C++ Data Structures
Prerequisites

CHAPTER 2: C++ ITERATORS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Overview of Iterators.
- Iterators for STL Containers.
- Address calculation
- Iterator class and object

LECTURE 1

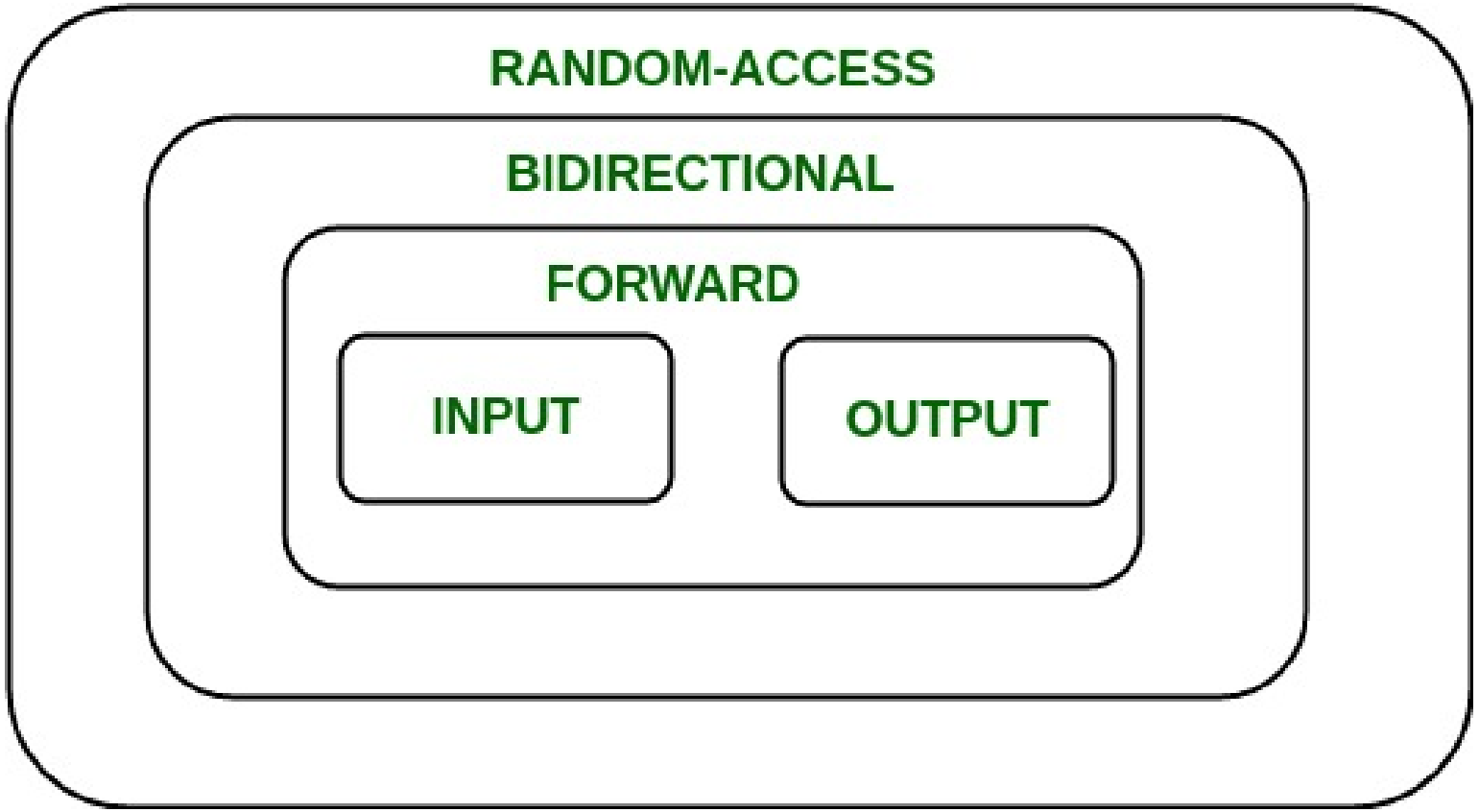
Overview

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 visualized as something similar to a pointer pointing to some location and we can access the 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 an 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.

Introduction to Iterators in C++

- 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.


RANDOM-ACCESS
BIDIRECTIONAL
FORWARD
INPUT
OUTPUT

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

Iterators

- 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:

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 the 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 a forward direction and that too one step at a time.

Types of iterators

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

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 are same as pointers.

Types of iterators

- 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=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

LECTURE 2

Benefits of Iterators

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
    2. Code reusability
    3. Dynamic processing of the container

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 worried about 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.

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main(){
    vector<int> v = { 1, 2, 3 };
    vector<int>::iterator i;
    int j;
    cout << "Without iterators = ";
    for (j = 0; j < 3; ++j){
        cout << v[j] << " ";
    }
    cout << "\nWith iterators = ";
    for (i = v.begin(); i != v.end(); ++i){
        cout << *i << " ";
    }
    v.push_back(4);
    cout << "\nWithout iterators = ";
    for (j = 0; j < 4; ++j){
        cout << v[j] << " ";
    }
    cout << "\nWith iterators = ";
    for (i = v.begin(); i != v.end(); ++i){
        cout << *i << " ";
    }
    return 0;
}
```

iterator1.cpp

```
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.

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 don't 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.

Dynamic processing of the container

- Iterators provide us the ability to dynamically add or remove elements from the container as and when we want with ease.

iterator2.cpp

```
5 2 3
```

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main(){
    vector<int> v = { 1, 2, 3 };
    vector<int>::iterator i;
    for (i = v.begin(); i != v.end(); ++i) {
        if (i == v.begin()) {
            i = v.insert(i, 5);
        }
    }
    for (i = v.begin(); i != v.end(); ++i) {
        if (i == v.begin() + 1) {
            i = v.erase(i);
        }
    }
    for (i = v.begin(); i != v.end(); ++i) {
        cout << *i << " ";
    }
    return 0;
}
```

Explanation

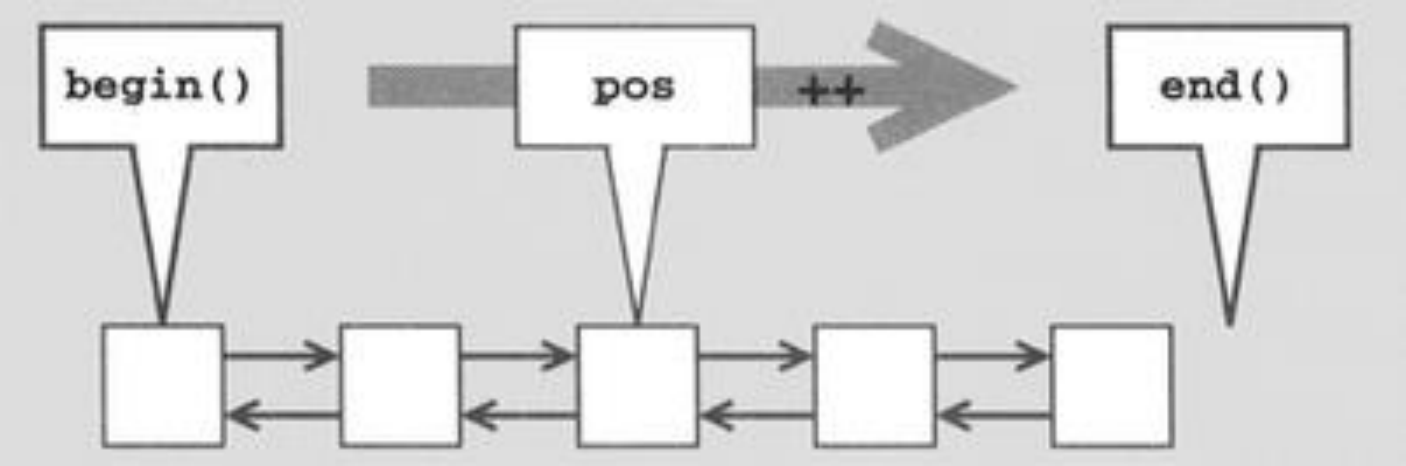
- As seen in the above code, we can easily and dynamically add and remove elements from the container using iterator, however, doing the same without using them would have been very tedious as it would require shifting the elements every time before insertion and after deletion.

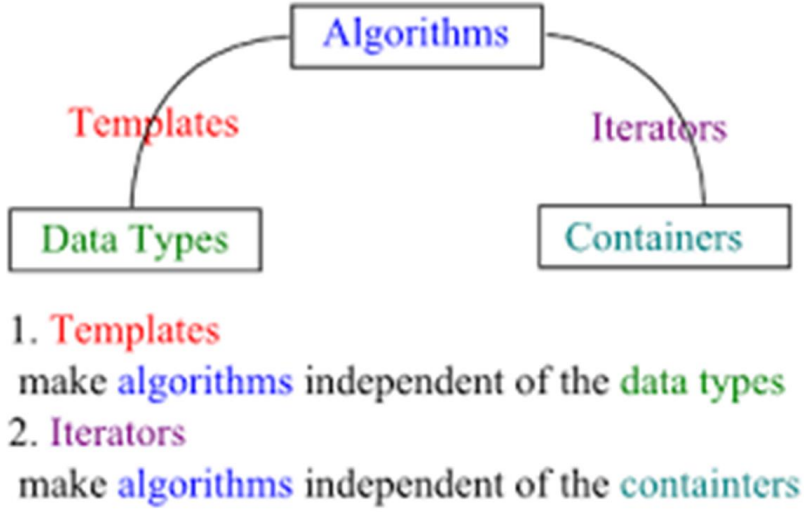
LECTURE 3

Iterators in C++ STL

Iterators in C++ STL

- Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequence of numbers, characters etc. They reduce the complexity and execution time of program.





Iterators in C++ STL

- Operations of iterators :-

  1. begin() :- This function is used to return the beginning position of the container.

  2. end() :- This function is used to return the after end position of the container.

  3. advance() :- This function is used to increment the iterator position till the specified number mentioned in its arguments.

iterator3.cpp

```cpp
// C++ code to demonstrate the working of
// iterator, begin() and end()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main(){
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int>::iterator ptr;

    // Displaying vector elements using begin() and end()
    cout << "The vector elements are : ";
    for (ptr = ar.begin(); ptr < ar.end(); ptr++)
        cout << *ptr << " ";

    return 0;
}
```

```
The vector elements are : 1 2 3 4 5
```

iterator4.cpp

```
// C++ code to demonstrate the working of
// advance()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main(){
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int>::iterator ptr = ar.begin();

    // Using advance() to increment iterator position
    // points to 4
    advance(ptr, 3);

    // Displaying iterator position
    cout << "The position of iterator after advancing is : ";
    cout << *ptr << " ";
    return 0;
}
```

```
The position of iterator after advancing is : 4
```

Iterators in C++ STL

- Operations of iterators :-

  4. next() :- This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.

  5. prev() :- This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

  6. inserter() :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted**.

iterator5.cpp

```cpp
// C++ code to demonstrate the working of
// next() and prev()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main(){
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int>::iterator ptr = ar.begin();
    vector<int>::iterator ftr = ar.end();
    // Using next() to return new iterator points to 4
    auto it = next(ptr, 3);
    // Using prev() to return new iterator points to 3
    auto it1 = prev(ftr, 3);
    // Displaying iterator position
    cout << "The position of new iterator using next() is : ";
    cout << *it << " ";
    cout << endl;
    // Displaying iterator position
    cout << "The position of new iterator using prev() is : ";
    cout << *it1 << " ";
    cout << endl;
    return 0;
}
```

```
The position of new iterator using next() is : 4
The position of new iterator using prev() is : 3
```

iterator6.cpp

```cpp
// C++ code to demonstrate the working of
// inserter()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main(){
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int> ar1 = {10, 20, 30};
    vector<int>::iterator ptr = ar.begin();
    // Using advance to set position
    advance(ptr, 3);
    // copying 1 vector elements in other using inserter()
    // inserts ar1 after 3rd position in ar
    copy(ar1.begin(), ar1.end(), inserter(ar, ptr));
    // Displaying new vector elements
    cout << "The new vector after inserting elements is : ";
    for (int &x : ar) cout << x << " ";
    return 0;
}
```

```
The new vector after inserting elements is : 1 2 3 10 20 30 4 5
```

STL Iterators

- Iterators are allow to traverse sequences
- Methods
  - operator*
  - operator->
  - operator++, and operator—
- Different types of iterators - to support read, write and random access
- Containers define their own iterator types
- Changing the container can invalidate the iterator

LECTURE 4

Iterators, Pointer and Address Calculation

Iterators are pointers

- Iterators are pointers.
- Iterators can be used for address calculation.
- Iterators can access data by dereferencing.
- Iterator is a class of pointers. **begin** is an object (pointer). **end** is also an object (pointer).

itrs1.cpp

```cpp
// example of using pointer as iterator
#include <iostream>
using namespace std;

int main(int argc, const char* argv[]){
    int a[] = {11, 22, 33, 44, 55};
    int *begin = a, *end = a + 5;
    for (int i=0; i<5; i++){
        cout << a[i] << endl;
    }
    for (int *p=begin; p!=end; p++){
        cout << *p << endl;
    }
    return 0;
}
```

```
11
22
33
44
55
11
22
33
44
55
```

Iterator is a pointer for sequential traversal

Iterator is a pointer designed for sequential traversal.

Advantages of Iterators:

- Traversal over the whole data structure
- Serialize the data structure
- Apply the iterator to an algorithm regardless of the data container type.
- Access each individual data by de-referencing

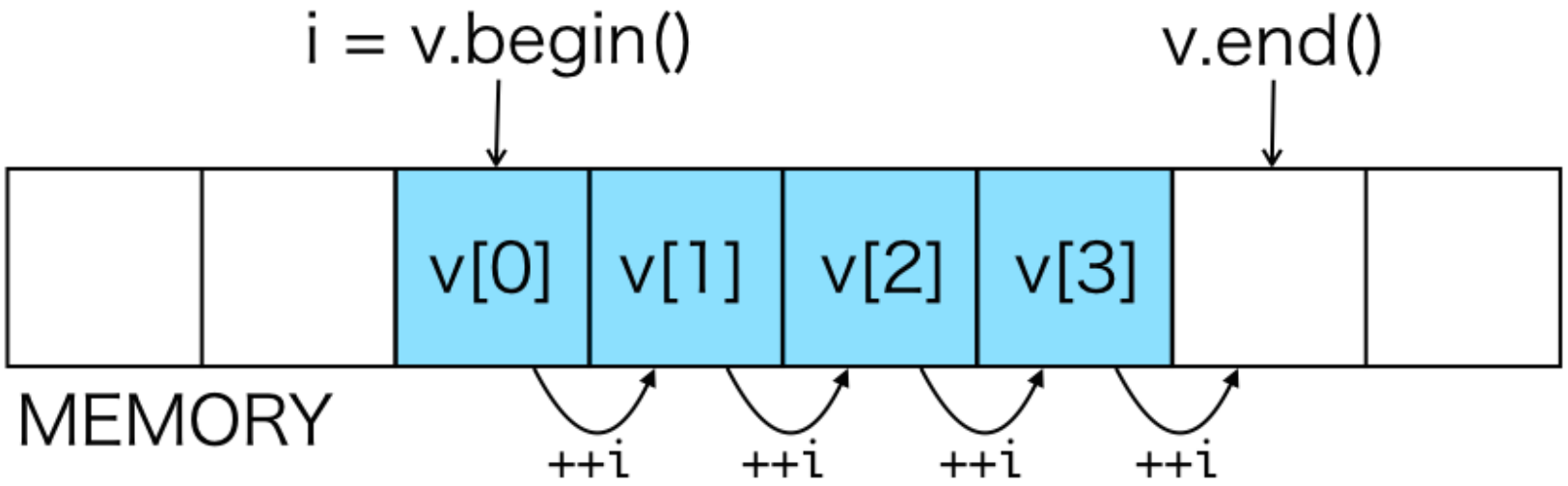
```cpp
// example of using pointer as iterator
#include <iostream>
#include <vector>
using namespace std;

int main(int argc, const char* argv[]){
    vector<int> a = {11, 22, 33, 44, 55};
    int *begin = &a[0], *end = &a[a.size()];
    cout << "Listing by a[i]: " << endl;
    for (int i=0; i<5; i++){
        cout << a[i] << endl;
    }
    cout << "Listing by pointer p: " << endl;
    for (int *p=begin; p!=end; p++){
        cout << *p << endl;
    }
    cout << "Listing by Iterator " << endl;
    for (auto itr = a.begin(); itr != a.end(); itr++){
        cout << *itr << endl;
    }
    return 0;
}
```

itrs2.cpp

```
Listing by a[i]:
11
22
33
44
55
Listing by pointer p:
11
22
33
44
55
Listing by Iterator
11
22
33
44
55
```

Address Calculation for Iterator




```
v[k] == *(v.begin()+k)
```

itrs3.cpp

The vector elements are : 1 2 3 4 5

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char *argv[]){
   vector<int> a = {11, 22, 33, 44, 55};
   int *begin = &a[0], *end = &a[a.size()];    // there is no a[a.size()].
                                               // only address calculation
   for (int i=0; i< (int) a.size(); i++){
      cout << *(a.begin()+i) << endl;
   }
   return 0;
}
```

Not all Iterator can take Arithmetic Calculation → not for list

```cpp
#include <iostream>
#include <list>
using namespace std;
int main(int argc, char* argv[]){
  list<int> a = {11, 12, 13, 14, 15};
  for (int i=0; i<a.size(); i++){
      cout << *(a.begin()+i) << endl;
  }
  return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
aits, _Allocator>&&, _CharT)'
 6139 |     operator+(basic_string<_CharT, _Traits,
      |     ^~~~~~~~
C:/msys64/mingw64/include/c++/10.1.0/bits/basic_str
itrs4.cpp:7:27: note:   'std::__cxx11::list<int>::i
s, _Allocator>'
    7 |       cout << *(a.begin()+i) << endl;
      |                         ^
```

Not all Iterator can take Arithmetic Calculation

```cpp
#include <iostream>
#include <list>
using namespace std;
int main(int argc, char* argv[]){
  list<int> a = {11, 12, 13, 14, 15};
  for (auto p = a.begin(); p!=a.end(); p++){
      cout << *p << endl;
  }
  return 0;
}
```

```
11
12
13
14
15
```

Design of an Iterator Class

- Design a class which a pointer as its data field. (This can be used to create begin() and or end() function in a data structure.
- Three operators need to be overloaded: !=, ++, and * (dereferencing)
- != : to check if two pointers are the same
- ++ : return current Iterator object reference
- * : return the value pointed by this Iterator's pointer.

itrs5.cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;
class PointerIterator{
      int *p;
    public:
      PointerIterator(int *address): p(address){}
      bool operator!=(const PointerIterator& other){
         return p != other.p;
      }
      PointerIterator& operator++(){ // return a reference
         ++p;
         return *this; // current pointer.
      }
      int operator*(){
        return *p;
      }
};
int main(int argc, const char* argv[]){
    vector<int> a = {11, 22, 33, 44, 55};
    PointerIterator end(&a[a.size()]);
    for (PointerIterator it(&a[0]); it != end; ++it){
        cout << *it << endl;
    }
    return 0;
}
```