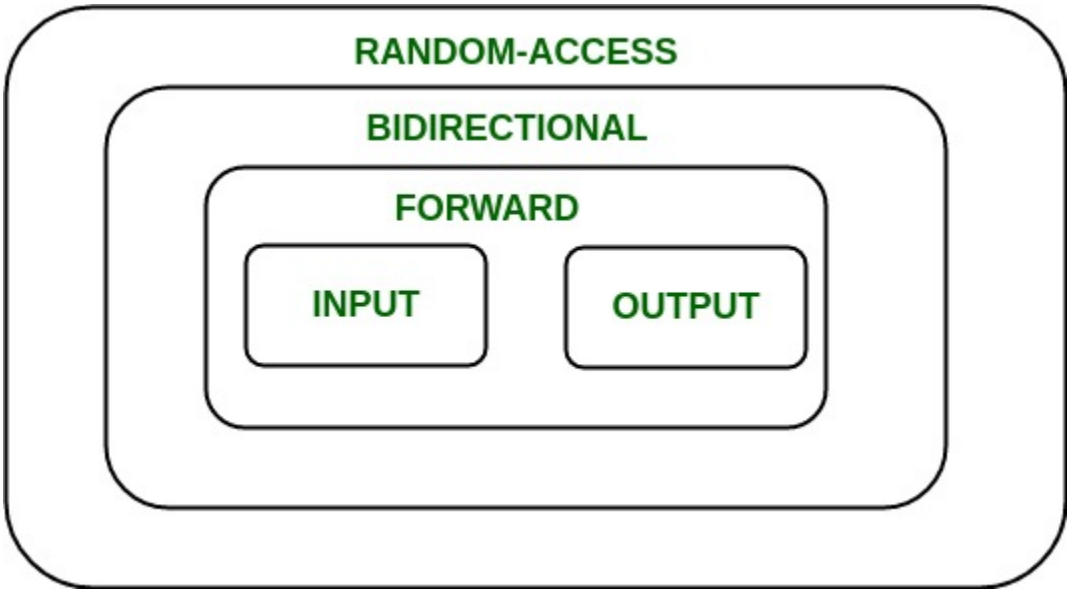
Introduction to Iterators in C++

An **iterator** is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access content at that particular location using them.
Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of iterator is a pointer. A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers.

Depending upon the functionality of iterators they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality.



Now each one of these iterators are not supported by all the containers in STL, different containers support different iterators, like vectors support Random-access iterators, while lists support bidirectional iterators. The whole list is as given below:

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

Types of iterators: Based upon the functionality of the iterators, they can be classified into five major categories:

1. **Input Iterators**: They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially such that no element is accessed more than once.
2. **Output Iterators**: Just like input iterators, they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.
3. **Forward Iterator**: They are higher in hierarachy than input and output iterators, and contain all the features present in these two iterators. But, as the name suggests, they also can only move in forward direction and that too one step at a time.

4. **Bidirectional Iterators**: They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both the directions, that is why their name is bidirectional.
5. **Random-Access Iterators**: They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality is same as pointers.

The following diagram shows the difference in their functionality with respect to various operations that they can perform.

ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

Benefits of Iterators

There are certainly quite a few ways which show that iterators are extremely useful to us and encourage us to use it profoundly. Some of the benefits of using iterators are as listed below:

1. **Convenience in programming:** It is better to use iterators to iterate through the contents of containers, as if we will not use an iterator and access elements using [ ] operator, then we need to be always vary of the size of the container, whereas with iterators we can simply use member function end() and iterate through the contents without having to keep anything in mind.

```
// C++ program to demonstrate iterators

#include <iostream>
#include <vector>
using namespace std;
int main()
{
        // Declaring a vector
        vector<int> v = { 1, 2, 3 };
```

```
	// Declaring an iterator
	vector<int>::iterator i;

	int j;

	cout << "Without iterators = ";

	// Accessing the elements without using iterators
	for (j = 0; j < 3; ++j)
	{
		cout << v[j] << " ";
	}

	cout << "\nWith iterators = ";

	// Accessing the elements using iterators
	for (i = v.begin(); i != v.end(); ++i)
	{
		cout << *i << " ";
	}

	// Adding one more element to vector
	v.push_back(4);

	cout << "\nWithout iterators = ";

	// Accessing the elements without using iterators
	for (j = 0; j < 4; ++j)
	{
		cout << v[j] << " ";
	}

	cout << "\nWith iterators = ";

	// Accessing the elements using iterators
	for (i = v.begin(); i != v.end(); ++i)
	{
		cout << *i << " ";
	}

	return 0;
}
```

Output:

```
Without iterators = 1 2 3
```

```
With iterators = 1 2 3

Without iterators = 1 2 3 4

With iterators = 1 2 3 4
```

Explanation: As can be seen in the above code that without using iterators we need to keep track of the total elements in the container. In the beginning there were only three elements, but after one more element was inserted into it, accordingly the for loop also had to be amended, but using iterators, both the time the for loop remained the same. So, iterator eased our task.

2. **Code reusability:** Now consider if we make **v** a list in place of vector in the above program and if we were not using iterators to access the elements and only using [ ] operator, then in that case this way of accessing was of no use for list (as they donot support random-access iterators).

However, if we were using iterators for vectors to access the elements, then just changing the vector to list in the declaration of the iterator would have served the purpose, without doing anything else
So, iterators support reusability of code, as they can be used to access elements of any container.

3. **Dynamic processing of container:** Iterators provide us the ability to dynamically add or remove elements from the container as and when we want with ease.

```
// C++ program to demonstrate iterators

#include <iostream>
#include <vector>
using namespace std;
int main()
{
        // Declaring a vector
        vector<int> v = { 1, 2, 3 };

        // Declaring an iterator
        vector<int>::iterator i;

        int j;

        // Inserting element using iterators
        for (i = v.begin(); i != v.end(); ++i) {
                if (i == v.begin()) {
                        i = v.insert(i, 5);
                        // inserting 5 at the beginning of v
                }
        }
```

```
        // v contains 5 1 2 3

        // Deleting a element using iterators
        for (i = v.begin(); i != v.end(); ++i) {
                if (i == v.begin() + 1) {
                        i = v.erase(i);
                        // i now points to the element after the
                        // deleted element
                }
        }

        // v contains 5 2 3

        // Accessing the elements using iterators
        for (i = v.begin(); i != v.end(); ++i) {
                cout << *i << " ";
        }

        return 0;
}
```

Output:

```
5  2  3
```

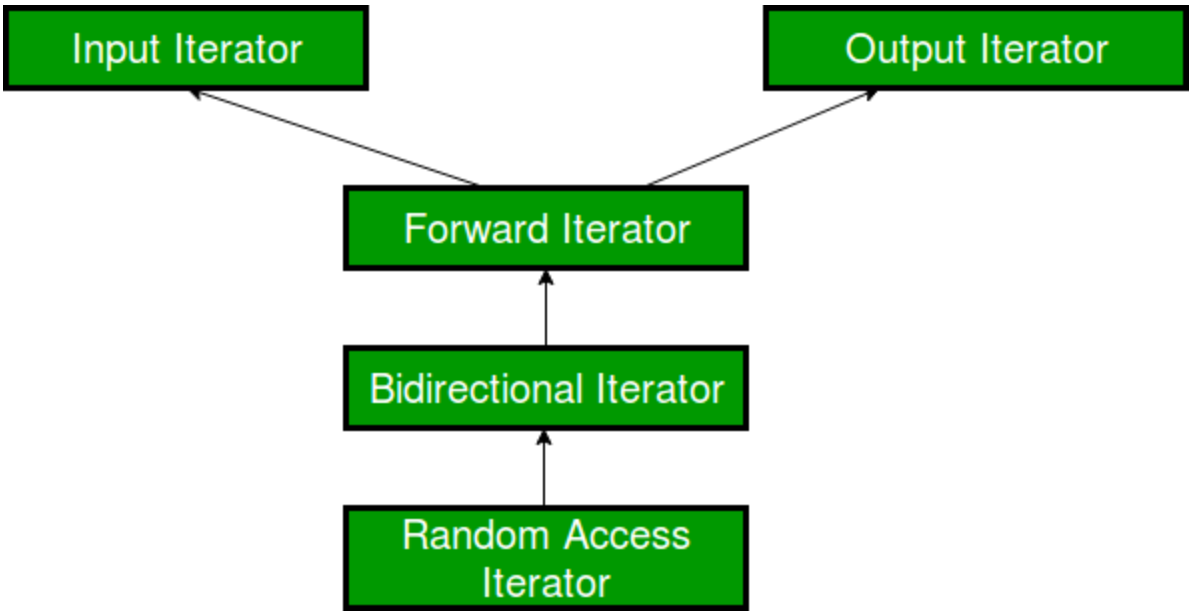
Explanation: As seen in the above code, we can easily and dynamically add and remove elements from the container using iterator, however doing the same without using them would have been very tedious as it would require shifting the elements every time before insertion and after deletion.

Input Iterators in C++

After going through the template definition of various STL algorithms like **std::find**, **std::equal**, **std::count**, you must have found their template definition consisting of objects of type **Input Iterator**. So what are they and why are they used ?

Input iterators are one of the five main types of iterators present in C++ Standard Library, others being **Output iterators**, **Forward iterator**, **Bidirectional iterator** and **Random – access iterators**.

Input iterators are considered to be the **weakest** as well as the **simplest** among all the iterators available, based upon their functionality and what can be achieved using them. They are the iterators that can be used in sequential input operations, where each value pointed by the iterator is read only once and then the iterator is incremented.



One important thing to be kept in mind is that **forward**, **bidirectional** and **random-access** iterators are also valid input iterators, as shown in the iterator hierarchy above.

1. **Usability:** Input iterators can be used only with **single-pass algorithms**, i.e., algorithms in which we can go to all the locations in the range at most once, like when we have to search or find any element in the range, we go through the locations at most once.
2. **Equality / Inequality Comparison:** An input iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not.
   So, the following two expressions are valid if A and B are input iterators:

   A == B // Checking for equality

   A != B // Checking for inequality

3. **Dereferencing:** An input iterator can be dereferenced, using the operator * and -> as an rvalue to obtain the value stored at the position being pointed to by the iterator.
   So, the following two expressions are valid if A is an input iterator:

```
*A       // Dereferencing using *
A -> m   // Accessing a member element m
```

4. **Incremental:** An input iterator can be incremented, so that it refers to the next element in sequence, using operator ++().

Note: The fact that we can use input iterators with increment operator doesn't mean that operator – -() can also be used with them. Remember, that input iterators are unidirectional and can only move in the forward direction.
So, the following two expressions are valid if A is an input iterator:

```
A++   // Using post increment operator

++A   // Using pre increment operator
```

5. **Swappable:** The value pointed to by these iterators can be exchanged or swapped.

std::find: As we know this algorithm is used to find the presence of an element inside a container. So, let us look at its internal working **std::copy:** As the name suggests, this algorithm is used to copy a range into another range. Now, as far as accessing elements are concerned, input iterators are fine, **but as soon as we have to assign elements in another container, then we cannot use these input iterators** for this purpose.

```
<strong>// Definition of std::copy()</strong>
template
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result)
{
    while (first != last)
    *result++ = *first++;
    return result;
}


<strong>// Definition of std::find()</strong>
template
InputIterator find (InputIterator first, InputIterator last,
                    const T& val)
{
    while (first!=last)
    {
        if (*first==val) return first;
        ++first;
    }
    return last;
}
```

Output Iterators in C++

Output iterators are considered to be the **exact opposite** of input iterators, as they perform the opposite function of input iterators. They can be **assigned values in a sequence**, but cannot be used to access values, unlike input iterators which do the reverse of accessing values and cannot be assigned values. So, we can say that **input and output iterators are complementary** to each other.

1. **Usability:** Just like input iterators, Output iterators can be used only with **single-pass algorithms**, i.e., algorithms in which we can go to all the locations in the range at most once, such that these locations can be dereferenced or assigned value only once.
2. **Equality / Inequality Comparison:** Unlike input iterators, output iterators cannot be compared for equality with another iterator.
   So, the following two expressions are invalid if A and B are output iterators:

```
A == B  // Invalid - Checking for equality

A != B  // Invalid - Checking for inequality
```

3. **Dereferencing:** An input iterator can be dereferenced as an rvalue, using operator * and ->, whereas an **output iterator can be dereferenced as an lvalue** to provide the location to store the value.
   So, the following two expressions are valid if A is an output iterator:

```
*A = 1      // Dereferencing using *

A -> m = 7   // Assigning a member element m
```

4. **Incrementable:** An output iterator can be incremented, so that it refers to the next element in sequence, using operator ++().
   So, the following two expressions are valid if A is an output iterator:

```
A++   // Using post increment operator

++A   // Using pre increment operator
```

5. **Swappable:** The value pointed to by these iterators can be exchanged or swapped.

std::move: As the name suggests, this algorithm is used to move elements in a range into another range. Now, as far as accessing elements are concerned, input iterators are fine, **but as soon as we have to assign elements in another container, then we cannot use these input iterators** for this purpose, that is why here using output iterators becomes a compulsion.

```
// Definition of std::move()
template
OutputIterator move (InputIterator first, InputIterator last,
                     OutputIterator result)
{
    while (first!=last)
    {
        *result = std::move(*first);
        ++result;
        ++first;
    }
    return result;
}
```

Forward Iterators in C++

Forward iterators are considered to be the **combination of input as well as output iterators**. It provides support to the functionality of both of them. It permits values to be both accessed and modified.

std::replace: As we know this algorithm is used to replace all the elements in the range which are equal to a particular value by a new value. So, let us look at its internal working (Don't go into detail just look where forward iterators can be used and where they cannot be):

```
// Definition of std::replace()
template void replace(ForwardIterator first, ForwardIterator last,
                      const T& old_value, const T& new_value)
{
    while (first != last) {
        if (*first == old_value) // L1
            *first = new_value; // L2
        ++first;
    }
}
```

Bidirectional Iterators in C++

Bidirectional iterators are iterators that can be used to access the sequence of elements in a range in **both directions** (towards the end and towards the beginning). They are similar to forward iterators, except that they can **move in the backward direction** also, unlike the forward iterators, which can move only in the forward direction.

std::reverse_copy: As the name suggests, this algorithm is used to copy a range into another range, but in reverse order. Now, as far as accessing elements and assigning elements are concerned, forward iterators are fine, **but as soon as we have to decrement the iterator, then we cannot use these forward iterators** for this purpose, and that's where bidirectional iterators come for our rescue.

```
// Definition of std::reverse_copy()
template
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result)
{
    while (first != last)
    *result++ = *--last;
    return result;
```

```
}
```

Here, we can see that we have declared last as a bidirectional iterator, as we cannot decrement a forward iterator as done in case of last, so we cannot use it in this scenario, and we have to declare it as a bidirectional iterator only.

Random-access Iterators in C++

Random-access iterators are iterators that can be used to **access elements at an arbitrary offset position** relative to the element they point to, offering the same functionality as pointers. Random-access iterators are the **most complete iterators in terms of functionality**. All pointer types are also valid random-access iterators.

1. **Usability:** Random-access iterators can be **used in multi-pass algorithms**, i.e., algorithm which involves processing the container several times in various passes.
2. **Equality / Inequality Comparison:** A Random-access iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not.
   So, the following two expressions are valid if A and B are Random-access iterators:

   ```
   A == B  // Checking for equality

   A != B  // Checking for inequality
   ```

3. **Dereferencing:** A random-access iterator can be **dereferenced both as a rvalue as well as a lvalue.**

   ```
   // C++ program to demonstrate Random-access iterator
   #include<iostream>
   #include<vector>
   using namespace std;
   int main()
   {
     vector<int>v1 = {10, 20, 30, 40, 50};

     // Declaring an iterator
     vector<int>::iterator i1;

     for (i1=v1.begin();i1!=v1.end();++i1)
     {
             // Assigning values to locations pointed by iterator
             *i1 = 7;
     }

     for (i1=v1.begin();i1!=v1.end();++i1)
   ```

```
    {
        // Accessing values at locations pointed by iterator
        cout << (*i1) << " ";
    }

    return 0;
}
```

Output:

```
7 7 7 7 7
```

So, as we can see here we can both access as well as assign value to the iterator, therefore the iterator is a random-access iterator.

4. **Incrementable:** A Random-access iterator can be incremented, so that it refers to the next element in sequence, using operator ++(), as seen in the previous code, where i1 was incremented in the for loop.
   So, the following two expressions are valid if A is a random-access iterator:

```
A++   // Using post increment operator

++A   // Using pre increment operator
```

5. **Decrementable:** Just like we can use operator ++() with Random-access iterators for incrementing them, we can also decrement them.

```
// C++ program to demonstrate Random-access iterator
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int>v1 = {1, 2, 3, 4, 5};

    // Declaring an iterator
    vector<int>::iterator i1;

    // Accessing the elements from end using decrement
    // operator
    for (i1=v1.end()-1;i1!=v1.begin()-1;--i1)
    {

        cout << (*i1) << " ";

    }
    return 0;
}
```

Output:

```
5 4 3 2 1
```

Since, we are starting from the end of the vector and then moving towards the beginning by decrementing the pointer, which shows that decrement operator can be used with such iterators.

6. **Relational Operators:** Although, Bidirectional iterators cannot be used with relational operators like , =,but random-access iterators being higher in hierarchy support all these relational operators.

```
If A and B are Random-access iterators, then


A == B     // Allowed

A <= B     // Allowed
```

7. **Arithmetic Operators:** Similar to relational operators, they also can be used with arithmetic operators like +, – and so on. This means that Random-access iterators can move in both the direction, and that too randomly.

```
If A and B are Random-access iterators, then

A + 1     // Allowed

B - 2     // Allowed
```

```
// C++ program to demonstrate Random-access iterator
#include<iostream>
#include<vector>
using namespace std;
int main()
{
        vector<int>v1 = {1, 2, 3, 4, 5};

        // Declaring first iterator
        vector<int>::iterator i1;

        // Declaring second iterator
        vector<int>::iterator i2;

        // i1 points to the beginning of the list
        i1 = v1.begin();

        // i2 points to the end of the list
        i2 = v1.end();

        // Applying relational operator to them
```

```
        if ( i1 < i2)
        {
                cout << "Yes";
        }

        // Applying arithmetic operator to them
        int count = i2 - i1;

        cout << "\ncount = " << count;
        return 0;
}
```

Output:

```
Yes

count = 5
```

Here, since i1 is pointing to beginning and i2 is pointing to end , so i2 will be greater than i1 and also difference between them will be the total distance between them.

8. **Use of offset dereference operator ([ ]):** Random-access iterators support offset dereference operator ([ ]), which is used for random-access.

```
If A is a Random-access iterator, then

A[3]    // Allowed
```

```
// C++ program to demonstrate Random-access iterator
#include<iostream>
#include<vector>
using namespace std;
int main()
{
        vector<int>v1 = {1, 2, 3, 4, 5};
        int i;

        // Accessing elements using offset dereference
        // operator [ ]
        for(i=0;i<5;++i)
        {
                cout << v1[i] << " ";
        }
        return 0;
}
```

Output

```
1 2 3 4 5
```

9. **Swappable:** The value pointed to by these iterators can be exchanged or swapped.

std::random_shuffle: As we know this algorithm is used to randomly shuffle all the elements present in a container. So, let us look at its internal working (Donot go into detail just look where random-access iterators can be used):

```
// Definition of std::random_shuffle()
template
void random_shuffle(RandomAccessIterator first,
                    RandomAccessIterator last,
                    RandomNumberGenerator& gen)
{
    iterator_traits::difference_type i, n;
    n = (last - first);
    for (i=n-1; i>0; --i)
    {
        swap (first[i],first[gen(i+1)]);
    }
}
```