.

Suppose 'A' and 'B' are the two iterators:

```
A==B;        // valid
A<=B;        // invalid
```

- **Arithmetic operator**: An arithmetic operator cannot be used with the bidirectional iterator as it accesses the data sequentially.

```
A+2;         // invalid
A+1;         // invalid
```

- **Offset dereference operator**: A Bidirectional iterator does not support the offset dereference operator or subscript operator [] for the random access of an element.

← Prev Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Forward Iterator

- **Forward Iterator** is a combination of Bidirectional and Random Access iterator. Therefore, we can say that the forward iterator can be used to read and write to a container.
- **Forward iterators** are used to read the contents from the beginning to the end of a container.
- **Forward iterator** use only increments operator (++) to move through all the elements of a container. Therefore, we can say that the forward iterator can only move forward.
- A Forward iterator is a multi-pass iterator.

Operations Performed on the Forward Iterator:

Properties	Valid Expressions
It is default constructible.	A x;
It is a copy-constructible.	A x(y);
It is a copy-assignable.	y = x;
It can be compared either by using an equality or inequality operator.	a==b; a!=b;
It can be incremented.	a++; ++a;
It can be dereferenced as an rvalue.	*a;
It can also be dereferenced as an lvalue.	*a = t;

Where 'A' is a forward iterator type, and x and y are the objects of a forward iterator type, and t is an object pointed by the iterator type object.

Let's see a simple example:

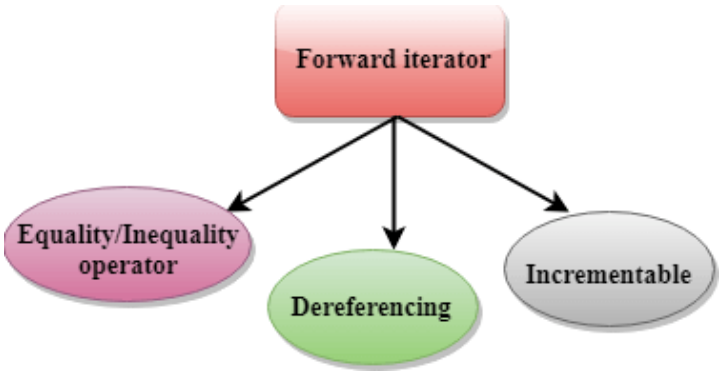
```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
```

```
using namespace std;

template<class ForwardIterator>                          // function template
void display(ForwardIterator first, ForwardIterator last)          // display function
{
 while(first!=last)
 {
    cout<<*first<<" ";
    first++;
 }
}

int main()
{

  vector<int> a;                    // declaration of vector.
  for(int i=1;i<=10;i++)
  {
     a.push_back(i);
  }
  display(a.begin(),a.end());              // calling display() function.

  return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Features of the Forward Iterator:



- **Equality/Inequality operator**: A forward iterator can be compared by using equality or an inequality operator.

Suppose 'A' and 'B' are the two iterators:

```
A==B;        // equality operator
A!=B;        // inequality operator
```

- **Dereferencing**: We can dereference the forward iterator as an rvalue as well as an lvalue. Therefore, we can access the output iterator and can also assign the value to the output iterator.

Suppose 'A' is an iterator and 't' is an integer variable:

```
*A = t;
 t = *A;
```

- **Incrementable**: A forward iterator can be incremented but cannot be decremented.

Suppose 'A' is an iterator:

```
A++;
++A;
```

Limitations of the Forward Iterator:

- **Decrementable**: A forward iterator cannot be decremented as it moves only in the forward direction.

Suppose 'A' is an iterator:

```
A--;        // invalid
```

- **Relational Operators**: A forward iterator can be used with the equality operator, but no other relational operators can be applied on the forward iterator.

Suppose 'A' and 'B' are the two iterators:

```
A==B;      // valid
A>=B;      // invalid
```

- **Arithmetic Operators**: An arithmetic operators cannot be used with the forward iterator.

```
A+2;       // invalid
A+3;       // invalid
```

- **Random Access**: A forward iterator does not provide the random access of an element. It can only iterate through the elements of a container.

← Prev Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

C++ Input Iterator

- Input Iterator is an iterator used to read the values from the container.
- Dereferencing an input iterator allows us to retrieve the value from the container.
- It does not alter the value of a container.
- It is a one-way iterator.
- It can be incremented, but cannot be decremented.
- Operators which can be used for an input iterator are increment operator(++), decrement operator(--), dereference operator(*), not equal operator(!=) and equal operator(==).
- An input Iterator is produced by the **Istream**.
- A Forward iterator, bidirectional iterator, and random access iterator are all valid input iterators.

Property	Valid Expressions
An input iterator is a copy-constructible, copy-assignable and destructible.	X b(a); b= a;
It can be compared by using a equality or inequality operator.	a==b; a!=b;
It can be dereferenced.	*a;
It can be incremented.	++a;

Where 'X' is of input iterator type while 'a' and 'b' are the objects of an iterator type.

Features of Input iterator:

- **Equality/Inequality operator**: An input iterator can be compared by using an equality or inequality operator. The two iterators are equal only when both the iterators point to the same location otherwise not. Suppose 'A' and 'B' are the two iterators:


```
A ==B;   // equality operator
```

```
A!=B;   // inequality operator
```

Let's see a simple example:

```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()
{
   vector<int> v{1,2,3,4,5};
   vector<int>::iterator itr,itr1;
   itr=v.begin();
   itr1=v.begin()+1;
   if(itr==itr1)
   std::cout << "Both the iterators are equal" << std::endl;
   if(itr!=itr1)
   std::cout << "Both the iterators are not equal" << std::endl;
   return 0;
}
```

Output:

```
Both the iterators are not equal
```

In the above example, itr and itr1 are the two iterators. Both these iterators are of vector type. The 'itr' is an iterator object pointing to the first position of the vector and 'itr1' is an iterator object pointing to the second position of the vector. Therefore, both the iterators point to the same location, so the condition itr1!=itr returns true value and prints "**Both the iterators are not equal**".

- **Dereferencing an iterator**: We can dereference an iterator by using a dereference operator(*). Suppose 'A' is an iterator:

```
*A    //  Dereferencing 'A' iterator by using *.
```

Let's see a simple example:

```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()

  vector<int> v{11,22,33,44};
  vector<int>::iterator it;
  it = v.begin();
cout<<*it;
return 0;
```

Output:

```
11
```

In the above example, 'it' is an iterator object pointing to the first element of a vector 'v'. A dereferencing an iterator *it returns the value pointed by the iterator 'it'.

- **Swappable**: The two iterators pointing two different locations can be swapped.

Let's see a simple example:

```
#include <iostream>
#include<vector>
#include<iterator>
using namespace std;
int main()
{
   vector<int> v{11,22,33,44};
   vector<int>::iterator it,it1,temp;
   it = v.begin();
   it1 = v.begin()+1;
   temp=it;
   it=it1;
   it1=temp;
   cout<<*it<<" ";
```

```
    cout<<*it1;
    return 0;
}
```

Output:

```
22 11
```

In the above example, 'it' and 'it1' iterators are swapped by using an object of a third iterator, i.e., **temp**.

← Prev Next →

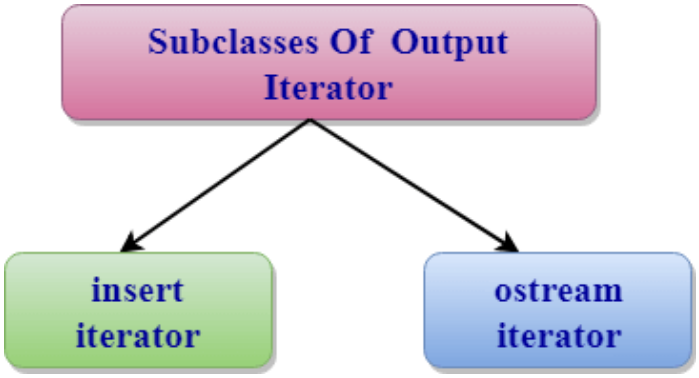
Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

C++ Output Iterator

- Output Iterator is an iterator used to modify the value in the container.
- Dereferencing an output iterator allows us to alter the value of the container.
- It does not allow us to read the value from the container.
- It is a one-way and write-only iterator.
- It can be incremented, but cannot be decremented.
- Operators that can be used for an output iterator **are increment operator(++), decrement operator(--) and assignment operator(=)**.
- **There are two main subclasses of an Output Iterator are:**
  - **insert iterator**
  - **ostream iterator**



Insert Iterator

- An insert iterator is an iterator used to insert the element in a specified position.
- An assignment operator on the insert_iterator inserts the new element at the current position.

Syntax

```
template<class Container, class Iterator>
insert_iterator<container> inserter(Container &x,Iterator it);
```

Parameters

x: It is the container on which the new element is to be inserted.

it: It is an iterator object pointing to the position which is to be modified.

Let's see a simple example:

```
#include <iostream>     // std::cout
#include <iterator>     // std::front_inserter
#include <vector>       // std::list
#include <algorithm>    // std::copy
using namespace std;
int main () {
  vector<int> v1,v2;
  for (int i=1; i<=5; i++)
  {
  v1.push_back(i);
  v2.push_back(i+2);
  }
 vector<int>::iterator it = v1.begin();
  advance (it,3);
 copy (v2.begin(),v2.end(),inserter(v1,it));
  cout<<"Elements of v1 are :";
  for (  it = v1.begin(); it!= v1.end(); ++it )
  cout << ' ' << *it;
  cout << '\n';
  return 0;
}
```

Output:

```
Elements of v1 are : 1 2 3 3 4 5 6 7 4 5
```

In the above example, insert_iterator is applied on the copy algorithm to insert the elements of the vector v2 into the vector v1 at a specified position pointed by it.

Ostream iterator

- An ostream iterators are the output iterators used to write to the output stream such as cout successively.
- An ostream iterator is created using a basic_ostream object.
- When an assigenment operator is used on the ostream iterator, it inserts a new element into the output stream.

Syntax

```
template<class T, class charT=char, class traits=char_traits<charT>>
class ostream_iterator;
```

Member functions of Ostream Iterator class

```
Ostream_iterator<T, charT, traits>& operator=(const T& value);
Ostream_iterator<T, charT, traits>& operator*();
Ostream_iterator<T, charT, traits>& operator++();
Ostream_iterator<T, charT, traits>& operator++(int);
```

Parameters

- **T**: It is the type of elements to be inserted into the container.
- **charT**: The type of elements that ostream can handle, for example, char ostream.
- **traits**: These are the character traits that the stream handles for the elements.

Let's see a simple example:

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

```
int main()
{
  vector<int> v;
  for(int i=1;i<=5;i++)
  {
     v.push_back(i*10);
  }
 ostream_iterator<int> out(cout,",");
 copy(v.begin(),v.end(),out);
   return 0;
}
```

Output:

```
10,20,30,40,50
```

In the above example, out is an object of the ostream_iterator used to add the delimiter ',' between the vector elements.

Let's see another simple example of ostream iterator:

```
#include <iostream>
#include<iterator>
#include<vector>
#include<algorithm>
using namespace std;
int main()
{
  ostream_iterator<int> out(cout,",");
  *out = 5;
  out++;
  *out = 10;
  out++;
  *out = 15;
  return 0;
}
```

Output:

```
5,10,15,
```

Features Of Output Iterator

- **Equality/Inequality Operator**: Output iterators cannot be compared either by using equality or inequality operator. Suppose X and Y are the two iterators:

```
X==Y;  invalid
X!=Y;   invalid
```

- **Dereferencing**: An output iterator can be dereferenced as an lvalue.

```
*X=7;
```

- **Incrementable**: An output iterator can be incremented by using operator++() function.

```
X++;
++X;
```

Limitations Of Output Iterator

- **Assigning but no accessing**: We can assign an output iterator as an lvalue, but we cannot access them as an rvalue.

Suppose 'A' is an output iterator type and 'x' is a integer variable:

```
*A = x;          // valid
 x = *A;         // invalid
```

- **It cannot be decremented**: We can increment the output iterator by using operator++() function, but we cannot decrement the output iterator.

Suppose 'A' is an output iterator type:

```
A++;        // not valid
++A;        // not valid
```

- **Multi-pass algorithm**: An output iterator cannot be used as a multi-pass algorithm. Since an output iterator is unidirectional and can move only forward. Therefore, it cannot be used to move through the container multiple times

- **Relational Operators**: An output iterator cannot be compared by using any of the relational operators.

Suppose 'A' and 'B' are the two iterators:

```
A = =B;     // not valid
A = =B;     // not valid
```

- **Arithmetic Operators**: An output iterator cannot be used with the arithmetic operators. Therefore, we can say that the output iterator only moves forward in a sequential manner.

Suppose 'A' is an output iterator:

```
A + 2;      // invalid
A + 5;      // invalid
```

← Prev Next →