

Week 07

DM2233 ADVANCED DATA STRUCTURES & ALGORITHMS

Module Schedule

Week	Lecture	Remarks
1	Overloading and Templates I	
2	Overloading and Templates II	Labour Day (Fri) – Lab 2 Make up on 27-Apr
3	Overloading and Templates III	
4	Overloading and Templates IV	
5	Exception Handling I	
6	Exception Handling II	
7	Standard Template Library / Assignment 1	Vesak Day (Mon)
Week 8 and 9: Mid-Sem Break		
10	Sorting and Searching I	
11	Sorting and Searching II	
12	Sorting and Searching III	
13	Binary Tree I	Hari Raya Puasa (Fri)
14	Lab Test	
15	Binary Tree II	
16	Binary Tree III	SG50 Day (Fri)
17	Preprocessing / Assignment 2	National Day (Mon)

Template re-visited

- ⦿ Templates allows us to write one set of codes for different data types
- ⦿ Function template and class templates
- ⦿ When you instantiate a class template, you will need to give a data type
 - Recall: Calculator<float> myCalFloat;

Introduction to Standard Template Library (STL)

- In Week 4, template classes were introduced
- Promotes code reusability
- STL consists of template classes
- Prewritten and included in ANSI/ISO standard
- Provides great flexibility in the types supported as elements

How STL helps

- ⦿ The main objective of a code is to manipulate data and generate results
- ⦿ Achieving this goal requires the ability to store data into memory, access a particular piece of data and write algorithms to manipulate the data
- ⦿ STL is equipped with some standard features that support the above

How STL helps

- ④ The container manages the storage space for its elements. Programmers need not worry about the internal mechanism
- ④ Some containers replicate very commonly used data structures such as linked list, queues, stacks, etc
- ④ Have functions that are commonly across all containers (standardization)

How STL doesn't help

- STL as the name goes, provides only standard features, though its offering is quite large. If feature cannot be found, programmer still have to code his/her own
- On certain platforms, STL may not be available, we have a problem with the porting of the code

3 Main Components

- ⦿ STL has 3 main components
 - Containers
 - Iterators
 - Algorithms
- ⦿ Containers and Iterators are class templates
- ⦿ Iterators are used to step through the elements of a container
- ⦿ Algorithms are used to manipulate data

Container Types

- ⦿ Containers are used to manage objects of a given type
- ⦿ STL containers are classified into 3 categories:
 - Sequence containers (also called sequential containers)
 - Associative containers
 - Container adaptors

Sequence Containers

- ⦿ Every object in a sequence container has a specific position.
- ⦿ 3 predefined sequence containers are:
 - vector
 - deque
 - list

vector

- ⦿ A vector container stores and manages its objects in a dynamic array
- ⦿ It can be accessed randomly, like an array
- ⦿ `#include <vector>`
- ⦿ To define an object of this class, since it is a template class, we need to specify its data type

Declaring vector object

```
vector<data type> name_of_list;
```

Declaring vector object

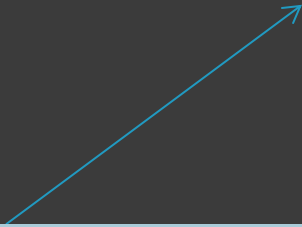
⦿ There are a few ways to declare vector objects:

- `vector<int> vecList; //default`
- `vector<int> vecList(othervecList); //use other
//vector object`
- `vector<int> vecList(10); //fixed size`
- `vector<int> vecList(100, 5); //all elements to 5`


Declaring vector object

- Remember this?

```
vector<GameObject *> m_goList;
```



This list will contain
pointers to GameObjects.



Name of your list

Manipulate vector object

- ⦿ Basic operations:
 - Item insertion
 - Item deletion
 - Stepping through vector elements
- ⦿ The following operations are available:
 - `vecList[index]` //get the vector element at index
 - `vecList.at(index)` //similar to `vecList[index]`
 - `vecList.front();` //get front element, no empty
//check
 - `vecList.back();` //get last element, no empty
//check

Manipulate vector object

Example

```
vector<int> myList(5);
```

```
for(int i = 0; i < 5; i++)  
{  
    myList[i] = i;  
}
```


Manipulate vector object

Example

```
vector<int> myList(5);
```

```
for(int i = 0; i < 5; i++)  
{  
    myList[i] = 5-i;  
}
```

```
cout << myList.capacity() << " " << myList.size() << endl; //5 5  
myList.push_back(45); //insert at the back  
cout << myList.capacity() << " " << myList.size() << endl; //7 6
```

```
for(int i=0; i<myList.size(); i++)  
{  
    cout << myList[i] << " "; //5 4 3 2 1 45  
}
```

Manipulate vector object

- ⦿ `vecList.capacity();`
- ⦿ `vecList.empty();`
- ⦿ `vecList.size();`
- ⦿ `vecList.max_size();`
- ⦿ `vecList.clear();` //delete all elements
- ⦿ `vecList.erase(3);` //erase element at index 3
- ⦿ `vecList.insert(pos, 23);` //insert 23 at pos
- ⦿ `vecList.push_back(11);` //insert 11 at the end
- ⦿ `vecList.pop_back();` //delete last element
- ⦿ `vecList.resize(50);` //resize to fit 50 elements

Iterator

- Similar to a pointer, inherit pointer operations
- Use to run through the elements of the vector
- Type dependent, have to use the correct iterator for the **vector** object

```
vector<int> myList(5);  
vector<int>::iterator intVecIter;
```

Iterator

- Every container has member functions **begin()** and **end()**

```
for(int i = 0; i < 5; i++)  
{  
    myList[i] = 5-i;  
}
```

```
intVecIter = myList.begin();  
++intVecIter;  
myList.insert(intVecIter, 22);
```

```
for(intVecIter = myList.begin(); intVecIter !=  
    myList.end(); ++intVecIter)  
{  
    cout << *intVecIter << " "; //5 22 3 2 1  
}
```

Iterator vs Indexing

```
vector<int> myList(5,3);

for(int i = 0; i < 5; i++)
{
    cout << myList[i] << " ";
}
```

```
vector<int> myList(5,3);
vector<int>::iterator intVecIter;

for(intVecIter = myList.begin();
    intVecIter != myList.end();
    ++intVecIter)
{
    cout << *intVecIter << " ";
}
```

Iterator vs Indexing

Iterator

- More complex
 - But it supports more algorithm
- Works with all containers
- More efficient for random access
- Etc..

More generic!

Indexing

- Simple to understand
 - May not be, remember we sometimes do fall into the trap of start the indexing at 1 instead of 0?
- Only works for sequential containers
- Etc..

deque

- ⦿ Stands for **d**ouble-**e**nded **q**ueue
- ⦿ Elements can be added at both ends
- ⦿ Provide similar functionalities to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence and not only at its end

deque

- While vectors are very similar to a plain array that grows by reallocating all of its elements in a unique block when its capacity is exhausted, the elements of a deque can be divided in several chunks of storage, with the class keeping all this information and providing a uniform access to the elements

deque

- ⦿ Some function member differences from vector
 - `push_front()`
 - `pop_front()`

list

- ⦿ A doubly linked list system
- ⦿ list performs generally better in inserting, extracting and moving elements in any position within the container, and therefore also in algorithms that make intensive use of these, like sorting algorithms
- ⦿ Elements cannot be referenced directly by position (index), time is needed to traverse

Associative Containers

- ⦿ Elements in associative containers are automatically sorted accordingly to some ordering criteria
- ⦿ Default ordering is $<$ (less than), ascending order
- ⦿ A new element is always placed in the proper place accordingly to the ordering criteria

Associative Containers

- ⦿ The predefined associative containers in the STL are:
 - Sets
 - Multisets
 - Maps
 - Multimaps
- ⦿ The difference between set and multiset is that multiset allows duplicates while set does not

Set

- ⦿ Set is a kind of associative container that stores unique elements, and in which the elements themselves are the keys (hashing)
- ⦿ Efficient access of element
- ⦿ No duplicated elements allowed
- ⦿ Internally, the elements in a set are always sorted from lower to higher following a specific strict weak ordering criterion set on container construction

Set

```
set<string> mySet;
set<string>::iterator it;

mySet.insert("John");
mySet.insert("Apple");
mySet.insert("Zoe");
mySet.insert("Linda");
mySet.insert("Apple");
mySet.insert("Jason");

for (it = mySet.begin(); it != mySet.end(); ++it)
{
    string temp;
    temp = *it;
    cout << temp << ", ";
}
// Apple, Jason, John, Linda, Zoe
```

Map

- ◉ Map is a kind of associative containers that stores elements formed by the combination of a key value and a mapped value
- ◉ In a map, the key value is generally used to uniquely identify the element, while the mapped value is some sort of value associated to this key. Types of key and mapped value may differ
- ◉ Internally, the elements in the map are sorted from lower to higher key value following a specific strict weak ordering criterion set on construction

Map

```
map<string,string> myMap;
```

```
myMap["John"] = "13x";
```

```
myMap["May"] = "14x";
```

```
myMap["June"] = "15x";
```

```
myMap["David"] = "16x";
```

```
myMap["Amy"] = "17x";
```

```
cout << "David's admin no is: " << myMap.find("David")->second << endl;
```


Summary

- Introduction to STL
- Use of vector and Iterators
- Introduction into deque, list and associative containers, set and map

Links for Reading

- ◎ <http://www.cplusplus.com/reference/stl/>
- ◎ http://en.wikipedia.org/wiki/Standard_Template_Library