Overview

1. Introduction
2. Programming Principles
3. Refactoring
4. Design Patterns
   - Adapter Pattern
   - Facade
   - Strategy
   - Composite
   - MVC
   - Decorator
   - Factory
5. C++ STL
   - Containers
   - Algorithms
6. Templates and Lambdas
Details

Date: Tuesday, August 7th, 2018
Time: 4:00 - 6:30 PM
Location/Seat: DC 1351, see Odyssey for your seat information
Resources

Course website:
- Lectures
- Tutorials
- Assignments
- Design Patterns
Exam Writing Advice

- read through the whole exam **first**!
- consider writing the exam out of order; reverse order? highest score to lowest?
- determine the halfway point in terms of time and halfway point in terms of marks, aim to make those two match up
  - helps spread out your time better, otherwise you end up rushing at the end
- **breathe!**
  - before the exam starts just take a minute to breathe and relax, don’t try to cram in the last 5 minutes
See the midterm review slides on the tutorial page.

The question people struggled the most with was the Iterators question

- what do we need for the for range loop?
- see iterator.zip on the bottom of the Lectures page
- practice!
Open Closed Principle

Open for extension but closed to modification

- An abstract base class which is the interface for the client to interact with
- Extend the functionality through concrete classes
- Also known as "Program to an Interface, not an Implementation"
Open Closed Principle

- Flexibility
- Reusability
- Maintainability
We choose composition over inheritance because it is possible to modify the component at run-time while we are able to use it (whereas you can’t change your parent class at runtime.).
Favour Composition over Inheritance

Problem:
A set of furniture: chair, bed and table in different colors: red, green and yellow.
1. If using inheritance,...
2. If we add a new color now, ...
3. If using composition,...
Choose Inheritance:

- Using the entire interface of an existing class
- Using Polymorphism
Single Responsibility Principle

Encapsulate each changeable design decision in a separate module.

- Easier to maintain and debug
- Low coupling
Liskov Substitutability Principle

A derived class must be substitutable for its base class.

- Objects accept the base class’s messages.
- Methods require no more than base class methods.
- Methods promise no less than base class methods.
Law of Demeter

Method $A::m1$ can only call methods of

- $A$ itself
- $A$’s data members
- $m1$’s parameters
- any object constructed by $A::m1$

Law of Demeter tests encapsulation.
Refactoring

Sometimes, you will want to re-write your code, for one reason or another - perhaps to get rid of duplicate code, or to re-organize code to make it more concise, more modular, etc. Sometimes, you may *not* want to refactor - because there is a deadline coming up, or because the people you are working with are used to the code being a certain way (even if it is “wrong”).
Duplicate Code

What do you do if you have duplicate code...

- in the same class?
- in two related classes?
- in two unrelated classes?
Duplicate Code

In the same class: factor out common code into a private helper method.
In two related classes: put a helper method into a parent class.
In two unrelated classes: A little more interesting...
  - Maybe they should be related? Could they be children of a single abstract parent class?
  - Can the code be moved to a single class?
  - Can the duplicate code be encapsulated in its own class?
What do you do if you have one large class, or one long method?
Large Classes, Long Methods

Break it up! Each module should handle a specific piece of functionality, and each function should have a specific task. A long function should be broken up into multiple functions and helpers, each of which does something small and concise. A function with a long parameter list is also a good sign that the function is trying to do too many things at once.
An adapter pattern converts the interface of (a) class(es) into another interface the clients expect.

Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
Facade Pattern provides a unified - simplified interface to a complex subsystem or set of interfaces.

Facade Pattern provides a higher level interface simultaneously decoupling the client from the complex subsystem.

Subsystems are still accessible.
The Strategy pattern encapsulates different implementations of a function. The function you want to use can be encapsulated in an object, and overridden in the child classes. The main program can then use polymorphism to swap between different child classes (and therefore, different implementations of said algorithm) at runtime.
The Composite pattern is used to create recursive, tree-like structures in which nodes can have child nodes, which can themselves have child nodes, which can themselves have child nodes, etc. The parent class interface offers all operations that any of its children offer.
Combination of design patterns to decouple UI code from application code (the "model").

- The model takes care of the logic of the application, updating the view when it changes.
- The view uses the observer pattern to interact with the model, receiving a notification, and then requesting information from the model.
- The controller takes user input and sends commands to the model.

How many design patterns are used in MVC?
The Decorator Pattern provides a way for a base object to have extra features added and removed at runtime. This is achieved by creating a linked list of decorations that ends with the object you are decorating.

What’s the difference between decorator and composite design pattern?
Encapsulation: client code is not directly tied to specific classes, so classes can be changed, added, or refactored without changing client code.

Polymorphism: delegating the creation of an object to the right factory means that we can decide what kind of object to create at runtime.
The Factory Method pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

The Abstract Factory Pattern extends this idea by making an object responsible for creating families of related objects.
Containers

STL Containers let you store data in different ways depending on your needs.

- Arrays have a fixed size
- Vectors can grow from one direction
- Deques can grow from both directions

All three have random access, and are ordered based on insertion.
Containers

Linked-lists can efficiently insert items into the middle. They are also ordered based on insertion. Sets and Maps are not ordered based on insertion.
Iterator Hierarchy

- input/output iterator - can read/write and increment
- forward iterator - can increment with multiple passes (a == b => ++a == ++b)
- bidirectional iterator - can decrement
- random access iterator - can access any element in constant time
Algorithms

Typically do something to all the items in the range \([first, last)\).

- `for_each` applies a function to everything in a range (ignores the return values)
- `transform` is similar to `for_each`, but also takes a “destination” iterator, and uses it to store the return values.
- `sort` will sort elements in a range. If all you give it is a range then it will use `operator<` by default, but you can also give it a particular function or function object to use.
- **remove_if** will take a range and a boolean function. It will then move all of the elements that don’t satisfy the boolean to the end of the range, and return a pointer to the first invalid item.

- **copy_if** takes a source range, a destination iterator, and a boolean function. It takes all of the elements within the range for which the boolean returns true, and copies them over to the destination iterator.

- **find_if** takes a range and a boolean, and returns an iterator pointing to the first item in the range that satisfies the boolean.
A function Object, or functor, is a class that overloads its operator() to act like a function that can refer to data other than the iterated item. In other words, it can act as a function that can store state between calls.
Templates solve the problem of having overloaded functions/containers which have the same behaviour, but different types of their arguments/elements.

Uses duck typing: a template can be instantiated with type T if and only if type T has all the necessary functions that the template relies on. For example, the below templated function relies on type T having an overloaded output operator.

```cpp
#include <iostream>

template <typename T>
void print (T myT) {
    std::cout << myT << std::endl;
}
```

```cpp
int main () {
    print ("Hello, world!");
    return 0;
}
```
Lambda functions are small, in-line functions that can be used instead of a function or a function object.

\[
\text{[ capture list ] ( param list ) -> return type}
\]
\[
\{ \text{function body} \};
\]
Practice

- Draw each design pattern’s UML (including any necessary functions)
- Re-do old assignment questions using what we have learned about STL containers, algorithms, lambda functions, etc.
- For each STL algorithm, consider what type of iterator (and containers) can be used as arguments
- For each OO design principle, write a situation where it is violated and how to fix it
All the best on the final!