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

STL: Intro

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

```
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: Intro

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

```
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: Intro

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

```
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

vector:	a scalable array
set:	a balanced binary tree
map:	a sorted associative array (a set of pairs key,val)
priority_queue:	a heap
list:	a linked list (use vectors instead if possible)
deque:	a double-ended queue (very powerful)
pair:	an arbitrary ordered pair
string:	a convenient class for strings

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

```
set<int> S;
for (int i=0; i<1234567; ++i) S.insert(i);
S.erase(7);
cout << S.count(47) << " " << S.size() << endl;
```

STL: Iterators

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:

```
for (it = cont.begin(); it != cont.end(); ++it) process(*it);
```

STL: Algorithms

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

STL: Algorithms

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!)

STL: Algorithms

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));
}
```

STL: Algorithms

next_permutation example: *k*-element subsets!

```
#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 $0, \ldots, N-1$

a subset	{	0,			3,		5	}
good/bad numbers		0,	1,	2,	3,	4,	5	
yes/no bits		1	0	0	1	0	1	<- binary!
the number	2^0+			2^3+		2^5		= 41

Bitwise operations

union: bitwise or

intersection: bitwise and

invert mask: bitwise xor

set $\{i\}$: bitwise shifts: `1 << i`

Bitsets

Tricks to compute size

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

Iterate over all subsets

```
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`.

New syntax in C++11

Type inference (new "auto" keyword)

Previously:

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

Or maybe:

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

A new alternative:

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

New syntax in C++11

Type inference (new "auto" keyword)

Previously:

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

Or maybe:

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

A new alternative:

```
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)

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

New stuff (containers)

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

Even better

```
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)

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

New stuff (containers)

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

Even better

```
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.

```
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.

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

Even works with references!

```
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.

```
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.

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

Even works with references!

```
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.

```
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.

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

Even works with references!

```
for (int &x : V) x *= 2;
for (int x : V) cout << x << endl;
```

Tuples

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
```

Tuples

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;
```

Tuples

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;
```

Unordered sets (hash tables)

Unordered set example

```
#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

```
#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

```
#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

```
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 new stuff in C++11

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!