Lecture 27

- Standard Library
  - Containers (templates)
  - Streams (I/O facilities)

Standard Library

- Portable, type-safe, efficient
- Try to use as much as possible
- Heavy use of templates

Containers:
```
#include <vector>
#include <list>
#include <queue>
#include <stack>
using name space std;
```

Streams:
```
#include <sstream>
#include <fstream>
#include <iostream>
using namespace std;
```


Containers

- A container is an object that holds other objects (lists, vectors, associative arrays)
- Container-iterator model
  - Simple and efficiency
  - Commonality provided through iterators
- Different than the “based object” approach of Java (everything inherits from base class Object)

STL Containers

- Vector: basically a dynamic array

Iterators:
```
c.begin();   (first element of a container)
c.end();     (last element of a container)
++p          (next element)
```

```
vector<int> vi(10);    // create a vector of 10 integers
```

Each element is initizlied by default constructor (then one that has no arguments)

Vector continued

```
v[i]      // uchecked access (no range checking)
v.at(i)   // checks range (throws exception)
```

Vectors are typically large so in many cases they are passed as references:

```
void f1(vector<int>& );          // can modify vector
void f2(const vector(int>&);     // can not modify vector
void f3(vector<int> );           // uncommon inefficient
                                 // copy constructors will be called
                                 // for each vector element
```


Incrementally adding to a vector

- In C realloc would be used

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;
    while (cin >> buf) {
        if (buf == sentinel) return;
        // check new point
        cities.push_back(buf);
    }
}
```

Vector operates like a stack. New element is added at the end (same as:)

```
cities.insert(cities.end(), buf);
```

Also:

```
cities.erase(cities.begin()+1);
```


Sequences

- Other sequences are very similar to vector so I will not cover them (list, queue, dequeue etc)

```
vector<int> numbers;
numbers.push_back(1);
numbers.push_back(2);
...
vector<int>::iterator vi;
for (vi = numbers.begin(); vi != numbers.end(); ++vi)
{
    cout << "Number = " << *vi << endl;
}
```


Associative Containers

- A map is a sequence of (key,value) pairs (dictionary)

```
map<string, int> tbl;
map<string, MapSiste *, less<string> > neighbors_;

void f(map<string, number>& phone_book)
{
    typedef map<string,number>::const_iterator CI;
    for (CI p = phone_book.begin(); p != phone_book.end(); ++p)
        cout << p->first << "\t" << p -> second << '\n';
}
```

An example

input: nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250
output: sum for each item

```
void readitems(map<string, int>& m)
{  string word;
   int val = 0;
   while (cin >> word >> val) m[word] += val;   // subscripting with string
}
```


Operations

- find(li.begin(), li.end(), 42)
  - first occurance of value
- find_if
  - first match of predicate
- Many more - basically similar to the standard functional programming stuff but more primitive
- Higher-order functions can be simulated by function objects (not covered in this class)

Strings

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

string v1,v2;
v1 = "Hello"
v2 = " world";
string v3 = v1 + v2;

// convert to C style string
v3.c_str();
```

A string is a sequence of characters. Can be initialized by C-style string and C-style strings can be assigned to strings.

Try to use as much as possible Much safer, better, easier than the C-style mess of string handling

Streams

- Main idea: I/O conversion of objects of types such as int, char *, Employee into sequences of characters
- Ability to overload for user-defined types
- Formatting
- Files and Streams

One of my favorite ideas in C++

The usual way:

```
put(cerr, "x = ");
put(cerr, x);
put(cerr, '\n');
```

The C++ way:

```
cerr << "x = " << x << '\n';
```

User can overload operator<< and operator>>

Output of user-defined Types

```
class Complex {
public:
    double real() const {return re;}
    double imag() const {return im;}
}

ostream& operator<<(ostream& s, const complex& z)
{
    return s << "(" << z.real() << "," << z.imag() << ")";
}
```


Virtual output functions

ostream members are not virtual (for reasons of efficiency)
Sometimes you want to output an object for which only a base class is known. Solution:

```
class My_base P  { virtual ostream& put(ostream &s) const = 0; // write *this }
ostream& operator<<(ostream& s, const My_base& r)  {return r.put(s); }
class SomeType: public My_base
{
  ostream& put(ostream& s) const;
};
void f(const My_base& r, Sometype& s)
{ cout << r << s;     // calls right put()
}
```


Input is similar

```
int x;
string v;
cin >> x >> v;
```

Will read an integer and a string seperated by whitespace
(when reading from stdin enter must be pressed)

Input of user defined types:

```
istream& operator>>(istream& s, complex& a)
{
    ....
}
```

Formatting

- Control over how the object is printed
- Streams have formatting state
- State remains after function call

For integers:
cout.setf(ios_base::oct, ios_base::basefiled); // octal
For floating point numbers:
cout.precision(8); // 8 digits
cout.precision(4); // 4 digits
Another way (Manipulators) :

cout << setprecision(4) << angle << endl;

File Streams and String Streams

```
#include <fstream>
#include <sstream>
using namespace std;


ifstream from("test.txt");
ofstream to("out.txt");

while (from.get(ch)) to.put(ch);


from << "Hello world " << endl;
```

```
string message(int x)
{
ostringstream oss;
oss << "Message " << x << endl;
return oss.str();
}

void word_per_line(const string& s)
{
    istringstream ist(s);
    string w;
    while (ist >> w) cout << w << '\n';
}
```