

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 ที่สามารถใช้งานกับสมาชิกใน container

C++ : สิ่งที่เราควรรู้ก่อนใช้ 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 ใช้หลักการนี้เพื่อให้ 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;

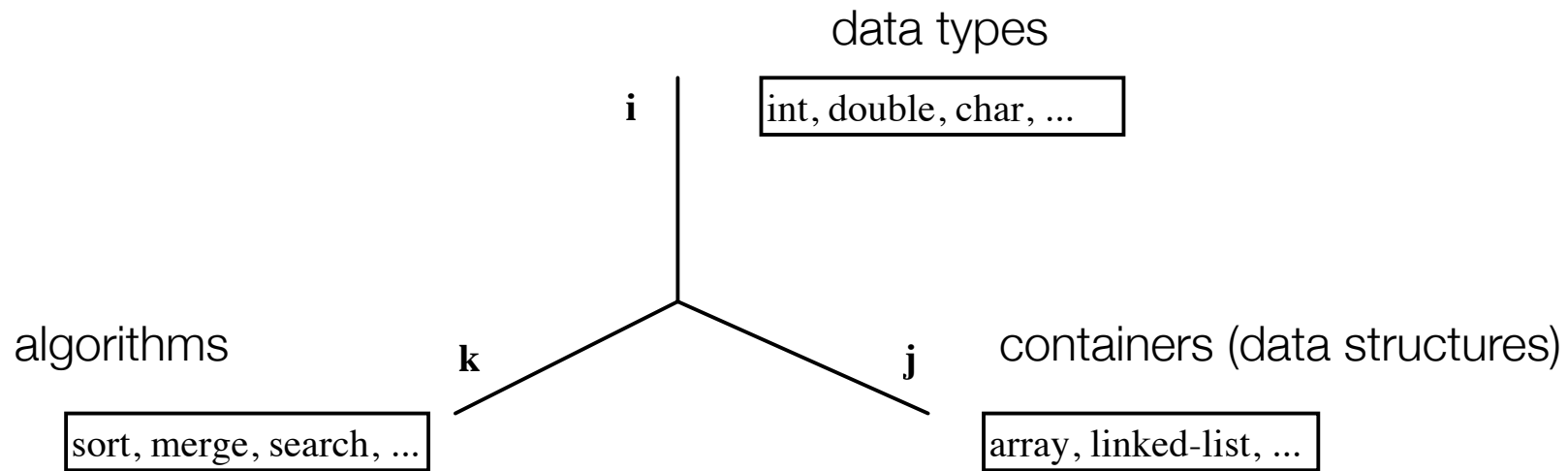
template <class T> class Stack {
    vector<T> s;
public:
    int size() { return s.size(); }
    void push(T x) {
        if (s.size() < 100)
            s.insert(s.begin(),x); }
    T pop() {
        T item = s[0];
        s.erase(s.begin());
        return item;
    }
    void purge() { s.erase(s.begin(),s.end()); }
    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
 - ไม่จำเป็นต้องเนื้อที่ล่วงหน้า (แต่จะจองก็ได้) เพราะ vector สามารถขยายขนาดได้เองถ้าเราใส่ข้อมูลเกินขนาดของ 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
```

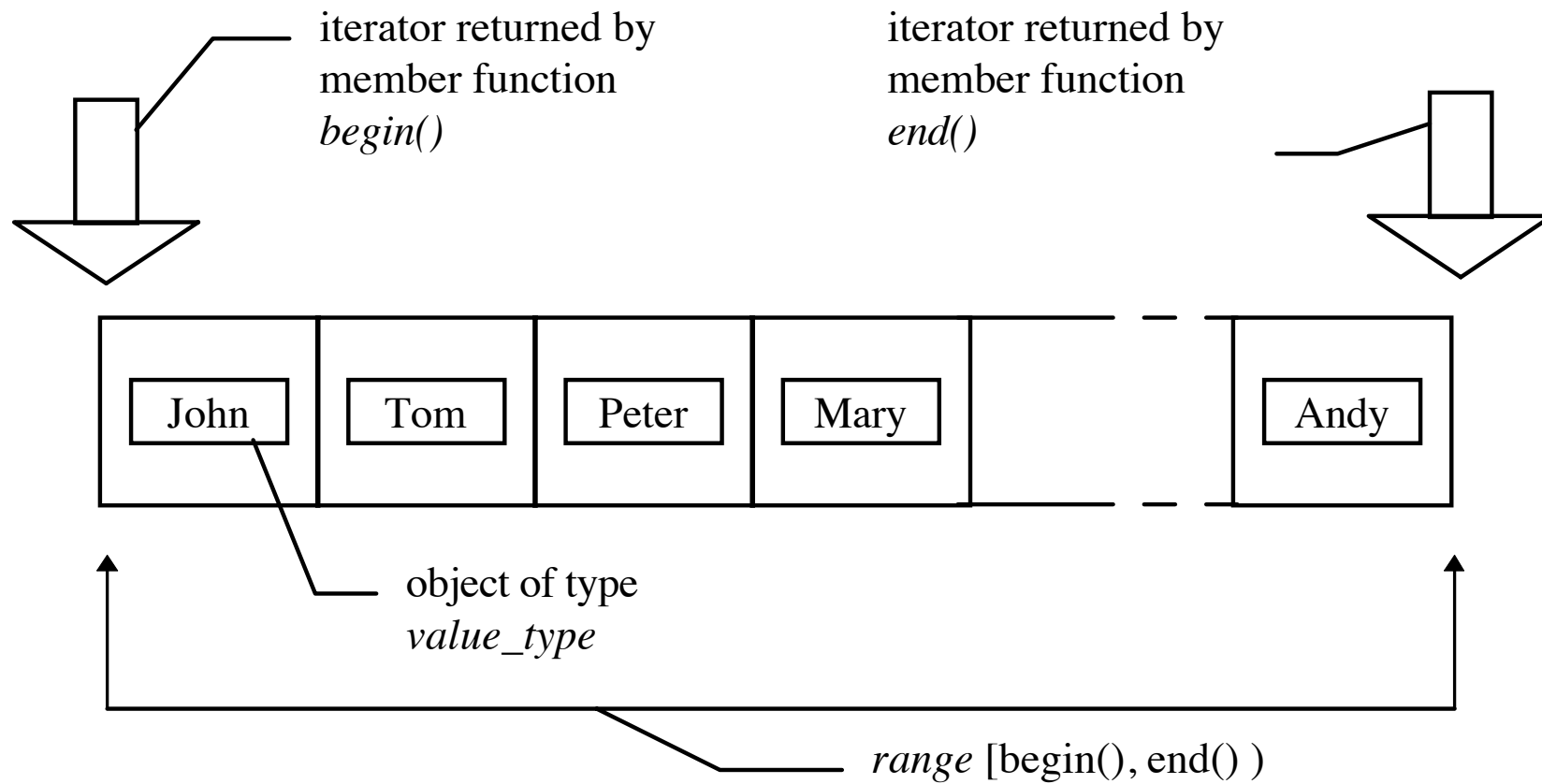
Iterator

- 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

```
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