Standard Template Library

Parts of a Computer program

- Data Structure:

It refers to the way how data is stored in the memory.

- Algorithms:

It refers to the way how the stored data is manipulated.

Standard Template Library

- Container Class Libraries:

Libraries offered by the compiler to handle the storage and processing of data.

- Standard Template Library:

C++ have its own container class library called as Standard Template Library.

- Developed by Alexander Stepanov and Meng Lee of Hewlett Packard.

Entities of STL

- Container:

It’s a way how the stored data is organized in library. Eg : Stacks, Arrays. The STL containers are implemented by template classes, so they can be easily customized to hold different kinds of data.

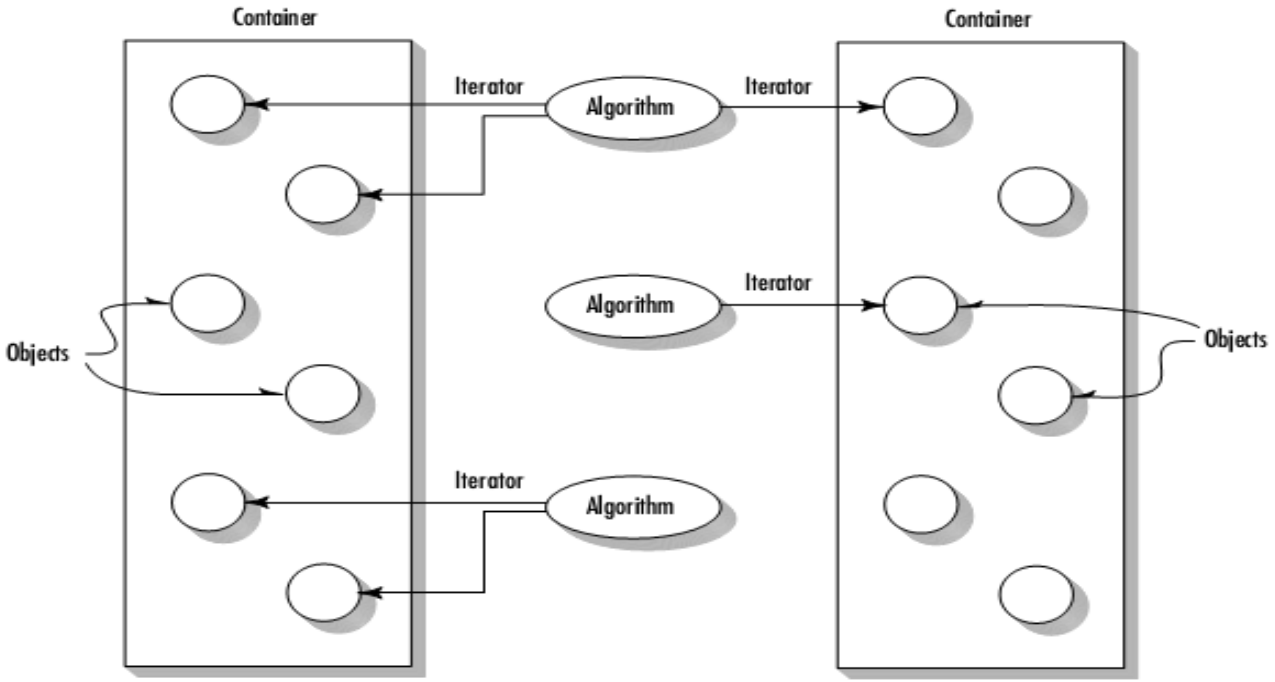
- Algorithms:

These are the procedures that are applied to containers to process their data in various ways. Eg: Sorting, copy. Algorithms are represented by template functions. These functions are not member functions of the con-tainer classes. Rather, they are standalone functions

Entities of STL

- Iterators:

They point to elements in a container. They are the generalization of the concept of pointers. Iterators are a key part of the STL because they connect algorithms with containers.



Algorithms use iterators to act on objects in containers

Containers

- A container is a way to store data, whether the data consists of built-in types such as int and float,or of class objects. The STL makes seven basic kinds of containers available, as well as three more that are derived from the basic kinds.
- Containers in the STL fall into two main categories: sequence and associative.
- The sequence containers are : vector, list, and deque.
- The associative containers are set, multiset, map, and multimap.
- Several specialized containers are derived from the sequence containers. These are stack, queue, and priority queue

Sequence Containers

- A sequence container stores a set of elements. Each element (except at the ends) is preceded by one specific element and followed by another. An ordinary C++ array is an example of a sequence container.

TABLE 15.1 Basic Sequence Containers

Container	*Characteristic*	*Advantages and Disadvantages*
ordinary C++ array	Fixed size	Quick random access (by index number) Slow to insert or erase in the middle Size cannot be changed at runtime
vector	Relocating, expandable array	Quick random access (by index number) Slow to insert or erase in the middle Quick to insert or erase at end
list	Doubly linked list	Quick to insert or delete at any location Quick access to both ends Slow random access
deque	Like vector, but can be accessed at either end	Quick random access (using index number) Slow to insert or erase in the middle Quick insert or erase (push and pop) at either the beginning or the end

Sequence Containers

- Initializing container object :

1. First you must include an appropriate header file.
2. Then you use the template format with the kind of objects to be stored as the parameter.

Example:

```
vector<int> aVect; //create a vector of ints
or
list<airtime> departure_list; //create a list of airtimes
```

- There's no need to specify the size of STL containers. The containers themselves take care of all memory allocation.

Associative Containers

- An associative container is not sequential; instead it uses keys to access data. The keys, typically numbers or stings, are used automatically by the container to arrange the stored elements in a specific order.

TABLE 15.2 Basic Associative Containers

Container	*Characteristics*
set	Stores only the key objects Only one key of each value allowed
multiset	Stores only the key objects Multiple key values allowed
map	Associates key object with value object Only one key of each value allowed
multimap	Associates key object with value object Multiple key values allowed

Associative Containers

- Creating associative containers is just like creating sequential ones:

set<int> intSet; //create a set of ints

or

multiset<employee> machinists; //create a multiset of employees

Member Functions

- Algorithms are the heavy hitters of the STL, carrying out complex operations like sorting and searching. However, containers also need member functions to perform simpler tasks that are specific to a particular type of container.

TABLE 15.3 Some Member Functions Common to All Containers

Name	*Purpose*
`size()`	Returns the number of items in the container
`empty()`	Returns true if container is empty
`max_size()`	Returns size of the largest possible container
`begin()`	Returns an iterator to the start of the container, for iterating forwards through the container
`end()`	Returns an iterator to the past-the-end location in the container, used to end forward iteration
`rbegin()`	Returns a reverse iterator to the end of the container, for iterating backward through the container
`rend()`	Returns a reverse iterator to the beginning of the container; used to end backward iteration

Container Adapters

- It's possible to create special-purpose containers from the normal containers mentioned previ-ously using a construct called container adapters.
- These special-purpose containers have sim-pler interfaces than the more general containers.
- The specialized containers implemented with container adapters in the STL are stacks, queues, and priority queues.

TABLE 15.4 Adapter-Based Containers

Container	*Implementation*	*Characteristics*
stack	Can be implemented as vector, list, or deque	Insert (push) and remove (pop) at one end only
queue	Can be implemented as list or deque	Insert (push) at one end, remove (pop) at other
priority queue	Can be implemented as vector or deque	Insert (push) in random order at one end, remove (pop) in sorted order from other end

Container Adapters

- Use a template within a template to instantiate these classes.
- For example, here's a stack object that holds type int, instantiated from the deque class:

stack< deque<int> > aStak;

A detail to note about this format is that you must insert a space between the two closing angle

brackets. You can't write

stack<deque<int>> astak; //syntax error

because the compiler will interpret the >>as an operator.

Algorithms

- An algorithm is a function that does something to the items in a container (or containers).
- Algorithms in the STL are not member functions or even friends of container classes, as they are in earlier container libraries, but are standalone template functions.
- Suppose you create an array of type int,with data in it:

```
int arr[8] = {42, 31, 7, 80, 2, 26, 19, 75};
```

You can then use the STL sort()algorithm to sort this array by saying

```
sort(arr, arr+8);
```

Where arr is the address of the beginning of the array, and arr+8is the past-the-end address

(one item past the end of the array).

Algorithms

TABLE 15.5 Some Typical STL Algorithms

Algorithm	*Purpose*
`find`	Returns first element equivalent to a specified value
`count`	Counts the number of elements that have a specified value
`equal`	Compares the contents of two containers and returns true if all corresponding elements are equal
`search`	Looks for a sequence of values in one container that corresponds with the same sequence in another container
`copy`	Copies a sequence of values from one container to another (or to a different location in the same container)
`swap`	Exchanges a value in one location with a value in another
`iter_swap`	Exchanges a sequence of values in one location with a sequence of values in another location
`fill`	Copies a value into a sequence of locations
`sort`	Sorts the values in a container according to a specified ordering
`merge`	Combines two sorted ranges of elements to make a larger sorted range
`accumulate`	Returns the sum of the elements in a given range
`for_each`	Executes a specified function for each element in the container

Iterators

- Iterators are pointer-like entities that are used to access individual data items (which are usually called elements), in a container.
- Often they are used to move sequentially from element to element, a process called iterating through the container.
- Different classes of iterators must be used with different types of container. There are three major classes of iterators: forward, bidirectional, and random access.

TABLE 15.6 Iterator Characteristics

Iterator Type	*Read/Write*	*Iterator Can Be Saved*	*Direction*	*Access*
Random access	Read and write	Yes	Forward and back	Random
Bidirectional	Read and write	Yes	Forward and back	Linear
Forward	Read and write	Yes	Forward only	Linear
Output	Write only	No	Forward only	Linear
Input	Read only	No	Forward only	Linear

Vectors

Member Functions push_back(), size(), and operator[]

Our first example,VECTOR,shows the most common vector operations.

```
// vector.cpp
// demonstrates push_back(), operator[], size()
#include <iostream>
#include <vector>
using namespace std;
int main()
{
vector<int> v; //create a vector of ints
v.push_back(10); //put values at end of array
v.push_back(11);
v.push_back(12);
v.push_back(13);
v[0] = 20; //replace with new values
v[3] = 23;
for(int j=0; j<v.size(); j++) //display vector contents
cout << v[j] << ' '; //20 11 12 23
cout << endl;
return 0;
}
```

Vectors

Member Functions swap(), empty(), back(), and pop_back()

The next example,VECTCON,shows some additional vector constructors and member functions.

```
// vectcon.cpp
// demonstrates constructors, swap(), empty(), back(), pop_back()
#include <iostream>
#include <vector>
using namespace std;
int main()
{ //an array of doubles
double arr[] = { 1.1, 2.2, 3.3, 4.4 };
vector<double> v1(arr, arr+4); //initialize vector to array
vector<double> v2(4); //empty vector of size 4
v1.swap(v2); //swap contents of v1 and v2
while( !v2.empty() ) //until vector is empty,
{
cout << v2.back() << ' '; //display the last element
v2.pop_back(); //remove the last element
} //output: 4.4 3.3 2.2 1.1
cout << endl;
return 0;
}
```

Vector

Member Functions insert()anderase()

```
// vectins.cpp
// demonstrates insert(), erase()
#include <iostream>
#include <vector>
using namespace std;
int main()
{
int arr[] = { 100, 110, 120, 130 }; //an array of ints
vector<int> v(arr, arr+4); //initialize vector to array
cout << “\nBefore insertion: “;
for(int j=0; j<v.size(); j++) //display all elements
cout << v[j] << ‘ ‘;
v.insert( v.begin()+2, 115); //insert 115 at element 2
cout << “\nAfter insertion: “;
for(j=0; j<v.size(); j++) //display all elements
cout << v[j] << ‘ ‘;
v.erase( v.begin()+2 ); //erase element 2
cout << “\nAfter erasure: “;
for(j=0; j<v.size(); j++) //display all elements
cout << v[j] << ‘ ‘;
Chapter 15
746
cout << endl;
return 0;
}
```

LISTS

Member Functions push_front(), front(), and pop_front

Our first example,LIST,shows how data can be pushed, read, and popped from both the front and the back.

```
//list.cpp
//demonstrates push_front(), front(), pop_front()
#include <iostream>
#include <list>
using namespace std;
int main()
{
list<int> ilist;
ilist.push_back(30); //push items on back
ilist.push_back(40);
ilist.push_front(20); //push items on front
ilist.push_front(10);
int size = ilist.size(); //number of items
for(int j=0; j<size; j++)
{
cout << ilist.front() << ' '; //read item from front
```

The Standard Template Library

```
15
THESTANDARD
TEMPLATELIBRARY
747
ilist.pop_front(); //pop item off front
}
cout << endl;
return 0;
}
```

Lists

Member Functions reverse(), merge(), and unique()

```
// listplus.cpp
// demonstrates reverse(), merge(), and unique()
#include <iostream>
#include <list>
using namespace std;
int main()
{
int j;
list<int> list1, list2;
int arr1[] = { 40, 30, 20, 10 };
int arr2[] = { 15, 20, 25, 30, 35 };
for(j=0; j<4; j++)
list1.push_back( arr1[j] ); //list1: 40, 30, 20, 10
for(j=0; j<5; j++)
list2.push_back( arr2[j] ); //list2: 15, 20, 25, 30, 35
list1.reverse(); //reverse list1: 10 20 30 40
list1.merge(list2); //merge list2 into list1
list1.unique(); //remove duplicate 20 and 30
int size = list1.size();
while( !list1.empty() )
{
cout << list1.front() << ' '; //read item from front
list1.pop_front(); //pop item off front
}
cout << endl;
return 0;
}
```


Thank you!