STL

Object-Oriented Programming with C++

What is STL

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

Why should I use STL?

- Reduce development time.
  - –Data-structures already written and debugged.
- Code readability
  - –Fit more meaningful stuff on one page.
- Robustness
  - –STL data structures grow automatically.
- Portable code.
- Maintainable code
- Easy

C++ Standard Library

- Library includes:
  - –A pair class (pairs of anything, int/int, int/char, etc)
  - –Containers
    - vector **(expandable array)**
    - deque **(expandable array, expands at both ends)**
    - list **(double-linked)**
    - set **and** map
  - –Basic Algorithms (sort, search, etc)
- All identifiers in library are in std namespace

using namespace std;

The three parts of STL

- Containers
- Algorithms
- Iterators

The ‘Top 3’ data structures

- map
  - –Any key type, any value type.
  - –Sorted.
- vector
  - –Like c array, but auto-extending.
- list
  - –doubly-linked list

All sequential containers

- vector: variable array
- deque: dual-end queue
- list: double-linked-list
- forward_list: as it
- array: as “array”
- string: char. array

Example using the vector class

- Use "namespace std" so that you can refer to vectors in C++ library
- Just declare a vector of ints (no need to worry about size)
- Add elements
- Have a pre-defined iterator for vector class, can use it to print out the items in 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 << " ";
        return 0;
}
```

Basic vector operations

- Constructors

  vector<Elem> c;

  vector<Elem> c1(c2);

- Simple methods

  V.size( ) // num items

  V.empty( ) // empty?

  ==, !=, <, >, <=, >=

  V.swap(v2) // swap

- Iterators

  I.begin( ) // first position

  I.end( ) // last position

- Element access

  V.at(index)

  V[index]

  V.front( ) // first item

  V.back( ) // last item

- Add/Remove/Find

  V.push_back(e)

  V.pop_back( )

  V.insert(pos, e)

  V.erase(pos)

  V.clear( )

  V.find(first, last, item)

Pay attention to efficiency

- Estimate and preserve the memory
- Avoid extra copies

code & demo

List class

- Same basic concepts as vector
  - –Constructors
  - –Ability to compare lists (==, !=, <, <=, >, >=)
  - –Ability to access front and back of list
    x.front(), x.back()
  - –Ability to assign items to a list, remove items
    x.push_back(item),
    x.push_front(item) x.pop_back(),
    x.pop_front() x.remove(item)

Sample list application

- Declare a list of strings
- Add elements
  - –Some to the back
  - –Some to the front
- Iterate through the list
  - –Note the termination condition for our iterator

    p != s.end( )
  - –Cannot use **p < s.end( )** as with vectors, as the list elements may not be stored in order

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

int main( ) {
      list<string> s;
      s.push_back("hello");
      s.push_back("world");
      s.push_front("tide");
      s.push_front("crimson");
      s.push_front("alabama");
      list<string>::iterator p;
      for (p=s.begin(); p!=s.end(); p++)
                  cout << *p << " ";
      cout << endl;
}
```

maps

- Maps are collections that contain pairs of values.

- Pairs consist of a key and a value.

- Lookup works by supplying a key, and retrieving a value.

- An example: a telephone book.

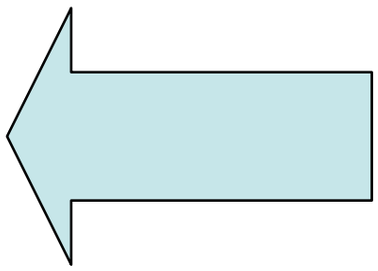
Using maps

- A map with strings as keys and values

:map<string, string>

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

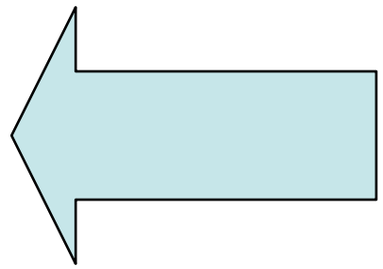
Example program



```
#include <map>
#include <string>
map<string, float> price;
price[“snapple”] = 0.75;
price[“coke”] = 0.50;
string item;
double total=0;
while ( cin >> item )
        total += price[item];
```

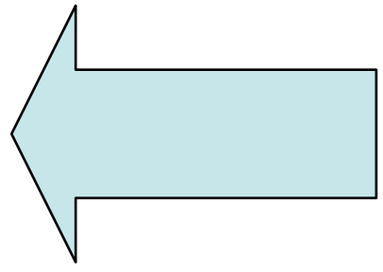
Example program

```
#include <map>
#include <string>
map<string, float> price;
price[“snapple”] = 0.75;
price[“coke”] = 0.50;
string item;
double total=0;
while ( cin >> item )
      total += price[item];
```



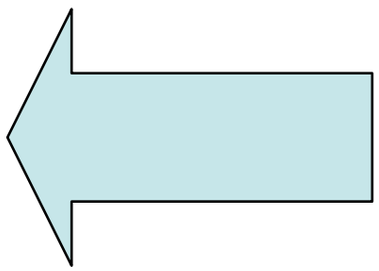
Example program

```
#include <map>
#include <string>
map<string, float> price;
price[“snapple”] = 0.75;
price[“coke”] = 0.50;
string item;
double total=0;
while ( cin >> item )
        total += price[item];
```



Example program

```
#include <map>
#include <string>
map<string, float> price;
price[“snapple”] = 0.75;
price[“coke”] = 0.50;
string item;
double total=0;
while ( cin >> item )
        total += price[item];
```



Algorithms

- Take iterators as arguments

```
list<int> L;
vector<int> V;
// put list in vector
copy (L.begin(),
      L.end(),
      V.begin());
```

code & demo

The World Map of C++ STL Algorithms

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;`
- Easy to change implementation.

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;`
- Easy to change implementation.
- C++ 11 : *auto, using*

Using your own classes

- Might need:
  - –Assignment Operator, `operator=()`
  - –Default Constructor
- For sorted types, like `map<>`
  - –Need less-than operator: `operator<()`
    - Some types have this by default:
      - –int, char, string
    - Some do not:
      - –char *

Example of user-defined type

- Sorted container needs sort function.

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

Pitfalls

- Accessing an invalid `vector<>` element.

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

Solutions:

- use `push_back()`
- Preallocate with constructor.
- Reallocate with `resize()`
- Check `size()`

Pitfalls

- Inadvertently inserting into `map<>`

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

Use `count()` to check for a key without creating a new entry.

```
if (foo.count("bob"))
```

Or `contains()` introduced in C++20

```
if (foo.contains("bob"))
```

Pitfalls

- Using `empty()` on `list<>`
  - Slow

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

  - Fast

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

Pitfalls

- 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
```

Other data structures

- set, multiset, multimap
- queue, priority_queue
- stack, deque
- slist, bitset, valarray