RECAP - TEMPLATES

- C++ provides us with a feature of templates that allows functions and classes to operate with generic types. This allows the reusability of a function or class and allows it to work on many different data types without being rewritten for each one.

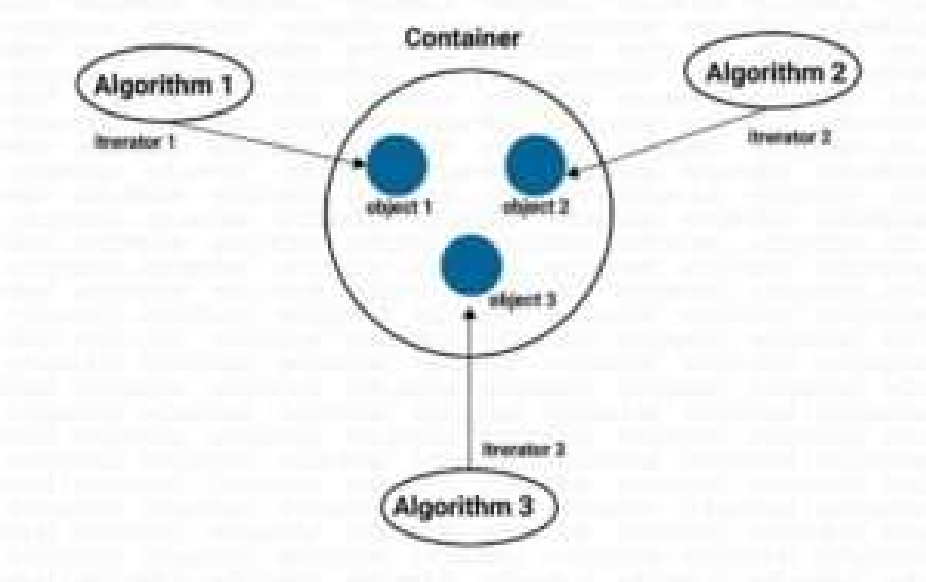
STANDARD TEMPLATE LIBRARY

- While programming many a time there is a need for creating functions that perform the same operations but work with different data types.
- So to overcome this problem C++ provides a feature to create a single generic function instead of many functions which can work with different data type by using the template parameter.
- The collection of these generic classes and functions is called Standard Template Library(STL)
- Syntax:

```
Using namespace std;
```

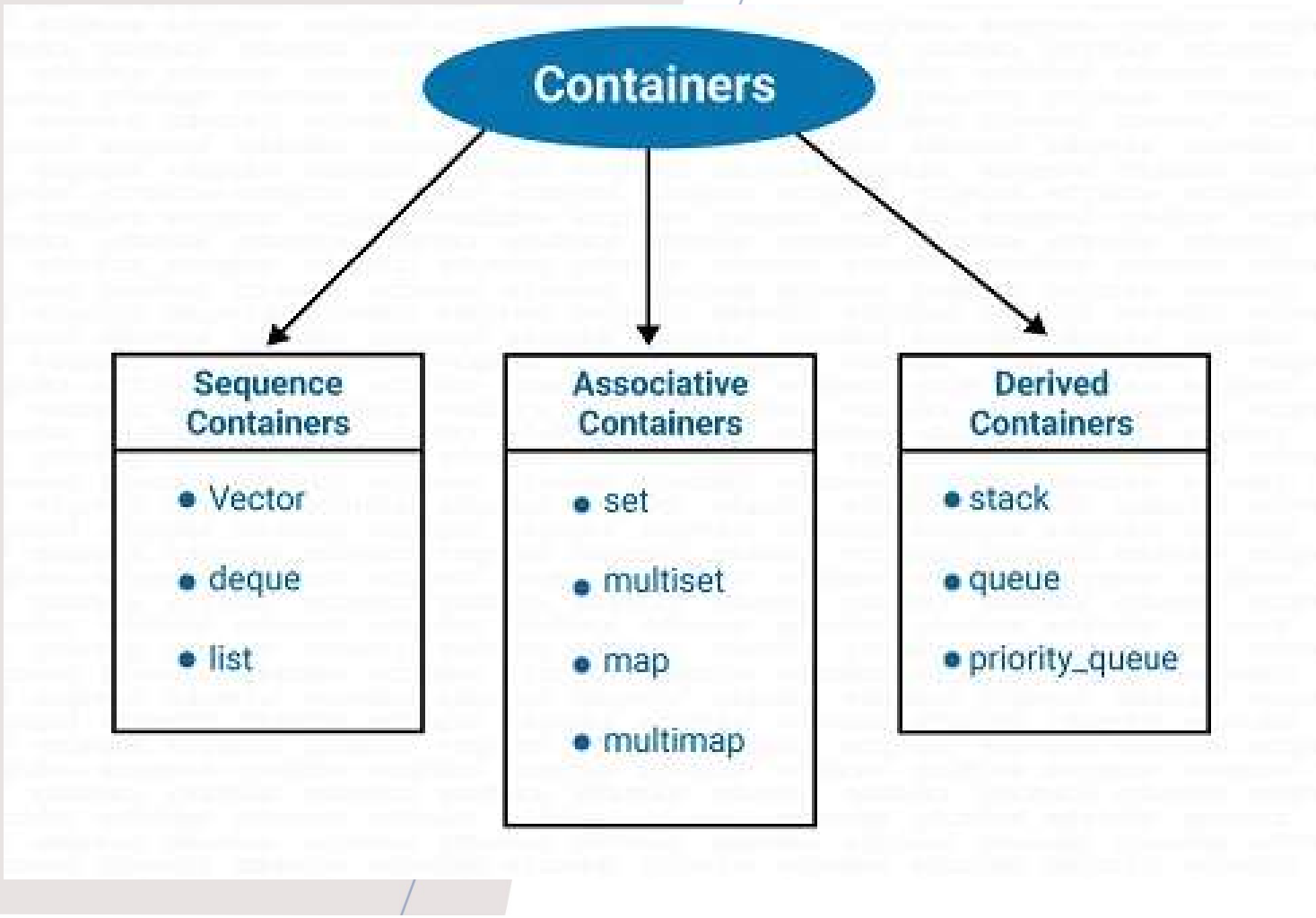
COMPONENTS OF STL

- STL has three components
  - Containers
  - Algorithms
  - Iterators
- Algorithm employ iterators to perform operation stored in containers.



- Container - A container is 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.
- Algorithm - A procedure that is 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 - An 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.

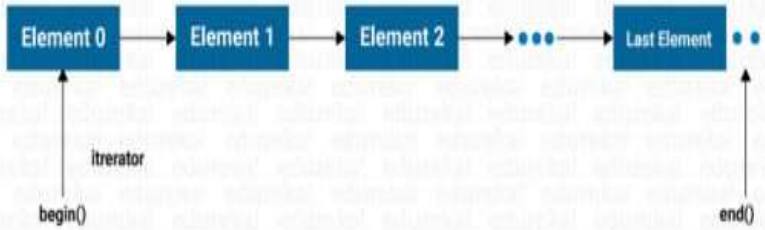
CONTAINERS



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

Containers	Description	Header file	Iterator
Vector	It can be defined as a dynamic array. It permits direct access to any element.	<vector>	Random access
List	It is a bidirectional linear list. It allows insertion and deletion anywhere	<list>	Bidirectional
deque	It is a double-ended queue. Allows insertions and deletions at both the ends. Permits direct access to any element.	<deque>	Random access
set	It is an associate container for storing unique sets. Allows rapid lookup.	<set>	Bidirectional
multiset	It is an associate container for storing non-unique sets.	<set>	Bidirectional
map	It is an associate container for storing unique key/value pairs. Each key is associated with only one value.	<map>	Bidirectional
multimap	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.	<map>	Bidirectional
stack	A standard stack follows last-in-first-out(LIFO)	<stack>	No iterator
queue	A standard queue follows first-in-first-out(FIFO)	<queue>	No iterator
priority-queue	The first element out is always the highest priority element	<queue>	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.

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:
  - 1.Stack
  - 2.Queue
  - 3.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 reinforce the philosophy of reusability. By using these algorithms, programmers can save a lot of time and effort. To have access to the 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:
    1. Input
    2. Output
    3. Forward
    4. Bidirectional
    5. Random

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

VECTOR

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array.
- Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

VECTOR - ITERATOR

- begin() – Returns an iterator pointing to the first element in the vector
- end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector
- rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
- rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

VECTOR - CAPACITY

- size() – Returns the number of elements in the vector.
- max_size() – Returns the maximum number of elements that the vector can hold.
- capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
- resize(n) – Resizes the container so that it contains ‘n’ elements.
- empty() – Returns whether the container is empty.
- shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
- reserve() – Requests that the vector capacity be at least enough to contain n elements.

VECTOR - ELEMENT ACCESS

- reference operator [g] – Returns a reference to the element at position ‘g’ in the vector
- at(g) – Returns a reference to the element at position ‘g’ in the vector
- front() – Returns a reference to the first element in the vector
- back() – Returns a reference to the last element in the vector
- data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

VECTOR - MODIFIERS

- assign() – It assigns new value to the vector elements by replacing old ones
- push_back() – It push the elements into a vector from the back
- pop_back() – It is used to pop or remove elements from a vector from the back.
- insert() – It inserts new elements before the element at the specified position
- erase() – It is used to remove elements from a container from the specified position or range.
- swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
- clear() – It is used to remove all the elements of the vector container
- emplace() – It extends the container by inserting new element at position
- emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

VECTOR VS LIST

- Both vector and list are sequential containers of C++ Standard Template Library. But there are many differences between them because of their internal implementation i.e.
- List stores elements at non contiguous memory location i.e. it internally uses a doubly linked list
- Insertion and Deletion in List is very efficient as compared to vector because to insert an element in list at start, end or middle, internally just a couple of pointers are swapped.
- Whereas, in vector insertion and deletion at start or middle will make all elements to shift by one. Also, if there is insufficient contiguous memory in vector at the time of insertion, then a new contiguous memory will be allocated and all elements will be copied there.

VECTOR VS LIST CONTINUED

- Random Access: As List is internally implemented as doubly linked list, therefore no random access is possible in List. It means, to access 15th element in list we need to iterate through first 14 elements in list one by one.
- Whereas, vector stores elements at contiguous memory locations like an array. Therefore, in vector random access is possible i.e. we can directly access the 15th element in vector using operator []

```
std::vector<int> vec(20);
vec[15] = 10;
```

- Iterator Invalidation : Deleting or Inserting an element in List does not invalidate any iterator because during insertion and deletion no element is moved from its position only a couple pointers are changed. Whereas, in vector insertion and deletion can invalidate the iterators