The Standard Template Library (STL)

ชวลิต ศรีสถาพรพัฒน์
แก้ไขปรับปรุงจากต้นฉบับของ อ.สุธี สุดประเสริฐ

Credits

- The C++ Programming Language (Third Edition) - Bjarne Stroustrup
- The Standard Template Library Tutorial - Johannes Weidl
- C++ References - http://www.cprogramming.com/tutorial/references.html
- Standard Template Library Programmer's Guide - http://www.sgi.com/tech/stl/

STL คืออะไร?

- The standard C++ library
  - general class, function, template, etc.
- สิ่งที่เราจะเรียนกันในครั้งนี้มีแค่
  - container : ใช้ในการสร้าง data structure แบบต่างๆ
  - iterator: ใช้ในการวนรอบข้อมูลใน container
  - algorithms และ member functions: algorithm ที่ใช้กับ container และ function ที่สามารถใช้ทำงานกับสมาชิกใน container

C++ : สิ่งที่ควรรู้ก่อน ใช้ STL

- Class
- References (Smart pointers)
- Templates

Classes

- คล้ายกับ struct ในภาษา C
- Object oriented programming (OOP)
- User-defined types
  - เราสามารถสร้างชนิดของข้อมูลใหม่ขึ้นมาได้ โดยการประกาศคลาสใหม่
  - เราสามารถกำหนดคุณสมบัติ (properties) และ กระบวนการทำงาน (method) ของคลาสได้
  - เราสามารถกำหนดความหมายเมื่อนำคลาสที่สร้างขึ้นไปใช้กับตัวดำเนินการ

Classes

```
class shape {
private:
   int x_pos;
   int y_pos;
   int color;
public:
   shape () : x_pos(0), y_pos(0), color(1) {}
   shape (int x, int y, int c = 1) : x_pos(x), y_pos(y), color(c) {}
   shape (const shape& s) : x_pos(s.x_pos), y_pos(s.y_pos), color(s.color) {}
  ~shape () {}
   shape& operator= (const shape& s) {
      x_pos = s.x_pos, y_pos = s.y_pos, color = s.color; return *this; }
   int get_x_pos () { return x_pos; }
   int get_y_pos () { return y_pos; }
   int get_color () { return color; }
   void set_x_pos (int x) { x_pos = x; }
   void set_y_pos (int y) { y_pos = y; }
   void set_color (int c) { color = c; }
   virtual void DrawShape () {}
   friend ostream& operator<< (ostream& os, const shape& s);
};
ostream& operator<< (ostream& os, const shape& s) {
   os <<  "shape: (" << s.x_pos << "," << s.y_pos << "," << s.color << ")";
   return os;
}
```

Classes

```
shape MyShape (12, 10, 4);

int color = MyShape.get_color();

shape NewShape = MyShape;
```

```
shape MyShape;
shape NewShape (MyShape);
```

References

- หลักการใหม่ที่มีเพิ่มขึ้นมาจากภาษา C ใช้ในการอ้างถึงตัวแปรโดยไม่ต้องใช้ pointer
  - ไม่สามารถใช้แทน pointer ได้ในทุกกรณี
- syntax: & เขียนตามหลังชนิดตัวแปร เช่น int& foo;
- ในการใช้งานจะแตกต่างจาก pointer คือ
  - ไม่ต้องใช้ * ในการอ้างค่ากลับ (dereference) และ ไม่ต้องใช้ & ในการให้ค่าที่อยู่ (address) ของตัวแปร
  - NULL reference ไม่มี
  - ต้องกำหนดค่าเริ่มต้นให้กับ reference เมื่อมีการประกาศทันที
  - เมื่อกำหนดค่าให้กับ reference อ้างถึงตัวแปรใดแล้ว จะเปลี่ยนการอ้างถึงอีกไม่ได้

References

pointer

```
int x = 10;
int *p;

p = &x;
*p = 20;

printf("%d\n", x);
```

reference

```
int x = 10;
int &p = x;

p = 20;

printf("%d\n", x);
```

References

pointer

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
...
int x=10, y=20;
swap(&x, &y);
```

reference

```
void swap(int &a, int &b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
...
int x=10, y=20;
swap(x, y);
```

Templates

- templates เป็นหลักการที่ช่วยให้เราออกแบบ function และ class โดยไม่ขึ้นกับชนิดของข้อมูล

- ตัวอย่างเช่น container ใช้หลักการนี้เพื่อทำให้ data structures ชนิดต่างๆ ไม่ขึ้นกับชนิดของข้อมูล

Templates: function templates - 1

```
template <class T> void swap(T *a, T *b)
{
    T tmp = *a;
    *a = *b;
    *b = tmp;
}
...
int x=10, y=20;
swap(&x, &y);

float m=9.5, n=2.3;
swap(&m, &n);
```

Templates: function templates - 2

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

template <class T> void printArray(const T *array, const int count) {
    for (int i=0; i<count; i++)
        cout << array[i] << " ";
    cout << endl;
}
int main( int argc, char **argv )  {
    const int aCount = 5;
    const int bCount = 7;
    const int cCount = 6;
    int a[aCount] = {1,2,3,4,5};
    double b[bCount] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7};
    char c[cCount] = "HELLO";

    cout << "Array a contains: ";
    printArray(a,aCount);
    cout << "Array b contains: ";
    printArray(b,bCount);
    cout << "Array c contains: ";
    printArray(c,cCount);
    return 0;
}
```

Templates: class templates - 1

```
template <class T> class vector
{
   T* v;
   int sz;
public:
   vector (int s) { v = new T [sz = s]; }
  ~vector () { delete[] v; }
   T& operator[] (int i) { return v[i]; }
   int get_size() { return sz; }
};
```

Templates: class templates - 2

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

template <class T> class vector {
   T* v;
   int sz;
public:
   vector (int s) { v = new T [sz = s]; }
   ~vector () { delete[] v; }
   T& operator[] (int i) { return v[i]; }
   int get_size() { return sz; }
};

int main() {
   vector<int> intStore(10);
   vector<double> doubleStore(20);

   doubleStore[0] = 5;
   cout << "doubleStore[0]=" << doubleStore[0] << endl;
   cout << "intStore[0]=" << intStore[0] << endl;
   return 0;
}
```

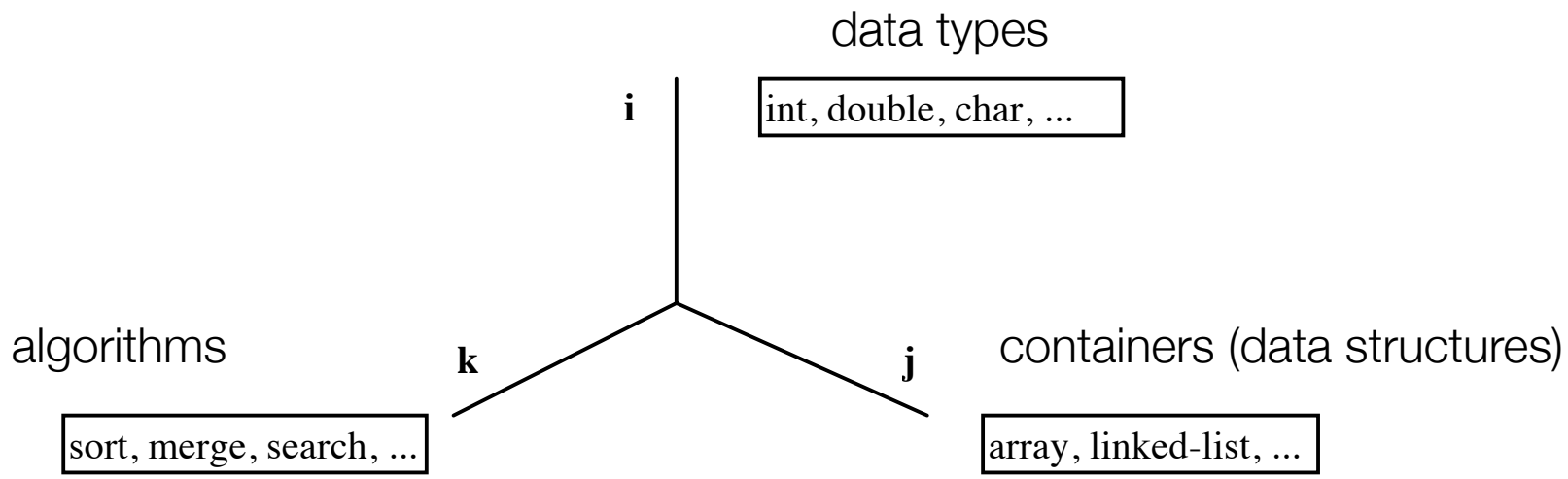
Templates: class templates - 3

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

template <class T> class Stack {
  vector<T> s;
public:
  int size() { return s.size(); }
  void push(T x) {
    if (s.size() < 100)
      s.insert(s.begin(),x); }
  T pop() {
      T item = s[0];
      s.erase(s.begin());
      return item;
    }
  void purge() { s.erase(s.begin(),s.end()); }
  void print() {
    int i=0;
    printf("S:[");
    for (i=s.size()-1;i>=0;i--) {
      printf(" %d",s[i]);
    }
    printf(" ]\n");
  }
};
```

Templates: class templates - 4

```
int main() {
  Stack<int> s;
  int x;
  char cmd;

  while (1) {
    if (cmd != '\n')
         cout << "input>";
    scanf("%c",&cmd);
    if(cmd == 'p') {
       s.print();
    } else if (cmd == 'u') {
       scanf("%d",&x);
       s.push(x);
    } else if (cmd == 'o') {
       if (s.size() > 0)
          cout << s.pop() << endl;
    } else if (cmd == 'q') {
       break;
    }
  }
  return 0;
}
```

The idea behind STL



หากทำแบบถึกๆ เราจะมีโค้ดทั้งหมด (i * k * j) + j

หาก container ไม่ขึ้นกับ data types เราจะมีโค้ดทั้งหมด (k * j) + j

และหาก algorithm ไม่ขึ้นกับ containers เราจะมีโค้ดทั้งหมด k + j

Containers

Containers

- Containers แบ่งได้เป็น 2 ประเภทคือ
  - Sequence containers
    - vector, deque (double ended queue), list
  - Associative containers
    - set, multiset, map, mutlimap

Vector

- vector เทียบได้กับ array ในภาษา C แต่มีความสามารถที่พิเศษกว่ามากมาย เช่น
  - ไม่จำเป็นจองเนื้อที่ล่วงหน้า (แต่จะจองก็ได้) เพราะ vector สามารถขยายขนาดได้เอง ถ้าเราใส่ข้อมูลเกินขนาดของ vector ที่จองไว้
  - สามารถรู้จำนวนข้อมูลที่มีอยู่ใน vector ได้
  - สามารถลบข้อมูลในตำแหน่งที่ต้องการออกจาก vector ได้

Vector

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> v;
  cout << v.size() << “\n”; // 0
  v.push_back(3);
  v.push_back(5);
  cout << v.size() << “\n”; // 2
  cout << v[0] << “ “ << v[1] << “\n”; // 3 5
  return 0;
}
```

Vector

```
vector<int> v(3);
cout << v.size() << “\n”; // print 3
v[0] = 2;
v[1] = 3;
v[2] = 4;
v.push_back(5);
cout << v.size() << “\n”; // print 4
```

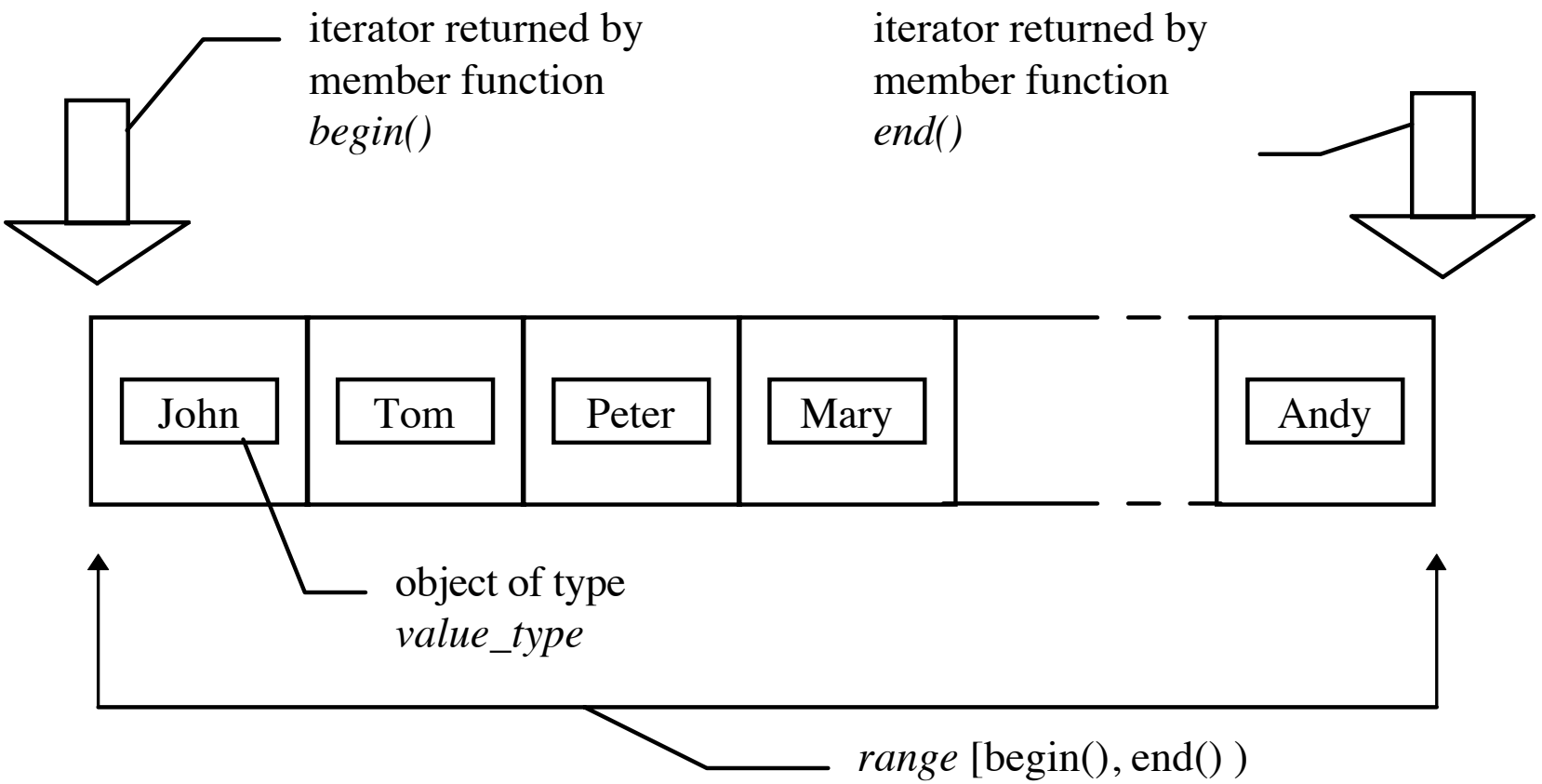
Iterator

- iterator ใช้สำหรับการอ้างถึงตำแหน่งของข้อมูลใน container
  - แต่ละ container จะมี iterator เฉพาะของตัวเอง แต่วิธีการใช้งานจะเหมือนกัน
- iterator มักจะถูกใช้ในการ
  - วนรอบข้อมูลใน container
  - เพิ่ม/ลบ ข้อมูล ในช่วงหรือตำแหน่งที่ต้องการ
  - กำหนดจุดเริ่มต้นในการค้นหาข้อมูล

Iterator

- รูปแบบการใช้งาน iterator จะเหมือนกับ pointer คือ
  - ใช้ + และ - ในการเลื่อนตำแหน่งการชี้ข้อมูล
  - ใช้ * ในอ้างกลับ (dereference)
- เราสามารถได้ค่าของ iterator จากคำสั่ง .begin() และ .end() ของ container ใดๆ

Iterator


iterator returned by
member function
begin()
iterator returned by
member function
end()
John
Tom
Peter
Mary
Andy
object of type
value_type
range [begin(), end())

Iterator

```
vector<int> v(3, 5); // [5, 5, 5]
...
for (vector<int>::iterator itr=v.begin(); itr != v.end(); ++itr) {
    cout << *itr << “\n”;
}
```

```
vector<int> w(1, 3); // [3]
w.insert(w.begin(), 2, 4); // [4, 4, 3]
w.insert(w.end(), v.begin(), v.end()); // [4, 4, 3, 5, 5, 5]
```

```
w.erase(w.begin()); // [4, 3, 5, 5, 5]
w.erase(w.end()-2, w.end()); // [4, 3, 5]
```

```
vector<int>::const_iterator p = find(w.begin(), w.end(), 4);
cout << (p != w.end()) ? “Found\n” : “Not found\n”;
```

List

- เปรียบเทียบได้กับ doubly-linked list
- รูปแบบการใช้งานเหมือน vector แต่ไม่สามารถอ้างถึงข้อมูลแบบ random access ได้
  - การอ้างถึงข้อมูลจะทำได้ผ่านทาง iterator อย่างเดียว
- ข้อดีของ list คือ การเพิ่มหรือลบข้อมูล เข้าไปใน list ที่ตำแหน่งใดๆ จะมีประสิทธิภาพมากกว่า vector

List

- list มีการทำงานแบบพิเศษที่ไม่มีใน vector ซึ่งเป็นการทำงานที่เกี่ยวกับการเปลี่ยนแปลงข้อมูลใน list
  - splice : ลบข้อมูลจาก list หนึ่ง แล้วเอามาใส่ในอีก list หนึ่ง
  - sort : เรียงลำดับข้อมูลใน list
  - merge : ลบข้อมูลทั้งหมดใน list หนึ่ง แล้วเอามาใส่ในอีก list หนึ่ง (ถ้าข้อมูลใน list ทั้งสองเรียงลำดับอยู่แล้ว ผลลัพธ์ที่ได้จากการ merge จะเรียงลำดับ)
- การทำงานเหล่านี้ จะไม่มีการทำซ้ำข้อมูลและเปลี่ยนตำแหน่งการเก็บของข้อมูล แต่จะใช้การเปลี่ยนแปลงตัวชี้ของข้อมูลใน list แทน

List : splice

fruit:
apple *pear*

citrus:
orange *grapefruit* *lemon*

```
list<string>::iterator p = find(fruit.begin(), fruit.end(), “pear”);
fruit.splice(p, citrus, citrus.begin());
```

fruit:
apple *orange* *pear*

citrus:
grapefruit *lemon*

List : splice

fruit:
apple *orange* *pear*

citrus:
grapefruit *lemon*

```
fruit.splice(fruit.begin(), citrus);
```

fruit:
grapefruit *lemon* *apple* *orange* *pear*

citrus:
<empty>

List : sort, merge

f1:
apple quince pear
f2:
lemon grapefruit orange lime

```
f1.sort();
f2.sort();
f1.merge(f2);
```

f1:
apple grapefruit lemon lime orange pear quince
f2:
<empty>

List : front operations

```
template <class T, class A = allocator<T> > class list {
public:
        // ...
        // element access:

        reference front();                  // reference to first element
        const_reference front() const;

        void push_front(const T&);          // add new first element
        void pop_front();                   // remove first element

        // ...
};
```

front operations มีประสิทธิภาพเท่ากับ back operation แต่ถ้าเป็นไปได้ควรใช้ back operation เพราะโปรแกรมที่เขียนจะสามารถใช้ container ตัวอื่น แทนได้

List : others

```
template <class T, class A = allocator<T> > class list {
public:
        // ...

        void remove(const T& val);
        template <class Pred> void remove_if(Pred p);

        void unique();                                  // remove duplicates using ==
        template <class BinPred> void unique(BinPred b);  // remove duplicates using b

        void reverse();                                 // reverse order of elements
};
```

List : remove

fruit:

apple *orange* *grapefruit* *lemon* *orange* *lime* *pear* *quince*

```
bool initial(string s, char c) {
    return s[0] == c ? true : false;
}
```

```
fruit.remove("orange"); // apple grapefruit lemon orange line pear quince
fruit.remove_if(initial('l')); // apple grapefruit pear quince
```

List : unique

```
fruit:
  apple pear apple apple pear
```

```
fruit.unique() // apple pear apple pear
```

```
fruit:
  apple pear apple apple pear
```

```
fruit.sort() // apple apple apple pear pear
fruit.unique() // apple pear
```

```
fruit:
  pear pear apple apple
```

```
fruit.unique(initial('p')) // pear apple apple
```

List : reverse

fruit:

banana cherry lime strawberry

```
fruit.reverse();
```

fruit:

strawberry lime cherry banana

Deque

- deque (อ่านว่า deck) หรือ double-ended queue
- deque สามารถเพิ่มหรือลบข้อมูล ที่ตำแหน่ง หัวและท้าย ได้มีประสิทธิภาพเท่ากับ list และ สามารถเข้าถึงข้อมูลแบบ random access ได้เหมือน vector และ มีประสิทธิภาพเท่ากัน
- การเพิ่มหรือลบข้อมูลในส่วนที่ไม่ใช่หัวและท้าย จะมีประสิทธิภาพที่แย่เหมือนกับ vector แต่ list จะทำได้ดีกว่า

Sequence adapters

- sequence adapters คือ containers พิเศษ ที่นำ sequence containers พื้นฐาน คือ vector, list และ deque มาใช้เป็นฐานในการสร้าง
    - stack, queue, priority queue

Stack

```
template <class T, class C = deque<T> > class std::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) { }

        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(); }
};
```

Queue

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

    explicit queue(const C& a = C()) : c(a) { }

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

    value_type& front() { return c.front(); }
    const value_type& front() const { return c.front(); }

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

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```

Priority queue

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type> >
class std::priority_queue {
protected:
        C c;
        Cmp cmp;
public:
        typedef typename C::value_type value_type;
        typedef typename C::size_type size_type;
        typedef C container_type;

        explicit priority_queue(const Cmp& a1 = Cmp(), const C& a2 = C())
                : c(a2), cmp(a1) { }
        template <class In>
        priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

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

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

        void push(const value_type&);
        void pop();
};
```

Associative containers

- associative array บางทีเรียกว่า dictionary หรือ map เป็นชนิดข้อมูลที่ในการเก็บข้อมูลในแบบที่เป็นคู่ คือ key และ mapped value
    - หรือ อาจจะมองได้ว่า associative array คือ array แบบปกติที่ไม่จำเป็นต้องใช้ integer เป็น index
- ใน STL มี associative containers อยู่ 4 แบบ คือ
    - map, multimap, set, multiset

Map

```
map<string, int> m; // key:string, mapped value:int

int x = m[“Henry”]; // create new entry for “Henry”, initialize to 0, return 0

m[“Harry”] = 7;     // create new entry for “Harry”, initialize to 0, assign 7

int y = m[“Henry”]; // return the value from “Henry”’s entry

m[“Harry”] = 9;     // change the value from “Harry”’s entry to 9
```

Map

Input

nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250

Output

```
hammer    9
nail      1350
saw       7
----------------
total     1366
```

Map

```
void readitems(map<string,int>& m)
{
    string word;
    int val = 0;
    while (cin >> word >> val) m[word] += val;
}

int main()
{
    map<string,int> tbl;
    readitems(tbl);

    int total = 0;
    typedef map<string,int>::const_iterator CI;

    for (CI p = tbl.begin(); p!=tbl.end(); ++p) {
        total += p->second;
        cout << p->first << '\t' << p->second << '\n';
    }

    cout << "----------------\ntotal\t" << total << '\n';

    return !cin;
}
```

iterator ของ map จะเก็บข้อมูลประเภท pair
โดย first คือ key และ second คือ mapped value

Map operations

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
    // ...
    // map operations:

    iterator find(const key_type& k);                 // find element with key k
    const_iterator find(const key_type& k) const;

    size_type count(const key_type& k) const;         // find number of elements with key k

    iterator lower_bound(const key_type& k);          // find first element with key k
    const_iterator lower_bound(const key_type& k) const;
    iterator upper_bound(const key_type& k);          // find first element with key greater than k
    const_iterator upper_bound(const key_type& k) const;

    pair<iterator,iterator> equal_range(const key_type& k);
    pair<const_iterator,const_iterator> equal_range(const key_type& k) const;

    // ...
};
```

Map : find

```
void f(map<string,int>& m)
{
        map<string,int>::iterator p = m.find("Gold");
        if (p!=m.end()) {                       // if "Gold" was found
                // ...
        }
        else if (m.find("Silver")!=m.end()) {   // look for "Silver"
                // ...
        }
        // ...
}
```

Map : list operations

- นอกจากการใช้ [ ] ในการอ้างถึงข้อมูลใน map แล้ว เรายังสามารถใช้วิธีการในรูปแบบเดียวกับที่ใช้กับ list ได้ เช่น insert และ erase

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
    // ...
    // list operations:

    pair<iterator, bool> insert(const value_type& val);     // insert (key,value) pair
    iterator insert(iterator pos, const value_type& val);   // pos is just a hint
    template <class In> void insert(In first, In last);     // insert elements from sequence

    void erase(iterator pos);                   // erase the element pointed to
    size_type erase(const key_type& k);         // erase element with key k (if present)
    void erase(iterator first, iterator last);  // erase range
    void clear();

    // ...
};
```

Map : list operations

insert แบบที่หนึ่ง

```
void f(map<string,int>& m)
{
        pair<string,int> p99("Paul",99);

        pair<map<string,int>::iterator,bool> p = m.insert(p99);
        if (p.second) {
                // "Paul" was inserted
        }
        else {
                // "Paul" was there already
        }
        map<string,int>::iterator  i = p.first;         // points to m["Paul"]
        // ...
}
```

Map : list operations

insert แบบที่สอง

```
void f(map<string,int>& m)
{
        m["Dilbert"] = 3;     // neat, possibly less efficient
        m.insert(m.begin(),make_pair(const string("Dogbert"),99));        // ugly
}
```


พารามิเตอร์ตัวแรกที่ ใส่เข้าไปเป็นคำตำแหน่งที่เราแนะนำ ซึ่งเป็นไปได้ว่า ข้อมูลที่เรา ใส่เข้าไปอาจจะไม่ได้ถูก ใส่เข้าไปที่ตำแหน่งนั้นจริงๆ ก็ได้

ถ้าตำแหน่งที่เราแนะนำเหมาะสมประสิทธิภาพจะดีกว่าการ ใช้ [ ]

Map : list operations

สอง statements นี้ได้ผลเหมือนกัน

```
m[k];

(* (m.insert(make_pair(k, V()))).first)).second;
```

V( ) คือ ค่า default ของตัวแปรที่เป็น mapped value
ดังนั้น ถ้าตัวแปรที่ใช้เป็น mapped value ไม่มีค่า default
เราจะใช้ [ ] กับ map ไม่ได้

Map : list operations

```
void f(map<string,int>& m)
{
        int count = phone_book.erase("Ratbert");
        // ...
}
```

ใช้ key ในการลบ และ return จำนวนของรายการข้อมูลที่ถูกลบกลับมา (0 หรือ 1)

```
void g(map<string,int>& m)
{
        m.erase(m.find("Catbert"));
        m.erase(m.find("Alice"),m.find("Wally"));
}
```

ใช้ iterator ในการลบ วิธีนี้เราสามารถบอกเป็นช่วงที่ต้องการลบได้ (ไม่ return ค่ากลับ)

Map : other functions

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class map {
public:
    // ...
    // capacity:

    size_type size() const;          // number of elements
    size_type max_size() const;      // size of largest possible map
    bool empty() const { return size()==0; }

    void swap(map&);
};
```

nonmember functions

```
template <class Key, class T, class Cmp, class A>
bool operator==(const map<Key,T,Cmp,A>&, const map<Key,T,Cmp,A>&);

// similarly !=, <, >, <=, and >=

template <class Key, class T, class Cmp, class A>
void swap(map<Key,T,Cmp,A>&, map<Key,T,Cmp,A>&);
```

ทุก container จะมี <, ==, swap เพื่อทำ ให้ทุก container ใช้อัลกอลิทึม sort ได้

Multimap

- เหมือน Map แต่หนึ่ง key มีได้หลาย mapped values และ ไม่สามารถใช้ [ ]

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key,T> > >
class std::multimap {
public:
    // like map, except:

    iterator insert(const value_type&);    // returns iterator, not pair

    // no subscript operator []
};
```

- คำสั่ง insert จะคืนค่ากลับมาเป็น iterator อย่างเดียว เพราะ Multimap ไม่มีการตรวจสอบว่า key ที่ใส่เข้าไปซ้ำกับของที่มีอยู่หรือเปล่า

Multimap

- การใช้คำสั่ง .find(k) จึงไม่ค่อยมีประโยชน์และไม่ค่อยถูกใช้งานกับ multimap
- multimap จะมีคำสั่ง
  - lower_bound(k) สำหรับหาค่าของ iterator ที่ชี้ไปยัง mapped value ตัวแรก
  - upper_bound(k) สำหรับหาค่าของ iterator ที่ชี้ไปยัง mapped value ตัวสุดท้าย
  - equal_range(k) สำหรับหาค่าของ iterator ที่ชี้ไปยัง mapped value ตัวแรกและตัวสุดท้าย (คืนค่ากลับมาเป็น pair)

Multimap

```
void f(multimap<string, int> &m)
{
  typedef MMI multimap<string,int>::iterator;

  MMI lb = m.lower_bound(“Gold”);
  MMI ub = m.upper_bound(“Gold”);

  for(MMI p = lb; p!=ub; ++p) {
    // ...
  }
}
```

Multimap

```
void print_numbers(const multimap<string,int>& phone_book)
{
        typedef multimap<string,int>::const_iterator I;
        pair<I,I> b = phone_book.equal_range("Stroustrup");
        for (I i = b.first; i != b.second; ++i) cout << i->second << ´\n´;
}
```

ตัวอย่างของการใช้ Multimap เช่น การเก็บเบอร์โทรศัพท์ เพราะคนหนึ่งคนมีได้หลายเบอร์

Almost containers

- String <string>
  - subscripting, random-access iterators, ข้อมูลเป็น characters เท่านั้น
- Valarray <valarray>
  - เหมือน vector แต่ถูก optimized ให้เหมาะสมกับข้อมูลประเภทตัวเลข ใช้งานได้แค่ size( ) และ subscripting ส่วน iterator เป็นแบบ random-access
- Bitset <bitset>
  - สามารถใช้ bitwise operators ได้, ใช้ subscripting ได้ แต่ไม่มี iterator

Bitset

- เราสามารถกำหนดจำนวนที่ใช้ได้เอง (ไม่จำเป็นต้องเป็น 2^n)
- bitset<N> คืออาเรย์ของ N bits
  - แตกต่างจาก vector<bool> เพราะเปลี่ยนแปลงขนาดไม่ได้
  - แตกต่างจาก set เพราะสามารถใช้ index ที่เป็นตัวเลขได้
  - มี operation เฉพาะสำหรับการเปลี่ยนแปลงบิต
- การอ้างถึงบิตโดยใช้ index เป็นตัวเลขจะมีลำดับจากขวาไปซ้าย

Bitset : example

```
bitset<10> b1; // all 0
bitset<16> b2 = 0xaaaa; // 1010101010101010
bitset<32> b3 = 0xaaaa; // 00000000000000001010101010101010

bitset<10> b4(string("1010101010"));           // 1010101010
bitset<10> b5(string("10110111011110"), 4);    // 0111011110
bitset<10> b6(string("10110111011110"), 2, 8); // 0011011101

bitset<10> b7(string("n0g00d")); // invalid_argument thrown
bitset<10> b8 = "n0g00d"; // error: no char* to bitset conversion
```

Bit manipulation operations

```
template<size_t N> class std::bitset {
public:
        // ...
        // bitset operations:

        reference operator[](size_t pos);           // b[i]

        bitset& operator&=(const bitset& s);        // and
        bitset& operator|=(const bitset& s);        // or
        bitset& operator^=(const bitset& s);        // exclusive or

        bitset& operator<<=(size_t n);              // logical left shift (fill with zeros)
        bitset& operator>>=(size_t n);              // logical right shift (fill with zeros)

        bitset& set();                              // set every bit to 1
        bitset& set(size_t pos, int val = 1);       // b[pos]=val

        bitset& reset();                            // set every bit to 0
        bitset& reset(size_t pos);                  // b[pos]=0

        bitset& flip();                             // change the value of every bit
        bitset& flip(size_t pos);                   // change the value of b[pos]

        bitset operator~() const { return bitset<N>(*this).flip(); }     // make complement set
        bitset operator<<(size_t n) const { return bitset<N>(*this)<<=n; }  // make shifted set
        bitset operator>>(size_t n) const { return bitset<N>(*this)>>=n; }  // make shifted set

        // ...
};
```

Bitset : other operations

```
template<size_t N> class bitset {
public:
    // ...

    unsigned long to_ulong() const;

    template <class Ch, class Tr, class A> basic_string<Ch,Tr,A> to_string() const;

    size_t count() const;                  // number of bits with value 1
    size_t size() const { return N; }      // number of bits

    bool operator==(const bitset& s) const;
    bool operator!=(const bitset& s) const;

    bool test(size_t pos) const;           // true if b[pos] is 1
    bool any() const;                      // true if any bit is 1
    bool none() const;                     // true if no bit is 1
};
```

Algorithms and Function objects

“There are no awards for using the highest number of standard algorithms in a program. Nor are there awards for using standard algorithms in the most clever and obscure way. Remember, a primary aim of writing code is to make its meaning clear to the next person reading it – and that person just might be yourself a few years hence.”

Bjarne Stroustrup - The C++ Programming Language

Algorithms

- algorithm แต่ละตัวจะถูกแสดงในรูปแบบของ template function ดังนั้น algorithm หนึ่งตัวจึงสามารถถูกได้สามารถ หลาย containers

- STL ได้เตรียม algorithm ทั่วไปสำหรับ containers ไว้แล้วจำนวนหนึ่ง เช่น การเรียงลำดับ การค้นหา การเพิ่ม/ลบ ข้อมูล และ การท่องไปในข้อมูล (traversal)

```
void f(list<string>& ls)
{
        list<string>::const_iterator p = find(ls.begin(),ls.end(),"Fred");

        if (p == ls.end()) {
                // didn't find "Fred"
        }
        else {
                // here, p points to "Fred"
        }
}
```

ตัวอย่างการใช้ algorithm find( )

Non-modifying Sequence Operations <algorithm>	
`for_each()`	Do operation for each element in a sequence.
`find()`	Find first occurrence of a value in a sequence.
`find_if()`	Find first match of a predicate in a sequence.
`find_first_of()`	Find a value from one sequence in another.
`adjacent_find()`	Find an adjacent pair of values.
`count()`	Count occurrences of a value in a sequence.
`count_if()`	Count matches of a predicate in a sequence.
`mismatch()`	Find the first elements for which two sequences differ.
`equal()`	True if the elements of two sequences are pairwise equal.
`search()`	Find the first occurrence of a sequence as a subsequence.
`find_end()`	Find the last occurrence of a sequence as a subsequence.
`search_n()`	Find the nth occurrence of a value in a sequence

Modifying Sequence Operations <algorithm>	
`transforms()`	Apply an operation to every element in a sequence.
`copy()`	Copy a sequence starting with its first element.
`copy_backward()`	Copy a sequence starting with its last element.
`swap()`	Swap two elements.
`iter_swap()`	Swap two elements pointed to by iterators.
`swap_ranges()`	Swap elements of two sequences.
`replace()`	Replace elements with a given value.
`replace_if()`	Replace elements matching a predicate.
`replace_copy()`	Copy sequence replacing elements with a given value.
`replace_copy_if()`	Copy sequence replacing elements matching a predicate.
`fill()`	Replace every element with a given value.
`fill_n()`	Replace first n elements with a given value.
`generate()`	Replace every element with the result of an operation.
`generate_n()`	Replace first n elements with the result of an operation.
`remove()`	Remove elements with a given value.
`remove_if()`	Remove elements matching a predicate.
`remove_copy()`	Copy a sequence removing elements with a given value.
`remove_copy_if()`	Copy a sequence removing elements matching a predicate.
`unique()`	Remove equal adjacent elements.
`unique_copy()`	Copy a sequence removing equal adjacent elements.
`reverse()`	Reverse the order of elements.
`reverse_copy()`	Copy a sequence into reverse order.
`rotate()`	Rotate elements.
`rotate_copy()`	Copy a sequence into a rotated sequence.
`random_shuffle()`	Move elements into a uniform distribution.

Sorted Sequences <algorithm>	
`sort()`	Sort with good average efficiency.
`stable_sort()`	Sort maintaining order of equal elements.
`partial_sort()`	Get the first part of sequence into order.
`partial_sort_copy()`	Copy getting the first part of output into order.
`nth_element()`	Put the nth element in its proper place.
`lower_bound()`	Find the first occurrence of a value.
`upper_bound()`	Find the first element larger than a value.
`equal_range()`	Find a subsequence with a given value.
`binary_search()`	Is a given value in a sorted sequence?
`merge()`	Merge two sorted sequences.
`inplace_merge()`	Merge two consecutive sorted subsequences.
`partition()`	Place elements matching a predicate first.
`stable_partition()`	Place elements matching a predicate first, preserving relative order.

Set Algorithm <algorithm>	
`includes()`	True if a sequence is a subsequence of another.
`set_union()`	Construct a sorted union.
`set_intersection()`	Construct a sorted intersection.
`set_difference()`	Construct a sorted sequence of elements in the first but not the second sequence.
`set_symmetric_difference()`	Construct a sorted sequence of elements in one but not both sequence.

Heap Operations <algorithm>	
`make_heap()`	Make sequence ready to be used as a heap.
`push_heap()`	Add element to heap.
`pop_heap()`	Remove element from heap.
`sort_heap()`	sort the heap.

Minimum and Maximum <algorithm>	
`min()`	Smaller of two values
`max()`	Larger of two values
`min_element()`	Smallest value in sequence.
`max_element()`	Largest value in sequence.
`lexicographical_compare()`	Lexicographically first of two sequences.

Permutations <algorithm>	
`next_permutation()`	Next permutation in lexicographical order.
`prev_permutation()`	Previous permutation in lexicographical order.

Function objects

- A function object, also called a functor, functional, or functionoid,[1] is a computer programming construct allowing an object to be invoked or called as if it were an ordinary function, usually with the same syntax (a function parameter that can also be a function). (http://en.wikipedia.org/wiki/Function_object)
  - pointer to function ในภาษา C จัดว่าเป็น function object หรือเปล่า?
- ประโยชน์เหมือนกับการใช้ pointer to function แต่การใช้ function object จะดีกว่า เพราะที่เราสามารถเก็บไว้ใน function object ได้

Function objects : Example

- ตัวอย่างการใช้ function object ร่วมกับ for_each( ) เพื่อหาผลรวมของตัวเลขใน list

```
template<class T> class Sum {
        T res;
public:
        Sum(T i = 0) : res(i) { }               // initialize
        void operator()(T x) { res += x; }      // accumulate
        T result() const { return res; }        // return sum
};
```

เราสามารถสร้าง function object เองได้ โดยการสร้าง class ใหม่ พร้อม override operator( )

```
void f(list<double>& ld)
{
        Sum<double> s;
        s = for_each(ld.begin(),ld.end(),s);        // invoke s() for each element of ld
        cout << "the sum is" << s.result() << '\n';
}
```

Function object bases

- STL ได้เตรียม function objects ที่มีจำเป็นไว้แล้วจำนวนหนึ่ง นอกจากนั้นยังเตรียมคลาสพื้นฐานไว้ 2 คลาส เพื่อใช้ในการสร้าง function object ใหม่

```
template <class Arg, class Res> struct unary_function {
        typedef Arg argument_type;
        typedef Res result_type;
};

template <class Arg, class Arg2, class Res> struct binary_function {
        typedef Arg first_argument_type;
        typedef Arg2 second_argument_type;
        typedef Res result_type;
};
```

Predicates

- predicate คือ function object ที่คืนค่ากลับมาเป็นข้อมูลชนิด bool ตัวอย่างเช่น

```
template <class T> struct logical_not : public unary_function<T,bool> {
        bool operator()(const T& x) const { return !x; }
};

template <class T> struct less : public binary_function<T,T,bool> {
        bool operator()(const T& x, const T& y) const { return x<y; }
};
```

- ตัวอย่างการใช้ predicate ร่วมกับ algorithm mismatch

```
void f(vector<int>& vi, list<int>& li)
{
        typedef list<int>::iterator LI;
        typedef vector<int>::iterator VI;
        pair<VI,LI> p1 = mismatch(vi.begin(),vi.end(),li.begin(),less<int>());
        // ...
}
```

ค่าข้อมูลตัวแรกที่ ใน vi ที่น้อยกว่าข้อมูล ใน li

Predicates

Predicates <functional>		
equal_to	Binary	arg1==arg2
not_equal_to	Binary	arg1!=arg2
greater	Binary	arg1>arg2
less	Binary	arg1<arg2
greater_equal	Binary	arg1>=arg2
less_equal	Binary	arg1<=arg2
logical_and	Binary	arg1&&arg2
logical_or	Binary	arg1||arg2
logical_not	Unary	!arg

Arithmetic function objects

- เป็น function object ที่เตรียมไว้ให้ใช้กับข้อมูลประเภทตัวเลขโดยเฉพาะ

Arithmetic Operations <functional>		
plus	Binary	arg1+arg2
minus	Binary	arg1–arg2
multiplies	Binary	arg1*arg2
divides	Binary	arg1/arg2
modulus	Binary	arg1%arg2
negate	Unary	–arg

- ตัวอย่างการใช้ multiplies ร่วมกับ algorithm transform

```
void discount(vector<double>& a, vector<double>& b, vector<double>& res)
{
        transform(a.begin(),a.end(),b.begin(),back_inserter(res),multiplies<double>());
}
```

นำข้อมูล ใน a และ b มาจับคู่ตามลำดับแล้วคูณกัน และนำผลลัพธ์ที่ไปใส่ต่อทาง ใน res

Adapters

- adapter คือ function object ที่ใช้ในการเปลี่ยน**สิ่งที่มีอยู่แล้ว**ให้เป็น function object
  - binder : เปลี่ยน two-argument function object ให้เป็น single-argument function object โดยกำหนดตายตัวให้เป็น argument ตัวใดตัวหนึ่ง
  - member function adapter : เปลี่ยน member function
  - pointer to function adapter : เปลี่ยน pointer to function
  - negater : เปลี่ยน predicate เดิมให้เป็นค่าตรงข้าม

Binders, Adapters, and Negaters <functional>		
`bind2nd(y)`	`binder2nd`	Call binary function with y as 2nd argument.
`bind1st(x)`	`binder1st`	Call binary function with x as 1st argument.
`mem_fun()`	`mem_fun_t`	Call 0-arg member through pointer.
	`mem_fun1_t`	Call unary member through pointer.
	`const_mem_fun_t`	Call 0-arg const member through pointer.
	`const_mem_fun1_1`	Call unary const member through pointer.
`mem_fun_ref()`	`mem_fun_ref_t`	Call 0-arg member through reference.
	`mem_fun1_ref_t`	Call unary member through reference.
	`const_mem_fun_ref_t`	Call 0-arg const member through reference.
	`const_mem_fun1_ref_t`	Call unary const member through reference.
`ptr_fun()`	`pointer_to_unary_function`	Call unary pointer to function.
`ptr_fun()`	`pointer_to_binary_function`	Call binary pointer to function.
`not1()`	`unary_negate`	Negate unary predicate.
`not2()`	`binary_negate`	Negate binary predicate.

Binders

- ตัวอย่างการใช้ bind2nd ในการสร้าง unary predicate “less than 7” จาก binary predicate “less” และ ค่า 7

```
void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(),c.end(),bind2nd(less<int>(),7));
    // ...
}
```

- หากไม่ใช้ binder เราอาจสร้าง predicate “less_than” จาก unary predicate ดังนี้

```
template <class T> class less_than : public unary_function<T,bool> {
    T arg2;
public:
    explicit less_than(const T& x) : arg2(x) { }
    bool operator()(const T& x) const { return x<arg2; }
};

void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(),c.end(),less_than<int>(7));
    // ...
}
```

Member function adapters

- ในบางครั้งเราต้องการที่จะใช้ member function ร่วมกับ algorithm เช่น

```
void draw_all(list<Shape *> &lsp)
{
    for_each(lsp.begin(), lsp.end(), &Shape::draw); // opps! error
}
```

- ในกรณีนี้ STL ได้เตรียม mem_fun( ) และ mem_fun_ref( ) เพื่อความสะดวกในการใช้ member function ร่วมกับ algorithm

```
void draw_all(list<Shape*>& lsp)      // call 0-argument member through pointer to object
{
    for_each(lsp.begin(),lsp.end(),mem_fun(&Shape::draw));    // draw all shapes
}


void f(list<string>& ls)      // use member function that takes no argument for object
{
    typedef list<string>::iterator LSI;
    LSI p = find_if(ls.begin(),ls.end(),mem_fun_ref(&string::empty)); // find ""
}
```

Pointer to function adapters

- algorithm สามารถใช้งานร่วมกับทั้ง functions และ function objects แต่ binder สามารถรับ argument ที่เป็น function object ได้อย่างเดียว

- ดังนั้น ptr_fun จึงใช้เพื่อเปลี่ยน pointer to function เป็น function object เพื่อให้สามารถใช้เป็น argument ของ binder ได้

```
class Record { /* ... */ };

bool name_key_eq(const Record&, const Record&);  // compare based on names
bool ssn_key_eq(const Record&, const Record&);   // compare based on number

void f(list<Record>& lr)    // use pointer to function
{
        typedef typename list<Record>::iterator LI;
        LI p = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(name_key_eq),"John Brown"));
        LI q = find_if(lr.begin(),lr.end(),bind2nd(ptr_fun(ssn_key_eq),1234567890));
        // ...
}
```

Negaters

- ตัวอย่างการใช้ negater เพื่อสร้าง predicate “not less than” จาก “less than”

```
void f(vector<int> &vi, list<int> &li)
{
    // ...
   p1 = mismatch(vi.begin(),vi.end(),li.begin(),not2(less<int>()));
    // ...
}
```

- ตัวอย่างการใช้ not1, bind2nd และ ptr_fun ร่วมกัน เพื่อหา “funny” ใน list<char *>

```
extern "C" int strcmp(const char*,const char*);      // from <cstdlib>

void f(list<char*>& ls)      // use pointer to function
{
      typedef typename list<char*>::const_iterator LI;
      LI p = find_if(ls.begin(),ls.end(),not1(bind2nd(ptr_fun(strcmp),"funny")));
}
```