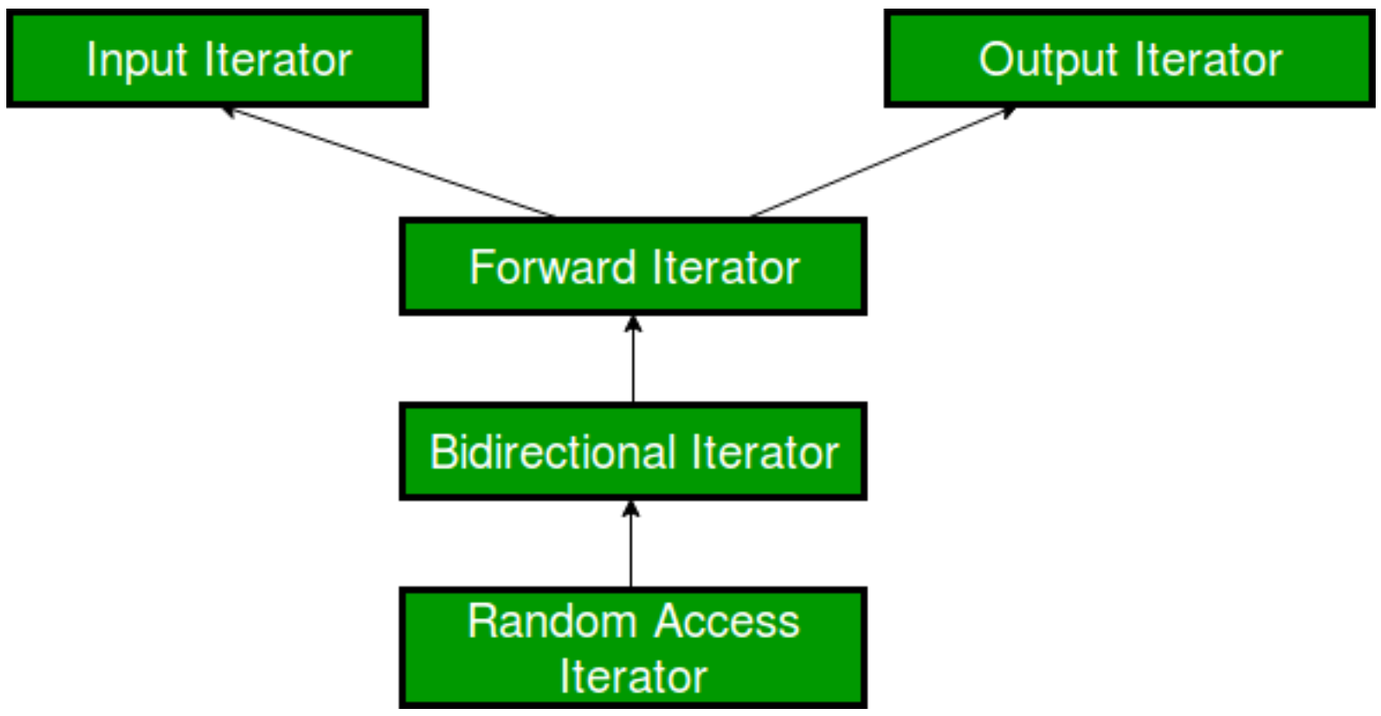
Random-access Iterators in C++

After going through the template definition of various STL algorithms like std::nth_element, std::sort, you must have found their template definition consisting of objects of type **Random-access Iterator**. So what are they and why are they used ?

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

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.

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



So, from the above hierarchy it can be said that random-access iterators are the strongest of all iterator types.

Salient Features

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.**

```cpp
// 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.

```cpp
// 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.

Practical implementation

After understanding its features, it is very important to learn about its practical implementation as well. As told earlier, Random-access iterators can be used for all purposes and in every scenario. The following STL algorithm shows one of its practical implementations:

- **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)]);
        }
}
```

Here, we can see that we have made use of Random-access iterators, as there is no other type of iterator which supports arithmetic operators, that is why we have used it.

Infact if you are thinking of uses of random-access iterators, then you can **use random-access iterator in place of any other type of iterator**, since it is the strongest and the best type of iterator available in C++ Standard library.

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++

Output Iterators in C++

Forward Iterators in C++

Input Iterators in C++

Introduction to 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]

C++ | asm declaration

Pointers and References in C++

Strings in C++ and How to Create them?

Enum Classes in C++ and Their Advantage over Enum DataType

Article Tags : C++ cpp-iterator STL

Practice Tags : STL CPP

2

1.5

Based on **2** 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