Data Structures (15B11CI311)

Odd Semester 2021

3rd Semester , Computer Science and Engineering

Jaypee Institute Of Information Technology (JIIT), Noida

Outline

- Introduction of STL
- Review of Array without using STL
- Implementation of Array using STL

STL (Standard Template Library)

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- It is a library of container classes, algorithms, and iterators.
- It is a generalized library and so, its components are parameterized.
- A working knowledge of template classes is a prerequisite for working with STL.
- STL has three components
  - ✓ Containers
  - ✓ Algorithms
  - ✓ Iterators

STL (Standard Template Library)

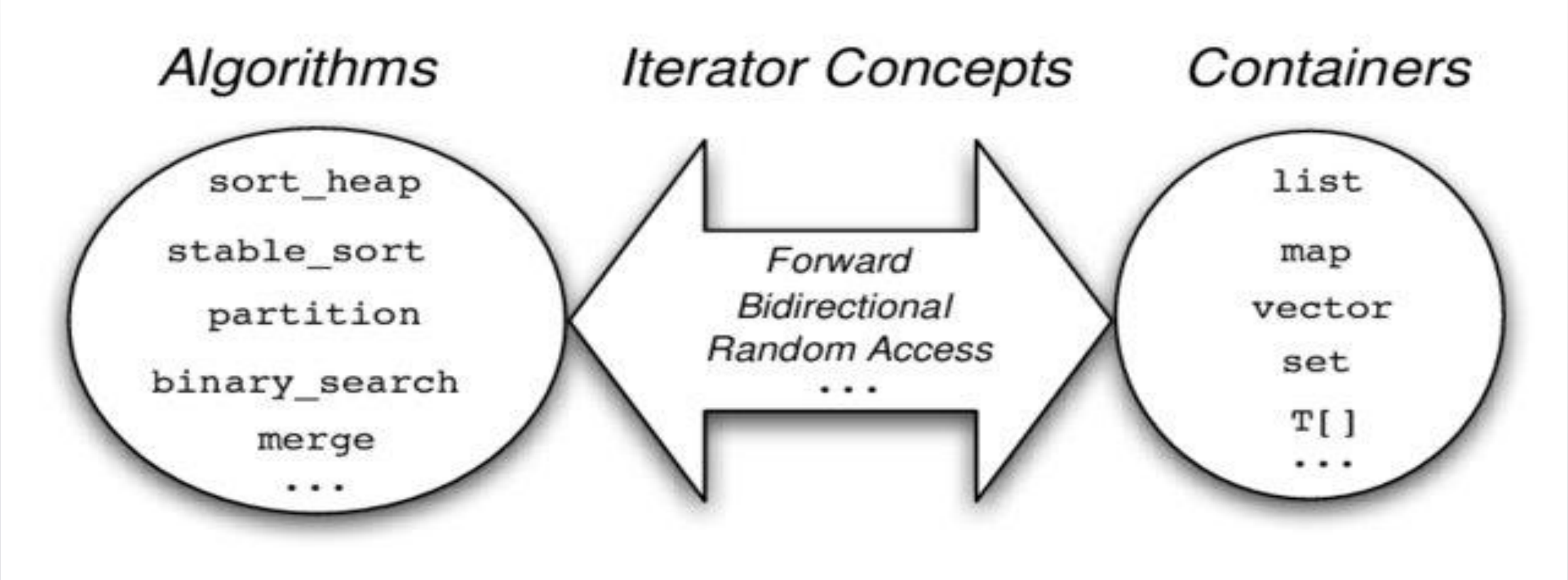
- These three components work together with one another in synergy to provide support to a variety of programming solutions.
- Algorithm employ iterators to perform operation stored in containers.

- **Container**
  - ✓ An object that stores data in memory into an organized fashion.
  - ✓ The containers in STL are implemented by template classes and therefore can be easily modified and customized to hold different types of data.

STL (Standard Template Library)

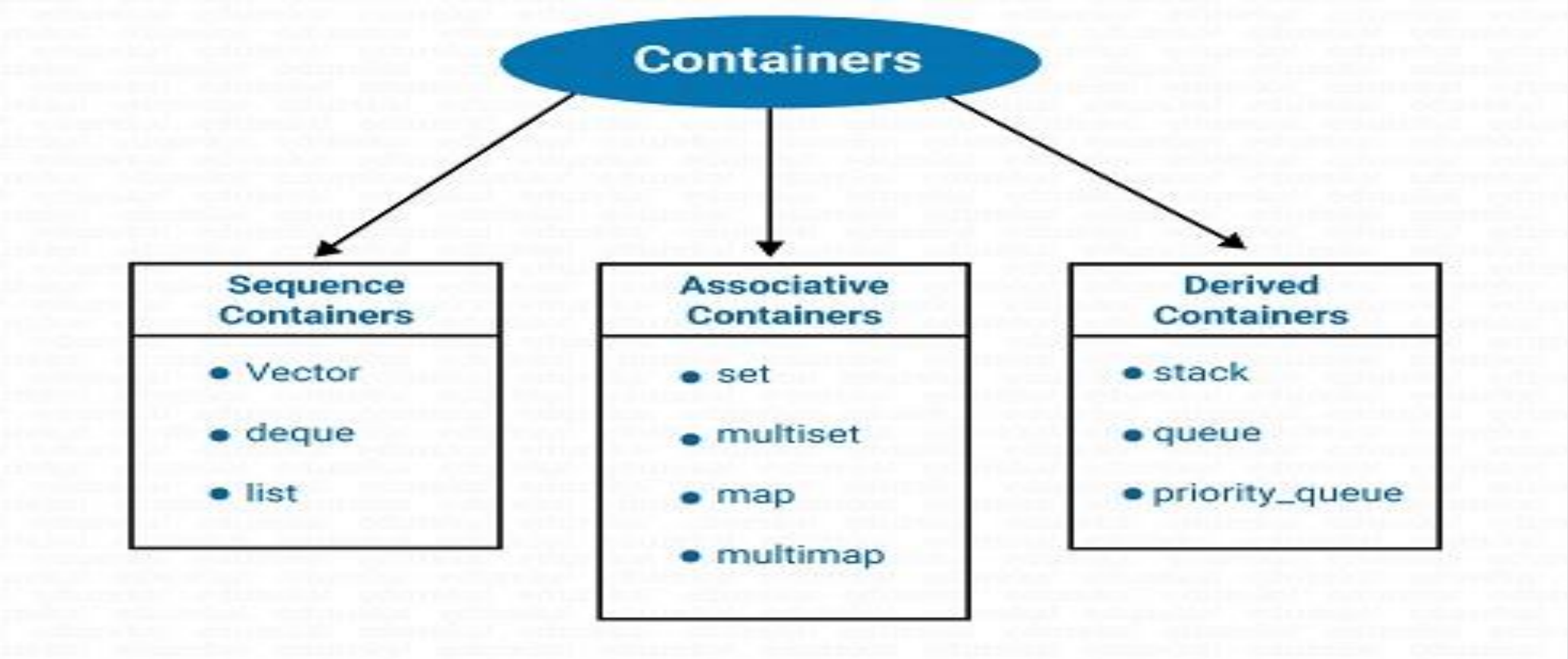
- **Procedure (Algorithms)**
  - ✓ used to process the data contained in the containers is defined as an algorithm.
  - ✓ The STL includes many different kinds of algorithms to provide support to tasks such as initializing, searching, copying, sorting, and merging, copying, sorting, and merging.
  - ✓ Algorithms are implemented by template functions.
- **Iterator**
  - ✓ can be defined as an object that points to an element in a container.
  - ✓ Iterators can be used to move through the contents of containers.
  - ✓ Iterators are handled just like pointers.
  - ✓ We can increment or decrement them.
  - ✓ Iterators connect algorithm with containers and play a key role in the manipulation of data stored in the containers.

STL (Standard Template Library)



Containers

- STL defines ten containers which are grouped into three categories.



Containers

Container : Vector

- ✓ Description: It can be defined as a dynamic array. It permits direct access to any element.
- ✓ Header File: <vector>
- ✓ Iterator: Random access

Container : List

- ✓ Description: It is a bidirectional linear list. It allows insertion and deletion anywhere
- ✓ Header File: <list>
- ✓ Iterator: Bidirectional

Containers

Container : deque

- ✓ Description: It is a double-ended queue. Allows insertions and deletions at both the ends. Permits direct access to any element.
- ✓ Header File: <deque>
- ✓ Iterator: Random access

Container : set

- ✓ Description: It is an associate container for storing unique sets. Allows rapid lookup.
- ✓ Header File: <set>
- ✓ Iterator:Bidirectional

Containers

Container : multiset

- ✓ Description: It is an associate container for storing non-unique sets.
- ✓ Header File: <set>
- ✓ Iterator: Bidirectional

Container : map

- ✓ Description: It is an associate container for storing unique key/value pairs. Each key is associated with only one value.
- ✓ Header File: <map>
- ✓ Iterator: Bidirectional

Containers

Container : Multimap

- ✓ Description: It is an associate container for storing key/value in which one key may be associated with more than one value (one-to-many mapping). It allows a key-based lookup.
- ✓ Header File: <map>
- ✓ Iterator: Bidirectional

Container : stack

- ✓ Description: A standard stack follows last-in-first-out(LIFO)
- ✓ Header File: <stack>
- ✓ Iterator: No iterator

Containers

Container : queue

- ✓ Description: A standard queue follows first-in-first-out(FIFO)
- ✓ Header File: <queue>
- ✓ Iterator: No iterator

Container : priority-queue

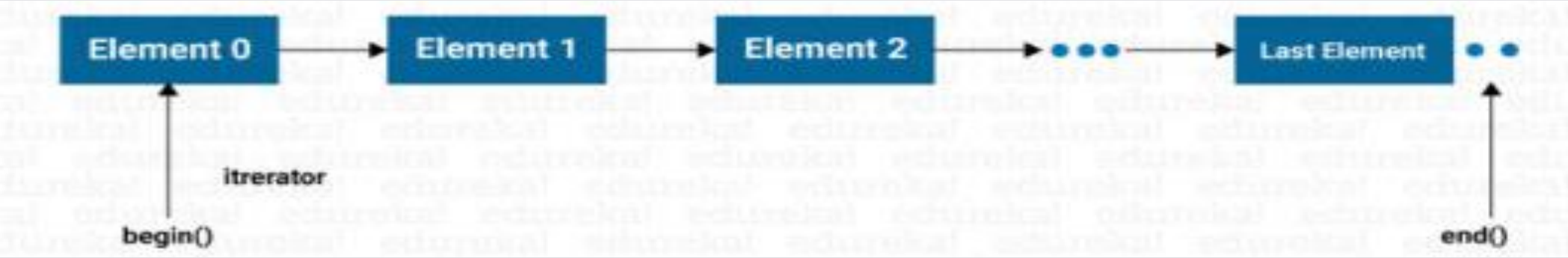
- ✓ Description: The first element out is always the highest priority element
- ✓ Header File: <queue>
- ✓ Iterator: No iterator

Sequence Containers

- Sequence containers store elements in a linear order.
- All elements are related to each other by their position along the line.
- They allow insertion of element and all of them support several operations on them.

The STL provides three types of sequence elements:

- Vector
- List
- Deque



Associative containers:

- They are designed in such a way that they can support direct access to elements using keys. They are not sequential.
- There are four types of associative containers:
  - Set
  - Multiset
  - Map
  - Multimap
- All the above containers store data in a structure called tree which facilitates fast searching, deletion, and insertion unlike sequential.
- Container set or multiset can store various items and provide operations for manipulating them using the values as the keys.
- And map or Multimap are used to store items in pair, one called the key and other called the value.

Derived containers:

- The STL provides three derived containers namely, stack, queue, and priority_queue. These are also known as container adaptors.
- There are three types of derived containers:
  - Stack
  - Queue
  - Priority_quue
- Stacks, queue and priority queue can easily be created from different sequence containers.
- The derived containers do not support iterators and therefore we cannot use them for data manipulation.
- However, they support two member function pop() and push() for implementing deleting and inserting operations.

Algorithms

- Algorithms are functions that can be used generally across a variety of containers for processing their content.
- Although each container provides functions for its basic operations, STL provides more than sixty standard algorithms to support more extended or complex operations.
- Standard algorithms also permit us to work with two different types of containers at the same time.
- STL algorithms save a lot of time and effort of programmers
- To access STL algorithms, we must include <algorithm> in our program.
- STL algorithm, based on the nature of operations they perform, may be categorized as under :
  - Nonmutating algorithms
  - Mutating algorithms
  - Sorting algorithms
  - Set algorithms
  - Relational algorithm

Iterators

- Iterators act like pointers and are used to access elements of the container.
- We use iterators to move through the contents of containers.
- Iterators are handled just like pointers.
- We can increment or decrement them as per our requirements.
- Iterators connect containers with algorithms and play a vital role in the manipulation of data stored in the containers.
- They are often used to pass through from one element to another, this process is called iterating through the container.
- There are five types of iterators:
  - Input
  - Output
  - Forward
  - Bidirectional
  - Random

Iterators

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

Iterators

- Different types of iterators must be used with the different types of containers such that only sequence and associative containers are allowed to travel through iterators.
- Each type of iterators is used for performing certain functions.
- The input and output iterators support the least functions.
- They can be used only to pass through in a container.
- The forward iterators support all operations of input and output iterators and also retain its position in the container.
- A Bidirectional iterator, while supporting all forward iterators operations, provides the ability to move in the backward direction in the container.

Standard Template Library in C++

Container

- Sequence Containers
- Associative Containers
- Container Adapters
- Unordered Associative Containers

Iterator

- begin()
- next()
- prev()
- advance()
- end()

Algorithm

- Sorting Algorithms
- Search algorithms
- Non modifying algorithms
- Modifying algorithms
- Numeric algorithms
- Minimum and Maximum operations

Vector

- Vectors in C++ function the same way as Arrays in a **dynamic manner** i.e. vectors can resize itself automatically whenever an item is added/deleted from it.
- The data elements in Vectors are placed in contagious memory locations and Iterator can be easily used to access those elements.
- Moreover, **insertion of items in takes place at the end of Vector**.

Vector Syntax:

```
vector<data_type> vector_name;
```

Vector Functions:

- **vector.begin():** It returns an iterator element which points to the first element of the vector.
- **vector.end():** It returns an iterator element which points to the last element of the vector.
- **vector.push_back():** It inserts the element into the vector from the end.
- **vector.pop_back():** It deletes the element from the end of the vector.
- **vector.size():** This function gives the size i.e. the number of elements in the vector.
- **vector.empty():** Checks whether the vector is empty or not.
- **vector.front():** It returns the first element of the vector.
- **vector.back():** It returns the last element of the vector.
- **vector.insert():** This function adds the element before the element at the given location/position.
- **vector.swap():** Swaps the two input vectors.

Example: Vector

```
#include <iostream>

#include <vector>

 using namespace std;

 int main()

{

    vector<int> V1;

     for (int i = 1; i <= 4; i++)

        V1.push_back(i);

     cout << "Displaying elements of vector using begin() and
end():\n";

    for (auto i = V1.begin(); i != V1.end(); ++i)

        cout << *i << " ";

cout << "\nSize of the input Vector:\n" << V1.size();

     if (V1.empty() == false)
            cout << "\nVector isn't empty";
    else
            cout << "\nVector is empty";

    cout << "\nvector.front() function:\n" << V1.front();

    cout << "\nvector.back() function:\n" << V1.back();

    V1.insert(V1.begin(), 8);

    cout<<"\nVector elements after the insertion of element using
vector.insert() function:\n";

    for (auto x = V1.begin(); x != V1.end(); ++x)
        cout << *x << " ";

    return 0;
}
```

Output:

- Statement **V1.insert(V1.begin(), 8)** inserts the element (8) at the beginning i.e. before the first element of the vector.

```
Displaying elements of vector using begin() and end(): 1 2 3 4

Size of the input Vector: 4

Vector isn't empty

vector.front() function:1

vector.back() function:4

Vector elements after the insertion of element using vector.insert() function:8 1 2 3 4
```

Vector Example

```
#include <iostream>
#include <vector>
using namespace std;
 int main() {
   // create a vector to store int
   vector<int> vec;
   int i;
   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;
   // push 5 values into the vector
   for(i = 0; i < 5; i++) {
      vec.push_back(i);
   }
 // display extended size of vec
   cout << "extended vector size = " << vec.size() <<
endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++) {
      cout << "value of vec [" << i << "] = " << vec[i] <<
endl;
   }

   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
      v++;
   }

   return 0;
}
```

Output:

vector size = 0

extended vector size = 5

value of vec [0] = 0

value of vec [1] = 1

value of vec [2] = 2

value of vec [3] = 3

value of vec [4] = 4

value of v = 0

value of v = 1

value of v = 2

value of v = 3

value of v = 4

- ✓ The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.
- ✓ The size( ) function displays the size of the vector.
- ✓ The function begin( ) returns an iterator to the start of the vector.
- ✓ The function end( ) returns an iterator to the end of the vector.

Implementation of Array without using STL

Array

- Array is a data structure that represents a collection of the homogenous data elements.

Declaration of Array

- Datatype Arrayname [size]

Example:

int Str [10];

- Elements of an array denoted by the subscript notation $Str_{0,}$ $Str_{1,}$ $Str_{2,}$ St_{3}.... $Str_{9.}$

Array

- The element of array are referenced by an index.
- The element of array are stored respectively in successive memory location.
- Once an array is created, its size is fixed. It cannot be changed.

Examples Using Arrays

- Initializers

  int n[5] = { 1, 2, 3, 4, 5 };

  - If not enough initializers, rightmost elements become 0

    int n[5] = { 0 }

    - All elements 0
  - C arrays have no bounds checking
- If size omitted, initializers determine it

  int n[ ] = { 1, 2, 3, 4, 5 };

  - 5 initializers, therefore 5 element array

Length of Array

Length = Upper Bound – Lower Bound + 1

Memory representation of Array

- LOC(LA[K])= Base(LA)+ w(K-lower Bound)
- LOC(LA[K])address of the Kth element of the array LA.
- Base(LA) = Address of LA array or the address of first element.
- W= number of words per memory cell for the array LA.

Example

- Array AUTO store the records of number of automobile sold each year from 1932 through 1984. Base(AUTO) = 200 and w=4 words per memory cell for AUTO.
- Find out the address of K = 1965 element.

Solution

- Base(LA) = 200
- LA = 1932
- W = 4
- LOC(LA[K]) = Base(LA) + w(K-lower Bound)

LOC(LA[1965])= 200 + 4(1965 – 1932)

200 + 4 (33)

200 + 132

332

Array - Operation

- Traversing
- Insertion
- Deletion
- Searching
- Sorting
- Merging

Traversing

Goal : Traverse each element of a LA.

Let LA is a linear array with lower bound LB and upper bound UB and operation process.

1. Set K= LB (K is a variable)
2. Repeat steps 3 and 4 while K<= UB.
3. [Visit element] Apply process to LA[K].
4. [Increase Counter] Set K = K+1.

[End of loop]

5. Exit

Insertion

- Insert(LA,N, K, item)

  where

  LA is linear array of N element .K is a positive integer such that K<=N.

- Goal: insert an element item into the Kth position in LA.

1. [Intialize counter] Set J= N (J is a variable)
2. Repeat steps 3 and 4 while J>=K.
3. [Move Jth element downward.] Set LA[J+1] =LA[J].
4. [Increase Counter] Set J = J-1.

[End of step 2 loop.]

5. [Insert element] Set LA[K] = item.
6. Reset N= N+1.
7. Exit.

Deletion

LA is linear array of N element. K is a positive integer such that K<=N.

Goal: delete Kth element from LA.

- 1. set item = LA[K]
- 2. Repeat for J= k to N-1
- [Move j + 1st element upward] LA[J] = LA[J+1]
- [End of loop]
- 3.Reset N = N-1
- 4. Exit

Search

- Linear Search:- if the list is unordered
- Binary Search:-list must be ordered

Linear search

- 1.[Insert ITEM at the end of DATA] Set DATA [N+1]:= ITEM
- 2. [Initialize counter] Set LOC: =1.
- 3. [Search for ITEM]
- Repeat while DATA [LOC]! = ITEM (LOC < =N and DATA [LOC]! = ITEM.)
- Set LOC: = LOC+1.
- [End of loop].
- 4. [Successful?] If LOC= N+1, then Set LOC: =0.
- 5. EXIT

Binary Search

Condition

- Elements in the array must be ordered.

e.g. 2 4 7 10 11 45 50 59 60 66 69 70 79

- Find the middle element. 50
- Break list in two part.
- 2 4 7 10 11 45
- 59 60 66 69 70 79

Binary Search, cont.

- If the key is less than the middle element, you only need to search the key in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you only need to search the key in the second half of the array.

Algorithm

- Binary search(Data, LB, UB, Item, LOC) Data is a sorted array,

  BEG, End, Mid are variables.

1. Set Beg = LB, End= UB and Mid = ((Beg + End)/2).
2. while (Data[mid]=! Item) and (Beg<= End)
3. if Item<Data[Mid], then

   set End= Mid – 1.

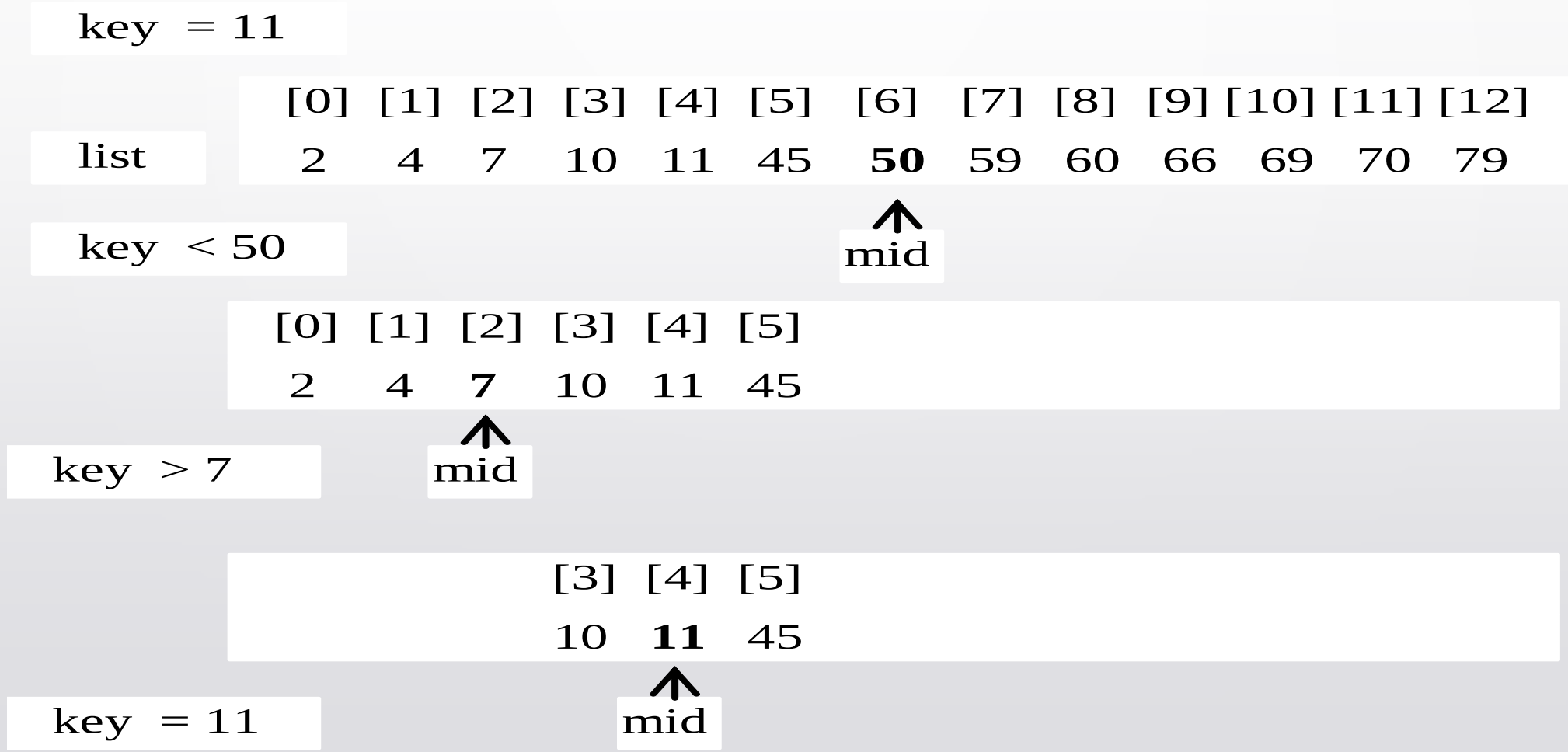
   else

   set Beg = Mid + 1.
4. Set Mid = ((Beg + End)/2).
5. if Data[Mid]=Item,

   then set LOC = MID

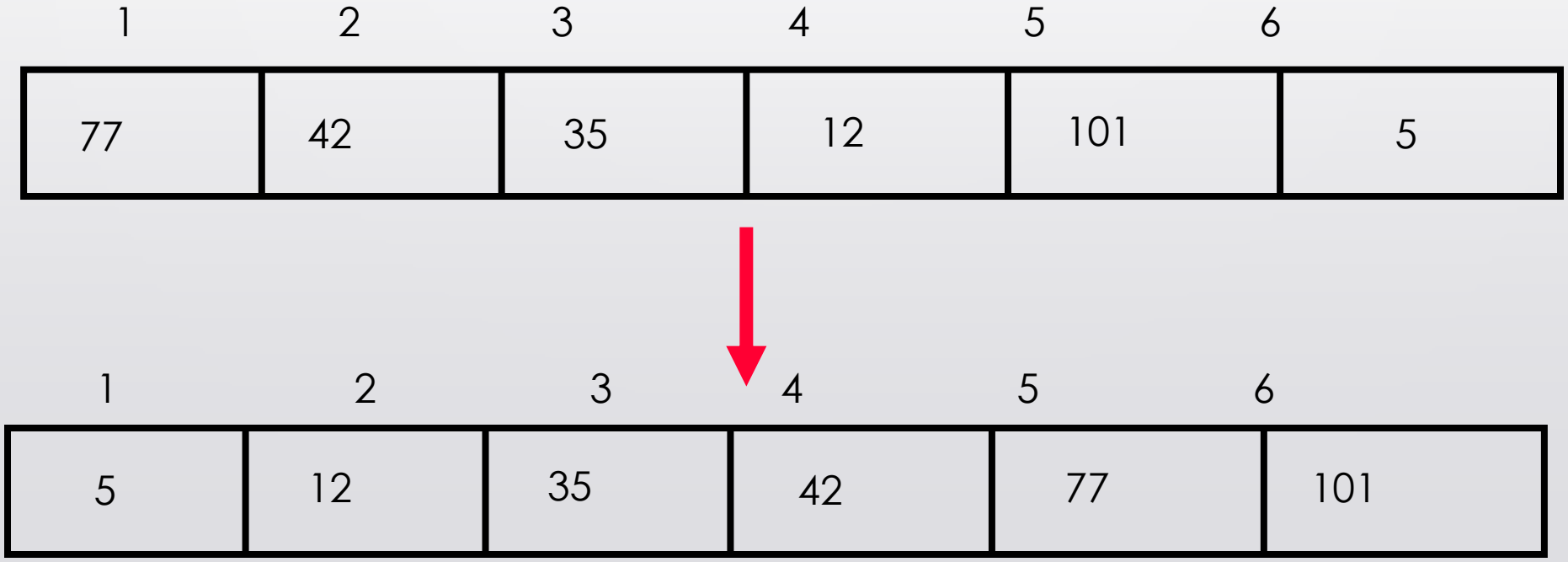
   else set LOC = Null.
6. Exit

Binary Search, Example



Sorting

- Sorting takes an unordered collection and makes it an ordered one.



Implementation of Array using STL

Array

- Array is a container in C++ STL which are used to store homogeneous (same) type of data and provides several useful functionalities over it.
- Arrays in STL provides the static implementation of arrays that is the size of array does not increase once created.
- **In order to use array using STL, the array library must be included.**
- #include <array>
- **SYNTAX of array container:**
- array<object_type, array_size> array_name;
- The above code creates an empty array of object_type with maximum size of array_size

Array

- **Initialization of Array:**

```
#include <array>
int main()
{
    //It will create an empty integer array of size 3
    array<int, 3> numbers;
    //It will create an integer array of size 3 having elements 1,2 & 3
    array<int, 3> num = {1, 2, 3};
    return 0;
}
```

Array

Functionalities served by Array are as follows:

- at() function
- front() function
- back() function
- fill() function
- empty() function
- max_size() function
- data() function
- operator []
- size() function
- swap() function
- Reverse() function
- Sort() function

at() function:

at() function in array container returns the value stored at given position or index.

```
#include <iostream>
#include <array>
using namespace std;

int main(){
   //It will create an integer array of size 3 having elements 1,2 & 3
   array<int, 3> num = {1, 2, 3};

   cout<<num.at(0); // Prints 1

   cout<<num.at(2); // Prints 3
return 0;
}
```

front() function:

front() function in array returns the value stored at first position in array. It is used to fetch the first element from array.

```
#include <iostream>
#include <array>
using namespace std;

int main(){
    //It will create an integer array of size 3 having elements 1,2 & 3
    array<int, 3> num = {1, 2, 3};

    cout<<num.front(); // Prints 1

return 0;
}
```

back() function:

back() function in array container returns the value stored at last position in array. It is used to fetch the last element from array.

```
#include <iostream>
#include <array>
using namespace std;

int main(){
    //It will create an integer array of size 3 having elements 1,2 & 3
    array<int, 3> num = {1, 2, 3};

    cout<<num.back(); // Prints 3

return 0;
}
```

fill() function:

fill() function in array fill the array with a particular value.

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
    //It will create an empty integer array of size 3
    array<int, 3> num;

    num.fill(10); //fill value 10 in array at every position

    for(int i=0;i<3;i++)
        cout<<num.at(i)<<" ";

return 0;
}
```

Output **10 10 10**

empty() function

empty() function of array container returns true if array is empty otherwise returns false.

```
#include <array>
#include <iostream>
using namespace std;

int main(){
    array<int, 0> num;

    if(num.empty())
        cout<<"Array is Empty";
    else
        cout<<"Array is Not Empty";
return 0;
}
```

Output: Array is Empty

max_size() function

max_size() function of array container returns the maximum number of elements the container is able to hold.

```
#include <array>
#include <iostream>
using namespace std;

int main(){
    array<int, 10> num;

    cout<<"Array can store upto "<<num.max_size()<<" elements";
return 0;
}
```

Output
Array can store upto 10 elements

data() function

data() returns the pointer to the first element of the array

```
#include<iostream>
#include<array>
using namespace std;

int main(){
    array<int , 5> arr{1,2,3,4,5};
    cout<<"First Element of array = "<<*(arr.data());
    return 0;
}
```

Output
First Element of array = 1

operator[]

operator[] This operator is used to reference the element present at position given inside the operator. It is similar to the at() function

```
#include<iostream>
#include<array>
using namespace std;

int main(){
    array<int,5> arr{1,2,3,4,5};
    cout<<"Second Element = "<<arr[1]<<endl;
    cout<<"Forth Element = "<<arr[3];
    return 0;
}
```

Output

Second Element = 2
Forth Element = 4

size() function

size() it returns the size of array or length of array

```
#include<iostream>
#include<array>
using namespace std;

int main(){
    array<int,5> arr{1,2,3,4,5};
    cout<<"Size of Array = "<<arr.size();
    return 0;
}
```

Output

Size of Array = 5

swap() function: swap() This function is used to swap the contents of one array with another array of same type and size.

```
#include <array>
#include <iostream>
using namespace std;

int main()
{
    array<int, 4> myarray1{ 1, 2, 3, 4 };
    array<int, 4> myarray2{ 3, 5, 7, 9 };

        // using swap() function to swap elements of
arrays
    myarray1.swap(myarray2);

        // printing the first array
    cout<<"myarray1 = ";
    for(auto it=myarray1.begin();
it<myarray1.end(); ++it)
        cout<<*it<<" ";

// printing the second array
    cout<<endl<<"myarray2 = ";
    for(auto it=myarray2.begin(); it<myarray2.end(); ++it)
        cout<<*it<<" ";
    return 0;
}
```

Output

```
myarray1 = 3 5 7 9
myarray2 = 1 2 3 4
```

reverse() function.

Reversing can be done with the help of reverse() function.

```
// C++ program to reverse Array
// using reverse() in STL

#include <bits/stdc++.h>
using namespace std;

int main()
{
	int a[] = { 1, 7, 2, 4, 8, 3 };
	int l = sizeof(a) / sizeof(a[0]);
	reverse(a, a + l);
	for (int i = 0; i < l; i++)
		cout << a[i] << " ";
	return 0;
}
```

Output: 3 8 4 2 7 1

sort() function.

C++ STL provides a sort() function that sorts an element of array

```
// C++ program to sort Array
// using sort() in STL

#include <bits/stdc++.h>
using namespace std;

int main()
{
	int a[] = { 1, 7, 2, 4, 8, 3 };
	int l = sizeof(a) / sizeof(a[0]);
	sort(a, a + l);
	for (int i; i < l; i++)
		cout << a[i] << " ";
	return 0;
}
```

Output: 1 2 3 4 7 8

References

- https://www.geeksforgeeks.org/working-with-array-and-vectors-using-stl-in-c/

- https://www.studytonight.com/cpp/stl/stl-container-array