CS202: Programming Systems

Karla Steinbrugge Fant

www.cs.pdx.edu/~karlaf

What to expect this term? Object Oriented Programming!

- The majority of the term will be spent introducing you to object-oriented programming while learning advanced C++ syntax.
- Experience object oriented design and programming
- Learn the difference between procedural abstraction and object oriented solutions
- Spend the term designing and programming with inheritance hierarchies, with the goal of solving problems efficiently: producing high quality, robust, maintainable as well as efficient object oriented solutions

What to expect this term? Object Oriented Programming!

- Programming assignments will focus on advanced data structures while at the same time accomplishing these other goals.
- Learn about C++'s function overloading, operator overloading, copy constructors, and be introduced to inheritance
- Then, the rest of the term compares Java to C++, where we will work through example projects in both languages
- Course requirements consist of five programming assignments in C++ and Java

What to expect of Programs

- Programming assignments will focus on advanced data structures and OOP
- 20% of each program's grade is based on the program style, comments, and documentation provided with the program
- 10% of each program's grade is based on a written discussion of how the unix debuggers (gdb, xxgdb, or ddd) assisted them in their development. Each assignment must have an accompanying **one full page (8 1/2 x 11)** debugger write up of your experiences!

Your programming assignments

- 10% of each program's grade is based on a written discussion of the major design consideration encountered when solving the specified problem.
- **Think in terms of analyzing your solution!** This means discussing the efficiency of the approach as well as the efficiency of the resulting code.

A term paper is part….

- In addition, each student will be required to submit a typed term paper.
- The paper must **explore how well your C++ programs have met our objectives to become object oriented programmers!**
- **Discuss how the designs meet the criteria set out for OOP, and how they can be improved**
- The paper must be a minimum length of 4 pages and a maximum of 7 pages (double spaced, 12 point font).
- Tables and sample code should be attached as exhibits and should not be part of the 4-7 page count

Object Oriented Programming

- Programs are populated by objects, which communicate with each other
- By delegating specific and narrow responsibilities to each object, the programmer can break down the programming problem into chunks and thus manage complexity easier
- Objects are characterized as having: *state, operations and identity*

Object Oriented Programming

- An object can store information that is a result of prior operations
- That information may determine how the object carries out operations in the future
- The colleciton of all information held by an object is the object's *state*
- An object's state may change over time, but not spontaneously
- State changes must be the consequence of operations performed on the object

Object Oriented Programming

- Objects permit certain operations and do not support others
- OO programs contain *statements* in which objects are asked to carry out certain operations
- It is possible for two or more objects to support the same operations and to have the same state, yet to be different from each other – each object has its own *identity*
- A class then describes a collection of related objects
- Question: Consider a word processor. Find at least five classes of objects that it manipulates.

Object Oriented Programming

- A very important aspect of OOP is to exploit similarities between classes.
- This often happens when one class is a specialized version of another class – where most operations are identical or similar, but a few differences do exist.
- For example, a system administrator of an electronic mail system. They receive messages and manage their message collections in the same way as all other users do. In addition, they have special ability to create new accounts (and remove accounts). The set of all administrators is a subset of the set of all users

Object Oriented Programming

- *Inheritance* is the way to model this "subset" relationship.
- A class, called a *subclass* or *derived* class inherits from a class (a superclass, parent, or base class), if its objects form a subset of the base class objects
- The objects in the subclass must support all operations that are supported by the superclass, but they may carry out these operations in a special way.
- They may also support additional operations
- Think about bank accounts – do checking and savings accounts differ in any way? Are they separate classes or instances of the same class? Is there anything common between them?

Object Oriented Design

- At the foundation of OOP is object oriented design
- The goal of object oriented design is to decompose a programming task into different *data types or classes* and to define the functionality of these classes
- Clearly, a structured approach to finding classes and their features will be helpful. Three helpful goals are:
  - Identify classes
  - Identify functionality between these classes
  - Identify the relationships between these classes
- This should be an iterative process. Understanding on aspect of a class may force changes to others

Object Oriented Design: ideas

- Remember, a class is simply a data type
- It corresponds to an "abstract data type" from CS163
- So, identify classes by looking for *nouns*
- For example, for a message system nouns are: mailbox, message, user, passcode, extension, administrator, mailsystem, menu
- BUT, just because you find "nouns" doesn't mean they are good choices for your classes! And, you may need others that provide functionality behind the scenes

Object Oriented Design: ideas

- A central role of object oriented design is to group each operation with one class
- Each operation must have exactly one class that is responsible for carrying it out
- This means you must be careful to *not* have one class act on the data of *another* class.
- For example, if we had a message class which allowed a message to be left, erased, or played. What would be wrong with adding this message to a mailbox of messages? *How could a message add itself to a mailbox? A mailbox, instead could add a message! This would have required a message to know the internal mailbox structure*

Object Oriented Design: ideas

- Always assume that an object has no insight into the internal structure of another object
  - Play close attention to this!
- All activities on objects other than itself must be achieved by performing an operation, not by direct manipulation of the internal data
- Always ask yourself "how can an object of this class carry out an operation on another class' data?
- This is a hard question – especially since you may not be aware at the design phase of all of the implementation details

Object Oriented Question

- Take the bank account situation, where you have checking and savings accounts
- For an automated bank teller program, describe all of the operations that should be supported and describe how these relate to the classes you are designing
- Should there be a “transfer to account” operation?
- If not – what other class might take on the responsibility of managing transfers

Finding Class Relationships

- Three relationships are common between classes:
  - Use or awareness
    - Creating/using objects of another class, passing objects of another class as arguments
    - Minimize the number of these relationships
    - One class should be as unaware as possible of the existance of another class! Eases future modification!
  - Aggregation or containment
    - “has a” relationship, a special case of usage
    - One object contains another object (a rectangle has 4 points)
  - Inheritance or specialization
    - “is a” relationship (a rectangle is a polygon)
    - Can lead to very powerful and extensivle designs

Hints to Follow!

- You should not use a class to describe a single object
- Every noun should not be a class
- Classes should collect objects with similar behavior
- Classes should be large enough to describe a significant set of objects, but, they should not be too large either
  - You may find it tempting to design classes that are very general, but that usually is not helpful
- Avoid public data and cluttered interfaces
- Operations in a class should be consistent with each other in regard to their: names, arguments, return values, and behavior
- Good use of inheritance requires finding a "common" set of data and/or functionality to all classes

Hints to Follow!

- Sometimes inheritance is difficult to see because you do not see what is common
  - Look for classes that have shared responsibilities and see if it is possible to define a base class that can assume those responsibilities
  - Recognizing common base classes is particularly important for OOP. Common code need only be provided once in the base class and is automatically inherited by the derived class (*inheritance*)
- Consider dropping classes that are not coupled with any other classes (coupled: using “is a”, “has a” “uses” relationships)

Hints to Follow!

- On the other hand, classes that are coupled with too many other classes are a sign of trouble ahead.
  - Classes that are "highly" coupled with other classes will force changes to other classes when you update the class in question
  - Of course, reducing coupling can require major reorganization! So, make EVERY effort at design time to minimizing coupling
- It should be as easy to create multiple objects of a class as it is to create one object
  - Question any design where there are single aggregations!

Hints to Follow!

- Split up classes with too much responsibility
  - Sometimes a top level class ends up with far too many operations because all commands are simply added to it
- Eliminate classes with too few responsibilities
  - A class with no operations is not useful. What would you do with its objects?
  - A class with only one or two operations may be useful, but you should convince yourself that there is really no better way of distributing the responsibilities
  - If another class can meaningfully carry out the task – move the operations there and eliminate the class
- Eliminate unused responsibilities (operations)
- Reorganize unrelated responsibilities – move it to a different class or add it to a new class

Hints to Follow!

- Express repeated functionality with inheritance
  - Is the same responsibility carried out by multiple classes?
  - Check whether you can recognize this commonality by finding a common base class and have the other classes inherit from it
- Keep class responsibilities at a single abstraction level
  - Every project exhibits a layering of abstractions
  - The responsibilities of a single class should stay at one abstraciton level and should not go to a higher or lower level
- Class names should be nouns
  - Don't use object is a class name – it adds no value
- Operations should be verbs or short sequence of words that contain a verb
  - There should be unique identifiers in the first 1-2 words of a name
  - Keep names consistent "don't mix "get" and "give"

Hints to Follow!

- Minimize operations that return pointers to new heap objects
  - Doing so would require the caller to (1) capture the returned pointer and (2) eventually deallocate the memory
- Never return a reference to a local stack object
  - The object is gone when the function exits
- Never return a reference to a newly allocated heap object
  - Deleting the memory is not intuitive
- Therefore, only return a reference to an object that exists prior to the function call

Hints to Follow!

- Elegance does pay off!
  - This term spend time with your design before throwing code at the problem. Each program will be building inheritance hierarchies – look for commonalities – resist the urge to hurry (it will only slow you down!)
- Break it down!
  - If the problem you are looking at is too confusing, try to imagine what the basic operation of the program would be (divide and conquer), given the existence of "black box" that handles the hard parts. That "black box" then can be a class that encapsulates the intended functionality

Hints to Follow!

- Remember CS163? Keep the philosophy of separating the class creator, from the client programmer (the class user).
  - The client should never know the internal data structures of how operations perform
  - Write your classes so that they are useable by others but still work well within the application's domain
- Make classes as atomic as possible
  - Give each class a single clear purpose
  - If your classes grow too complicated, break them into simpler ones
  - Avoid complicated switch statements – use polymorphism
  - Avoid a large number of operations that cover many different types of operations – consider using several classes

Hints to Follow!

- Watch for long argument lists
  - Break it down into using relationships passing objects (as const references whenever possible) to functions
- Don't repeat yourself!
  - If a piece of code is recurring in many operations, pull it out to be in a common base class
- Don't extend functionality by building derived classes
  - If an interface element is essential to a class it should be in the base class – not added onto the end
  - If you are adding operations by inheriting, you might want to rethink your design!

Hints to Follow!

- Realize the inheritance will complicate your design
  - Use it only if it is required by your design
  - Use it to express differences in behavior of classes that are derived from a common base class
  - The clearest designs add new capabilities to inherited ones
  - Poor designs remove old capabilities during inheritance without adding new ones
- Implementation rules for this course:
  - No global objects
  - Avoid magic numbers (hardwired into the code)
  - No string classes allowed
  - Use comments liberally
  - Remember your code is read much more than it is written!!