Final Exam Review

CS 247

University of Waterloo

cs247@uwaterloo.ca

August 7th, 2017
Overview

1. Pre-Midterm Stuff
2. Programming Principles
3. Design Patterns
4. C++ STL
Special Member Functions

- Default constructor
- Destructor
- Copy constructor/assignment operator
- Move constructor/assignment operator
- Equality operator
Entity vs. Value based ADT

- Equality operations
- Type conversions
- Copy operations
- Mutability
Namespace

- Package together related classes, functions and types in a common named scope
- Prevent names conflicts
- Global namespace, unnamed namespace
Exceptions

- Try-catch block
- Stack Unwinding: When an exception is thrown, the program begins to pop off activation frames from the run time stack until either a matching catch is found, or the stack is empty.
- Exception vs. Assertion
Resource Acquisition Is Initialization Idiom

- Resource is allocated inside an object’s constructor.
- Resource is deallocated inside an object’s destructor.
unique_ptr: supports exclusive ownership, disallow copy operations, defined to use move semantics

shared_ptr: supports shared ownership, uses reference count

weak_ptr: constructed with a shared pointer, but doesn't contribute towards the reference count
Interface Specification

Spec fields: the client’s abstract view of the ADT’s data members
Requires: a set of preconditions
Throws: names of all possible exceptions that can be thrown and under what conditions
Modifies: a list of all members modified by this method
Ensures: a description of how this method changes members
Returns: what this function returns
A Representation Invariant is a property of an ADT that must be true at all times for the ADTs state to be considered valid. A representation invariant is a boolean expression - something that evaluates to “true” or “false”.
An Abstraction Function maps the concrete code implementation to the abstract model of an ADT. This can be trivial (when the data looks exactly like what the user thinks), but sometimes not.
class IntSet {
    std::vector<int> els;
public:
    IntSet();
    ~IntSet();
    void insert(int x);
    void remove(int x);
    bool isIn(int x);
    int size();
    int getIndex(int x);
};
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.).
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,…
Favour Composition over Inheritance

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?
Large Classes, Long Methods

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.
A Singleton object is set up so that its constructor is private, disallow copy operations, and there is a static method to create the instance and get a reference to it.
#include <Database.h>
class Database {
    static Database database; // actual database object
    Database();

public:
    Database( const Database &database ) = delete;
    Database &operator=( const Database &database ) = delete;
    static Database &getDB() {
        return database;
    }
    // members to access database
};
Database &database = Database::getDB(); // user 1
Database &db = Database::getDB(); // user 2
Database &info = Database::getDB(); // user 3
The Template Method pattern sets up a non-virtual method with virtual components (or “helper” methods). Thus, the general structure of the function doesn’t change, but each subclass can customize the pieces.
Adapter Method

- 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 Method

- 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 Observer pattern allows observers to subscribe to a particular subject and receive updates, instead of periodically checking for updates itself.
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.
An Iterator provides a simple interface to visit every element in a particular collection. The client programmer does not need to know how the ADT is structured - they can just go from the beginning to the end. No matter what the container you’re iterating over looks like, you will need:

- `first()` - a way to reset the iterator to the beginning
- `hasNext()` - a way to check if the iterator has any elements left to iterate over
- `next()` - a way to get a pointer to the current object and move the iterator to the next item.
MVC

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?
Decorator

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?
Factory Method Pattern and Abstract Factory 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.
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.
Iterators

STL Iterators provide a slightly different interface from the Iterator Design Pattern. Specifically:

- `begin()` is a function of the relevant container that *creates* the relevant iterator.
- `hasNext()` doesn’t exist - instead, we compare our current iterator to one created by `end()` to see if we have reached the end.
- `next()` is replaced by `operator++` and possibly `operator->` and `operator*` to get a pointer to whatever the iterator is currently pointing to.
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 functor 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 functor is a class that overloads its operator() to act like a function that can refer to data other than the iterated item.
mem_fn_ref converts an object’s member function into a functor that can then be used in an algorithm (member function is called objects).

mem_fn converts an object’s member function into a functor that can then be used in an algorithm (member function is called on pointers to objects).

bind turns a function into a functor, while also “setting” the value of one or more of its parameters. That way, the resulting functor can be used in algorithms even if the original function had too many parameters to be compatible.
Lambda Functions

Lambda functions are small, in-line functions that can be used instead of a function or a functor.

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

- Draw each design pattern’s UML (including any necessary functions)
- Redo A3Q2 using functors and lambda functions
- 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
Good luck on the final!