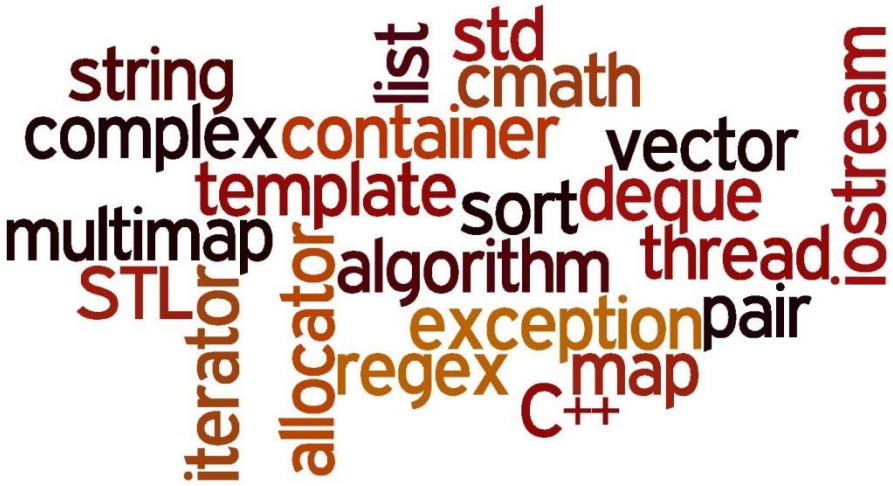
Karlsruhe Institute of Technology

GridKa School 2013: Effective Analysis C++ Standard Template Library

Introduction

Jörg Meyer, Steinbuch Centre for Computing, Scientific Data Management



What is STL? Why use it? What is boost?

- Standard Template Library (STL): standard libraries of C++
- generic, standardized, high performance, portable, bug-free, very useful, …
- try to use STL as much as possible (don‘t write your own container classes, string classes, …)
- STL offers huge functionality, this introduction will cover only a tiny fraction

- if some general purpose library is missing in STL, search for it in boost
  - www.boost.org
  - boost libraries work well with STL
  - some might enter STL in up-coming standards

Headers and Namespace

- headers of the STL have no `.h` ending
- examples:
  - `#include<iostream>`
  - `#include<string>`
  - NOT: `#include<iostream.h>`
- Names of standard C-libraries that are included in the C++ standard start with c:
  - `#include<cstdio>`
  - `#include<cmath>`
  - NOT: `#include<math.h>`
- STL classes and functions are in the namespace std
  - `using namespace std` (if you are sure there are no conflicting names)
  - never put `using` declarations in header files

Strings

- string: sequence of characters
- `std::string` defined in `<string>`
  - find, insert, replace, concat, append, compare
  - iterators, `[]`-operator
  - ==, !=, +=, …
  - memory management, buffer save

```
string s1 = "Hello";
string s2 = "World";
s1 += ' ' + s2;
cout<<s1<<endl; //Hello World
cout<<s1[1]<<endl; //e
cout<<s1.length()<<endl; //11
```

string constructors

```
string s1; //empty string
string s2 = “”; //empty string
string s3 = “myString“; //cstring
string s4 = s3; //copy c‘tor string
string s5(5,‘*‘); //“*****“
string s6(s3,2,3); //“Str“
char c[] = “cstring“;
string s7(c+1,3); //“str“
vector<char> v(c,c+sizeof(c)/sizeof(c[0]));
//{‘c‘,‘s‘,‘t‘,‘r‘,‘i‘,‘n‘,‘g‘,‘\0‘}
string s8(v.begin(),v.end());//“cstring\0“
```

strings and cstrings

- Conversion to cstrings

```
void f(const char* c) {
…
}
string s = "C++";
f(s.c_str());
```

- cstring functions
  - <cstring>: str[n]cpy, str[n]cat, strlen, str[n]cmp, strchr, strstr, strpbrk, str[c]spn

Regular expressions

- regular expressions:
  - powerful tool to do string manipulations, parsing, …
  - supported by all modern programming languages
  - example pattern for MAC addresses:
    - `^([0-9a-fA-F]{2}:){5}[0-9a-fA-F]{2}$`
    - matches 01:AB:7F:CC:42:D0
- C++ libraries:
  - boost: http://www.boost.org/libs/regex
    - not part of C++ standard
    - very useful libraries, extension of STL
    - some boost libraries might enter next version of STL
    - linking: g++ -lboost_regex
  - C++11
    - regular expressions now included in C++11 standard
    - not yet fully supported by gcc (http://gcc.gnu.org/wiki/Regex/Status)

Regular expressions (boost)

```
#include<iostream>
#include<string>
#include<boost/regex.hpp>

int main() {
  const boost::regex e("^(\\d\\d(?:\\d\\d)?)[-/](\\
d\\d)[-/](\\d\\d)$");
  const std::string     German_date("\\3.\\2.\\1");
  const std::string s("1990/10/03");
  std::cout<<regex_replace(s, e, German_date,
boost::match_default | boost::format_sed)<<std::endl;
  return 0;
} // 03.10.1990
```

Regular expressions (STL C++11)

```
#include<iostream>
#include<string>
#include<regex>

int main() {
  const std::regex e(R"(^(\d\d(?:\d\d)?)[-/](\d\d)[-
/](\d\d)$)");
  const std::string German_date("$3.$2.$1");
  const std::string s("1990/10/03");
  std::cout<<std::regex_replace(s, e, German_date);
  return 0;
} //03.10.1990
```

stringstream

- header: `#include<sstream>`
- streams: characterwise reading/writing from/to standartd input/output `(cin/cout)`, files `(fstream)`, strings `(stringstream)`, ...
- example: converting strings to integers:

```
int string2int(string const& number) {
  stringstream s(number);
  int ret;
  s >> ret;
  return ret;
}
int i = string2int("42");
string s = std::to_string(42); //int2string
```

Container

- container: homegenous collection of objects of same type
  - high performance, standardized interface, many algorithms, …
  - don‘t write your own container class!
  - (almost) never a reason to use arrays (use std::vector instead)
- list of standard containers:
  - **vector**
  - **list**
  - deque
  - stack
  - queue
  - priority_queue
  - **map**
  - multimap
  - set
  - multiset
  - string
  - valarray
  - bitset
  - unordered_map
  - unordered_multimap
  - unordered_set
  - unordered_multiset
  - forward_list

C++11

vector

- header: `#include<vector>`
- some constructors

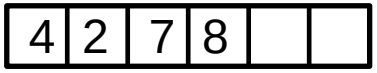
```
vector<myClass> v1; //empty vector for myClass
vector<double> v2(8,42.); //vector with 8 elements
                          with value 42.
vector<int> v3(10); //vector with 10 elements with
                    value int()
```

- first examples

```
int i=7;
v3.push_back(i); //append copy of i to v3
v3.size(); //11, number of elements in vector
int k = v3[0]; //first element of v3
v3[1] = 5; // assign new value to second element
vector<vector<int> > table; //2-dimensional table
int j = v3.at(2); //3rd element of v3; out-of-range test
```

vector: internal data structure

- size: number of elements stored in vector
- capacity: number of elements fitting in allocated memory
  - data are stored in dynamically allocated arrays
  - if the capacity is exceeded a new memory block gets allocated and all data are moved to new array



size: 4, capacity: 6

- data are guaranteed to be in a continous memory block

```
void f(int* array, int n) {
  for (int i=0;i!=n;++i) array[i]*=2;
}
vector<int> v(3,2);
f(&v[0],v.size());
```

&v[0]: address might change after push_back!

Does not work with vector<bool>!

Looping over vector elements

```
vector<string> v = {“only“, “C++“, “11“};
```

- using []-operator:

```
for (int i=0,N=v.size();i!=N;++i) {
  cout<<v[i]<<‘\n‘;
}
```

v[i]↔*(v.begin()+i)

- using iterators:

```
for (vector<string>::const_iterator it=v.begin();
it!=v.end();++it) {
  cout<<*it<<‘\n‘;
}
```

- using std::copy

```
std::copy(v.begin(),v.end(),
      std::ostream_iterator<string>(cout, "\n"));
```

- using for_each

```
for_each(v.begin(),v.end(),print);
```

```
struct printClass {
void operator() (string& s)
{cout<<s<<‘\n’;} } print;
```

Looping over vector elements C++11

- using iterators (1):

```
for(auto it=v.cbegin(); it!=v.cend();++it) {
  cout<<*it<<'\n';
}
```

- using iterators (2):

```
for(auto it=begin(v); it!=end(v);++it) {
  cout<<*it<<'\n';
}
```

- new **`for`** syntax:

```
for(auto& s : v) {
  cout<<s<<'\n';
}
```

- using for_each

```
for_each(v.cbegin(),v.cend(),[](string s)
 {cout<<s<<"\n"});
```

iterators

- used to iterate though containers
- access to container elements independent of container type
- begin(): points to first element (iterator or const_iterator)
- end(): points to past-the-end element
  - begin(), end() : half-open interval
  - never try to access *end()
- rbegin(): points to last element (reverse_iterator or const_reverse_iterator)
- rend(): points to theoretical element preceding the first element

```
vector<char> v = { 4, 3, 1, 5 }; //C++11 initialization
vector<char>::iterator it=std::find(v.begin(),v.end(),1);
std::find(v.begin(),it,6); //returns v.end()
```

list

- header: `#include<list>`
- double linked list: optimized for insertion and removal of elements
- same types and operations like vector, but `[],at(), capacity(), and reserve()`
- special: `splice(), merge(), sort()`
- front operations: `front(), push_front(), pop_front()`
- more: `remove(), unique(), reverse(), …`

```
list<myClass> l1(4,myClass(1));
list<myClass> l2(3,myClass(2));
l1.splice(l1.begin(),l2);//move elements from l2
```

→ `l1: { myClass(2), myClass(2), myClass(2), myClass(1), myClass(1), myClass(1), myClass(1) }`

→ `l2: {}`

vector vs. list

- always choose the right container for your purpose

	vector	list
random access	☺	☹
insert/remove	☹	☺
mem usage	☺	😐

map

- header: `#include<map>`
- map: associative container, dictionary, key-value pairs
- works like phone book (map<string, unsigned int>):
  - What‘s the number of “Jörg Meyer“? → fast
  - Who has number 123555? → slow
- maps are sorted: requires ‘<‘ operator for key
  - insertion slower than vector<T>::push_back()

```
map<string,unsigned> phonebook;
phonebook[“Meyer“] = 242347; //new entry
phonebook[“Smith“] = 424242;
phonebook[“Schmidt“] = 171717;
typedef map<string,unsigned>::const_iterator ci;
for(ci it=phonebook.begin();it!=phonebook.end();++it)
  cout<<it->first<<‘ ‘<<it->second<<‘\n‘;
```

map

- b