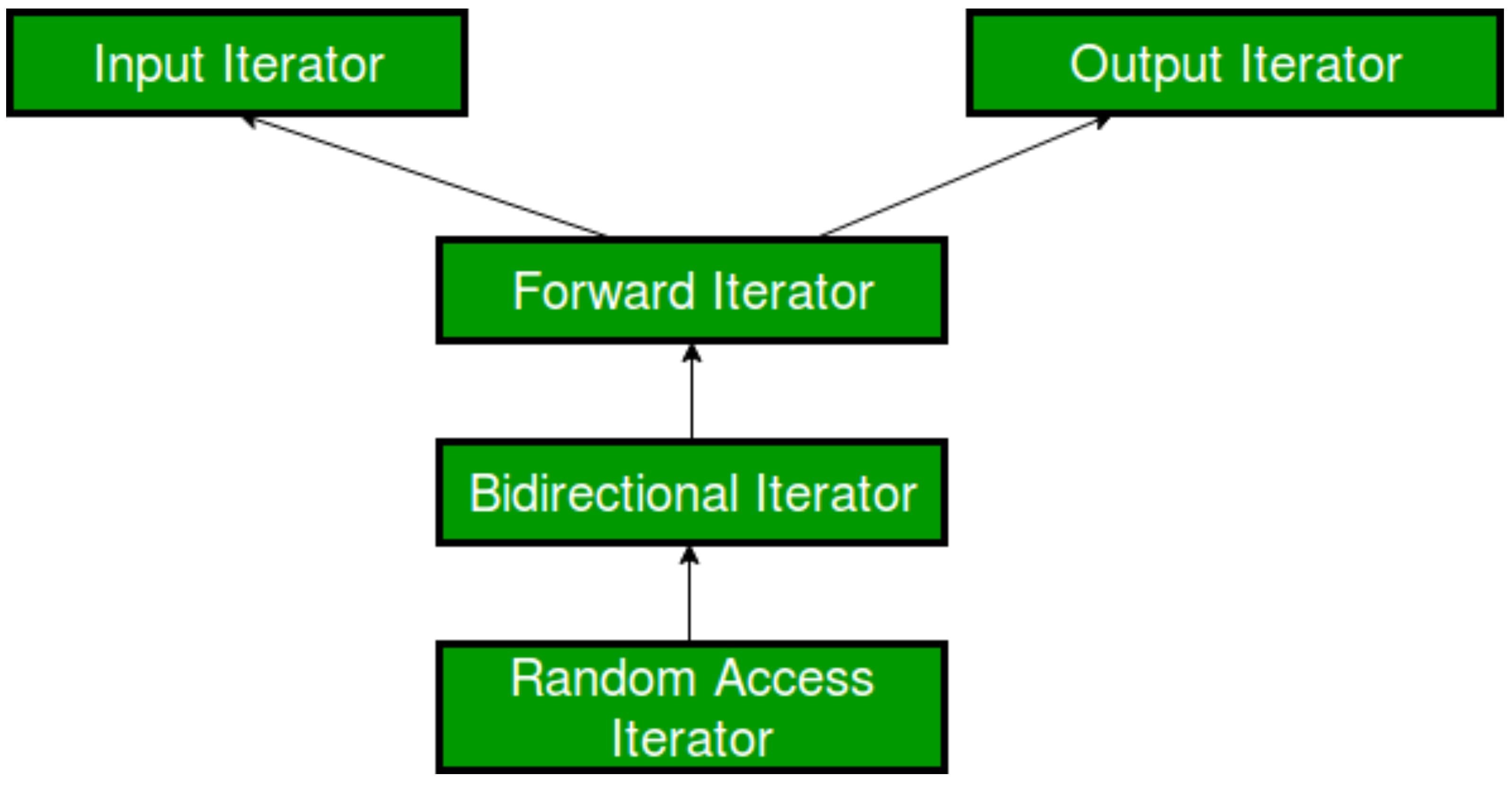
Bidirectional Iterators in C++

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

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

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.

It is to be noted that containers like **list, map, multimap, set and multiset support bidirectional iterators**. This means that if we declare normal iterators for them, and then those will be bidirectional iterators, just like in case of vectors and deque they are random-access iterators.



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

Salient Features

1. **Usability:** Since, forward iterators can be used in multi-pass algorithms, i.e., algorithm which involves processing the container several times in various passes, therefore bidirectional iterators can also be used in multi-pass algorithms.
   .
2. **Equality / Inequality Comparison:** A Bidirectional 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 Bidirectional iterators:

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

3. **Dereferencing:** Because an input iterator can be dereferenced, using operator * and -> as an rvalue and an output iterator can be dereferenced as an lvalue, so forward iterators being the combination of both can be used for both the purposes, and similarly, bidirectional operators can also serve both the purposes.

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

    // Declaring an iterator
    list<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 a bidirectional iterator.

4. **Incrementable:** A Bidirectional 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 a bidirectional iterator:

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

5. **Decrementable:** This is the feature which differentiates a Bidirectional iterator from a forward iterator. Just like we can use operator ++() with bidirectional iterators for incrementing them, we can also decrement them.
   That is why, its name is bidirectional, which shows that **it can move in both directions**.

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

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

    // Accessing the elements from end using decrement
    // operator
    for (i1=v1.end();i1!=v1.begin();--i1)
    {
        if (i1 != v1.end())
        {
            cout << (*i1) << " ";
        }
    }
    cout << (*i1);

    return 0;
}
```

Output:

```
5 4 3 2 1
```

Since, we are starting from the end of the list and then moving towards the beginning by decrementing the pointer, which shows that decrement operator can be used with such iterators. Here, we have run the loop till the iterator becomes equal to the begin(), that is why the first value is not printed inside the loop and we have printed it separately.

6. **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, Bidirectional iterators can be used for all the purposes that forward iterator can be used along within those situations where iterator needs to be decremented. The following two STL algorithms can show this fact:

- **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.

- **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 (Don't go into detail just look where bidirectional iterators can be used and where they cannot be):

  ```
  // 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)]);
      }
  }
  ```

  Here, we can see that we have made use of Random-access iterators, as although we can increment a bidirectional iterator, decrement it as well, but **we cannot apply arithmetic operator like +, –** to it and this what is required in this algorithm , where (last – first) is done, so, for this reason, we cannot use a bidirectional iterator in these scenarios.

Note: As we know that Bidirectional iterator is higher in the hierarchy than forward iterator which is itself higher than input and output iterators, therefore, all these three types of iterators can be substituted by bidirectional iterators, without affecting the working of the algorithm.

So, the two above examples very well show when, where, why and how bidirectional 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. **Relational Operators:** Although, Bidirectional iterators can be used with equality operator (==), but it can not be used with other relational operators like , =.

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

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

2. **Arithmetic Operators:** Similar to relational operators, they also can't be used with arithmetic operators like +, – and so on. This means that bidirectional iterators can move in both the direction, but sequentially.

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

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

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

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

    // Declaring second iterator
    list<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";
    }
    // This will throw an error because relational
    // operators cannot be applied to bidirectional iterators


    // Applying arithmetic operator to them
    int count = i1 - i2;
    // This will also throw an error because arithmetic
    // operators cannot be applied to bidirectional iterators
    return 0;
}
```

Output:

```
Error, because of use of arithmetic and relational operators
with bidirectional iterators
```

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

```
If A is a Bidirectional 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:

Iterators in C++ STL

Introduction to Iterators in C++

Forward Iterators in C++

Output Iterators in C++

Input Iterators in C++

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]

Random-Scan Dispaly

Raster-Scan Displays

C++ | asm declaration

Pointers and References in C++

Improved By : Akanksha_Rai

Article Tags : C++ Misc cpp-iterator STL

Practice Tags : Misc Misc STL CPP

2

2

☐ To-do ☐ Done

Based on **1** vote(s)

Feedback/ Suggest Improvement Add Notes Improve Article