BCSE102L- Structured and object-oriented programming

STL

Dr. P.Keerthika
Associate Professor
School of Computer Science & Engineering
VIT, Vellore.

BCSE102L- Structured and object-oriented programming

Module:1	**C Programming Fundamentals**	**2 hours**
Variables - Reserved words – Data Types – Operators – Operator Precedence - Expressions - Type Conversions - I/O statements - Branching and Looping: if, if-else, nested if, if-else ladder, switch statement, goto statement - Loops: for, while and do…while – break and continue statements.		
Module:2	**Arrays and Functions**	**4 hours**
Arrays: One Dimensional array - Two-Dimensional Array – Strings and its operations. User Defined Functions: Declaration – Definition – call by value and call by reference - Types of Functions - Recursive functions - Storage Classes - Scope, Visibility and Lifetime of Variables.		
Module:3	**Pointers**	**4 hours**
Declaration and Access of Pointer Variables, Pointer arithmetic – Dynamic memory allocation – Pointers and arrays - Pointers and functions.		
Module:4	**Structure and Union**	**2 hours**
Declaration, Initialization, Access of Structure Variables - Arrays of Structure - Arrays within Structure - Structure within Structures - Structures and Functions – Pointers to Structure -		
Module:5	**Overview of Object-Oriented Programming**	**5 hours**
Features of OOP - Classes and Objects - "this" pointer - Constructors and Destructors - Static Data Members, Static Member Functions and Objects - Inline Functions – Call by reference - Functions with default Arguments - Functions with Objects as Arguments - Friend Functions and Friend Classes.		
Module:6	**Inheritance**	**5 hours**
Inheritance - Types of Inheritance: Single inheritance, Multiple Inheritance, Multi-level Inheritance, Hierarchical Inheritance - Multipath Inheritance - Inheritance and constructors.		
Module:7	**Polymorphism**	**4 hours**
Function Overloading - Operator Overloading – Dynamic Polymorphism - Virtual Functions - Pure virtual Functions - Abstract Classes.		
Module:8	**Generic Programming**	**4 hours**
Function templates and class templates, Standard Template Library.		
	Total Lecture hours:	**30 hours**

BCSE102L- Structured and object-oriented programming – Text Books and Reference Books

Text Book(s)	
1.	Herbert Schildt, C: The Complete Reference, 4th Edition, McGraw Hill Education, 2017
2.	Herbert Schildt, C++: The Complete Reference, 4th Edition, McGraw Hill Education, 2017.
Reference Books	
1.	Yashavant Kanetkar, Let Us C: 17th Edition, BPB Publicaitons, 2020.
2.	Stanley Lippman and Josee Lajoie, C++ Primer, 5th Edition, Addison-Wesley publishers, 2012.

VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

BCSE102P- Structured and object-oriented programming Laboratory

	Indicative Experiments
1.	Programs using basic control structures, branching and looping
2.	Experiment the use of 1-D, 2-D arrays and strings and Functions
3.	Demonstrate the application of pointers
4.	Experiment structures and unions
5.	Programs on basic Object-Oriented Programming constructs.
6.	Demonstrate various categories of inheritance
7.	Program to apply kinds of polymorphism.
8.	Develop generic templates and Standard Template Libraries.

Text Book(s)	
1.	Robert C. Seacord, Effective C: An Introduction to Professional C Programming, 1st Edition, No Starch Press, 2020.
Reference Book(s)	
1.	Vardan Grigoryan and Shunguang Wu, Expert C++: Become a proficient programmer by learning coding best practices with C++17 and C++20's latest features, 1st Edition, Packt Publishing Limited, 2020.

BCSE102L- Structured and Object-Oriented Programming

- **Module-8: GENERIC PROGRAMMING**
  - **Function Template**
  - **Class Template**
  - **Standard Template Library**

Standard Template Libraries

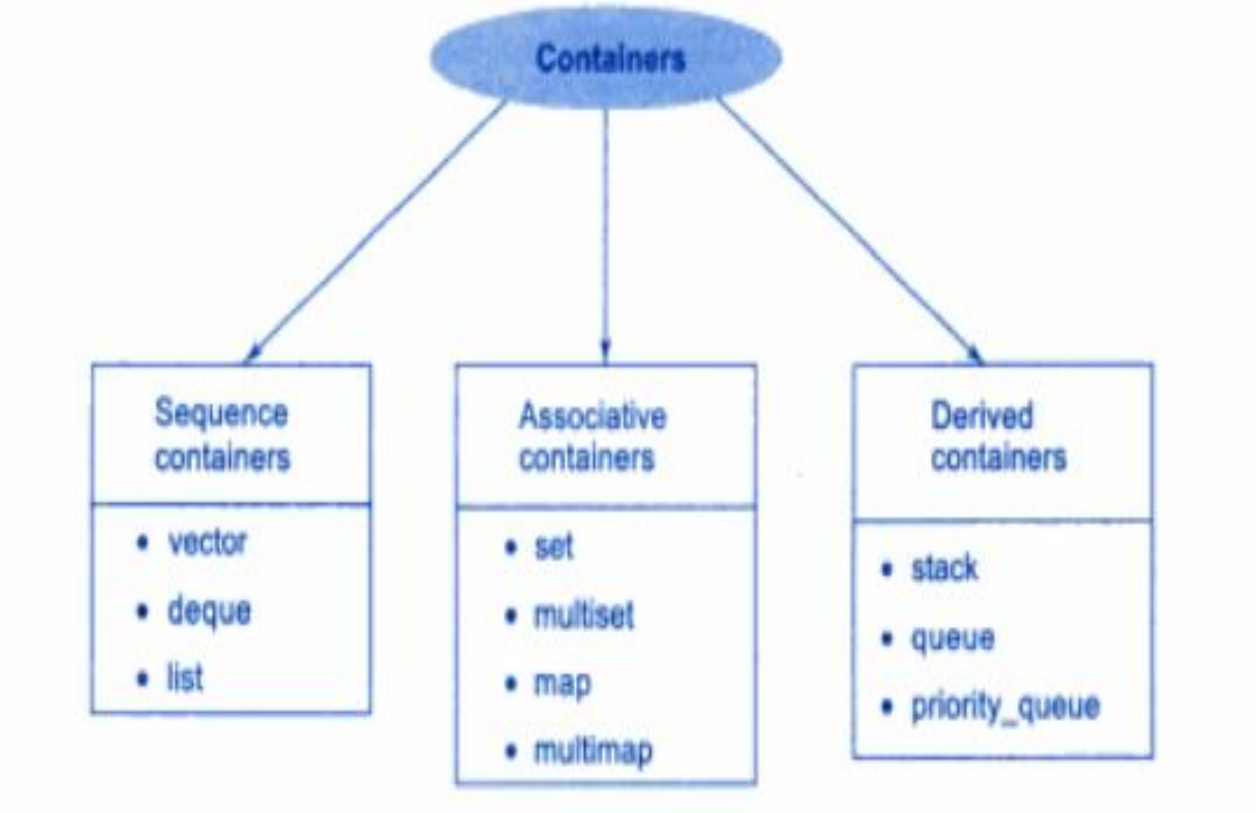
- Important Feature in C++ Programming.
- Standard approach for storing and processing of data.
- In detail: Set of general purpose templatized classes(Data structures) and functions(algorithms) that could be used as a Standard approach for storing and processing of data.
- Collection of these generic classes and functions is called **Standard Template Libraries.**
- **Very Broad area- so we discuss only some of the most important features.**
- **Using STL – Saves time and effort and leads to high quality programs.**
- **Reusing "well written" and "well tested components" defined in the STL.**

Components of STL

- It Contains several Components, But the core key components are:
  - **Containers, Algorithms & Iterators**
- **Containers:** It is an object that stores data. It is implemented by template classes and therefore can be easily customized to hold different types of data.
- **Algorithms:** It is a procedure to process the data stored in containers. STL includes algorithms to provide support to tasks such as **sorting, searching, copying, merging etc**. It was implemented by template functions.
- **Iterators:** It is like a pointer that points to an element in the container. We can use iterators to move through the contents of the container. In other words it was handled just like pointers.

STL Containers

- Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures for example arrays, list, trees, etc.



STL Containers

Container	Description	Header file	iterator
vector	vector is a class that creates a dynamic array allowing insertions and deletions at the back.	<vector>	Random access
list	list is the sequence containers that allow the insertions and deletions from anywhere.	<list>	Bidirectional
deque	deque is the double ended queue that allows the insertion and deletion from both the ends.	<deque>	Random access
set	set is an associate container for storing unique sets.	<set>	Bidirectional
multiset	Multiset is an associate container for storing non-unique sets.	<set>	Bidirectional

STL Containers

map	Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping).	<map>	Bidirectional
multimap	multimap is an associate container for storing key-value pair, and each key can be associated with more than one value.	<map>	Bidirectional
stack	It follows last in first out(LIFO).	<stack>	No iterator
queue	It follows first in first out(FIFO).	<queue>	No iterator
Priority-queue	First element out is always the highest priority element.	<queue>	No iterator

Drawbacks of Arrays in C++

- Size of the array is fixed constant.
- Passing an array as a parameter to a function is inconvenient, you must also pass the size of the array as a separate parameter.
- No way to conveniently insert elements at the beginning or in the middle of an array.
- Similarly, it is inconvenient to remove an element from an array.
- You cannot return an array from a function

STL Containers- Vector

- A vector is a sequence type container class that implements dynamic array, means size **automatically changes when appending elements.** A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.
- defined inside the **<vector>** header file
- SYNTAX: vector<object_type> v1;

Collection of objects of same data type
Alternatively- Class Template

For Example:

```
int main()
{
        int a[5]={1,2,3,4,5}; // Array Declaration & Initialization
        vector<int>v1; // Declaration of vector
        vector<string>v2;
        return 0;
}
```

STL Containers- Vector

```
Class Room
{
        int length;
        int breadth;
};
int main()
{
        int a[5]={1,2,3,4,5}; // Array Declaration & Initialization
        vector<int>v1;
        vector<string>v2;
        vector<float>v3
        Room room; // object creation
        vector<Room>rooms; // Collection of objects
        return 0;
}
```

Likes this we can use for all data types

Declaring a Vector

- Consider the syntax for declaring an array of 10 integers
- SYNTAX:

```
const int size =10;
int numbers[size];
```

The corresponding declaration for a vector of 10 integers is

```
const int size =10;
vector<object_type> v1; //syntax
vector<int>numbers[size];
vector<int>v{1,2,3,4}; // other ways
```

Initializing a Vector

- We can also pass an optional second parameter when we declare a vector, which denotes the value that should be placed in every element of the vector.

```
const int SIZE = 5;
vector<int> Numbers(SIZE);
```

0	0	0	0	0

```
const int SIZE = 5;
vector<int> Numbers(SIZE, 18);
```

18	18	18	18	18

```
const int SIZE = 5;
vector<double> Numbers(SIZE, 3.5);
```

3.5	3.5	3.5	3.5	3.5

Initializing a Vector- other forms

```
vector<int>v1;
vector<int>v2= {1,2,3,4};
vector<int>v3(v2);
vector<int>v4=v2;
vector<int>v5{5,6,7,8};
vector<int>v6(10);
```

Accessing a Vector

- Vectors provide a size() method that reports the number of elements currently in the vector.
- Accessing elements by their position in a vector uses exactly the same syntax as accessing elements of an array:

```
for (int i = 0; i < Vec.size(); i++)  //for(auto i:vec)
 {
        cout << "Vec[" << i << "] = " << Vec[i] << endl;
}
```

Vectors Without an Initial Size

- We can also declare a variable without an initial size

```
vector<int> V;
```

- When we do this, the vector is empty. It has no elements, so you can't even access V[0].
- Calling the size() method will return 0.
- There is also a method named **empty()** that returns true if the vector is empty or false if it is not.
- An empty vector doesn't seem to be very useful. Fortunately since **it can grow and shrink arbitrarily,** we have other ways of inserting data into it.

Resizing a vector

V.resize(int N)

- Resizes the vector V so that it contains exactly N elements. If V previously had less than N elements (thus making it larger), all of the old elements will remain as they were. If V previously had more than N elements (thus shrinking it), any elements at index N or greater will be removed.

V.clear()

- Removes all of the elements from the vector and sets its size to 0.

Example:

```
vector<int> V; // V starts out empty
V.resize(10); // can now access V[0]...V[9]
V[9] = 35;
V.resize(9); // can now only access up to V[8]
// The 35 that we set previously no longer exists
V.clear(); // Now V is empty again
```

Vector- Inserting at end

For Example:

```
int main()
{
        vector<int>v={1,2,3,4}; Vector Declaration & Initialization
        v.push_back(5);
        for (auto i:v)
                cout<<i<<endl;
        return 0;
}
```

Printing all elements inside a vector

```
using namespace std;
int main()
{
        vector<int> a(10);
        a[0] = 12;
        a[1] = 23;
        for (int i = 0; i < a.size(); i++)
        {
                cout << a[i];
        }
        return 0;
}
```

Comparing Two Vectors

```
using namespace std;
int main()
{
        vector<int> test1(3, 100);
        vector<int>test2(test1);
        if (test1 == test2)
        {
                cout << "Both the vectors are equal";
        }
        else
        {
                cout << "Both Vectors are not equal";
        }
        return 0;
}
```

Passing a vector to a function

```
void display(vector<int> k)
{
        for (int i =0; i < k.size(); i++)
        {
                cout << k[i];
        }
}
int main()
{
        vector<int> test;
        test.push_back(1);
        test.push_back(8);
        test.push_back(4);
        display(test);
        return 0;
}
```

STL Containers- Vector Example

```
int main()
{
        vector<string>itemlist;
        string item;
        cout<<"Enter products to be added in itemlist"<<endl;
        while(cin>>item)
        {
                itemlist.push_back(item);
        }
        cout<<"Received Item List"<<endl;
        for(auto i : itemlist)
                cout<<"i"<<endl;
        return 0;
}
        // some other functions
        // itemlist.empty(), itemlist.size(),v1==v2
```

C++ Vector Functions

Function	Description
at()	It provides a reference to an element.
back()	It gives a reference to the last element.
front()	It gives a reference to the first element.
swap()	It exchanges the elements between two vectors.
push_back()	It adds a new element at the end.
pop_back()	It removes a last element from the vector.
empty()	It determines whether the vector is empty or not.
insert()	It inserts new element at the specified position.
erase()	It deletes the specified element.
resize()	It modifies the size of the vector.
clear()	It removes all the elements from the vector.
size()	It determines a number of elements in the vector.
capacity()	It determines the current capacity of the vector.
assign()	It assigns new values to the vector.

C++ Vector Functions

operator=()	It assigns new values to the vector container.
operator[]()	It access a specified element.
end()	It refers to the past-lats-element in the vector.
emplace()	It inserts a new element just before the position pos.
emplace_back()	It inserts a new element at the end.
rend()	It points the element preceding the first element of the vector.
rbegin()	It points the last element of the vector.
begin()	It points the first element of the vector.
max_size()	It determines the maximum size that vector can hold.
cend()	It refers to the past-last-element in the vector.
cbegin()	It refers to the first element of the vector.
crbegin()	It refers to the last character of the vector.
crend()	It refers to the element preceding the first element of the vector.
data()	It writes the data of the vector into an array.
shrink_to_fit()	It reduces the capacity and makes it equal to the size of the vector.

STL Algorithms

- Algorithms are the functions used across a variety of containers for processing its contents.
- STL provides around 60 algorithm functions to perform the complex operations
- Algorithms are not the member functions of a container, but they are the standalone template functions.
- Algorithms save a lot of time and effort.
- If we want to access the STL algorithms, we must include the <algorithm> header file in our program.

STL Algorithms

- Based on the nature of the operations they perform, it was categorized as
  - Non-mutating algorithms- Not altering any value
  - Mutating Algorithms- Able to alter the value in container
  - Sorting algorithms
  - Set algorithms
  - Relational algorithms

STL- Sorting Algorithms

Operations	Description
binary_search( )	Conducts a binary search on an ordered sequence
equal_range( )	Finds a subrange of elements with a given value
inplace_merge( )	Merges two consecutive sorted sequences
lower_bound( )	Finds the first occurrence of a specified value
make_heap( )	Makes a heap from a sequence
merge( )	Merges two sorted sequences
nth_element( )	Puts a specified element in its proper place
partial_sort( )	Sorts a part of a sequence
partial_sort_copy()	Sorts a part of a sequence and then copies
Partition( )	Places elements matching a predicate first
pop_heap( )	Deletes the top element
push_heap( )	Adds an element to heap
sort( )	Sorts a sequence
sort_heap( )	Sorts a heap
stable_partition( )	Places elements matching a predicate first matching relative order
stable_sort( )	Sorts maintaining order of equal elements
upper_bound( )	Finds the last occurrence of a specified value

STL- Set Algorithms

Operations	Description
includes( )	Finds whether a sequence is a subsequence of another
set_difference( )	Constructs a sequence that is the difference of two ordered sets
set_intersection( )	Constructs a sequence that contains the intersection of ordered sets
set_symmetric_difference()	Produces a set which is the symmetric difference between two ordered sets
set_union( )	Produces sorted union of two ordered sets

STL- Relational Algorithms

Operations	Description
equal( )	Finds whether two sequences are the same
lexicographical_compare()	Compares alphabetically one sequence with other
max( )	Gives maximum of two values
max_element( )	Finds the maximum element within a sequence
min( )	Gives minimum of two values
min_element( )	Finds the minimum element within a sequence
mismatch( )	Finds the first mismatch between the elements in two sequences

STL- Mutating Algorithms

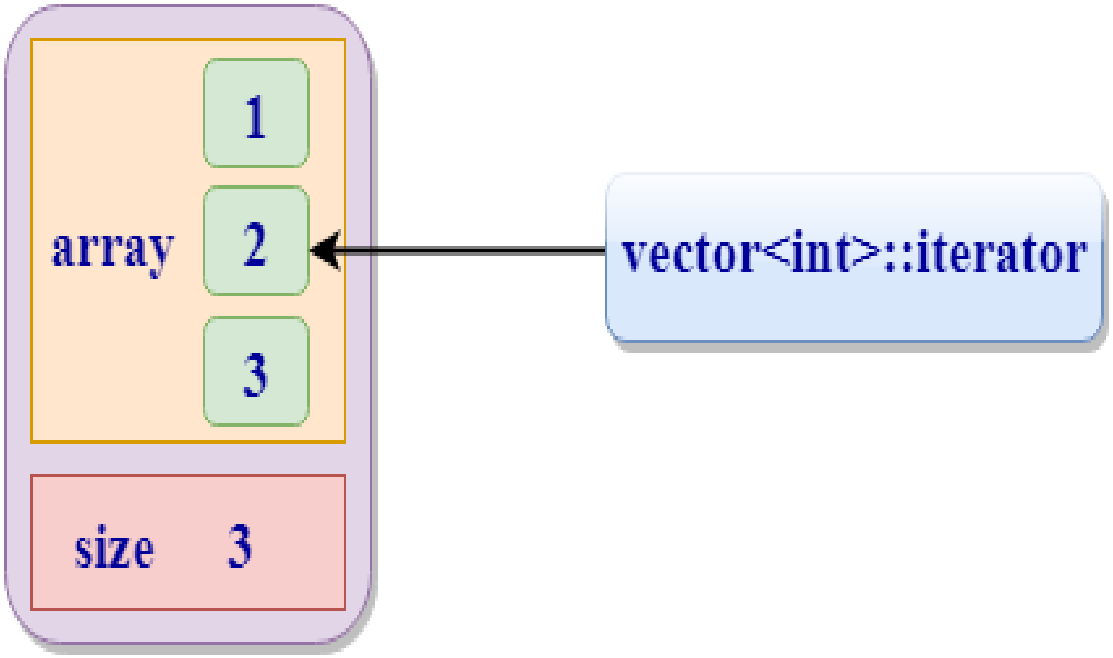
Operations	Description
fill_n( )	Fills first n elements with a specified value
generate( )	Replaces all elements with the result of an operation
generate_n( )	Replaces first n elements with the result of an operation
iter_swap( )	Swaps elements pointed to by iterators
random_shuffle( )	Places elements in random order
remove( )	Deletes elements of a specified value
remove_copy( )	Copies a sequence after removing a specified value
remove_copy_if( )	Copies a sequence after removing elements matching a predicate
remove_if( )	Deletes elements matching a predicate
replace( )	Replaces elements with a specified value
replace_copy( )	Copies a sequence replacing elements with a given value
replace_copy_if( )	Copies a sequence replacing elements matching a predicate
replace_if( )	Replaces elements matching a predicate
reverse( )	Reverses the order of elements
reverse_copy( )	Copies a sequence into reverse order
rotate( )	Rotates elements
rotate_copy( )	Copies a sequence into a rotated
swap( )	Swaps two elements
swap_ranges( )	Swaps two sequences
transform( )	Applies an operation to all elements
unique( )	Deletes equal adjacent elements
unique_copy( )	Copies after removing equal adjacent elements

STL Algorithms – Example

```
using namespace std;
int main()
{
        int i=10;
        int j=20;
        cout<<"Value of i:"<<i<<endl;
        cout<<"Value of j:"<<j<<endl;
        swap(i,j);
        cout<<"After Swap Value of i:"<<i<<endl;
        cout<<"After Swap Value of j:"<<j<<endl;
        return 0;
}
```

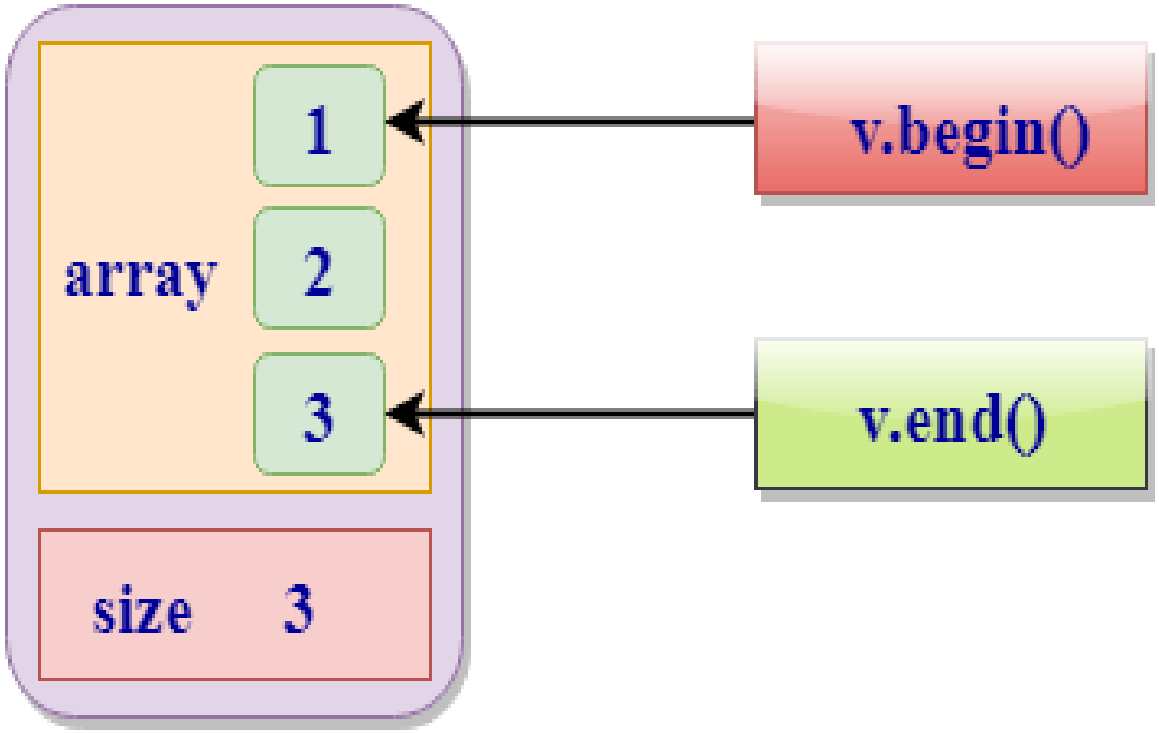
STL Iterators

- Iterators are pointer-like entities used to access the individual elements in a container.
- Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.



STL Iterators - Functions

- **begin()**: The member function begin() returns an iterator to the first element of the vector.
- **end()**: The member function end() returns an iterator to the past-the-last element of a container.



STL Iterators - Categories

Iterator	Access method	Direction of movement	I/O capability	Remark
Input	Linear	Forward only	Read only	Cannot be saved
Output	Linear	Forward only	Write only	Cannot be saved
Forward	Linear	Forward only	Read/Write	Can be saved
Bidirectional	Linear	Forward and backward	Read/Write	Can be saved
Random	Random	Forward and backward	Read/Write	Can be saved

INPUT ITERATOR:

- An Input iterator is an iterator that allows the program to read the values from the container.
- Dereferencing the input iterator allows us to read a value from the container, but it does not alter the value.
- An Input iterator is a one way iterator.
- An Input iterator can be incremented, but it cannot be decremented.

STL Iterators - Categories

- **Output iterator:**
  - An output iterator is similar to the input iterator, except that it allows the program to modify a value of the container, but it does not allow to read it.
  - It is a one-way iterator.
  - It is a write only iterator.

- **Forward iterator:**
  - Forward iterator uses the ++ operator to navigate through the container.
  - Forward iterator goes through each element of a container and one element at a time.

STL Iterators - Categories

- **Bidirectional iterator:**
  - A Bidirectional iterator is similar to the forward iterator, except that it also moves in the backward direction.
  - It is a two way iterator.
  - It can be incremented as well as decremented.
- **Random Access Iterator:**
  - Random access iterator can be used to access the random element of a container.
  - Random access iterator has all the features of a bidirectional iterator, and it also has one more additional feature, i.e., pointer addition. By using the pointer addition operation, we can access the random element of a container.

STL Iterators – Operations Supported

iterator	Element access	Read	Write	Increment operation	Comparison
input	->	v = *p		++	==,!=
output			*p = v	++	
forward	->	v = *p	*p = v	++	==,!=
Bidirectional	->	v = *p	*p = v	++,--	==,!=
Random access	->,[ ]	v = *p	*p = v	++,--,+,-,+=,--=	==,!=,<,>, <=,>=

STL Iterators – Example

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;
int main()
{
        vector<int> a(9);
        a[0] = 12;
        a[1] = 23;
        vector<int>::iterator iter;
        for (iter = a.begin(); iter != a.end(); iter++)
        {
                cout << *iter<<endl;
        }
        return 0;
}
```

STL Iterators – Sorting

```
#include<algorithm>
int main()
{
        vector<int> test;
        test.push_back(1);
        test.push_back(8);
        test.push_back(4);
        vector<int>::iterator itr1,itr2;
        itr1 = test.begin();
        itr2 = test.end();
        sort(itr1, itr2);
        for (int i = 0; i < test.size(); i++)
        {
                cout << test[i];    }
        return 0;
}
```

sorting takes place between the two passed iterators or positions.

Thank you