CS302
Programming Methodologies and Software Implementation

General Background Information

Background for CS302 Programming Assignments

Vision for all CS302 Assignments:

In CS302, all five of our programs will focus on developing solutions using abstractions. Therefore, unlike CS163 were we focused on Data Abstraction, now we will turn our focus to developing complete applications using a set of classes working together to solve the problem. This may use object oriented programming techniques, generic programming or even functional programming. The key idea is to break down the problem into small pieces – maybe building blocks – and assign those responsibilities to individual classes. Then, as a team these set of classes create an solution. You will notice that the assignments are practical problems that face the real world. In reality, each of these problems could take many "man-months" to create. Your job will be to create an application program using a sequence of classes to show a solid understanding of classes and data structures. Focusing on just the data structures is not enough. Think this term about the application and how it could be set up if this was a real-world application. You will want to focus on how to design classes that are well structured, efficient, that work together, and where each class has a specific **"job"**.

Whenever you write a class in CS302 – you need to ask yourself "What is the purpose of this class". If it doesn't have a reason to exist, then it probably shouldn't be a class. Or, if its responsibilities are too broad, then it should be broken down into further classes. This is key.

Each class should have a specific job. *The responsibilities should be the public member functions and the data that they work with should be the private (or protected) data members.* The classes that you design should work in conjunction with one another – not in isolation of one another. This represents a big change. This means that you will really need to focus more on the application than you may have done in CS163. In fact, you no longer need to limit I/O to the client program…instead you will want that "job" to be done where it makes the most sense!

If you find you are writing a class that is always using "set" and "get" functions of another class, ask yourself why isn't that other class doing the job to manipulate the data for you?! This is how we use classes to build abstractions.

Structs do not perform a job and may not be used. Public data is also to be avoided. ***Neither should be done!***

Working with your Virtual Group

- For every programming assignment, there will be an online discussions that corresponds to that assignments' cycle:
  - In the first week, the discussions will ask you to focus on the design and UML diagram and plan your classes, the relationships between those classes and the prototypes
    - Make your original post by Tuesday 7pm
    - Respond to posts by Thursday 7pm

  - In the second week, we move to discuss how to integrate the data structures into the design, build the application and critique your design from the first week.
    - Share ideas by Tuesday 7pm
    - Respond to posts by Thursday 7pm

  - Use the discussion forum to capture your thoughts, perspectives, ideas, and revisions as you work together on planning how to develop your object oriented solution. This activity is discussion-based, meaning you will participate through a collaborative exchange and critique of each other's ideas and work.

  - **This is a graded activity** and it is expected that you will embrace this process. Everyone is expected to participate in the discussion and through it you will be encouraged to refine your ideas and plans as that discussion develops**.**

  - **It is important to make your initial posts and subsequent responses in a timely manner. There is no such thing as getting late credit after the fact. You are also <u>expected to make multiple posts</u> during each stage**

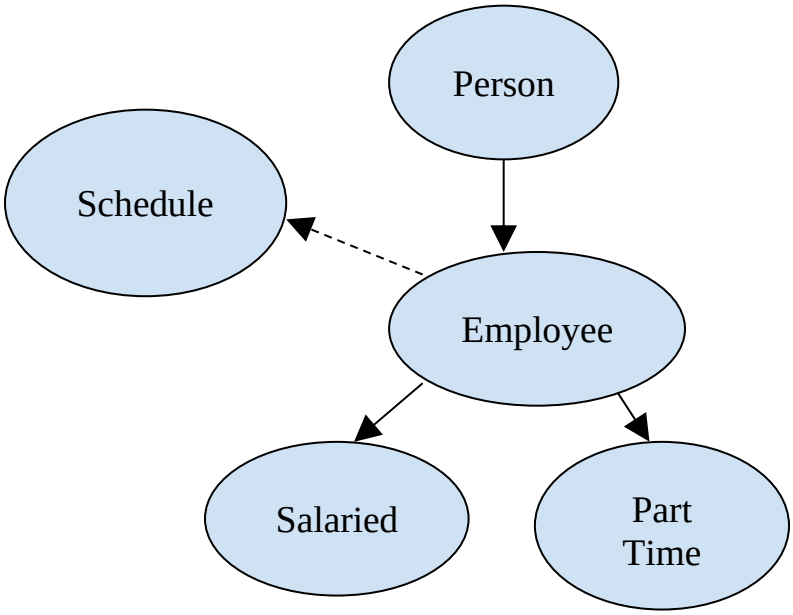
  - The purpose of the Virtual Group discussions is to collaboratively begin planning before physically programming. In CS302, we want to shift our focus to creating solutions that are comprised of multiple classes to solve the assigned problem.

  - Your discussions could include any of the following points – the more you investigate, the clearer your design::
    a. It should cover the major design considerations
    b. Discuss what classes you are intending to create
    c. Discuss the relationship between those classes
    d. Discuss what methods are needed to avoid excessive use of "getters"
    e. Outline the functions that you need for each class and how they will be used by other classes in your design
    f. **Create and share a UML diagram with your Virtual Group**

Creating a UML Diagram

1. UML diagrams are useful to communicate the relationships between class objects
2. The UML diagrams should show the detail of what classes are expected and how those classes will interact.
3. First, start by listing out all of the "nouns". Group those nouns into classes and build relationships between those classes. You might start with a simple diagram such as this to map out your ideas.
4. This first diagram should be used to raise the question as to what functions one class will need to use of another class.

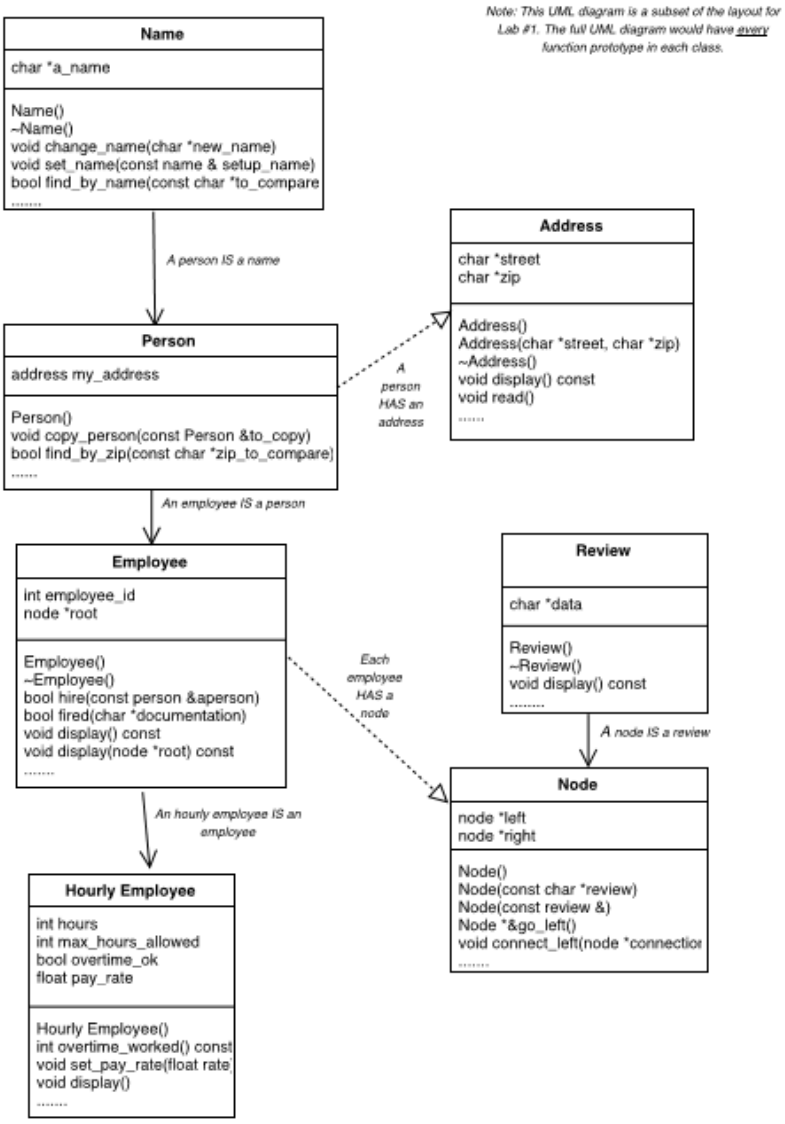
5. **THIS IS NOT** a meaningful UML diagram:



Turn the page for a better example!

Example UML Diagram

- Represent each of your classes with a shape
    - Write its name in that shape
    - Write the class' 'job' or what functions you think it will need to have
- In this example:
    - Inheritance (is a): the base class points to its derived class
    - Containing (has a): a class will point to the type of class that *it has*
- A solid line connecting two classes represents an "*is a*" relationship (inheritance)
    - An Employee *is a* Person
- A dashed line connecting two classes represents a "*has a*" relationship (containing)
    - An Employee *has a* work Schedule
- Whichever direction the line is "pointing" is ultimately your preference (but always be consistent and clear about it!)
- Then, refine your diagram for submission. We prefer the hierarchy being drawn from the top-down (like this one is!)



CS302 Assignment Requirements: Overview

1. **All five of the programming assignments** must include the following **write-ups (turned in at the same time as the program):**

Efficiency Writeup:

A minimum 400 word written document analyzing your design and discussing how effective the classes that you created were. Discuss the validity of your approach and analyze it in terms of object oriented programming. **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. Discuss the effectiveness of your design and classes
b. Discuss the validity of your approach
c. Analyze your results in terms of object oriented programming methodologies
d. What major changes did you have to make in your design and explain why
e. Describe the efficiency of the approach and the efficiency of the resulting code
f. **Think in terms of analyzing your solution!**

• For the **data structures** reflect on these questions:

a. How well did the data structure perform for the assigned application?
b. Would a different data structure work better? Which one and why...
c. What was efficient about your design and use of the data structure?
d. What was <u>not</u> efficient?
e. What would you do differently if you had more time?

• Consider how well your program meets the goals of being Object Oriented:

a. Were there classes that had clear responsibilities?
b. Did one class do the job that another should have?
c. Where did hierarchical relationships fit in and would it be effective in a larger application?
d. What was <u>not</u> Object Oriented?
e. Can you envision a design that would have been more Object Oriented

GDB Writeup:

A minimum 200 word written discussion of how debuggers (gdb, xxgdb, ddd, etc.) assisted in the development. This write up must describe experiences with the debugger, how it assisted code development, or how it could be used to enhance the programming experience.

a. Discuss the effectiveness of the debugger
b. Discuss what problems it helped you solve
c. Did you discover how it could be used to enhance the programming experience
d. Discuss features that you would like to learn about so that you could use them the next time you program
e. Discuss the validity of your approach

CS302 Assignment Requirements: Specifics

1. **Every** program will need the following **constructs**:

    a. Every program must have 5 or more classes
    b. Class designs must include single inheritance hierarchies
    c. Each class that manages dynamic memory must have (a) constructor, (b) destructor (C++), (c) copy constructors (C++).
    d. When designing member functions, think about where the member functions will get the data to work on. If they do not have arguments, then the only option is the user which may not be appropriate in all situations.
    e. Starting with Program #3, each class with dynamic memory must have an assignment operator
    f. Pass class objects as constant references whenever appropriate.
    g. Never pass a class object by value. Never return a class object by value.
    h. No use of structs, except for exception handling. (The more you use structs, the less OO your results will be!). Examine the lab .h files to learn how to use classes for node implementations.
    i. No classes can exist that have only "setter" and "getter" member functions, except for a "node" class.
    j. Minimize the use of "get" functions in general. If you implement a "get" function, first ask yourself why the function exists and if a helper function can be implemented instead to perform a specific task. Justify your use of getters in your design writeups.
    k. All data members must be private or protected (never public).
    l. All arrays must be dynamically allocated with new; no statically allocated arrays are allowed
    m. Global variables/objects are not allowed; global constants are fine
    n. **Do not use the String class! (use arrays of characters instead!).** You may use the cstring functions!

2. Use **modular design**, separating the .h files from the .cpp files.
    a. Remember, .h files should contain the class header and any necessary prototypes.
    b. Never implement member functions within a .h file.
    c. The .cpp files should contain all function definitions.
    d. You must have at least 1 .h file and 2 .cpp files for each programming assignment
    e. **Never "#include" .cpp files!**
    f. Keep the grading process in mind. **Do not** have over FIVE .cpp files); if you plan to have more files than this, please seek authorization from the instructor first.

3. **When developing software**, always think about how someone else could maintain the software afterwards. Ask yourself the following questions:
    a. Is the code clear and understandable?
    b. Are there flags being used that the reader of your code might not understand without comments?
    c. Are variable names self-documenting or do they need additional clarification?

4. Make sure there are **comments in each file**.
    a. There should be header comments in each file (each .cpp and .h file) which contain your name, the course you are taking, the date, and the purpose of the code in the file
    b. Each class interface should have comments about how the client programmer could use the public member function listed in the class.
    c. Each class implementation (where the member functions are physically implemented) should have a comment for each function about how they support the data structure and any algorithms used for those functions. If the function is modifying the data members, this should be clearly outlined in the header comments. If the function returns information back to the client program, then this too should be documented.

5. Please keep in mind that the **comments** in the **header files** (.h) and the **source code files** (.cpp) should not be the same.
    a. Comments in the header files should discuss **how to use** the functions declared. These comments tell the client programmer developing the application software how to use the functions declared in the .h file. This should include any assumptions that are required for the client programmer to know (such as setting counters passed in to zero or pre-loading data into arguments before calling a function). The comments in the .h file should also specify how to call the functions, including returned values that the client programmer might need to know about.
    b. Comments in the source code files (.cpp) should discuss how to maintain the functions that have been implemented. These comments should discuss the implementation of the functions and how they solve the assigned tasks with any algorithmic dependencies. The comments in a .cpp file should be aimed at programmers that will maintain the code in the future.
    c. **Every function** must have a **header comment** describing the purpose of the function and the purpose of the arguments!!

6. Make sure your code is **readable**. Here are some *suggestions* to follow:
    a. Use a consistent pattern of indentation.
    b. Line up the starting and ending curly bracket
    c. Keep curly brackets on their own line
    d. Use white space (blank lines) to separate program sections. At least three lines of white space is best to separate functions.
    e. Use mnemonic names for identifiers that relate to their purpose
    f. Use constructs that reduce side effects
    g. NEVER use global variables in these programs!
    h. Use the prefix versus postfix operator, except in situations where your code needs the pre-incremented value at the same time as incrementing the variable
    i. Write comments following each variable declaration telling what it will be used for; this should happen even if the variable names are self-documenting. For example, a variable named "count" obviously keeps track of a count but what is it counting? When is it incremented? These are the types of things the comments should describe.
    j. Provide comments to explain any program action whose purpose is not obvious to anyone who reads the code.

7. Use **structured programming** techniques; this is expected!
    a. Always have the conditional expression of the loop indicate why the loop continues and when the loop will end
    b. Avoid the use of exit, continue, or break to alter the flow of control of loops
    c. Avoid using while(1) type of loop control
    d. In fact, there should **never** be a return statement from within the body of a loop!

8. When implementing **abstract data types** stay within the "rules" of **data** abstraction as is outlined in CS163:
    a. Abstract Data Types should NEVER prompt the user or read information in. All information should be passed from the client program or application
    b. Abstract Data Types should NEVER display error messages but instead supply success/fail information back to the calling routine.
    c. Abstract Data Types should NEVER have public member functions with detailed information about the hidden data structure; for example, none of the public member functions should have "node" arguments.
    d. All data should be in the private or protected sections of the class.
    e. You may create your own version of the string class in CS302, but if this is done it must follow the rules of (1) data abstraction, (2) properly overload the appropriate operators, and (3) be your own code (100%)

Using Test Plans in Lab

As a computer scientist or computer engineer, your job is to evaluate the steps of the requirements and think about different cases that need to be tested. Then, consider how you want your code to work if these cases take place. Later, when the code is written, test plans can be used to "verify" if the case was tested and worked the way that was expected. Most of the exercises in this manual will have you create test plans for the code that you are developing. Please be aware, as a software developer it is vital to exercise very line of code and in all conditions, that might be expected.

Test plans should consider the (a) special cases and (b) necessary test conditions for each of the data structures.

Let's take an example of the following implementation:

"Implement a class to manage a list of movies, sorted by title"

Given:

Both head and tail pointers as data members

The special cases for insert are:

Case #1 The list is empty
Case #2 The movie title is smaller (alphabetically) than the all others in the list
Case #3 The movie title is larger (alphabetically) than all others in the list
Case #4 The movie title is the same as an existing movie title
Case #5 Otherwise, the movie title is unique and inserted in the middle of the list

We would need to do this for each of the operations: display, search, remove, etc.

For example, let's make a test plan for the above functionality. In many of the labs, you would encounter test plan sheets that look like the below table:

Test case	Expected Result	Verified (yes/no)

Testing for Errors

When implementing software, the last thing you want to do is provide software that doesn't work in all situations. Most of us have used software that hangs or behaves unusually when unexpected data is received. Creating test plans encourages us to spend the time to think about the different scenarios that might take place and how our software will behave.

For this example, depending on what order the client program calls the member functions, display might be called prior to any data be added. Never make assumptions on what the user or in this case application programmer will do!

> **Test Case:** Insert an item with a matching title
> **Expected Result:** Return an error flag to the client program that duplicate movies are not supported in this list.

How would the test case be described using the similar format above? What would the expected result be? Here is an example

Test cases for Insert	Expected Result	Verified (yes/no)
1. Empty List	Add the movie as the first node with head and tail being affected.	
2. Inserting when the data is less, alphabetically than the first item	Add the movie as the first node; this new node should point to the original first node (where head was pointing). Head should be altered to point. *Double check that there is no memory leak!*	
3. Inserting duplicate data	Return a failure flag and no insert is performed. The list should not change	
4. Adding at the end where the data being added is greater, alphabetically, than tail's data	The new movie should be added at the end and tail should be updated to this new last node. The previous last node should now point to this new node. The new node's next pointer should be set to nullptr.	
5. Inserting elsewhere	Head and tail are not changed. Make sure the data is alphabetical	

(*) For some applications, we would only want to give the user a set number of chances to enter data that is invalid

Algorithms: Thinking in Parallel

Most computers have more than one processor. A computer with more than one processor can process more than one operation at the same time, and doing more than one operation at a time results in programs running faster. Unfortunately, software written without thinking in parallel will not run faster even if you have multiple core! What this means is that a program written for a computer having only one processor may not run faster by adding processors. The program must be written to use multiple processors. If we write a program without thinking about parallel processing, only a single core will be used. This means that running programs on a computer with more than one processor may not provide any speedup if they are not written to take advantage of these processors. In the earliest computers, only one task could be processed at a time. Astonishing!

In the real world, problems can be too large to run in any reasonable amount of time if everything was done serially – one by one. So, we need to break down our work into **threads**. In fact, the world today is really all about parallel processing. You've probably heard the words "multi-core", "distributed computing", or "parallel computing". The world's largest computers have 20-30,000 cores! But to get the performance out of this, we would need to be able to divide up our work.

The goal is to perform more work simultaneously by dividing the problem up. The idea is to break down the problem into components that can be done at the same time without affecting each other. Maybe we can perform two tasks at the same time or even more. In CS162, we want to begin by learning how to recognize such situations.

For example, if you wanted to count the number of people riding the MAX, it might take you the entire trip to campus. But, if your job was to count only the people in a particular section or car and others were assigned to their sections, we would quickly have our answer. This is the **speedup** that is referred to with parallel processing. In a problem where the data can be processed in parallel (**data parallel**), we can divide up the data into the number of reasonable tasks or core (the "processors" counting); each core will do the same task but on a different section of the data. Of course, if we went too far and divided up the work such that there would be two people counting per seat, we would have more overhead in managing the parallelism (**parallel overhead**) and the speedup achieved is limited (**Amdahl's** law). The speedup of using parallel processing is limited because the amount of parallelism is limited by the number of tasks

There is also the notion of **task parallel**, or **concurrency**. Concurrency happens if there are any steps that can be done independent of each other and therefore concurrently. For example, when we cook spaghetti we can boil the noodles at the same time as we make the sauce. They do not affect each other until the end when we must put them together (by **synchronizing**).

In reality, most problems will have some parts that can be done at the same time and others that cannot. **Pipelining (Producer-Consumer)** is the concept where we move from one task that use its result for the next set of tasks that may need to wait until the first task is completed. For example, if the spaghetti sauce is finished before the noodles are done, we wouldn't want to dish up the sauce onto the plates yet! We would want to wait until the noodles are done before continuing on. In this case, we experience some **data dependency,** causing a **data race**. Both tasks will use the same plate and the noodles must be placed on that plate prior to the sauce. One thread will need to be locked until the other finishes its task to put the pasta on the plate before continuing with plating the sauce!

When we thinking about designing an algorithm, we need to consider the parallel design:

1. How many tasks can be done in parallel (partitioning them)
2. Look for the patterns
3. Group these together
4. Determine if there are tasks that depend on others (pipelining, data dependency)

In summary, you now have been introduced to:

1. Threads
2. Concurrency
3. Speedup
4. Synchronization
5. Amdahl's Law
6. Parallel Overhead
7. Pipelining (Producer-Consumer)
8. Data Parallel
9. Task Parallel
10. Data Race