STL and new stuff in C++11

Michal misof Forišek

Department of Theoretical Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava, Slovakia

February 2012
Standard Template Library (STL): the Swiss Army Knife (not only) for programming contests.

(Some weird things like those little scissors, but several very useful tools.)

Template: code with a variable instead of a type.

Example code template

```cpp
template<class T> void swap(T &a, T &b) { T c=a; a=b; b=c; }
```

Three basic parts of STL

- algorithms
- containers (data structures)
- iterators (smart pointers)
Standard Template Library (STL): the Swiss Army Knife (not only) for programming contests.

(Some weird things like those little scissors, but several very useful tools.)

Template: code with a variable instead of a type.

Example code template

```cpp
template<class T> void swap(T &a, T &b) { T c=a; a=b; b=c; }
```

Three basic parts of STL

- algorithms
- containers (data structures)
- iterators (smart pointers)
Standard Template Library (STL): the Swiss Army Knife (not only) for programming contests.

(Some weird things like those little scissors, but several very useful tools.)

Template: code with a variable instead of a type.

Example code template

```cpp
template<class T> void swap(T &a, T &b) { T c=a; a=b; b=c; }
```

Three basic parts of STL

- algorithms
- containers   (data structures)
- iterators   (smart pointers)
### STL: Containers

A bunch of data structures for free

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector</td>
<td>a scalable array</td>
</tr>
<tr>
<td>set</td>
<td>a balanced binary tree</td>
</tr>
<tr>
<td>map</td>
<td>a sorted associative array (a set of pairs key,val)</td>
</tr>
<tr>
<td>priority_queue</td>
<td>a heap</td>
</tr>
<tr>
<td>list</td>
<td>a linked list (use vectors instead if possible)</td>
</tr>
<tr>
<td>deque</td>
<td>a double-ended queue (very powerful)</td>
</tr>
<tr>
<td>pair</td>
<td>an arbitrary ordered pair</td>
</tr>
<tr>
<td>string</td>
<td>a convenient class for strings</td>
</tr>
</tbody>
</table>
# STL: Containers

## Advantages of using STL containers
- As efficient as possible – if you use the right one!
- You do not reinvent the wheel
- Less bugs
- Shorter, more readable code
- Less time spent on the implementation
- BUT: you still have to understand what’s going on

## Example: using a set
```cpp
set<int> S;
for (int i=0; i<1234567; ++i) S.insert(i);
S.erase(7);
cout << S.count(47) << " " << S.size() << endl;
```
What’s an iterator?

An iterator is a “smart” pointer.
The iterator “knows” what it points to.
- increased pointer: the next memory location
- increased iterator: the next element in the container!

All STL containers are the same

Each container has methods begin(), end().
These return two iterators that determine a **half-open** range.
Three equal expressions: empty() ; size()==0 ; begin()==end()

Iterating over all elements of a container:

```cpp
for (it = cont.begin(); it != cont.end(); ++it) process(*it);
```
And a bunch of algorithms for free

**min, max**: pairwise comparison

*min_element*,

**max_element**: convenient linear search

**swap**: exchange elements (NEVER use anything else)

unique, reverse,

rotate, fill,

**random_shuffle**: array manipulation

sort, stable_sort,

**nth_element**: sorting and searching
More algorithms for free

- lower_bound,
- upper_bound: generalized bsearch
  (also set/map methods!)
- next_permutation: quickly try all possibilities
  (also works with equal elements!)
- \_\_gcd: greatest common divisor (undocumented!)
next_permutation example

// generate all numbers with digits 1,1,3,4,7
#include <algorithm>
#include <iostream>
using namespace std;
int A[] = {1,1,3,4,7};
int main() {
do {
    for (int i=0; i<5; ++i) cout << A[i];
    cout << endl;
} while (next_permutation(A,A+5));
}
next_permutation example: k-element subsets!

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
int A[] = {0,0,0,1,1,1,1};
int main() {
    do {
        for (int i=0; i<7; ++i) if (A[i]) cout << i;
        cout << endl;
    } while (next_permutation(A, A+7));
}
```

Iterates over all 4-element subsets. (Almost) optimal time complexity!
Bitsets

Subsets of 0, ..., N − 1

<table>
<thead>
<tr>
<th>a subset</th>
<th>0, 3, 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>good/bad numbers</td>
<td>0, 1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>yes/no bits</td>
<td>1 0 0 1 0 1</td>
</tr>
<tr>
<td>the number</td>
<td>$2^0 + 2^3 + 2^5 = 41$</td>
</tr>
</tbody>
</table>

Bitwise operations

union: bitwise or
intersection: bitwise and
invert mask: bitwise xor
set $\{i\}$: bitwise shifts: $1 << i$
Tricks to compute size

```c
int size=0, tmp=subset; while (tmp) ++size, tmp&=tmp-1;
__builtin_popcount(subset);
```

Iterate over all subsets

```c
for (int subset=0; subset < (1<<N); ++subset) {
    for (int member=0; member<N; ++member) {
        if (subset & 1<<member) ...
    }
}
```

Important property: $\forall A$: all subsets of $A$ are processed before $A$

Alternative for larger sets: `bitset<N>` in STL.
What’s that?
A new C++ standard, approved in 2011. Previously called C++0x. (Was expected sooner.)

How to compile code in g++
Best results with

```
g++ -std=gnu++0x
```

(g++ 4.6 or lower; or use `-std=c++0x`, but lose some extensions)

If not allowed to set compiler options:
Parts of C++11 already available in namespace std::tr1.
Type inference (new “auto” keyword)

Previously:

```cpp
for (map<string, int>::iterator it = my_map.begin();
             it != my_map.end(); ++it) ...
```

Or maybe:

```cpp
for (typeof(my_map.begin()) it = my_map.begin();
             it != my_map.end(); ++it) ...
```

A new alternative:

```cpp
for (auto it = my_map.begin(); it != my_map.end(); ++it) ...
```
New syntax in C++11

Type inference (new “auto” keyword)

Previously:

```cpp
for (map<string, int>::iterator it = my_map.begin();
     it != my_map.end(); ++it) ...
```

Or maybe:

```cpp
for (typeof(my_map.begin()) it = my_map.begin();
     it != my_map.end(); ++it) ...
```

A new alternative:

```cpp
for (auto it = my_map.begin(); it != my_map.end(); ++it) ...
```
New syntax in C++11

Initializer lists now work everywhere.

Old style (already worked before)

```cpp
struct PlainStruct { double x; int y; };
PlainStruct ps = {0.47,42};
PlainStruct pss[] = { {0.47,42}, {0.2,9} };
```

New stuff (containers)

```cpp
std::vector<int> V = {10,20,30};
std::vector<int> W {40,50,60,70};
```

Even better

```cpp
struct Employee { string name; int salary; };
Employee getEmployee() { return {"Johnny",47}; }
```
New syntax in C++11

Initializer lists now work everywhere.

Old style (already worked before)

```cpp
struct PlainStruct { double x; int y; }
PlainStruct ps = {0.47, 42};
PlainStruct pss[] = { {0.47, 42}, {0.2, 9} };
```

New stuff (containers)

```cpp
std::vector<int> V = {10, 20, 30};
std::vector<int> W {40, 50, 60, 70};
```

Even better

```cpp
struct Employee { string name; int salary; }
Employee getEmployee() { return {"Johnny", 47}; }
```
New syntax in C++11

Range-based for cycle!

Works for C-style arrays. Works for containers.

```cpp
int A[] = {10, 20, 30};
vector<int> V = {40, 50, 60};
for (int x : A) cout << x << endl;
for (int x : V) cout << x << endl;
```

Works for initializer lists.

```cpp
for (int x : {2, 3, 5, 7, 11, 13}) cout << x << endl;
```

Even works with references!

```cpp
for (int &x : V) x *= 2;
for (int x : V) cout << x << endl;
```
Range-based for cycle!

Works for C-style arrays. Works for containers.

```cpp
int A[] = {10, 20, 30};
vector<int> V = {40, 50, 60};
for (int x : A) cout << x << endl;
for (int x : V) cout << x << endl;
```

Works for initializer lists.

```cpp
for (int x : {2, 3, 5, 7, 11, 13}) cout << x << endl;
```

Even works with references!

```cpp
for (int &x : V) x *= 2;
for (int x : V) cout << x << endl;
```
New syntax in C++11

Range-based for cycle!

Works for C-style arrays. Works for containers.

```c++
int A[] = {10, 20, 30};
vector<int> V = {40, 50, 60};
for (int x : A) cout << x << endl;
for (int x : V) cout << x << endl;
```

Works for initializer lists.

```c++
for (int x : {2, 3, 5, 7, 11, 13}) cout << x << endl;
```

Even works with references!

```c++
for (int &x : V) x *= 2;
for (int x : V) cout << x << endl;
```
Variadic templates allow us to define tuples (header file `<tuple>`):

**Tuple example**

tuple<string, int, Employee> job {"Wash dishes", 45, {"Bobby", 12}};
cout << get<1>(job) << endl;  // outputs 45
get<0>(job) = "Cook";        // get<> actually returns a reference
More cool tuple stuff

tuple<string,int,int> getStuff() { return make_tuple("Hello",4,7); } ...

string word;
int a,b;
tie(word,a,b) = getStuff();
cout << word << endl;

A neat little trick

int x=10, y=20, z=30;
tie(x,y,z) = make_tuple(y,z,x);
cout << x << " " << y << " " << z << endl;
More cool tuple stuff

```cpp
tuple<string, int, int> getStuff() { return make_tuple("Hello", 4, 7); } ... string word;
int a, b;
tie(word, a, b) = getStuff();
cout << word << endl;
```

A neat little trick

```cpp
int x=10, y=20, z=30;
tie(x, y, z) = make_tuple(y, z, x);
cout << x << " " << y << " " << z << endl;
```
Unordered sets (hash tables)

Unordered set example

```cpp
#include <unordered_set>

struct Employee { string name; int age, salary; };  
bool operator==(const Employee &A, const Employee &B)  
    { return A.name==B.name && A.age==B.age; }

struct Employee_hasher {  
    size_t operator()(const Employee &E) const { // note: ”const”!
        return hash<int>()(E.age) ^ hash<string>()(E.name);  
    }
};

int main() {
    unordered_set<Employee, Employee_hasher> workers;
    workers.insert( {"Johnny",23,42000} );
    workers.insert( {"Johnny",23,47000} );
    workers.insert( {"Thomas",27,42000} );
}
```
Unordered sets (hash tables)

Unordered set example

```cpp
#include <unordered_set>

struct Employee { string name; int age, salary; };

bool operator==(const Employee &A, const Employee &B)
    { return A.name==B.name && A.age==B.age; }

struct Employee_hasher {
    size_t operator()(const Employee &E) const { // note: ”const”!
        return hash<int>()(E.age) ^ hash<string>()(E.name);
    }
};

int main() {
    unordered_set<Employee, Employee_hasher> workers;
    workers.insert( {"Johnny", 23, 42000} );
    workers.insert( {"Johnny", 23, 47000} );
    workers.insert( {"Thomas", 27, 42000} );
}
```

Unordered sets (hash tables)

Unordered set example

```cpp
#include <unordered_set>

struct Employee { string name; int age, salary; };

bool operator==(const Employee &A, const Employee &B) {
    return A.name==B.name && A.age==B.age;
}

struct Employee_hasher {
    size_t operator()(const Employee &E) const {
        // note: ”const”!
        return hash<int>()(E.age) ^ hash<string>()(E.name);
    }
};

int main() {
    unordered_set<Employee, Employee_hasher> workers;
    workers.insert( {"Johnny",23,42000} );
    workers.insert( {"Johnny",23,47000} );
    workers.insert( {"Thomas",27,42000} );
```
Unordered sets syntax

Alternate syntax

```cpp
namespace std {
    template <> class hash<Employee> {
        public :
            size_t operator() (const Employee &E) const {
                return hash<int>()(E.age) ^ hash<string>()(E.name);
            }
    };
};
```
Unordered maps

Convenient associative arrays

unordered_map<string, int> age;
age["Billy"] = 17;

Almost infinite array, anyone?

unordered_map<long long, int> why_not;
Other stuff we shall mention quickly:

- long long int
- anonymous (lambda) functions
- a null pointer constant nullptr
- parsing right angle brackets (set<vector<int>>)
- raw and unicode string literals
- regex library
Conclusions

- A strong correlation between:
  - working, reliable code
  - short code
  - easy-to-maintain code
  - beautiful code

- Never reinvent the wheel.
- Extend your “vocabulary”.
- Programming is art, like poetry!
Conclusions

- A strong correlation between:
  - working, reliable code
  - short code
  - easy-to-maintain code
  - beautiful code

- Never reinvent the wheel.
- Extend your “vocabulary”.
- Programming is art, like poetry!