1 Topics

- Know how virtual methods/abstraction are used/useful.
- Smart pointers.
- Templating.
- Obviously design patterns. Likely ask you to either write out a specific design pattern or ask you one.
- Using correct visibilities, proper use of friends, proper use of abstract/virtual, even if not specifically mentioned.
- UML design.
- Recall big 5, not focus but you should recall.
- Iterators.
- NO BASH OR MAKEFILE.

2 Review

```
class A {
    public:
    A(int x, int y);
};

int main() {
    A a;
}
```

- Make sure to remember that if you overload the default constructor, you must modify it to either have default parameters or not accidentally call the default constructor w/o arguments!

```
class A {
    public:
    ~A();
};

class B : public A {
    public:
    ~B();
}
```

```
int main() {
    A* a = new B();
    delete a;
}
```

- If we do not use virtual on a function, then when we make a pointer that points to a subclass, it will run the function of type pointer - NOT the one that we are pointing to. Virtual will invoke the proper method for the subclass (IOW, tell the compiler to look at the ACTUAL type of the object, NOT the type of the thing that is pointing to said object).

```
class A {
};

class B : public A {
    int d;
};

int main() {
    A *a = new B();
    cout << a->d << endl;
}
```

- Remember you cannot access subclass fields with superclass variables!
- Always remember that you cannot call a class that has an abstract superclass unless you override ALL pure virtual methods!
- Note that smart pointers != no memory leaks. For example, let's say we had a struct w/ a unique pointer. If we changed it to another (unique) pointer, then the memory would be lost! Another possible problem is that smart pointers are not exception-safe. If there is a problem, it is very possible that the memory is lost when an exception is thrown. For example:

```
struct A {
    unique_ptr<int> pi;
    unique_ptr<double> pd;
    A(int* pi, double* pd) : pi{pi}, pd{pd}{}
};

int main() {
    A a {new int{42}, new double{3.14}};
}
```

- If pi works, but, say, pd fails, then pi will be lost.
- Given the following, rewrite to follow NVI:

```
class C_old {
    public:
        void f();
        virtual int g();
         virtual ~C();
};
////////////////////////////////
class C_new {
    public:
        void f();
        int g() {
            return doG();
        }
        virtual ~C();
    private:
        virtual int doG();
};
```

- T/F? In the Observer pattern, should Subject() delete all of the Subject's observers? —¿ Obviously, this is false. Subject DOES NOT OWN ANY OF THE OBSERVERS. Furthermore, we may still need the observers to live past the subject. Thus, it is false.
- Consider the following:

```
const int c = 5;
int main() {
    int *d = const_cast<int*>(&c);
    *d = 10;
    //...
}
```

- What is the behaviour? We realize that this will compile, but not run safely (undefined behaviour). This is as we told the compiler that d is a pointer to a constant integer that is const_casted — but in the VERY NEXT LINE, we try to modify the value of what it points to!
- We must realize that const_cast allows us to make a non-const pointer to const values, but we must abide by this ourselves as we tell the compiler to basically not worry about this (normally we must do const int *d = c;).
- Why can unique_ptrs not be copied? The reason they cannot be copied is due to their design - if we copy, then they will share values on heap. Deleting one will cause the second one to cause a double-free upon its own deletion.
- Given the following code, is this a valid use of unique_ptrs?

```
class C {...};
unique_ptr <C> f() {
    //...
    unique_ptr<C> p = make_unique<C> (...);
    //...
    return p;
}

int main() {
    unique_ptr<C> q = f();
}
```

- We see that q invokes the move constructor. This is valid! This is as when we use move constructor, we DO NOT have two unique_ptrs pointing at the same object, so move is valid! Thus, there are no problems with this code.