- **Basic Class Design**
  - *Goal of OOP: Reduce complexity of software development by keeping details, and especially changes to details, from spreading throughout the entire program.*
  - *Actually, the same goal as software design concepts throughout:*
    - Originally, "structured programming" means using the information-hiding abilities of subroutines (functions) to help organize the code; a subroutine can be modified internally without requiring the calling code to change.
    - Then, "modular programming" means breaking up the program into separate modules that can be developed and modified separately, and perhaps even re-used - like a library. A library is just a generally useful module.
  - *Definitions*
    - Client Code - the code that uses the classes under discussion.
    - Coupling - code in one module depends on code in another module
      - Change in one forces rewrite (horrible!), or recompile (annoying), of code in the other.
- **Two of the Four key concepts of OOP**
  - *Abstraction - responsibilities (interface) is different from implementation. Distinguish between interface and implementation.*
    - A class provides some services, takes on some responsibilities, that are defined by the public interface. How it works inside shouldn't matter.
    - Client should be able to use just the public interface, and not care about the implementation.
  - *Encapsulation - guarantee responsibilities by protecting implementation from interference*
    - Developer of a class can guarantee behavior of a class only if the internals are protected from outside interference. Specifying private access for the internals puts a wall around the internals, making a clear distinction between which code the class developer is responsible for, and which code the client is not supposed to know or care about.
  - *Both are ways of decoupling the client code from the class implementation:*
    - Don't need to know about the implementation;
    - Can't accidentally interfere with it or depend on it.
  - *A class is a software component that consists of some data and some procedures, coupled together, with an interface that abstracts and encapsulates the data and procedures, insulating the client from the details of implementation.*
    - Here, only concerned with ”Concrete" classes - no inheritance or polymorphism involved.
      - Objects interact with each other, contain each other, refer to each other.

        - Main program causes initial objects to be created, delegates work to objects as needed.

- **Two kinds of classes:**
    - *Objects that are in the problem domain.*
        - Managing Media: Records, Collections
        - Managing meeting schedule: Rooms, Meetings, Persons
        - Banking: Customers, accounts, banks
    - *Objects that support the domain objects (useful to implement them).*
        - E.g. String, Ordered_list, std::map<>, etc
        - Choose these to best get the work done.
        - But you have to understand what the actual work is first!

- **Guidelines for Designing Individual Classes**
  - *Designing Domain classes*
    - What kinds of objects are in the domain?
      - Start with thinking about the objects, then group into classes.
      - Which classes to they naturally belong to?
    - What characterizes each domain object?
      - Member variables - the data
      - Member functions - operations on the data
      - These are about the domain, not the implementation!
    - How are different kinds of objects related to each other?
      - Inclusion versus association -
        - Part-of relation versus "using" or "interacts with"
        - Do objects “know” about each other? one-way, or two-way?
          - One-way: An object points to another object, but which does not point back
          - Two-way: An object points to a second object which points back to the first
          - Classes with two-way associations are:
            - harder to work with, so do not use unless they represent the domain exactly
            - have to ensure cross-pointers are valid, and no dangling pointers left.
      - Relative lifetimes -
        - Do they exist independently of each other?
  - *Design a class by choosing a clear set of responsibilities for it.*
    - Make classes responsible for working with their own data.
      - Main code should delegate the work down into the classes that have the information.
    - Deciding which class or module should be responsible for the work.
      - Most general rule: who has the data? That component is probably the best one to do the work. Classes generally should be responsible for their data, and the work done with the data.
      - If client code is doing the work, something is probably wrong - rethink it!
    - If class responsibilities can't be made clear, then OOP might not be a good solution

- Lots of problems work better in procedural programming than in OOP, so there is no need to force everything into the OO paradigm.
  - Making this distinction is critical to understanding the difference between traditional procedural programming and OOP.
- But often, you just need to think about it more carefully.
  - Anthropomorphize it - e.g. in P2, imagine a person playing the role of each class object. What is her job? How does she interacted with other objects?

- Beware of classes that do nothing more than hold data, like a C struct type.
  - Sometimes you need simple “holders of data” - no associated functions or operations - if so, then it should not be a class.
  - Or, you’ve mis-assigned responsibilities - maybe this class should be doing the work, but some other component is doing it instead.
  - Is it really a "Plain Old Data" object, like C struct, or did you overlook something?
    - If it is a simple bundle of data, define it as a simple struct.
    - If there are functions that operate on the data, maybe they should be member functions, and maybe these objects really are responsible for something.
- Beware of classes that are only supposed to get instantiated once during the program execution.
  - Almost always, a class represents a kind of object that we will have many of, so a class that is supposed to correspond to only one object is suspicious.
  - There are important design patterns in which having only one object is the concept.
    - Used to clearly delimit responsibilities, separate concerns.
  - But can be over-used - unless you are using one of the design pattern concepts, don’t use a one-instance class - probably a bad idea.
  - E.g. maybe there is only one of these objects because it is trying to do a whole bunch of things - like replace the main function. Maybe it is a “god” class - see below.

- **Some specific rules for class design - if breaking these, there might be something wrong with the design.**
  - *Make all member variables private in each class*
    - Concept: Programmer of a class has to be able to guarantee that class will fulfill its responsibilities - do what he/she says it will do.
      - encapsulation - making member data private- is the basic step that makes this guarantee possible - prevents other code from tampering with the data.
    - No public member variables.
    - Beware of get_ functions that return a non-const pointer or reference to a private member variable - breaks the encapsulation!
    - Be careful if it is necessary to return a non-const reference or pointer to an item in a container member variable - even if client can't alter the container, might be able to alter the item in a way that violates design intent - or disorder the container!
  - *Resist the temptation to provide getters/setters for everything.*
    - Amounts to making the member variables public - why is this needed?
    - Similar problem: providing getters that return iterators pointing into a container member variable
      - if not const iterators, allows client code to modify private data
      - regardless, client has to know an implementation detail - coupled to the choice of container for the member variable - why is this something the client needs to know?
    - If you have to do this, something is probaby wrong with your design -
      - why does somebody else have to put data in and pull data out of the object?
      - why aren't the class's member functions doing the work?
  - *Put in the public interface only the functions and declarations that clients can meaningfully use.*
    - Functions that are only helpers for the implementation should be private.
    - constants and enum types:
      - should be private in the class if only the implementation needs them
      - public in the class if the class client needs them
      - at top level of header file ony if needed independently of any class - if so, why are they in the same header as the class declaration?
  - *Friend classes and functions are actually part of the public interface of the class, and belong with the class.*

- Friend class or function must be part of the same module or component.
- Most clear if declaration and implementation is in the same .h and .cpp files.
- A class developer should declare a class or functions to be a friends only if he/she/they are also responsible for, and have control over, that class or function.
- If class A uses class X for its internal work, and the client shouldn't have to see class X, then consider declaring class X as a private member of class A rather than have X be visible to the client with a friendship relation to A.

▼ *Make member functions const if they do not modify the logical state of the object.\*

- Don’t use mutable to fake a const member function in this course.

▼ *Make a class fully responsible for initializing itself with constructor functions.*

- It is error-prone and bad design if the client has to "stuff" initial data into the object.
- ▼ Take care that all member variables get a good initial value.
    - Only supply these where necessary - if the member variable is a class type, the compiler will call its default constructor for you.
- ▼ Take care if the class design requires post-creation intialization - e.g. an initialization function called after construction
    - e.g. a set_pointer() function to store the address of some other object in a pointer member variable.
    - Sometimes required for designs in which objects point to each other
    - Use initialize to nullptr in constructor, and then assertions to make sure pointer has been set to a real value before any code tries to use it.

▼ *Do not write constructors, assignment operators, or destructors when the compiler-supplied ones will work correctly.*

- ▼ Unnecessary code is simply places for bugs to hide!
    - Especially when revisions are made!
    - E,g, did you remember to fix the copy constructor when adding another member variable?
- Let the compiler do the work as much as possible - it will automatically respond to changes in the code.

▼ *Explicitly decide whether the compiler-supplied “special member functions” (the destructor and the copy/move functions) are correct, and let the compiler supply them if so.*

- Do not write code that the compiler will supply.
- Unnecessary code is an unnecessary source of bugs.

- ▼ *Try to follow the “Rule of five or zero” - either explicitly declare all five of the special member functions, or declare none of them and let the compiler supply them automatically.*
    - “Declare” here means to either declare and define your own version of the functions, or declare what you want the compiler to do with =default or =delete.
    - If you have to write even one of these functions for some reason, explicitly declare the status of the rest of them to avoid confusion or possible undesired behavior.
    - If you have to write your own destructor function to manage a resource (like memory), you almost certainly have to either write your own copy/move functions or tell the compiler not to supply them (with =delete).
    - In writing a copy constructor, remember to copy over all member variables - a common error.
- ▼ *If copy or move operations are not meaningful for a class, explicitly prevent the compiler from supplying them.*
    - For example, to enforce the concept that objects in the domain are unique.
    - In C++11, disable compiler-supplied copy and move functions with the =delete syntax.

- **A couple of General DO NOT rules**
  - *General Don’t: Don’t overengineer -*
    - Overengineering - a more complex solution than necessary.
      - Often a result of anticipating future needs inappropriately.
        - “Yes, it is more complex, but if I do it this way, then in the future, it will be easier to do yada-yada.” - but will this be needed?
        - Problem: the code is harder to work with NOW, and you don’t actually know whether you will need to do the future thing.
        - “YAGNI” principle - “you aren’t going to need it”.
      - Wisdom of the gurus: If the code is a simple solution that is clear and well-designed, it will be easy to change it in the future if necessary.
      - So design and code a current solution well, instead of making a mess trying to anticipate an unknown future.
    - Another reason: Just getting complex without thinking through what the responsibilities of the classes really are - misdelegating, misassigning - the result is simply unnecesssarily complex.
  - *General Don’t: Don’t create heavy-weight, bloated, or "god" classes - prefer clear limited responsibilities.*
    - If a class does everything, it is probably a bad design. Either you have combined things that should be delegated to derived classes or peer classes, or you have misunderstood the domain.
    - Example: In project 2’s restore function, the work of reading and interpreting the data file, and creating the right objects or relationships, was delegated to the Person/Meeting/Room or Record/Collection classes, which do all the work, and just signal a problem if they can’t. Only thing the main module knew is that Person/Record data comes first, then Room/Collection data. If a member variable was added to Person/Record, only the Person/Record class class would need to be changed.
      - Contrast with a god-like main module that knows how Persons and Meetings and Rooms / Records and Collections are structured, and what the details of what data file looks like - it reads the data, validates it, creates objects, and stuffs the data into them from the outside while they just sit there passively. If a member variable is added to Person/Record, both the Person/Record declaration and the main module code would need to be changed.

- **Some Top-level Design suggestions**
- **Do the class design work at the level of the public interfaces, not the private implementations.**
  - *Don't get bogged down in implementation details like "I can do with this with a map container and a deque!"*
  - *Think only about what the class responsibilities are and what they do in their public interfaces:*
    - Class X is responsible for …., class Y for ….
    - When an X object needs … it calls the public member function … of the appropriate Y object with … as parameters, which returns …
  - *Try writing pseudo code just for the interactions between class objects through their public interfaces.*
  - *Keep this up until you can't stand it any more, only then make implementation choices and write the code.*
- **Continue design thinking until you have thought of at least two reasonable ways to solve each design problem.**
  - *"Reasonable" here means "not obviously stupid."*
  - *Don't just jump on the first design you think of and hack it out.*
  - *All designs are imperfect - they all involve trade-offs. They are good in some ways, bad in others.*
  - *A good design is good in the most important ways, and bad in the less important ways.*
  - *But there might be more than one good design - just different in the specific tradeoffs.*
  - *You can't make an intelligent choice if you have only thought of one design - there could be another, better, simpler one.*
- **If the implementation turns out to be difficult or confusing to code, the reason might be a defective design - step back and rethink the design!**
  - *Generally, good designs code easily, but bad designs tend to make such a mess of the code that it becomes harder and harder to complete it and debug it. Stop before it gets worse!*
  - *It is impossible to anticipate every issue at the design stage - sometimes you do not discover important facts about the problem until you try to implement the design.*
    - If this happens, do not panic! Go back and rethink the design using what you now know about the problem. This should be relatively easy because the original design thinking is still helpful. But perhaps you made some assumptions or overlooked some issue; be prepared to revise the design, or consider a new one.
  - *Fixing a bad design is usually a better strategy than try to push through the messy implementation of a bad design.*

- Even if none of the previous code can be used, the coding of the new design goes faster once the design is clarified or corrected - you understand the problem better and are more clear about what has to be done.