
FACE

Standard Template Library

- A generic collection of class templates and algorithms
- Allow programmers to easily implement standard data structures like queues, lists, and stacks
- Not bound to a specific object
- Uses templates hence the code is reusable and extensible

Standard Template Library

- Helps the developer deliver fast, efficient, and robust code
- Used to implement the hard part of using complex data structures easily
- Components of :
    i. Containers
    ii. Iterators
    iii. Algorithms
    iv. Functors

FACE

Containers

- Manages collections of objects of a certain kind
- Provides value rather than reference semantics
- Elements inside a container have a specific order
- To meet different needs, the STL provides different kinds of containers;
    i. Sequence containers
    ii. Associative containers
    iii. Unordered containers

FACE

Sequence Containers

- Ordered collections in which every element has a certain position

- Predefined sequence container are;

    i. Array
    ii. Vector
    iii. Deque
    iv. List
    v. forward_list.1

FACE

1. Arrays

- A sequence of elements with constant size
- To use an array, <array> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T, size_t N>
    class array;
}
```

- T is the type of elements of an array
- N specifies the number of elements

FACE

```
1   #include<iostream>
2   #include<array>
3   using namespace std;
4   int main()
5   {
6       array<int,6> ar = {1, 2, 3, 4, 5, 6};
7       array<int,6> ar1 = {7, 8, 9, 10, 11, 12};
8       cout << "The array elements are (using at()) : ";
9       for ( int i=0; i<6; i++)
10      cout << ar.at(i) << " ";
11      cout << endl;
12      cout << "The array elements are (using operator[]) : ";
13      for ( int i=0; i<6; i++)
14      cout << ar[i] << " ";
15      cout << endl;
16      cout << "The number of array elements is : ";
17      cout << ar.size() << endl;
18      cout << "Maximum elements array can hold is : ";
19      cout << ar.max_size() << endl;
20
21
22
```

```
1       ar.swap(ar1);
2       cout << "The first array elements after swapping are : ";
3       for (int i=0; i<6; i++)
4       cout << ar[i] << " ";
5       cout << endl;
6       cout << "The second array elements after swapping are : ";
7       for (int i=0; i<6; i++)
8       cout << ar1[i] << " ";
9       cout << endl;
10      ar1.empty()? cout << "Array empty":cout << "Array not empty";
11      cout << endl;
12      ar.fill(0);
13      cout << "Array after filling operation is : ";
14      for ( int i=0; i<6; i++)
15          cout << ar[i] << " ";
16      return 0;
17  }
18
19
20
21
22
```

OUTPUT

```
The array elements are (using at()) : 1 2 3 4 5 6
The array elements are (using operator[]) : 1 2 3 4 5 6
The number of array elements is : 6
Maximum elements array can hold is : 6
The first array elements after swapping are : 7 8 9 10 11 12
The second array elements after swapping are : 1 2 3 4 5 6
Array not empty
Array after filling operation is : 0 0 0 0 0 0
```

2. Vectors

- An abstraction that manages its elements with a dynamic C-style array
- To use a vector, <vector> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T,typename Allocator = allocator<T> >
    class vector;
}
```

- T is the type of elements of vector
- The default memory model is the model allocator, which is provided by the C++ standard library.

FACE

```cpp
#include <vector>
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
    vector<string> sen;
    sen.reserve(5);
    sen.push_back("Hello,");
    sen.insert(sen.end(),{"how","are","you","?"});
    copy (sen.cbegin(),sen.cend(),ostream_iterator<string>(cout," "));
    cout << endl;
    cout << " max_size(): " << sen.max_size() << endl;
    cout << " size(): " << sen.size() << endl;
    cout << " capacity(): " << sen.capacity() << endl;
    swap (sen[1], sen[3]);
    sen.insert (find(sen.begin(),sen.end(),"?"),"always");
    sen.back() = "!";


```

```
1       copy (sen.cbegin(), sen.cend(),ostream_iterator<string>(cout," "));
2       cout << endl;
3       cout << " size(): " << sen.size() << endl;
4       cout << " capacity(): " << sen.capacity() << endl;
5       sen.pop_back();
6       sen.pop_back();
7       sen.shrink_to_fit();
8       cout << " size(): " << sen.size() << endl;
9       cout << " capacity(): " << sen.capacity() << endl;
10      return 0;
11  }
12
13
14
15
16
17
18
19
20
21
22
```

OUTPUT

```
Hello, how are you ?
 max_size(): 2305843009213693951
 size(): 5
 capacity(): 5
Hello, you are how always !
 size(): 6
 capacity(): 10
 size(): 4
 capacity(): 4
```

3. Deque

- Manages its elements with a dynamic array and provides random access
- The dynamic array is open at both ends.
- To use a Deque, <deque> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T,typenameAllocator =allocator<T>>
    class deque;
}
```

- T is the type of elements of deque
- The default memory model is the model allocator, which is provided by the C++ standard library.

FACE

```cpp
#include <iostream>
#include <deque>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
    deque<string> coll;
    coll.assign (3, string("string"));
    coll.push_back ("last string");
    coll.push_front ("first string");
    copy (coll.cbegin(), coll.cend(),ostream_iterator<string>(cout,"\n"));
    cout << endl;
    coll.pop_front();
    coll.pop_back();
    for (unsigned i=1; i<coll.size(); ++i)
        coll[i] = "another " + coll[i];
    coll.resize (4, "resized string");
    copy (coll.cbegin(), coll.cend(),ostream_iterator<string>(cout,"\n"));
    return 0;
}
```

OUTPUT

```
first string
string
string
string
last string

string
another string
another string
resized string
```

4. List

- Manages its elements as a doubly linked list
- To use a List, <list> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T,typename Allocator = allocator<T> >
    class list;
}
```

- T is the type of elements of List
- The default memory model is the model allocator, which is provided by the C++ standard library.

FACE

```
1   #include <list>
2   #include <iostream>
3   #include <algorithm>
4   #include <iterator>
5   using namespace std;
6
7
8
9   void printLists (const list<int>& l1, const list<int>& l2)
10  {
11      cout << "list1: ";
12      copy (l1.cbegin(), l1.cend(), ostream_iterator<int>(cout," "));
13      cout << endl << "list2: ";
14      copy (l2.cbegin(), l2.cend(), ostream_iterator<int>(cout," "));
15      cout << endl << endl;
16  }
17
18
19
20
21
22
```

```
1   int main()
2   {
3       list<int> list1, list2;
4       for (int i=0; i<6; ++i)
5       {
6           list1.push_back(i);
7           list2.push_front(i);
8       }
9       printLists(list1, list2);
10      list2.splice(find(list2.begin(),list2.end(),3),list1);
11      printLists(list1, list2);
12      list2.splice(list2.end(),list2,list2.begin());
13      printLists(list1, list2);
14      list2.sort();
15      list1 = list2;
16      list2.unique();
17      printLists(list1, list2);
18      list1.merge(list2);
19      printLists(list1, list2);
20      return 0;
21  }
22
```

OUTPUT

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0

list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0

list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5

list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5

list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

5. Forward List

- Manages its elements as a singly linked list
- To use a Forward List, <forward_list> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T,typename Allocator = allocator<T> >
    class forward_list;
}
```

- T is the type of elements of Forward List
- The default memory model is the model allocator, which is provided by the C++ standard library.

FACE

```
1  #include <forward_list>
2  #include <iostream>
3  #include <algorithm>
4  #include <iterator>
5  #include <string>
6  using namespace std;
7
8
9  void printLists (const string& s, const forward_list<int>& l1,const
10 forward_list<int>& l2)
11 {
12     cout << s << endl;
13     cout << " list1: ";
14     copy (l1.cbegin(), l1.cend(), ostream_iterator<int>(cout," "));
15     cout << endl << " list2: ";
16     copy (l2.cbegin(), l2.cend(), ostream_iterator<int>(cout," "));
17     cout << endl;
18 }
19
20
21
22
```

```
1   int main()
2   {
3       forward_list<int> list1 = { 1, 2, 3, 4 };
4       forward_list<int> list2 = { 77, 88, 99 };
5       printLists ("initial:", list1, list2);
6       list2.insert_after(list2.before_begin(),99);
7       list2.push_front(10);
8       list2.insert_after(list2.before_begin(), {10,11,12,13} );
9       printLists ("6 new elems:", list1, list2);
10      list1.insert_after(list1.before_begin(),list2.begin(),list2.end());
11      printLists ("list2 into list1:", list1, list2);
12      list2.erase_after(list2.begin());
13      list2.erase_after(find(list2.begin(),list2.end(),99),list2.end());
14      printLists ("delete 2nd and after 99:", list1, list2);
15      list1.sort();
16      list2 = list1;
17      list2.unique();
18      printLists ("sorted and unique:", list1, list2);
19      list1.merge(list2);
20      printLists ("merged:", list1, list2);
21      return 0;
22  }
```

OUTPUT

```
initial:
 list1: 1 2 3 4
 list2: 77 88 99
6 new elems:
 list1: 1 2 3 4
 list2: 10 11 12 13 10 99 77 88 99
list2 into list1:
 list1: 10 11 12 13 10 99 77 88 99 1 2 3 4
 list2: 10 11 12 13 10 99 77 88 99
delete 2nd and after 99:
 list1: 10 11 12 13 10 99 77 88 99 1 2 3 4
 list2: 10 12 13 10 99
sorted and unique:
 list1: 1 2 3 4 10 10 11 12 13 77 88 99 99
 list2: 1 2 3 4 10 11 12 13 77 88 99
merged:
 list1: 1 1 2 2 3 3 4 4 10 10 10 11 11 12 12 13 13 77 77 88 88 99 99 99
 list2:
```

Associative Containers

- The elements are automatically ordered according to a certain ordering criterion
- The elements can be either values of any type or key/value pairs
- By default, the containers compare the elements or the keys with operator <(this can be changed)
- Predefined sequence container are;
    i. Set
    ii. Multiset
    iii. Map
    iv. Multimap

FACE

1. Set

- A collection in which elements are sorted according to their own values
- Here duplicates are not allowed

2. Multiset

- Similar to sets except that here duplicates are allowed.

```
1   #include <set>
2   #include <string>
3   #include <iostream>
4   using namespace std;
5   int main()
6   {
7       multiset<string> cities
8       {
9           "Bangalore", "Hanover", "Frankfurt", "New York","Chicago",
10           "Toronto", "Paris", "Frankfurt"
11      };
12      for (const auto& elem : cities)
13          cout << elem << " ";
14      cout << endl;
15      cities.insert( {"London", "Munich", "Hanover", "Bangalore"} );
16      for (const auto& elem : cities)
17      {
18          cout << elem << " ";
19      }
20      cout << endl;
21      return 0;
22  }
```

OUTPUT

```
Bangalore Chicago Frankfurt Frankfurt Hanover New York Paris Toronto

Bangalore Bangalore Chicago Frankfurt Frankfurt Hanover Hanover London
Munich New York Paris Toronto
```

1. Map

- contains elements that are key/value pairs
- Each element has a key that is the basis for the sorting criterion and a value
- Duplicate keys are not allowed

2. Multimap

- Similar to Maps except that here duplicates are allowed.
- Can also be used as dictionary

FACE

```
1   #include <map>
2   #include <string>
3   #include <iostream>
4   using namespace std;
5   int main()
6   {
7       multimap<int,string> coll;
8       coll = { {5,"tagged"},
9                {2,"a"},
10               {1,"this"},
11               {4,"of"},
12               {6,"strings"},
13               {1,"is"},
14               {3,"multimap"} };
15      for (auto elem : coll)
16      {
17          cout << elem.second << ' ';
18      }
19      cout << endl;
20      return 0;
21  }
22
```

OUTPUT

```
this is a multimap of tagged strings
```

Unordered Containers

- Elements have no defined order
- Typically implemented as a hash table
- Advantage : Accessing elements is much faster
- Disadvantage : Consumes a lot of space
- Predefined unodered container are;
    i. Unordered Set
    ii. Unordered Multiset
    iii. Unordered Map
    iv. Unordered Multimap

FACE

1. Unordered Set

- Collection of elements stored in unordered manner
- Here duplicates are not allowed

2. Unordered Multiset

- Similar to sets except that here duplicates are allowed.

FACE

```
1  #include <unordered_set>
2  #include <string>
3  #include <iostream>
4  using namespace std;
5  int main()
6  {
7      unordered_multiset<string> cities
8      {
9          "Bangalore", "Hanover", "Frankfurt", "New York","Chicago",
10         "Toronto", "Paris", "Frankfurt"
11     };
12     for (const auto& elem : cities)
13         cout << elem << " ";
14     cout << endl;
15     cities.insert( {"London", "Munich", "Hanover", "Bangalore"} );
16     for (const auto& elem : cities)
17         cout << elem << " ";
18     cout << endl;
19     return 0;
20 }
21
22
```

OUTPUT

```
Paris Toronto Chicago New York Frankfurt Frankfurt Hanover Bangalore
Munich London Frankfurt Frankfurt New York Braunschweig Bangalore
Chicago Toronto Hanover Hanover Paris
```

3. Unordered Map

- contains elements that are key/value pairs
- Each element has a key that is the basis for the sorting criterion and a value
- Duplicate keys are not allowed

4. Unordered Multimap

- Similar to Maps except that here duplicates are allowed.
- Can also be used as dictionary

FACE

Container Adaptor

- Predefined containers
- Provides a restricted interface to meet special needs
- Implemented by using the fundamental container classes
- Different kinds of container adaptors;
    i. Stack
    ii. Queue
    iii. Priority Queue

FACE

1. Stack

- Manages its elements by the LIFO (last-in-first-out) policy
- To use an Stack, <stack> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T,typename Container = deque<T>>
    class stack;
}
```

- T is the type of elements of an array
- Optional second template parameter defines the container that the stack uses internally for its elements

FACE

```cpp
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> st;
    st.push(1);
    st.push(2);
    st.push(3);
    cout << st.top() << ' ';
    st.pop();
    cout << st.top() << ' ';
    st.pop();
    st.top() = 77;
    st.push(4);
    st.push(5);
    st.pop();
```

```
1    while (!st.empty())
2        {
3            cout << st.top() << ' ';
4            st.pop();
5        }
6        cout << endl;
7        return 0;
8    }
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

OUTPUT

```
3 2 4 77
```

2. Queue

- Manages its elements by the FIFO (first-in-first-out) policy
- To use an Queue, <queue> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T,typename Container = deque<T>>
    class queue;
}
```

- T is the type of elements of an array
- Optional second template parameter defines the container that the stack uses internally for its elements

FACE

```
1   #include <iostream>
2   #include <queue>
3   #include <string>
4   using namespace std;
5   int main()
6   {
7       queue<string> q;
8       q.push("These ");
9       q.push("are ");
10      q.push("four words!");
11      cout << q.front();
12      q.pop();
13      cout << q.front();
14      q.pop();
15      q.push("delete!!");
16      q.push("some");
17      cout << q.front()<< endl;
18      q.pop();
19      cout << q.front()<< endl;
20      cout << "no of elements in the queue: " << q.size();
21      return 0;
22  }
```

OUTPUT

```
These are four words!
delete!!
no of elements in the queue: 2
```

3. Priority Queue

- a queue from which elements are read according to their priority
- To use an Priority Queue, <queue> header file must be included
- There, the type is defined as a class template inside namespace std:

```
namespace std
{
    template <typename T,typename Container = vector<T>,
    typename Compare = less<typename Container::value_type>>
    class priority_queue;
}
```

- T is the type of elements of an array
- Optional second parameter defines the container that the stack uses internally for its elements
- Optional third parameter defines the sorting criterion

FACE

```
1   #include <iostream>
2   #include <queue>
3   using namespace std;
4   int main()
5   {
6       priority_queue<float> q;
7       q.push(66.6);
8       q.push(22.2);
9       q.push(44.4);
10      cout << q.top() << ' ';
11      q.pop();
12      cout << q.top() << endl;
13      q.pop();
14      q.push(11.1);
15      q.push(55.5);
16      q.push(33.3);
17      q.pop();
18
19
20
21
22
```

```
1  while (!q.empty())
2       {
3            cout << q.top() << ' ';
4            q.pop();
5       }
6       cout << endl;
7       return 0;
8  }
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

OUTPUT

```
66.6 44.4
33.3 22.2 11.1
```

Iterators

- Similar to pointers
- Used to access members of the container classes
- Each of the container classes is associated with a type of iterator
- Different types:
    i. Input iterators
    ii. Output iterators
    iii. Forward iterator
    iv. Bidirectional iterator
    v. Random – access iterators

FACE

Types of Iterators

Iterator	Description
input_iterator	Read values with forward movement.
output_iterator	Write values with forward movement.
forward_iterator	Read or write values with forward movement.
bidirectional_iterator	Read and write values with forward and backward movement.
random_iterator	Read and write values with random access.
reverse_iterator	Either a random iterator or a bidirectional iterator that moves in reverse direction.

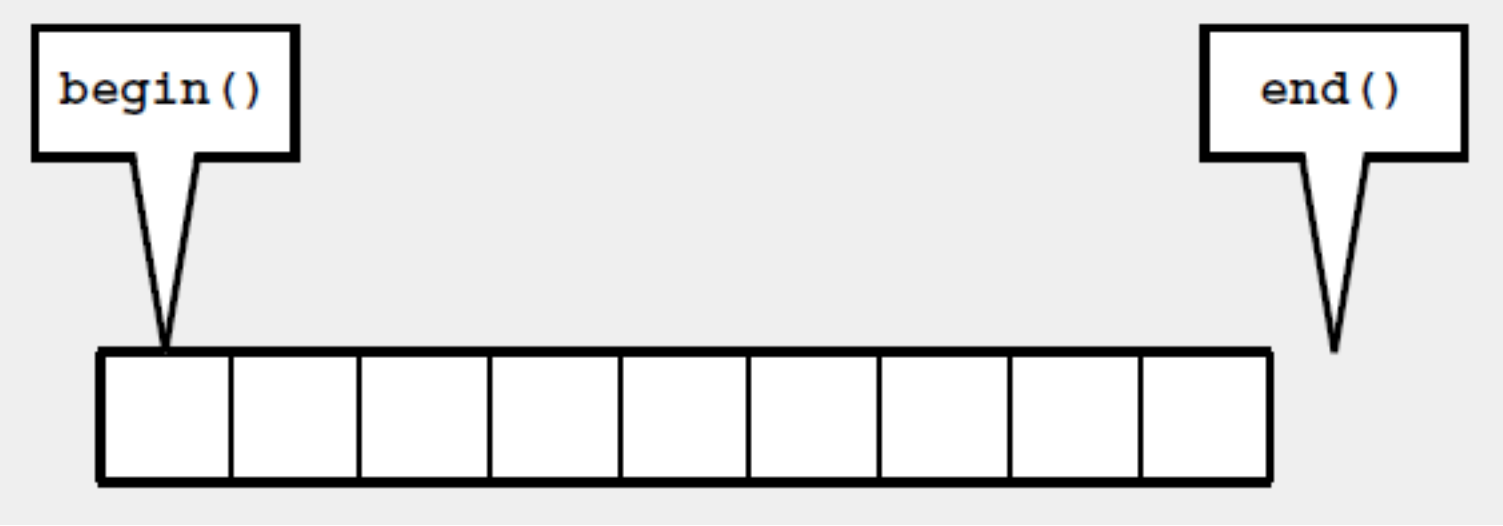
Operation on Iterators

Operator	Description
* ->	Returns the element of the current position If the elements have members, operator -> is used to access those members directly from the iterator
++	iterator step forward to the next element
== and !=	Checks whether two iterators represent the same position
=	assigns an iterator

FACE

Most common function

- Container classes provide member functions that enable them to use iterators, they are
    i. begin(): returns an iterator that represents the beginning of the elements in the container
    ii. end(): returns an iterator that represents the end of the elements in the container.



FACE

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6      vector<int> the_vector;
7      vector<int>::iterator the_iterator;
8      for( int i=0; i < 10; i++ )
9          the_vector.push_back(i);
10         int total = 0;
11         the_iterator = the_vector.begin();
12         while( the_iterator != the_vector.end() )
13         {
14             total += *the_iterator;
15             the_iterator++;
16         }
17         cout << "Total=" << total << endl;
18 }
19
20
21
22
```

OUTPUT

```
Total=45
```

THANK YOU

FACE