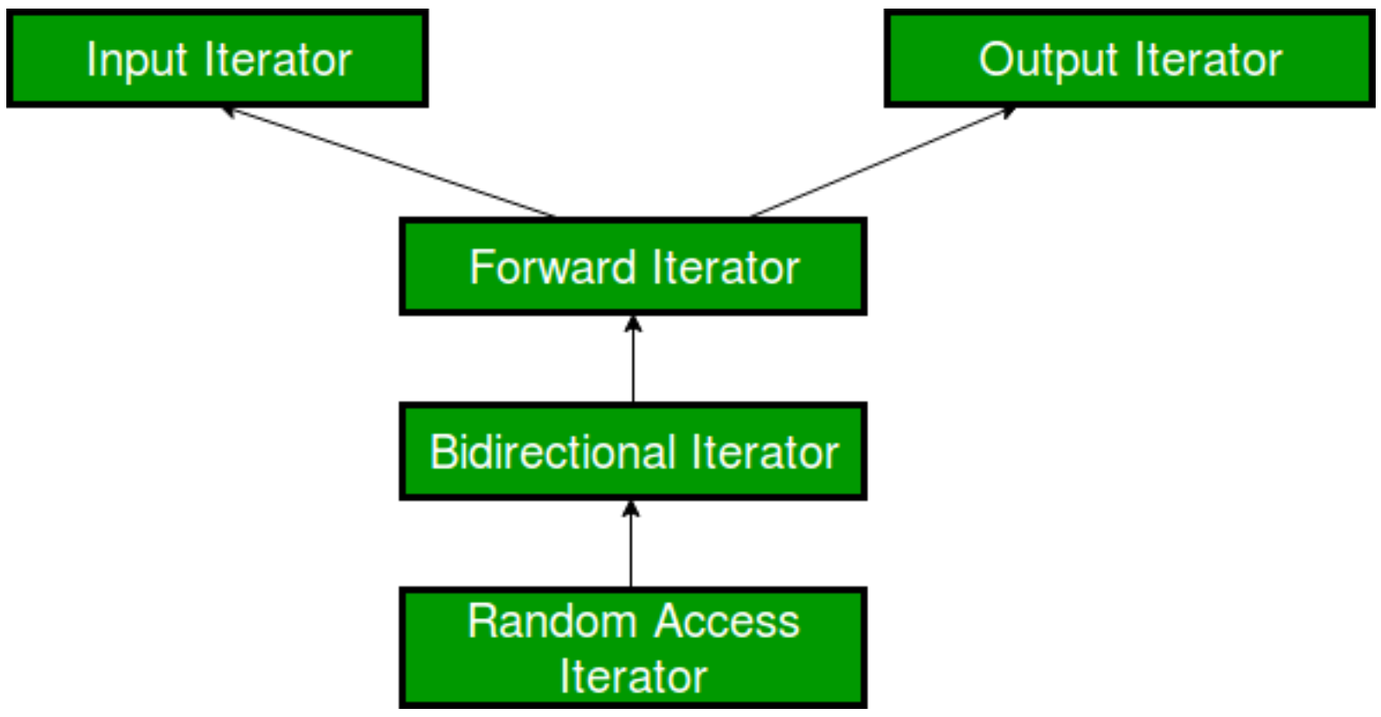
Input Iterators in C++

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.

Salient Features

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. **Incrementable:** 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.

Practical implementation

After understanding its features and deficiencies, it is very important to learn about its practical implementation as well. As told earlier, input iterators are used only when we want to access elements and not when we have to assign elements to them. The following two STL algorithms can show this fact:

- **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 (Donot go into detail just look where input iterators can be used and where they cannot be):

```
<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;
}
```

  So, this is the practical implementation of input iterators, **single-pass algorithms where only we have to move sequentially and access the elements and check for equality** with another element just like first is used above, so here they can be used. More such algorithms are **std::equal**, **std::equal_range** and **std::count.**

- **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;
}
```

  Here, since the result is the iterator to the resultant container, to which elements are assigned, so for this, we cannot use input iterators and have made use of output iterators at their place, whereas for first, which only needs to be incremented and accessed, we have used input iterator.

Limitations

After studying about the salient features, one must also know its deficiencies as well which make it the weakest iterator among all, which are mentioned in the following points:

1. **Only accessing, no assigning:** One of the biggest deficiency is that **we cannot assign any value** to the location pointed by this iterator, it can only be used to access elements and not assign elements.

```
// C++ program to demonstrate input 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;

    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        // Accessing elements using iterator
        cout << (*i1) << " ";
    }
    return 0;
}
```

Output:

```
1 2 3 4 5
```

The above is an example of accessing elements using input iterator, however, if we do something like:

```
*i1 = 7;
```

So, this is not allowed in input iterator. However, if you try this for above code, it will work, because vectors return iterators higher in hierarchy than input iterators.

That big deficiency is the reason why many algorithms like std::copy, which requires to copy a range into another container **cannot** use input iterator for the resultant container, because we can't assign values to it with such iterators and instead make use of output iterators.

2. **Cannot be decremented:** Just like we can use operator ++() with input iterators for incrementing them, we cannot decrement them.

```
If A is an input iterator, then

A--    // Not allowed with input iterators
```

3. **Use in multi-pass algorithms:** Since, it is unidirectional and can only move forward, therefore, such iterators cannot be used in multi-pass algorithms, in which we need to process the container multiple times.
4. **Relational Operators:** Although, input iterators can be used with equality operator (==), but it can not be used with other relational operators like , =.

```
If A and B are input iterators, then

A == B     // Allowed
A <= B     // Not Allowed
```

5. **Arithmetic Operators:** Similar to relational operators, they also can't be used with arithmetic operators like +, – and so on. This means that input operators can only move in one direction that too forward and that too sequentially.

```
If A and B are input iterators, then

A + 1     // Not allowed
B - 2     // Not allowed
```

So, the two above examples very well show when, where, why and how input iterators are used practically.

This article is contributed by **Mrigendra Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Recommended Posts:

Iterators in C++ STL

Bidirectional Iterators in C++

Forward Iterators in C++

Output Iterators in C++

Introduction to Iterators in C++

Random-access Iterators in C++

Basic Input / Output in C++

Clearing The Input Buffer In C/C++

Input an integer array without spaces in C

How to use getline() in C++ when there are blank lines in input?

Stitching input images (panorama) using OpenCV with C++

getchar_unlocked() - faster input in C/C++ for Competitive Programming

Check input character is alphabet, digit or special character

Conditional or Ternary Operator (?:) in C/C++

Improved By : Akanksha_Rai

Article Tags : C++ cpp-iterator STL

Practice Tags : STL CPP

2

3.5

Based on **4** vote(s)

To-do Done

Feedback/ Suggest Improvement Add Notes Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

GeeksforGeeks
A computer science portal for geeks

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org

COMPANY

About Us
Careers
Privacy Policy
Contact Us

LEARN

Algorithms
Data Structures
Languages
CS Subjects
Video Tutorials

PRACTICE

Courses
Company-wise
Topic-wise
How to begin?

CONTRIBUTE

Write an Article
Write Interview Experience
Internships
Videos