Lecture 10

1. Lab4 problems
2. More about templates
3. Some STL

Lab4

- Template (size, type) vs. Input file
- Friend Function, increment, decrement
- Operator Overloading http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B *(see the rules there)*
- Ostream operator <<

  friend ostream& operator<<(ostream &os,const matrix &s); (why friend here?)

Dependent vs. Non-dependent

- Dependent names are names whose definitions are considered to depend upon the template parameters and for which there is no declaration within the template definition. They are resolved only when the template is instantiated. Those that are intended to refer to types or templates may require disambiguation.

- Non-dependent names are those names that are considered not to depend upon the template parameters, plus the name of the template itself and names declared within it (members, friends and local variables). They are resolved when the template is defined, in the normal way, and do not require disambiguation.

Indicating Typename

- In a template, the name of a member of another class that depends on its template parameter(s) (first<T>::pointer in this example, dependent on the T parameter) is a dependent name that is not looked-up immediately.

- To tell the compiler that it is meant to refer to a type and not some other sort of member, you must add the keyword typename before it.

```
template<typename T>
 struct first {
        typedef T * pointer;
};
 template<typename T>
class second {
        first<T>::pointer p; // syntax error
 };
```

Indicating Typename

- A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword typename.

- In short, if the compiler can't tell if a dependent name is a value or a type, then it will assume that it is a value.

```
template <typename T>
void foo(const T& t)
 { // declare a pointer to an object of type T::bar
      T::bar * p;
 }
struct S {
     typedef int bar;
};
 int main() {
     S x;
     foo(x);
}
```

Indicating Base Classes

- Base_func is inherited.
- However, the standard says that unqualified names in a template are generally non-dependent and must be looked up when the template is defined.
- Since the definition of a dependent base class is not known at that time (there may be specialisations of the base class template that have not yet been seen), unqualified names are never resolved to members of the dependent base class.

```
template<typename T>
 class base {
            public: void base_func();
 };
 template<typename T>
class derived : public base<T> {
            public: void derived_func() {
                        base_func(); // error: base_func not defined
                         }
 };
```

Specialisation Member Function

- According to the standard, you can declare a full specialisation of a member function of a class template.
- It's possible to specialise a member function of a class template without specialising the whole template.

```
template<typename T>
class my_class {
        public: bool func(); // other functions
};
 template<>
bool my_class<int>::func();
```

STL – Pair in Map

- the STL defines a class pair that simply stores 2 things.
- pair has 2 member variables (or fields): first is of type A and second is of type B. To create a pair, you need to manually specify what A and B are, like this.
- pair< string, double > p("pi", 3.14); //make_pair(“pi”, 3.14); p.second = 3.14159;
- The first line calls the constructor.
- To use pair, you need to #include <map>. (map: like Java‘s TreeMap - basically, a set of key-value pairs, sorted by the key.)

```
int min( int a, int b ) {
return a < b ? a : b;
}
(templates, chars, doubles,
strings, and anything else that has the
'<' operator defined on it. )

template< class C >
C min( C a, C b ) {
return a < b ? a : b;
}
```

```
template< class A, class B >
class pair {
public:
        A first;
        B second;
        pair( A a, B b ) {
            first = a;
            second = b; }
};
```

STL - Stacks, Queues, and Deques

- Stacks, queues, and deques (for double-ended queues) are simple containers that allows O(1) insertion and deletion at the beginning and/or end.
- You cannot modify any elements inside the container, or insert/remove elements other than at the ends. However, because they are templatized, they can hold just about any data types, and are tremendously useful for many purposes.
- A queue is a First-In-First-Out (FIFO) container, whereas a stack is Last-In-First-Out (LIFO).
- A deque is **both a stack and a queue.**
- A stack has pretty much all the methods of a queue, including **push(), pop(), size(), empty().** However, instead of **front(),** a stack accesses the top of the stack with **top().** And of course, **pop()** will retrieve the element at the **top (or end) of the stack, not the front.**
- Finally, a deque has both the features of a stack and a queue. It has the member methods **size() and empty(),** but instead of the **push(), pop()** combination, it now provides 4 different methods, all pretty much self-explanatory: **push_front(), push_back(), pop_front(), pop_back().**

STL - Stacks, Queues, and Deques

```
#include <stack>
#include <queue> // either one will provide deque
queue< int > Q;   // Construct an empty queue
 for( int i = 0; i < 3; i++ ) {
          Q.push( i );  // Pushes i to the end of the queue
          // Q is now { “0”, “1”, “2” }
}

int sz = Q.size(); // Size of queue is 3

while( !Q.empty() ) { // Print until Q is empty
            int element = Q.front(); // Retrieve the front of the queue
             Q.pop(); // REMEMBER to remove the element!
            cout << element << endl; // Prints queue line by line
       }
```