CSCI 3110
Standard Template Library **Lecture notes**

The **Standard Library** is a fundamental part of the C++ Standard. It provides C++ programmers with a comprehensive set of efficiently implemented tools and facilities that can be used for most types of applications.

Standard Template Library (STL)

- Standard library of data structures. It provides most of the classic data structures, such that creating complex programs can be made relatively easy.
- Any compiler that claims to support "standard C++" must have STL.
- Terms to know about STL
  - **Container**
  - **Iterator**
  - **Adaptor**

Container class is a class where an object of this type holds other objects.

- **Container classes:**

Sequence containers:

vector	rapid insertion and deletions at back Direct access to any element
deque	rapid insertions and deletions at front or back Direct access to any element
list	doubly linked list, rapid insertion and deletion anywhere

Associative containers:
Set, multiset, map, multimap

Container adaptors:

Stack	last in first out
Queue	first in first out
Priority queue	highest priority element is always the first element out

- **Containers are implemented via template class definitions.**

```
template <class T, class A=allocator<T>>
class vector     // or class list  {
public:
   …  //  member functions
};
```

- **Vector container class**: contains fixed-length group of elements of uniform type. It is similar to array, but the size of the vector can be dynamically increase/decreased (copy of existing data is needed when dynamically increasing memory).

```
// C++ array does not perform boundary check
/// how about vector container class? Yes, with aVector.at(i)
//                          direct subscripting does not do boundary check
```

- **deque container class** : an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. As opposed to `std::vector`, the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays, with additional bookkeeping, which means indexed access to deque must perform two pointer dereferences, compared to vector's indexed access which performs only one. Expansion of a deque is cheaper than the expansion of a `std::vector` because it does not involve copying of the existing elements to a new memory location. On the other hand, deques typically have large minimal memory cost
    - Direct access is cheaper with vector
    - Memory size expansion is faster with deque
    - Smaller amount of data is more suitable with vector
    - Deque is double ended

- **list container class** : doubly linked list, performs efficient insertion/deletion operations. (number of elements in a list cannot be bounded)

Example:

```
int main()
{
        vector <int> aVector(100);
        // vector<int>aVector;
        // vector<int>aVector(100, 0);

        vector <sellerClass> sellers;  // default constructor called for each element
        int i=0;
        while (cin)
        {
                cout << "Enter an integer value :";
                cin >> value;
                aVector[i++] = value;
                        // aVector.at(i++) = value;  throw out-of-range exception if …

                if ((i+1)%100 == 0)
                        // incrementally obtaining more space as it goes
                        aVector.resize(aVector.capacity()*2);
        }
        …
}
```

Or, a better approach is:

```
        vector <int> aVector;
        aVector.reserve(1000);  // reserve a large amount of space to start
                                // reserve only, no initialization

        aVector.resize(aVector.size());  // reduce the amount of space to what is needed
```

(b)

```
class NumClass
 {
  public:
        NumClass(double d) {value=d;}  // ← no default constructor
```

```
        …
  private: double value;
};

int main()
{
        vector <NumClass> v1(1000);  // ← ?? compilation error
        vector <NumClass> v2(200, NumClass(0.0));
        …
}
```

- container object is typically passed to function by reference

```
void f1(vector <int>&);
void f2(const vector <int> &);
```

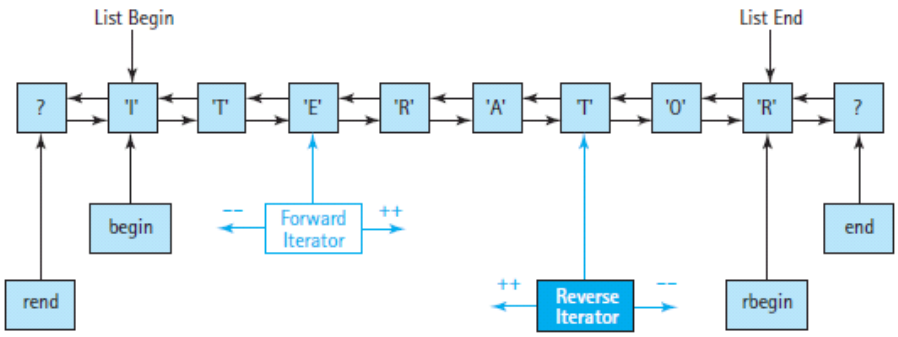
- vector operations: push_back(dataType), pop_back() and front(), back() member functions

```
void f(vector <char>&s) {
        s.push_back('a');
        s.push_back('b');
        s.push_back('c');
        s.pop_back();
        if (s[s.size()-1] != 'b') error("impossible!");
        s.pop_back();
        if (s.back()!='a')  error("should never happen!");
}
```

- Examine deque example code (deque.cpp)

- **An iterator** is a pointer-like object (that is, an object that supports pointer operations) that is able to "point" to a specific element in the container. Often used to cycle through the objects in a container.
    - **Iterator is container-specific**

```
vector<int>::iterator   current;
list <employType>::iterator emIte;
```



    - **Container-specific Iterators share the same set of operations:**
      begin(), end(), increment, decrement, dereferencing.
      This makes iterator operation generic across container types

```
vector<int>::iterator   current;
```

```
for (current=aVector.begin(); current!=aVector.end(); current++)
        sum += *current;
…
```

- declare an **iterator** for a **vector** of integers (i.e., a pointer that will "point" to integers contained in a **vector** object),
- initialize it "pointing" to the first element in values. Given that values is now an object and not a standard array, we need to ask that object to give us a "pointer" to the first element (more exactly, to give us an **iterator** that is "pointing" to the first element). This is done with the member-function **begin()** , which is provided by all the containers.
- The value of the **iterator** pointing to one past the end of the array is also provided by the container object, via the member-function **end()**.
- Use the ++ operator to advance ("point") to the next element.
- directly dereference current to obtain the value. Remember that **current** is an object of type **vector<int>::iterator** . The unary * operator internally does something similar to **current→value**. (But the dereferencing operation is totally transparent for us, and we always use directly the * dereferencing operator.)
- iterators may be compared for equality
- The iterator mechanism applies to list container class in the same way.
- Additional iterator related:
  - const_iterator
  - reverse_iterator, const_reverse_iterator
  - rbegin() → points to the last element
  - rend() → points to a position before the first element

Sequence containers, and iterators of sequence containers share many common member functions

	Description	**vector**	**list**	**deque**
container()	empty container	yes	yes	yes
container(n)	n elements with default values	yes	yes	yes
container(n, x)	n copies of x	yes	yes	yes
container(first, last)	initial elements from first to last	yes	yes	yes
container(x)	copy constructor	yes	yes	yes
~container()		yes	yes	yes
empty()	true if no element	yes	yes	yes
size()	number of elements	yes	yes	yes
=, <=, <, >=, >, ==, !=		yes	yes	yes
capacity()	total reserved memory	yes	no	yes
resize()	change to a new size	yes	no	yes
assign()	constructor alternative assign values to objects in the container	yes	no	yes
front()	first element	yes	yes	yes
back()	last element	yes	yes	yes
[]	subscripting	yes	no	yes
at()	subscripting	yes	no	yes

push_back()	add to end	yes	yes	yes
pop_back()	remove last element	yes	yes	yes
push_front()	add new first element	no	yes	yes
pop_front()	remove first element	no	yes	yes
// *p, first* and *last* are iterators				
insert(p, x)	add x before p	yes	yes	yes
insert(p, n, x)	add n copies of x before p	yes	yes	yes
erase(p)	remove element at p	yes	yes	yes
erase(first, last)	erase[first:last]	yes	yes	yes
clear()	erase all elements	yes	yes	yes

Common methods are provided to different sequence containers so that they are logically interchangeable. The choice of container is dependent on the problem to be solved.

- If the data set requires a lot of random accessing ➔ vector; if the data set requires much insertion/deletion operation ➔ list
- Deque provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes *undefined behavior*.

Examples :

```
class ItemClass
{
public: …
        bool operator < (const ItemClass & rhs);
private:
        string productName;
        float   price;
        int     number;
};

bool ItemClass::operator < (const ItemClass & rhs)
{
        return (productName < rhs.productName)
}
```

```
#include <list>
using namespace std;
int main()
{
    list <char> aCharList;
    list<ItemClass> myList;  // ItemClass is a user-defined class

    ItemClass  newItem, secondItem;

    while (cin >> newItem) // ← overloaded >> for ItemClass
    {
        myList.push_front(newItem);
    }
```

```
        Display(myList);
        myList.sort();   // using the overloaded < operator for the elements in container
        Display(myList);

        cin >> secondItem;

        list<ItemClass>::iterator iter;
        iter=Find (myList, secondItem);
        if (iter != myList.end())
              myList.erase(iter);

        /* or :
        cin >> thirdItem;
        myList.insert(iter, thirdItem);
        thirdItem is inserted before the second Item */
        …
    }
```

Traverse a list:

```
void Display(const list<ItemClass>& aList)
{
        list<ItemClass>::const_iterator  itr1;

        for (itr1 = aList.begin(); itr1!=aList.end(); iter1++)
                cout << *itr1 << endl;     // overloaded << for ItemClass

        // print in reverse
        list<ItemClass>::const_reverse_iterator  itr2;
            for (itr2=aList.rbegin(); itr2!=aList.rend(); iter2++)
                cout << *iter2 << endl;
}

//assuming the != operator has been overloaded for ItemClass
list<ItemClass>::iterator   Find (list<ItemClass>& aList, const ItemClass & value)
{
        list<ItemClass>::iterator itr;
        itr = aList.begin();
        while (itr != aList.end()&&*itr != value ) // operator != overloaded for ItemClass
        {
                itr++;
        }
        if (itr == aList.end())
                cout << "Item not found";

        else
                cout << "value found";

        return itr;
}
```

```
//assuming the < operator has been overloaded for ItemClass
void InsertInOrder(list<ItemClass>&aList, ItemClass &value) // aList can not be passed as
"const" parameter, why?
{
        list<int>::iterator itr = aList.begin();
        while ((itr != aList.end()) && (*itr < value)   // operator < overloaded for ItemClass
                itr ++;
        aList.insert(itr, value);
}
```

Example: define a sorted list of integers by inheriting from the STL list

File : sortedList.h

```
#ifndef SORTEDLIST
#define SORTEDLIST
#include <list>
using namespace std;

class sortedList: public list<ItemClass> {
public:
        // adds item to the list maintaining ascending order
        void insertInOrder(ItemClass& item);

        // print the list one item per line
        void traverse();
}
#endif
```

file : sortedList.cpp

```
#include "sortedList.h"
#include <iostream>
using namespace std;

void sortedList::insertInOrder(ItemClass & item)
{
        list<ItemClass>::iterator itr = begin();
        while (itr!=end() && *itr < item)   // < operator overloaded
                itr++;
        insert(itr, item);
        return;
}

void sortedList::traverse() {
        list<ItemClass>::iterator iter;

        // handle an empty list
        if ((empty())
                cout << "The list is empty" << endl;
        else    {
```

```
                // print list one item at a time
                cout << "The list contains: " << endl;
                for (itr=begin(); itr!=end(); itr++)
                        cout << *itr << endl;
        }
        return ;
}
```

file : client program

```
#include<iostream>
#include"sortedList.cpp"
using namespace std;
int main()
{       sortedList   myList; // the sorted list container

        while (cin)       {
                cin >> product >>price >> number;
                ItemClass newItem(product, price, number);
                myList.insertInOrder(newItem);
        }

        // traverse the list
        myList.traverse();
        return 0;
}
```

- **Adaptor : stack, queue**
  - not separate containers, but implemented as adaptors of basic containers
  - a container adapter provides a restricted interface to a container
    - no iterators
    - only limited member functions are available through specialized adaptor interface

stack adaptor

```
template <class T, class C=deque<T>>
class stack {
protected:
        C    c;
public:
        typedef   typename  C::value_type   value_type;
        typedef   typename  C::size_type      size_type;
        typedef   C   container_type;

        explicit  stack(const  C&a = C()) : c(a) {}. // does not allow auto type conversion

        bool empty() const  {return c.empty();}
        size_type size() const {return c.size();}

        value_type & top() {return c.back();}
        const value_type & top() const {return c.back();}
```

```
        void push(const value_type&x) {c.push_back(x);}
        void pop() {c.pop_back();}
};
```
/* stack is simply an interface to a container of the type passed to it as a template argument. All stack does is to eliminate the non-stack operations on its container from the interface and give back(), push_back(), and pop_back() their conventional name : top(), push(), and pop() */

To create objects of stack type:

```
        stack<char> s1;
        stack<int, vector<int>> s2;
```