CS202
LAB MANUAL

Programming Systems

Karla Fant

Spring 2020

Student Name:____

CS202

Programming Systems

Karla S Fant

Lab Manual

Spring 2020

Student Name:_______________________________________

PSU ID #: ____

CS Login Name: _____________________

Place Barcode Sticker Here

CS202
Programming Systems

Contents

FIRST READ CS202 EXPECTATIONS SECTION

I, ________________________________ have read and understand the CS202 Expectations
(printed full name) in the CS202 Lab Manual for **Spring 2020**.

I understand each of the following requirements for each lab session:

1. I agree to be sensitive to the students around me in lab. I should limit the volume of discussion and keep it related to the course material.
2. I understand that I must be actively working the entire time for attendance to count.
3. I will do my best to use the lab time to further my knowledge of the material, participating in the group activities, designing test plans, and programming in the lab manual.
4. The group activities are the first activity we start with each class period and I understand that it is graded as part of the lab manual requirements; they are not optional!
5. I will attend the lab section that I am enrolled in. I will seek permission if I want to attend a different section. I can only attend another section if there are seats available.
6. I will attempt to arrive within the first 5 minutes and stay for the entire session.
7. I understand that my PSU ID is used to scan-in and out of labs each time.
8. I will always come with **both of my lab manuals: the Linux & Vim manual and the CS202 Lab Manual**. I know that I can't attend without the manuals; there will be exercises from both manuals in the labs. The CS202 Lab manual must be specifically for **Spring 2020.**
9. I understand that starting with Lab #2, prelab exercises are required to attend the lab.
10. If I haven't done the linux & vim exercises from CS162 and CS163 at PSU, then I will start with the level #1 and level #2 exercises and complete them before performing the level #3 exercises. I should complete all level #1-3 exercises this term!
11. I also understand that if I miss a lab session, I need to make it up within a two week period. I must make an appointment to attend the official makeup lab. I can't make up labs whenever I want. If I end up needing to make up more than two labs, I will need to contact my Instructor to develop a plan on how to proceed.
12. I agree to let the technical assistants make notations in the lab manual and corrections to the prelab exercises for each lab that I attend.
13. I agree to ask for assistance if I don't understand a concept. I am aware that I can meet with my Professor or Instructor to receive further assistance.
14. I am aware that food and drink are not allowed in the lab classroom, except in designated areas as controlled by the lab facilitator. Some lab facilitators may allow sealed containers.
15. **I won't use my cell phone during the lab or my computer for non-lab related work. I will not play games or work on non-lab related work. I will excuse myself if I need to handle a personal matter during lab.**

I certify that I have read and understand the lab requirements as discussed above.

Signature**:** __________________________ TCSS Initials________

Lab Manual Grading Information

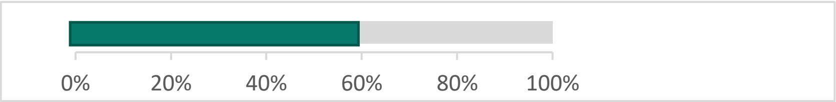
Lab manuals are graded twice a term, once at the midterm proficiency demonstration and then again at the final proficiency demonstration. To prepare for the grading process, a self-evaluation form is completed by each student and turned in with the manuals for both evaluations.

The self-evaluation form indicates:

1. Which labs were completed and/or skipped
    a. Remember one lab may be missed without making it up.
    b. This information needs to be indicated on the self-evaluation form.

2. How much progress was made along the proficiency scale
    a. For each lab, the proficiency scale reached will be filled out by each student on the self-evaluation form:

Lab #1:



Then, when we grade the manuals, we will use this information in addition to the following:

1. How many questions were answered versus left blank
    a. How completely were the pre-lab questions answered?
    b. Were the group activities worked on?

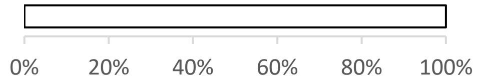
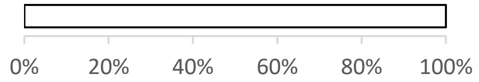
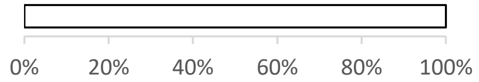
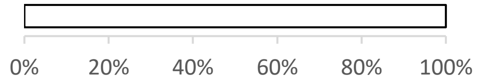
2. The degree to which the coding exercises were completed.
    a. How many of the coding questions were completed?
    b. Were the short answer/fill in the blank questions also completed that were part of the coding exercises?

3. The legibility of the written work
    a. Are the answers sensible?
    b. Are the answers applicable and relevant; do they apply to the question(s) being asked?
    c. Were the answers readable?

CS202 Lab Manual Self Evaluation Form - Midterm

Name:__________________ PSU ID:_________

For each lab, fill out the proficiency scale reached based on the completion of the lab manual materials. Bring your manual to the proficiency demos; this page will be removed at that time. Make sure your name is on this page!!

Attended	**Self-Evaluation: Coding Proficiency Scale Met:**	**For Official use Only**
□ Lab 1	Pre-Lab Finished □ Group Activity Finished □ 	_____ % Completed (not blank) (readable and relevant!)
□ Lab 2	Pre-Lab Finished □ Group Activity Finished □ 	_____ % Completed (not blank)
□ Lab 3	Pre-Lab Finished □ Group Activity Finished □ 	_____ % Completed (not blank)
□ Lab 4	Pre-Lab Finished □ Group Activity Finished □ 	_____ % Completed (not blank)

□ Linux/Vim Manual	# Labs Completed _____________ Level # ______ (1. 2. 3)	_____ # Labs Completed _____ Level #

Calculate Grade	Sum total of all percentages completed__________ Total number of labs completed_________ (3 or 4) Divide the total by # of labs___________% Multiply by 5__________ (0-5 grade)	_____ Total Percentage _____ # Labs Completed _____ Calculated grade

CS202
Programming Systems

Expectations and Overview

IMPORTANT

Everyone will need an MCECS (CS) account.

This account is needed for
both lab work and programming assignment work.

Visit FAB 88 to obtain your account for the first time.

This needs to take place PRIOR to the first lab!

CS202 Lab Session Policies - Expectations

There are two LAB manuals that must be brought to each lab to attend:

- **CS202 Lab Manual for Spring 2020**
- **Linux & Vim Exercises Manual for 2019-2020 or 2020**

- **LAB manuals will be turned in twice during the term for grading**

CS202 Labs are designed to have students gain hands on experience with the lecture topics in a smaller environment where there are lab assistants available to assist with syntax, design, and testing questions. The concepts learned can then be applied to the individual programming assignments in a larger context. These represent a vital component to becoming proficient with the course materials.

Lab sessions will include hands-on computer work, linux exercises, and group activities. All portions of the lab are expected to be completed during the hour and 50-minute lab sessions. If a lab is not completed, students should consider attending a makeup session to become proficient with the materials.

Attendance:

1. Attendance is required. **Lab attendance is required to pass CS202**, with pre-labs completed to attend. Lab participation is graded Pass/No Pass.
2. For attendance to count, students must arrive within the first 10 minutes of the lab and stay for the entire lab period. Do **not** expect to leave early!
3. It is also **expected** that students will attend the lab that they are enrolled in.
4. **Upon arrival**, the lab assistants will scan each student's ID or supplied barcode.
5. **Do not** expect to leave early!
6. If more than one lab is missed, the lab must be made up by reserving a seat in a makeup lab. Approval by the instructor is required to attend a lab other than the published makeup lab for the term.

Preparation:

1. It is **expected** that students will come into these labs having read the assigned readings and are up to date on the latest lectures.
2. It is also **expected** that students will come into these labs having read the **background information** for the lab that is being performed.
3. Starting with the second lab, **pre-lab worksheets** must be completed and brought to the lab. The background information should also be reviewed first!
4. **Upon completion of the lab**, the ID or supplied barcode will again be scanned
5. If the lab work is completed prior to the end of the period, the lab assistants will assign additional practice questions or facilitate group activities.

CS202 Lab Etiquette

1. **Use notes and books** and be prepared to discuss lab code and concepts with other students (except for testing or proficiency demonstrations).

2. **Be prepared** to write your own code! It won’t help to copy the code!
   a. The labs will get students ready for assignments, quizzes and exams.
   b. They are a great place to practice for the proficiency demos.

3. **Be courteous to other students around.**
   a. Talking should be limited to the concepts covered in the lab materials and course work kept to a minimum to avoid distractions.
   b. Be aware of your surroundings and keep the volume of discussion within reason

4. No food or drink while labs are in session (leave drinks at the designated location)

5. Power cords may not be placed across walk ways
   a. Be aware of anything that could be a trip hazard (including power cords or back packs)

6. *****NOT ALLOWED DURING LAB SESSIONS *****
   a. No use of the internet for web surfing, social media, or email during lab time, with the exception of D2L, karlaf’s website, and the use of putty, ssh, or terminal to work remotely with the CS systems.
   b. No cell phone use (please excuse yourself for that time)
   c. The lab sessions are not for working directly on homework or programming assignments.

Making the Most of your Time!

The best way to become proficient with the course content is to ***program, daily!***

The Labs are a valuable time where you can experiment with the code learned in lecture and through the readings and ask questions from the lab assistants. Your goal in working through this material should be to prepare yourself to work with the concepts and syntax outside of lab on your individual programming assignment.

If you find that you cannot make enough progress with the labs coding (and stay within the introductory portion of the lab manuals), you will want to be proactive as soon as possible. Do not wait until the end of the term! Instead, contact your instructor to find out the best way to learn this material, make sure to perform the reading assignments in the textbook prior to lab, work through some of the self-check exercises in the textbook to get a grasp of the concepts, bring your book to lab, and attend makeup sessions as much as possible.

Practice Exercises and Self Check Quizzes:

To prepare for written exams (quizzes, the midterm exam and the final exam) and programming proficiency demos, make sure to spend time with the non-graded portion of the lab manual. Each lab has practice questions to assist with preparing you.

Take these seriously and try to problem solve rather than memorize solutions. After implementing solutions, use Valgrind to look for memory leaks, draw pointer diagrams, check your function calls – do you properly use the returned value coming back from a function. Are you properly using recursion? Is there code existing after a return statement? Have you properly returned the correct information? Could a wrapper function have assisted to perform non-repetitive tasks? Do you understand the data structure algorithms? If not, go back and review the CS163 course materials.

Make use of the practice questions and self check quizzes in this manual.

1. You can work on these individually on your own, in a group, with the lab assistants or with the tutors in Fab 88.
2. These augment the self-check exercises that are supplied in the textbook at the back of the chapters assigned for reading.
3. Also, keep working on your data structures, as you will be tested on data structure proficiency during and at the end of the term. Work on traversal, insertion, and removal algorithms for linear, circular and doubly linked lists. Also, focus on arrays of linked lists and binary search trees. **Don't wait!** Everyone needs to be proficient with recursion and linked data structures by the midterm.
4. We suggest starting with iterative solutions and then re-implementing all repetitive algorithms recursively for practice.

CS202 Overview of Lab Content

Each lab contains the following information and should be processed in the order listed:

1st: Background Information:

READ THE BACKGROUND INFORMATION EACH WEEK BEFORE ATTENDING LAB!!!

a. **Background information** is supplied to assist with the completion of the labs.
b. The background information is intended to augment the reading materials assigned for the week and lecture materials.
c. The background information should be read prior to completing each week's prelab.

2nd: Prelab Worksheets:

a. **Prelab worksheets** are placed right before the lab material for a session, starting with Lab #2.
b. The prelab worksheets must be completed prior to attending the associated lab.
c. If you are unclear about an answer, make sure to ask for assistance in lab and have the material explained.

3rd: Group Activities:

a. Most labs will begin with students working in **groups** to ask and answer questions about the Pre-labs. This should be the **first** thing that student perform when starting a lab.
b. Participation in the **group activities** is **expected** as part of the lab participation grade. No more than 10-15 minutes should be spent on these activities.
c. Working in groups provides the benefits of collaborative learning where you can share knowledge and learn from one another. It is particularly important in CS202 for the more complex concepts. Please don't skip the group activities!!

4th: Linux and Vim Exercises

a. These are completed from a separate manual. These exercises are worked-on during the first 10 minutes of each lab.
b. Remember to bring both manuals!
c. Start these <u>after</u> completing the group exercises. These begin the individual portion of our work in lab

5th: Coding Exercises:

a. All labs will have you work with cs202lab.cs.pdx.edu using your MCECS account (user name and password).

b. For each lab, the coding exercises are divided into three levels. Level #1 problems will work with the fundamental concepts. Then, level #2 progresses to intermediate exercises. Lastly, level #3 introduces students to more advanced work. All three levels should be experienced and mastered to be considered proficient at the concept.

c. You will be given a specific directory to work in
(e.g., CS202/Lab1).

d. **At the end of the lab session, perform ./submit even if all of the work is not completed.**

e. If the programming components are not completed, attending makeup labs and recitation sessions are high encouraged.

f. Coding exercises prepare students to apply concepts to individual programming activities and ultimately to proficiency examinations. At critical steps in the lab manual, a "Proficiency Scale" shows the relationship between completing the step and reaching proficiency with the concept.

g. The grade received on the lab manual at midterm and final time will be depending on how far you have gotten through the coding exercises and answering the related questions.
   - At the midterm period, we grade labs 1-4
   - At the final, we grade labs 5-10
   - One lab may be missed and won't affect your grade

Afterwards, Quiz yourself:

1. Use the self-check quiz questions at the end of the lab materials for each section to test if you fully understand the material.
2. The best way to treat this material is to answer the self-check quiz questions using a closed book/closed notes technique.
3. If you are unable to answer the questions, make sure to spend time with the lab assistants, recitation sessions or the tutors in FAB 88.

CS202 Lab Policies – Makeup Labs

1. Makeup labs should be scheduled by appointment. Use this link to book a reservation in a makeup lab: https://cs202_makeup_lab.youcanbook.me/

2. Do not expect to attend a makeup lab without a reservation!

3. If you want to attend a lab OTHER than the official makeup lab (one that you are not registered for), then seek authorization from your instructor.

4. Labs should be made-up within 2 weeks of having missed a lab.

5. **Inform the Lab Assistants** which lab number is being made up.

6. Only one lab may be made up in each lab session. A lab session is an hour and 50 minutes in length.

7. Show the pre-lab for the lab that is being made up (e.g., to make up Lab #5, come prepared with the pre-lab for Lab #5 already completed).

8. It is also **expected** that students will come into these labs having read the **background information** for the lab that is being performed.

9. **Upon arrival**, the lab assistants will scan each student's ID or supplied barcode; this will not take place if you are attending a lab outside of the official makeup lab schedule. In those special situations, you will need to talk with the lab facilitator and show your picture id.

10. If the lab work is completed prior to the end of the period, the lab assistants will assign additional practice questions or facilitate group activities.

11. Do **not** expect to leave makeup labs early!

CS202 Lab Session Policies: Pre-labs

1. Most labs will have a **pre-lab** that **must be filled out** prior to attending the lab. These may be found in this manual associated with the lab. The purpose of the pre-labs is to prepare you for the lab itself.
   - **Completing the prelab worksheets each week is required to pass.**
   - Pre-lab worksheets may not be filled out during the lab time.
   - The intent of the worksheet is to solidify the concepts learned from the readings and lecture and prepare us for performing hands-on work in the labs.
   - The pre-lab worksheet is a **fantastic study tool** for the midterm and final exams.
   - Pre-labs are required to attend any makeup labs.
   - *The Course Outline on D2L indicates when pre-lab worksheets are required.*

2. The pre-lab worksheets are part of this lab packet.
   - The lab assistants will come around during lab and check-off the pre-lab.
   - Expect the lab assistants to initial the pre-lab as it is checked.
   - The lab assistants will also make notations in this lab manual with corrections to the pre-lab answers.
   - Please ask the lab assistant to explain solutions to the pre-lab if the answer is unclear.

CS202 Lab Policies – Group Activities

3. Most labs will begin with students working in groups to ask and answer questions about the pre-labs.
   - In this case, spend no more than 10-15 minutes on group activities and then move ahead to the coding exercises.
   - Lab assistants will be available to answer questions and discuss solutions.
   - These discussions are a valuable part of the learning environment.
   - Participation in the group activities is expected as part of the lab participation grade.
   - Working in a group provides the benefits of collaborative learning where you can share knowledge and learn from one another. Working on these individually will not provide the same benefits.

CS202 Lab Session Policies – Computer Systems

Computers:

1. For labs that take place in classrooms, students must come with a laptop, netbook, or tablet that is fully charged. **Power may not be available during the period.**
2. Most CS202 labs are in classrooms without computers or many power outlets.
3. Students may check out a school computer if one is available
    a. To check out a computer, show PSU ID or other photo ID
4. Labs scheduled for **FAB 88-09** and **FAB 88-10** are **computer labs** and do not require extra reservations.
    a. When a lab is scheduled for one of these rooms, you may elect to use the computer at the desk you at or use your own laptop.

Operating System and Editors:

1. **Required.** All work during the lab will be on **linux**, through ssh, putty, or terminal. **The only editors** that may be used are **vi, vim,** or **emacs.** Editors such as pico or nano are not allowed. ***No IDE's may be used.***
2. All lab work takes place in the CS linux environment.
3. We use a system called "cs202lab system" for working on the lab exercises.
    a. Lab work is to be done on **cs202lab.cs.pdx.edu**; login using your MCECS user name and password.
    b. This account holds all of the supporting code for the labs needed to compile and run your function; this code is not designed for other systems. This environment is also available outside of lab time for practice. Practicing is highly advised. This environment is just for labs and practice.
    c. The cs202lab system is available ONLY while enrolled in CS202.
    d. The cs202 lab account and all of the files are deleted at the end of the term.
    e. We **do not** advise working on your programming assignments on the cs202lab system or keeping material on the cs202lab system that you want to access after the term is over!
4. **Graded Programming assignments** should be created using **linux.cs.pdx.edu**.
    a. Students have access to this system between terms and files remain persistent from term to term.
5. If you have used Visual Studio, Dev C++, or other IDEs, please suspend their use for the term. Using such tools will not assist in developing programming proficiencies.
6. For assistance getting an account or signing in, ask the lab assistants or the CS tutors in FAB 88.

CS202
Programming Systems

General Background Information

Background for CS202 Programming Assignments

Vision for all CS202 Assignments:

In CS202, all five of our programs will focus on developing object oriented solutions to applications. 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. 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 Object Oriented 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 Object Oriented 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 CS202 – 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 OOP.

Structs are not OO. Public data is not OO. ***Neither should be done!***

CS202 Assignment Requirements: Overview

1. **There are five programming assignments** required in CS202 as outlined by the syllabus.

2. **Three of the programming assignments** will have **design writeups** and **UML diagrams.** These are turned in separately from the programming assignments themselves.

3. **The design writeups must consist of a minimum of 600 words**
    a. The write-up should cover the major design considerations: what classes you are using, how they are related to other classes (using, containing, hierarchical); it should discuss how the methods provided minimize the use of getters. It should be clear in the design what functions will be needed in each class and how they will be used by the other classes in your design.
    b. All designs must be your own work and may not be copied from the web or other students. Be careful to not plagiarize. Doing so will result in a zero on an assignment and a failure in the class.

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

5. Every writeup submitted in CS202 must be typed and spell-checked. The diagrams may be hand-drawn (and scanned) or electronically generated and submitted as a pdf.

6. **All writeups and diagrams should be submitted as separate documents (do archive using zip or tar).**

CS202 Programming Assignment Requirements

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. (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 total files (including a combination of .h and .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 CS202, 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%)

OOP Design Write-up Expectations

1. The purpose of the OOP design writeups are to begin planning before physically programming. In CS202, we want to shift our focus to creating solutions that are comprised of multiple classes to solve the assigned problem. It is also design how best to integrate the advanced data structures that are included with each assignment
2. The written OOP design write-ups are turned in **prior** to the programs, when assigned. Included with each design write-up must be a UML diagram to provide an overview of the design.
3. The design write-up must be a minimum of 600 words.
4. These must be submitted to D2L.
5. The design write-up should be in a file named: design#.txt. or design#.doc
6. The design writeup and UML diagram **should not** be archived (zip or tar). This means you will upload two files to D2L.
7. It should describe the major design considerations for your program, the relationships developed between the classes and all data structures and algorithms that you intend to use. *You must have your name in this file.*
8. The OOP design write-up must be written in English using complete sentences. It should describe:
    a. It should cover the major design considerations
    b. Discuss what classes you are intending to create
    c. Discuss the relationship between those classes (using, containing, hierarchical)
    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. UML diagrams are also required

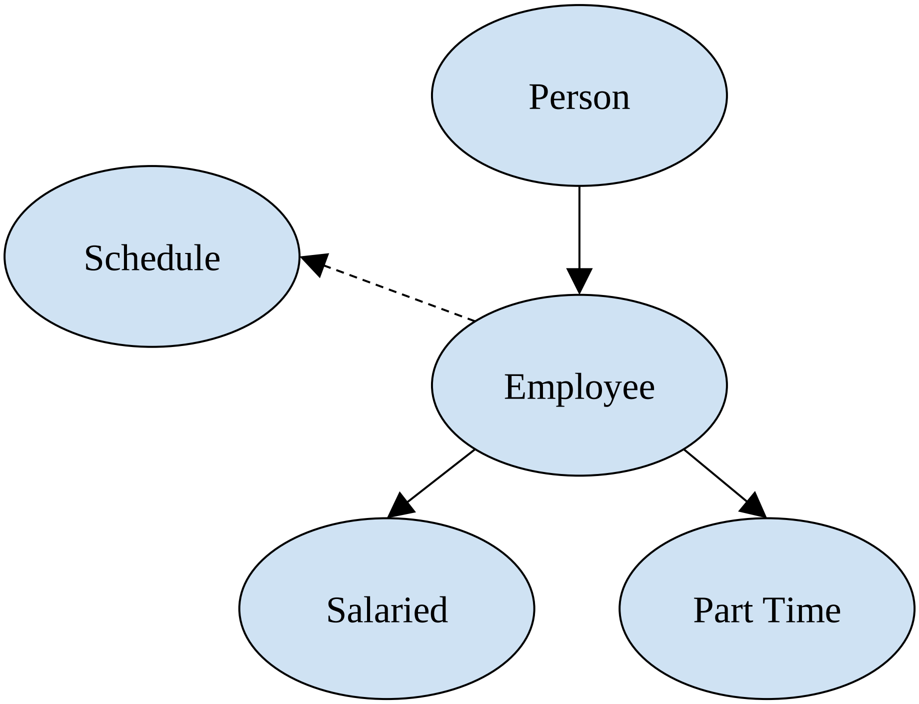
9. Each major design issue should be described. In the design considerations, discuss what the main design considerations are, why they are the main design considerations, how you solved them, and why you solved the way you did. **Think in terms of analyzing your solution!**

10. Please note that your design write-up should not look like C++ code; words such as "cin", "cout", "++", "--" should not be part of your design document. Instead, describe the major tasks that your program accomplishes. Any design document submitted which looks identical to C++ code will not be accepted.

11. By thinking through the design and creating a detailed design writeup, it should ease the implementation phase so that you can then focus on the syntactic details rather than the design. This takes time and practice!

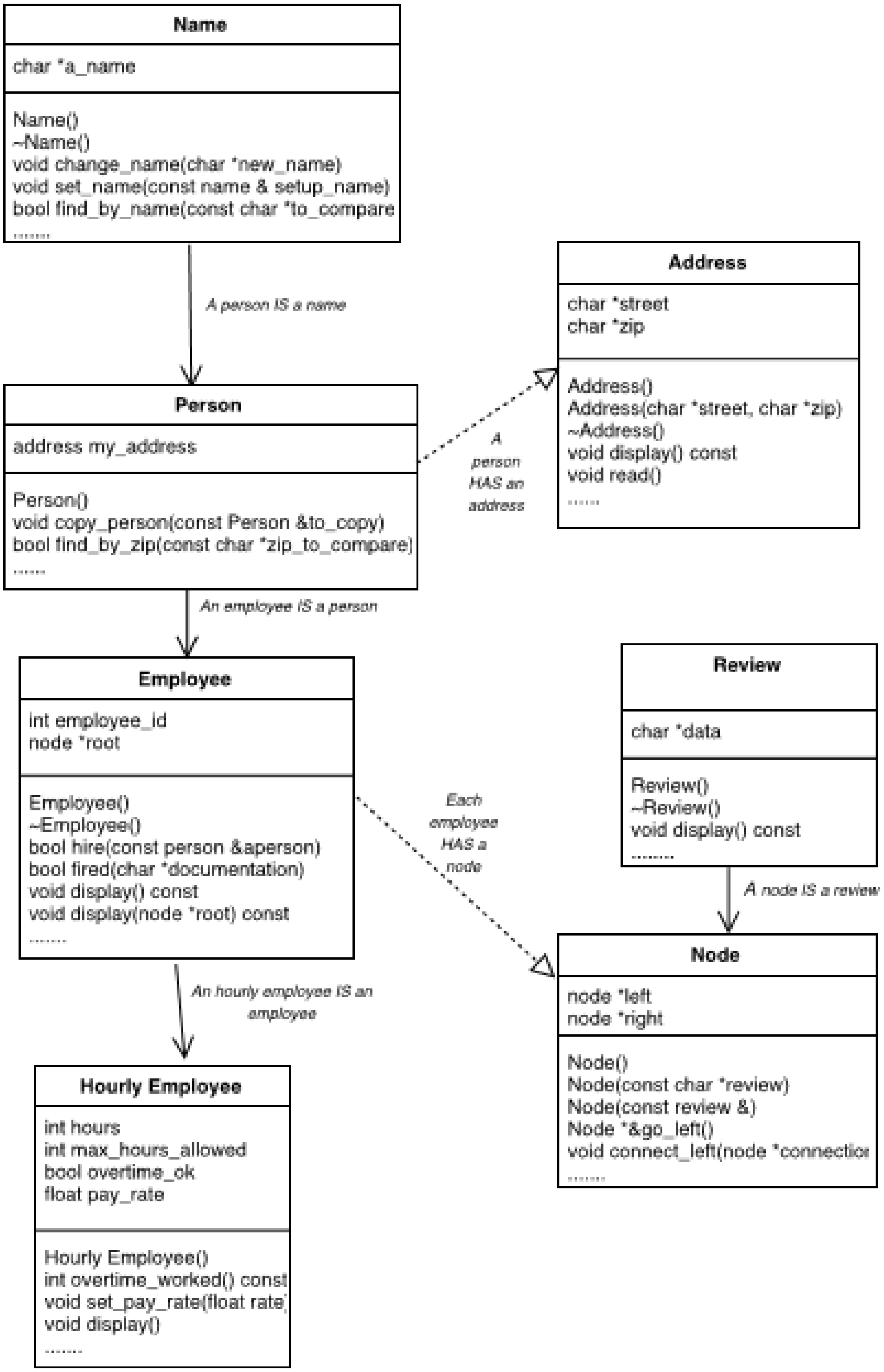
Creating a UML Diagram

1. UML diagrams will also be assigned as part of the programming assignment process and are turned in with the OOP design writeup.
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. **IT IS NOT** what you will be turning in!



- 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!)



Analyzing your Design: Postmortem

1. **Once an OO solution and data structure(s) been fully** implemented, we want to examine the overall efficiency and effectiveness of our design and implementation. This is done with a writeup that is submitted with your programming assignment.
2. **With each program (all 5 programs),** submit a file (e.g., review1.txt) that reviews the design after creating your program, your OO design and all data structures.
3. **Upload to** D2L this file as a separate document. **DO NOT include this file in your tar archive of your source code (.cpp and .h)**
4. The efficiency write-up should be **at least 400 words**.
5. It should be written in **paragraph** form.
6. The efficiency write-up must be a plain text document, written in English. *You must have your name in this file.*
7. General topics to cover in this writeup:
    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!**

8. 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 not efficient?
    e. What would you do differently if you had more time?

9. Answer the following questions concerning your OO design:
    a. Were there classes that had clear responsibilities?
    b. Did one class do the job that another should have? (e.g., is a class using a "getter" to compare the data when that should have been done with the underlying class?)
    c. Where did hierarchical relationships fit in and would it be effective in a larger application?
    d. What was not Object Oriented?
    e. Can you envision a design that would have been more Object Oriented (if you had more time)?

Using Test Plans in this Manual

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 NULL.	
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

CS202
Programming Systems

Data Structures Review

Review of Linear Linked Lists

Linear linked lists are used to be hold a collection of the same type of data. But, unlike an array, a linear linked list can grow and shrink as memory is needed. There is no need to predict before we have the data how much memory is needed. This is vastly different than how we deal with arrays. The side effect with a linear linked list is that we sacrifice the ability to directly access an individual element. Instead, one data item is connected to the next using a concept called a "node". A node hold the data and a pointer to another node, linking the data together in a linear fashion. Direct access is not achievable because there is no guarantee that the memory is contiguous.

- A node is used to hold the data and a pointer to another node called **link or next**.
- We need a head pointer to point to the first node in the list. The data type is: **node * head;**
- The last node in a linear linked list has a NULL next pointer
- The data in a node could be an integer, a dynamically allocated array (e.g., char *) or even a structure or class instance!
- We know a list is empty when **head** is **NULL**
- To access the list we must **traverse** from one node to the next
- The process of traversal requires that we access the next pointer to find out where in memory the next node exists. A common traversal would be:

  current = current->next

- The -> is called the **indirect member access operator**
  - We use the -> to access any member through **a pointer** to a class or struct object

- The . is called the direct member access operator
  - We use the . to access any member through an object of a class or struct
- We use the -> to access to access both the next pointer as well as the data, since both are members!
  - (*current).member is the same as **current->member**
  - This is the case regardless of the data type of the member!

- To traverse we must say: **current = current->next;**
  - This dereferences current
  - Accesses the "next" member of the node being pointed to
  - Stores that address (the content of "next") in current
  - This means that current is pointing to the same place that the node's next pointer was referencing!
- To access the data, say:
  - To output the data where head points: cout << head->data;
  - To output the data where current points: cout << current->data;

Working with Pointers

Remember that pointer variables are used to store the memory address of another variable. Just like any variable, you must first define a pointer variable, initialize it, and ultimately set it to point to an area of memory.

Operator Precedence

- When dealing with structure and class members and pointers to structures, you have a choice on how to reference individual members.

- One way is to use the following form.: (*head).price
  The parenthesis are necessary because of precedence. The direct member access operator (i.e., **.**) has a higher precedence than dereference operator (i.e., ***)**. So, if we had said *head.price, it would have meant *(head.price) which would be illegal unless "head" was an object of the struct and "price" was a pointer!

- Because pointers are so frequently used with structures, we can use the **->** operator to refer to a member of a structure. So, for example **head->price** is the same thing as saying **(*head).price**.

- The structure operators **.** and **->** have the highest precedence and bind very tightly. This means that saying **++head->barcode** increments the barcode NOT the pointer!! The meaning is: **++(one->barcode)**. If you didn't want this to happen...then you need to explicitly use parenthesis.

- Also, ***one->next** tries to fetch what next points to.

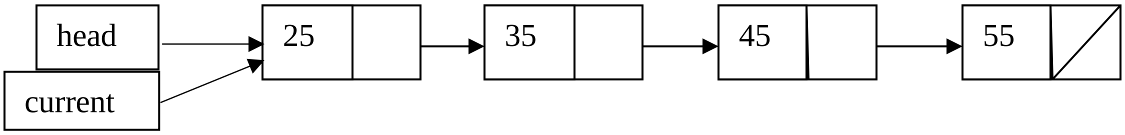
- We use the -> to access to access both the next pointer as well as the data, since both are members!
  - (*current).next is the same as **current->next**
  - (*current).data is the same as **current->data**
  - *current.member is NOT the same as (*current).member because of "precedence and associativity"
  - Remember – a member can be data or a pointer!

Tracing a LLL with Pointer Diagrams

The key to successfully understanding how to work with linear linked lists is in becoming proficient tracing code using pointer diagrams. This page demonstrates the step by step process for stepping through code with a pointer diagram. Make sure to work line by line and do not skip any steps. We will begin with line #7.

```
1)  #include "cs202_list.h"
2)  //Display just the last item in a linear linked list and return the data in the last node
3)  //If there aren't any node, just return zero
4)
5)  int list::display_last()
6)  {
7)          node * current = head;
8)          //Start by checking for special cases
9)          if (!head)
10)           return 0;
11)
12)         while (current->next != NULL)      //Traverse to the LAST node
13)                 current = current->next;          //Traversal
14)         cout << current->data <<endl;      //Output the last item
15)
16)    return current->data;
17) }
```

Step 1   Line #7: node * current = head;

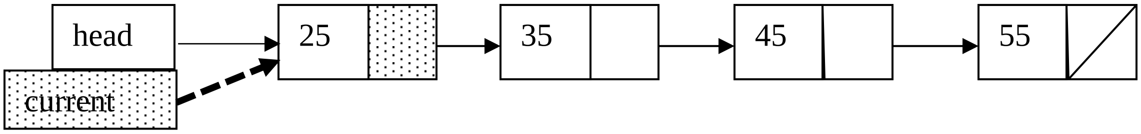


Step 2   Line #9: if (!head)

This is the same as: if(head == NULL) *or* if (NULL == head)
If head was NULL, then head would be zero and !zero would result in true!

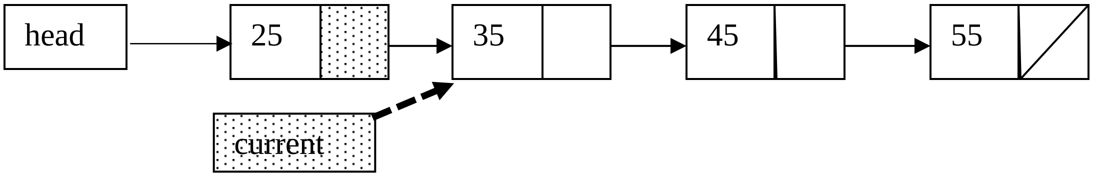
For this example, head is not NULL, so the result is false and we will continue to line 12

Step 3   Line #12: while (current->next != NULL)



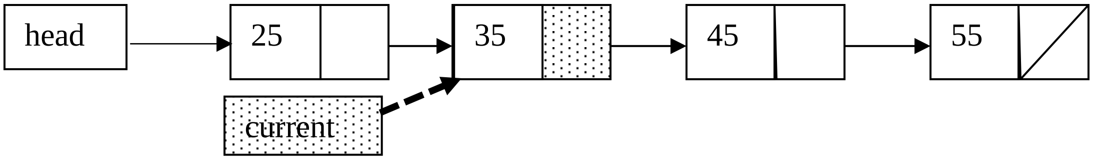
Checks if the first node's next pointer is not NULL. True, it is not NULL, so execute the body of the loop with line #13

Step 4 Line #13: current = current->next;



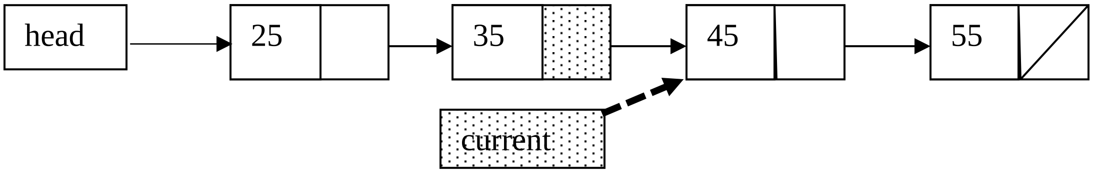
Takes the value of current->next, which was a pointer to the second node, and stores that address back into current. This acts as a traversal to the second node. Notice head is locked into the first node; our list stays intact without a memory leak!

Step 5 Back to Line #12: while (current->next != NULL)



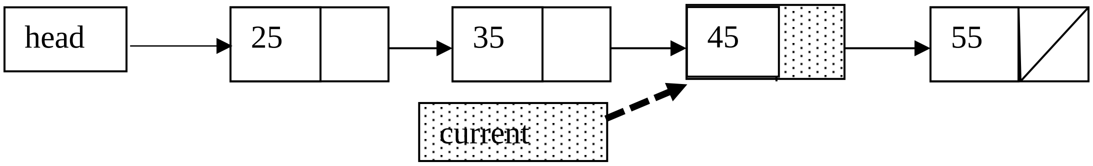
Checks if the second node's next pointer is not NULL. True, it is not NULL, so execute the body of the loop with line #13

Step 6 Back to Line #13: current = current->next;



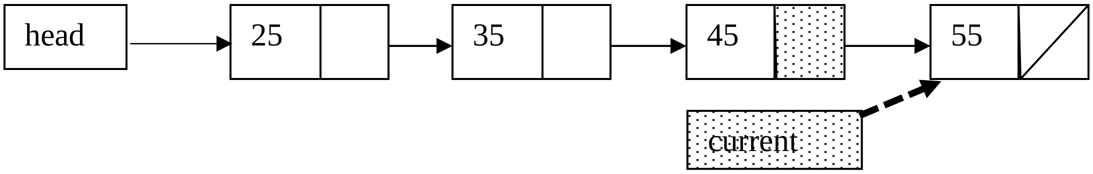
Takes the value of current->next, which was a pointer to the third node, and stores that address back into current. This acts as a traversal to the third node.

Step 7 Back to Line #12: while (current->next != NULL)



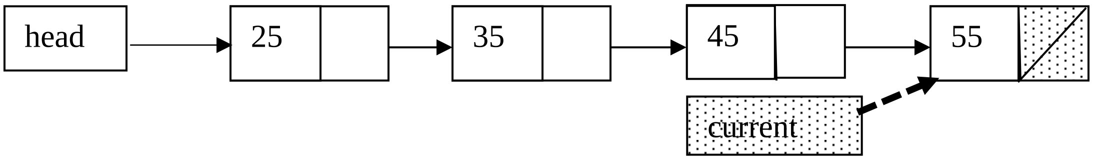
Checks if the third node's next pointer is not NULL. True, it is not NULL, so execute the body of the loop with line #13

Step 8 Back to Line #13: current = current->next;



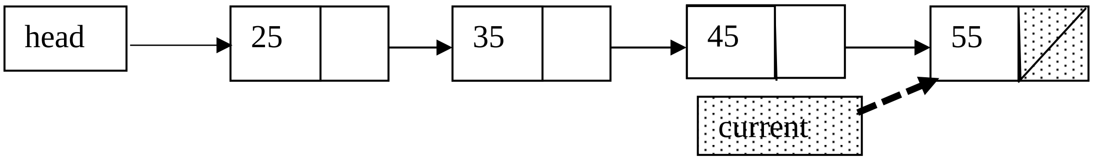
Takes the value of current->next, which was a pointer to the fourth node, and stores that address back into current. This acts as a traversal to the last node. Notice, current is now pointing to the last node!

Step 9 Back to Line #12: while (current->next != NULL)



Checks if the last node's next pointer is not NULL. False, it IS NULL. The loop is finished.

Step 10 Line #14: cout << current->data <<endl;

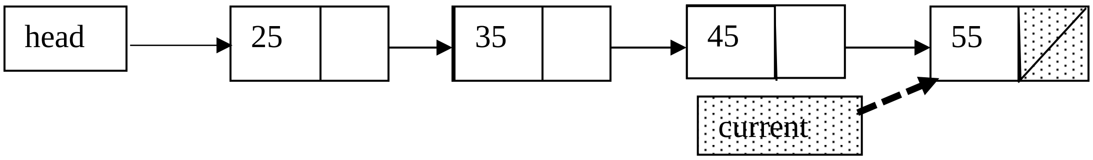


Displays the value 55.

- Remember whenever we use the indirect member access operator (->), ask yourself if the pointer prior could be NULL. If the pointer is NULL, then current->data would seg fault!

- In this case, we know that (a) if the list was empty Step #2 would have caused zero to be returned and this code would not have executed and (b) our loops ends when current->next is NULL not when current is NULL. This means the loop ends before current gets set to NULL. Therefore, we are confident that current cannot be NULL with a valid linear linked list!

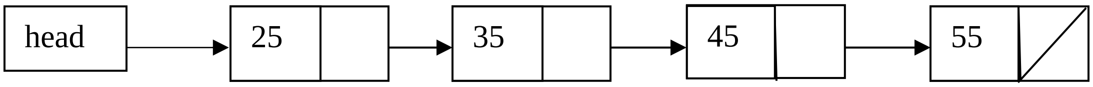
Step 11 Line #16: return current->data;



Returns the value 55.

Notice no use of delete is needed, since we are not yet done with our dynamically allocated linear linked list.

The memory for current will automatically be released at the end of the function and we will be left with the following pointer diagram:



Review of Recursion

- Recursion is repetition (by self-reference); it is caused when a function calls/invokes itself. Such a process will repeat forever unless terminated by some control structure.

- So far, we have learned about control structures that allow C++ to iterate a set of statements a number of times. In addition to iteration, C++ can repeat an action by having a function call itself. This is called recursion. In some cases it is more suitable than iteration. You will soon see why!

- When a recursive call is encountered, execution of the current function is temporarily stopped. This is because the result of the recursive call must be known before it can proceed. So, it saves all of the information it needs in order to continue executing that function later (i.e., all current values of all local variables and the location where it stopped). Then, when the recursive call is completed, the computer returns and completes execution of the function.

- In order for your recursive calls to be useful, they must be designed so that your program will ultimately terminate. As with iteration or looping, there is danger of creating a recursive function that is an infinite loop! We need to be careful to prevent infinite repetition.

- The best way to do this is to use an *if* statement to determine if a recursive call should be made -- depending on the value of some conditional expression.

- Eventually, every recursive set of calls should reach a point that does not require recursion (i.e., this will stop recursion).

- Therefore, there are <u>three</u> requirements when using recursion:

  A) Every recursive function must contain a control structure that prevents further recursion when a certain state is reached.
  B) That state must be able to be reached each time you run the program.
  C) When that state is reached, the function must have completed its computation and (if the function returns a value) return the appropriate value for each recursive call.

SAMPLE Design using Recursion

When solving problems with recursion, think about the special cases that would help us move from an iterative solution to a recursive solution. Remember recursion is all about breaking the problem down into smaller sub-problems and re-calling the function over and over with a different piece of the problem. This can be done by asking ourselves a set of relatively simple questions.

Problem: Display just the last node’s data and return the number of items in the list

1. What is the simple case: *(what case does not require a loop?)*

   If head is NULL, there is nothing to display
   Return zero in this case

2. Are there other cases to consider: *(why would a loop stop?)*

   If head->next is NULL, display the last node

3. What needs to be done BEFORE going to the smaller sub-problem:

   Make sure this node gets counted!

4. What is the incremental step that transforms the problem into a smaller sub-problem?

   Call the function with head->next
   ***** Make sure to use the returned value (the count) from the function call!!**

5. Is there anything that needs to be done AFTER the smaller sub-problem has been solved?

   Return the count

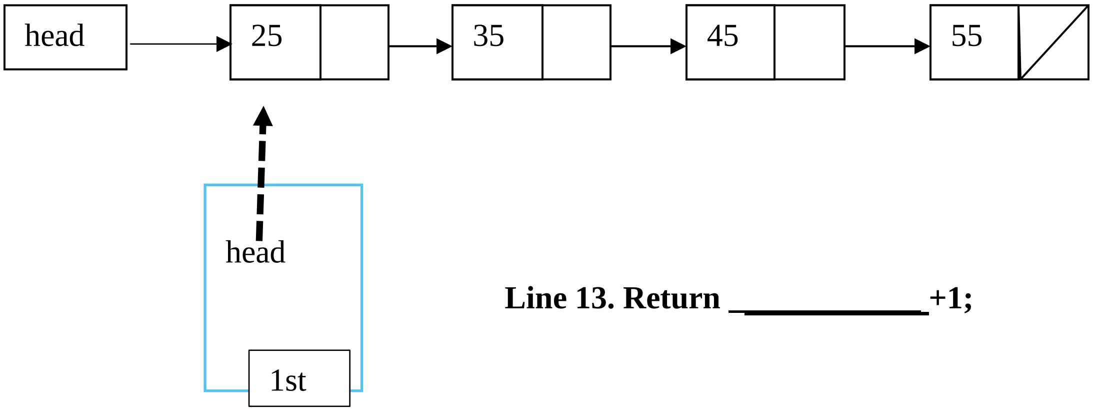
Tracing Recursion with Pointer Diagrams

The key to successfully understanding how to work with linked data structures is in becoming proficient tracing code using pointer diagrams. This page demonstrates the step by step process for working through code with a pointer diagram. Make sure to work line by line and do not skip any steps.

```
1)  #include "cs202_list.h"
2)  //Display the contents of a LLL recursively and return the number of items in the list
3)  int display(node * head)
4)  {
5)      //Base case
6)      if (!head)
7)          return 0;
8)
9)       //Performing the operations required
10)      cout << head->data <<endl;
11)
12)     //Progressing to the next smaller sub problem
13)     return display(head->next) + 1;
14) }
```

Step 1 First invocation.

a. Line 6. Head is not NULL, so the condition is False
b. Line 10. Output the data at head->data (25)
c. Line 13. Suspend execution and call the function again with head->next

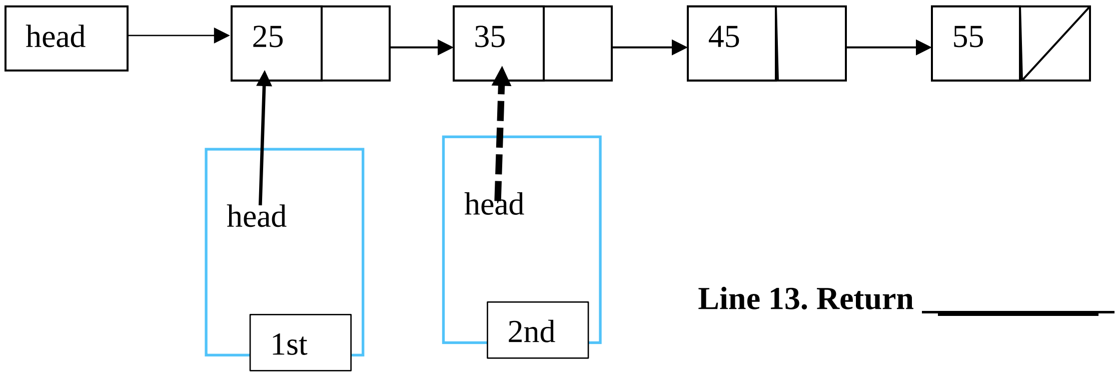


Line 13. Return ____________+1;

Notice with pass by value, in this first invocation head is a copy of the real head of the list sent in as an argument. Although we can change the data at the first node, we do not have the ability to change the actual head pointer to the list.

Step 2 Second invocation.

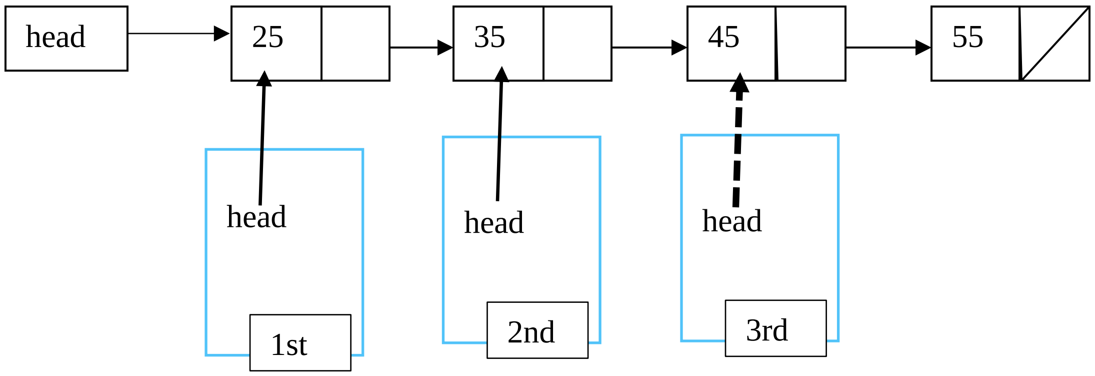
a. Line 6. Head is not NULL, so the condition is False
b. Line 10. Output the data at head->data (35)
c. Line 13. Suspend execution and call the function again with head->next



Line 13. Return ____________ +1;

Step 3 Third invocation.

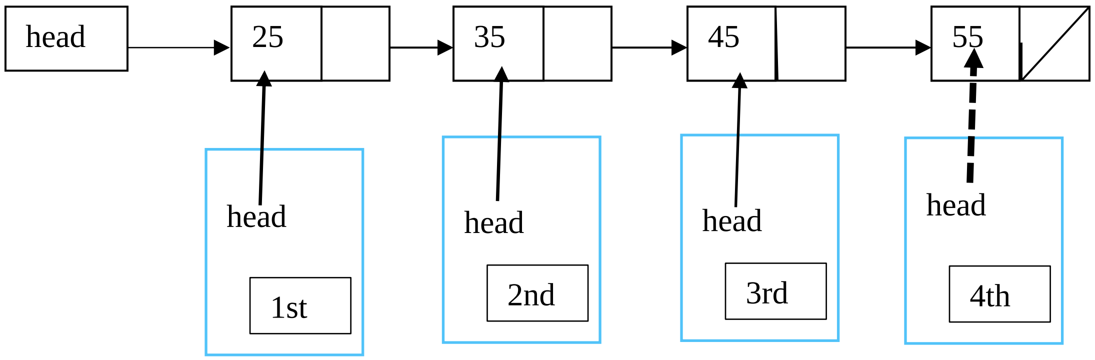
a. Line 6. Head is not NULL, so the condition is False
b. Line 10. Output the data at head->data (45)
c. Line 13. Suspend execution and call the function again with head->next



Line 13. Return ____________ +1;

Step 4 Fourth invocation.

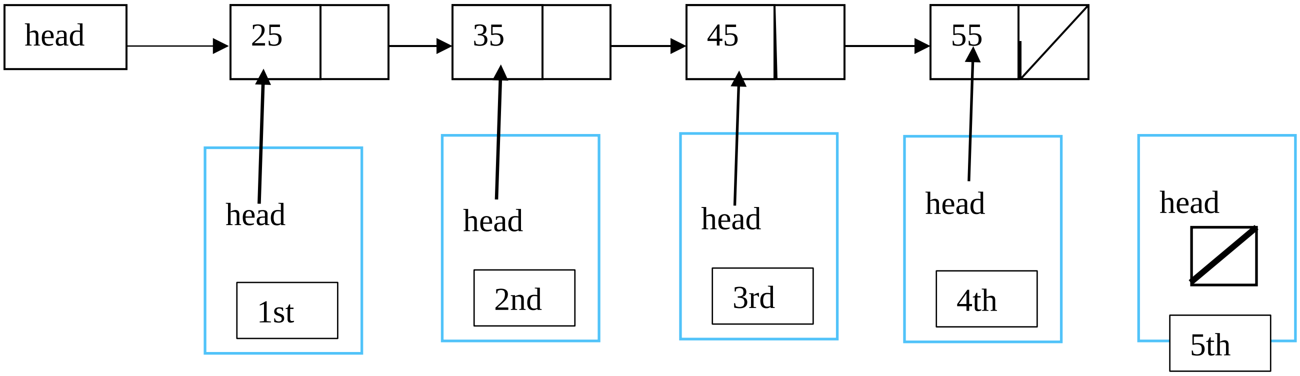
a. Line 6. Head is not NULL, so the condition is False
b. Line 10. Output the data at head->data (45)
c. Line 13. Suspend execution and call the function again with head->next



Line 13. Return ____________ +1;

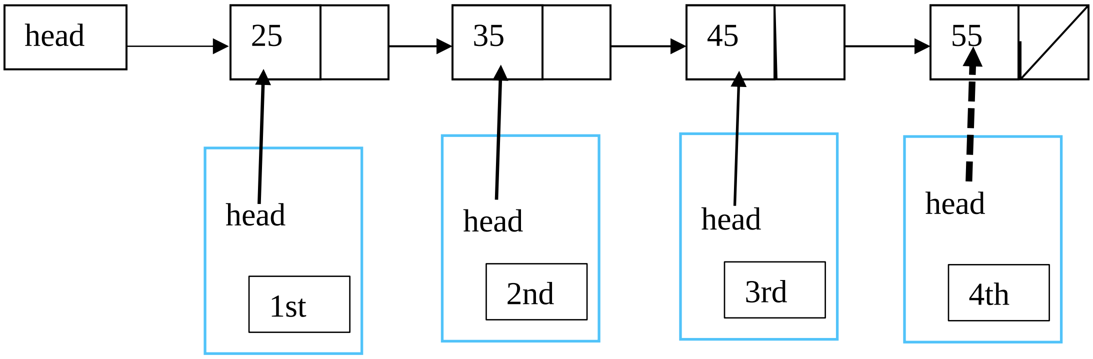
Step 5 Fifth invocation.

a. Line 6. Head is NULL, so the condition is True
b. Line 7. Return 0



Step 6 Return back to the Fourth invocation.

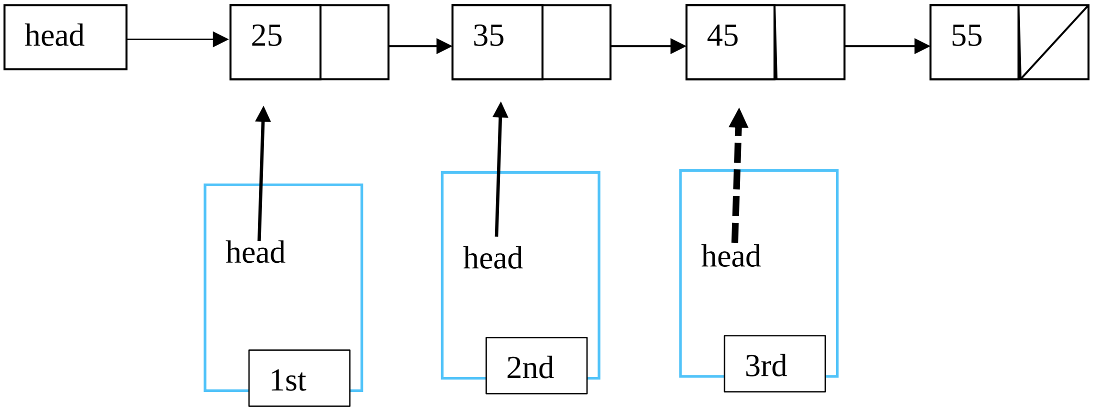
a. Line 13. We received zero (returned)
b. Line 13. Return 0 + 1



Line 13. Return _0___ +1;

Step 7 Return back to the Third invocation.

a. Line 13. We received one (returned)
b. Line 13. Return 1 + 1

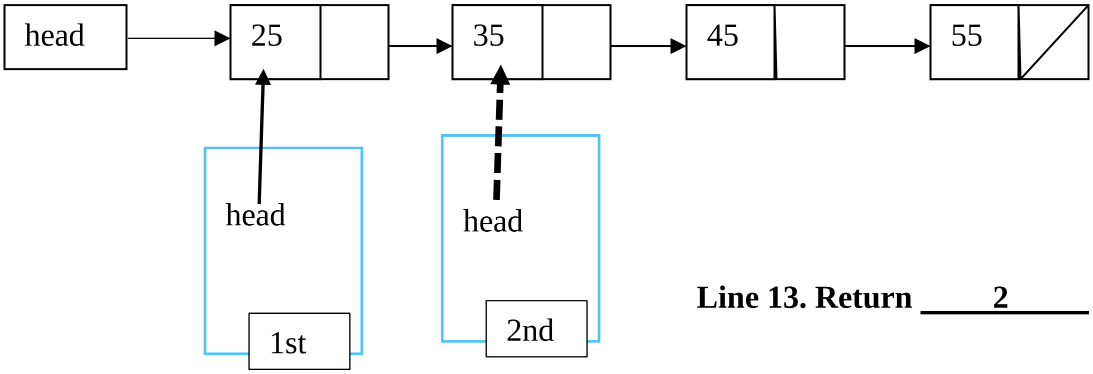


Line 13. Return 1 **+1;**

Step 8 Return back to the Second invocation.

a. Line 13. We received two (returned)

b. Line 13. Return 2 + 1

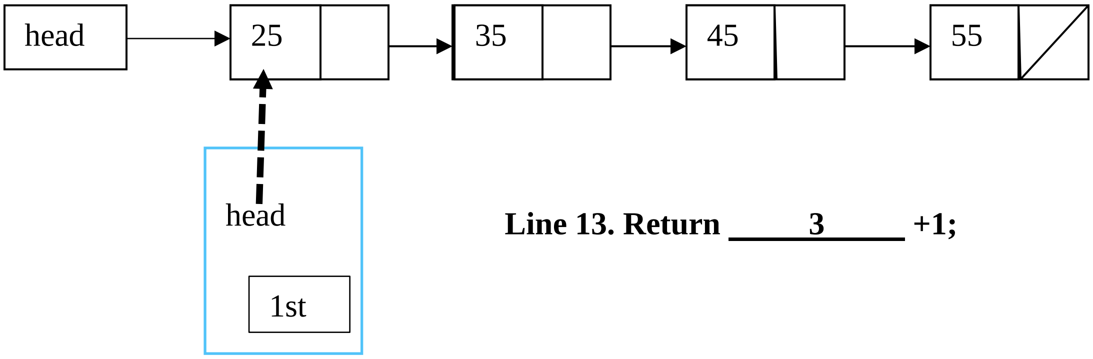


Line 13. Return _____2______ +1;

Step 9 Return back to the First invocation.

a. Line 13. We received three (returned)

b. Line 13. Return 3 + 1



Line 13. Return ______3______ +1;

Step 10 Return 4 back to original calling routine, the number of nodes in the list!

Review of Binary Search Trees (BST)

A binary search tree applies the same binary search algorithm we used with arrays to a tree data structure – where nodes are hierarchically arranged. Binary trees have two child pointers, one to the left subtree and the other to the right subtree. The binary search algorithm requires that our data be ordered to begin with – with data that is smaller than the root being placed to the left and data larger (or equal) to the right. We will implement the BST algorithms recursively.

Some information to keep in mind when working with BSTs:

1. Always add at a leaf, requiring traversal
2. The shape of a BST is dependent on the order in which data is inserted
3. The shape of a tree greatly influences the ability to reach the performance expected by the binary search algorithm
4. Instead of using “head” we now use “root” as the pointer into this data structure
5. Nodes are not stored in a linear fashion
6. An empty tree has a NULL root pointer
7. A leaf has NULL left and right child pointers
8. A full tree is where every leaf is at the height of the tree; in the strictest scenario, a full tree is one where every leaf is at the height of the tree and every node that is not a leaf has two children. (Be cautious of other definitions)
9. The height of a tree is the longest path from root to farthest leaf
10. A complete tree is one where you have a full tree at height minus 1 and at the full height of the tree, nodes are filled from left to right.

Tracing a BST with Pointer Diagrams

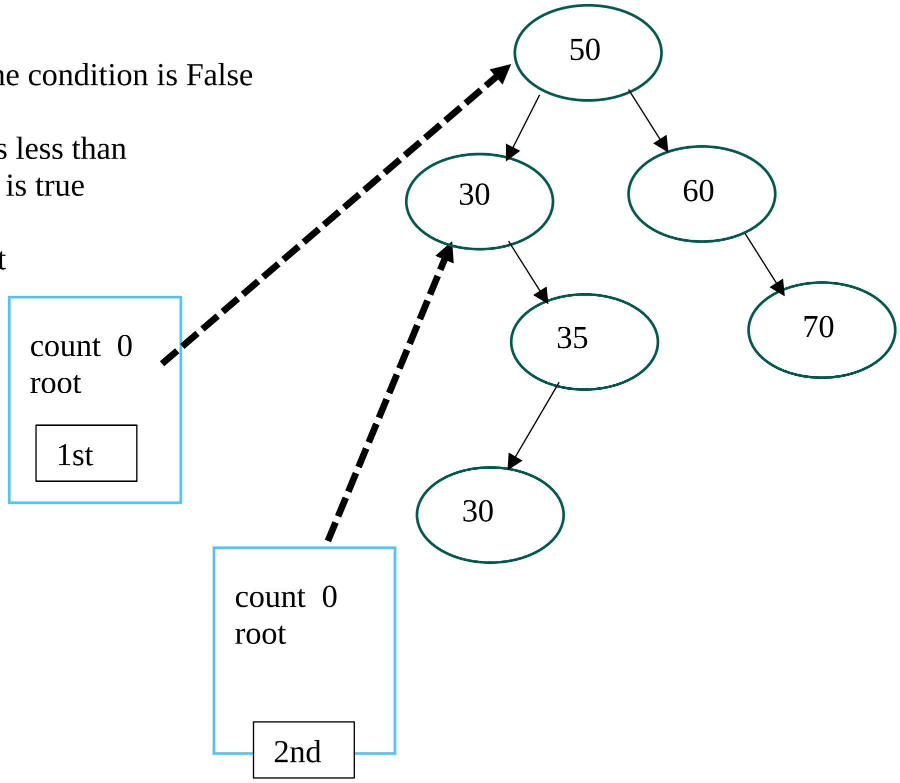
The key to successfully understanding how to work with linked data structures is in becoming proficient tracing code using pointer diagrams. This page demonstrates the step by step process for stepping through code with a pointer diagram. Make sure to work line by line and do not skip any steps.

For this example, we will count the number of times the value 30 is in the BST.

```
1.  int table::count_matches(node *root, int to_find)
2.    {
3.      int count = 0;
4.      //Base case
5.      if (!root)
6.        return 0;
7.
8.      if (root->data == to_find)
9.        ++count;
10.
11.     if (to_find < root->data)
12.       return count_matches(root->left, to_find);
13.   return count_matches(root->right,to_find) +count;
14.   }
```

Step 1 First invocation.

a. Line 5. Root is not NULL, so the condition is False
b. Line 8. The data is not a match
c. Line 11. Because to_find (30) is less than root->data (50), the conditional is true
d. Line 12. Suspend execution.
   Call the function with root->left



Line 12. Return ____________ ;

Step 2 Second invocation.

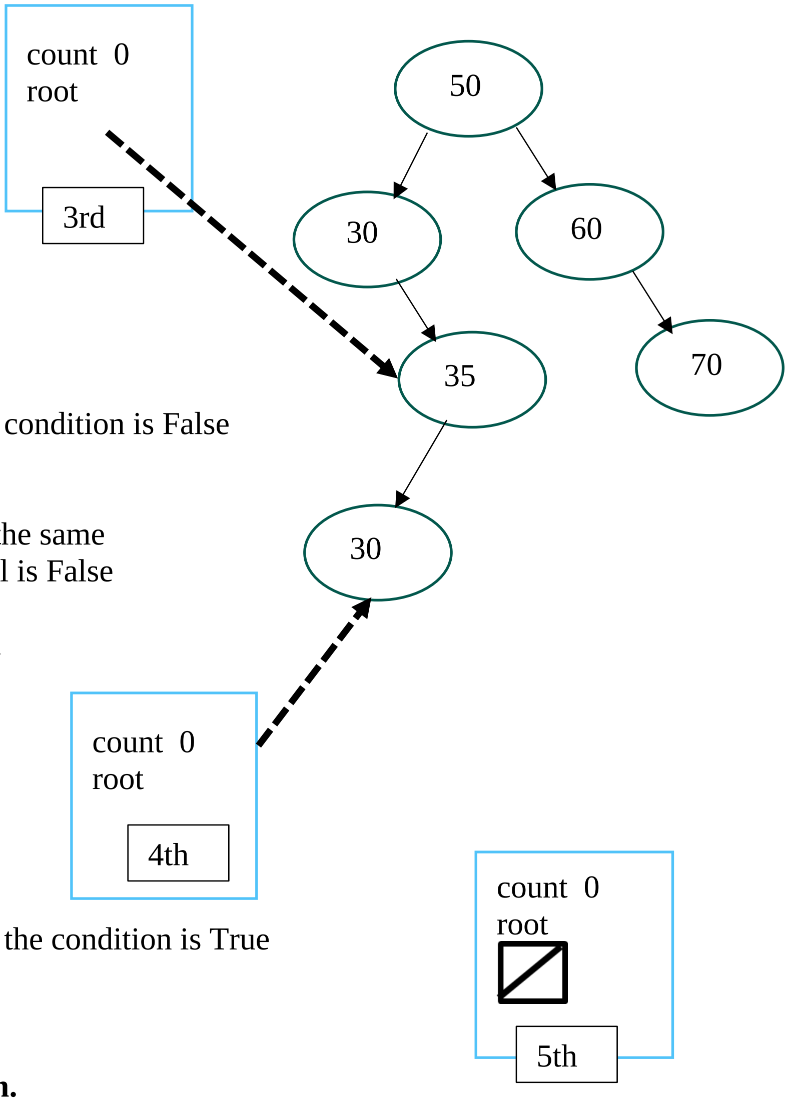
a. Line 5. Root is not NULL, the condition is False
b. Line 8. The data is a match
c. Line 9. Increase count to 1
d. Line 11. Because to_find (30) is equal to root->data (30), the conditional is false
e. Line 13. Suspend execution. Call the function with root->right

Line 13. Return _____________ +1;

Step 3 Third invocation.

a. Line 5. Root is not NULL, so the condition is False
b. Line 8. The data does not match
c. Line 11. Because to_find (30) is less than root->data (35), the conditional is True
d. Line 12. Suspend execution. Call the function with root->left

Line 12. Return _____________ ;



Step 4 Fourth invocation.

a. Line 5. Root is not NULL, so the condition is False
b. Line 8. The data is a match!
c. Line 9. Increase count to 1
d. Line 11. Because to_find (30) is the same as root->data (30), the conditional is False
e. Line 13. Suspend execution. Call the function with root->right

Line 13. Return _____________ +1;

Step 5 Fifth invocation.

a. Line 5. Root is NULL, so the condition is True
b. Line 6. Return zero

Step 6 Return to the Fourth invocation.

a. Line 13. **Return ____0____+1**

Step 7 Return to the Third invocation.

a. Line 12. **Return** ____**1**_____

Step 8 Return to the Second invocation.

a. Line 13. **Return** ____**1**_____ **+1**

Step 9 Return to the First invocation.

a. Line 12. **Return** ____**2**_____ to the original calling routine

CS202
Programming Systems

Lab #1: Getting Started with Object Oriented Programming

CS202 Lab #1: Goals & Process

Important!

Prior to the lab, make sure to get a MCECS account (required); visit FAB 88

Goals:
The primary goals for the first session are to: *****Please do not skip the group activities**

1. **First, collaborate:** We will begin each lab this term with the group exercises.
    - **Spend the first 10-15 minutes** of each lab on the group activities
    - For the first lab, this will be a great way to help you get to know the other students in lab and begin understanding the fundamentals of object oriented programming and advanced C++.
    - For the first set of group activities, we will:
        - Learn to use inheritance to create derived classes,
        - Distinguish the difference between containing, using, and derivation.
2. **Next, get prepared** to work in the CS linux environment for the labs and programming assignments this term for anyone new to PSU
    - You will need to use the linux and vim manual for this part
3. **Using linux:** If you are new to PSU and Computer Science, you will want to spend time during the first Lab to get comfortable with the following concepts. Students who have already taken CS163 at PSU can skip this step.
    - Learn how to download **putty** or ssh for PCs and use **terminal** for Macs
    - Experience logging in, entering a program, compiling and running
    - Learn how to submit programming assignments to D2L
    - Most information can be obtained by visiting the cat website in their Remote Access information:

    http://cat.pdx.edu/network/shell-and-ssh-access-2.html

4. **Individual Programming:**
    - Gain experience using multiple classes
    - Experience building hierarchical relationships
    - Ongoing practice of data structures whenever time permits!
5. Once the lab is completed, you can perform the **self-check quiz** at the end of the lab materials.
    - Use the self-check quiz questions at the end of the lab materials for each section to test if you fully understand the material

CS202 Lab #1 - Background Information

Inheritance Overview

1. All members of a base class are inherited in a derived class. But, a derived class has no access to a base class' private members. Protected members are accessible by derived class but not by clients or classes that are not derived.

2. Inheritance is established via a derivation list using this format**:**

   ```
   class derived: public base { /* body */}
   ```

   or,

   ```
   class derived: public class base { /* body */}
   ```

   - The keyword **class** in front of the base class name is called an incomplete declaration and allows classes to be used before they are defined

3. OOP should include single inheritance but not multiple inheritance; limit your use of multiple inheritance. This means each class should have at most one direct base class when working on programming assignments.

4. Derived class methods (eg., member functions) that are of the same name as the base class' methods "hide" the base class' version of the function.

   - The base class' public function can be called using the class name and scope resolution operator from outside the class:

     ```
     object.base::function(argument_list);
     ```

   - The base class' public or protected function can be called using the class name and scope resolution operator from a derived class' member function:

     ```
     base::function(argument_list);
     ```

Initialization Lists

Initialization lists are a critical component to building hierarchies

1. Constructors are the only functions that can have initialization lists
   An initialization list belongs where you implement the constructor and not with the prototype:
   ```
   derived::derived(int arg): base(arg), data_member(0)
   { /* body */}
   ```
2. Initialization lists must be used to initialize any constant data members
3. Initialization lists may be used to also initialize private data members
4. Initialization lists must be used to cause the base class' constructor to be invoked when working with constructors with arguments.
5. In fact, the only way to cause a base class' constructor with arguments to be used is for a derived class to specify the constructor "invocation" via an initialization list
6. The **order** of the items in the initialization list **is** important. The base class constructor must be specified first, followed by the data members that you elect to initialize. The order that data members are initialized is based on the order in which memory is allocated and the order in which constructors are invoked.
7. Initialization lists are a critical component to building hierarchies

Tips when Programming

1. Avoid passing or returning a class object by value
2. Always, write a default constructor for classes. This is a constructor that either has no arguments or all arguments have default values.
3. Anytime there is a constructor in a derived class that has an argument, always have an initialization list to cause the correct base class constructor to be invoked. The only exception to this is when you want the "default" constructor to be invoked from the base class.
4. Whenever there are classes with dynamic memory, make sure to have a destructor
5. Avoid using the assignment operator with class objects, if the class manages dynamic memory. Wait until we learn about operator overloading before using the = operator
6. When passing an object of a class to a function, if that function doesn't modify the argument then pass it as a constant reference
7. When writing member functions that don't modify the data members, make those member functions "constant" member functions. This can be done via:

   ```
   //notice the placement of the const keyword
   void display() const;
   ```

CS202 Lab #1 – OOP

IMPORTANT!
DO NOT SKIP STEP #16

Group Activity: Preparing to use OOP

Spend 10-15 minutes on this section. DO NOT SKIP THIS!

Level 1 - Introductory

_____Step 1. Let's say you wanted to create an object oriented program to manage employees at a local department store (eg., Macy's or Kohls). There are a variety of different types of employees (managers, part-time, full-time, etc.). There is also information that needs to be collected concerning each employee and their pay.

_____a. What classes make sense

____

_____b. What kind of relationship should exist between the classes ("is a", "using", "containing")

- "is a" – Hierarchical. A derived object IS a base object plus more
- "using" – A function uses an object of a class as an argument or as a local
- "containing" – A class "has an" object of a class as a data member

____

____

____

_____c. For any containing relationships, will any of the member functions of that class need to be called? ____________

- If so, how will it be possible to gain access? *Brainstorm*

____

_____d. What data members are needed and should they be private or protected?

____

Level 2 - Intermediate

_____Step 2. For the same problem, discuss what constructors you would need.

Hint: A constructor in a base class is not inherited by the derived class. If a base class constructor has an argument list, then the derived class must explicitly invoke that constructor through the initialization list.

Initialization lists belong in the implementation file (.cpp) not the class interface.

_____a. What constructors will be needed?__________________________

_____b. What arguments should the constructors have?

_____c. Why did you select these arguments?

_____Step 3. Think about the member functions.

Hint: Member functions without arguments can only get their data from (a) data members, (b) from input operations (e.g., user or files), and (c) globals.

_____a. What arguments did you select? And why?

****** RECEIVE feedback from Lab assistants before continuing with linux & vim exercises***

Individual Coding

Check-off each step as you proceed! Submit the code at the end of lab.

In the second phase of this lab, we will be working with an existing set of classes implementing a subset of the problem. You have access to the .h files to see what functions and data members are available. It is important to follow the guidelines below precisely! Your job will be to implement the following pieces to experience (a) inheritance and (b) initialization lists.

Level 1 - Introductory

_____Step 4. Login to **cs202lab.cs.pdx.edu** using your assigned login and password

- Change into the CS202/Lab1 directory **cd CS202/Lab1**
- Use a **linux editor** such as **vi, vim, or emacs** to type in your recursive functions
- **Compile** and link to my object code on **linux**. **g++ *.cpp *.o -g -Wall**
- **Run** your program by typing: **./a.out**

_____Step 5. Examine the .h files for the correct prototypes. *Spend time understanding the design.*

_____a. Modify the **person.h** file to have **derivation lists** for person to be derived from a name. *A person "is a" name but "has an" address.*

_____b. Add a wrapper function prototype in the person class so clients can change the address.

Prototype: ____

_____c. Modify the **employee.h** file to have **derivation lists** for employee to be derived from a person. *An employee "is a" person and "has a" review.*

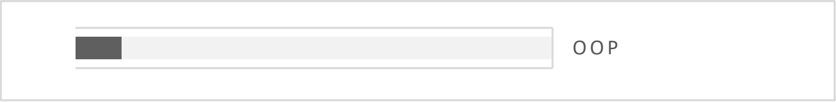
Write the derivation list here:__________________________________

_____d. Notice the use of the const keyword.
What is common about all of the functions that have the const keyword afterward?_____________________

_____e. Notice that there isn't a destructor in the person class. Explain why:

____

Proficiency Scale after Step #5:



_____Step 6. Implement the following member functions in the **person.cpp** file:

_____a. Implement the constructors. Initialization lists must be used when implementing a derived class constructor with arguments to ensure the base class constructor is invoked. They can also be used to initialize data members. Specify them here:

address::address():______________________________________

address::address(const address & to_copy):__________________

name::name(const name & to_copy):_______________________

person::person(char * initial_name, char * street, char * zip): ____

____

_____b. For the person default constructor, no initialization list is needed. Explain why:

____

_____c. Implement the wrapper function in the person class that calls the change_address function for a person object.

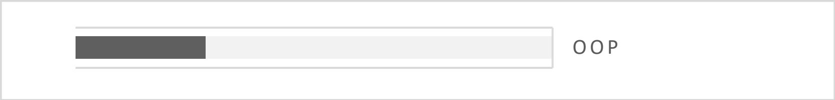
Why can't we have an initialization list for this wrapper function?

____

_____d. **Compile** and link to my object code on **linux**.

```
g++  *.cpp  *.o  -g  -Wall
```

Proficiency Scale after Step #6:



_____Step 7. We will start by working on the constructor in the **employee.cpp** file. The **constructor for the employee class** accepts the employee’s name and other values as arguments. Remember the purpose of the constructor is to initialize the data members to their zero equivalent value.

First plan:

_____a. Should an initialization list be used to initialize data members? _________

_____b. If an initialization list was not used, how else could the employee id be initialized to the value passed in:________________________________

_____c. Which is more efficient?______________

_____d. How can a base class constructor be invoked from a derived class?
____

_____e. In what file (.cpp or .h) does the initialization list belong?____________

_____f. Remind yourself of the syntax of an initialization list:

________::_________(char * name, char * street, char * zip, int id)_______________

Now implement the code:

_____g. Create an initialization list to cause the base class’ constructor with a name argument to be invoked

_____h. Use the initialization list to also set up the employee id accordingly

_____i. Did you use an initialization list to initialize the root pointer?__________

_____j. **Compile** and link to my object code on **linux**.

```
g++  *.cpp  *.o  -g  -Wall
```

Proficiency Scale after Step #7:

Level 2 - Intermediate

_____Step 8. Implement the **hourly** employee **constructor** with a person as an argument in the **employee.cpp** file

_____a. Initialize the data members to their zero equivalent value

_____b. Create an initialization list to cause the base class’ constructor with a person as an argument to be invoked

_____c. **Compile** and link to my object code on **linux**.

```
g++  *.cpp  *.o  -g  -Wall
```

Proficiency Scale after Step #8:

_____Step 9. Implement the “wrapper” **display** function for the **employee** class in the **employee.cpp** file. The performance review information for an employee is stored in a binary search tree. Your job is to write the employee “wrapper” display function that will call a recursive display function.

_____a. Implement the wrapper display function and have the employee display function call the base class display to display the base class members. This should be done first.

_____b. Then, display the ID number for the employee

_____c. Lastly, have the employee display function act as a “wrapper” function by calling a recursive display function, for the personnel records. (We implement the recursive display function in Step 11)

_____d. **Compile** and link to my object code on **linux**.

```
g++  *.cpp  *.o  -g  -Wall
```

Proficiency Scale after Step #9:

Level 3 - Proficient

_____Step 10. **Prepare** to work with the performance review data structure (a **BST**) by examining the node class **(node.h)** to find out what functions to use. *This will have a benefit for your programming assignments since we will no longer be using structs for nodes in CS202!*

_____a. Start by examining the **node.h** file and look at each of the function prototypes available.

_____b. How would you traverse. Instead of saying root->left in our function call, we would instead say: display (____________________)

_____c. Why do the go_left and go_right functions return a pointer by reference?

____

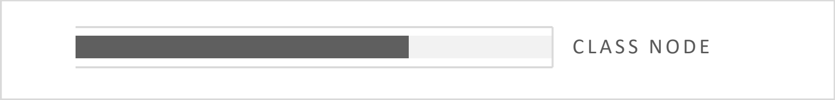
_____d. What do you think the connect_left and connect_right functions do?

____

_____e. Notice, there is no default node constructor. Therefore, when allocating a node we must always provide a value. Let’s step through the syntax:

```
root = new node________________________________;
```

Proficiency Scale after Step #10:



_____Step 11. Implement the recursive **display** function for the **employee** class in the **employee.cpp** file. This will display the performance review.

First plan:

_____a. What is the simple (base) case? ______________________

_____b. What is the incremental step to get to the next smaller sub-problem

____

_____c. What needs to get done before going to that next smaller sub-problem?

____

_____d. What needs to get done after returning from that smaller sub-problem?

____

_____e. **Implement** the recursive function.

_____f. **Check** main in **cs202_lab1.cpp** to make sure it fully tests this function

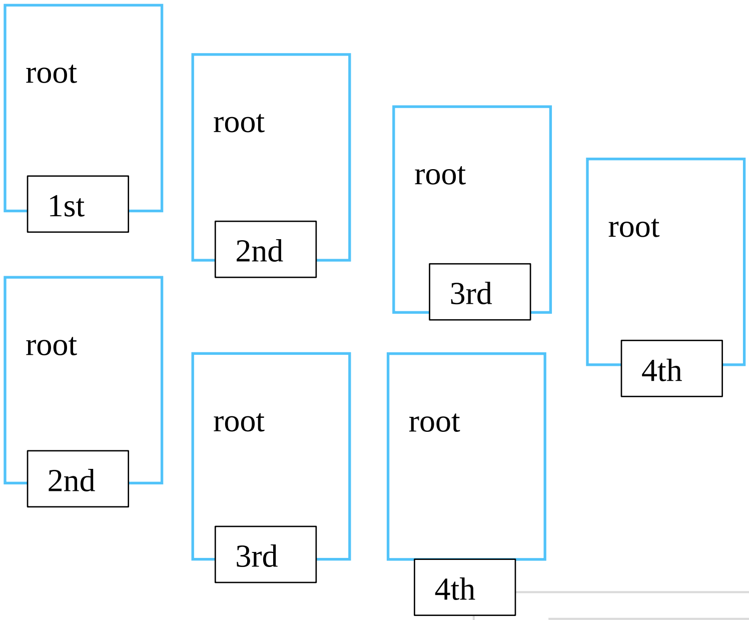
_____g. **Compile** and link to my object code on **linux**.

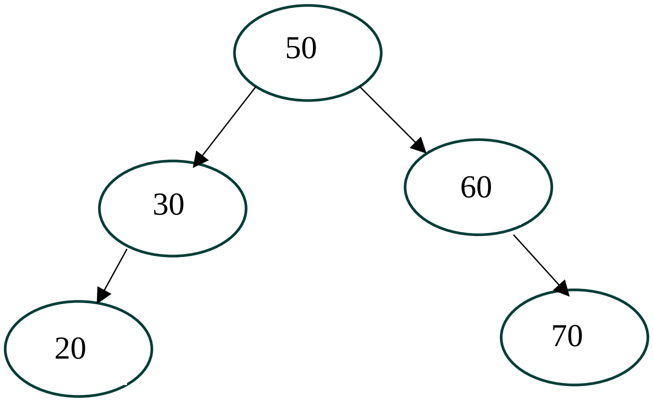
```
g++  *.cpp  *.o  -g  -Wall
```

_____h. **Develop the test plan:** Think about how to test out all conditions:

Test Case(s)	Expected Result	Verified? (yes/no)

_____i. Trace your code with a pointer diagram





Proficiency Scale after Step #11:

______Step 12. Implement the **recursive copy** of a BST, creating a new BST in the **employee.cpp** file

______a. From the node class (**node.h**) what functions would you need to use:

____

______b. What is the simple (base) case? _______________________

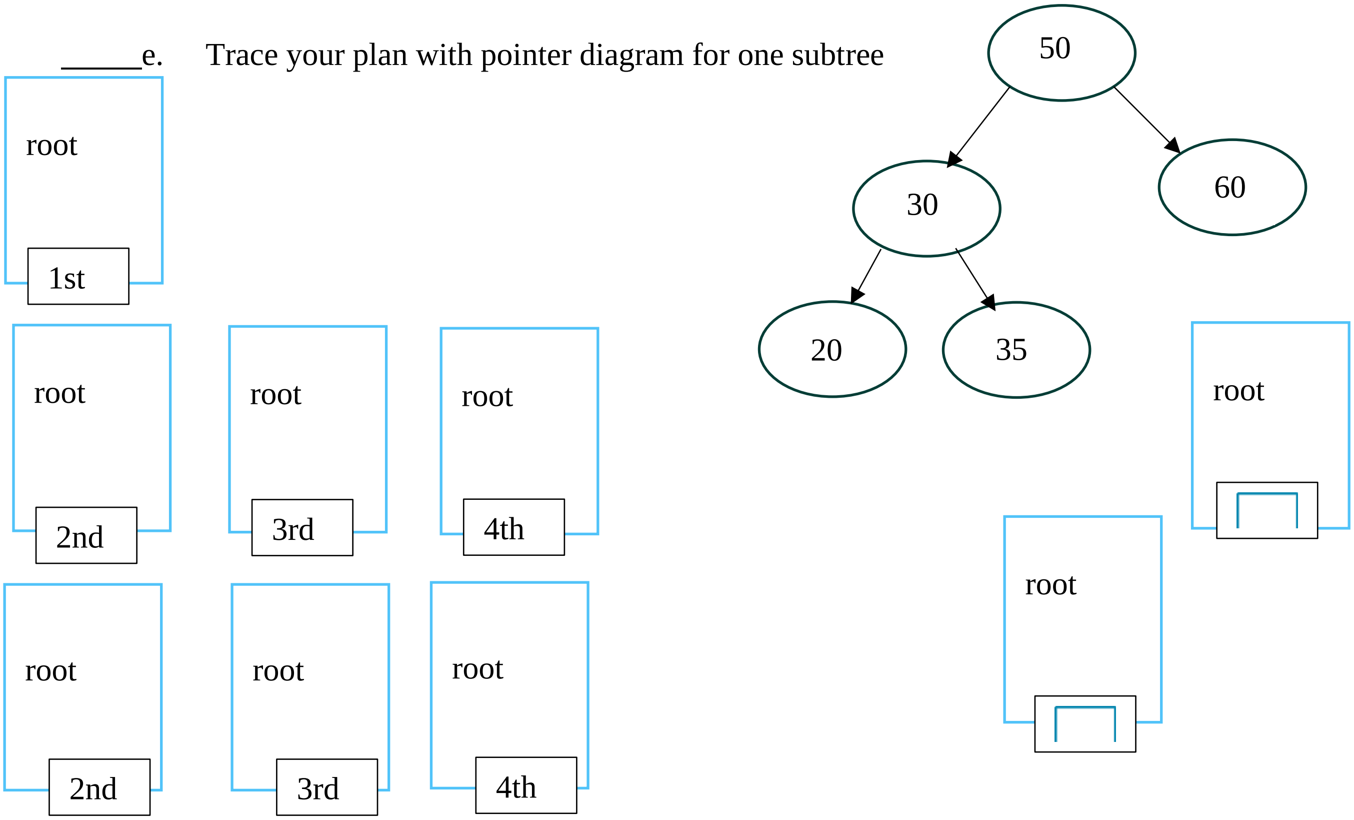
______c. What is the increment step to get to the next smaller sub-problem

____

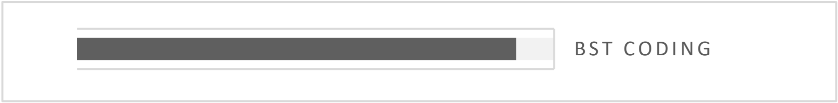
______d. What needs to get done before going to that next smaller sub-problem?

____

______e. Trace your plan with pointer diagram for one subtree



Proficiency Scale after Step #12:



______Step 13. **Modify** main in **cs202_lab1.cpp** to test out this function

______Step 14. **Compile** and link to my object code on **linux**.

```
g++  *.cpp  *.o  -g  -Wall
```

_____Step 15. **Run** your program by typing: **./a.out**

IMPORTANT!

PERFORM the NEXT step
even if you did NOT finish the entire lab!

_____Step 16. **Before** submitting, we will create a script to help avoid issues when creating a tarball. It is common for people to forget to specify the "destination" file as the first argument when building a tar archive. Often, source files are placed first causing your code to be completely overwritten.

_____a. At the linux prompt create a new file (**vim cs202_tar**)

_____b. Enter the following lines. $1 represents the first argument. This will take all of your .cpp and .h files and place them in an archive of the name provided when you run the script

```
tar –cf   $1.tar  *.cpp   *.h
```

_____c. Save and exit (:wq with vim)

_____d. Make the script executable (**chmod +x cs202_tar**)

_____e. Run the script by typing: **./cs202_tar Lab1**

_____f. Look at the files in your directory (**ls**). Notice that there is an additional file now named **Lab1.tar**

_____g. Now let's double check that the archive contains everything we wanted! Make a new directory so we can unpack the archive (type: **mkdir Unpack**)

_____h. Change into this new directory (type: **cd Unpack**)

_____i. Unpack the archive, type: **tar –xvf ../Lab1.tar**

_____j. Examine the files in your Unpack directory.

Are there .cpp and .h files? ____________ (type: **ls**)

Are their contents intact? _____________(type: **cat *.cpp**)

_____k. Go back to the directory with your lab files and submit (type: **cd ..**)

_____Step 17. **Submit** your program by typing **./submit** at the linux prompt

Lab #1 Completed

ID Scanned on Completion (TCSS Initials):__________

CS202 Lab #1 – Practice Data Structures (LLL)

Work on these after finishing the lab or to practice outside of Lab

Practice LLL: The following questions should be implemented on a continual basis to prepare for your midterm proficiency demo. Start by implementing these iteratively, then implement each recursively. We suggest working in the CS202/mpdemo/LLL directory for these.

You will notice that in CS202 we will be using classes to manipulate our data structures. This means for each private recursive function there will be a public wrapper function!

After completing each of these, re-examine each with **a tail pointer**. Implement them first iteratively and then follow that with recursive solutions! Make sure you can **draw the pointer diagram** for each!

a. Solve each Iteratively

b. Solve each Recursively

c. Double check that each path through the function returns the correct value if you implement this with more than a single return statement in the function

d. Make sure when calling the function that the returned value is being used!

1. Display only unique data in the LLL
   - Return the number of items displayed

2. Display just the last piece of data in a linear linked list, if it isn't the same as the first node's data
   - Return the data displayed

3. Calculate the average of all unique data in a LLL
   - Return the average

4. Move the first node and add it to the end of a linear linked list
   - Return the sum of all the data in the list

5. Duplicate the first node and place it at the end of a linear linked list
   - Return the sum of the last two nodes in the list

6. Remove all duplicate data in the LLL
   - Return the number of data removed

7. Remove all BUT the last node in a linear linked list
   - Return the sum of all of the data in the list

8. Copy the contents of a linear linked list and make a new linear linked list
   - Return the number of items in the new list

9. Copy the contents of a linear linked list and place the data into an array.
   - Return the number of items copied

10. Copy the contents of a linear linked list EXCEPT do not copy any 2's and place the data into an array.
    - Return the number of items copied

CS202 Lab #1 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

Single Inheritance: Assume that you have an hourly_employee derived from an employee class. Employee is derived from Person and has an address object as a data member.

1. List the containment relationships: ____
2. List the hierarchical relationships: ____
3. Write the derivation list for the hourly_employee class

   ____

Now assume that the address class has a char * data member for the street name, the person class has a char * data member for the person's name, and the hourly_employee class has a root pointer to a binary search tree of the employee's work history. The work history is stored as char * data, organized alphabetically

4. Write the default constructor for the hourly_employee class, using initialization lists:

   hourly_employee()
   {

   }
5. Write an hourly_employee constructor that takes previous work history (a char *) as an argument for a new employee:

   hourly_employee(_________________)
   {

   }

6. If we allowed an employee to start working with a large work history already established (maybe transferring from another part of the company, what would the hourly_employee constructor need to do if we were to implement it (just describe):

Assume each class has a "display" member function.

7. If the hourly_employee's display function is called, how can all the data within the base classes be displayed?

8. Let's say the person class has a display function that takes an integer argument, and all of the rest have no arguments. How can the client of an hourly_employee class call the base class function?

9. Describe what we mean by function overloading when working within a hierarchy

10. Explain the advantage of inheritance (1-2 sentences):

11. A derived class inherits which members from the base? ____________________

12. What is the advantage of a protected member?_____________________________

13. What is the advantage of a private member?_______________________________

14. If both a derived class and a base class have the same function, but with different arguments – show how it is possible to call the base class version of the function from a client program. Remember that function overloaded is limited to a single scope and does not work across the boundaries of clases:

    ____

CS202 Lab In Summary

Review the most important concepts

1. A derived class specifies its parent through a **derivation list**
2. A derived class with only a single parent creates a **single inheritance hierarchy**
3. We use **public derivation** to allow the public methods (i.e., member functions) to be used directly by the client of the derived class. This means that if you create a derived object, the public methods from the derived and base classes are available to be called.
4. There are three forms of relationships with OOP: Containing ("has a"), Specialization ("Inheritance" or "Is a") and Using. The benefit of specialization is that we can avoid the need to build wrapper functions to gain access to the contained object, simplifying the code!
5. A derived class <u>always</u> has full access to the base class public and protected members. There is nothing you can do to alter this. Is it the nature of inheritance.

6. Although tempting to implement, **multiple inheritance** (where there is more than a single parent) rarely reduces the complexity of the code and is typically not considered to be object oriented.

7. Every class has a default constructor (i.e., a constructor with no arguments). This constructor does nothing unless you implement one. As soon as you implement a constructor for a class, the automatically supplied default constructor is not provided. Therefore, you should <u>always</u> provide a default constructor implementation that sets the data members to their zero equivalent value. No exceptions should be made to this rule!
8. Keep in mind that arrays (statically and dynamically allocated) <u>always</u> expect there to be a default constructor.
9. Within a hierarchy, when we create an object of a derived class the default constructors are automatically invoked. There is nothing you need to do to cause this to happen. The first constructor invoked is for the most indirect base class. The last is for the derived class to which an object is formed.
10. As soon as you write a constructor with an argument, you need to start providing initializations lists.
11. Initialization lists apply only to constructors. They do not apply to any other function.
12. Initialization lists are the only way to tell the compiler which base class constructor should be used when working within a hierarchy where base class constructors have arguments. If an initialization is not used, the compiler will assume that the default constructor (with no arguments) will be used.
13. Remember to never pass a class object by value; if the argument is not being changed, then pass it as a constant reference.
14. Never return a class object by value
15. When we create functions that don't modify the data members, they need to be specified as **constant member functions**. Look closely at where the const keyword belongs.

CS202
Programming Systems

Lab #2: Inheritance

CS202 Lab #2: Goals

Important!

Always backup your files!!!!!!!!!!

Lab #2 contains the following components; **each** of these components **must be completed** as part of this lab:

1. **Start working in a group** and learn about using CRC cards to design an object orientated solution.
   - Limit these to 10-15 minutes
   - Do not skip the group activities!

2. **Work with the linux and vim exercises** for the next *10 minutes*
   - Perform exercises to become familiar using vim

3. Prepare to program by **planning**; examine the store.h file to now work with a BST of employees.

4. **Implement** C++ functions that are assigned on the cs202 lab system. We specifically will learn about copy constructors and continue to use a class to represent nodes.

5. Apply **test plans** and **pointer diagrams** to verify your software. **Using test plans** will help solidify the test plans that you will be creating for your programming assignments!

MAKE SURE TO READ THE

BACKGROUND INFORMATION for Labs 1 and 2

PRIOR to

completing the Prelab Exercises!

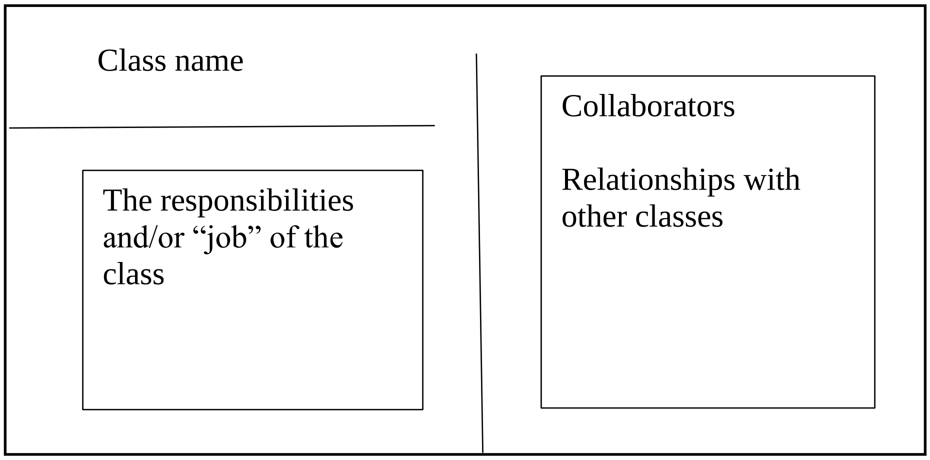
CS202 Lab #2 – Background Information

CRC Cards

In Lab #2 we will use CRC cards to assist with representing classes in an object oriented system. CRC stands for Class, Responsibilities, and Collaborators. CRC cards are a simple way to represent what classes exist in a system, what their responsibilities are, and how relationship between classes get established. Remember in CS202 we are no longer working with a single class. Instead, we are working with at least 5 classes in each problem dividing up the task and having the classes collaborate or work together, each responsible for their own part. In Lab we will represent CRC cards with 3x5 or 4x6 index cards.

The idea is that we want each index card to represent an individual class. We will use these cards to arrange them on our table in a way that helps us visualize the relationships between the classes being designed. It also lets us test out different changes and in a variety of combinations without significant.

CRC cards are set up by dividing a card in half. On the top we write the name of the class. On the left we write the responsibilities of the class. On the right side we write the collaborations (what classes this class will be working with). They may appear as follows:


Class name
The responsibilities and/or "job" of the class
Collaborators
Relationships with other classes

Rules of Recursion:

1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

When solving a problem recursively, ask yourself these questions:

1. Is the problem repetitive in nature?
2. What is the base case?
3. Are there other stopping conditions not considered?
4. What needs to be done in each of these cases; outline them specifically.
5. Does anything need to be done prior to the recursive call?
6. Does anything need to be done after the recursive call?
7. If information is being returned, are we returning the correct data?
8. Are there paths through the algorithm where nothing is being returned?
9. When data is being returned, are we using that data when it comes back after calling the function?

Remember when programming:

- Make sure to check before dereferencing your pointers.
- Keep in mind that a CLL ends when the next pointer is pointing to the same place as rear!
- **NEVER** say pointer.member (The direct member access operator (.) does not dereference your pointer!)
- **NEVER** have a return inside a loop!
- **NEVER** break from a loop

CS202 Lab #2 - Pre-lab Worksheet
Inheritance

This worksheet must be completed to attend the Lab.

Pre-Lab #2 (2 Pages)

Understanding OOP:

_____1. Let's think about extending the hierarchy we worked on for Lab #1. In that lab, we worked with a single employee – establishing their name, address, and other important information. But how could this be extended for multiple employees and multiple stores? What additional classes would make sense and what relationships would exist between these classes ("is a" versus "has a" or containing)?

_____2. Is there a place where it will be difficult to access or work with the data? *Remember a derived class has direct access to a base class' public and protected members, without the use of another object. But, this is not the case with containing relationships where an object is needed and public members are the only ones available!*

_____3. Where will your design need wrapper functions? *Why?*

_____4. Explain when the **.** (direct member access operator) is used versus the **->** (indirect member access operator) is used:

When do we use the . operator:______________________________

When do we use the -> operator:_____________________________
What is -> equivalent to? _______________.member

For Programming Assignment Assistance with OOP:

_____5. For your first programming project, list all nouns:

_____6. Group these nouns into categories. Ask yourself, does a class make sense for each category? If not, regroup!

_____7. Where is there inheritance? (Avoid basing the hierarchy solely on the data structures)

_____8. Are you designing the OOP independent of the data structure?__________

____________ If not, rethink!

_____9. Now for the data structure. How do you intend to represent a node?

_______What is the purpose of a node?

_______Can you design make sense without a node?

As a class, what member functions would be needed? (Make a list)

Pre-lab #2 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #2 – Inheritance

Check-off each step as you proceed! Submit the code at the end of lab.

Experience OO Design using CRC cards: We will continue lab #1 where we were working with managing employees at a local department store. This time we want to extend it to allow for different types of employees: part-time, full-time salaried, and full-time hourly. And, we want to be able to handle multiple employees and ultimately multiple stores. You will be working in groups creating solutions to the following questions. You may use your pre-lab work to speed up the process!

Group Activity: Using CRC cards

*** Spend no more than 10-15 minutes on the group work

Level 1 - Introductory

_____Step 1. What additional classes would make sense for supporting different types of employees? Using CRC cards create 3 classes to represent this problem:

- class name
- responsibilities
- collaborations with other classes

____

____

____

_____Step 2. Now extend this to allow for multiple employees? What would you do (briefly)

____

____

Level 2 - Intermediate

_____Step 3. What would the relationship be between these classes (“is a” versus “has a”)

____

____

****** RECEIVE feedback from Lab assistants before continuing with linux & vim exercises***

Individual Coding

Practice Data Structures: In the second phase of this lab, we will be working with the same classes as lab #1 but this time completed. Your job will be to implement the following pieces to experience OOP and data structures. In this lab, you will be modifying **store.cpp** and **store.h**. All of the person and employee code itself has already been implemented. The store has multiple employees in a binary search tree.

Level 1 - Introductory

_____Step 4. Login to **cs202lab.cs.pdx.edu** using your assigned login and password

_____a. Change into the CS202/Lab2 directory **cd CS202/Lab2**

_____b. Use a **linux editor** such as **vi, vim, or emacs** to type in a program.

_____Step 5. Modify the **store.h** file.

_____a. First plan by examining the .h files; become familiar with the data members that are available to be used. DO NOT ADD any additional data members!

_____b. Create the prototype for the "**binary_tree**" copy constructor. Remember the argument is an object of the same type, by reference.
Prototype: _______________ (const ________________);

Proficiency Scale after Step #5:

_____Step 6. In the **store.cpp** file, let's work on the default constructors listed below:

_____a. Plan and implement the binary_tree default constructor using an initialization list

- **Plan the initialization list:**
  binary_tree::binary_tree():_________________
- **Now implement the function.**

_____b. Plan and implement the BST class default constructor:

- **Plan the initialization list:**
  BST::BST():___________________
- **Now implement the function.**

_____c. Plan and implement the **store** default constructor to set up the name and address; call the read() function to set up the data to start with
Plan the initialization list: **store::store():________________**

- Why don't we need to say store::store():BST() ?___________________
- **Now implement the function.**

Proficiency Scale after Step #6:

<u>Level 2 - Intermediate</u>

_____Step 7. Time to practice again with our data structures and recursive solutions. In the **store.cpp** file, implement the function to display all of the data recursively.

- _____a. Implement the store class' display_all
- _____b. Why shouldn't this function display the parent's data?________________
- _____c. How do we get the store name displayed____________________
- _____d. How do we get the store location displayed___________________
- _____e. How do we display the binary search tree?___________________

Proficiency Scale after Step #7:

_____Step 8. Implement the store constructor with arguments.

- _____a. Is the base class default constructor sufficient for initializing the base class members in this case?________________
- _____b. Therefore, is an initialization list necessary?_________________

_____Step 9. In the **store.cpp** file, implement the **binary_tree (BST) copy constructor**

- _____a. Implement the code for the BST copy constructor.
- _____b. Double check the argument list _________________
- _____c. Is an initialization list necessary in this case?________________
- _____d. Have the copy constructor call another function that will recursively copy the tree that we implement next

_____Step 10. In the **store.cpp** file, implement the **recursive function to copy the tree** for the copy constructor to use

- _____a. **Compile (g++ *.cpp *.o -g -Wall)** and **run ./a.out**

- _____b. **Test** your code**:**

Test Case(s)	**Expected Result**	**Verified? (yes/no)**

Proficiency Scale after Step #10:

______Step 11. Implement the **binary_tree destructor** and the recursive function **delete_all**

______a. Have the destructor call the delete_all function

______b. Does the destructor reset the data members to their zero equivalent values?____________________

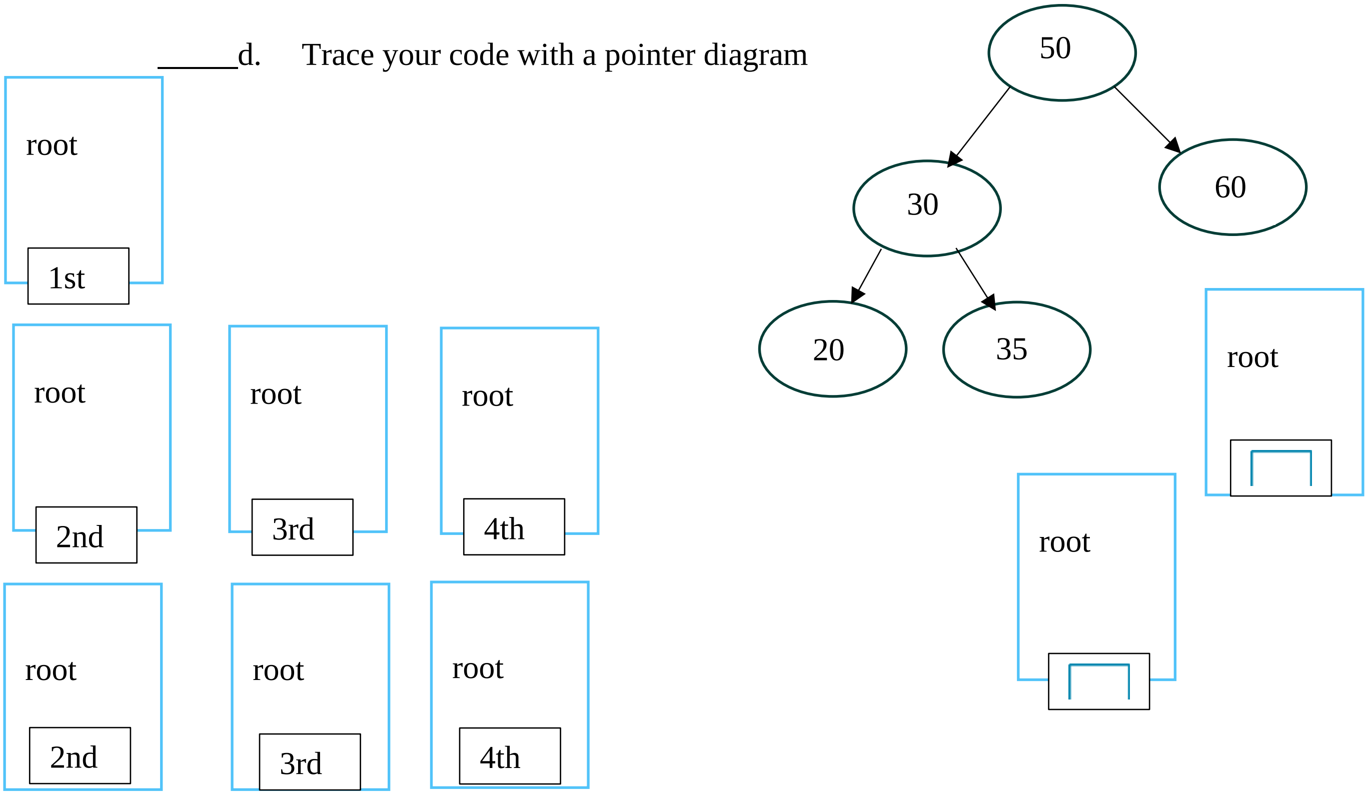
______c. For the recursive delete all, answer these questions:
What is the simple (base) case? ________________________
What is the incremental step to get to the next smaller sub-problem

____

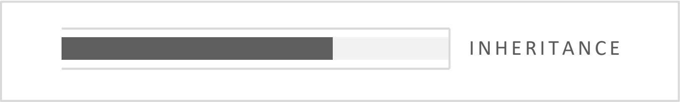
What needs to get done after returning from that smaller sub-problem?

____

______d. Trace your code with a pointer diagram



Proficiency Scale after Step #11:



Level 3 - Proficient

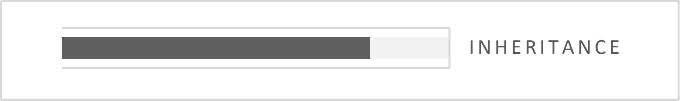
_____Step 12. ***Now that you are done implementing this code,*** examine the design and evaluate your decisions; this is an important step!

_____a. Why does a binary_tree contain a node pointer as a data member rather than use derivation?

_____b. What would happen if a binary_tree was derived from a node – would it simplify the solution or cause extra unused objects to be created?

_____c. **IMPORTANT:** Why is a node derived from an hourly_employee? We are familiar with containing relationships from CS163 (having a data member as a hourly_employee). What would have been the problem if we had used a "has a" relationship instead?

Proficiency Scale after Step #12:



_____Step 13. **Copy the cs202_tar script from Lab1** to the current working directory**:**

```
cp ../Lab1/cs202_tar .
```

_____Step 14. Tar **the files** using our script: **./cs202_tar CS202_Lab2**

Remember, this is the same as typing: tar -cf CS202_Lab2.tar *.cpp *.h

_____Step 15. **List the files** in the directory to confirm that the .tar file exists

_____Step 16. **Unpack the files** to determine if tar was used correctly; remember to do this in a different directory

_____Step 17. **Submit** your program by typing **./submit** at the linux prompt

LAB #2 Completed
ID Scanned on Completion (TCSS Initials):___________

CS202 Lab #2 – Practice Data Structures (CLL)

Work on these after finishing the lab or to practice outside of Lab

Practice CLL: The following questions should be implemented on a continual basis to prepare for your midterm proficiency demo. Start by implementing these iteratively, then implement each recursively. We suggest working in the CS202/mpdemo/CLL directory for these.

You will notice that in CS202 we will be using classes to manipulate our data structures. This means for each private recursive function there will be a public wrapper function!

After completing each of these, re-examine your approach. Do you break the CLL and turn it into a LLL? If so, re-implement each of these problems without doing so. Make sure you can **draw the pointer diagram** for each!

a. Solve each Iteratively

b. Solve each Recursively

c. Double check that each path through the function returns the correct value if you implement this with more than a single return statement in the function

d. Make sure when calling the function that the returned value is being used!

1. Write a function to **display all data except the last node's data item**. Return the sum of data items displayed.
2. Write a function to **remove the "last" node in a CLL (pointed to by rear)**. Return whether or not this was successful.
3. Write a function to **make a copy of a circular linked list**. Return the number of items copied.
4. **Implement** a function to **count all even data within a CLL** recursively. Return the count. DO NOT BREAK THE CLL into a LLL. KEEP IT A CLL and solve the problem
5. Write a function to **add a new node to the end of a CLL but only if the data doesn't already exist in the CLL**. DO THIS WITH A SINGLE TRAVERSAL!
6. Implement a recursive function to **add a node before each node** that has "**even**" data in a CLL, returning the number of nodes added
7. **Implement** a function to **remove every node that has the same data as the rear's data. Do not remove the rear's node.** Perform this recursively. Return if anything was removed (true) or not (false). DO NOT BREAK THE CLL into a LLL. KEEP IT A CLL and solve the problem

CS202 Lab #2 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

Copy Constructors and Initialization Lists

1. In which of the following situations do you need to provide an initialization list in the derived class' copy constructor: *(Put a **check mark** if it needs an initialization list)*

   a. _____The derived class has a copy constructor (no other classes have a copy constructor implemented by the programmer)

   b. _____Both the derived and base classes have copy constructors (implemented by the programmer)

   c. _____The base class has a copy constructor (no other classes have a copy constructor implemented by the programmer)

2. In what situations does a copy constructor need to be written:

3. For a list class with a node * head as a data member, write the prototype for the copy constructor:

   ___________________________(const____________________________);

Inheritance Concepts

1. If a derived class has the same named member as the base class, in what situation(s) can the base class version of the function be invoked. Explain.

2. List two situations where initialization lists are absolutely required:

______________________________   _________________________

3. Can a derived class always modify its derived class' public and protected members, regardless of the type of derivation? __________________ (**yes or no**)

<u>CS202 Lab In Summary</u>

Review the most important concepts

1. Every class has a few functions that are automatically supplied. One is the default constructor that we talked about in Lab #1. The other is a copy constructor.
2. The purpose of the automatically supplied copy constructor is to perform a copy of the data members. This is called **memberwise copy.**
3. Copy constructors are implicitly invoked when you pass a class object by value, return a class object by value, or create an object with the initial value of another object of that same class. Only the latter of these should be done.
4. **Memberwise copy** is sufficient when the data members are primitive or class types (and not pointer types).
5. However, if the data members include pointers, then memberwise copy performs a **shallow copy** operation. This will cause multiple objects to share the same memory (since the pointer values are copied). This in turn typically will cause destructors to be invoked on that same memory multiple times causing for double free errors. Something to watch out for.
6. If you have every found yourself in a situation where you needed to comment out a destructor, then double check the following:
    a. Did you pass a class object by value (by mistake)
    b. Did you return a class object by value (by mistake)
    c. Did you forget to implement a copy constructor where it was needed?
7. The solution is to make sure to implement a copy constructor whenever you have a class that manages dynamic memory. Make no exception to this rule! This is called performing a **deep copy** operation. Your goal is to implement the constructor to completely copy all of the dynamic memory as well as its contents.
8. Therefore, every class that manages dynamic memory must have a copy constructor written by the programmer.
9. If the copy constructor is being implemented for a derived class, then it is your responsibility as a programmer to provide an initialization list to cause the base class copy constructor to be invoked. If this is not done, then the base class default constructor will be invoked that sets the data members to their zero equivalent value. This concept is <u>not</u> well understood!

CS202
Programming Systems

Lab #3: Dynamic Binding

CS202 Lab #3: Goals

Important!

Make sure to keep up with the reading assignments in the Prata textbook.

When learning about OOP syntax (inheritance, initialization lists, copy constructors, dynamic binding), the best approach is to write small programs to experiment with new concepts. This is how we gain proficiency!

Make sure to keep practicing your data structures!!

Goals:

1. Start by experiencing **collaborative group activities** with dynamic binding
   a. We will be working in pairs with this lab. As such, team up with someone in this class and sit next to each other.
   b. **Spend no more than 10-15 minutes on group activities**
   c. **But continue to discuss the lab questions and code with your group as you progress through the rest of the lab**

2. **Next, work with the linux and vim manual** for no more than *10 minutes*. Learn more about using vim and linux tools

3. **Plan** to work with dynamic binding
   With dynamic binding we must use upcasting. This is where a pointer of the base class data type can point to any derived object within its hierarchy. This can be done without explicit type conversion because each derived object IS a base object plus more!

4. **Implement** dynamic binding individually to prepare for the second programming assignment.

Lab #3 contains:
Linux exercises, group activities and individual programming activities
All **must be completed** as part of this lab

CS202 Lab #3 – Background Information

Dynamic Binding

1. **Dynamic binding** is useful when we have a self-similar public interface between classes all derived from the same base class.
2. **The common base class** may or may not be the most indirect base class. But it is common to all of the derived classes that have the self-similar behavior. Think of this class as the "hub" of the hierarchy.
3. **The common base class** is where the virtual keyword is used. The virtual keyword is what tells the compiler to delay the binding of a pointer or a reference to a member function **until run time**.
4. **Dynamic binding** is where you have the:
   a. Same function names, same arguments, same return types throughout the hierarchy
   b. The base class has the keyword virtual in front of the prototype for the function(s)
   c. The function is called via a base class pointer or reference, but assigned to point to the desired derived class object
   d. Does not apply to constructors
5. **The Destructor** in the common base class must always be virtual. If it is not virtual, then the base class' destructor will be invoked even if we are working at run time with a derived class object, causing a memory leak.
6. **It is not necessary** to put the keyword virtual on derived class destructors, unless they are acting as a common base class.
7. **A pure virtual function** is one where there is no body for the member function. This is useful in situations where the function is needed in the common base class to allow for dynamic binding but where the function has no real purpose in that class. Pure virtual functions also document which functions MUST be implemented in the derived classes!
8. **For a pure virtual function,** place =0 after the prototype, before the semicolon.
   ```
   void display() = 0;
   ```
9. **Never** have a pure virtual destructor, even if there is nothing to do in the base class for the destructor, have an empty body implementation {} in the .cpp file. (Avoid the temptation to place the empty body implementation {} in the .h file.)
10. **An abstract base class** is created when you have at least one pure virtual function. With an abstract base class, no explicit objects of that class may be created.
11. **Abstract base classes** can still have data members and member functions; they exist when there is at least one pure virtual function. C++ does not use the keyword abstract.

RTTI (Run Time Type Identification) Syntax

1. Many times we need RTTI to determine what kind of data to allocate. The dynamic_cast operation allows us to determine what kind of data we are pointing at (within a particular path of hierarchy):

//Let’s find out if our base class pointer is in fact pointing to a derived object that has the
//memory expected for this particular type of derived pointer

```
Derived * ptr = dynamic_cast <Derived *> (a_base_class_pointer);


if (ptr) //if this is not NULL, allocate memory for this Derived
        Data_member = new Derived(*ptr);
```

2. If we are copying from **constant references** (an object rather than a pointer), then we will need to the const keyword; we will also need to use the “address-of” operator as part of the dynamic cast!

```
const Derived * ptr = dynamic_cast <const Derived *> (& const_derived_obj);

//or
const Derived * ptr = dynamic_cast <const Derived *> (& const_base_reference);

//then
if (ptr) //if this is not NULL, allocate memory for this Derived
        Data_member = new Derived(*ptr);
```

3. If we are working with a constant pointer or constant reference, then we must dynamic cast to a constant pointer or reference.
4. Remember, **a base class pointer can point to a base object or a derived object** because the memory is available for the base class members in both situations. We have integrity of the data.
5. **However**, a derived class pointer can ONLY point to a derived object. IT IS NOT feasible for a derived class pointer to EVER point to just a base class object because a base object does NOT have the memory for the derived members. We cannot maintain integrity of the data.

CS202 Lab #3 – Pre-lab Worksheet
Dynamic Binding

This worksheet must be completed to attend the Lab

Pre-Lab #3 (2 Pages): Fundamentals of Dynamic Binding

Assume we have a hierarchy of a base class (device), and derived classes of (pick three current gaming systems)

______1. In the base class, a virtual function has this prototype:

_____________ void play();

______2. In the derived class, a virtual function has this prototype:

_____________ void play ();

______3. In the client program, if they want to start playing through the xbox_one, show the function call – using dynamic binding:

___________________ //create a pointer to the base class

___________________ //create an object of xbox_one

_____________ play();

______4. What if a client has all three gaming systems. Create a dynamically allocated array of base class pointers that can handle playing games from any of these three:

_________________________ //Define your pointer variable

___________________ //Allocate memory for the array

_________________________//Set up the array with these three game systems

______5. Show how to call the play function for each of the 3 game systems stored in the array:

____

______6. Show the syntax of a pure virtual function:

____

______7. Show the syntax of upcasting:

____

______8. Show the syntax of downcasting:

____

Working with an Abstract base class:

______9. Explain the benefit of an abstract base class:

______10. Can an Abstract Base Class have data members?_________________

______11. Can we create objects of an Abstract Base Class?________________

______12. Do all of the methods in an Abstract Base Class need to be pure virtual?_________ if not, how many need to be pure virtual?__________

Experience OO Design:

Let's take the store management program a step further in lab #3. This time, you have salaried employees that receive a fixed amount of money regardless of the hours worked, you have hourly employees that are full time but are paid based on the hours worked and must be paid overtime if they work more than 8 hours a day, and you have part-time temporary (hourly) workers who fill in when the schedule requires additional staff (e.g., holidays) but there is no overhead (no benefits) and no overtime

______13. List the additional classes that would make sense:

______14. For each class consider what kind of functionality makes sense – **in regards to dynamic binding**. Find the similarities and differences between the classes.
Hint, data and functions that are common across the hierarchy should be placed in the base class.

Pre-lab #3 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #3 – Dynamic Binding

Check-off each step as you proceed! Submit the code at the end of lab.

Goal:

Gain experience with the OOP design and dynamic binding concepts.

Collaboration

- We will be working in pairs with this lab. As such, team up with someone in this class and sit next to each other. Although you will be writing your own code, individually, you will be evaluating each other's code
- Each lab question is designed to be done in a step by step fashion. The success of later steps depends on earlier.
- **At the end of the first page, have your answers checked by a Lab Assistant**

Designing with Dynamic Binding

Level 1 - Introductory

_____Step 1. List 3 functions that should be virtual

_____Step 2. List 3 functions that don't apply to each class (that are unique). Is there a way a general purpose function could be written that could apply across the hierarchy instead? Or, will RTTI be the only solution for clients to access these functions?

Level 2 - Intermediate

_____Step 3. Would this be a good situation for an abstract base class? Why or why not?

****** RECEIVE feedback from Lab assistants before continuing with the linux & vim exercises***

Observing Dynamic Binding

Level 1 - Introductory

_____Step 4. Login to **cs202lab.cs.pdx.edu** using your assigned login and password

_____a. Change into the CS202/Lab3 directory **cd CS202/Lab3**

_____b. Use a **linux editor** such as **vi, vim, or emacs** to type in a program.

_____Step 5. **Plan:** Examine the main program in **CS202_lab3.cpp** and answer these questions:

_____a. What is the data type of the all_employees array:_________

_____b. Show how we store different kinds of employees into this array:

____

_____c. When we say: all_employees[i]->display(), which display function is called?

____

_____Step 6. Compile and Run the program, entering full time, hourly, and salaried employees. Answer these questions:

_____a. Are the correct display functions called?
(look at the beneficiary, insurance, pay)

Which display functions are called for Full_time employees:_____

Which display functions are called for Hourly employees:_______

_____b. Which destructors are invoked for each of the following type of objects

Which destructors are called for Full_time employees:________
Which destructors are called for Hourly employees:__________

Which destructors are called for Salaried employees:_________

_____c. Look at the .h files and explain why each class doesn't have its own destructor:

Proficiency Scale after Step #6:

****** RECEIVE feedback from Lab assistants before continuing with the next steps***

Individual Coding of Dynamic Binding

Now let’s experience coding dynamically bound functions. Let’s build a manager class, derived from the salaried employee class. A manager should have additional information about (a) the name of the group that they manage (e.g., “Engineering”), (b) the number of employees in their group and (c) an array of employees. In this code, you will experience the process of using RTTI as well.

RTTI Background:

An employee pointer can point to any derived class object (called upcasting). This allows us to be able to call the right function within the hierarchy dynamically at run time (called dynamic binding). However, constructors cannot be dynamically bound. Therefore, we must know what type to create to use new. Therefore, to implement the add_to_group function or copy constructors function, we must experience the dynamic_cast operations to let us know what type of data we are pointing at, at run time.

Many times, we need RTTI to determine what kind of data to allocate. The dynamic_cast operation allows us to determine what kind of data we are pointing at (within a particular path of hierarchy):

```
Derived * ptr = dynamic_cast <Derived *> (a_base_class_pointer);

//Now find out if we were actually pointing to this type of object
if (ptr) //if this is not NULL, allocate memory for this Derived
        Data_member = new Derived(*ptr);
```

If we are copying from **constant references**, then we will need to the const keyword:

```
const Derived * ptr =
        dynamic_cast <const Derived *> (& const_derived_obj); //using a base class reference
//or
const Derived * ptr = dynamic_cast <const Derived *> (& const_base_reference);

//then
if (ptr) //if this is not NULL, allocate memory for this Derived
        Data_member = new Derived(*ptr);
```

If we are working with a constant pointer or constant reference, then we must dynamic cast to a constant pointer or reference!

Experiencing Dynamic Binding and RTTI with Coding Exercises:

_____Step 7. **Plan:** Examine the manager.h file, where a manager is derived from a salaried employee.

- _____a. What are the data members:________________
- _____b. What should their initial value be?___________
- _____c. Should initialization lists be used when we implement the constructors and if so, for what?

  ____
- _____d. Brainstorm on where RTTI will be needed:

  ____

Proficiency Scale after Step #7:

Level 2 - Intermediate

_____Step 8. Implement the following in the **manager.cpp** file:

_____a. Complete the implementation of the manager constructor **that takes a salaried employee in as an argument**; this argument represents the information about the manager who is going to lead this new group that is being formed. Please note that this is not the copy constructor. This constructor might be used in cases where an employee is being promoted to become a manager!

manager(const salaried_employee & new_managers_info);

- ______Don’t forget the initialization list. Without it the incorrect base class constructor will be invoked!
- Will we need to use RTTI in this function?_________________

_____b. Implement the manager destructor

- What are the data member values afterward? ________________
  *** NEVER leave them as garbage!
- Since there is an array of employees, how does each employee destructor get invoked?________________________________

Proficiency Scale after Step #8:

_____Step 9. **Use dynamic binding** to experience the binding happening at run-time.

_____a. Modify main to create a manager object and use the constructor that was just created in Step 8. This means that you will need to first create a salaried_employee object and pass it in as an argument constructor

_____b. **Compile (g++ *.cpp *.o -g -Wall) and Run**

_____c. **Fix all warning** received from -Wall

_____d. **Discuss** your findings. Did the "right" functions expected get called?

____

Proficiency Scale after Step #9:

<u>Level 3 - Proficient</u>

_____Step 10. **RTTI** – Complete the implementation of the **add_to_group** function

To complete the implementation of the **add_to_group** function, we are faced with a problem. We are given a base class reference as an argument (**const employee &**). But, we don't know what derived object it is that we are referencing at run time. This is because when we use dynamic binding, we perform upcasting. This means that we have a base class referring to SOME derived object – but we don't know which one. We will need to know which object type we are referring to in order to allocate a new object and add it to the group. This requires the use of **downcasting**!

_____a. Prepare by planning to use the **dynamic_cast** to perform **downcasting. A downcast operation is needed to type convert** a base class reference type to the correct derived class reference. Keep in mind that we are working with a **constant reference!**

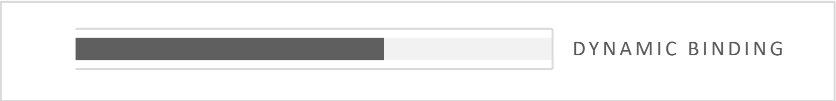
Plan:

const ________* ptr = dynamic_cast<const ______ *> (**&**______);

(Note the above **&** is the "**address of**" operator)

_____b. Complete the implementation of the add_to_group function.

Proficiency Scale after Step #10:



_____Step 11. **THIS STEP IS REQUIRED! (Located in Step11_RTTI.cpp)**
RTTI Apply what you have learned to create the copy constructor for the manager class; this will be a useful exercise to prepare for the programming assignment! *Please keep in mind that this is not the same constructor as was implemented previously! This time we are copying an entire manager's group and creating a new copy!*

manager(const manager & to_copy);

_____a. Double check that the prototype is in the manager.h file

_____b. Add an initialization list:

manager(const manager & to_copy):_____________________

_____c. What should the initialization list need to do?______________

_____d. What would happen if we didn't have an initialization list?

____

_____e. Create an array of employee pointers, dynamically

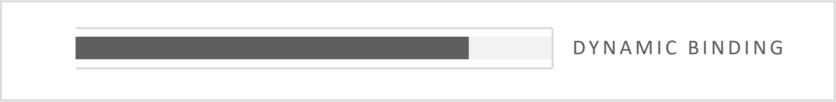
________________ = new employee ________;

_____f. Perform a deep copy (be cautious not to do a shallow copy).

_____g. Write an example of how you will use the dynamic cast operation:

___________ = dynamic_cast<______ > (____);

Proficiency Scale after Step #11:



_____Step 12. **Copy the cs202_tar script from Lab1** to the current working directory**:**
cp ../Lab1/cs202_tar .

_____Step 13. Tar **the files** using our script: **./cs202_tar CS202_Lab3**
Remember, this is the same as typing: tar -cf CS202_Lab3.tar *.cpp *.h

_____Step 14. **Submit** your program by typing **./submit** at the linux prompt

LAB #3 Completed
ID Scanned on Completion (TCSS Initials):___________

CS202 Lab #3 – Practice Data Structures (DLL)

Work on these after finishing the lab or to practice outside of Lab

Practice DLL: The following questions should be implemented on a continual basis to prepare for your midterm proficiency demo. Start by implementing these iteratively, then implement each recursively. We suggest working in the CS202/mpdemo/DLL directory for these.

You will notice that in CS202 we will be using classes to manipulate our data structures. This means for each private recursive function there will be a public wrapper function! When working with recursion, we first need to decide what part of the problem is repetitive in nature and which part isn't. A "wrapper" function is required for the non-repetitive process which in turn should call the "recursive" function that solves the repetitive part. Planning is the best first step.

After completing each of these, re-examine your approach. Can you traverse the DLL forwards and backwards without seg faults? If not, re-implement the solution. Make sure you can **draw the pointer diagram** for each!

a. Solve each Iteratively
b. Solve each Recursively
c. Double check that each path through the function returns the correct value if you implement this with more than a single return statement in the function
d. Make sure when calling the function that the returned value is being used!

1. Write a function to **remove every item larger than the first item**. Return the number of items displayed
2. Write a function to **remove every other item from a doubly linked list and return the number of items removed**
3. Write a function to **duplicate every item that is a '2' in the list.** Return the number of items duplicated. Place the new nodes immediately after the previous '2'.
4. Implement a recursive function to **Display the contents of the DLL in reverse;** return the last node's data item (the first to be displayed) and display that again from main.
5. Write a function to **add a new node to the end of a DLL but only if the data doesn't already exist in the DLL**. Consider efficiency and stop traversal early if possible.
6. **Make a complete and duplicate copy of a doubly linked list**; return if the new list has any data in it (true) or not (false)
7. **Add a node to the end of the list that contains the average of the list. Perform this with a single traversal. Return the average to main.**

CS202 Lab #3 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

Dynamic Binding

1. List two forms of binding: ________________ ________________
2. When we want to call the right function within a hierarchy based on where we are pointing or referring at run time, we are using ___________________
3. To do this, we must place the ____________________ keyword in front of prototype in the _______________ class
4. Do constructors also need this keyword?________________
5. If the base class has no dynamic memory, do the destructors also need this keyword?_______________________
6. A ________ ____________ member function means that it has no body
7. If we redefine a function that exists in the base class, and place it also in a derived class, *(discuss your answer for both kinds of binding)*

    a. What happens if the argument list is different?

    b. What happens if the argument list is the same but the return type is different?

    c. What happens if the argument list is the same and the return type is the same?

8. **Show** the syntax for downcasting:

    ____

9. **Show** the syntax for upcasting:

    ____

Assume that you have the following hierarchy:

a. **Employee, Customer and Manager are all derived from Person**
b. **Person has an address object as a data member.**

10. Why shouldn't Person be an abstract base class?

11. Write the prototype for the display function in the Person class

_____________________ display (_________________) _____________ ;

12. Show how to call the display function, demonstrating dynamic binding: (Show all variable definitions

____

13. List the rules of dynamic binding:

____

____

____

____

CS202 Lab In Summary

Review the most important concepts

1. Dynamic binding allows the desired function to be invoked throughout the hierarchy given just one function call.
2. Dynamic binding supports one of the purest forms of polymorphism as it allows the same function from the surface to exist all through the hierarchy but the implementation details can be unique.
3. Dynamic binding requires that the functions dynamically bound have the same name, same return type, and same argument list. The functions dynamically bound must be declared as **virtual** in the base class. They must be called through a base class pointer or reference. Through the concept of upcasting, we can have that base class pointer or reference refer to the desired type of object at run time.
4. The base class chosen for dynamic binding can be any class in the hierarchy that makes the most sense. It can be an indirect base class or one that is a "hub" for the rest of the hierarchy.
5. Make sure that the base class' destructor is also made **virtual** in the event that dynamic binding is being used for this class.
6. Avoid over use of the keyword **virtual.** It should only be used in classes where we are specifically going to use (and test) dynamic binding.
7. Only functions can be dynamically bound, not data members.
8. A pure virtual function is one where the specified function has not implementation in the base class. This automatically makes the base class an **abstract** base class. An abstract class is one where base class types cannot be instantiated explicitly. This means that you can't create an object of that base class.
9. It is <u>not </u>necessary to have an abstract class to perform dynamic binding.
10. In fact, the only time we want an abstract class is if (a) some of the virtual methods in the base class make no sense and do not need to be written and (b) you never need to make an explicit object of that base class.
11. An abstract base class can have data members, implemented member functions, and be derived from another class.
12. **Upcasting** is the concept where a base class pointer or reference can point to any class derived. This does not require any explicit type conversion syntax because every object within hierarchy is a base object plus more.
13. **Downcasting** is the concept where we have a base class pointer or reference. It is pointing somewhere in the hierarchy. But, we need to determine if we are pointing to a specific derived type. Downcasting can be used to determine this and it requires the use of RTTI and explicit type conversion.
14. **Avoid** creating derived classes with methods that are different (in name, argument list, or return type) than the virtual versions of those function in the base class.

Programming Systems

Lab #4: Review Data Structures and Recursion

CS202 Lab #4: Goals

Important!

It is important to become confident with each of the data structures. Practice daily the following data structures implementing recursive algorithms:

- Linear Linked Lists
- Circular Linked Lists
- Doubly Linked Lists
- Arrays of Linear Linked Lists

Goals:

1. **Today start with the linux and vim manual** for the first *10 minutes*. Learn more about using vim and linux tools

2. **Plan** to practice implementing recursive data structure algorithms
   a. Gain practice with Arrays of LLL

3. **Implement** recursive algorithms on the cs202 lab system. The data structure we will be implementing is a **doubly linked list.** This means that each node points back to the previous node as well as to the next node. This means there are two pointers in each node. Most doubly linked lists have both a head and a tail pointer. All coding should be performed individually.

4. Apply **test plans** and **pointer diagrams** to verify your software. **When working with test plans**, think about all the cases that need to be verified.

Make sure to Practice Daily
and
Seek Assistance!

CS202 Lab #4 – Background Information

Recursion Review

For each of recursive problem, think about the special cases that would help us move from an iterative solution to a recursive solution. Remember recursion is all about breaking the problem down into smaller sub-problems and re-calling the function over and over with a different piece of the problem.

Remember the 3 rules of recursion:

1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

When solving a problem recursively, ask yourself these questions:

1. Is the problem repetitive in nature?
2. What is the base case?
3. Are there other stopping conditions not considered?
4. What needs to be done in each of these cases. Outline them specifically.
5. Does anything need to be done prior to the recursive call?
6. Does anything need to be done after the recursive call?
7. If information is being returned, are we returning the correct data?
8. Are there paths through the algorithm where nothing is being returned?
9. When data is being returned, are we using that data when it comes back after calling the function?

CS202 Lab #4 - Pre-lab Worksheet
Recursion

This worksheet must be brought to the lab to attend.

Pre-lab #4 (3 Pages)

For each of the problems, think about the special cases that would help us move from an iterative solution to a recursive solution. Remember recursion is all about breaking the problem down into smaller sub-problems and re-calling the function over and over with a different piece of the problem.

Concepts of Recursion

For these questions, do some research:

_____Step 1. What is meant by “tail recursion”:

_____Step 2. Explain the concept of “back tracing”:

Understanding Recursion

Assume you have a function that sums together all of the data in a linear linked list. It takes a pointer to a node as an argument and returns an integer (of the total). Answer the following questions:

_____Step 3. What is the base case (the most trivial case):_________________________

_____Step 4. Do you need a local variable to hold the total?________

_____Step 5. Show the recursive function call:

_____________________ = sum (head->next) + ___________________;

_____Step 6. How does the total get supplied back to the calling routine?

Problem: Assume you have a linear linked list of single(unsorted) characters. What is the simple case: *(what case does not require repetition?)*

_____Step 7. Design the algorithm to determine if **every character is unique** in the list, recursively.

a. What is the base case:

b. What is the incremental step that transforms the problem into a smaller sub-problem?

c. Is there anything that needs to be done BEFORE going to the smaller sub-problem:

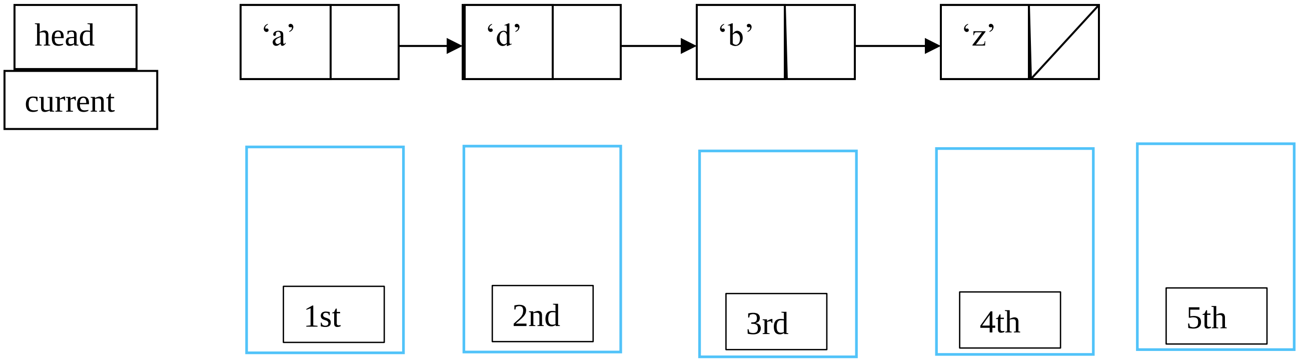
If so, what is it:______________________________________

d. Create your own test plan for this:

Test Case(s)	**Expected Result**	**Verified? (yes/no)**

e. Show your algorithm:

f. Work through your plan using a pointer diagram outlining each invocation:



Problem:

_____Step 8. Design the algorithm to **determine if the last item is the largest item in a linear linked list recursively**

a. Is a wrapper function required? If so, how will it be used?

b. What is the base case:

c. What is the incremental step that transforms the problem into a smaller sub-problem?

d. Is there anything that needs to be done BEFORE going to the smaller sub-problem:

If so, what is it:________________________________

e. Is there anything that needs to be done AFTER, upon return?

If so, what is it:________________________________

f. Show your algorithm and/or draw the pointer diagram:

_____Step 9. Prepare for vim exercises:

_____a. To insert where the cursor is, type: ______________

_____b. To end insertion mode, type: _____________

_____c. To go to the end of a line, type: _____________

_____d. To go down 10 lines from your current position type: __________

_____e. To go back up to the beginning of a file, type: ___________

_____f. To go to the end of a file, type: _____________

_____g. To delete the current line, type: _____________

_____h. To undo the last change, type:______________

_____i. Show how to search for the word NULL:_________

_____j. Show how to replace the next occurrence of NULL with a zero:___________

Prelab #4 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #4 – Data Structures & Recursion

Check-off each step as you proceed! Submit the code at the end of lab.

Coding: With this lab we will be working with existing linear, circular and doubly linked lists of integers. Your job will be to implement functions to experience manipulating the lists using **recursion!**

There are separate directories for each of these data structures:

- CS202/Lab4/LLL – linear linked lists
- CS202/Lab4/CLL – circular linked lists
- CS202/Lab4/DLL – doubly linked lists
- CS202/Lab4/ARR – array of linear linked lists

When using recursion, always think about what is the **repetitive** part of the problem versus the **non-repetitive**. Recursion is about repetition. Consider non-repetitive functions, such as **wrapper functions**, to handle operations that only need to be done once.

When calling functions, pay very close attention to the **returned value** coming back from the function call. Make sure to **ALWAYS** take care to handle this information upon the return. And of course, if there is a non-void returning function, make sure to return the correct information through all paths in the function. If you are using multiple returns throughout the function, also use care. Common mistakes come from returning sometimes but not always, or returning early before completing all necessary operations.

The following questions start simple and then increase in difficulty. Get through as many as possible, solving each recursively. All will need public wrapper functions with private recursive functions. Additional functions may be written in addition to this.

As you write these functions, keep in mind the following concepts:

1. Minimize traversal. For example, to computing an average should be done within one traversal. Please do not count the number of items just to re-traverse to calculate a sum!
2. Avoid solving the problems through the use of globals or static locals. Get used to using arguments.
3. Be aware that locals will be re-created for each recursive call.
4. Never have two returns in a row!!!!! *Think about this!*

Level 1 - Introductory

______Step 1. **Count the number of times the requested data (sent in as an argument) is in a linear linked list; return the number of items found: (cd CS202/Lab4/LLL)**

Public Prototype: **int num_times(int match);**
Private Prototype: **int num_times(node * head, int match);**

______a. **Plan** the algorithm using these questions:

i. What is the simple (base) case?

____

ii. What is the incremental step to get to the next smaller sub-problem?

____

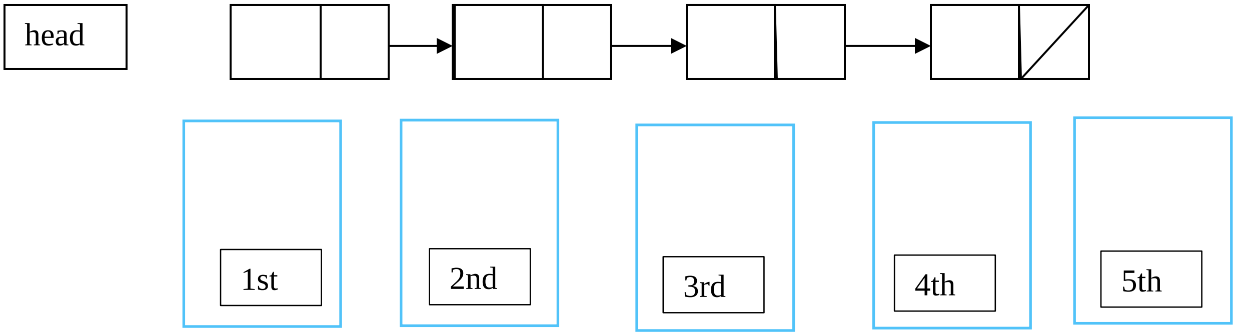
iii. How do we make sure the calling routine gets the count back?

____

iv. At what point are you going to add 1?____________________________
Is a local variable needed?_____________

v. Show the recursive call: ____

______b. **Place** the prototype in the list.h file
______c. **Call** the function from main
______d. In main, did you use the returned value?__________
______e. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**
______f. **Before continuing,** step through your code with the pointer diagram, tracing every line of code showing how the code influences on the list:



Proficiency Scale after Step #1:

______Step 2. **Display <u>everything except</u> the first and the last item in the linear linked list; return the sum of the first and last data items to the main**

Public Prototype: **int display_except();**
Private Prototype: **int display_except(node * head);**

______a. **Plan** the algorithm using these questions:

i. What is the non-repetitive part of this problem?________________
ii. What is the repetitive part of this problem?____________________
iii. How is a wrapper function going to be used?___________________
iv. What is the simple (base) case for the recursive function?

____

v. What is the incremental step to get to the next smaller sub-problem?

____

vi. How do we make sure the calling routine gets the sum back?

____

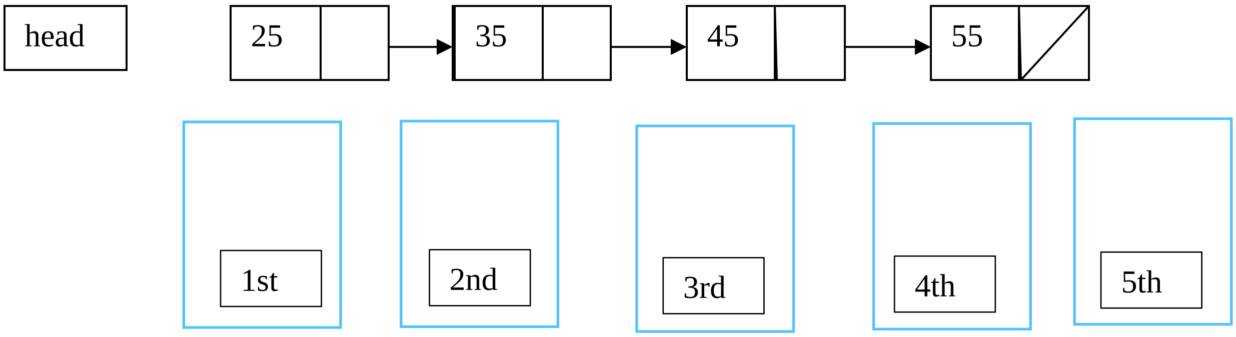
vii. At what point are you going to add the data together?_________ Is a local variable needed?_____________
viii. Show the recursive call: ________________________________

______b. **Call** the function from main

______c. In main, did you use the returned value?__________

______d. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**

______e. **Before continuing,** step through your code with the pointer diagram, tracing every line of code showing how the code influences on the list:



Proficiency Scale after Step #2:

______Step 3. **Remove all but the last two items in a linear linked list;** return the number of items removed.

Public Prototype: **int remove_except();**
Private Prototype: **int remove_except(node * & head);**

______a. **Plan** the algorithm using these questions:

i. How is a wrapper function going to be used?___________________
ii. What is the simple (base) case? _____________________________
iii. Should the removal happen before the recursive call or after?________
iv. Will we need a local variable?______ For what?___________
v. Be specific in designing the removal process:

____

vi. How do we make sure that the correct value gets returned?

____

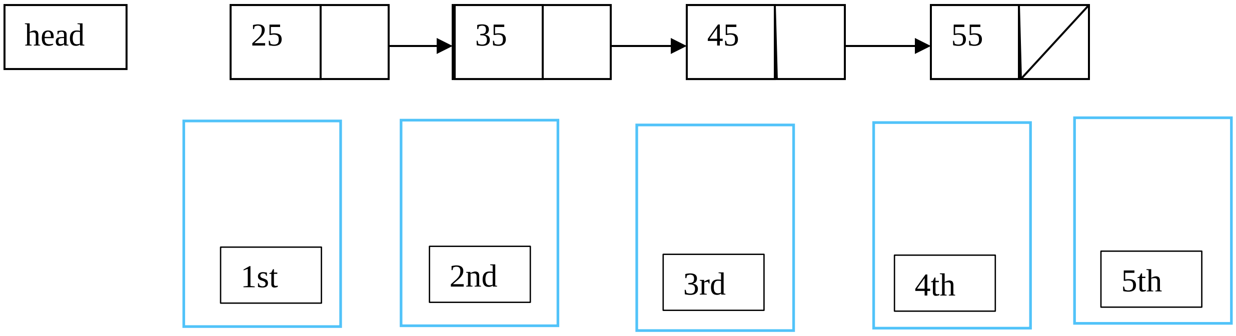
vii. Show the recursive call: ________________________________
viii. Write the **algorithm** for the remove:

______b. **Now, implement the function** and then **call** the function from main

______c. In main, did you use the returned value?_________

______d. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**

______e. **Before continuing,** step through your code with the pointer diagram, tracing every line of code showing how the code influences on the list:



Proficiency Scale after Step #3:

Level 2 - Intermediate

______Step 4. **Make a complete and duplicate copy of a circular linked list creating a new CLL; exclude any multiples of 2 (cd CS202/Lab4/CLL)**; return the number of items copied

Public Prototype: **int copy_special(list & new_list);**

Private Prototype: **int copy_special(node * & new_copy,**
node * original);

______a. **Plan** the algorithm using these questions:

i. How is a wrapper function going to be used?____________________

ii. Show the function call from the wrapper to the recursive function:

____

iii. For the recursive function, what is the simple (base) case?

____

iv. Should the copy happen before the recursive call or after?________

v. Will we need a local variable?_______ For what?____________

vi. Show the recursive call if the data is a multiple of 2?:

____

vii. Show the recursive call if the data is NOT a multiple of 2?:

____

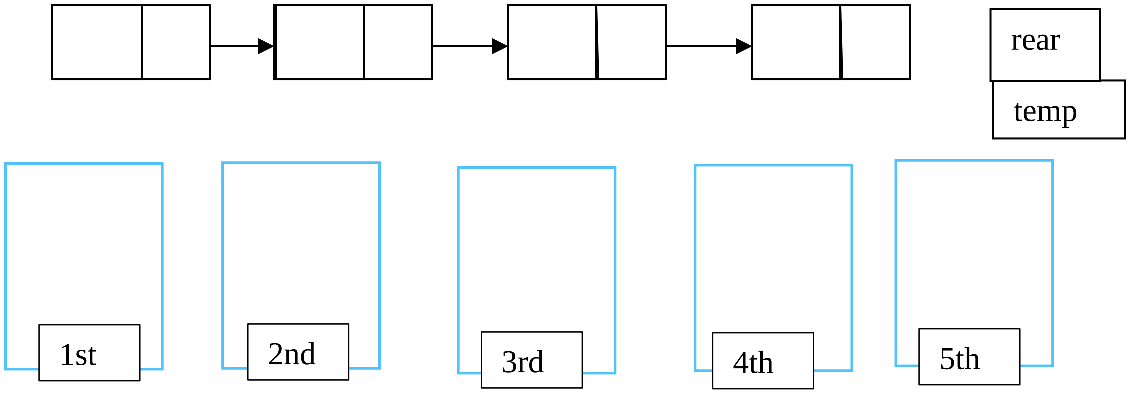
______b. **Now, implement the function** and then **call** the function from main

______c. In main, did you use the returned value?__________

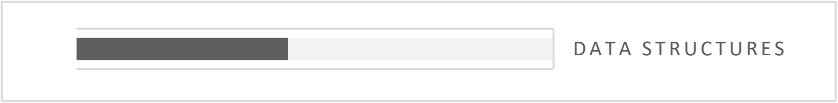
______d. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**

______e. **Before continuing,** step through your code with the pointer diagram, tracing every line of code showing how the code influences on the list:

- Remember the special case where there is only one node.



Proficiency Scale after Step #4:



______Step 5. **Make a complete and duplicate copy of a doubly linked list** excluding the first and the last item; return the number of nodes in the new list **(cd CS202/Lab4/DLL)**

Public Prototype: **int copy_DLL(list & new_list);**

Private Prototype: **int copy_DLL(node * & new_copy,**
node * original);

______a. **Plan** the algorithm using these questions:

i. What is the non-repetitive part of this problem?________________

ii. What is the repetitive part of this problem?____________________

iii. How is a wrapper function going to be used?___________________

iv. Show the function call from the wrapper to the recursive function:

____

v. For the recursive function, what is the simple (base) case?

____

vi. Should the copy happen before the recursive call or after?________

vii. Will we need a local variable?_______ For what?____________

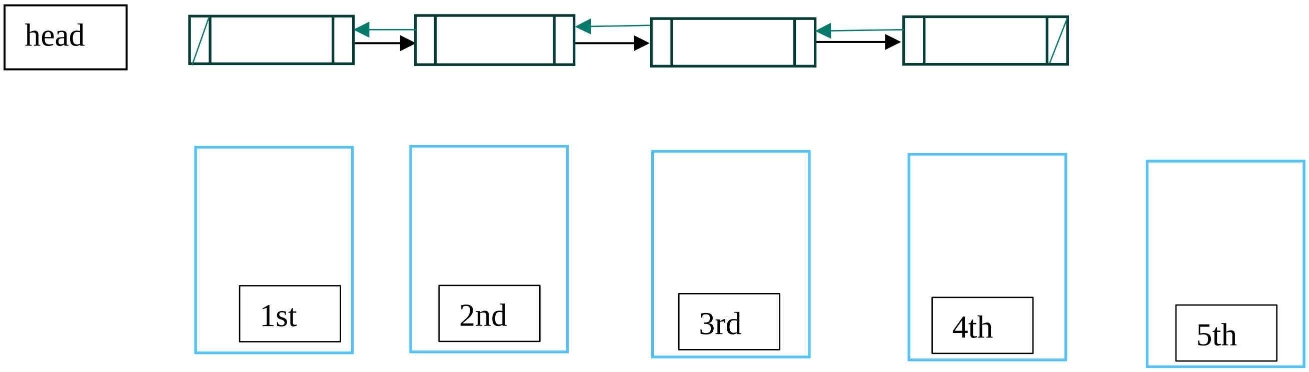
viii. Plan the algorithm for how you will connect all of the nodes (for the DLL):

______b. **Now, implement the function** and then **call** the function from main

______c. In main, did you use the returned value?__________

______d. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**

______e. **Before continuing,** step through your code with the pointer diagram, tracing every line of code showing how the code influences on the list:



Proficiency Scale after Step #5:

_____Step 6. Write a recursive function to determine if two linear linked lists contain the same values – only traverse as far as necessary (ie., do not traverse through past the length of the smaller of the two!). **(cd CS202/Lab4/LLL)**

Public Prototype: **bool same_contents(list & second_list);**
Private Prototype: **bool same_contents(node * head1,**
node * head2);

_____a. **Plan** the algorithm using these questions:

i. Think about how you would use a loop for this problem and describe what you would use the loop for *(be specific):*

____

ii. Imagine how this loop could be replaced with a recursive call. What would be the base case or stopping condition? What needs to get done before going to that next smaller sub-problem?

____

iii. How is a wrapper function going to be used?__________________

iv. Show the function call from the wrapper to the recursive function:

____

v. For the recursive function, what is the simple (base) case?

____

vi. How do we make sure that the correct value gets returned?

____

vii. What would be the difference in efficiency if we compared the data before the recursive call versus after?

____

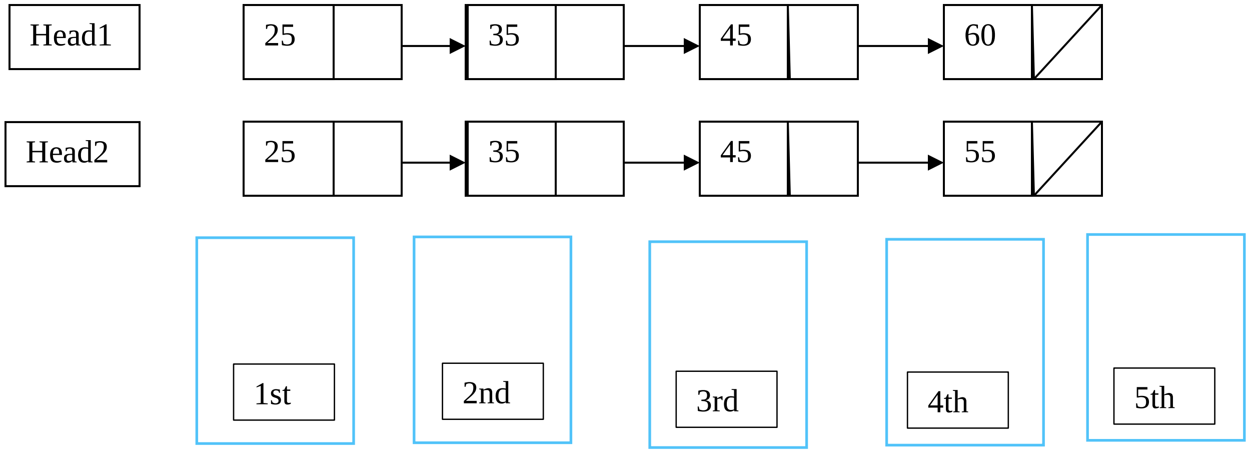
_____b. **Now, implement the function** and then **call** the function from main

_____c. In main, did you use the returned value?_________

_____d. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**

Proficiency Scale after Step #6:

______e. **Before continuing,** step through your plans with the pointer diagram



Level 3 - Proficient

______Step 7. Write a recursive function to display all data in an array of linear linked lists and return the number of items displayed. Use pointer arithmetic to process through the array. DO NOT use any loops!!!!

(cd CS202/Lab4/ARR)

Public Prototype: **int display_ARR();**

Private Prototype: **int display_ARR(node * head);**

______a. **Plan** the algorithm using these questions:

i. Think about how you would use a loop for this problem to process through the array:____

ii. Remind yourself of pointer arithmetic. Show how to process through an array using pointers:

____

iii. **Now replace** the loop with a recursive call. Show the call that will process through the array:

____

iv. **Next**, show the recursive call for processing through the linear linked list:

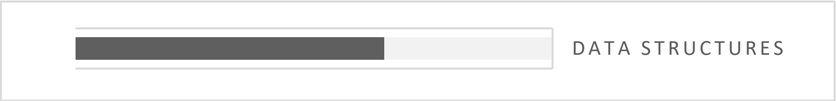
____

______b. **Now, implement the function** and then **call** the function from main

______c. In main, did you use the returned value?__________

______d. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**

Proficiency Scale after Step #7:



_____Step 8. Write a recursive function to remove the first and last node in the last linear linked list in an array of linear linked lists. Use pointer arithmetic to process through the array. Return the sum of the data removed. DO NOT use any loops!!!! **(cd CS202/Lab4/ARR)**
Public Prototype: **int remove_first_last();**
Private Prototype: **int remove_first_last(node * & head);**

_____a. **Plan** the algorithm using these questions:

i. Show how to access the last array element using pointer arithmetic: ____

ii. **How many functions** will you need to solve this problem?____

iii. **What will be the purpose** of each of the functions: ____

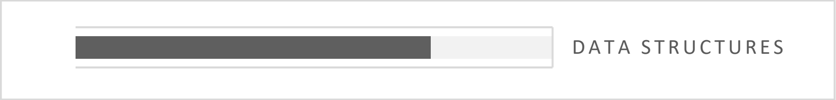
iv. **Next**, show the recursive call for processing through the linear linked list:_______________________________________

_____b. **Now, implement the function** and then **call** the function from main

_____c. In main, did you use the returned value?_________

_____d. **Compile (g++ *.cpp *.o -g -Wall)** and **run (./a.out)**

Proficiency Scale after Step #8:



_____Step 9. **Prepare for vim Testing**
You will be asked about navigation and search/replace functionality with vim for the midterm and certainly the final proficiency demos. Make sure you know how to perform the following:

_____a. Show how to navigate down 10 lines in one command?_____

_____b. Show how to search forward for some text? __________________

_____c. How do you search again for the same thing? _________________

_____d. How do you search again for the previous match?______________

_____e. How do you search backwards for a match instead of forwards?___

_____f. How do you undo the last change to your code?_______________

_____g. How do you search and replace for some text?________________

_____Step 10. Make sure you are in the CS202/Lab4 directory (**cd CS202/Lab4**)

_____Step 11. Tar **the files and directories**: tar -cf CS202_Lab4.tar *

_____Step 12. **Submit** your program by typing **./submit** at the linux prompt

LAB #4 Completed
ID Scanned on Completion (TCSS Initials):___________

CS202 Lab #4 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

1. When working with an **array of linear linked lists**, answer the following questions:

   a. The data type of each element of the array is ________________

   b. To create a dynamically allocated array of LLL, the list class' data member would need to be what data type? _________ array;

   c. Create an array that is dynamically allocated of size M:

   array = new _________________________ ;

   d. Show how to allocate the first node for index I:

   array[I]______ = new ___________ ;

   e. Rewrite the above using pointer arithmetic

   ______array_______ = new ___________ ;

   f. If the data in the node was a char * title and an int length, show how you could initialize the length:

   array[I]_____length = 120;

   g. Rewrite the above using pointer arithmetic

   ______array_______ = 120;

   h. Using pointer arithmetic, show how to deallocate an array of LLL:

Programming Systems

Midterm Proficiency Demo

CS202 Midterm Proficiency Demos

Preparing for Midterm Proficiency Demo

The midterm proficiency demo during most terms is on **recursive solutions to linear, circular, doubly linked list and array of LLL problems**. All of these are tested within the class construct. The best way to prepare is to program daily; try to set aside some time each day just to practice coding your data structures using recursion. Start small, and first think about what IS the recursive part versus what is NOT. Consider using a wrapper function to perform operations that are not done repetitively (such as computing the average). Never forget the base case, even if you think it has already been handled in the wrapper. Practice the material from each of the labs (in the data structure practice section). For practice, you can work on the cs202 system (e.g., the CS202/mpdemo directory). In that directory, compile via: g++ *.cpp *.o -Wall

We recommend that you consider the questions that you have been working through in this manual as you program:

1. Plan first. Draw pointer diagrams!
2. Make sure you understand how to work with pointer diagrams.
3. Determine how best to support recursion. Determine what is the non-repetitive portions for the wrapper function and how additional functions can help.
4. Think about how you will traverse the data structure. It is important to avoid multiple traversals through the data structure. Functions that traverse should not modify the data structure unless it is required by the assigned question. Practice the following on your own as you prepare. In CS202, the solutions may not use loops.
5. For each recursive call, are you using the returned value coming back?
6. For each recursive return, are you returning the right information?
7. If you like to program with many returns in a function, is there a return in each path appropriately?
8. Are you compiling with -Wall and are all of the warnings fixed?

The midterm proficiency demo will be taking place on a system much like what we use for the labs. The process begins by coming to a pre-scheduled appointment; we recommend arriving 10 minutes early. After confirming your appointment and showing picture ID, you will be given a randomly selected question. If you don't like it, you can give it back and select another. You can only do this process twice (the third question "sticks"). There is no going back to a previous question. It is a closed book/closed notes/closed internet event.

Please keep in mind that the midterm proficiency demos are only by appointment.

Practice Questions

NONE of your functions should have void return types!

1. Implement the following with a LLL, CLL and DLL data structures:
    a. Count the number of items in the list that are the same as the first node and return the count.
    b. Count the number of items in the list that are the same as the last node and return the count.
    c. Count the number of items in the list that are greater than ‘4’ and return the count.
    d. Display all of the unique data that exists in the data structure and return the sum of all unique items.
    e. Remove the first node and place it at the end of the list; return the number of items in the list.
    f. Remove the last ‘2’ and return the total number of 2’s in the list. Do this is only a single traversal.
    g. Remove all nodes after the last ‘2’ and return the number of nodes removed. Do this in a single traversal.
    h. Remove every ‘2’ except the first one. Return the number of 2’s removed. Do this in a single traversal.
    i. Remove the last two nodes, but only if the last node is not a ‘2’. Return the sum of these nodes.
    j. Make a copy of the data structure, but only copy even data. Return the number of nodes copied. Do not implement this with an insert function! Traverse only once.
    k. Create an average of all of the even data in the data structure
    l. Sum all of the data less than 10 in the data structure
    m. Make a copy of all even data items
    n. Make a copy of all unique data

2. Implement the following with an Array of LLL; do not use loops and use pointer arithmetic to access the elements of the array
    a. Count the number of items in a LLL and return the count
    b. Display the last item in each LLL and return the sum of each
    c. Display all of the data in the last LLL and return the largest item
    d. Compare the first node in each LLL and return true if they are the same
    e. Remove the first node in each LLL and return the sum of each
    f. Remove the last node in each LLL and return the number of nodes removed

Midterm Proficiency Process

In your midterm proficiency demo, you will be asked to implement a recursive solution for a LLL, CLL, DLL to insert, remove, find, or otherwise modify a node or the data in the data structure. Make sure to practice with the code supplied in the midterm proficiency demo directory (**CS202/mpdemo**) of the cs202lab system. Practice the following on your own as you prepare. Because it is closed notes, closed book, closed internet, you may not access any electronic devices (e.g., phones or smart watches) besides the computer being used. And, when working on your computer, only one cs202lab session may be open and no other programs may be running in the background. *Logging into another cs202lab or linux session or using help/man during the proficiency demo will result in an automatic failure.*

Important for the Proficiency Demo Process

1. Plan to come 10 minutes early to get set up and accustomed to the environment.
2. Upon arrival make sure all electronic devices, including your computer, are fully shut down and turned off. Smart watches and phones should be safely stored away.
3. The technical assistants will check you in. First show them your picture id
4. You will be asked to read a document that describes in detail what is expected during the demonstration. We ask you to sign this **contract** indicating that you understand. We would be happy to clarify or explain; just ask! It is important to read this to familiarize yourself with the requirements. It will contain information described in this section.
5. When space becomes available in the testing arena, the next step will be to select a question, randomly.
   a. Technical assistants will provide you with a set of questions to select from.
   b. Select a question and read it. If you are happy with the question, then you will proceed to the next step. If not, give it back to the technical assistant and select again. Again, if you are happy with this second question, you can proceed. If not, give it back to the technical assistant and select again.
   c. The third question "sticks" and you can't go back to a previous question (there are no "go backs").
   d. *Questions are refreshed periodically.*

   e. Once a question has been selected, it is considered a "closed book, closed notes" session and no further internet access is allowed.
   f. Once you have started working on a question, it cannot be replaced

6. You may use your own computer or check out one of ours.
   a. All applications must be closed prior to coming into the demo if you use your own computer.
   b. There must NOT be any code in the clipboard or on the screen. In the event that code is on your screen or clipboard, it will be an automatic failure.
   c. Upon check-in, the technical attendants will be confirming that your computers are ready to use and free of code being displayed. A hard reboot will be expected.
   d. *Make sure to have already done any system updates prior to the proficiency demo taking place.*

7. When coding, you are encouraged to spend time designing your solution and drawing pointer diagrams. Scratch paper will be available and must be turned in with your paperwork
   a. The prototype supplied may not be altered. But you may always write more than a single function to solve the problem.
   b. The **main function needs to call the prototype that we supply.**
   c. When coding, please **comment out code rather than deleting it.**

8. **During the demonstration:**
   a. **No use** of the **internet** except to login to our testing environment
   b. **No use** of **help** or the **man pages** during the demonstration
   c. **No** other accounts may be logged into during the demonstration
   d. **Only one shell** session may be open
   e. **Proficiency using** vi, vim or emacs is expected.
   f. **We will ask you to demonstrate navigation** and **search and replace**

9. **Syntax Requirements:**
   a. **No global variables** may be used
   b. **No** use of the **string class**
   c. **No** use of **static locals**
   d. Although you will be using a class for the list of data, a struct is used for the node for the proficiency demos
   e. All functions will have a non-void return type, so get used to correctly returning a value from a function and using that returned value!

10. **Design Process:**
    a. You are encouraged to spend time with **design**; scratch paper is provided.
    b. You are also allowed to **compile, test, and debug** your work multiple times. Please consider performing incremental implementation, compiling code as you go.
    c. All questions will expect a **recursive solution** be used to solve a repetitive problem.
    d. **NO LOOPS may be used in your solution;** *there are no exceptions to this!*
    e. It is expected that your solution will minimize traversals. This means that if you have a question to remove the last node and return the total number of nodes in the list, then it all needs to be done through one traversal. Do not attempt to traverse to remove and then traverse again to count!
    f. You may write **more than the function** supplied if necessary, but main must call the function assigned.
    g. The prototype(s) supplied **may not be changed.**
    h. For questions that have you count, average, or traverse in some way, your **solution should NOT modify the data structure.**
    i. For questions that have you insert or copy, your solution should **NOT delete items.**
    j. **Avoid extra traversals** just to calculate the returned value (for example, if the question asks to count or sum the data, that should be accomplished as part of the recursive problem and not through an extra pass through the data set).

11. **The midterm proficiency demo** is scored in CS202 within 60 minutes.
    a. During this time frame, if we have determined your score we will ask you to submit the results.
    b. If you finish prior to this, please raise your hand and an assistant will be with you to complete the scoring process
    c. DO NOT submit your work without an assistant present!
    d. It is possible that we will score you work prior to completion.
    e. When we have scored your work you will be asked to submit the results using the ./submit script. Please wait until prompted to do so.

CS202
Programming Systems

Lab #5: Exception Handling

CS202 Lab #5: Goals

Important!

Make sure to keep up with the course materials! Lab #5 we will be working on CS202 Topic #5 on exception handling!

Goals:

1. **Start with the group exercises.** Gain experience determining where exception handling is most useful and how to incorporate it into a program. Practice recursion and data structures in preparation for the midterm exam and proficiency demonstrations that will be taking place.

2. **Then, work with the linux and vim manual** for the next *10 minutes*. Learn more about using vim and linux tools

3. **Plan** to use exception handling
   a. **Learn how to apply exception handling in our programs.**

4. Apply **test plans** and **pointer diagrams** to verify your software. **When working with test plans**, think about all the cases that need to be verified.

5. Once the lab is completed, you can perform the **self-check quiz** at the end of the lab materials.
   a. Use the self-check quiz questions at the end of the lab materials for each section to test if you fully understand the material
   b. The best way treat this material is to answer the self-check quiz questions using a closed book/closed notes technique.

REMEMBER

Practice the vim exercises!

Do you know how to search and replace?

CS202 Lab #5 – Background Information

Exception Handling

The most important part of exception handling is the separation of error detection from error handling. We can detect an error with an if statement. Then, if a condition happens, we throw an exception that starts unwinding the stack until the error is handled. Error handling only takes place for exceptions that are "thrown from within a try block". Errors are then handled with the catch blocks that occur immediately after the try block.

Errors can be handled within the function where the error is detected, or within outer functions. With exception specifications, programmers can examine the prototypes and function headers to determine which exceptions they need to handle after calling a function. An exception specification list indicates which type of exceptions may be thrown from the function (but not caught). By examining the exception specification list, we know to call such functions within a try block. If errors are not handled, the program will terminate and abort when exceptions are thrown.

Make sure to read through Topic #5 and learn about these terms:

1. Try block
2. Catch block
3. Arguments for catch
4. Throwing an exception
5. Exception specification list

There is an extra benefit when working with exception handling with classes. Constructors have no simple solution for "returning" a success/failure flag like we learned in CS163. So if you were trying to allocate an array of head pointers in a constructor and new fails or the argument passed in is invalid (maybe negative), how could this be handled? A global? A public data member? Neither of which are viable structured programming or OO solutions.

Instead, exception handling allows us to throw exceptions from member functions and then "catch" them in the calling routine. This can happen whenever the "use" of the constructor (e.g., creating an object of the class) occurs WITHIN a try block! *Think about this!*

CS202 Lab #5 – Pre-lab Worksheet
Exception Handing

This worksheet must be completed to attend the Lab.

Pre-lab #5 (2 Pages)

Gain experience determining where exception handling is most useful and how to incorporate it into a program. Practice recursion and data structures in preparation for the midterm.

Applying Exception Handling:

Look back at your first programming assignment and answer these questions:

_____ Step 1. List three error conditions that you checked for in one of your programming assignments?

a. When head is NULL for removal…

b.

c.

_____ Step 2. Create a struct type with a meaningful name for one of these conditions:

_____ Step 3. Describe an example situation where a try block and corresponding catch blocks should occur within the same function:

_____ Step 4. Now describe another situation where a function throws an exception that is caught in the calling routine (e.g., main) instead:

General Concepts of Exception Handling

_____Step 5. For exception specification lists,

a. When these are used and what do they communicate to a programmer calling the function that uses them?

b. What type of exceptions can be thrown when we use an exception specification list with a throw()?________________

c. What type of exceptions can be thrown when we use an exception specification list with a throw(int, float)?__________

d. Where is the exception specification list placed?

e. Show sample syntax for an exception specification list:

_____Step 6. Show the syntax for a default catch:

Pre-lab #5 Completed
Checked in-Lab (TCSS Initials):_________

CS202 Lab #5 – Exception Handling

Check-off each step as you proceed! Submit the code at the end of lab.

Collaboration

- We will be working in pairs with this lab. As such, team up with someone in this class and sit next to each other. Although you will be writing your own code, individually, you will be evaluating each other's code
- Each lab question is designed to be done in a step by step fashion. The success of later steps depends on earlier.
- **At the end of the first page, have your answers checked by a Lab Assistant**

Designing with Exception Handling

Level 1 - Introductory

_____Step 1. List 3 reasons you would want to use exception handling

Level 2 - Intermediate

_____Step 2. With classes, we use exception specification lists to specify which exceptions could be thrown by a member function. How are structures used in combination with this?

Level 3 - Proficient

_____Step 3. Examine the .h files from the previous labs. Think about what functions might experience errors where exception handling could be useful. List three functions that would benefit by throwing an exception:

****** RECEIVE feedback from Lab assistants before continuing with linux & vim exercises***

Implementing Exception Handling

With this part of the lab, we will be working with the same code from Lab #2 where there we have just hourly employees and then collections of those employees for each store. In this lab you are given access to parts of the code in order to implement exception handling.

Level 1 - Introductory

_____Step 4. Login to **cs202lab.cs.pdx.edu** using your assigned login and password

_____a. Change into the CS202/Lab5 directory **cd CS202/Lab5**

_____b. Use a **linux editor** such as **vi, vim, or emacs** to type in a program.

_____c. **Compile** and link to my object code on **linux**.

```
g++  *.cpp  *.o  -g  -Wall
```

_____Step 5. Next implement the functions in the **cs202_lab5.cpp** file, adding exception handling to each. For each function, even if it doesn't't have a try block, you should add an exception specification list. For those functions that do not have a try block, experiment with throwing an except from the function and placing the try block in main.

_____a. For the **again** function, read in a user's response, remove extra characters, capitalize the response, and return a true if a 'Y' is entered.

i. Throw an exception if the user enters in an invalid response (not a Y or an N) when asked a question.

ii. **Question to discuss**: Where should this exception be handled?____

iii. _________Experiment with the default catch.

_____b. For the **address constructor**, if the arguments passed in are NULL throw an exception.

i. Create a "NULL_STRING" struct in a new file called error.h.

ii. Have members for erroneous street and zip pointers

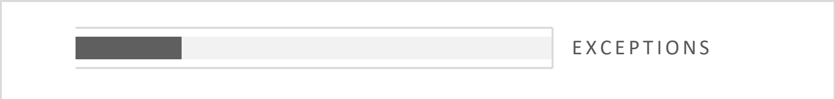
```
struct ______________
{


};
```

iii. Then, #include your error.h in the person.h file *(and no where else!)*

iv. **Throw the error from the address constructor**

v. **Catch the error in main.** Discuss where the try block belongs and what data types to catch. Does the catch need an argument?_____

Proficiency Scale after Step #5:



_____c. For the **hourly employee's set_pay_rate** function, what should happen if the pay specified is a zero or negative value?

i. Discuss what should happen

ii. Create an "INVALID_PAY" struct.:

iii. Use an **exception specification** list. Make sure to place the exception specification list in both the prototype and the function header. Write your exception specification list here:

____

Proficiency Scale after Step #5:

EXCEPTIONS

Level 2 - Intermediate

_____Step 6. Next, in **main.cpp**, add exception handling for the situation where the user enters in too many characters, throw an exception.

Remember the cin.peek() function!

i. Discuss what should happen

ii. Create a name for a struct to represent this error:___________________.

iii. Discuss where the try block belongs:

Proficiency Scale after Step #6:

EXCEPTIONS

Level 3 - Proficient

_____Step 7. **Challenge:** For the **employee** class, implement the two **compare** functions, one that compares the name and the other that compares looking for a review. Throw an exception if matches are not found.

Plan First:

Proficiency Scale after Step #7:

_____Step 8. **Practice important vim and linux commands:**

_____a. Show how to search and replace the first occurrence of text type:_______

_____b. Show how to replace all occurrences: ______________________________

_____c. What do you need to do to compile in preparation to use gdb?__________

_____d. Set a break point?_______________

_____e. Set a break point at a particular line:_______________________

_____f. List the break points _______________________

_____g. How do you single step, entering functions ____________

_____h. Go to the next line of code, not entering functions ___________

_____i. Now delete the all breakpoints:____________________________

_____Step 9. **Tar the files** for the lab by typing:

```
tar   -cvf   CS202_Lab5.tar   *.cpp   *.h
```

_____Step 10. **List the files** in the directory to confirm that the .tar file exists

_____Step 11. **Submit** your program by typing **./submit** at the linux prompt

LAB #5 Completed
ID Scanned on Completion (TCSS Initials):___________

CS202 Lab #5 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

1. What is the advantage of exception handling?

2. What type of code should be put in a try block?

3. What is the syntax of a default catch (or "catch all")?___________________________

4. In what order should the catch blocks be placed?

5. Can more than one catch block be executed for a single exception that is thrown?____
6. What is the syntax of throw?____________
7. What happens if a throw happens without being in a try block?___________________
8. What happens if the function call is in a try block, but a throw happens from within the function (and not within another try block)?_________________________
9. What happens if throw is used without an argument?__________________
10. What is the purpose of an exception specification list?

11. If a function has thrown an exception, but expects the calling routine to catch the exception – what happens to any memory allocated with new in the function (that is not controlled by the class)

    ____

12. When we throw an exception which is caught in an outer function, when are destructors of local class objects invoked?

    ____

CS202
Programming Systems

Lab #6: Operator Overloading

CS202 Lab #6: Goals

Important!

Start preparing for final proficiency demos and review binary search tree algorithms!

Start practicing using gdb and vim (navigation, and search/replace); these will be demonstrated as part of your final proficiency demo!

Goals:

1. **Prepare** by reviewing Topic #6 on operator overloading prior to Lab #6

2. **Always start with group activities collaborating with other students in the class.** Limit your time on these activities to 10-15 minutes. Gain experience using friends, operator overloading, and apply these to an abstraction

3. **Then, work in the linux and vim manual** for the next *10 minutes*. Learn more about using vim and linux tools

4. Once the lab is completed, make sure to start practicing with data structures and advanced tree algorithms.

5. Don’t forget to perform the **self-check quiz** at the end of the lab materials.
   a. Use the self-check quiz questions at the end of the lab materials for each section to test if you fully understand the material
   b. The best way treat this material is to answer the self-check quiz questions using a closed book/closed notes technique.

Make sure to Practice!

Every day implement one BST algorithm

Test your code on cs202lab.cs.pdx.edu in CS202/fpdemo

CS202 Lab #6 – Background Information

Operator Overloading

Operator overloading is provided to allow us to create class types and have them behave as if they were built-into the language. It is best used when combined with data abstraction. Now, with operator overloading, our abstract data types will have not only functions but operators associated with the data!

Operator overloading allows operators to support our class objects. In fact, the istream and ostream classes are great examples of operator overloading. The >> and << operators are used for input and output when applied to istream and ostream objects (e.g, cin and cout). In C and C++, the >> and << operators are bit-shifting operators when working with built-in types. By overloading them in the context of input and output, we have a simpler interface for I/O than one might experience in other programming languages.

When working with data abstraction, we can build a list, stack, queue, string, or other “data type” and specify operators to work on objects of those classes. By overloading the >> and << operators, we can read and write lists, stacks, queues, strings, or others. No longer do we need to create named functions such as “display”, “read”, or “input”. We can also use the + and += operators to concatenate lists or strings together and the [] operator to access individual items in a collection. And this is just the beginning.

The key benefit of operator overloading is that it allows us to build a class type that can be used consistently with other classes and data types. It allows us to use objects of the class with template classes and functions with ease.

For such benefits, we need to be very careful in creating solutions that use operator overloading. The operators must be created to be consistent with how they behave with the built in types. They need to have the expected operand types and residual values to allow for seamless use. And, whenever possible avoid excessive copying through the use of copy constructors.

Guidelines to Overloading Operators

Guidelines are useful to maintain consistency with the built-in definitions of operators. Maintaining consistency is essential for transparent use. Clients should be able to use any of the relevant operators with abstract data types or classes' in the same way in which they are accustomed.

Overloading Operators as Members or Non-Members

Choosing whether to overload operators as members or non-members is dependent on the data type of the first operand and whether symmetry must be considered. There are three categories of operators that we recommend be defined as members: those that must be members, unary operators where the only operand is required to be an object, and binary operators that make the most sense with the first operand being an object. Notice that all unary operators should be members, since all unary operators require that the one and only operand be an object of a class when overloading. The following table summarizes these recommendations.

Operators that must be overloaded as members:	
=	simple assignment operator
[]	subscript operator
()	function call operator
->	indirect member access operator
->*	indirect pointer to member access operator
All unary operators make sense as members:	
&	address of operator
*	dereference operator
++	increment operator (both prefix and postfix)
--	decrement operator (both prefix and postfix)
+	plus operator
-	minus operator
~	bitwise negation operator (complement)
!	logical negation operator
And, these binary operators should be members:	
*= /= %= += -=	arithmetic assignment operators
<<= >>= &= ^= |=	bitwise assignment operators

Guidelines – Operators that should be Overloaded as Members

Operators that allow either the first or second operand to be an object of a class are best defined as non-member functions. All the arithmetic, bitwise, relational, and equality binary operators fall into this category. For each of these operators, if both operands are not of the same type, then we should make sure that either type can be placed as either the first or second operand. This means that we typically overload more than one version of these operators to handle all possible combinations. The following table summarizes those operators that we recommend be defined as non-members.

* / % + -	arithmetic operators
<< >> & ^ |	bitwise operators
< <= > >=	relational operators
== !=	equality operators

Guidelines – Operators that should be Overloaded as Non-Members

There are three other operators that can be overloaded but that are not listed in the previous two tables. These are the logical and comma operators. **We do not recommend overloading these operators** as there is no way for us to overload them in a way that is consistent with their built-in meaning. These recommendations are summarized in the following table:

&& ||	logical operators
,	comma operator

Guidelines – Operators that should not be Overloaded

To obtain behavior consistent with built-in types, we should begin by evaluating whether both the extraction and insertion operators are needed. This will allow us, at a minimum, to read and write objects of that type.

For classes that dynamically allocate memory on an object-by-object basis, we should overload the copy constructor and assignment operator to perform a deep copy. Therefore, it is common for classes to have at a minimum a copy constructor, an assignment operator, and the extraction and insertion operators overloaded.

Whenever we overload any of the arithmetic or bitwise binary operators, we should also consider overloading the corresponding arithmetic or bitwise compound assignment operators. And, when we overload any operator that has a counterpart operator (++ and --, + and -, << and >>, * and /, etc.) then we should overload neither or both. Only when the occasion requires will we overload one but not the other (e.g., it would be strange to be able to use ++ to increment but not -- to decrement).

Whenever we overload any of the equality or relational operators, we recommend overloading all relational operators. This means that if we can compare for equality, we should be able to compare for inequality.

Operators That Modify the First Operand

To maintain consistency with the built-in definitions, only the following operators should be designed to modify the contents of the current object: =, +=, *=, /=, %=, -=, ++, --, [], and * (dereference). These should be implemented as class members where the first operand is an object of that class. This is preferred over friend functions, so that the object is sent implicitly and the number of friends can be minimized. All other operators implemented as members should be specified as constant members. This will allow us to use those operators with constant objects as the first operand.

Please notice that the ++ and -- operators modify the current object. However, the postfix version of these operators result in a rvalue result. This means that it is possible for some operators that return a rvalue to also modify the current object.

Using Temporaries

When overloading operators, we have the ability to return objects as rvalues, lvalues, or modifiable lvalues. By returning an object by value, the information returned can only be used as an rvalue. By returning an object by reference, the information returned can be used as an rvalue, lvalue, or modifiable lvalue. Operators that use temporary objects instead of modifying the current object only can return those objects by value. This is because objects local to a function have a lifetime from their definition until the end of the function. Returning a reference to a temporary isn't possible. This is because return by reference requires that the returned object's lifetime correspond to the client's expectations (and be within the client's control). This means that we must never return a reference to local object.

Operators That Allow for Constant Operands

To maintain consistency with the built-in definitions, make sure to consider when it is appropriate to allow the operands to be constant or non-constant objects. Operators that do not modify the first operand should either be constant members or friend functions that specify the first argument by value or as a constant reference. Binary operators that do not modify the second operand should specify the corresponding argument either by value or as a constant reference. Use pass by value when working with built-in types and use constant references when working with class types that need to perform deep copies.

What Operators to Consider Overloading

We are now ready to review some general recommendations that when followed will assist in creating abstract data types that behave as clients might expect. As we design our classes, we should consider each of the following suggestions. However, there may be times when some, or all, of these suggestions will not be appropriate. Therefore, take them into consideration, but don't follow them blindly! The following two tables summarize these general recommendations and shows which operators work best in pairs. Consider using these tables as a quick reference when double checking if all the necessary operators have been overloaded for a new abstract data type.

To be able to read and write objects, overload:	
>> <<	as extraction and insertion operations
If dynamic memory is part of the object, overload:	
=	simple assignment operator
Provide compound assignment operators if overloading any of:	
* / % + -	arithmetic operators
<< >> & ^ |	bitwise operators
Provide all relational operators if overloading any:	
< <= > >=	relational operators
Provide all equality operators if overloading any:	
== !=	equality operators
*= /= %= += -=	arithmetic assignment operators

Guidelines – What Operators to Consider Overloading

Provide both forms as pairs:	
-> ->*	indirect member and pointer to member access operators
& *	address of and dereference operators
++ --	increment and decrement operators (prefix and postfix)
+ -	plus and minus operators
* /	multiplication and division operators
+ -	addition and subtraction operators (except when “appending”)
new delete	dynamic allocation/deallocation
new[] delete[]	dynamic array allocation/deallocation

Guidelines – Operators that Work Best in Pairs

A Summary of the Guidelines

The following summarizes the guidelines discussed in this chapter when overloading an operator. This represents a step-by-step approach to designing our overloaded operators. Remember, it takes careful planning to meet the objectives of operator overloading.

1) Determine which operators should be overloaded to ensure that the abstract data type will behave consistently with the fundamental data types.

2) For each operator, determine the data types for the operands. When the first operand can be an object of the target class, then it should be overloaded as a member. Otherwise, it should be overloaded as a non-member friend.

3) For each operator, determine whether the first operand can be modified. If the first operand cannot be modified for an overloaded member, then the member should be constant; for a non-member, then it should be specified either by value or as a constant reference.

4) For each operator, determine whether the second operand can be modified. If the second operand cannot be modified, it should either be by value or a constant reference.

5) For each operator, determine the data type of the residual value.

6) For each operator, determine whether the residual value should be an rvalue, a non-modifiable lvalue or a modifiable lvalue. If it is an rvalue, then the operator can either return by value or by const reference. If it is an lvalue, then the operator must return by reference.

Conventions for Operator Overloading

The following table summarizes the behavior of each of the built-in operators. This table should be used to assist in overloading operators for abstract data types to keep the usage in line with common practice.

	Modifies	Allows		Member
Category	Current	Constant	Residual	or
of Operator	Object	Operands	Value	Non-Member
Simple assignment	Yes	2nd only	lvalue	Member
Subscript	No	Yes	lvalue	Member
Function call	-	-	-	Member
Indirect member access	No	Yes	lvalue	Member
Indirect pointer to member	No	Yes	lvalue	Member
Address of	No	Yes	lvalue	Member
Dereference	No	Yes	lvalue	Member
Prefix increment /decrement	Yes	No	lvalue	Member
Postfix increment/decrement	Yes	No	rvalue	Member
Plus and minus	No	Yes	rvalue	Member
Bitwise negation	No	Yes	rvalue	Member
Logical negation	No	Yes	rvalue	Member
Arithmetic	No	Yes	rvalue	Non-Member
Bitwise	No	Yes	rvalue	Non-Member
Relational	No	Yes	rvalue	Non-Member
Equality	No	Yes	rvalue	Non-Member
Arithmetic assignment	Yes	2nd only	lvalue	Member
Bitwise assignment	Yes	2nd only	lvalue	Member
Extraction	Yes	No	lvalue	Non-Member
Insertion	Yes	2nd only	lvalue	Non-Member

Guidelines – Conventions

Rules for Operator Overloading:

1. **Binary** operators are overloaded as member functions or as friends
   - **As a member,** there is one argument representing the second operand
   - **As a friend,** there are two arguments, one being an object of a class
   - If the FIRST operand is a class object, overloaded it as a member function.

2. **Unary** operators must have an object of a class as the only operand. As such if it is of your class type – then they are overloaded as member functions.
   - **As a member,** there would be no arguments
   - **As a friend,** there would be one argument (a class type, passed by reference)

3. Operators that **alter the first operand**, are usually required to be **member functions**
   - But, as a friend, the first argument would be passed by reference
   - These are lvalue operators
   - All of the rest are rvalue operators (*when the first operand is not modified*)

4. Operators that **alter the second operand**…
   - Require the second operand to be **a reference** (not a constant reference)

5. **Rvalue** operators typically cause the copy constructor to be invoked.

6. With **chaining** (also known as member function concatenation),
   - The residual value is a **reference** to a class object
   - If the residual value is the current object, return the **dereferenced this pointer** (return *this)

7. The >> and << operators specifically (as they are already overloaded operators)
   - Must be overloaded as non-member functions that are friends of your class
   - Make sure to pass the istream and ostream objects by reference
   - And, don’t forget about supporting chaining with these operators; to support this, the istream or ostream reference must be returned.
   - If you were to return by value, the stream would be closed. Therefore, always return an istream or ostream object by reference (never by value)

8. **Never** have any operators with **void** return types!
   - A void return type means that the operator would not have a residual value
   - All operators that can be overloaded in C++ have a residual value when used with built-in types.

9. **The assignment operator** needs to consider these operations:
   - **Check for self assignment**. If the address of the argument being copied is the same as the current object, no copying! `if (&argument == this)`

   - **Copy the base class object**. If the class is derived from a base class, make sure to cause the base class' assignment operator to be invoked

     ```
     base::operator=(source_argument);
     ```

   - **Deallocate**. If the class manages dynamic memory, deallocate prior to reallocating as part of the copy process

   - **Return the current object**. Return the dereferenced this pointer, as it contains the current object's new contents

CS202 Lab #6 – Pre-lab Worksheet
Operator Overloading

This worksheet must be completed to attend the Lab.

Pre-lab #6 (2 Pages)

Gain experience using friends, operator overloading, and apply these to an abstraction

Fundamentals of Operator Overloading:

As we think about the operators we use when programming, which ones act as follows.

Pick from this list of operators:

+   +=   =   !=   <<   >>   []   ++ (prefix)   (postfix) ++

_____1. Which operators alter the first operand?____________________________

_____2. Which operators cause the copy constructor to be invoked?___________

_____3. Which operators result in an “rvalue”?____________________________

_____4. Which operators result in a “lvalue”?____________________________

_____5. Which operators should return by reference?_______________________

_____6. Which operators are always overloaded as member functions_________

_____7. Which operator always has a non-constant class object as an argument___

_____8. Which operators should be implemented as friend functions (rather than as member functions for our classes)______________________

_____9. Show an example of chaining (also known as member function concatenation) – this is the user of one operator’s residual value in another expression:

____

_____10. What would it mean for an operator to have a void return type?

____

_____11. **Fill in the blank.** Let's now gain some experience writing prototypes for operators for **a list class of employees**.

- Remember the residual value is represented by the returned type.
- Rvalue operators return by value, Lvalue operators return by reference.
- Constant member functions mean that the function doesn't alter the data members. This only applies when writing the operator as a member function and not as a friend
- Pass as constant references whenever possible:

a. Concatenate two lists together:
__________ operator + (const list &) ___________ ;

b. Concatenate a list onto our current object's list:
__________ operator += (__________ list _______);

c. Add an employee to our current object's list:
__________ operator += (__________ employee _______);

d. Assign one list to be our new current object's list:
__________ operator = (___________list &);

e. Compare two lists to see if they are the same:
__________ operator != (__________ list &)__________;

f. Access one employee in the current object's list of employees:
employee & operator [] (_____________) __________;

g. Output the current list:
_______ _________ operator << (__________ , ________ list ______);

Pre-lab #6 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #6 – Operator Overloading

Check-off each step as you proceed! Submit the code at the end of lab.

Group Activities - Collaboration:

- We will be working in pairs with this lab. As such, team up with someone in this class and sit next to each other. Although you will be writing your own code, individually, you will be evaluating each other's code
- Each lab question is designed to be done in a step by step fashion. The success of later steps depends on earlier.
- **At the end of the first page, have your answers checked by a Lab Assistant**

Applying Operator Overloading:

The classes in today's lab are based on some simple geometric concepts. The Point class represents a single point in 2D space. The Shape class is a generic shape and has an origin and a color. The Circle class is a subclass of shape and contains the radius. In today's lab we will be implementing some simple operations to manipulate the Points and Shapes using operator overloading.

Level 1 - Introductory

_____Step 1. List some operators that might make sense for the Point and Shape classes:

_____Step 2. List some subclasses for shape that might make sense:

Level 2 - Intermediate

_____Step 3. Discuss the importance of the **const** for each of these functions:

a. Point operator + (**const** Point&) **const;**

b. Point& operator += (**const** Point&);

Proficiency Scale after Step #3:

Individual Coding - Follow the steps exactly as specified to alter the provided code:

All functions are to be implemented in the cs202_lab6.cpp file. Function stubs have been provided to simplify testing. A simple test suite has also been provided to give **PASS/FAIL** feedback for each function. For each step compile and run to ensure you are passing the provided tests:

Level 1 – Introductory

_____Step 4. Login to **cs202lab.cs.pdx.edu** using your assigned login and password

- Change into the CS202/Lab6 directory **cd CS202/Lab6**
- Use a **linux editor** such as **vi, vim, or emacs** to type in a program.
- Compile and run the program to see the expected formatting starting
- **Remember, when it is time to compile** and link to my object code on **linux** perform the following: **g++ *.cpp *.o**

_____Step 5. Implement each of the following operators in the Point class

_____a. Addition (+) operator

- What information is being returned?_____________________
- Is a temporary Point object needed?_____________________

_____b. Addition assignment (+=) operator

- What information is being returned?_____________________
- Is a temporary Point object needed?_____________________
- How will this be different from the addition operator?

____

_____c. Equality (==) operator

- What information is being returned?_____________________

_____d. Insertion (<<) operator to output the Point

- What information is being returned?________________________
- What would happen if the second argument was not a constant reference?____
- Does this need to be a friend function and why/why not:

____

_____e. **Compile and run** the program to see the expected results

Proficiency Scale after Step #5:

Level 2 - Intermediate

_____Step 6. Implement each of the following operators in the Shape class.

_____a. Assignment (=) operator

- Did you check for self-assignment?______________
- How was that done? if (____________________)
- If self-assignment is true, what should be returned?_____________

_____b. Addition assignment (+=) operator

- Should we return by value or by reference?_______________
- Should it modify the current object's data members?________

_____c. Insertion (<<) operator to output the Shape

- Each subclass of Shape has a display member function, how does this impact your implementation?

  ____

_____d. Addition (+) operator

- Should we return by value or by reference?_______________
- Should it modify the current object's data members?________
- What information is being returned?_____________________
- Is a temporary Shape object needed?_____________________

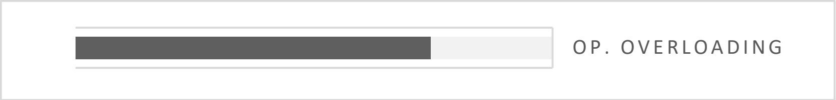
_____e. **Compile and run** the program to see the expected results

Proficiency Scale after Step #6:

Level 3 - Proficient

_____Step 7. Implement the remainder of the operators outlined in the Circle class

Proficiency Scale after Step #7:



_____Step 8. **Tar the files** for the lab by typing:

```
tar   -cvf    CS202_Lab6.tar   *.cpp   *.h
```

_____Step 9. **List the files** in the directory to confirm that the .tar file exists

_____Step 10. **Submit** your program by typing **./submit** at the linux prompt

LAB #6 Completed
ID Scanned on Completion (TCSS Initials):___________

CS202 Lab #6 – Practice Operator Overloading

Examine the code from Lab #2 (using inheritance) and apply operator overloading to that code.

In Lab #2 we worked with employee classes. To simplify, let's take just one class – an employee class – and think about how operator overloading might apply:

1. List five of the operators that would make sense when creating an employee class:

   _______ ________ _______ ________ ________

Now create a prototype for one operator in each of the following categories and compare with others:

2. Binary member function (1 argument):

   ______ operator ___ (_______________________)_________;

   Double check

   - Data type of the residual value
   - Should it be returning by value or by reference
   - Could the argument being passed in possibly be a constant?
   - Are you passing class objects by reference rather than by value?
   - If you are passing a pointer, ask yourself how the client program will be able to use this operator

3. Binary friend function (2 arguments):

   ______ operator ___ (_______________________)_________;

   Double check

   - Data type of the residual value
   - Should it be returning by value or by reference
   - Could the argument being passed in possibly be a constant?
   - Are you passing class objects by reference rather than by value?
   - If you are passing a pointer, ask yourself how the client program will be able to use this operator

Continue to create prototypes for operators in each of the following categories and compare with others:

4. Unary member function (no arguments):

______ operator ___ (_______________________)___________;

Double check

- Data type of the residual value
- Should it be returning by value or by reference
- Could the argument being passed in possibly be a constant?
- Are you passing class objects by reference rather than by value?
- If you are passing a pointer, ask yourself how the client program will be able to use this operator

5. Unary friend function (1 arguments):

______ operator ___ (_______________________)_________;

Double check

- Data type of the residual value
- Should it be returning by value or by reference
- Could the argument being passed in possibly be a constant?
- Are you passing class objects by reference rather than by value?
- If you are passing a pointer, ask yourself how the client program will be able to use this operator

CS202 Lab #6 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

Operator Overloading

List **two operators** for each of the following questions

1. Must be overloaded as members

   ______________ ________________

2. Should be overloaded as non-members

   ______________ ________________

3. Cause the copy constructor to be invoked

   ______________ ________________

4. If overloaded as a member, should be a constant member

   ______________ ____________

5. Should have all arguments as constant objects

   ______________ ________________

6. Result in an lvalue ______________ ________________
7. Result in an rvalue ______________ ________________
8. Return the *this ______________ ________________
9. Returns by value ______________ ________________
10. Returns by reference ______________ ________________
11. Are binary operators ______________ ________________

Short Coding

12. Show an example of "chaining": ____

13. Show the prototype for the copy constructor for a list class:

    list::list(___________________________________)

14. **Fill in the blank.** Write prototypes for operators for **a list class, of employees**. We are adding a new employee to the list of employees:

    a. For the += operator
        i. Is it a member function or a non-member friend?___________
        ii. Write the prototype:
        _______________ operator += (_______________________);

    b. For the + operator
        i. Is it a member function or a non-member friend?___________
        ii. Write the prototype:
        _______________ operator + (_______________) ______________ ;

15. **Why shouldn't the** += operator use the + operator in its implementation?

16. When does the concept of **self assignment** become important?

17. Explain the difference between the copy constructor and the assignment operator when implementing the body of these functions:

CS202
Programming Systems

Lab #7: Java Workshop

CS202 Lab #7: Goals

Important!

IF POSSIBLE BRING YOUR OWN LAPTOP to Lab #7 so that you can learn how to install and work with an IDE

INSTALL and Learn to use an IDE

1. **Start today with the linux and vim manual** for the first *10 minutes*. Learn more about using vim and linux tools

2. **Prepare** by looking up information about IntelliJ's IDEA to and begin the installation process. You may use other IDE's.

3. **Experience** the use of an IDE with tutorial sessions in lab

4. **As time permits**, begin on Lab #8 which begins the Java workshop!

Java Workshop Begins

1. **Prepare** by reading through the Bruce Eckel book on "Thinking in Java" and review Topic #8 power point slides.

2. **Experience** the implementing Java software starting with basic programs and ending with linked data structure problems. Gain experience with each of the following concepts:

    a. I/O
    b. Functionality of main
    c. Creating objects of a class
    d. Concept of inheritance in Java

3. **As time permits**, begin on Lab #8 applying recursion!
4. **Don't forget**, to work on the self-check quiz as a closed-book, closed notes verification of your progress through this material!

IT IS IMPORTANT TO INSTALL the IDE PRIOR to the LAB Session (this is your prelab)

CS202 Lab #7 – Background Information

Installing an IDE

The purpose of this pre-lab is to get set up to install the Jetbrains Java IDE called IntelliJ IDEA as we transition from C++ to learning about new programming languages. Some of the installations can be time consuming, so it is best to do as much prep work before Lab #7 as possible! Once you complete the installation, you can begin with the Java Workshop in Lab #8.

Using IntelliJ's IDEA

Using an IDE like IntelliJ is helpful once you have already established strong programming skills; at that point it can be used as a tool for organizing and working on a larger projects. It provides suggestions and hints which are especially useful if working with a language that is relatively new to you. However, these hints only increase productivity if you already know some basics and can make an educated decision on whether the hint the IDE is providing is actually what you want to use. IDEs handle all of the basic "boilerplate" code that is required in most if not all programs being created within them. Also, there are many ever changing libraries and programming frameworks which IntelliJ and other IDEs can help manage.

Most IDEs have some sort of plugin repository or ability to extend its functionality. IntelliJ is particularly good at this; they have an active community of contributors and regularly update and maintain their software. It is important to note that Jetbrains products (IntelliJ, CLion, PyCharm, etc) are free to students with an academic email address while in school.

Using IntelliJ Idea

Creating a Project

1. Open Idea
2. Click “Create New Project”
3. Click “Next” at the bottom right
4. Check the “Create project from template” box
5. Highlight “Command Line App” then click “Next”
6. Name your project, then click “Finish” at the bottom right

Running/Debugging your program

1. To compile and run a program you’ve written, click the green “play” button at the top right
    a. You may have to click the “Make Project” button (the down arrow with ‘1’s and ‘0’s after it at the top-right) before the “play” button becomes available
2. To debug a program:
    a. Click to the left of a line to set a breakpoint
    b. Click the bug at the top right
    c. Features like stepping over/into functions or resuming, are all located in the “Run” menu

Adding Classes

1. The left side pane of your screen is the “Project Tool Window”
2. Expand the name of your project (click the arrow to the left of the name)
3. Expand “src” or source
4. Right click on your package name (“com.company” by default)
5. Go to “New” at the top of the menu, then click “Java Class”
6. Name your new class and click “Ok”

CS202 Lab #7 – Pre-lab Worksheet

Preparing to Install of IntelliJ's IDEA

Installation of IntelliJ's IDEA

Information about IntelliJ's IDEA can be found at: **https://www.jetbrains.com/idea/**

_____Step 1. Go to **http://www.jetbrains.com/student/**
_____Step 2. At the bottom of the page click "Apply Now" for a student account
_____Step 3. Fill out the form, making sure to use your pdx.edu email address
_____Step 4. Confirm your account by clicking the link in the email from IntelliJ
_____Step 5. After creating your account, click on IntelliJ Idea
_____Step 6. Download either version (The Community version has fewer features and plugins, but still has much more than we'll be using in this course)
_____Step 7. Install the program

Installing the Java Development Kit:

_____Step 8. Go to **oracle.com/technetwork/java/javase/downloads/**
_____Step 9. Click the left box with "Java" in it
_____Step 10. Install the appropriate JDK for your OS to the default directory

(Continue to the next page for the Java prelab)

Introduction to Java

This worksheet must be completed to attend the Lab

Review Topic #8 materials and answer the following to be prepared to program in Java:

_____1. List the data types in Java that are different than in C++:
____

_____2. List the differences between identifiers in Java vs. C++:
____

_____3. Why don't classes in Java have a semicolon afterward?

_____4. Are arrays statically or dynamically allocated in Java?_______________

_____5. If we wanted a constant integer (MAX) of size 10, show how we could create this as a field in Java:_____________________________________

_____6. Show an example for creating an array of 10 integers in Java:

_____7. When we pass an "array" to a function, what gets copied? ___________

_____8. In Java, what can we use to determine the length of an array? ________

a. When using this feature, does it tell us the number of items stored in the array or the actual length of the array itself? _______________________

b. How is it possible to determine the number of items stored in an array?

_____9. If we were to pass a head (a node reference) to a function, is it possible for the function to modify the value of the node reference (indirectly in the calling routing)? *Think about this!*_____________________________

Pre-lab #7 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #7 – Working with an IDE

Check-off each step as you proceed! Submit the code at the end of lab.

Pointing Idea to the SDK

_____Step 1. Start Idea

_____Step 2. If a box pops up asking for your license key
Select “Jetbrains Account” and enter your credentials

_____Step 3. Click “Create New Project”

_____Step 4. Click “New” next to Project SDK at the top-right of the screen

_____Step 5. Click “JDK”

_____Step 6. Navigate to the SDK default directory

_____Step 7. Click “OK” with jdk[version #] highlighted
NOTE: The directory, not a particular file, should be highlighted

Setting up IdeaVIM

_____Step 8. Click “Configure” at the bottom of the main Idea start screen

_____Step 9. Click “Plugins” in the resulting menu

_____Step 10. Click “Install JetBrains plugin…” at the bottom right

_____Step 11. Scroll down to IdeaVIM and click on it

_____Step 12. Click “Install Plugin” at the top right

NOW BEGIN…
The Java Workshop!!

LAB #7 IDE Installation Completed
ID Scanned on Completion of IDE Installation (TCSS Initials):______

CS202 Lab #7: Programming in Java
Worksheet #1 – Writing a Java Program

Programming with Java: *First Step*

<u>Level 1 - Introductory</u>

_____Step 1. Open the IDE that was installed.

_____a. Lookup “Create a Class” in IntelliJ

_____b. What did you find and how do you create a class?

____

____

_____Step 2. Now it is time to enter a program.

a. Many constructs are the same as in C++ (most built-in types, conditionals, loops, comments, \n)

b. For output, strings can be concatenated together with the + symbol:

System.out.println(“Welcome to CS202’s Java Workshop\n” + “Enjoy!”);

_____Step 3. Set up the Demo1 class:

(The arguments for main are optional but recommended)

```
public class Demo1 {
     /*Send out a welcome message, getting started with Java! */
    public static void main(String[] args) {
        // Let's get started with Demo #1
        System.out.println("Welcome to CS202's Java Workshop\n" + "Enjoy!");
    }
}
```

_____Step 4. Compile and run in the console window:

_____a. Right Click **Run As > Java Application**

_____b. Or, you can use “Run” from the toolbar

Proficiency Scale after Worksheet #1:

Worksheet #1 Completed (TCSS Initials):______

CS202: Programming in Java
Worksheet #2 – Fundamental Constructs

Getting used to Writing Java Programs

_____Step 1. Based on what you know about C++, let's experience creating locals, conditionals, loops, fields, constructors, and methods.

a. **System.out.println(variable);** //will output the contents of your variable
//(and newline)

b. **System.out.print(variable);** //does the same without a newline added

c. **System.out.print(variable + "\t");** // adds a tab after each value

d. **System.out.println();** //will output a newline only (cout <<endl;)

_____Step 2. Write a program to output all odd numbers from 1-100. You can use System.out.println to output a number.

_____Step 3. Revise your program to only output 10 numbers per line

_____Step 4. Now, add conditionals in this to only output those odd numbers if they are not evenly divisible by 3 or 5

_____Step 5. Compile and run in the console window:

a. Right Click **Run As > Java Application**

b. Or, you can use "Run" from the toolbar

Proficiency Scale after Worksheet #2:

Worksheet #2 Completed (TCSS Initials):______

CS202: Programming in Java
Worksheet #3 – User Interaction

Receiving Input from the User

Level 2 - Intermediate

_____Step 1. There are many approaches for performing I/O (at the byte level, character, buffered, scanning, etc.). In this lab, we will use objects of type Scanner – which are useful for breaking down formatted input into tokens and translating those tokens according to their data type.

_____Step 2. First you will need to import from the Scanner utility.

import java.io.*;
import java.util.Scanner;

_____Step 3. The next step is to create an object of type Scanner in main to start with. As you progress, these may be placed as fields in a base class or utility class.

protected Scanner *input*=*null*; //Place this as a field
input = new Scanner(System.in); //Place in your class constructor

_____Step 4. Now we can use the object "input" to read in any type of data, using:

a. **input.nextInt()**; //returns the next whole number
b. **input**.nextFloat(); //returns the next floating point value
c. **input**.next (); //returns the next single word "string"
d. **input**.nextLine(); //returns the next phrase (string) ended in newline

_____Step 5. Just like C++, you need to be aware of the newlines in the input stream being scanned and make sure to appropriately ignore them:

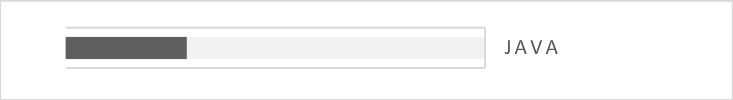
a. **input**.nextLine(); //returns the next phrase (string) ended in newline

_____Step 6. Write a Java function to read in your name, age, and hourly wage (e.g., 13.50/hour)

a. Make String, int and float as locals of main.
   - Remember a **String** (capitalized) is a class, so to create an object of type String: **String name = new String();**
   - **Remember the parentheses are required for constructor invocation of the String class**
   - Remember that all variables must be <u>initialized</u> in Java before they are used

b. Right click **Run As > Java Application;** or, you can use "Run" (toolbar)

Proficiency Scale after Worksheet #3:



Worksheet #3 Completed (TCSS Initials):______

CS202: Programming in Java
Worksheet #4 – Classes with fields and methods

Creating new *Classes*

_____Step 1. The next step is to incorporate this into a class with fields (i.e., data members) and methods (i.e., member functions)

_____Step 2. Create a class called Person **File > New > Class**

class Person {
} ← *Remember, no semicolon here!*

_____Step 3. Place the name, age and wage as protected fields:

a. Remember the public, private or protected qualifier goes before each member
b. Place fields first – before methods
c. When accessing them in a method, it is common practice to place "this." In front of each: **this.name = null; *//remember null is lower case!***

_____Step 4. Create four methods:

a. Constructor to initialize the fields to their values to zero
b. Input – to prompt and read in the values
c. Display – to display the contents of the fields
d. Equals – to compare a name (String) passed in as an argument to the field and return true if there is a match

_____Step 5. To compare strings, you can't use the == operation (this would compare the address in the String references). Instead your choices are:

a. **this**.name.equals(match) //returns true if they are the same value
b. **this.**name.equalsIgnoreCase(source) //same…but ignores case
c. **this.**name.compareTo(source_name) //returns the same as strcmp
d. **this.**name.compareToIgnoreCase(source) //ignores case while comparing

_____Step 6. Lastly, you will need to create an object of type Person in the main to call these functions:

a. **Person object = new Person();** //don't forget the use of new or the ()

Proficiency Scale after Worksheet #4:

Worksheet #4 Completed (TCSS Initials):______

CS202: Programming in Java
Worksheet #5 – Working with Arrays

Work with Arrays

_____Step 1. Remember that when we work with arrays, we are working with references

_____Step 2. To create an array of integers you first would need to create a reference:
int array[]; or, int [] array;

_____Step 3. To allocate memory for the array itself, it is much as we have been doing with C++'s dynamically allocated memory:
array = new int[some_size];

_____Step 4. Arrays have a length fields, which provides the size of the array (NOT the number of items in the array). If you want to know how many items are actually in the array, you would still need to keep a count or check for a null reference

a. **array.length** is the array's length
b. The first element of an array is index zero
c. The last element of an array is index size-1

_____Step 5. Add to your Person class an array of grades – where the size of the array is determined from user input

_____Step 6. Add to your Person class an array of Strings – for each grade's name (e.g., "Midterm", "Assignment 1", etc.).

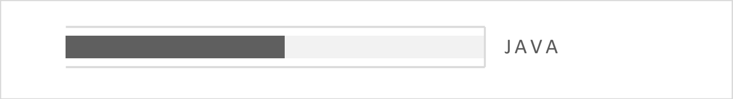
a. **String array[];** creates now a reference that can point to an array of strings
b. **array = new String[some_size];** creates an array of references (i.e., pointers)
c. Next, write a for loop to allocate the correct number of actual string objects!

_____Step 7. Challenge: Place the Grade items in a new class called "Grades"

a. For the Scanner object to be available one option is to create a utility class and derive all classes from it
b. Or, create a local in the desired function:

Scanner input = new Scanner(System.*in*);

Proficiency Scale after Worksheet #5:



Worksheet #5 Completed (TCSS Initials):______

CS202: Programming in Java
Worksheet #6 – Arrays of Class Types

Arrays of Class Objects

<u>Level 3 - Proficient</u>

_____Step 1. In worksheet #5 we saw the use of arrays of integers and arrays of Strings, but let's take this to the next level and create an array of People, dynamically allocated (with names, age, wages, and grades)

_____Step 2. Just like with strings, it will begin with a reference – that can refer to the first element of the array once it is allocated:
Class_type Array_name []; or, Class_type [] Array_name;

_____Step 3. Then, we go the next step to allocate the actual ***array of references*** dynamically at run time: *****<u>Think POINTERS!</u>** ***

Array_name = new Class_type[some_size]; *//we don't yet have objects!*

_____Step 4. To allocate an object, you do so via:

a. **Array_name[index] = new Class_type();**
b. Or, Array_name[index] = new Class_type(*<u>arguments_to_the_constructor</u>*);

_____Step 5. Now write a function called by main to interactively allow the user to enter in as many people's information as they desire, up to a maximum limit established at first:

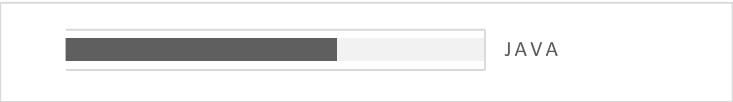
a. First ask for the maximum number of people
b. Then, in a loop continue to read in people's information
c. Stop when the user wants to quit

_____Step 6. Capitalize the first letter of the name and grade name

a. String_object.toUpperCase(); //will upper case an entire string
b. But, to convert a single character requires converting the String object into an array of characters!
char [] array = **String_object**.toCharArray();
c. array[0] = Character.*toUpperCase*(array[0]); //will cap a single character
d. Then, you have to convert it back to a String:
String_object = **new** String(array);

Proficiency Scale after Worksheet #6:

Worksheet #6
Completed (TCSS Initials):_______



CS202: Programming in Java
Worksheet #7 – Linked Lists and Nodes

Programming Linked Data Structures

_____Step 1. If we wanted to work with a linear linked list we have two choices

a. We can create a Node class
b. Or, we can integrate the next or link pointers as part of the List itself

_____Step 2. First, let's create a Node class. You have a choice. You can place the Person data within the node (with a "has a" relationship) or extend the Node from a Person, adding a next ***reference***

a. Create setnext and getnext public methods
b. Setnext should take a Node reference as an argument
c. Getnext should return a Node reference

_____Step 3. Next, create a Generic_List class, that an Array_List and a Linked_List can both be derived from, using dynamic binding.

a. Here is the syntax for creating an abstract class:

abstract class Generic_List extends Utils { //class members go here }

b. Members in an abstract class may be regular members or abstract members. Abstract members are just like C++'s pure virtual functions except they look like: **abstract return_type function_name(arg_list);**

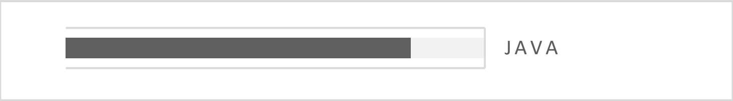
c. Then, we can use the abstract class by deriving other types of lists from it:

class Array_List **extends Generic_List** { //members }
class Linked_List **extends Generic_List** { //members }

_____Step 4. Implement the following functions in these classes:

a. **void Build_List()** // read and add it to the list
b. **void Display_List()** // display the contents of the list
c. **boolean Find(String match);** // find out if a name is in the list

Proficiency Scale after Worksheet #7:



LAB #7 Completed
ID Scanned on Completion (TCSS Initials):__________

CS202 Lab #7 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

Java

1. List 5 constructs that are **similar** between C++ and Java:
    1.
    2.
    3.
    4.
    5.

2. Explain 5 constructs that are **different** between C++ and Java – explain how they are different
    1.
    2.
    3.
    4.
    5.

3. Let's say you wanted to create an array of 10 integers in Java. Show the code

4. Now, show how you can create an array of 10 tree objects in Java. Show the code

5. Write the code to remove the last node in a LLL (in Java)

6. What if we had an array of linear linked lists. Show the code for making a complete copy iteratively:

CS202
Programming Systems

Lab #8: Recursion in Java

CS202 Lab #8: Goals

Important!

IF POSSIBLE BRING YOUR OWN LAPTOP to Lab #8 so that you can learn how to work with the IDE that mirrors what you will do with the last programming assignments.

Goals:

1. **Start today with the linux and vim manual** for the first *10 minutes*. Learn more about using vim and linux tools

2. **Remember** from CS163 (Data Structures) how to work with recursive algorithms when pass by reference is not available:

   a. C – Pass by pointer
   b. Java and C – Returning the altered pointer; the key with this approach is to make sure to use the returned value when calling a recursive function!

3. **Continue** to use either Eclipse or Idea have been installed on the computer you will be using.

4. **Experience implementing recursive algorithms** with Java

5. **As time permits**, begin practicing your data structure algorithms

MAKE SURE TO HAVE
a working LLL Implementation
in Java
prior to Attending Lab #8

CS202 Lab #8 – Pre-lab Worksheet
Recursion with Java

This worksheet must be completed to attend the Lab

The purpose of this pre-lab is to get used to using Recursion with Java. The complexity comes since Java does not support pass by reference. The arguments are passed by value which means they cannot be directly influenced (changes to the arguments will not affect the calling routines argument). Instead the arguments are local with an initial value of what was passed in. There are two possible solutions: (a) return the new values rather than modifying the argument list or (b) create a container for the item that is expected to be affected (such as a class to hold the root or head references). **Assume in the following that node is a class with standard private data and set/get functions for the next:**

_____Step 1. Write the code in C++ to reverse the order of a LLL, recursively:
int reverse(node * & head)

_____Step 2. Now modify this in C++ to return the new head, using pass by value:
node * reverse(node * head)

_____Step 3. **IMPORTANT!**

IMPLEMENT a <u>build</u> LLL function in Java

_____a. Build a LLL, recursively. First design the method:
Wrapper: public void build()
Recursive function: protected node build(node head)

_____b. Write the code in Java to build a LLL, recursively:

_____Step 4. Apply this now to Java. Assume that node is a class with standard set/get functions:

_____a. Reverse the order of a LLL, recursively. First design the method:
Wrapper: public void reverse()
Recursive function: protected ______ reverse(_________)

_____b. Write the code in Java to reverse the order of a LLL, recursively:

Pre-lab #8 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #8 - Recursion with Java

Once the worksheets for Lab #8 are completed, experience recursion with Java by extending the code in the last Java worksheet (worksheet #7) where you built and displayed a linear linked list.

Level 1 - Introductory

_____Step 1. **Implement the build function from the prelab, recursively**

_____Step 2. **RECURSIVELY, count the number of times the last item appears in the linear linked list:**

_____a. Header: **__________ num_times(____________________)**
Return the number of times found

_____b. Plan the code before writing it using these questions:

i. What is the simple (base) case?_________________________

ii. What needs to get done before going to that next smaller sub-problem?____

iii. Show the recursive call: _______________________________

iv. What happens to the returned value (the count)?

____

Proficiency Scale after Step #1:

Level 2 - Intermediate

_____Step 3. **RECURSVIELY, remove the first and the last item in the LLL**

_____a. Recursive function header:

_____remove_first_last(__________)

_____b. Plan the code before writing it using these questions:

i. What is the simple (base) case? __________________________

ii. What should the wrapper function do?_____________________

iii. What needs to happen instead of using delete?

____

iv. Will you do this work before or after the recursive call?_______

v. Show the recursive call: _______________________________

vi. If you didn’t use look ahead, how can you use the returned value to your advantage?__________________________________

Proficiency Scale after Step #2:

_____Step 4. **Remove all but the last two items in a linear linked list**

_____a. Recursive function header:

__________ remove_except(_________________)

_____b. Plan the code before writing it using these questions:

i. What is the simple (base) case? ______________________

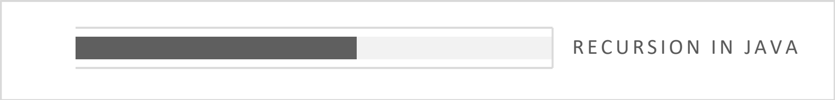
ii. Will you do the work before or after the recursive call?_______

iii. Show the recursive call: _______________________________

iv. If you didn't use look ahead, how can you use the returned value to your advantage?

____

Proficiency Scale after Step #3:



Level 3 - Proficient

_____Step 5. **Make a complete copy of a linear linked list**

_____a. Recursive function header:

________ copy_all(________________)

_____b. Plan the code before writing it using these questions:

i. What is the simple (base) case? _________________________

ii. Will you do the copying before or after the recursive call?_______

iii. Show the recursive call: _______________________________

iv. If you didn't use look ahead, how can you use the returned value to your advantage?

____

Proficiency Scale after Step #4:

LAB #8 Completed
ID Scanned on Completion (TCSS Initials):________

CS202 Lab #8 – BST Practice Questions

Work on these after finishing the lab or to practice

Start reviewing BST algorithms; solve these recursively. Implement these in Java.

For each of these recursive Java questions, keep the following in consideration:

- Double check the base case
- Is a wrapper needed to perform non-repetitive parts of the problem?
- Is the returned value properly being used when the function is called?
- Is the appropriate information being returned?
- Is the data structure correct after the function is called?

1. Begin by creating the insert function for a BST, returning the new root as the returned value

2. Count the number of data items in a BST and return the count
3. Count the number of nodes that have 2 children in a BST and return the count
4. Calculate the average of all of the data in a BST. Think about what your return value should be. Do you need another class to assist?

5. Remove the smallest item in a BST
6. Remove the smallest two items in a BST
7. Remove the smallest item but only if it isn't a 2; return true if the operation took place

8. Copy the contents of a BST and store it into a new LLL. Watch the order!
9. Copy every node with only a right child and store it in a new LLL.
10. Copy every leaf in a BST, forming a new BST
11. Duplicate every item in a BST except the root.

CS202 Lab #8 – Self-Check Quiz

Test to see if you are ready with this material!

Perform the following individually and "closed book, closed notes". This will indicate if you have learned the necessary knowledge and skills from the lectures, readings, and lab:

1. RECURSIVELY, write the code to remove the last node in a LLL (in Java) ; return the head reference

   Use this recursive function header: protected node remove_last(node head)

2. RECURSIVELY, write the code to make a complete copy of a binary search tree. Use the returned type for the new copy

   First design the recursive function header:

   protected ____________ copy (_________________)

   Now implement the function:

3. How would the above function header change if you were to do this in C++ using pass by reference:

   ____________ copy (__________________________)

4. How would the function header change if you were to simulate pass by reference

   ____________ copy (__________________________)

CS202
Programming Systems

Lab #9: Parallel Programming

CS202 Lab #9: Goals

Important!

Make sure to spend time reading the background information for this lab.

Goals:

1. **Always start with the group activities.** With this lab, the group activities will give you experience with parallel programming concepts from an "unplugged" perspective. Spend between 10-15 minutes on these activities.

2. **After this work on the linux and vim manual** for the next *10 minutes*. Learn more about using vim and linux tools

3. **Plan** to experience parallel programming using OpenMP with C++. Make sure to read the background information. There are also links in this material to some websites that have more detailed information. Parallel programming ties nicely with data structures.

4. **Implement** parallel algorithms on the cs202 lab system. We will begin with code that has already been written where you can experience the constructors of OpenMP. Then, we will end with you implementing a parallel program independently! ***THIS MATERIAL FOR PEOPLE WITH NO EXPERIENCE with parallel programming concepts

The following materials are for students who took CS163 in the Summer or prior to 2019

Handouts will be provided for students already experienced in parallel programming!

Algorithms: Thinking in Parallel

Background Information

Introduction to Parallel Concepts

Most computers have more than one processor. A computer with more than one processor can process more than one operation at the same time, where more than one operation at a time should result 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. Astonishing!

In the earliest computers, only one task could be processed at a time. 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 and learn a new way of programming.

The goal is to perform more work simultaneously by dividing the problem up. The idea is to break down the problem into components and/or regions that can be effectively processed at the same time without affecting each other. Maybe we can perform two tasks at the same time or even more. Maybe we can have multiple processors perform the same task on different regions of the data. In CS163, we want to begin by learning how to recognize such situations and experience how to apply some of these concepts to syntax.

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 core, data dependencies, and/or 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 a pasta dish, such as 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 to another, where there may be dependencies that one task must complete before proceeding to the next. In fact, we may need to wait until the first task is completed by all threads before continuing to the next. 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 think 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. Determine what information can be shared versus local to each thread
4. Group these together
5. Determine if there are tasks that depend on others (pipelining, data dependency)

In summary, you now have been introduced to:

11. Threads
12. Concurrency
13. Speedup
14. Synchronization
15. Amdahl's Law
16. Parallel Overhead
17. Pipelining (Producer-Consumer)
18. Data Parallel
19. Task Parallel
20. Data Race

On D2L, please read the paper "**How to Design a Parallel Program**" from *Mike Rogers, Sheikh Ghafoor, and Mark Boshart* from Tennessee Tech. This will give you the background needed to proceed with this lab.

Using C++ to Perform Parallel Programming

To work with multiple processes in parallel in C++, one of the easiest ways to use a threading library for C/C++ called OpenMP. OpenMP was designed to simplify the process of managing threads and allow the programmer to maintain their emphasis on problem solving versus the complexities of parallel programming. With OpenMP we will indicate which code is to be done in parallel by using compiler directives throughout the code. This will look foreign at first but the more you program with it, the easier it will become. The idea of introducing this now is that you will have the ability to work at a deeper level when reaching the 3xx and 4xx level courses.

In C and C++, we must #include another library called `omp.h`. Then, to control when we fork into threads and when we come back together we will use a compiler directive called `#pragma` to assist. we can use a compiler directive that starts with the pound sign (#) and is followed by the keyword `pragma`. Then, we add specific keywords at the end of the pragma directives to determine what behavior we are requesting.

Here is a list of the most common directives we will be using. Then refer to the next section where they are used in a sample problem. These directives may be combined together. A complete list of directives can be found at **www.openmp.org:**

1. `#pragma omp parallel num_threads(N)`

   Explicitly specify the number of threads to execute with a number inside the parentheses. The number can be a constant, a variable, or an expression. This will create a parallel region that will be executed by multiple threads and should be followed by a compound block ({…}). This region must contain a standard block of code and may not span multiple routines or source code files.

   The threads are numbers 0 through N-1 with thread zero being the **master** thread. After the end of the parallel block, only the master thread will continue.

   If any thread terminates within this region, then all threads will end. This means it is important to used structured programming concepts and make sure to normally process through a block without the use of break, return or exit from within the parallel region.

2. `#pragma omp parallel private(variable names)`

Adding private specifies which variables are going to be local for each thread where they will work with their own copy. The list of variables must already be defined and can be a comma separated list. This must be followed by a compound block ({…}) that represents the block where these private variables will be local. Alternatively, these can be defined inside of a region as local and therefore the keyword private can be avoided.

3. `#pragma omp parallel shared(variable names)`

Adding shared specifies which variables are going to be shared among each thread; when you select this, make sure there won't be a data race which means you don't want to be simultaneously changing the same memory by different threads. This must be followed by a compound block ({…}) that represents the block where these variables will be shared. This can be combined with private and num_threads. Variables defined prior to a parallel region will automatically be considered to be shared. Or you can specifically request this via default(shared):

```
#pragma omp parallel private(count) shared(array) num_threads(4)

Or,
#pragma omp parallel private(count) defaut(shared) num_threads(4)
```

4. `#pragma omp critical`

We can lock sections of the code that will only be executed by one thread, requiring all other threads to be completed before the critical section is executed. `Critical` means that only one thread enters the block at a time.

5. `Getting the thread number`

You can perform certain operations just on a given thread by using the function omp_get_thread_num() that will return an integer and then we can use an if statement based on that thread number; we will tend to use them when partitioning data into regions:

```
int thread_num = omp_get_thread_num();
if (thread_num == 0)  cout << "This is the master thread\n";
```

The following program outputs each thread number. Locking the output statement is important. If more than one thread can be performing the output at the same time, then the end line may not always be processed before the next thread number appears. Try out the following code with and without the critical block to see how that works!

```
//Output each thread number for each
#include <iostream>
#include "omp.h"
using namespace std;

int main()
{
  #pragma omp parallel num_threads(4)
  {
      #pragma omp critical    //this is important otherwise some thread nums
                              //will be displayed on the same line!!
      cout << omp_get_thread_num() <<endl;
  }
  return 0;
}
```

6. `#pragma omp for`

This allows us to execute a loop in parallel, where the for loops will implicitly partition the data into regions based on the number of threads. Without a parallel for, we would have to split the data into the regions explicitly. We will gain experiencing doing this in lab. This directive must be immediately followed by a for loop

To compile we type:

```
g++ -fopenmp  *.cpp
```

Solving Problems using Parallel Programming

Now let's take a problem and solve it using parallel programming concepts. Let's count the number of matching data items in an array of items. This problem using iteration is rather straightforward. All that is necessary is to use a loop to iterate through each element, starting at index zero and stopping when we reach the last element used (e.g., we need to know how many data items are stored in the array). However, by sequentially processing through the data structure we will have the worst case run time performance. By applying parallel processing techniques we can realize parallel speedup.

Remember we are not wanting to go until the SIZE of the array because in that case we may be comparing garbage and risking an ultimate seg fault or run time error. Now let's plan on how to solve this using parallel programming:

1. **How many tasks can be done in parallel (partitioning them)**
   The number of tasks in this case might best match the number of processors that we have, not to exceed the number of elements in the array. To have two threads, then we will divide the array in half and work on comparing and counting the two parts in parallel. If we want four threads, then we could divide the array in quarters and work on all four comparing and counting at the same time. **Amdahl's law** tells us that too many threads for this problem would lead to more overhead rather than realizing the expected speedups in runtime performance. We will only experience **speedup** when dealing with extremely large quantities of data. For small arrays, processing the data serially will most likely result in better performance due to **parallel overhead**!

2. **Look for the patterns**
   With parallel programming, we want to take this problem and find the patterns. We know that for each element we will be doing the exact same operations: (a) compare the data for a match and (b) if there is a match, increment a counter. Once we are done, the counter variable will hold the result that we are interested in. Since this is the same set of tasks being performed on different regions of the data, we will experience the process of **data parallel** work as multiple processors are doing the same computation but on different pieces of data.

   The process is also called **domain decomposition**, also sometimes called **data decomposition**; the idea is to process the data in the array in parallel. Domain decomposition requires dividing the array into equal parts and assigning each part to a processor.

3. **Determine what information can be shared versus private (to a thread)**
   There are four variables that are used in this problem: (1) the array of information, (2) the number of items in this array, (3) the data that we looking to match, and (4) the counter. We can definitely access the array at the same time in different regions without causing problems with synchronization. The array's contents is being accessed but not modified, so we won't experience any issues so we can share the array between threads. This also applies to the number of items in the array and the data we are looking for. All of these variables are not being modified and may be shared among all threads. However, we can't use a shared counter variable between each of the threads because it would lead to **data race** conditions. If we tried to change the same variable at the same time by multiple different threads, we could not be guaranteed that the correct count would be achieved. Each time we run the program we could get different results. So, the counter needs to be managed as a local or private to a thread. This means we will need the same number of counters as we have threads.

   Consider the simple example in Figure 1. The computation occurs in two phases. First, the work is divided equally among the three processors. In this case, each processor compares the data in the array and increments a counter (accum1, accum2, or accum3).

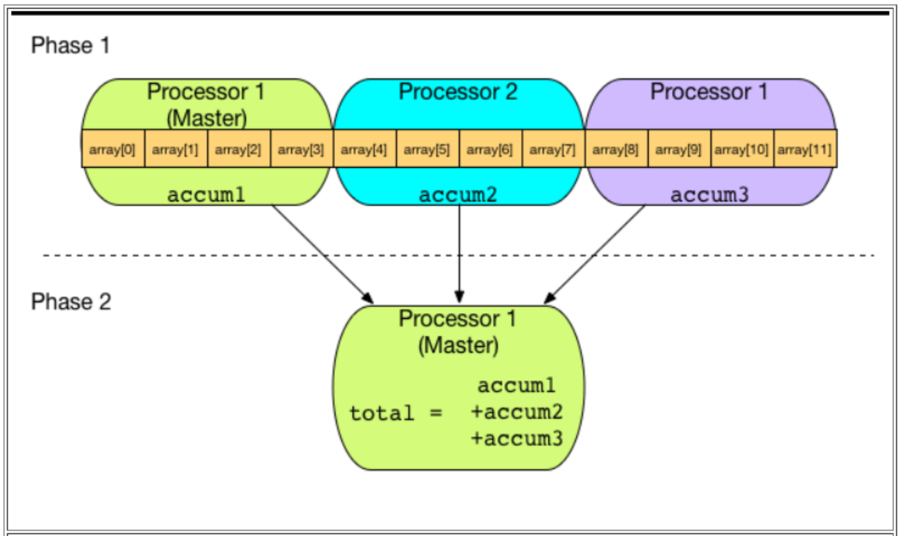


Figure 1: Three processors computing the sum in parallel

(Figure 1: How to Design a Parallel Program Mike Rogers, Sheikh Ghafoor, and Mark Boshart Tennessee Tech)

4. **Group these together**
   We know that if we were to separate out the tasks to compare and count, then when we are done counting each part of the array we would need to come back together and sum all of the counters together. This tells us that each thread will want to have local counters that keep their own tally's and then afterward we will need to sum them together. This last action will show that there is some overhead in managing a parallel solution. In a serial solution, we wouldn't need to create a grand total of all of local counters – we would already have that total in hand; this contributes to the **parallel overhead**.

   Let's look at Figure 1 again. The master processor sums all accumulators once. This is an example of **sequential dependency** where the global count cannot be computed before completing each of the local tasks.

5. **Determine if there are tasks that depend on others (pipelining, data dependency)**

   In this example, we need to complete counting all of array partitions prior to summing the counters. This means that we need to work on synchronizing the processes. We need to lock the process of summing the counters until all threads have been completed.

Using OpenMP to Solve this problem

To solve the problem using serial design would be simple but with the performance of a sequential search:

```
int counter = 0;

for (int i = 0; i < num_data; ++i)
{
       if (array[i] == match)
               ++counter;
}
cout << "The total count is: " <<counter <<endl;
```

To solve this problem using parallel design requires some parallel overhead to set up the problem. In this case we would need to break-up the array into the number of threads desired. So, if there were N threads, then we would need to divide the number of data items by the number of threads to determine the range of data for each core to process:

```
//Parallel overhead - setting everything up to be in parallel
    int N = 4;      //Number of threads
    int range = num_data/N;
```

Based on the parallel design, when we setup up the parallel section, we would want to specify the counter as private (local for each thread) but share the array and the variable that would hold the master counter:

```
int counter = 0;
int master_counter = 0;

#pragma omp parallel private(counter) shared (array, match,
master_counter) num_threads(N)
```

Or, we could say instead:

```
#pragma omp parallel private(counter) num_threads(N)
```

Then, inside the parallel part of the program, we would set up the beginning and ending indices based on the thread number.

```
    int first_index = omp_get_thread_num() * range;
    int last_index = first_index + range;

    for (int i = first_index; i < last_index; ++i)
    {
        if (array[i] == match)
            ++counter;
    }
```

And, lastly we would want to lock when the master_counter gets modified so that we don't experience a data race. This would be done by using `#pragma omp critical`

```
#pragma omp critical
{
     master_counter += counter;  //now sum the value
}
```

Putting it all together gives us the following code:

```
#pragma omp parallel private(counter) num_threads(N)
{
    int first_index = omp_get_thread_num() * range;
    int last_index = first_index + range;

    for (int i = first_index; i < last_index && i < SIZE; ++i)
    {
        if (array[i] == match)
            ++counter;
    }
    #pragma omp critical
    {
      cout << omp_get_thread_num() <<" " <<counter <<endl;
      master_counter += counter;  //now sum the value
    }
}
```

If we wanted access to each of the threads counts after the parallel portion is over, we would need to create a shared array of those integer values (creating an array of size N dynamically) that can be accessed after the parallel portion of the work has been completed.

Another approach is to use the `parallel for`, which is great for dividing up the work within an array implicitly.

```
#pragma omp parallel num_threads(N)
  {
    int counter = 0;  //local for each thread

    #pragma omp for
    for (int i = 0; i < num_data; ++i)
    {
        if (array[i] == match)
            ++counter;
    }
    #pragma omp critical
    {
      cout << omp_get_thread_num() <<" " <<counter <<endl;
      master_counter += counter;  //now sum the value
    }
  }
```

CS202 Lab #9 - Pre-lab
Parallel Programming Concepts

This pre-lab must be completed and brought to the lab to attend.

Pre-lab #9 – Parallel Programming with Data Structures

_____1. **Data Parallel** examples. List two different data structures that would work well with threads working on the data in parallel

a. ____

b. ____

_____2. **Data Parallel** examples. List two different data structures that would **not** work well with threads working on the data in parallel

a. ____

b. ____

_____3. **Amdahl's Law** examples. Let's say we have an array of 1000 elements where each element has a linear linked list. All of the nodes need to be updated. At what point will splitting up the work not make sense. Discuss your thoughts:

____

____

____

Will the length of the linear linked lists change your opinion? Discuss your thoughts:

____

____

____

_____4. **Data Race** examples. Discuss issues that we face if we wanted to display the contents of the array of linear linked list in parallel. What issues would we face?

____

____

____

____

_____5. **Data Dependency** examples. What if we wanted to sum all of the values in an array of linear linked lists in parallel.

_____a. Are there any data dependencies? Does one set of tasks need to be completed before we can move on to the next?

____

____

____

____

_____b. What issues with data dependency will we be faced with:

____

____

____

____

Pre-lab #9 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #9 –Parallel Programming

Group Activity – Spend between 10-15 minutes on these activities

Level 1 – Introductory

In this activity we will experience data partitioning where we will specify what each "thread" will be working on, parallel overhead in setting up the problem and serial versus parallel tasks.

_____Step 1. You will be given a bag of colored widgets. Your job will be to work as a group to sort, count and determine which colored widget we have the most quantity. With multiple people working in parallel, we imagine that speedup will be experienced but we will also experience the need to partition the widgets for people to work with and collect the data from each. **Group sizes should be between 3-5 people.**

Data Parallel: Start with each person performing the same task of sorting and counting on a smaller set of the items

_____a. **Before you begin, write down the time:_________________**

_____b. Can we figure out which color of widget is greatest prior to everyone finishing counting?________ If not, then we will be experiencing pipelining where the parallel tasks must be completed before we can continue with the next task.

_____c. How long did it take before everyone was done counting their task?_________

_____d. Did you have to wait for one (or more) people to finish before moving on to the next step:___________

_____e. Once everyone finished counting, how did you handle finding the maximum? Could this be done in parallel?___________

_____f. Or, will the tasks of sorting, counting, finding the maximum be divided between different people? (task parallel)_______(yes,no)

_____g. Were people waiting much without anything to do (this would be like a processor being idle)?___________

_____h. **When you end, write down the time:_________________**

Level 2 – Intermediate

_____Step 2. **Task Parallel:** Start over and do the sorting and counting again. But, this time give different tasks to the group members. In this case, we will be doing different tasks at the same time. One person should sort, another count, and the third find the maximum.

_____a. **Before you begin, write down the time:_________________**

_____b. At what point can the person counting begin?

____

_____c. Did you experience data dependency where we had to wait for one task to be completed before continuing to the next step:

____

_____d. Did you have to wait for one (or more) people to finish before moving on to the next step:___________

_____e. Were people waiting much without anything to do (this would be like a processor being idle)?___________

_____f. **When you end, write down the time:_________________**

Level 3 - Proficient

_____Step 3. Discuss among your group the results of the unplugged activity.

_____a. What benefits did you experience?_______________________

_____b. What limitations did you experience?_____________________

_____c. Did you experience parallel overhead?___________________

_____d. If we have 1,000 students, would the process have been faster?

____

_____e. Was there another method that could have been used to speedup the sorting and counting process?

____

****** RECEIVE feedback from Lab assistants before continuing with the linux & vim exercises***

Experiencing Parallel Programming

THE FOLLOWING LAB Material
is for students who did not take CS163 at PSU

Students who have already performed this lab should request a handout for advanced problems

We will now experience programing parallel concepts with arrays. We will begin by examining code that has already been written and is provided in the cs202lab system Lab5 directory. The concepts that you will experience will include the following. Then, you will implement a parallel program independently:

1. Data partitioning
2. Data parallel
3. Data dependency
4. Pipelining (Master-Client)
5. Shared data
6. Private data

Level 1 Introductory

_____Step 4. Login to **cs202lab.cs.pdx.edu** using your assigned login and password

_____a. Change into the CS202/Lab9 directory **cd CS202/Lab9**

_____b. Use a **linux editor** such as **vi, vim, or emacs** to type in a program.

_____Step 5. We will begin by examining code that uses OpenMP to perform parallel process in C++

a. Open the file named **CS202_sum.cpp**

b. What new library is #included?______________________________

c. Data Partitioning: To partition the data, look at the following variables and explain what they are used for:

i. range:____

ii. first _index:______________________________________

iii. last_index:_______________________________________

d. What value will the first_index be for the different thread numbers:

i. Thread 0: _______________

ii. Thread 1:_______________

iii. Thread 2:_______________

iv. Thread 3:_______________

e. What value will the last_index be for the different thread numbers:

i. Thread 0: _______________
ii. Thread 1:_______________
iii. Thread 2:_______________
iv. Thread 3:_______________

_____Step 6. Compile and run the program. `g++ -fopenmp CS202_sum.cpp *.o`

a. What results do you get?______

b. Edit the program and experiment with a different number of threads. Compile and run again:

Number of threads__________ Result before_____ Result after ________
Number of threads__________ Result before_____ Result after ________
Number of threads__________ Result before_____ Result after ________

Level 2 Intermediate

_____Step 7. **Edit the program** and comment out **#pragma omp critical**

a. Compile and run the program.

```
g++ -fopenmp CS202_sum.cpp *.o
```

b. What happens with the output?__________________________

c. **Remove the comment syntax (putting back the critical section!)**

_____Step 8. Now, do this again but with a `parallel for` construct in OpenMP. Add the following loop after the results are displayed to see if we get the same results with the parallel for:

```
//Start over but now with a parallel for
total_sum = 0;
#pragma omp parallel private (sum) num_threads(N)
{
   sum = 0;
   #pragma omp for  //this will partition the data
   for (int i = 0; i < num_data; ++i)
      sum += array[i];

   #pragma omp critical
      total_sum += sum;
}
cout << “The total sum is: " <<total_sum <<endl;
```

a. Do you get the same result?______________

_____Step 9. Now your for loop has been properly parallelized with the parallel for!

In our example, each processor must have its own accumulator, and then the master processor must sum the individual accumulators together to get a final total. Combining the results of the computations is called a *reduction* is parallel programming. Fortunately, OpenMP will handle this operation for you as well. You just need to tell OpenMP what variable to reduce and what operation needs to be applied. Therefore, add the following line to your sum function directly before the for loop.

a. Begin by copying the file to a new file name:

```
cp  cs202_sum.cpp   cs202_parallel_for.cpp
```

b. Now use the new file to make the following changes.

c. When using the parallel for with reduction, we no longer need the critical section to accumulate the total_sum. We also don't need the code to separate out Therefore, comment out the line in the critical section to be as following (you may also delete these lines of code. However, don't delete code if you are working with your original cs163_sum.cpp file):

```
/*#pragma omp critical
{
    //let's see what the final count value is for each thread locally
    cout << omp_get_thread_num() <<" " <<sum <<endl;
    //total_sum += sum;
} */
```

d. Now change the for loop to have the reduction keyword as shown below:

```
#pragma omp parallel num_threads(N)
{
#pragma omp for reduction (+:total_sum)
       for (int i = 0; i < num_data; ++i)
            total_sum += array[i];

}
```

_____Step 10. Or we can now combine this all together and just have the following for loop. Give it a try!

```
#pragma omp parallel for reduction (+:total_sum) num_threads(N)
        for (int i = 0; i < num_data; ++i)
                total_sum += array[i];
```

a. Comment out or delete the other for loops and #pragma directives. Note that the above line of code is valid if you named your accumulator variable total_sum. If you named your variable something else, which you probably did, then change the line to use your variable name instead of total_sum.

b. Compile your code and run it to make sure it works.

Level 3 Proficient

_____Step 11. **In the previous** program, there would have been a **data race** if we had not used a different variable to hold the master total. If we had just used one sum variable that all threads worked with, we would have received the wrong answer. This is because by the time we would have retrieve the array element and retrieved the value of sum to add them together and store it back into sum, another thread could have modified that same variable. Experiment with this to convince yourself of why experiencing data race would have been a problem. Summarize your thoughts:

____

_____Step 12. Write **your own** program using OpenMP to find the largest data item in the array of integers. You may start with the CS202_sum.cpp file and copy it over to a new file: `cp CS202_sum.cpp CS202_max.cpp`

_____Step 13. Start by planning the changes that you will need to make:

a. What data should be shared?___________________________________

b. What data should be private to each thread?______________________

c. Are you initializing the private data correctly?___________________

d. How will you handle having each thread have its own max?

____

_____Step 14. Write the loop here (using the parallel for) to find the maximum for each thread:

____

_____Step 15. What will be done in a critical section?

____

_____Step 16. Now, edit the CS202_max.cpp file to change the process from summing the data to finding the max.

a. Summarize what you learned about this process

____

_____Step 17. **Tar the files** for the lab by typing:

```
tar   -cvf    CS202_Lab9.tar   *.cpp   *.h
```

_____Step 18. **Submit** your program by typing **./submit** at the linux prompt

LAB #9 Completed
ID Scanned on Completion (TCSS Initials):___________

CS202 Lab #9– Practice Questions

Work on these after finishing the lab or to practice

Level 1 – Introductory

1. Explain why a program to output the contents of an array's data would not work well with parallel programming
2. Be able to detect data dependency from a real world example
   a. An airline flight with multiple stops

      ____

   b. A factory line for a truck or car assembly

      ____

   c. Taking classes at college with a pre-req structure

      ____

   d. Getting food for lunch from a buffet

      ____

Level 2 – Intermediate

3. Write a program using OpenMP to count the number of positive values in an array
4. Write a program using OpenMP to count the number of even values in an array
5. Write a program using OpenMP to average all of the data in an array

Level 3 – Proficient

6. Explain an example where data race would be an issue.
7. Write a program using OpenMP to sum all of the data in an array of linear linked lists.

   a. What part of this would work well in parallel? _________________________

   b. Which part is inherently a serial problem?______________________________

CS202
Programming Systems

Lab #10: BST Review

CS202 Lab #10: Goals

Important!

It is time to prepare for the final proficiency demo. Make sure to practice BST algorithms daily!

Goals:

1. **Start with the linux and vim manual** for the first *10 minutes*. Learn more about using vim and linux tools. Make sure you are confident with vim navigation, vim search and replace, and gdb. All of these concepts will be part of your proficiency demo!

2. **Remember** from CS163 (Data Structures) binary search tree algorithms:
    a. Height – is the longest path from root to the farthest leaf
    b. A leaf is a node that has no children
    c. A full tree has every leaf at the height of the tree and any node that is not a leaf has two children (for a BST)
    d. Data that is less, is placed on the left; data that is greater than or equal is placed on the right
    e. The inorder successor is the right child's farthest left node and this is used when deleting a node with two children.
    f. The smallest item in the tree is all the way left
    g. The largest item in the tree is all the way right
    h. The smallest item may have a right subtree
    i. The largest item may have a left subtree

3. **Experience implementing recursive BST algorithms** in C++ to prepare for the final proficiency demo

4. **Review** the guidelines for the final proficiency demo

USE THE Linux/vim Exercises to PREPARE

for the final proficiency demo!

CS202 Lab #10 – Background Information

Preparing for the Final

1. Our final proficiency demo will be on traversing and/or manipulating binary search trees. A tree will be created and it will be your responsibility to write a function that is randomly assigned to work with the binary search tree.

2. Remember when working with binary search trees to be aware of the following:
   - Never dereference a null pointer
   - Remember the -> dereferences the first operand (the pointer BEFORE the operator). This means that root->data will dereference the head pointer and access the data member
   - Now that we are back using C++, remember the . operator does NOT dereference the first operand (the pointer). So when accessing data through a root pointer we would say: root->data

3. Get familiar drawing pointer diagrams
   - Use them to plan the algorithm
   - Draw "before" and "after" pictures for each and every special case
   - Remember, always handle the case of the pointer being NULL
   - Whenever root gets affected (by adding or removing from the tree), it will need to be passed by reference.

4. **Common mistakes** to avoid
   - **Never have two returns in a row** within the same scope. Once we "return" from a function, we cannot continue processing to the next
   - **Make sure to have a returned value for a value returning function**. Check that each path through the function returns the appropriate information. We don't want garbage to be returned. You may think the answer is correct, but it may not be!!
   - **Make sure to "use" the returned value** when calling a function that returns a value.
   - **Avoid physical traversal**. Instead use the power of recursion calling the function with the next smaller sub tree.
   - **Never** access memory that has already been deallocated

5. **Practice recursion daily**!
   - Start easy and work through the cs202lab.cs.pdx.edu CS202/fpdemo directory to practice
   - Always plan what the public wrapper function will do (the non-repetitive tasks) and the private recursive function (the repetitive tasks)
   - Avoid multiple traversals if it is not needed to solve the problem!

vim in Review

On your final proficiency demo you will be expected to show fluency navigating with vim (without using the arrow keys!), searching for information, and search and replace. You will also be expected to demonstrate the use of gdb!

1. vim Tips (Background)

a.	^z	-Places vim into the background
b.	jobs	-At command line. Shows which jobs are in the background
c.	fg	-At command line. Brings most recent job to the foreground
d.	fg 3	-At command line. Bring job 3 to the foreground

2. Other Important Commands

a. If you want to delete the line just added, type **dd**

b. If you made a mistake and didn't want to delete, you can undo it by typing **u**

c. Single characters can be deleted with the **x** key. Try it!

d. To search and replace the first occurrence of text type:

:s/search text/replacement text/

e. Undo it by typing **u**

f. Now replace all occurrences by typing:

:%s/search text/replacement text/g

g. Undo it by typing **u**

CS202 Pre-Lab #10 – Review of Data Structures

This worksheet must be completed to attend the Lab

Assume that you have a BST of integers. You are given a root pointer to a node that holds an integer and a left and right pointer. *List the special cases to consider, and think about what wrapper functions would be required to work with recursive solutions.*

_____Step 1. **Count the number of nodes in a BST with values greater than 100**

i. Special cases:

ii. Wrapper functions

iii. Function prototype(s) in C++

iv. Function header(s) in Java

_____Step 2. **Traverse the tree to determine if it is a full tree**

i. Special cases:

ii. Wrapper functions

iii. Function prototype(s) in C++

iv. Function header(s) in Java

_____Step 3. **Remove the largest item from a BST**

i. Special cases:

ii. Wrapper functions

iii. Function prototype(s) in C++

iv. Function header(s) in Java

_____Step 4. **Explain where the second largest item could be in a BST:**

_____Step 5. **Explain where the root's inorder successor exists in a BST:**

_____Step 6. **Preparation for Proficiency Demos:**

_____a. How do you search for the word NULL with vim_______________

_____b. Show how to move down 10 lines from the current position in vim:_______

_____c. How do you compile a C++ program in order to use gdb?________

_____d. Show how to set a breakpoint:______________________________

_____e. What does **xx** do?: ______________

_____f. What does **u** do?: ______________

_____g. What does :**s/NULL/0/** do?: _________________________________

_____h. What does :%**s/NULL/0/g** do?: ______________________________

_____i. What does **q!** do?: ______________

Pre-lab #10 Completed
Checked in-Lab (TCSS Initials):________

CS202 Lab #10 – Data Structures Review

Check-off each step as you proceed! Submit the code at the end of lab.

Collaboratively Implement BST Algorithms

With this lab we will be working with an existing class which contains a binary search tree of integers. The root is a private data pointer. There are public wrapper functions and private recursive functions. Your job will be to implement functions to review for the proficiency demonstration. Make sure to use recursion**! Y**ou have access to the .h files to see what functions are available. *Assume that you have a BST of integers*

Login to **cs202lab.cs.pdx.edu** using your assigned login and password

- Change into the CS202/Lab10 directory **cd CS202/Lab10**
- Use a **linux editor** such as **vi, vim, or emacs** to type in your recursive functions
- **Compile** and link to my object code on **linux**. **g++ *.cpp *.o -g -Wall**
- **Always fix warnings** that are received by using the -Wall
- Run **(./a.out)** your program by typing: **./a.out**

Implementing BST Algorithms – Collaborative Process:

A. Have each team member **implement each of the functions assigned in this lab**,

B. **Remember** to plan the code before writing it using these questions:

   i. What is the simple (base) case?
   ii. Separate out the non-repetitive vs. repetitive parts of the problem
   iii. What needs to get done before going to that next smaller sub-problem? And, will temporaries be needed
   iv. What is the incremental step to get to the next smaller sub-problem (i.e., how do you use the recursive call to get to the next smaller sub-problem)
   v. Are you using the returned value when calling a recursive function?
   vi. What needs to get done after returning from that smaller sub-problem?

C. Each team member should develop the test plan for their function

D. Each team member should evaluate their work with a pointer diagram, showing each invocation

E. Once implemented, **compare the code and test plan** with others in your group

F. **Evaluate** what worked and what didn't

G. **Evaluate** the test plan. Did it cover all of the necessary cases?

H. **Summarize** what was learned.

Do not use any loops in this lab!

Begin implementing the member functions, in a .cpp file. Complete them here in **cs202_lab10.cpp** file. Implement the repetitive functions recursively.

Level 1 - Introductory

_____Step 1. Recursively count the number of nodes in a BST, **excluding the root and any node that matches the root's data**; return the count

_____a. What is the simple (base) case? ______________________

_____b. What should the wrapper function do to set up the problem?

____

_____c. What needs to get done before going to that next smaller sub-problem?

____

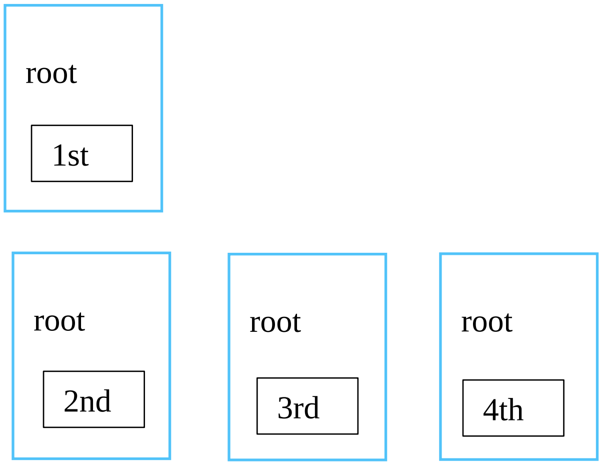
_____d. What is the incremental step to get to the next smaller sub-problem

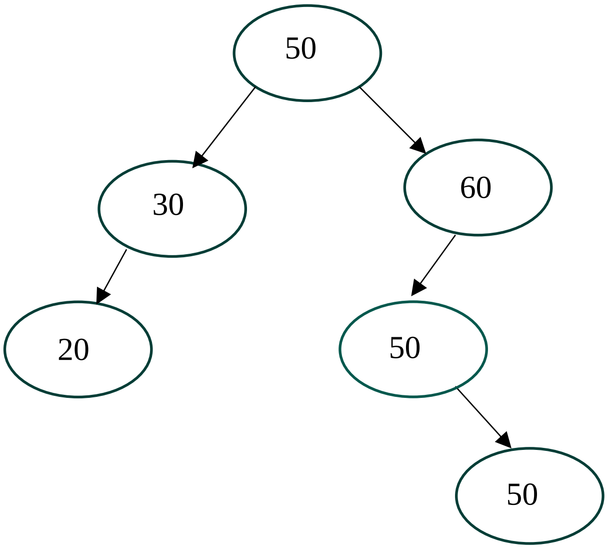
____

_____e. What needs to get done after returning from that smaller sub-problem?

____

_____f. Trace your code with a pointer diagram





CAREFULLY ANSWER these questions!

_____g. **After completing your work,** examine the code and answer the following:

i. Does every path through the function return a value?_____

ii. Are you using the returned value for each recursive call?_______

Proficiency Scale after Step #1:

______Step 2. Count the number of nodes in a BST that exist in the smallest nodes' right subtree. Return the count.

______a. What is the simple (base) case? ______________________

______b. What needs to get done before going to that next smaller sub-problem?

____

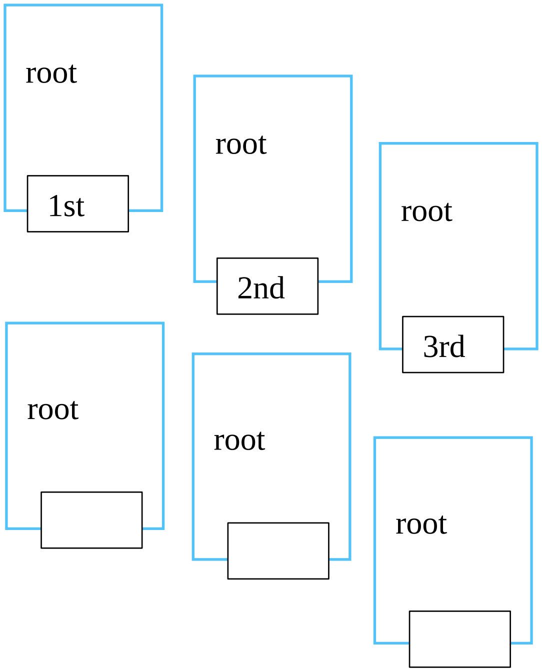
______c. What is the incremental step to get to the next smaller sub-problem

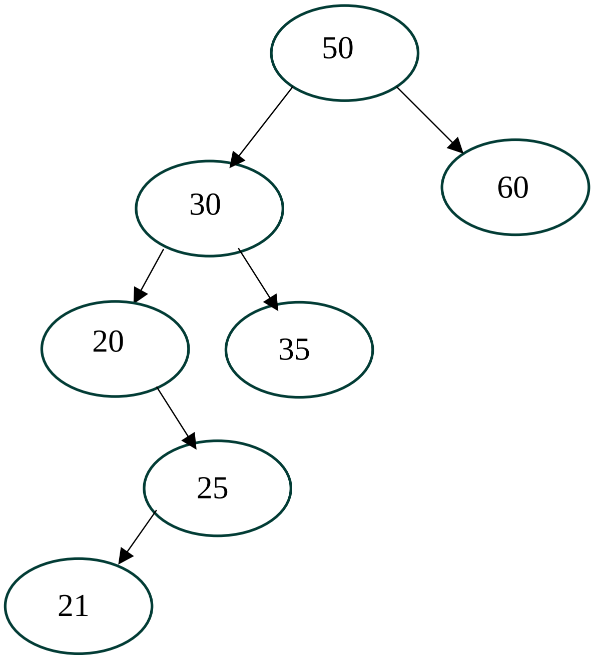
____

______d. What needs to get done after returning from that smaller sub-problem?

____

______e. Trace your code with a pointer diagram





CAREFULLY ANSWER these questions!

______f. **After completing your work,** examine the code and answer the following:

i. Does every path through the function return a value?________

ii. Are you using the returned value for each recursive call?_______

Proficiency Scale after Step #2:

Level 2 - Intermediate

______Step 3. Remove the largest item from a BST; return the data value of the largest

______a. What is the simple (base) case? _________________________

______b. What value should root have (recursively) after deleting the node? _______

______c. If root has no right child, what value is the largest?________________

______d. Is it possible that the largest has a right subtree?____________

______e. Is it possible that the largest has a left subtree?_____________

______f. If the largest value is in the tree multiple times, where would it exist?

____

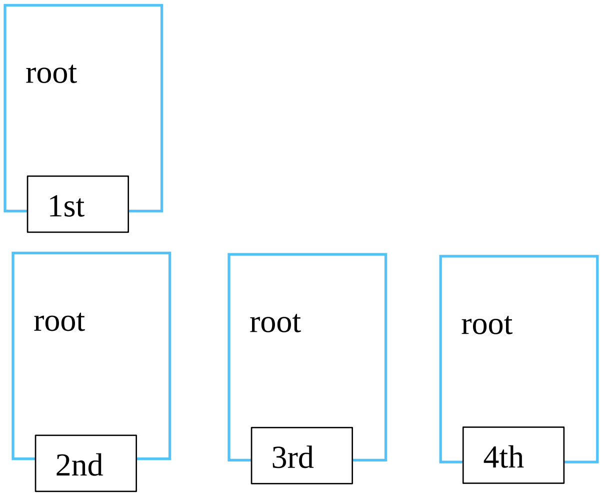
______g. Should the deletion take place before or after the recursive call?

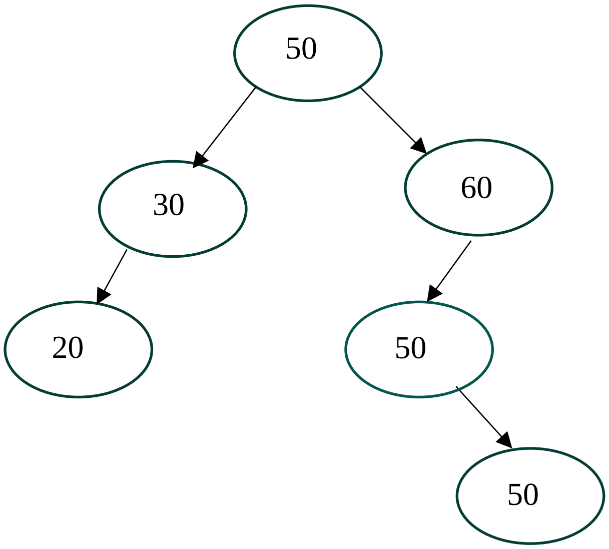
____

______h. Show the recursive call:

____

______i. Trace your code with a pointer diagram





CAREFULLY ANSWER these questions!

______j. **After completing your work,** examine the code and answer the following:

i. Does every path through the function return a value?_________

ii. Are you using the returned value for each recursive call?________

Proficiency Scale after Step #3:

______Step 4. Calculate the sum of all data that is a multiple of 3 or 5 in the BST, returning the average.

______a. What is the simple (base) case? ________________________

______b. What is the non-repetitive part of this problem?_____________________

______c. What is the repetitive part?_______________________________________

______d. Should the sum be computed prior to the recursive call, or after?

____

______e. Which is more efficient?___________________

______f. What operator do you use to determine a "multiple"?

____

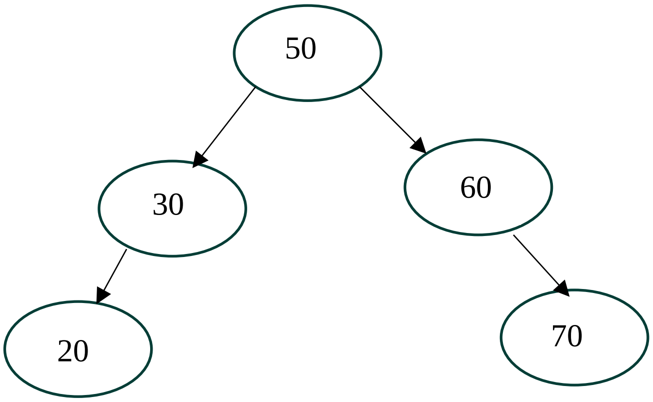
______g. Plan how to compute the average?

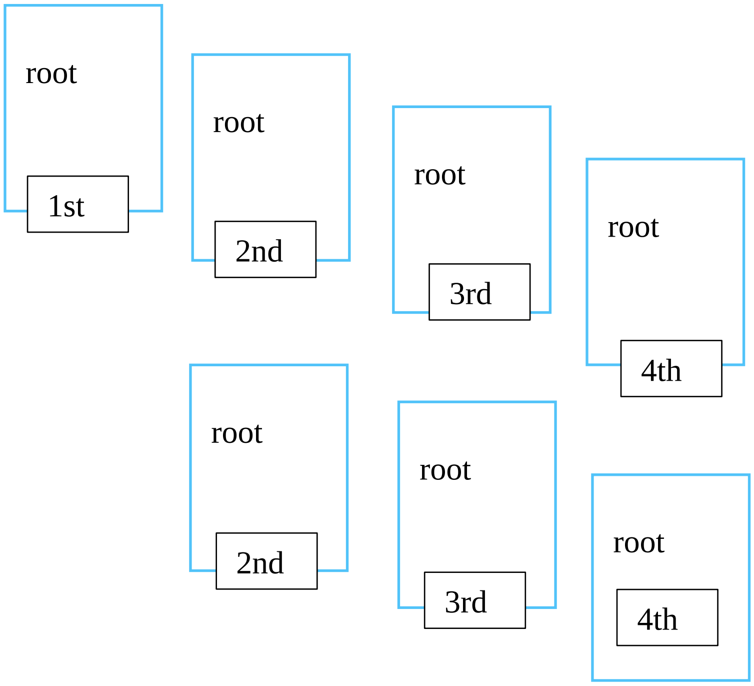
____

______h. Plan out the arguments and return type. DO NOT use a void return type!

________ sum_all (____)

______i. Trace your code with a pointer diagram



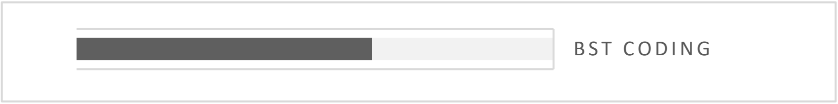


______j. **After completing your work,** examine the code and answer the following:

i. Does every path through the function return a value?_________

ii. Are you using the returned value for each recursive call?________

Proficiency Scale after Step #4:



Level 3 - Proficient

_____Step 5. Make a complete copy of all even data in a BST, creating a new BST

_____a. What is the simple (base) case? ______________________

_____b. In what situations should a new node be allocated?_______________

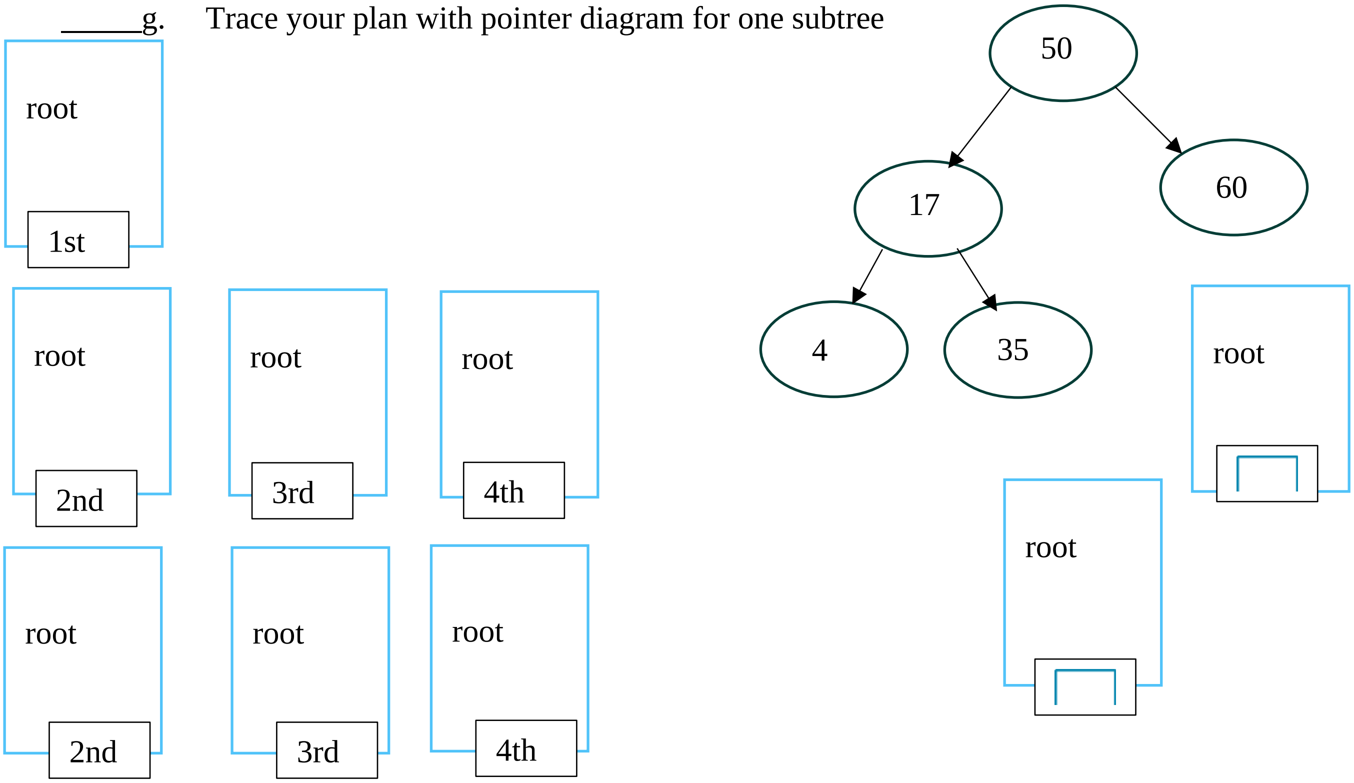
_____c. Show the recursive call when the data is even (for both left and right):

_____d. Show the recursive calls when the data is odd (for both left and right):

_____e. In the above recursive calls, is the destination pointer accurate?_______

_____f. In the above recursive calls, will a NULL pointer be dereferenced?______

_____g. Trace your plan with pointer diagram for one subtree



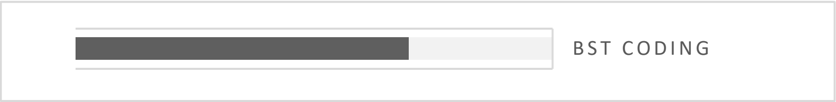
CAREFULLY ANSWER these questions!

_____h. **After completing your work,** examine the code and answer the following:

i. Does every path through the function return a value?________

ii. Are you using the returned value for each recursive call?_______

Proficiency Scale after Step #5:



_____Step 6. **Building Excellence in Problem Solving and Programming.** Complete these in review for the proficiency demo!

_____a. Create a full BST (required for the challenge questions)

_____b. Traverse the tree to determine if it is a full tree *(a full tree is where all leaves are at the height of the tree and every node that is not a leaf has 2 children; refer to the CS163 data structures book for a complete definition)* _____c. Display the largest item in the BST and return it

.

_____d. Display the largest two items in the BST. *Think about this one!*

Proficiency Scale after Step #6:

_____e. **Tar the files** for the lab by typing:
_____f. **tar -cvf CS202_Lab10.tar *.cpp *.h**
_____g. **List the files** in the directory to confirm that the .tar file exists
_____h. **Submit** your program by typing **./submit** at the linux prompt

LAB #10 Completed
ID Scanned on Completion (TCSS Initials):___________

CS202 Lab #10 – Practice Questions

Here are some sample questions to practice **recursively**:

1. Copy the data structure *(Do not delete as part of your answer!)*
    a. Make a copy of a BST, return the number of items copied
    b. Make a copy of all nodes that have no children, return the sum of all of the data copied
    c. Make a copy of the largest two items in the tree, return the average of these items
    d. Make a copy of the root and its inorder successor
    e. Make a copy of every even data item in a BST of integer data. Return the number of items copied

2. Counting values and return the count (Make sure to watch the values being returned!); none of your functions should be void

    a. Count the number of nodes in a BST and return the count
    b. Count the longest path and return the count
    c. Count the number of times the root's data appears in the tree and return the count
    d. Count the number of nodes in the inorder successor's right subtree and return the count
    e. Count the number of nodes in the largest's left subtree and return the count

3. Removal from a tree
    a. Remove the largest item in a BST, return the item removed
    b. Remove all nodes except for the root, return the number of items removed
    c. Remove the root's inorder successor, return the item removed
    d. Remove every node without any children, return the number of items removed
    e. Remove the largest item, return the value of the largest item
    f. Remove the largest two items in a BST, return the sum of these two numbers
    g. Determine if the root's data appears in the tree a second time. Avoid traversing the entire tree!

h. Count the number of duplicates in the tree! Use the power of the traversal algorithms. Avoid creating a separate data structure. Avoid traversing more than once!

4. **Remember to practice the vim commands outlined in this lab manual.**
5. **Practice gdb as well!**

CS202 Final Proficiency Demos

In your proficiency demo, you will be asked to implement a recursive solution for a BST to insert, remove, find, or otherwise modify a node or the data in the data structure. It will be part of a class, so you will be given a public wrapper function prototype and a private recursive function prototype. You may add additional functions but it is important to minimize traversal overhead! Make sure to practice with the code supplied in the final proficiency demo directory of the cs202lab system.

For your proficiency demos, there are two phases. You will be coding a BST algorithm to show fluency in working with recursion and binary search trees in C++. You will also be demonstrating navigation, search and replace as well as other linux editor commands and gdb. Make sure you are fluent using one of the linux editors (vi, vim, or emacs) and gdb.

Proficiency demo guidelines:

1. It is important to come at your appointment time; do not assume that you can "walk-in" at another time unless authorized to do so.

2. Plan to come 10 minutes early to get set up and accustomed to the environment.

3. The technical assistants will check you in. First show them your picture id

4. It is a closed book, close notes event. No additional technology (phones, smart watches, etc.) may be used during the demonstration. Make sure they are safely stowed away.

5. You will be asked to read a document that describes in detail what is expected during the demonstration. It is important to read this to familiarize yourself with the requirements.

6. You may use your own computer or check out one of ours. All applications must be closed prior to coming into the demo if you use your own computer. There must NOT be any code in the clipboard or on the screen that is not generated during the demonstration. In the event that code is on your screen or clipboard, it will be an automatic failure. Upon check-in, the technical attendants will be confirming that your computers are ready to use and free of code being displayed. A hard reboot will be expected. *(Make sure to have already done any system updates prior to the proficiency demo taking place.)*

7. When space becomes available in the testing arena, the next step will be to select a question, randomly. Technical assistants will provide you with a set of questions to select from. Select a question and read it. If you are happy with the question, then you will proceed to the next step. If not, give it back to the technical assistant and select again. Again, if you are happy with this second question, you can proceed. If not, give it back to the technical assistant and select again. The third question "sticks" and there are "no go backs". *(Questions are refreshed periodically.)*

8. Once a question has been selected, it is considered a "closed book, closed notes" session and no further internet access is allowed. Once you have started working on a question, it cannot be replaced. The prototype supplied may not be altered. But you may always write more than a single function to solve the problem.

9. Once you have entered the testing arena and begin working on the problem assigned, keep the following in mind:
   a. You are encouraged to spend time with **design**; scratch paper is provided in the testing arena
   b. You are also allowed to **compile, test, and debug** your work. Please consider performing incremental implementation, compiling code as you go.
   c. Remember when coding, **please comment out code rather than deleting it**
   d. It is expected that your solution will minimize traversals. This means that if you have a question to remove the last node and return the total number of nodes in the list, then it all needs to be done through one traversal. Do not attempt to traverse to remove and then traverse again to count!
   e. You may write **<u>more</u> than the function** supplied if necessary, but main must call the function assigned
   f. The prototype(s) supplied **may not be changed**
   g. For questions that have you count, average, or traverse in some way, your **solution should NOT modify the data structure**
   h. For questions that have you insert or copy, your solution should **NOT delete items**
   i. **Avoid extra traversals** just to calculate the returned value (for example, if the question asks to count or sum the data, that should be accomplished as part of the recursive problem and not through an extra pass through the data set)
   j. **No use** of the **internet** except to login to our testing environment
   k. **No use** of **help** or the **man pages** during the demonstration
   l. **No** other accounts may be logged into during the demonstration
   m. **Only one shell** session may be open

10. Syntax requirements include:
    a. **No global variables** may be used
    b. **No** use of the **string class**
    c. **No** use of **static locals**
    d. **Do not add** data members to the class
    e. All questions will expect a **recursive solution** be used to solve a repetitive problem
    f. **NO LOOPS may be used in your solution**

11. When we have scored your work you will be asked to demonstrate the use of the linux editor (navigation, search, and search/replace) and to demonstrate the use of gdb (breakpoints, displaying variables, listing information about the breakpoints, backtracing, etc). Afterwards, you will be instructed to submit the results using the ./submit script. Please wait until prompted to do so.

CS202 - Lab Manual Self Evaluation Form – Final

Name:___________________ PSU ID:__________

Attended	Self-Evaluation:	For Official use Only
□ Lab 5	Pre-Lab Finished □ Group Activity Finished □ 0% 20% 40% 60% 80% 100%	_____ % Completed (not blank) (readable and relevant)
□ Lab 6	Pre-Lab Finished □ Group Activity Finished □ 0% 20% 40% 60% 80% 100%	_____ % Completed (not blank)
□ Lab 7	Pre-Lab Finished □ Group Activity Finished □ 0% 20% 40% 60% 80% 100%	_____ % Completed (not blank)
□ Lab 8	Pre-Lab Finished □ Group Activity Finished □ 0% 20% 40% 60% 80% 100%	_____ % Completed (not blank)
□ Lab 9	Pre-Lab Finished □ Group Activity Finished □ 0% 20% 40% 60% 80% 100%	_____ % Completed (not blank)
□ Lab 10	Pre-Lab Finished □ Group Activity Finished □ 0% 20% 40% 60% 80% 100%	_____ % Completed (not blank)

□ Linux/Vim Manual	# Labs Completed _____________ Level # _____ (1, 2, 3)	_____ # Labs Completed _____ Level Completed

Calculate Grade	Sum total of all percentages completed__________ Total number of labs completed_________ (5 or 6) Divide the total by # of labs___________% Multiply by 5__________ (0-5 grade)	_____ Total Percentage _____ # Labs Completed _____ Calculated grade