LECTURE 4: TEMPLATES + SINGLETONS

TEMPLATES OVERVIEW

- Very similar to java generics

```
public class Foo<E> { … }
Foo<String> x;
```

- Can be applied to:
  - functions

```
template<class T> void func(T val) { … }
```

  - classes

```
template<class T>
class Foo
{     // Use T anywhere in here.
}
```

  - individual methods of a non-templatized class

```
class Foo
{
      template <class T> void abc(T val) { }
}
```

SUBTLETIES

- In C++
  - Created by compiler when referenced
  - Must be in a .h file (no .cpp files!)
- Template specialization:
  - Useful when some types behave differently than the norm.
  - This *can* go in a .cpp file (if you just need it one place)

```
template<>
void func<std::string>(std::string val)
{
    // special version of template for strings!
}
```

SUBTLETIES, CONT.

- You can specialize individual methods of a templatized class

```
template<>
void Foo<std::string>::method()
{
      // Change just this method for strings
}
```

STATIC ATTRIBUTES OF T-CLASSES

```
template <class T>
class Foo
{
     static T* svar;
};
```

This variable must be defined / initialized. Usually done in a .cpp w/ a template-specialization

```
template<>
std::string * Foo<std::string>::svar = NULL;
```

SINGLETONS

- First (of several) **design patterns**
- Goals:
  - Ensure only one of them
  - Controlled global access
- Many trolls disparage this…but at least understand it.

EXAMPLE

```
// My preferred method
class FooSing
{
protected:
        static FooSing * the_only_one;    // init this elsewhere (.cpp)
public:
        FooSing(int x)
        {
                // Make sure no others!
                the_only_one = this;
        }
        ~FooSing() { the_only_one = NULL; }
        static FooSing * get_singleton_ptr()
        {
                return the_only_one;
        }
};

// Discuss the alternate method (and why it won't work here…)
```

YOUR GOAL

- Make a templatized-singleton class!