Lecture 11

Standard
Template Library

Topics

- Containers
- Iterators
- Built-in Algorithms

Standard Template Library (STL)

The STL of C++ are used for storing and processing data in memory.

- **Containers** are built-in template **template classes.**
  They can be customized to hold different kinds of data.

Sequential Containers	:	vector, list
Associative Containers	:	set, map
Container Adaptors	:	queue, stack

- **Algorithms** are built-in **template functions.**
  They can be applied to containers to process their data.

- **Iterators** are specialized pointers.
  They point to elements in a container.
  An iterator can be incremented, so it points in turn
  to each element in a container.

STL Header Files

- STL containers are located in different C++ libraries.
- To use a container, the necessary header file should be added with the include directive.
- All of the STL classes and functions use the std:: namespace.

```
#include  <vector>
#include  <list>
#include  <set>
#include  <map>
#include  <queue>
#include  <stack>
#include  <iterator>

#include  <algorithm>
```

Template class headers

Template function header

Defining container variables

- Defining a variable of a STL container requires two steps.
- First, an appropriate STL header file is included.
- Then, the template format with the parameter is declared.
- There's no need to specify the size of STL containers.

Define a vector of Course class :

```
#include  <vector>
vector  <Course>   A;       // Variable A
```

Define a list of Course class :

```
#include  <list>
list  <Course>   B;         // Variable B
```

Define a map. The integers are Keys, the strings are Values :

```
#include  <map>
map  <int,  string>   C;    // Variable C
```

Common Member Functions of Containers

Some member functions common to all containers are given below.

Member Function	Description
size ()	Returns the number of items currently in container.
max_size ()	Returns the maximum size of container.
empty ()	Returns true if container is empty.
begin ()	Returns an iterator to the start of container for iterating forward.
end ()	Returns an iterator to the past-the-end location in container.
rbegin ()	Returns a reverse iterator to the end of container.
rend ()	Returns a reverse iterator to the beginning of container.

Sequential Containers

STL Container	Description
vector	Expandable array
list	Doubly linked list

Vectors

- Vectors are similar to plain C arrays, but they are implemented as built-in STL classes.
- Elements of vector can be accessed with the bracket [ ] operator.
- When adding a new element, the vector's size is automatically increased.
- The added items are not kept in sorted order.

- Member functions of vector class:
  - **constructors**
  - **push_back ()**
  - **pop_back ()**
  - **size ()**
  - **operator [ ]**
  - **empty ()**
  - **clear ()**
  - **insert ()**
  - **erase ()**
  - **swap ()**

Example : Adding elements to vectors

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

int main() {
   int  i;
   vector <int>   a;
   a . push_back ( 20 );
   a . push_back ( 10 );
   a . push_back ( 40 );
   a . push_back ( 30 );

   // Access elements with index operator []
   for (i = 0; i < a . size(); i++ )
         cout << a [i] << "  ";
   cout << endl;
```

```
   vector <string>   b;
   b . push_back ("BBB");
   b . push_back ("AAA");
   b . push_back ("CCC");

   for(i = 0; i < b . size(); i++ )
       cout << b [i] << "  ";
} // End of main
```

Screen Output

```
20   10   40   30
BBB   AAA   CCC
```

Range-Based For Loop

- The range-based for loop can be used to iterate over a container.
- For each element of the container, statements inside the code block are executed.

General syntax

```
for  (element_type   element_name   :   container_name)
{
     Statements;
}
```

Example : Accessing vector elements by range-based for loop

- An initializer block is used in vector declaration.
- Elements of vector are accessed with the range-based for loop.

```cpp
#include  <iostream>
#include  <vector>
using  namespace  std;

int main()
{
   // Initializer  block
   vector <int>   a  { 20,  10,  40,  30 };

   // Range-based  for loop
   for  (int  num   :   a)
         cout << num << "  ";
}
```

Screen Output

```
20   10   40   30
```

Example : Using a class as vector template parameter

Part1

```cpp
#include <iostream>
#include  <vector>
using namespace std;

class Product {
  int       id;
  string  name;
  int      amount;
public:
  Product (int i, string n, int a) : id(i), name(n), amount(a) {}   // Constructor
  void print() {
         cout << "ID = " << id
                << "   Name = " << name
                << "   Amount = " << amount << endl;
   }
};  // End of class
```

Part2

```
int main()
{
   vector  <Product>  V;       // Define variable V

   V.push_back ( Product (111, "AAAA", 10) );
   V.push_back ( Product (222, "BBBB", 20) );
   V.push_back ( Product (333, "CCCC", 30) );
   V.push_back ( Product (444, "DDDD", 40) );

   for (auto  prod  :  V)
         prod.print();      // Call member function
} // End of main
```

Screen Output

```
ID = 111    Name = AAAA     Amount = 10
ID = 222    Name = BBBB     Amount = 20
ID = 333    Name = CCCC     Amount = 30
ID = 444    Name = DDDD     Amount = 40
```

Topics

- Containers
- Iterators
- Built-in Algorithms

Iterators

- Iterators are specialized pointers to items in containers.
- Given an iterator variable **iter**, ***iter** represents the object the iterator points to.
- Alternatively, **iter->** can be used to reach the object the iterator points to.
- **++iter** or **iter++** advances the iterator to the next element
  in specific direction.
- There are four types of iterators.

Keyword	Iteration	Operation
iterator	forward	read and write
const_iterator	forward	read only
reverse_iterator	backward	read and write
reverse_const_iterator	backward	read only

Defining iterator variables

- STL containers include their own iterators.
- They produce iterators by using following two member functions.
  begin() : Returns a pointer to the first element.
  end()   : Returns a pointer to one past the last element.

```
vector <string>   a;                     // a is vector of strings

vector <string> :: iterator   i;         // i  is iterator variable

i  =   a . begin ();                     // i  points to first element of a

i++;                                     // i  points to next element
```

- The statement below is a type definition (typedef).
  vector <string> :: iterator

Example : Accessing vector elements by iterator

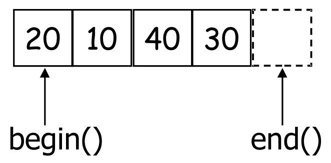
Iterator provides an alternative way to access vector elements.

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main()  {
  vector <int>   a  { 20,  10,  40,  30 };  // Initializer block
  vector <int>   ::   iterator    j;   // Iterator j to vector of integers

   for (j  =  a . begin();   j  !=  a . end();   j++)
        cout <<  *j  << " ";   // * is dereference operator (content)

} // End of main
```



Lists

- A STL list is a **Doubly Linked List** data structure.
- Each element contains a pointer to next element, and also a pointer to preceding element.
- The container stores the address of both the front (first) and the back (last) elements.
- Some member functions of list class:
  - **push_front ()**
  - **pop_front ()**
  - **reverse ()**
  - **merge ()**
  - **unique ()**

Example : Accessing list elements by iterator

Containers such as lists, which don't support direct access, require an iterator to access list elements.

```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;

int main()
{
  int  a [4]  =  { 20,  10,  40,  30 };   // Array as source

  list <int>   L (a,  a+4);      // List is constructor-initialized from array
  list <int>  ::  iterator    j;    // Iterator j to list of integers

  for  ( j  =  L . begin();    j  !=  L . end();    j++)
      cout <<   *j   << "  ";
}
```

Associative Containers

- An associative container is not a sequential structure.
- Instead it uses ***keys*** to access data.
- In all associative containers, keys (numbers or stings) are sorted automatically.

STL Container	Description
map	Associates a key with an element. Only one key of each value allowed (no duplication).
set	Stores only the keys. Only one key of each value allowed (no duplication).

Sets

- The set class implements a set of automatically sorted key values.
- Each value can be stored only once in a set (no duplication).
- Example: The program displays an entire set of cities.

```
#include <iostream>
#include <set>
#include <iterator>
using namespace std;

int main()
{
  set  <string>   city;
  set  <string>  ::  iterator    j;   // Iterator  j  to the set

  // Add city names to the set
  city . insert ("BBB");
  city . insert ("AAA");
  city . insert ("CCC");

  // Display city names in the set
  for (j  =  city . begin();    j  !=  city . end();     j++)
        cout <<  *j  << endl;
}
```

Screen Output

```
AAA
BBB
CCC
```

Maps

- The map class implements an automatically sorted associative array.
- (STL map is automatically sorted by key fields.)

- A map is filled with ***Key / Value*** pairs, which may be of any type.

- The **Key** is used for looking up the information (Value).

- The associated information is the ***Value***.

- For example, a phonebook uses telephone number as the key, and uses names of people as the value.

- Each key (telephone number) can be stored only once in a map.
- If the same key is entered twice, the last entered *key / value* pair is stored.

Example : The find member function of map

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

int main() {
  map <string,  int>    city_plates;

  // Add city names (keys) and plate numbers (values).
  // Overloaded built-in operator[] is used.
  city_plates [ "Bolu" ]  =  14;
  city_plates [ "Adana"]  =  1;
  city_plates [ "Istanbul" ]  =  34;

  string  name;
  cout << "Enter a city name for searching : ";
  cin >> name;

  if ( city_plates . find (name)   ==  city_plates . end()  )
      cout << name << " can not be found" << endl;
  else
      cout << "Plate number of " << name << ": "
           << city_plates [name];
}
```

Accessing map fields:
(With iterator)

- The keyword **first** is used to get the key (city name) field.
- The keyword **second** is used to get the value (plate number) field.

```
#include <iostream>
#include <map>
#include <iterator>
using namespace std;

int  main()
{
 // Block initialization of map
 map  <string,  int>    city_plates  =  { {"Bolu",14} , {"Adana",1} , {"Istanbul",34} };
 map  <string, int>  ::  iterator    j;

 for ( j   =  city_plates . begin();  j  !=  city_plates . end();  j++)
     cout << j -> first << " "
          << j -> second << endl;
} // End of program
```

Screen
Output

```
Adana     1
Bolu      14
Istanbul  34
```

Accessing map fields:
(With range-based for loop)

The built-in first and second fields of a map can be accessed in a loop without using an iterator.

```
#include <iostream>
#include <map>

using namespace std;

int  main()
{
 map  <string,  int>   city_plates  =  { {"Bolu",14} , {"Adana", 1} , {"Istanbul",34} };

 for  (auto   cp    :    city_plates)    // Range-based for loop
      cout <<  cp . first  << "  " <<  cp . second  <<  endl;
} // End of main
```

Screen Output

```
Adana     1
Bolu      14
Istanbul  34
```

Topics

- Containers
- Iterators
- Built-in Algorithms

Built-in STL Algorithms
(STL Functions)

- STL algorithms are built-in template functions.
- They are not members of any template class.
- Algorithms were designed to work with STL containers, but they can be used also for C arrays.
- Some of the STL algorithms:
  - **find ()**
  - **sort ()**
  - **count ()**
  - **copy ()**
  - **replace ()**
  - **nth_element ()**
  - **max_element ()**
  - **min_element ()**

STL max_element function

- Function prototypes:

  Iterator ***max_element*** *(IteratorFirst, IteratorLast);*
  Iterator ***max_element*** *(IteratorFirst, IteratorLast, comparison_function);*
- Function takes parameters the first and last iterators in an array or vector.
- Function returns an iterator (pointer) to the location of maximum value.
- The comparison_function is optional. (For comparision of two items in array).

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

int main()
{
  int a[3] = {10, 30, 20};
  auto  eb = max_element (a,  a+3);
  cout << *eb << endl;
}
```

Screen Output

```
30
```

STL find function

- Function prototype:
  *Iterator **find** (IteratorFirst, IteratorLast, Type const & Value);*
- Value is searched by the iterator between first and last.
- An iterator pointing to the first element found is returned.
- If the element was not found, **null** is returned.

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

int main()
{
  int  a [8]  =  { 10, 20, 30, 40, 50, 60, 70, 80 };  // array for searching
  int*  ptr;  // pointer for the result element
  ptr  =  find (a ,  a+8,  30);    // find data 30 in array
  cout << "Data with value 30 found at index " << (ptr - a) << endl;
}
```

Screen Output

```
Data with value 30 found at index 2
```

STL sort function

- There are two prototypes of the sort function.

- In the prototype below, the elements between first and last are sorted in ascending order (by default).

  *void **sort** (IteratorFirst, IteratorLast);*

- In the prototype below, the elements are sorted in a specified order using the *comparision_function* (i.e., condition function).

  *void **sort** (IteratorFirst, IteratorLast, comparison_function);*

Example : Sorting a vector of integers

```
#include <stdio.h>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  // Initialize vector
  vector <int>  V  =  {20,  10,  40,  30} ;


   // Sort vector from first element until last element.
  sort ( V . begin() ,   V . end() );

  printf ("Sorted vector V : \n");
  for (int num  :  V)     // Range-based loop
       printf ("%d \n", num );
}
```

Screen Output

```
Sorted vector V :
10
20
30
40
```

Example : Sorting a vector of strings

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
// Comparison function is used for sorting two strings by number of characters.
bool  condition_for_sorting (string x, string y) {
     return  x.length()  <  y.length();
}
//-------------------------------------------------------------------------------------------------
int main() {                          // Block initialization of vector
   vector <string>  isimler   { "EEEE", "CCCCCC", "BBBBBBBB", "AAAAA", "DD" };

   sort ( begin (isimler), end (isimler) );
   cout << "Names sorted alphabetically:    ";
   for  (string  ad  :  isimler)   cout << ad << "  ";   // Range loop
   cout << endl;
  //--------------------------------------------------------------------
   sort  ( begin (isimler),   end (isimler),  condition_for_sorting );  // Use comparison function
   cout << "Names sorted by number of chars:    ";
   for  (string  ad  :  isimler)   cout << ad << "  ";   // Range loop
   cout << endl;
}
```

Screen Output

```
Names sorted alphabetically:     AAAAA  BBBBBBBB  CCCCCC  DD  EEEE
Names sorted by number of chars: DD  EEEE  AAAAA  CCCCCC  BBBBBBBB
```