

OIJ-1156 Programming II

Standard Template Library

Imed Hammouda

Department of Software Systems

Tampere University of Technology

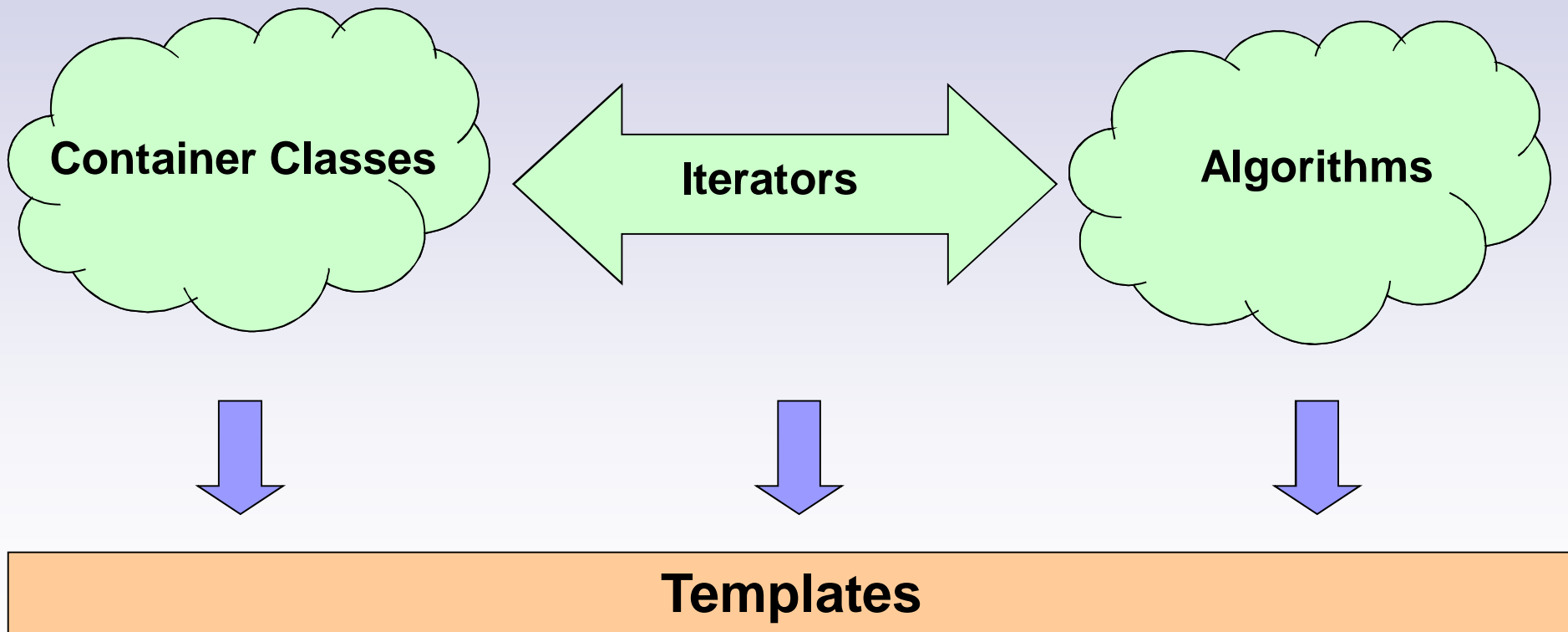


Objectives

- In this lecture you will learn:
 - What is **STL**.
 - What are the **elements of STL**.
 - How to **use STL**.



Standard Template Library (STL)



Standard Template Library (STL)

- A library of **class** and **function templates** based on work in generic programming done by Alex Stepanov and Meng Lee of the Hewlett Packard Laboratories in the early 1990s.
- STL has three components:
 - **Containers**: Generic "off-the-shelf" class templates for storing collections of data
 - **Algorithms**: Generic "off-the-shelf" function templates for operating on containers
 - **Iterators**: Generalized "smart" pointers that allow algorithms to operate on almost any container



STL's Containers

- In 1994, STL was adopted as a **standard part of C++**
There are **10** containers in STL
- **Sequential**: data items are **arranged** into a list so that there is a **first** element, a **next** element, and forth up to a **last** element
- **Associative**: stored data items have an associated value known as **key**
- **Adapters**: template classes that are implemented **on top of other classes**



STL's Containers

Kind of Container

STL Containers

Sequential:

deque, list, vector

Associative:

map, multimap, multiset, set

Adapters:

priority_queue, queue, stack

Non-STL:

bitset, valarray, string



The Vector Class

- The vector class was designed to be a "better" array class
- The vector class is an **ADT**
 - We are not concerned with how vectors are stored
 - We are concerned only with how to manipulate vectors
- The vector class is a **template class**
 - When a vector object is declared, we must specify the **type** of object to be stored in the vector
 - Form:

```
vector <type> name;
```
 - Example:

```
vector <int> v;
```



Vector Constructors

- Default constructor

Creates a vector containing no objects.

```
vector<int> v;
```

- Copy constructor

Creates a vector which is a copy of an existing vector.

```
vector<int> u = v;  
vector<int> w(v);
```

- Other constructor

Create a vector of a specified size

```
vector<int> v(100); // Set size
```

Create an initialized vector of a specified size

```
vector<char> u(100, 'a'); // Initialize
```



Vector Class Member Functions

- Get the number of elements in the vector. Returns the size:

```
v.size()
```

- See if the vector is empty. Returns true if vector is empty:

```
v.empty()
```

- Change the size of the vector:

- Changes the size of the vector to the specified size

- If the new size is less than the current size, then members are lost
- If the new size is greater than the current size, then the additional positions are initialized to the specified value
- If no value is specified, then the default constructor is used

```
v.resize(5);
```

```
v.resize(5, 123);
```



Vector Class Member Functions

- Make a vector empty. Removes all the members from the vector, leaving it empty:

```
v.clear();
```

- Swap two vectors. Swaps this vector with the specified vector:

```
v.swap(u);
```

- Assign one vector to another. Assigns the specified vector to this vector:

```
v = u;
```



Vector Class Random Access Methods

- Reference an element of the vector. Returns a reference to the object in position i :

```
v[i]
```

- Reference an element of the vector. Returns a reference to the object in position i :

```
v.at(i)
```

- Reference the first element of the vector. Returns a reference to the first member in the vector:

```
v.front()
```



Vector Class Random Access Methods

- Reference the last element of the vector. Returns a reference to the last member in the vector:

```
v.back()
```

- Remove the last element of the vector. Removes the last member of the vector:

```
v.pop_back();
```

- Add a new element onto the end of the vector. Appends a copy of the new value as the last member of the vector:

```
v.push_back(123);
```



Vector Class Example

/* This is a sample example on how to declare and use two-dimensional vector
It should clear how to declare it!
the member function push_back allocates memory, so you don't have to deal
with memory management

```
*/  
#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
int main()  
{  
int i,j;
```

```
/* Declaring an empty integer vector, i.e. a vector of integers of size 0*/  
vector<int> EmptyVector;
```



Vector Class Example

```
/* Declaring an empty vector of interger vectors, i.e. a two-dimensional vector*/
vector< vector<int> > TwoDimVector;

/* displaying the size of the TwoDimVector which is 0 */
cout << "Size of TwoDimVector is:" << TwoDimVector.size() <<endl;

/* Changing the size of TwoDimVector, by allocating five empty integer vectors*/
for(j=0;j<5;j++)
    TwoDimVector.push_back( EmptyVector );

/* displaying the size of the TwoDimVector which is 5 */
cout << "Size of TwoDimVector is:" << TwoDimVector.size() <<endl;

/* displaying the sizes of the 5 empty integer vectors, which are all 0*/
for(i=0;i<5;i++)
    cout << "Size of Empty Integer Vector " << i << " is :" <<
    TwoDimVector[i].size() << endl;
```



Vector Class Example

```
/* changing the sizes of the empty integer vectors to be 7, and storing some values in the
   allocated memory*/
for(j=0;j<5;j++)
{
    for(i=0;i<7;i++)
        TwoDimVector[j].push_back(i+j*7);
}

/* Displaying the contents of the two dimensional vector*/
for(j=0;j<5;j++)
{ for(i=0;i<7;i++)
    cout << TwoDimVector[j][i] << "    ";
    cout << endl;
}
return 0;
}
```



The List and Deque Classes

- `deque<type>` : Dynamic at the beginning and the end
- `list <type>` : Dynamic, fast in any location
- list and deque are defined in their own libraries
 - `#include <list>`
 - `#include <deque>`
- The interfaces are almost identical to vector, the internal **implementations** and **efficiency** of the containers differ
- They have special member functions **push_front** and **pop_front**



The Set Class

- Set is the simplest container, defined in the **library set**
- No duplicates are allowed
- For example:
`set <string> passwords; // Storing all passwords`
- Example member functions (s is a set object):
 - `s.insert(Element)` // inserts Element, no effect if Element exists
 - `s.erase(Element)` // removes Element, no effect if Element does not exist
 - `s.find(Element)` // returns a pointer to Element, or end if it does not exist
 - `s.size()` // returns the number of elements
 - `s.empty()` // returns true if empty, otherwise false
 - `s1 == s2` // returns true if s1 and s2 contain same elements, otherwise false



The Multiset Class

- Multiset allows duplicate, defined in the **library set**
- For example:
`multiset <string> students; // Storing all students`
- The member function **count** returns the number of elements
`int johns = students.count("John");`



The Map Class

- map and multimap are both defined in the **library map**
- The elements of map and multimap are **key/value pairs**
- For example:

```
map <string, int> phonebook;
```

- In map keys are unique and map can be indexed with the key in the same way as sequential containers:

```
phonebook [ "Tim" ] = 123456;
```

```
phonebook.insert(make_pair( "John" , 654321 ));
```

```
cout << "Tim's phone number is: " << phonebook[ "Tim" ] <<endl;
```



The Map Class

- Example member functions (m is a map object):
 - `m.insert(Element)` // inserts Element in the map
 - `m.erase(Target_key)` // removes Element with key Target_key
 - `m.find(Target_key)` // returns a pointer to the element with key Target_key
 - `m.size()` // returns the number of pairs
 - `m.empty()` // returns true if the map is empty, otherwise false
 - `m1 == m2` // returns true if the m1 and m2 contain the same pairs, otherwise false



The Multimap Class

- multimap allows **duplicates**
- multimap cannot be indexed with []
- To add a new pair to a multimap, a function `make_pair` needs to be used

- For example:

```
multimap <const string, int> multi_phonebook;  
multi_phonebook.insert(make_pair("Tim", 123456));
```

- Using `pair` (defined in **utility** library):

```
pair <const string, int > name_pair  
= make_pair("Tim", 123456);  
multi_phonebook.insert(name_pair);
```

- The member function **count** returns the number of elements



The Stack and Queue Classes

- `stack` is a template class for implementing **LIFO** structures: Last In, First Out
- `stack` is defined in **stack library**

- `queue` is a template class for implementing **FIFO** structures: First In, First Out
- `queue` is defined in **queue library**

- `priority_queue` is like a queue with the additional property that each entry is given a priority
- `priority_queue` is defined in **queue library**



Algorithms

- The STL library **algorithm** provide algorithms that can be used with (almost) all STL containers
- Example algorithms:
 - **copy (first, last, target)**: copies the elements in the range to the target
 - **count (first, last, value)**: counts the elements matching value in the range
 - **find (first, last, value)**: find the value from the range of elements
 - **fill (first, last, value)**: assigns the value to all the elements in the range
 - **for_each(first, last, Function)**: calls function for all elements in range
 - **max_element(first, last)**: finds the largest element in the range
 - **min_element(first, last)**: finds the smallest element in the range
 - **reverse(first, last)**: reverses the elements in the range
 - **sort(first, last)**: sorts the elements in the range in ascending order



Iterators

- Iterators are the **link** between the **containers** and the **algorithms**
 - They “**iterate**” over the data elements
 - Iterators can be used to “**point**” to the **elements** of a container and the elements can be handled through them
 - For example:
 - The container member functions insert and erase take an iterator to the wanted location in the container as a parameter
- ```
vector <int> v(2,1); // [1,1]
v.push_back(3); // [1,1,3]
v.insert(v.begin(), 2); // [2,1,1,3]
v.insert(v.end(), 1); // [2,1,1,3,1]
v.insert(v.begin()+2, 3); // [2,1,3,1,3,1]
v.erase(v.begin()); // [1,3,1,3,1]
v.erase(v.begin(), v.end()); // [empty]
```



# Iterators

- The member function `begin` returns an iterator to the first element, and `end` returns an iterator to the position after the last element
- Be careful with the end-iterator. Iterators don't **point** to the elements in the container but **between the elements**
- For example: removing the last element  
`v.erase(v.end()-1);`



# Iterators

## ■ Operators:

- \*: accessing the data
- ++: moving forward by one step
- --: moving backward by one step
- +, -: moving forward and backward
- ==, !=: whether two iterators point to the same data location, advance

## ■ Iterator types:

- `vector<int>::iterator`
- `list<int>::iterator`
- ...

## ■ For example:

```
vector <int> v;
v.push_back(1);
v.push_back(2);
for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
 cout << *i << endl;
```



# Iterators

- Example:

```
vector<int> container;
container.push_back(1);
container.push_back(2);
container.push_back(3);
vector<int>::iterator it = container.begin();
cout << container[2] << endl; // 3
cout << it[2] << endl; // 3
cout << *(it +2) << endl; // 3
*it=5;
cout << container[0] << endl; // 5
```



# Iterators

- If the elements are not supposed to change through the iterator, an iterator type `const_iterator` can be used instead

- Example:

```
vector<int> container;
container.push_back(1);
container.push_back(2);
container.push_back(3);
vector<int>::const_iterator it = container.begin();
cout << container[2] << endl;
cout << it[2] << endl;
cout << *(it +2) << endl;
*it=5; // illegal
cout << container[0] << endl;
```



# Iterators

- For a container with bidirectional iterators, there is a way to reverse everything using a **reverse iterator**

- Example:

```
vector v<int>;
```

```
...
```

```
for (vector<int>::reverse_iterator i = v.rbegin(); i
 != v.rend(); ++i)
```

```
 cout << *i << endl;
```



# Iterators

- Example of the **find** and **reverse** algorithms

```
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
vector<int>::const_iterator where;
where = find(v.begin(), v.end(), 2);
```

```
vector<int>::const_iterator p;
for (p = where; p != v.end(); ++p)
 cout << *p << endl;
```

```
reverse(v.begin(), v.end());
for (p = v.begin(); p != v.end(); ++p)
 cout << *p << endl;
```

```
vector<int>::iterator q = v.begin()+1;
v.insert(q, 4);
```

```
for (p = v.begin(); p != v.end(); ++p)
 cout << *p << endl;
```



# Iterators

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
 map<string, int> phonebook;

 phonebook["Mary"] = 111111;
 phonebook.insert(make_pair("Anna", 222222));
 pair<string, int> p;
 p = make_pair("John", 333333);
 cout << p.first << p.second << endl;
 phonebook.insert(p);

 if (phonebook.find("Mary") != phonebook.end())
 cout << "Mary is found!" << endl;
 for (map<string, int>::iterator i= phonebook.begin(); i != phonebook.end(); ++i){
 p = *i;
 cout << p.first << " " << p.second << endl;
 }
 return 0;
}
```

