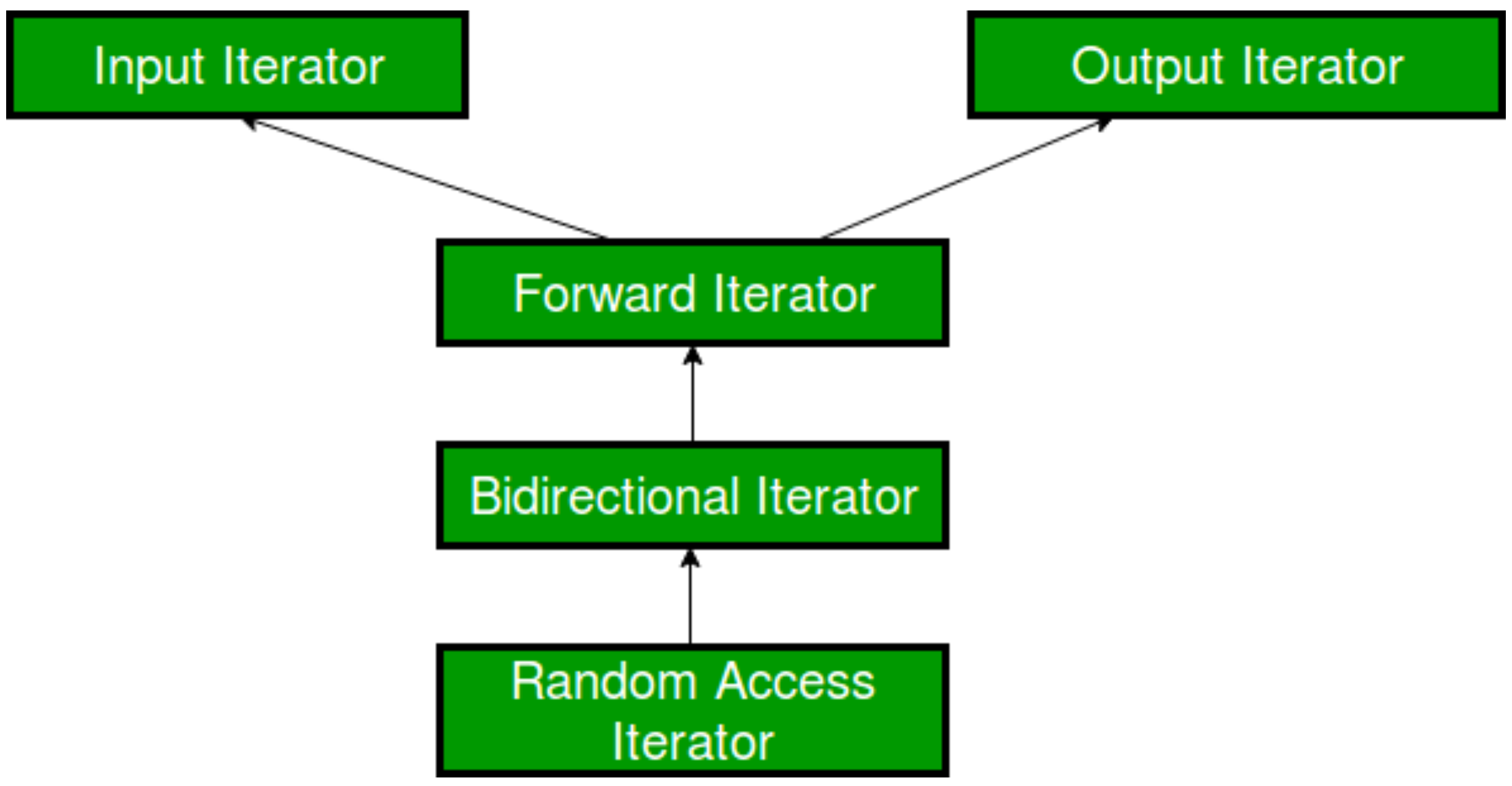
Forward Iterators in C++

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

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

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.



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

Salient Features

1. **Usability:** Performing operations on a forward iterator that is dereferenceable never makes its iterator value non-dereferenceable, as a result this enables algorithms that use this category of iterators to use multiple copies of an iterator to pass more than once by the same iterator values. So, it **can be used in multi-pass algorithms**.
2. **Equality / Inequality Comparison:** A forward 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 forward iterators:

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

3. **Dereferencing:** Because an input iterator can be dereferenced, using the operator * and -> as an rvalue and an output iterator can be dereferenced as an lvalue, so forward iterators can be used for both the purposes.

```
// C++ program to demonstrate forward 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) {
        // Assigning values to locations pointed by iterator
        *i1 = 1;
    }

    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        // Accessing values at locations pointed by iterator
        cout << (*i1) << " ";
    }

    return 0;
}
```

Output:

```
1 1 1 1 1
```

So, as we can see here we can both access as well as assign value to the iterator, therefore the iterator is at least a forward iterator(it can be higher in hierarchy also).

4. **Incrementable:** A forward iterator can be incremented, so that it refers to the next element in sequence, using operator ++().
Note: The fact that we can use forward iterators with increment operator doesn't mean that operator – -() can also be used with them. Remember, that forward iterators are unidirectional and can only move in the forward direction.

   So, the following two expressions are valid if A is a forward 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, it is very important to learn about its practical implementation as well. As told earlier, forward iterators can be used both when we want to access elements and also when we have to assign elements to them, as it is the combination of both input and output iterator. The following two STL algorithms can show this fact:

- **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;
    }
}
```

  Here, we can see that we have made use of forward iterators, as we need to make use of the feature of both input as well as output iterators. In L1, we are required to dereference the iterator first as a rvalue (input iterator) and in L2, we are required to dereference first as a lvalue (output iterator), so to accomplish both the tasks, we have made use of forward iterators.

- **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.

```
// 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, and not a forward iterator since we cannot decrement a forward iterator as done in case of last, so we cannot use it in this scenario.

Note: As we know that forward iterators are the combination of both input and output iterators, so if any algorithm involves the use of any of these two iterators, then we can use forward iterators in their place as well, without affecting the program.

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

Limitations

After studying the salient features, one must also know its deficiencies as well although there are not as many as there are in input or output iterators as it is higher in the hierarchy.

1. **Can not be decremented:** Just like we can use operator ++() with forward iterators for incrementing them, we cannot decrement them. Although it is higher in the hierarchy than input and output iterators, still it can't overcome this deficiency.
   That is why, its name is forward, which shows that **it can move only in forward direction**.

   ```
   If A is a forward iterator, then

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

2. **Relational Operators:** Although, forward iterators can be used with equality operator (==), it can not be used with other relational operators like =.

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

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

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

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

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

4. **Use of offset dereference operator ([ ]):** Forward iterators do not support offset dereference operator ([ ]), which is used for random-access.

   ```
   If A is a forward iterator, then
   A[3]    // Not allowed
   ```

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:

What are Forward declarations in C++
Implementing Forward Iterator in BST
Iterators in C++ STL
Forward List in C++ | Set 2 (Manipulating Functions)
Introduction to Iterators in C++
Bidirectional Iterators in C++
Output Iterators in C++
Input Iterators in C++
Forward List in C++ | Set 1 (Introduction and Important Functions)
Random-access Iterators in C++
fill in C++ STL
Conditional or Ternary Operator (?:) in C/C++
forward_list insert_after() function in C++ STL
Count of distinct remainders when N is divided by all the numbers from the range [1, N]

Improved By : Akanksha_Rai

Article Tags : C++ cpp-iterator STL

Practice Tags : STL CPP

2

1

☐ To-do ☐ Done

Based on **1** vote(s)

Feedback/ Suggest Improvement Add Notes Improve Article