Using STL in Programming Contests

Jongwook Choi

Seoul National University
2011 Algospot Seminar @ Google

Jan 9, 2011

Table Of Contents I

Table Of Contents II

About this Seminar

- Lecturer : Jongwook Choi (a.k.a. wookayin)

STL ?

STL (Standard Template Library)

- Implements useful data structures and basic algorithms
- Standard
  - Available in any C++ compiler (G++, VC++)
- Template
  - Classes and Functions with Generic type
  - They can work on many different data types, without being rewritten for each one.

Why we should use C++ libraries ?

To write a program accurately and quickly

Overview

Containers

- pair
- vector (array)
- string, stringstream[1]
- stack, queue, deque
- priority_queue (heap)
- set, map, multiset, multimap, unordered_set, unordered_map
- ....

Algorithms

- sort, stable_sort
- unique
- next_permutation, prev_permutation
- lower_bound, upper_bound
- ....

[1] Strictly speaking, string and stringstreams are not parts of STL

Preliminaries

All containers and functions are contained in the namespace `std`

```cpp
#include <vector>       // include appropriate headers
#include <algorithm>

int main()
{
    std::vector<int> a(3, 0);
    std::sort(a.begin(), a.end());
    return 0;
}
```

```cpp
#include <vector>
#include <algorithm>
using namespace std;    // (!)

int main()
{
    vector<int> a(3, 0);
    sort(a.begin(), a.end());
    return 0;
}
```

Pair

`std::pair<T1, T2>`

- Just pair : a bundle of two objects, of any type
- Two member fields : `first`, `second`
- Two ways to make a pair
  - Constructor : `pair<T1, T2>(first, second)`
  - The `make_pair` function
- Examples

```
pair<int, int> a = make_pair(3, 4);
cout << a.first + a.second << endl;      // 7
a = pair<int, int>(-1, 1);

pair<char*, pair<double, double> >
    t("Seoul", make_pair(37.541, 126.986) );    // nested pairs
printf("Location of %s is (%lf, %lf).",
    t.first, t.second.first, t.second.second);
```

Pair - Operators

Lexicographical Comparison Operators

- =, ==, !=, <, <=, >, >=
- Compare two pair of same type by first elments, and then by second elements
  - `(3, 4) < (6, 2)`
  - `(1, 0) > (0, 90)`
- Useful to represent a datatype which has a natural order (you don't have to define operators)

Pair - Advanced Usage

User-defined operators

- You can even overload existing operators

```
pair<int, int> operator + (const pair<int, int> &L,
                           const pair<int, int> &R)
{
    return make_pair(L.first + R.first, L.second + R.second);
}
```

typedefs and macros

- Looooooong names are sometimes painful

```
typedef pair<int, int> point;
typedef pair<int, vector<int> > pivi;

#define X first      // sometimes, in geometry problems
#define Y second     // warning : when using tokens named X or Y
int example = point(3, 3).X + point(0, 0).Y;
```

Vector

std::vector<T>

- A variable-length (linear) array

```
#include <vector>
```

- Constructing a vector

```
vector<int> a;           // creates an empty vector (of size 0)
vector<int> b(10);       // creates a vector of int's of size 10
                         //  (default 0)
vector<int> c[10];       // creates an array of 10 vector<int>'s
vector<int> d(10, -1);   // creates a vector<int> of size 10,
                         //  all items initialized by -1
vector<int> e(d);        // creates a vector copying from d

// creates a vector, copying some range (see the iterator part)
vector<int> test( b.begin(), b.end() );

int array[8] = {3,1,4,1,5,9,2,6};
vector<int> vec(array, array + sizeof(array)/sizeof(int));
```

Vector - Basic Usage

- `size_t size()` : return the size of vector

```
vector<int> kong(22, 2);
size_t n = kong.size();     // (unsigned)22
```

- `bool empty()` : test if a vector is of size 0 or not

```
vector<int> def;
assert( def.empty() == true );
```

- `void clear()` : clear a vector
  - $O(n)$, where n is the number of elements in the vector
  - Note that the allocated memorys are not freed immediately[2]

```
x.clear();  // x is a vector<T>
assert( x.size() == 0 );
```

[2] A way to free memorys : `x.swap( vector<T>() );`

Vector - Basic Usage

- `T& operator []` : accesses to an element (0-based index)
  - Run-time Error when an invalid index is given
  - `.at(index)` is equivalent

```
vector<int> a = GetTestVector();
if(!a.empty()) a[0] = -1;
for(int i=0; i<(int)a.size(); ++i)
{
    a[i] ++; a[i] += a[0];
    printf("a[%d] = %d\n", i, a[i]);
}
```

Vector - Inserting/Removing an item

- `T& back()` : returns the last element
- `T& first()` : returns first last element
- `void push_back(item)` : adds an element `item` into back
- `void pop_back()` : deletes the last element
  - Don't worry about memory allocation
  - `push_back`, `pop_back` runs in amortized $O(1)$ time

```
vector<int> A;
A.push_back(2011);
assert(2011 == A.back());
A.pop_back();
A.back() --;                    // runtime error (empty vector!)
```

Vector - Resizing

- `void resize(sz, with)` : resizes an vector to contain `sz` elements, filling with `with` when expanded.
  - `with` can be omitted (default value is used. e.g. `int : 0`)

```
vector<int> A;
for(int i=0; i<n; ++i)
    scanf("%d", &A[i]);      // error! A has no elements
```

```
vector<int> A;
A.resize(n);                 // expand A to have n elements
for(int i=0; i<n; ++i)
    scanf("%d", &A[i]);      // ok, it works
```

```
vector<int> K(2, 2);         // K = {2, 2}
K.resize(6);                 // K = {2, 2, 0, 0, 0, 0}
K.resize(4, 1);              // K = {2, 2, 0, 0}
K.resize(8, -1);             // K = {2, 2, 0, 0, -1, -1, -1, -1}
```

Vector - Multidimensional Arrays

```
vector< vector<int> > matrix;
matrix.resize(n);
for(int i=0; i<n; ++i)
    matrix[i].resize(m);

matrix[0][0] = matrix[n-1][m-1] = 1;
```

```
vector< vector<int> > matrix(n, m);
// vector< vector<int> > matrix(n, vector<int>(m));
matrix[0][0] = matrix[n-1][m-1] = 1;
```

```
vector<int> C[10];
for(int i=0; i<10; ++i)
{
    C[i].resize(i+1);
    C[i][0] = 1;
    for(int j=1; j<=i; ++j)
        C[i][j] = C[i-1][j-1] + C[i-1][j];
}
```

Vector - Multidimensional Arrays

Adjacency list representation of a graph

```
vector< vector<int> > G;

G.resize(V);    // V : # of vertices
for(int i=0; i<E; ++i)
{
    int x = edges[i].x, y = edges[i].y;
    G[x].push_back(y);
//  G[y].push_back(x);      // bidirectional ?
}

int u = 0;
for(int i=0; i<(int)G[u].size(); ++i) {
    int v = G[u][i];
    // for each edge (u, v) ...
}
```

Vector - Pitfalls

Common mistakes

- vector's (and as well as all other containers) `size()`[3] is of unsigned type

```
vector<int> v;  // an empty vector
for(int i=0; i<v.size() - 1; ++i) // (unsigned)0 - 1 == 4294967295
{
    // blahblah...
} /* Warning! This loop will not halt in time! */
```

```
for(int i=0; i<(int)v.size() - 1; ++i);     // good
```

- `vector<bool>` has different implementation
  - Continuous 8 elements are stored in 1 byte
  - Be cautious when using it — recommended not to use

[3] same for `length()`, `count()` in other containers

Iterator

(STL containers)`::iterator`

- An object which allows to traverse through all the elements of a container.
- Very useful and important when handling STL containers and using algorithms
- An abstraction of traditional C pointers
  - We can treat an iterator as a pointer !
- Since each container provides different mechanism to traverse through all the elements, implementations and interfaces of different containers may be differ

(Put images here)

Iterator - Common Features

- Declaration

```
vector<int> v = GetExampleVector();
set<int> s = GetExampleSet();

set<int>::iterator it;
vector<int>::iterator it  = a.begin();
vector<int>::iterator it2 = a.end();
```

- Pointer assignment, equality comparison
  - operator =, ==, !=
- Pointer dereferencing (as in C)
  - operator *, ->
- Iterate through elements
  - operator ++ : Moves to next object to me
  - operator -- : Moves to previous object to me

Iterator - Range

- In STL, a range (a sequence of continuous elements in a container) is represented as by a pair of two iterators, typically called begin and end
- The element pointed by end is excluded from the range
  - `[begin, end)`
  - `end` iterator is past-the-end iterator
- STL containers usually have `.begin()` and `.end()`

Iterator - Typical usages and Example

```
vector<int> a(10);
for(int i=0; i<10; ++i) a.push_back(i);

// iterate through all the elements in a vector
for(vector<int>::iterator it = a.begin(); it != a.end(); ++ it)
{
    *it ++;
    cout << *it << endl;
    *it = -1;
}
```

```
bool IsEmpty(vector<int> &a) { return a.begin() == a.end(); }
```

Iterator - std::vector<T>::iterator

- vector is a *linear* collection of elements, so it provides pointer arithmetics (same as C pointers)
  - operator +, −, +=, −=, <, <=, >, >=, []

```
vector<int> a(100);
vector<int>::iterator it = a.begin() + 50;      // &a[50]

// how to get the 0-based index of an element in a vector
printf("The index = %d \n", it - a.begin());    // 50
```

```
vector<int> a = ... ;    // suppose a = {0, 1, .. 99}
vector<int> x(a.begin() + 1, a.end());  // {1, .. 99}
/* Note : The constructors with two iterator parameters
          create a container copying elements from the
          'range' specified by these two iterators          */
```

Iterator - More

- `const_iterator` : analogy of const pointer
- `reverse_iterator`, `const_reverse_iterator`

```
// iterate through all the elements, but in a reverse order
vector<int>::reverse_iterator it;
for(it = a.rbegin(); it != a.rend(); ++it)
    DoSomething(*it);
```

Vector Revisited - Vector Manipulation

- erase
  - erases one or a range of elements (the size of vector is reduced)

```
vector<int> a = ...;                          // {1, 2, 3, 4, 5, 6}

//.erase a single element
a.erase(a.begin());                           // {2, 3, 4, 5, 6}

// erase a range of elements
a.erase(a.begin(), find(a.begin(), a.end(), 4), a.end());
                                              // {2, 3}
```

Vector Revisited - Vector Manipulation

- insert
  - inserts one or a range of elements before the given position

```
vector<int> a = ...;                    // {1, 2, 3, 4}
int b[] = {-1, -2, -3};

// insert a single element 0 at front
a.insert(a.begin(), 0);                 // {0, 1, 2, 3, 4}

// insert a range of elements in the middle
a.insert(a.begin() + 2, b, b+3);        // {0, 1, -1, -2, -3, 3, 4}
```

```
vector<int>& operator += (vector<int> &v, const vector<int> rhs)
{
    // append rhs into v
    v.insert(v.end(), rhs.begin(), rhs.end());
    return v;
}
```

String

`std::string`

- A container that handles a string
- Very convenient compared to naive C strings(`char[]`)

```
#include <string>
```

- Constructing a string

```
string eps;                    // creates an empty string of length 0
string a("Hello world!");      // using constructor
string b = "wookayin";         // substituting char*

string kong(2, '2');           // (length, char). result : "22"
string Kong(kong);             // copying.
```

String - Basic Usage

- `size_t length()` : returns the length of string

```
string a = "algospot.com";
cout << a.length() << endl;     // (unsigned)12
```

- `bool empty()` : tests if a string is of length 0
- `void clear()` : clears a string[4]

```
string x = "hello";
x.clear();
assert( x.empty() == true );
```

- `char operator []` : a reference to i-th character of a string

```
string str = "Corea";
str[0] = 'K';
cout << str << " " << str[2] << endl;   // Kr
```

- `push_back, pop_back, back, front`

[4] Allocated memorys won't be freed.

String - Useful operators

- = : assignment (copying from string, char*, ...)
- ==, != : equality comparison
- <, >, <=, >= : lexicographical comparison
- +, += : string concatenation

```
string s = "Algorithm";
s += " is fun";                 // "Algorithm is fun"
string t = "wook";
assert(t != "Wook");            // true
assert(s < t);                  // 'A' = 65, 'w' = 119
string opt = min(s, t);         // "Algorithm is fun"
opt += '!';                     // "Algorithm is fun!"
```

```
string s = "";
s += "1" + "234";               // compile error. Can't add two char*'s
s += string("1") + "234";       // OK
```

String - c_str()

- const char* c_str()
  - gets the C-style char* pointer of a string
  - useful when using printf, sscanf

```
string s = "You can't lose what you never had";
cout << s << endl;            // includes : <string>, <iostream>
printf("%s\n", s.c_str());    // good
printf("%s\n", s);            // bad (not a compile error, but...)
sscanf(s.c_str(), "%s", buf);   // buf <- "You"
```

- Remark : cannot input by scanf directly.

```
string s;
scanf("%s", s.c_str());       // Wrong! It may cause RE

char buffer[1024];
scanf("%s", buffer);
s = buffer;                   // do as this if you want to use scanf

cin >> s;                     // or, use std::cin directly
getline(cin, s);              // counterpart of gets()
```

String - substring

- `string substr(pos, len)`
  - returns a string from a substring, starting from 0-index `pos` and length `len`
  - Note that another string is created (memory is allocated)

```
string a = "abcdefghijk";

// when len is omitted, gets one to the end
cout << a.substr(7) << endl;    // hijk

// typical usage
cout << a.substr(3, 4) << endl; // defg

// example : a[i..j]
cout << a.substr(i, j-i+1) << endl;
```

String - iterators

- Usage of `string::iterator` is same as vector

```
string s = "this is a lower string";
for(string::iterator it = s.begin(); it != s.end(); ++it)
    *it = toupper(*it);
```

- `insert`, `erase` can be used similar as `vector`
  - 0-based indices can be sused rather than iterators

String - find

- `size_t find(s)`
  - find `s` as a substring (like `strchr`, `strstr`)
  - `s` may be a `string`, a `char*`, or a single `char`
  - return the 0-based *index* where `s` first occurs, or `string::npos`[5] if no occurences
  - Time Complexity : $O(nm)$ [6]

```
string s = "abcabcab", t = "Blahblah";
cout << s.find("abc") << ' ';                  // 0
cout << s.find(string("blah")) << endl;        // 4

if(s.find("algo") == -1)          // s.find("algo") < 0  won't work
    cout << "no matches!" << endl;

if(t.find("bl") != t::npos)       // or, string::npos
    cout << "matches!" << endl;
```

- `size_t rfind(s)` : almost same but find the last occurence

[5] -1 as unsigned type (`0xffffffff`).

[6] There is no specification in standards, and typically g++ and VC++ have $O(nm)$ implementations.

String - replace

- `.replace(from, to, with)`
  - replace the subrange $[\texttt{from}, \texttt{to})$ with a string[7] `with`
- `.replace(pos, len, with)`
  - replace the subrange which begins 0-based index `pos` and of length `len` with a string `width`
- Endure ineffectiveness of naive `string`'s operations

```
string s = "abracadabra";

cout << s << endl;                                  // abracadabra
cout << s.replace(10, 1, 'A') << endl;              // abracadabrA
cout << s.replace(0, 8, "co") << endl;                 // cobrA
cout << s.replace(&s[2], s.begin()+3, "RE") << endl; // coREA
```

[7] `std::string`, `char*`, `char`

Streams - std::stringstream

std::stringstream

- istringstream, ostringstream
- Performs input/output operations on a string, rather than stdandard I/O or file I/O
  - similar to sprintf, sscanf

```
#include <sstream>
```

Streams - istringstream

```
/* string -> int, long, ... */
inline long long ParseLong(const string &str)
{
    long long res = 0;
    istringstream(str) >> res;
    return res;
}
```

```
/* string tokenizing example */
string str = "split  me into   vector<string>s";
vector<string> result;

istringstream ss(str);  // splits on whitespaces
string x;
while(ss >> x) result.push_back(x);     // see loop condition
assert( result.size() == 4 );
```

```
/* invalid operation example */
istringstream ss("notdouble"); double t;
if(ss >> t) cout << t << endl;
else cout << "parsing failed" << endl;  // this is excuted
```

Streams - istringstream

```
/* BUT, if sscanf is simpler than istringstream, use sscanf !! */
pair<int, int> ParsePair(string &s) // s = "(3, 4)"
{
    pair<int, int> res;
    sscanf(s.c_str(), "(%d,%d)", &res.first, &res.second);
    return res;
}

/* istringstream is more concise due to variable-length */
vector<int> ParseVector(string s) // s = "{1, 2, 3, 4}"
{
    for(size_t i=0; i<s.length(); ++i)
        if(!isdigit(s[i]) && s[i] != '+' && s[i] != '-') s[i] = ' ';
    istringstream ss(s);
    vector<int> res;
    for(int x; ss >> x; ) res.push_back(x);
    return res;
}
```

Streams - ostringstream

```
/* int, long, ... -> string */
inline string ToString(const long long val)
{
    ostringstream ss;
    ss << val;
    return ss.str();    // .str() produces a string
} // alternatively, sprintf is also useful (maybe more faster)
```

Stack

`std::stack`

- Implements a FILO stack
  - Actually, `vector` can be used as stack

```
#include <stack>
```

- `bool empty()` : tests if a stack is empty
- `size_t size()` : gets the number of elements contained
- `void push(x)` : push `x` on top of the stack
- `T top()` : gets the top element (stack must not be empty)
- `void pop()` : pop out the top element

Queue

`std::queue`

- Implements a FIFO queue

```
#include <queue>
```

- `bool empty()` : tests if a queue is empty
- `size_t size()` : gets the number of elements contained
- `void push(x)` : push `x` on back of the queue
- `T back()` : gets the backmost element
- `T front()` : gets the frontmost element
- `void pop()` : pop out the frontmost element

Note : `stack` and `queue` have no iterators and do not support random-access, so if necessity, use `vector` or `deque` instead.

Queue - Simple BFS

```
int n;
vector<int> gr[MAXN];        // adjacency list, 0..n-1
vector<int> dist(MAXN);      // shortest distance from S

void BFS(int S)
{
    queue<int> Q;
    fill(dist, dist + n, 987654321);    // !?
    Q.push( S ); dist[S] = 0;

    while(!Q.empty()) {
        int u = Q.front(); Q.pop();     // dequeue
        for(int i = 0; i < gr[u].size(); ++ i)
        {
            int v = gr[u][i];
            if(dist[v] > dist[u] + 1) {
                dist[v] = dist[u] + 1;
                Q.push(v);              // enqueue
            }
        }
    }
}
```

Deque

`std::deque`

- Implements a deque data structure

```
#include <deque>          // or, automatically included by <queue>
```

- Deque have all features of `std::vector`, but in addition, $O(1)$ pushing or popping on the front (as well as on the back) is supported.
- In fact, deque is implemented by concatenation of two vectors
  - Note that memory locations of all elements are not continuous

(images here)

Deque

- `size()`, `empty()`, `clear()`, `operator []`, `resize()`
- `back()`, `push_back()`, `pop_back()`
- `push_front()`, `pop_front()` : $O(1)$

```
deque<int> Q(3, 1);                    // {1, 1, 1}
Q[0] = 3;                              // {3, 1, 1}
cout << ++Q.front() << endl;           // 4
Q.pop_front();                         // {1, 1}
Q.pop_back();                          // {1}
Q.push_front(-1);                      // {-1, 1}
int z = Q.front() + Q.back();          // 0
```

List

`std::list`

- implements a doubly-linked list

Priority Queue

`std::priority_queue<T>`

- Implements a priority queue (as a binary heap)
- Supports $O(\log n)$ push, pop operations
- All items of type `T` must be ordered – strict weak ordering (typically and by default, the less operator '<').
  - $x \not< x$ for all x
  - $x < y, y < z$ implies $x < z$ for all x, y, z
  - $x = y$ if and only if both $x \not< y$ and $y \not< x$ [8]

[8] Not necessary only for priority queue

Priority Queue - Usage

`std::priority_queue<T>`

- `bool empty()` : test if a queue is empty
- `size_t size()` : get the number of elements contained
- `void push(x)` : push `x` into PQ, in $O(\log n)$ time
- `T top()` : get a item with highest priority (= largest item)
- `void pop()` : pop out the top element, in $O(\log n)$ time

Therefore, it is a max-heap (given the less operator, by default)

Priority Queue - Speicifying Operator

- We can specify an ordering
  - It can be any deterministic function (binary relation), or any comparator class/object which has a `()` operator
  - The ordering must be a strict weak ordering

```
priority_queue<T, container, comparator>
```

- container is usually `vector<T>` (or `deque<T>`)
- comparator
  - by default, `less<T>`
  - If you use `greater<T>` instead[9], you get a min-heap
  - `less<T>` requires operator <,
    `greater<T>` requires operator >
  - Must callable with two `T`'s, like `comparator(v1, v2)`, and must be a strict weak ordering

[9] These are included in the header `<functional>`, but `<queue>` automatically includes it

Priority Queue - Example 1

```
int a[] = {9, 9, 0, 7, 6, 1, 6, 3};

priority_queue<int> Q;                                    // max-heap
for(int i=0; i<8; ++i) Q.push( a[i] );
while(!Q.empty()) { printf("%d\n", Q.top()); Q.pop(); }
// result : 9 9 7 6 6 3 1 0

priority_queue<int, vector<int>, greater<int> > Q;  // min-heap
for(int i=0; i<8; ++i) Q.push( a[i] );
while(!Q.empty()) { printf("%d\n", Q.top()); Q.pop(); }
// result : 0 1 3 6 6 7 9 9
```

Priority Queue - Example 2

```
struct myData {
    int id, key;
    myData(int id, int key) : id(id), key(key) {}
};

bool operator < (const myData &lhs, const myData &rhs)
{
//  return lhs.key < rhs.key;        // Bad
    if(lhs.key != rhs.key)
        return lhs.key > rhs.key;
    else return lhs.id < rhs.id;
}                                    // a strict weak-ordering

void test()
{
    // a max-heap w.r.t key
    priority_queue<myData> Q;
    Q.push( myData(1, 10) );
    Q.push( myData(3, 5) );

    printf("%d\n", Q.top().id); // 1
}
```

Priority Queue - Example 3 : Dijkstra's Algorithm

```
int n;
vector< pair<int, int> > gr[MAXN];      // (vertex, weight)
int dist[MAXN];                         // distance S->u

void dijkstra(int S)
{
    typedef pair<int, int> node;        // (cost, vertex)
    priority_queue<node, vector<node>, greater<node> > Q;   // minheap

    fill(dist, dist + n, 987654321);
    Q.push( node(0, S) ); dist[S] = 0;
    int visited = 0;

    while(!Q.empty() && visited < n)
    {
        int u = Q.top().second;
        if(dist[u] != Q.top().first) continue;  // why?
        Q.pop(); ++visited;

        for(int i=0; i<gr[u].size(); ++i)
        {
            int v = gr[u][i].first, w = gr[u][i].second;
            if(dist[v] > dist[u] + w)
            {
                dist[v] = dist[u] + w;
                Q.push( node(dist[v], v) );
            }
        }
    }
}
```

Set

`std::set<T>`

- Implements a set, supporting logarithmatic insertion, finding, deletion operations
- Balanced binary search tree (Red-Black tree) is used

```
#include <set>
```

- Data type should have a strict weak ordering !
  - By default, the less operator $<$ (`less<T>`)
  - Key comparison only depends on this ordering. That is, equality test is not done by == operator, even if it exists.
- `set` does not allow duplicated items

Set - Basic Usage

- Constructing a set

```
// creating an empty set
set<int> S;

// creating a set copying from some range
// for example, v : vector<int>, deque<int>, ...
set<int> S2( v.begin(), v.end() );
```

- `void clear()` : clear a set
- `size_t size()` : get the number of items contained in a set
- `bool empty()` : test if a set is empty or not

Set - Insertion

- `pair<set<T>::iterator, bool> insert(val)`
  - insert a single item `val`
  - returns a pair (the position inserted as an iterator, whether insertion is successful or not)
  - If another data of same key value as `val` was already contained, nothing will happen
- `void insert(from, to)` : insert multiple items from a specified range $[\texttt{from}, \texttt{to})$

```
set<int> T;
for(int i=1; i<=10; ++i) T.insert(i);
int f[] = {11, 12, 13, 14, 15};
T.insert(f, f+5);   // now T = {1, ..., 15}

T.insert(10);       // nothing happens
set<int>::iterator it = T.insert(0);
assert(*it == 0);
```

Set - Find

- `set<T>::iterator find(val)`
  - finds an element of key value `val`, and returns its location as an iterator
  - If there is no such item, then the `.end()` iterator is returned
- `size_t count(val)`
  - returns the number of elements of key value `val`
  - Actually, either 0 or 1

```
set<int> T;
set<int>::iterator it;
for(int i = -10; i <= 10; i += 2) T.insert(i);

printf("%u %u\n", T.count(-2), T.count(1));     // 1 0
it = T.find(4); assert(*it == 4);
it = T.find(7); assert(it == T.end());
if( T.find(0) != T.end() )                      // if 0 exists
    // ...
```

Set - Deletion

- `size_t erase(val)` :
  - erases the element whose value is equivalent to `val`
- `void erase(pos)` :
  - erases the element pointed by an iterator `pos`
- `void erase(from, to)` :
  - erases the elements in the range $[\texttt{from}, \texttt{to})$

```
set<string> S;
S.insert("A"); S.insert("B"); S.insert("C"); S.insert("D");

S.erase( S.rbegin() );  // (!) deletes the maximum

S.erase("A");           // S = {"B", "C"}

set<string>::iterator it = S.find("B");
S.erase( ++it );        // S = {"B"};

S.erase( S.end() ); // run-time error
```

Set - Iterate

- Iterating through all the elements

```
set<int> T;
int vals[] = {3, 1, 4, 1, 5, 9, 2, 6};
T.insert(vals, vals + sizeof(vals)/sizeof(int) );

// iterate through all the elements, in sorted order
// result : 1 2 3 4 5 6 9
for(set<int>::iterator it = T.begin(); it != T.end(); ++ it)
{
    printf("%d\n", *it);
//  *it = *it + 1;      // impossible (read-only)
}
```

- `set<T>::iterator` is bidirectional

```
set<int> T;
for(int i = -2; i <= 2; ++ i) T.insert(i);

set<int>::iterator it = T.find(0);
it --; printf("%d\n", *it);              // -1
it = T.find(10); printf("%d\n", *it);    // Runtime Error
```

Set - lower_bound, upper_bound

- `set<T>::iterator lower_bound(x)`
- `set<T>::iterator upper_bound(x)`
- return the foremost(lattermost) position where an item with key value key could be inserted without broking the orderings
- $O(\log n)$

```
set<int> S;
for(int i=-5; i<=5; i+=2) S.insert( i );

printf("%d %d\n", *S.lower_bound(-7), *S.upper_bound(-7)); // -5 -5
printf("%d %d\n", *S.lower_bound(3), *S.upper_bound(3));   //  3  5
printf("%d %d\n", *S.lower_bound(4), *S.upper_bound(4));   //  5  5
printf("%d %d\n", *S.lower_bound(7), *S.upper_bound(7));   // error
```

Set - Arbitrary Types and Ordering I

- Specify an ordering
  - Same as in priority_queue
- Example 1 : Predefined orderings

```
#include <functional>   // to use greater<T>, less<T>, ...

void test()
{
    set<int, greater<int> > S;      // must has '>' operator
    for(int i=1; i<=5; ++i) S.insert(i);

    set<int, greater<int> >::iterator it;
    for(it = S.begin(); it != S.end; ++it)
        printf("%d\n", *it);        // result : 5 4 3 2 1
}
```

Set - Arbitrary Types and Ordering I

- Example 2 : User-defined types

```
struct myPoint {
    int x, y;
    myPoint(int x, int y) : x(x), y(y) {}
};

bool operator < (const myPoint &lhs, const myPoint &rhs)
{
    if(lhs.x != rhs.x) return lhs.x < rhs.x;
    else return lhs.y < rhs.y;
}   // a strict weak-ordering

void test()
{
    set<myPoint> S;
    // Note : set<myPoint, less<myPoint> > requires operator '<'

    S.insert( myPoint(-3, 3) );
    S.insert( myPoint(2, -1) );
}
```

Map

`std::map<K, V>`

- Implements a mapping (a dictionary) 'key $\mapsto$ value'
- $O(\log n)$ insertion, finding, deletion operations
- Again, balanced binary search tree (Red-Black tree)

```
#include <map>
```

- Keys should be ordered (strict weak ordering) and each key is unique, but values are not needed to be ordered
- An item in a `map<K, V>` is a `pair<K, V>`

Map - Basic Usage

- Constructing a map

```
map<int, int> M;          // key : int -> value : int
map<string, int> T;       // key : string -> value : int
```

- `void clear()` : clear a map
- `size_t size()` : get the number of items contained in a map
- `bool empty()` : test if a map is empty or not
- Iterator points to an item(pair)

```
map<int, string>::iterator it = M.begin();  // M : map<int, string>
int key = it->first;                        // key
string value = it->second;                  // value
pair<int, string> &R = *it;
```

Map - Insert & Find

- `map<K, V>::iterator insert(KVpair)`
  - insert a key-value pair `KVPair` into a map
  - if there was no element of same key value, return the iterator of the item inserted
  - if there was already a element of same key value, return the iterator of the item previously existing (no changes on it)

```
map<string, int> M;
map<string, int>::iterator it;

it = M.insert( pair<string, int>("yuki", 181) );
printf("%s is %d cm.\n", it->first.c_str(), it->second);

it = M.insert( pair<string, int>("yuki", 182) );
printf("%s is still %d cm.\n", it->first.c_str(), it->second);

printf("there is %d person in the world\n", M.size());  // 1
```

Map - Insert & Find

- `map<K, V>::iterator find(key)`
  - find the item with key value `key` and return its iterator
  - If there is no such item, `.end()` iterator is returned
- `V& operator [] (key)`:
  - find the item with key value `key` and return a reference to its mapped value.
  - If there was no such item, a new item is created !
  - equivalent to `(this->insert(make_pair(x,V()))->second`

```
map<string, int> M;
map<string, int>::iterator it;

M["yuki"] = 181;                          // inserted
printf("yuki is %d cm.\n", M["yuki"]);

if( M.find("wook") == M.end() )           // if don't exist
    M["yuki"] = 180, M["wook"] = 177;
printf("yuki is now %d cm.\n", M.find("yuki")->second);
printf("total %d people", M.size());      // result : 2
```

Map - Basic Operations

- `void insert(from, to)`
- similar to `set::insert(from, to)`
- `size_t count(key)`
  - return the number of elements of key value `key`
  - Actually, either 0 or 1
- `size_t erase(key)` : erase the item of key value `key`
- `void erase(it)` : erase the item pointed by iterator `it`

```
map<int, int> M;
M[-1234] = 5678;
M[1234] = -5678;
M.erase(M.begin());                 // erase the smallest(-1234)
assert(M.count(-1234) == 0);
M.erase(M.find(1234));              // 1234 is erased
M.erase(1500);                      // nothing happens
```

Map - Iterate

- Iterating through all the elements

```
map<string, string> slaves;
slaves["ainu7"] = "Weonseok Yoo";
slaves["ryuwonha"] = "Wonha Ryu";
slaves["legend12"] = "Sukmin Koh";
slaves["altertain"] = "Taeyoon Lee";
slaves["libe"] = "Hyunhwan Jeong";
slaves["jongman"] = "Jongman Koo";
slaves["domeng"] = "Dokyung Lee";
slaves["astein"] = "Jinho Kim";
slaves["wook"] = "Jongwook Choi";

for(map<string, string>::iterator it = slaves.begin();
    it != slaves.end(); ++ it)
{
    cout << "Thanks to " << it->first << '(' << it->second << ")!";
//  it->first[0] = toupper(it->first[0]);       // not modifiable
    it->second = "Mr. " + it->second ;          // modifiable
}
for(map<string, string>::reverse_iterator it = slaves.rbegin();
    it != slaves.rend(); ++ it);
```

Map - lower_bound, upper_bound, equal_range

- Similar to `set`

Map - Pitfalls

- Note that '[]' operator creates a new node if such key does not exist

```
map<string, int> name2uid;
for(int i=0; i<100; ++i)
    name2id[ name[i] ] = i+1;        // insert 100 items

int NumberOfExistingNames = 0;
for(int i=0; i<100000; ++i)          // this loop is
{                                    // very slow
    if(name2uid[ pattern[i] ])       // due to this
        NumberOfExistingNames ++;
}
```

- In the above code, `.count()` or `.find()` should be used instead

Multiset, Multimap

- `multiset`, `multimap`
- Same as `set<T>` and `map<K, V>`, respectively, but the only difference is that `multiset`, `multimap` support duplicated keys
- Some Differences
  - `count()` can be arbitrary (nonnegative) integer
  - Now, `multimap` does not have `[]` operator anymore
  - `.erase()` does not recieve only a key value anymore
  - `.find()` returns a iterator to some item of specified key. To find multiple items (they are adjacent), use `.equal_range()`

Multiset, Multimap - Example

Multiset can be used as a priority queue which supports additional operations — finding/deleting a minimum, a maximum, or the element of specified key value

- A data structure which supports the following operations
  - insert(v) : insert an element of key value v
  - erase(x) : erase an element x (if several, only one is removed)
  - min() : get the element of smallest key value
  - max() : get the element of largest key value
  - find(v) : get a element of key value v
- will be useful in some problems$\cdots$.

Hash Set, Hash Map

- `unordered_set<T>`, `unordered_map<K, V>`
  - Similar interface as `set`, `map` but implemented by hash tables
- Used if keys can be hashed appropriately (rather than be ordered), for the sake of efficient search operations
  - Expected $O(1)$ but not so fast
- Available in C++0x
  - g++ $\geq$ 4.4.x
  - VC++ $\geq$ 10.0 (VS 2010)

```
#include <unordered_set>
#include <unordered_map>

using namespace std::tr1;

unordered_set<int> HashSet;
unordered_map<int, int> HashMap;
unordered_multiset<int> HashMultiSet;
unordered_multimap<int, int> HashMultiMap;
```

Generic Algorithms

Functions especially designed to be used on ranges of elements, typically specified by endpoint iterators.

```
#include <algorithm>
```

swap, min, max

std::swap

- Swaps two elements.

```
void loop(int x1, int x2)
{
    if(x1 > x2) swap(x1, x2);
    for(int x=x1; x<=x2; ++x) { ... }
}
```

```
void test(vector<int> &A, vector<int> &B) {
    A.swap(B);  // more efficient using pointer swapping tricks
}
```

std::min, std::max

- find a minimum(maximum) among two values of same type

```
minVal = min(minVal, now);
maxVal = max(maxVal, now);
int maxOfFour = max(max(a, b), max(c, d));
```

min_element, max_element

`std::min_element`, `std::max_element`

- returns an iterator pointing to the element with the smallest (largest) value in the given range[10]
- If there are more than one elements of smallest (largest) value, then the foremost one is returned
- $\Theta(n)$

```
for(int i=1; i<=n; ++i)
{
    int minpos = min_element(data, data+n) - data;
    int minval = data[minpos];
    /* ... */
    int maxval = *max_element(data, data+n);
}
```

[10] By default, `min_element` uses `operator <` and `max_element` uses `operator >`. User-specific comparison object(function) may be given

reverse, rotate

std::reverse

- reverses the order of the elements in the given range

```
string a = "Hello Algospot!";
reverse(a.begin(), a.end());
cout << a << endl;  // !topsoglA olleH
```

std::rotate

find

std::find

- iterator find(from, to, val)
- Returns the iterator of the first element whose value is same asHere, the == operator is used. value in the given range $[\texttt{from}, \texttt{to})$.
- If no such element, the past-the-end iterator to is returned
- $O(n)$

```
int a[] = {10, 1, 7, 4, 6, 8, 5, 3};
deque<int> d(a, a+8);
set<int> S(a, a+8);

printf("6 is %d-th.\n", find(a, a+8, 6) - a);   // 4
if(find(d.begin(), d.end(), 9) == d.end())
    printf("9 : not found in d\n");

// Note : this takes O(n), not O(log n)
printf("%d\n", *find(S.begin(), S.end(), 10) ); // 10
```

accumulate

std::accumulate

```
#include <numeric>
```

- Returns the result of accumulating all the values in the given range to an inititial value. (fold-left)
- By default, operator + is used, but specific binary functions can be used

```
vector<int> a;
int s1 = 0; for(int i=0; i<a.size(); ++i) s1 += a[i];
int s2 = accumulate(a.begin(), a.end(), 0);
```

```
vector<string> a;   // concatenate all !
string s1 = ""; for(int i=0; i<a.size(); ++i) s1 += a[i];
string s2 = accumulate(a.begin(), a.end(), string(""));
```

```
vector<double> pr;  // multiply all !
double p1 = 1.0; for(int i=0; i<pr.size(); +=i) p1 *= pr[i];
double p2 = accumulate(pr.begin(), pr.end(), multiplies<double>());
```

Sorting

`std::sort`

- `void sort(from, to)`
- Sorts the elements in the range $[\texttt{from}, \texttt{to})$ into ascending order
- Iterators should be random-accessible (e.g. `vector`, `deque`)
- Default ordering is the less operator <, but you can specifiy comparator which gives a strict weak ordering
- $O(n \log n)$ (usually, Quicksort)

```
int a[10] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
vector<int> b(a, a+10);
sort(a, a+10);                                  // ascending order
sort(b.begin(), b.end());                       // ascending order
sort(b.begin(), b.end(), greater<int>());       // descending order
```

Sorting - Example

How to sort items of user-defined type

```
struct slave {
    string name;
    int rating;

    // defining the < operator
    bool operator < (const slave &rhs) const {
        return rating < rhs.rating;
    }
};
```

```
// another way to define an operator
bool operator > (const slave &lhs, const slave &rhs)
    return lhs.rating > rhs.rating;
}
```

```
vector<slave> slaves = GetSlavesOfToday();
sort(slaves.begin(), slaves.end());
sort(slaves.begin(), slaves.end(), byName); // using comparator function
```

Sorting - Example

How to sort using a comparator function (e.g. indirect sort)

```
int red[] = {2240, 2611, 2257, 2225, 2736};
bool RatingAscending(const int &x, const int &y)
{
    return red[x] < red[y];
}
```

```
int idx[] = {0, 1, 2, 3, 4};
sort(idx, idx + 5, RatingAscending);    // result : {4, 1, 2, 0, 3}
```

Sorting - Example

How to sort using a comparator object – functor (e.g. indirect sort)

```
struct Comparator {
    int *array;
    Comparator(int *a) : array(a) {}

    bool operator () (const int &x, const int &y)
    {
        return array[x] < array[y];
    }
};
```

```
int red[] = {2240, 2611, 2257, 2225, 2736};
int age[] = {23, 29, 26, 22, 31};
int idx[] = {0, 1, 2, 3, 4};

// what if you want pass parameters to comparator ?
sort(idx, idx + 5, Comparator(red));    // 4, 1, 2, 0, 3
sort(idx, idx + 5, Comparator(age));    // 3, 0, 2, 1, 4
```

Sorting - other sorting functions

`std::stable_sort`

- same as `sort`, but preserves the relative order of the elements with equivalent values(keys)
- Here, equivalence $x = y$ means $x \not< y$ and $y \not< x$
- $O(n \log n)$ with merge sort

`std::partial_sort`

- `partial_sort(from, mid, to)`
- $[\texttt{from}, \texttt{mid})$ contains the smallest elements of entire range, and $[\texttt{mid}, \texttt{to})$ contains the remainings in any order.
- $O(n \log k)$, where $k = \texttt{mid} - \texttt{from}$.

Partitioning Functions

`std::nth_element`

- `nth_element(from, nth, to)`
- Finds k-th element ($k = \texttt{nth} - \texttt{from}$) x and rearrange the whole array such that
  - x is located at the position to which `nth` points,
  - smaller items than x precede x, and
  - greater items than x follow x.
- $\Theta(n)$, where $n = \texttt{to} - \texttt{from}$

`std::partitition`

`std::stable_partititon`

Unique - Removing Duplicated Entries

`std::unique`

- `unique(from, to)` or `unique(from, to, comp)`
- Removes the duplicate consecutive elementes from the given range, and returns the new end iterator
- does not alter the element past the new end
- Here, == operator is used for equality test (or, `comp` instead)

```
int t[] = {1, 1, 3, 3, 2, 2, 3, 1, 1};
vector<int> a(t, t+9);
vector<int>::iterator it;

it = unique(a.begin(), a.end());    // {1, 3, 2, 3, 1, 2, 3, 1, 1}
printf("%d\n", it - a.begin());     // 5

// (!) sort and erase out the duplicated items
sort(a.begin(), a.end());           // {1, 1, 1, 1, 2, 2, 3, 3, 3}
a.erase(unique(a.begin(), a.end()), a.end());         // {1, 2, 3}
```

Binary Search

`std::binary_search`

- `bool binary_search(from, to, key)`
- Performs a binary search on the range $[\texttt{from}, \texttt{to})$ to find out whether an item of value `key` exists or not
- Dependent on a strict weak ordering (same as previous), assuming all the items are sorted with respect to this ordering
- $O(\log n)$ if iterators are random-accessible

```
int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
vector<int> v(arr, arr + sizeof(arr)/sizeof(int));

sort(v.begin(), v.end());
for(int i=1; i<=10; ++i) {
    printf("%d %s\n", i,
        binary_search(v.begin(), v.end(), i) ?
        "exists" : "does not exist";
}
```

Binary Search

`std::lower_bound`, `std::upper_bound`

- `iterator lower_bound(from, to, key)`, `upper_bound(from, to, key)`
- return the foremost(lattermost) position where an item with key value `key` could be inserted without broking the orderings
- $O(\log n)$ if iterators are random-accessible

```
int arr[] = {10, 20, 20, 20, 20, 30, 30, 40};
vector<int> v(arr, arr + sizeof(arr)/sizeof(int));

// Case 1 : if key exists
printf("%d %d\n", lower_bound(v.begin(), v.end(), 20) - v.begin(),
      /* 1 5 */   upper_bound(v.begin(), v.end(), 20) - v.begin());

// Case 2 : if key does not exist
printf("%d %d\n", lower_bound(v.begin(), v.end(), 15) - v.begin(),
      /* 1 1 */   upper_bound(v.begin(), v.end(), 15) - v.begin());
```

Binary Search

lower_bound as a binary search

- One more comparison is required

```
int a[10] = {1,2,3,5,6,8,10,10,13,16};
vector<int> v(a, a+10);

// find 11
vector<int> p = lower_bound(v.begin(), v.end(), 11);
if(p != v.end() && *p == 11) {
    // 11 exists!
}
```

std::equal_range

- `pair<iterator, iterator> equal_range(from, to, key)`
- returns the largest subrange that includes all the elements of values equivalent to `key`.
- Also, all the items in the given range must be in sorted order

Permutations

`prev_permutation, next_permutation`

- Rearranges the elements in the given range into the lexicographically previous(next) permutation.
- Comparator may be given (by default, the less operator)

```
vector<int> a(n) = ...;

// lexicographically first
sort(a.begin(), a.end());

// brute-force attack (n!)
do {
    DoSomething(a);

    for(int i=0; i<n; ++i) printf("%d ", a[i]);
    printf("\n");
} while(next_permutation(a.begin(), a.end()));
```

```
1 1 2 3
1 1 3 2
1 2 1 3
1 2 3 1
1 3 1 2
1 3 2 1
2 1 1 3
2 1 3 1
2 3 1 1
3 1 1 2
3 1 2 1
3 2 1 1
```

Useful References

- TopCoder STL Tutorial

  `http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary`

- SGI C++ STL Programmer's Guide

  `http://www.sgi.com/tech/stl/`

- cplusplus.com C++ Reference

  `http://www.cplusplus.com/reference/`

Thank You Very Much

Any Questions?