Using Objects

Object-Oriented Programming with C++

Safe way to manipulate strings?

`std::string`

The string class

- You must add this at the head of you code

```
#include <string>
```

- Define variable of string like other types

```
string str;
```

- Initialize it with string contant

```
string str = "Hello";
```

- Read/write string with cin/cout

```
cin >> str;
```

```
cout << str;
```

Assignment

```
char cstr1[20];
char cstr2[20] = "jaguar";

string str1;
string str2 = "panther";

cstr1 = cstr2; // illegal
str1 = str2;   // legal
```

Concatenation

```
string str3;
str3 = str1 + str2;
str1 += str2;
str1 += "a string literal";
```

Constructors (Ctors)

```
string (const char *cp, int len);

string (const string& s2, int pos);

string (const string& s2, int pos, int len);
```

Sub-string

```
substr (int pos, int len);
```

Modification

```
assign (...);

insert (...);
insert (int pos, const string& s);

erase (...);

append (...);

replace (...);
replace (int pos, int len, const string& s);

...
```

Search

```
find (const string& s);
```

File I/O

```
#include <ifstream>  // read from file
#include <ofstream>  // write to file

ofstream File1("C:\\test.txt");
File1 << "Hello world" << std::endl;

ifstream File2("C:\\test.txt");
std::string str;
File2 >> str;
```

Assignment 001 on PTA

- due in 2 weeks

A Quick Tour of C++

Make them sorted!

```
int main()
{
  int arr[] = {64, 25, 12, 22, 11};
  int n = sizeof(arr)/sizeof(arr[0]);

  selection_sort(arr, n);
  return 0;
}
```

- how to write a *practical* sorting algorithm?
  - overloading, template, comparator...
  - native type, user-defined type, inheritance...

STL

What is STL

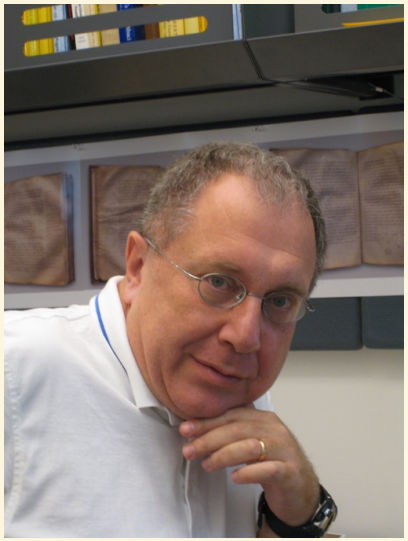
- STL = Standard Template Library
- Part of the ISO Standard C++ Library
- Data Structures and algorithms for C++

What is STL

Alexander Stepanov

Алекса́ндр Алекса́ндрович Степа́нов

Book: *From Mathematics to Generic Programming*



Why should I use STL?

- Reduced development time
  - Utilities well designed and written.
- Code readability
  - More meaningful stuff filled in one page.
- Robustness
  - Thoroughly used and tested.
- Portable code
- Maintainable code

The three parts of STL

- Containers
  - class templates, common data structures.
- Algorithms
  - Functions that operate on ranges of elements.
- Iterators
  - Generalization of pointers, access elements in a uniform manner.

Containers

- Sequential
- Associative
- Unordered associative
- Adaptors

Containers

- Sequential
  - `array` (static), `vector` (dynamic)
  - `deque` (double-ended queue)
  - `forward_list` (singly-linked), `list` (doubly-linked)
- Associative
- Unordered associative
- Adaptors

Containers

- Sequential
- Associative
  - `set` (collection of unique keys)
  - `map` (collection of key-value pairs)
  - `multiset`, `multimap`
- Unordered associative
- Adaptors

Containers

- Sequential
- Associative
- Unordered associative
  - *hashed* by keys
  - `unordered_set`, `unordered_map`
  - `unordered_multiset`, `unordered_multimap`
- Adaptors

Containers

- Sequential
- Associative
- Unordered associative
- Adaptors
  - `stack`, `queue`, `priority_queue`, ...

Using the `vector`

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> x;
  for (int a = 0; a < 1000; a++)
    x.push_back(a);

  vector<int>::iterator p;
  for (p = x.begin(); p < x.end(); p++)
    cout << *p << " ";
}
```

Basic operations for `vector`

- Constructor / Destructor
- Element access
- Iterators
- Capacity
- Modifiers

Basic operations for `vector`

- Constructor / Destructor
- Element access
  - `at`, `operator[]`, `front`, `back`, `data`, ...
- Iterators
- Capacity
- Modifiers

Basic operations for `vector`

- Constructor / Destructor
- Element access
- Iterators
  - `begin`, `end`, `cbegin`, `cend`, ...
- Capacity
- Modifiers

Basic operations for `vector`

- Constructor / Destructor
- Element access
- Iterators
- Capacity
  - `empty`, `size`, `reserve`, `capacity`, ...
- Modifiers

Basic operations for `vector`

- Constructor / Destructor
- Element access
- Iterators
- Capacity
- Modifiers
  - `clear`, `insert`, `erase`, `push_back`, ...

Basic operations for `vector`

More details here:

https://en.cppreference.com/w/cpp/container/vector

Pay attention to efficiency

- [todo] show this after copy ctor...
- Estimate and preserve the memory
- Avoid extra copies

Using the `list`

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main() {
  list<string> s;
  s.push_back("hello");
  s.push_back("world");
  s.push_back("stl");

  list<string>::iterator p;
  for (p = s.begin(); p != s.end(); p++)
    cout << *p << " ";
}
```

Using the `map`

- Collection of *key-value* pairs.
- Lookup by *key*, and retrieve a *value*.
- Example: a telephone book, `map<string,string>`

name	phone
"Charles Nguyen"	"(531) 9392 4587"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

Using the `map`

```
#include <map>
#include <string>
using namespace std;

int main() {
  map<string, float> price;
  price["snapple"] = 0.75;
  price["coke"] = 0.50;

  string item;
  double total = 0;
  while (cin >> item)
    total += price[item];
}
```

Using the `map`

More details here:

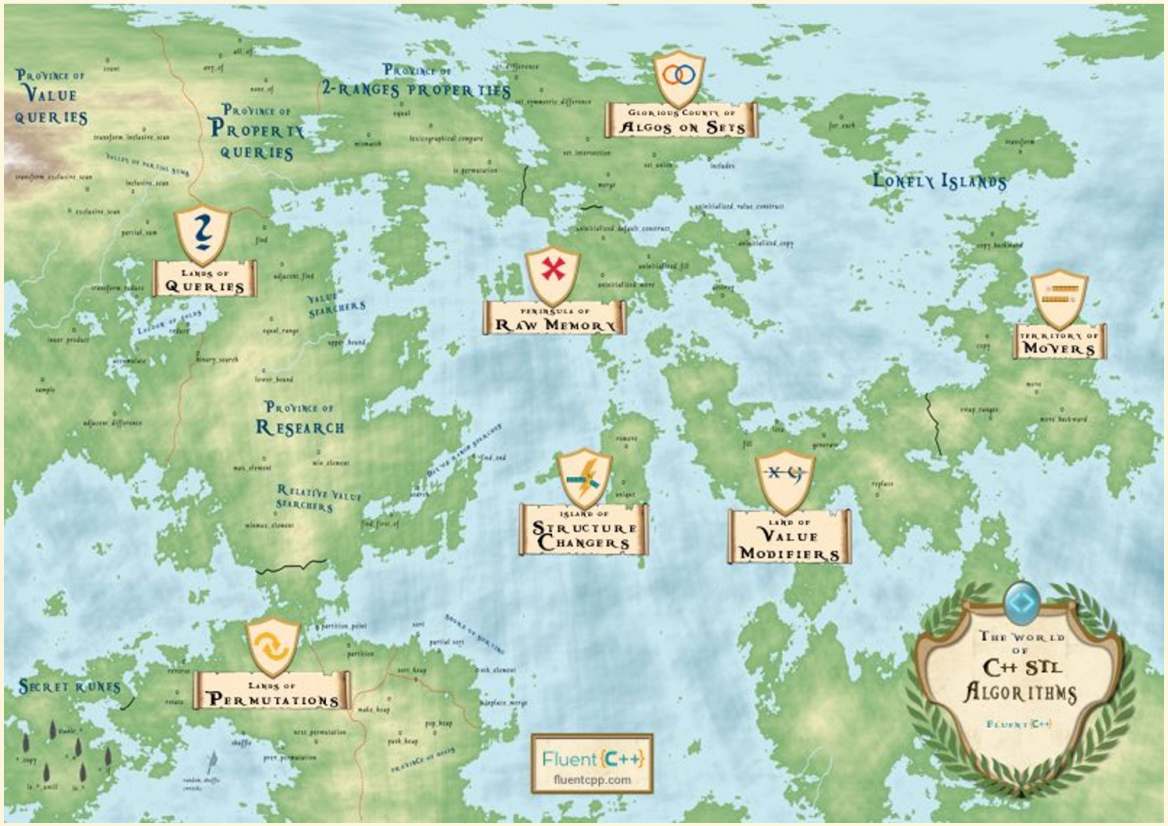
https://en.cppreference.com/w/cpp/container/map

Algorithms

- Works on a range defined as [*first*, *last*).
- `for_each`, `find`, `count`, …
- `copy`, `fill`, `transform`, `replace`, `rotate`, …
- `sort`, `partial_sort`, `nth_element`, …
- `set_difference`, `set_union`, …
- `min_element`, `max_element`, …
- `accumulate`, `partial_sum`, …
- …

The Algorithms World Map

[Good talk] https://www.fluentcpp.com/getthemap/



Typedefs

- Annoying to type long names
  - `map<Name,list<PhoneNum>> phonebook;`
  - `map<Name,list<PhoneNum>>::iterator finger;`
- Simplify with typedef
  - `typedef PB map<Name,list<PhoneNum>>;`
  - `PB phonebook;`
  - `PB::iterator finger;`
- C++11: `auto`, `using`

Using your own classes

- Might need:
  - assignment operator, `operator=()`
  - default constructor
- For sorted types, like `set`, `map`, ...
  - less-than operator, `operator<()`

Using your own classes

```
struct full_name {
  char * first;
  char * last;
  bool operator<(full_name & a) {
    return strcmp(first, a.first) < 0;
  }
};

map<full_name,int> phonebook;
```

Pitfalls - access safety

- Accessing an invalid element of a vector.

```
vector<int> v;
v[100] = 1; // Whoops!
```

- use `push_back()` for dynamic expansion.
- Preallocate with constructor.
- Reallocate with `resize()`.

Pitfalls - silent insertion

- Inadvertently inserting into `map<>`

```
if (foo["bob"] == 1) // create an entry "bob" silently
```

- Use `count()`, or `contains()`, to check for a key without creating a new entry.

```
if (foo.count("bob"))
if (foo.contains("bob")) // introduced in C++20
```

Pitfalls - `size()` on `list<>`

- This might be slow (linear time).

```
if (my_list.size() == 0) {
  ...
}
```

- Constant time guaranteed.

```
if (my_list.empty()) {
  ...
}
```

Pitfalls - invalid iterator

- Using invalid iterator.

```
list<int> L;
list<int>::iterator li;
li = L.begin();
L.erase(li);
++li; // WRONG
```

- Use return value of `erase()` to advance.

```
li = L.erase(li); // RIGHT
```

Iterators

- Connect containers and algorithms.
- Talk about it later
  - after templates and operator overloading.