CIS 200 NOTES (W21, with Dr. Jie Shen) – Demetrius E Johnson

Table of Contents

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

CIS 200 NOTES – Demetrius E Johnson

1-13-2021 notes:

- 1) Talked about Syllabus and intro to Unix

2.

- We generally write a computer program using a high-level language. A high-level language is one that is understandable by us, humans. This is called **source code**.

  However, a computer does not understand high-level language. It only understands the program written in **0**'s and **1**'s in binary, called the **machine code**.

  To convert source code into machine code, we use either a **compiler** or an **interpreter**.

  Both compilers and interpreters are used to convert a program written in a high-level language into machine code understood by computers. However, there are differences between how an interpreter and a compiler works.

- **Working of Compiler and Interpreter**

Figure: Compiler

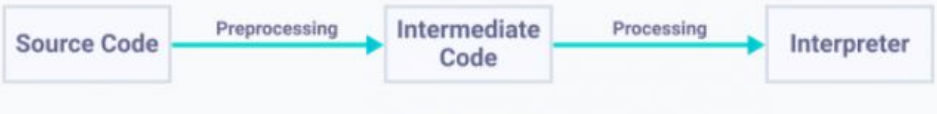


Figure: Interpreter

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

1-20-2021 notes:

- **Interpreter Vs Compiler**

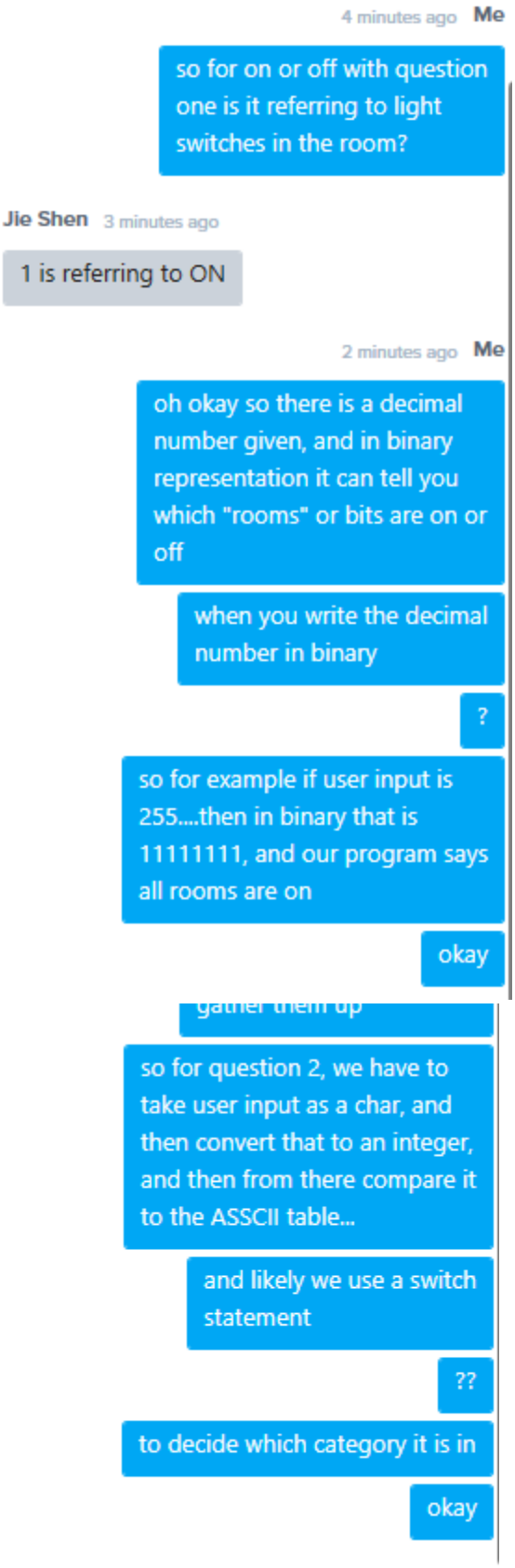
Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

- UNIX REALLY HELPFUL COMMAND: man [name of command]
    i. Man stands for “manual” to show how any command works
    ii. Similar to ? when you are logged into a switch
- Alt+prntscreen prints current window
- NOTE ON MACROS (#define function):
    1. The biggest diff between a macro function and a normal function is that you do not have to specify the data type as long as the data type can handle the operators that you use in the macro function
    2. NOTE ON INLINE specifier (ex: inline int functionname(int a, int b)
        - The purpose of inline function is so that there is less overhead during runtime; every time the function is called there is no need to allocate memory and build the function again, it simply replaces the line of code where the function was used to execute the line of code from the function’s code already defined (it is not called like a normal function is called, it simply replaces the line of code instead so that there is less overhead)→ no need to allocate new memory and then destroy it when the function exits/line of code finishes for the function.
            i. This is also the same purpose of macro functions too except macro function do not need to specify data types

LAB 3: 2-1-21

- Use modulus operator for wrap-around method with Row-Major convection

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



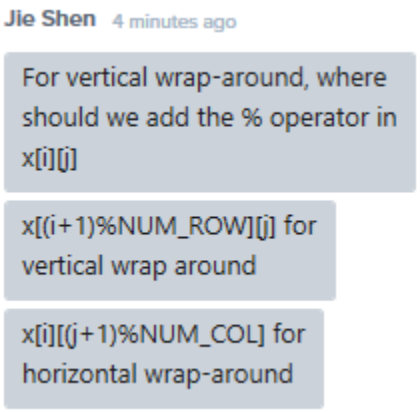
-
-

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

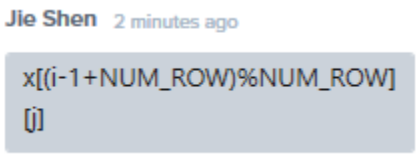
- Question 3: allowed to use whatever methods we want including using classes..etc..as long as we get the correct answer
- For question 3 part 4:
    i. When the element is at the corner, use wrap around for its other 2 adjacent elements (neighbors)
    ii. That's what this statement on the lab is referring to:
    iii. "Hint: You should consider wrap-around when element[i][j] is located at the boundary of the matrix."

2-03-2021 notes:

Wrap around:

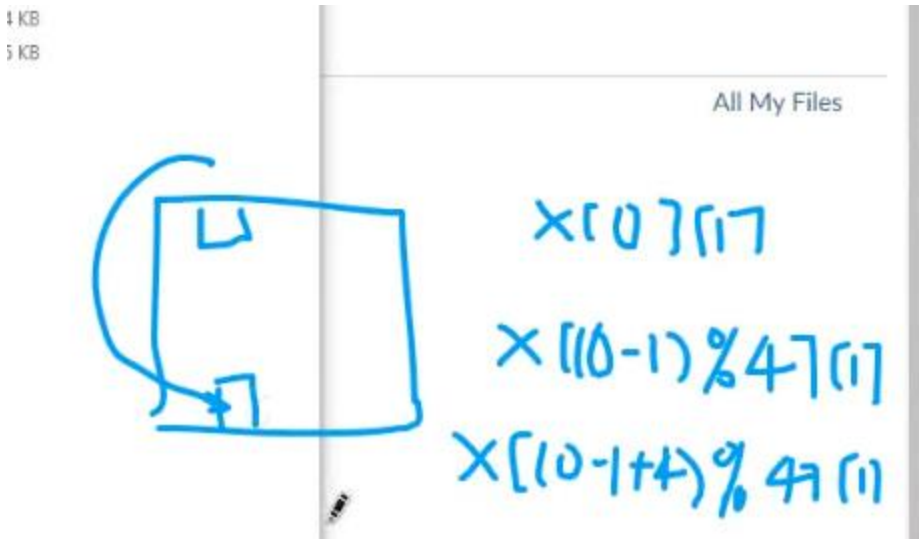


-
    - This question will be on midterm; jie shen asked the question and then the second comment is the solution
    - For example:
        - You do +1 since when you use the modulus operator it returns the remainder; thus for right neighbor → if col == 3, and we do (col+1) % 4 == (3+1) % 4 → 4%4 = 0; thus wrap around is successful: the right-neighbor of col 3 elements will be in row 0.
            - And this works for all row or col values even if you are not at a corner or edge; example: col = 1; right neighbor is (col+1) % 4 → 2 % 4 = 2; 2 is the correct right neighbor of the given element
        - You do the % operator since we can get any remainder and go back to start of the row or column; for example: if num rows = 3 (0, 1, 2), then doing(i+1)%3, to handle cases for when i > NUM_row – 1 (= 2 in this case), then you get 2+1 = 3; 3%3 = 0; so that means 3's neighbor is 0, which is the correct horizontal wraparound for an array with num_row = 3 (positions 0-2).
    - Special case:

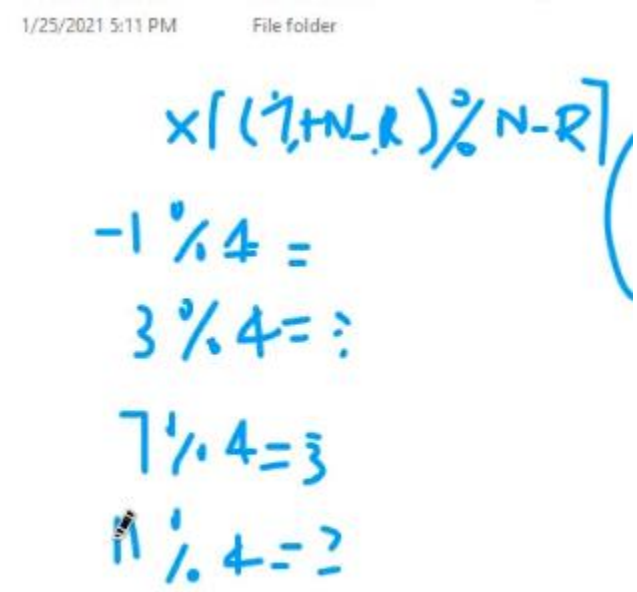


        -

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



- Do this because you cannot have a negative number since modulus operator cannot operate on negative values; so when doing a wrap around avoid using mod operator on negative value



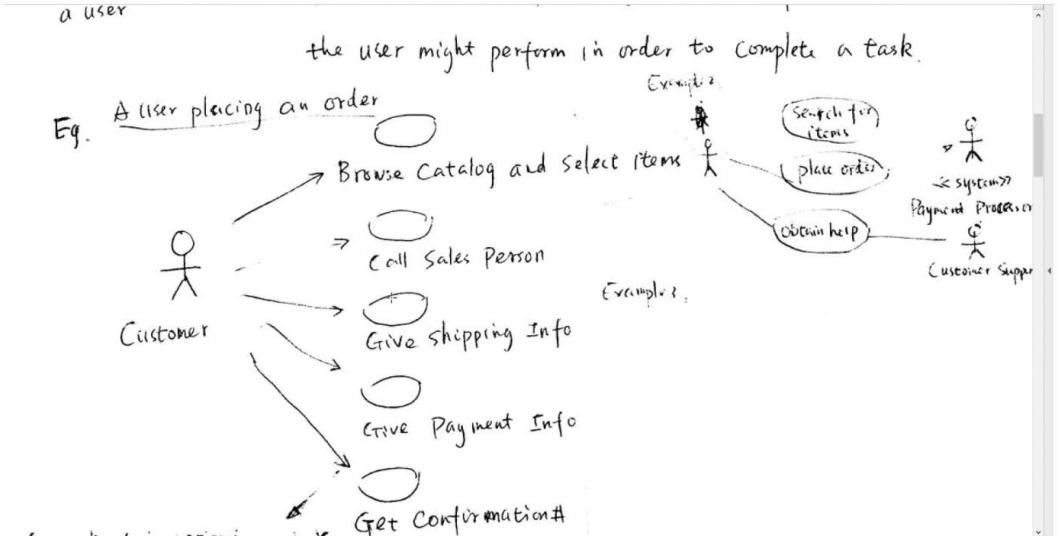
- As shown above; everything is okay except for -1% 4 → not allowed.

- Debugging
  - Remember you can use breakpoints and a debug output file to output debug information to check how functions are operating/if they are functioning properly and as expected
    - For example, you can use cout <<; or you could use debugFile << (output to .txt debug file)
- UML (Untitled Modeling Languages)
  - Types of UML diagrams
    - 1) Use Case Diagram: displays the relationship among actors and use cases

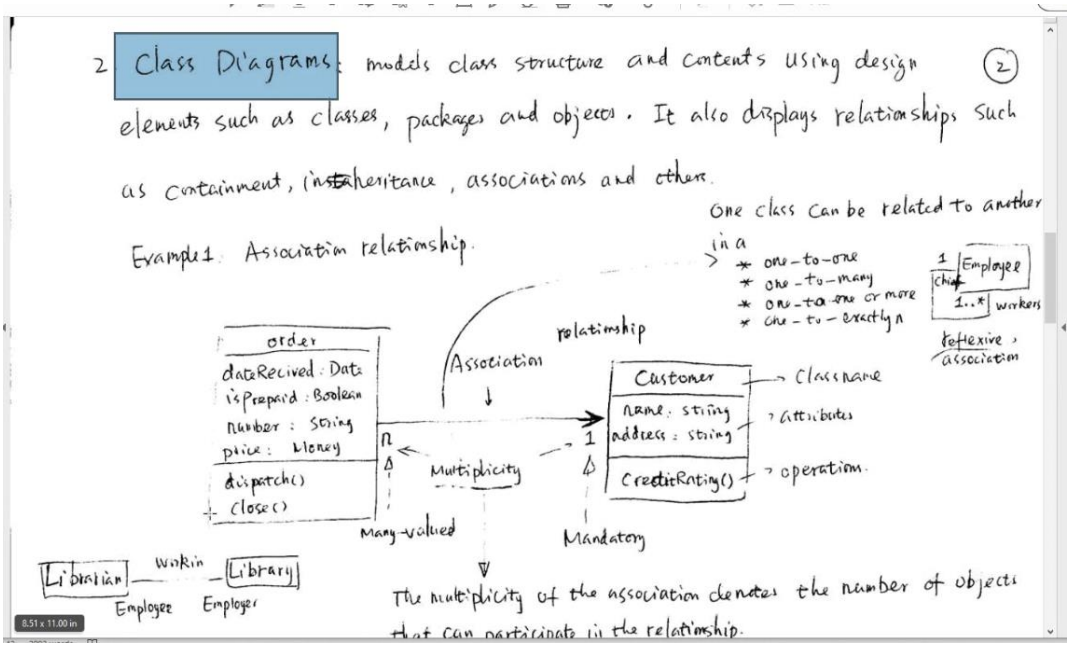
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Cases refers to: a set of scenarios that describe an interaction between a user and a system

- Two main components of case diagram: Actor, and Use Case.
  - Use case = an external view of the system that represent some action the user might perform in order to complete a task
  - Example of a use-case diagram:

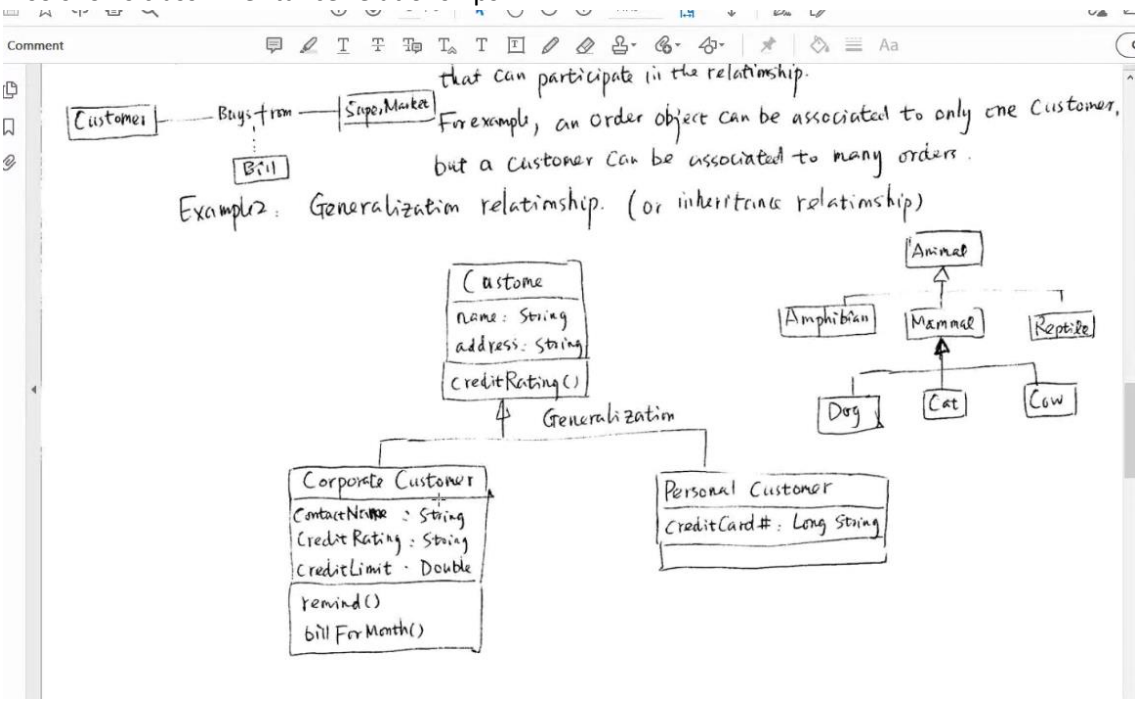


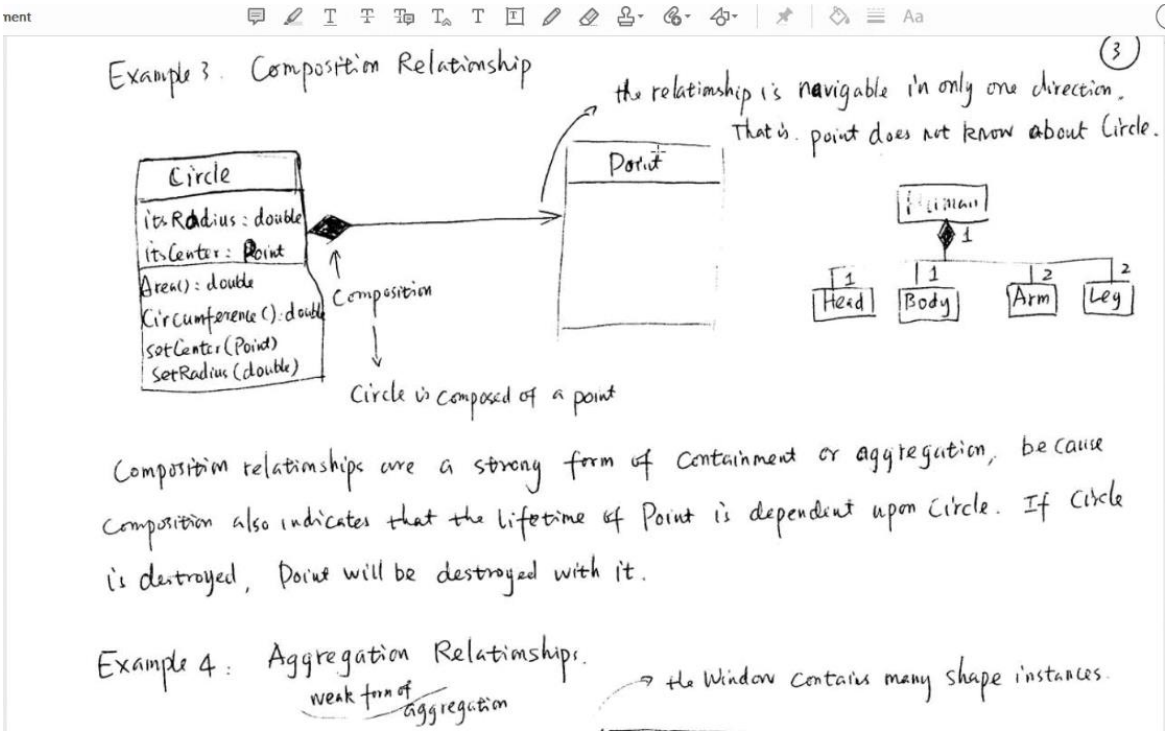
- Gives an idea of how many/types of user in system and which type of action each user can take
- 2) Class Diagrams:



- Helps you to get an idea of how many and which type of classes you want/need for a programming project; and it shows the attributes each program will contain
- Also shows relationships (through edges and arrows) between classes so that you can see the reliability and coordination of classes to perform an overall action desired by the whole program

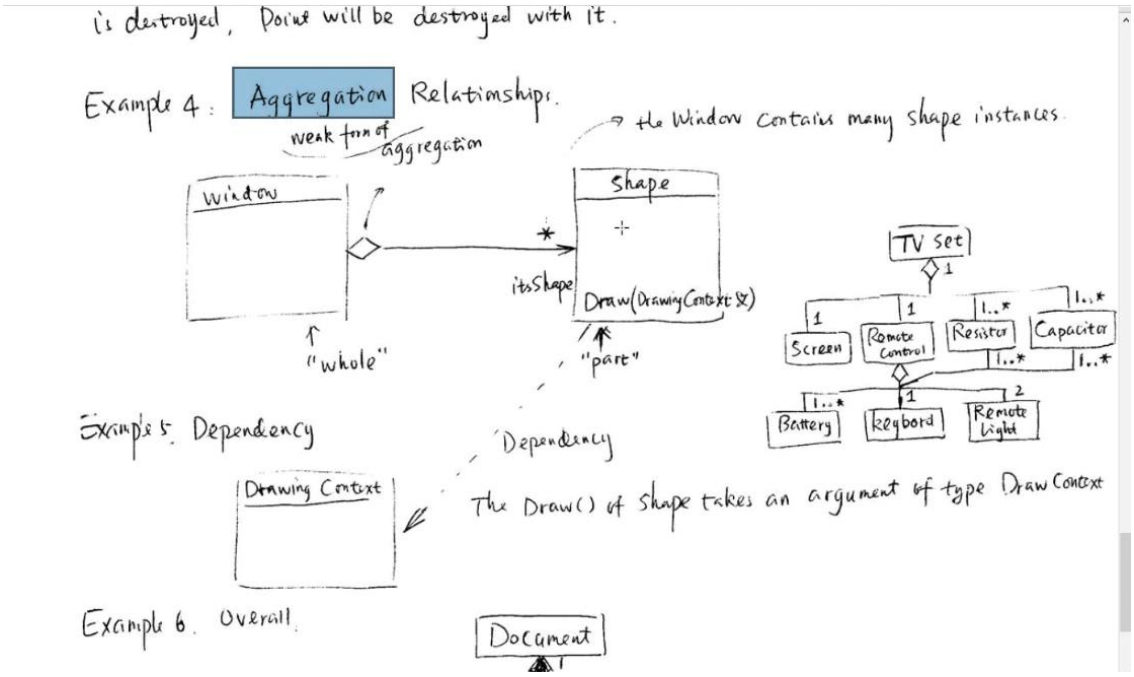
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Also each attribute can be labeled with a multiplicity to show how many times a related variable is used; for example if a variable called “name” is related and maintained and named the same in two different classes.
- Also shows class inheritance relationships
- 

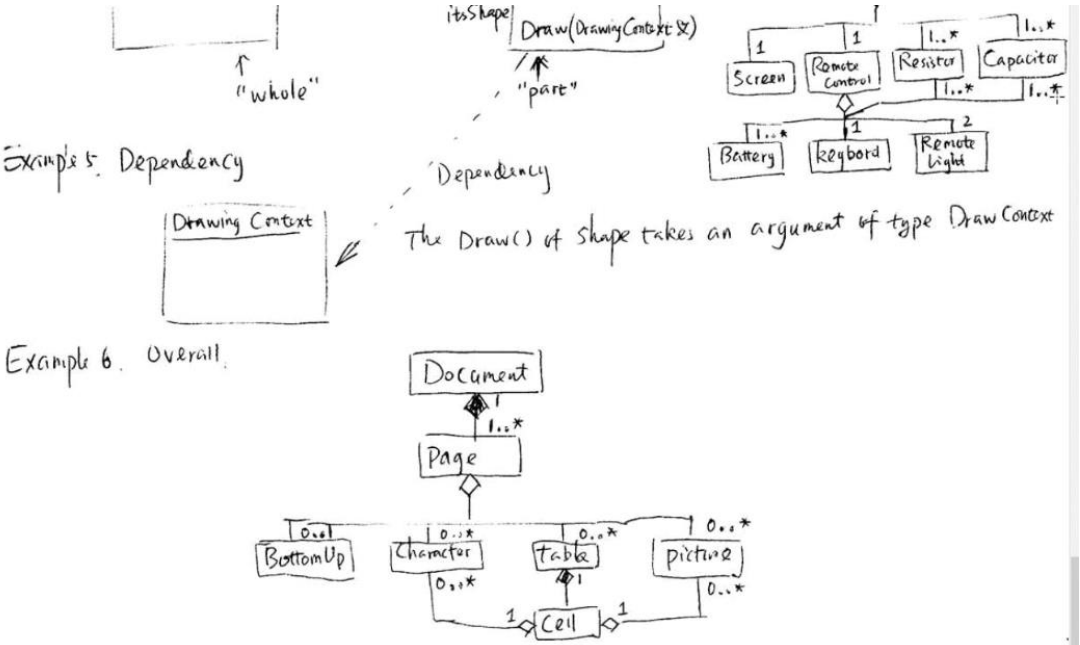
  - Composition relationship:
  - 

    - Notice the multiplicity shows for the human class example; ex: arms → humans have 2 arms…

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



- Hollow diamond: weak composition (means it does not have to contain all of the branches that comes out from it
- Solid diamond: strong composition
- 1…*: 1+ for multiplicity
- Number = multiplicity



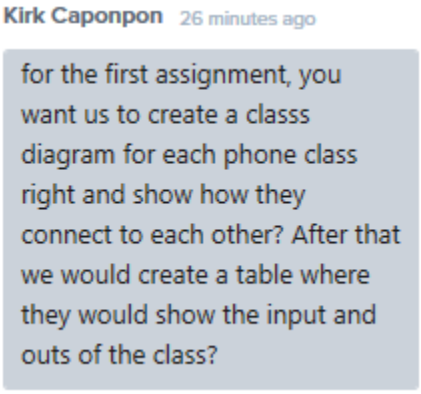
- Robustness → Assertion (useful for debugging)
  - Ability of a program to recover by handling errors; very key because it can help reduce debugging time and reduce errors so that you can more easily find critical or hard-to-find logical errors.
  - Exception handling; assert functions

- Need to use #include <assert.h> to use assert functions
- Example:
  -
```
#include <assert.h>

Int main()
{
    Int x = 10;

    assert(x>5);  // do nothing

    assert(x<9);  // program halts
```
  - So if assert conditional statement is true; nothing happens
  - If false, program halts (similar to a breakpoint); very useful for debugging; particularly for finding logical errors (use it as a watch dog to catch any logical errors and to find out the culprit variable with a bad/undesired value)
    - Good to spread asserts all throughout the program
    - Precondition or postcondition is a good candidate for assert()

```
Float sqrt_fun( float x)
{
    // precondition is a logic condition you want to
    // impose at the beginning of your function

    ....

    // postcondition is a logic condition you want to
    // impose at the end of your function

}
```

    - Make sure that if values are logically good, you set up your asserts so that the program does not halt; set it up so that only if a value will cause a logic error, then assert will evaluate to false and thus halt the program.

- Better to use assert because program won't crash; so that you can have a better chance to find out which variable is causing the problem.



-
  - Professor Jie Shen answer: yes

2-08-2021 notes:

- 3 cornerstones of object-oriented program
  - Data abstraction (data hiding) → classes/structs
  - Inheritance (hierarchical structure among class relationships)
  - Polymorphism "many forms"; with objects having ability to have many forms of itself (from the properties of inheritance)
- Functional Design Modules (functional decomposition)
- Inheritance
  - private makes it so only the class that has the attribute is able to access the variable...protected allows only the next level inheritance class to also access the variable
  - Special case: private can be accessed through a friend function, but protected can't be accessed through a friend function
- Remember you under private, public, and protected:
  - You can have attributes declared and/or functions defined
- Classes
  - If function name matches class name it is a Constructor
    - If no parameters described for the constructor then it is a Default Constructor
    - If there are parameters, it is called parameterized constructor
      - Can have multiple constructors with various parameter types; this is the use of overloading
    - Whenever we create an object (aka instance of a class), c++ calls a constructor to build the called abstract data type (allocate space in memory for all of the attributes)
  - Remember the "this" operator: this-> (is used to distinguish the current object versus any similar object passed into a function; sometimes this is necessary to get rid of

ambiguity, sometimes it is used solely for readability and to make it less likely to write errors in your code)

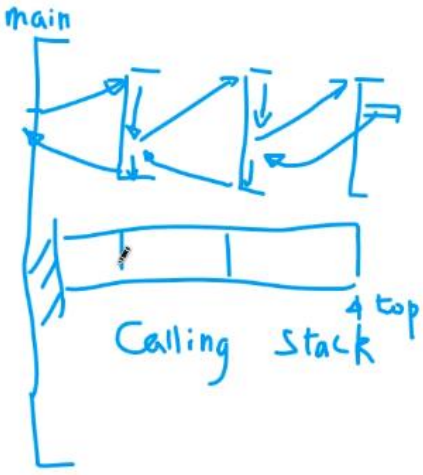
- Reasons for writing a copy constructor
  - *if you no pointers involved and nothing special needs to be done (i.e. a full member-wise copy is needed only) then you do not have to explicitly define the copy constructor; you can allow the program to call the default implicit copy constructor which does full member-wise for every attribute
  - = operator invokes the copy constructer when an object is first created and is equated/assigned the values of a current already created object
    - Otherwise the assignment operator (= operator) is called
  - Copy attributes member-wise
  - Sometimes because a pointer/array attribute is a member of the class object
  - Another reason for copy constructor/writing the copy constructor (instead of the default copy constructor) is the implicit call to a copy constructer when you pass an object by value to a function (remember when passed by value, the function will build its own object of the class object as a copy to be used in the function); again if you don't define the copy constructor yourself and there are pointers involved, this could cause memory leaks and dangling pointers and cause program to crash
    - Even if no pointers involved, sometimes you may want a copy constructor to invoke only specific attributes to be copied, while others may be copied and then manipulated, or just given some other default value (and not copy anything at all from the object to be copied/partially copied)
  - Remember you need to overload the assignment operator (=) for the same reasons you need to write a copy constructor function
  - Remember: don't allow your program to implicitly call the destructor for an object if there is a pointer attribute contained in the object to be destroyed; you must write a destructor yourself to destroy dynamically allocated memory for the pointer (used the "new" operator during the objects construction)
    - If you don't delete the dynamically allocated memory, you will have a memory leak

- Classes
  - Remember we define large member functions outside of the scope of the class for better organization
    - Scope of a class ends after the declaration of the class ends:
      - Class A {//start of scope of class
        …}; //end of scope of class
    - So if you want to continue writing information for a class member function outside of its scope (i.e. a class definition), use the scope operator ::

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

LAB 4: 02-08-2021

- Break points
  - Two types
  - 1) regular break point
    - 3 Options:
    - Step-over: don't get into a function
    - Step-into: get into the function
    - Step-out: if you are already inside a function, you can complete execution lines of the remaining part of function and then get out of the function
  - 2) conditional breakpoint
    - Useful for loops that will iterate many times (large number of iterations)
    - Can set up break points to watch a variable in a loop so that the breakpoint will activate only when a certain variable matches the given condition
      - For example, if at iteration 999 there is an issue, you can set up a break point in the loop to watch the counter variable (or any other variable for some condition) so that then the program will hault at the conditional breakpoint location so that you can begin manually stepping through the program/function however you choose
  - For Visual Studio: Right-click the breakpoint and click on "condition" option, select "stop", which will take you to a condition window prompt so you can select the variable and the condition
- Recursive functions
  - Calling stack: a very important data type



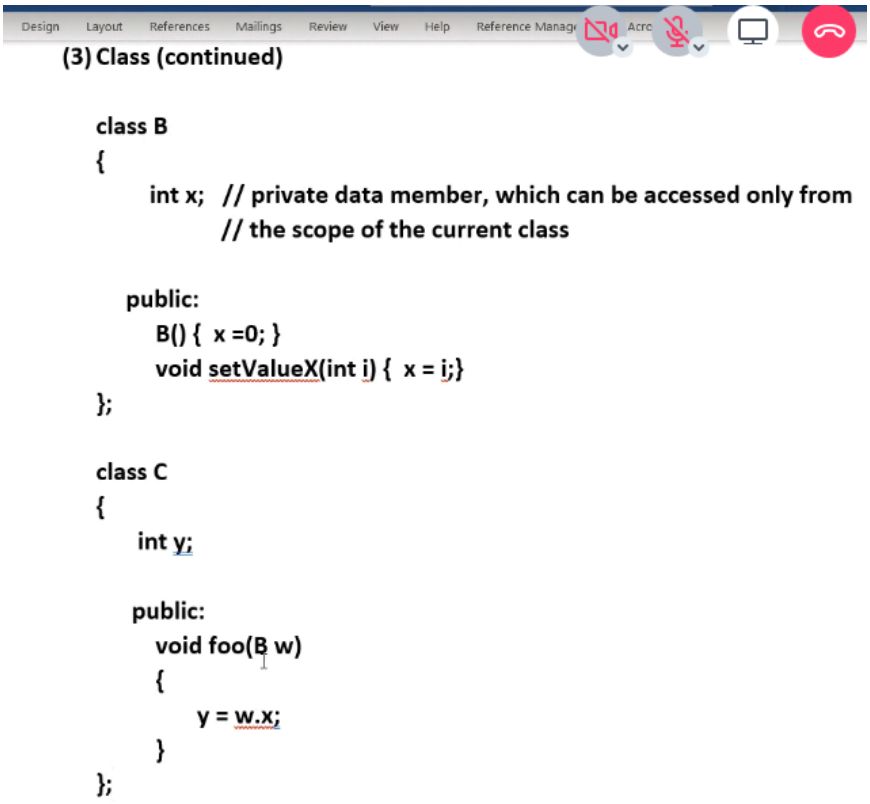
  -
    - Elegantly describes the process of recursion
    - Debugging using break points can help you to understand recursive function calls
    - Example of recursive function: factorial function
- Using assert functions (extremely useful for debugging)

  - If assert condition is violated (resolves as false) then program will abort and you will get a debug error that will tell you what assert conditioned was violated (evaluated as false) to cause the program to abort
  - Thus, if an assert condition evaluates as true, then the program will continue and the assert will do nothing/not cause the program to abort
  - **Make sure** to include library <assert.h>
  - if condition inside assert call is **true** the program will proceed as normal
    - thus goal is to make all asserts evaluate as true so the program runs all the way through and you will know that no odd values have entered your program since no assert conditions were violated
  - can also be used as a tool to guarantee the correctness of you code, it is a safeguard for your code
- unsigned char can be used to represent values 0 to 255
  - more efficient for many types of codes; only 8 bits(1 byte) and is used for things such as RGB color code
  - when you use unsigned char, it is used purely as an integer number
  - all data types are really numbers
    - this is why if("a" > "A") evaluates to true
    - so char c = 't' is the same as char c = 74(hex)
    - atoi function: atoi("11") converts a character into an integer

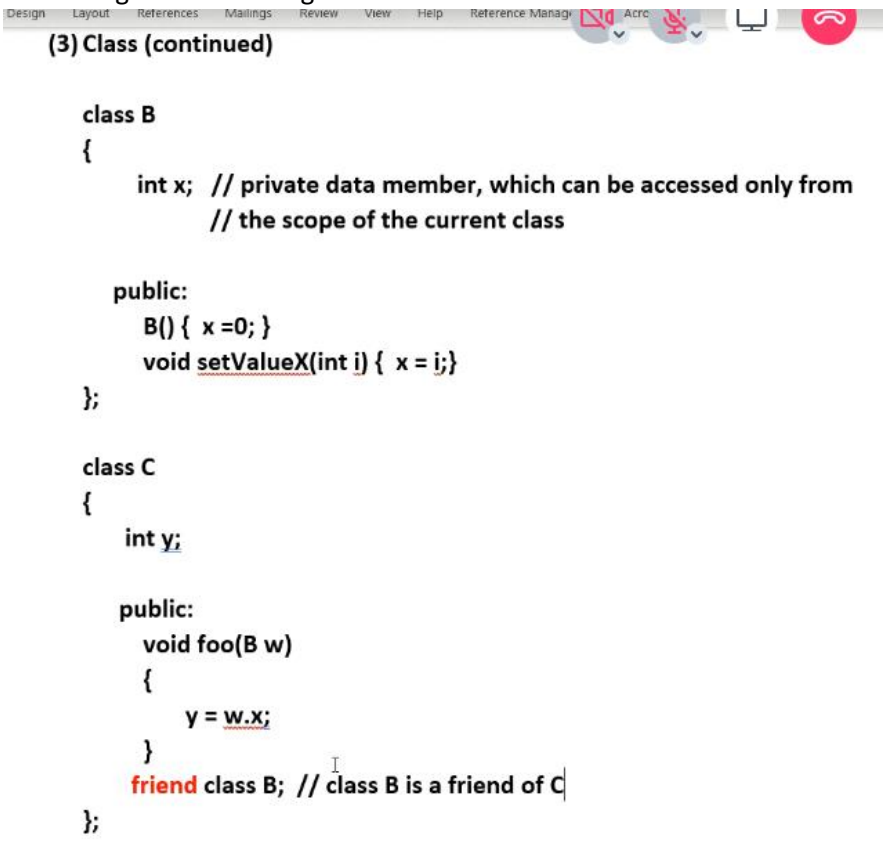
2-10-2021 notes:

- classes
  - remember, private data members can only be manipulated **inside the scope** of the class to which it belongs
- friend functions
  - allow the barrier to be broken from not being able to access private data members of a class; so instead of using a set or get function to access the member or making the data member a public member: solution is to use a friend.
  - Example of incorrectly trying to access a private data member:

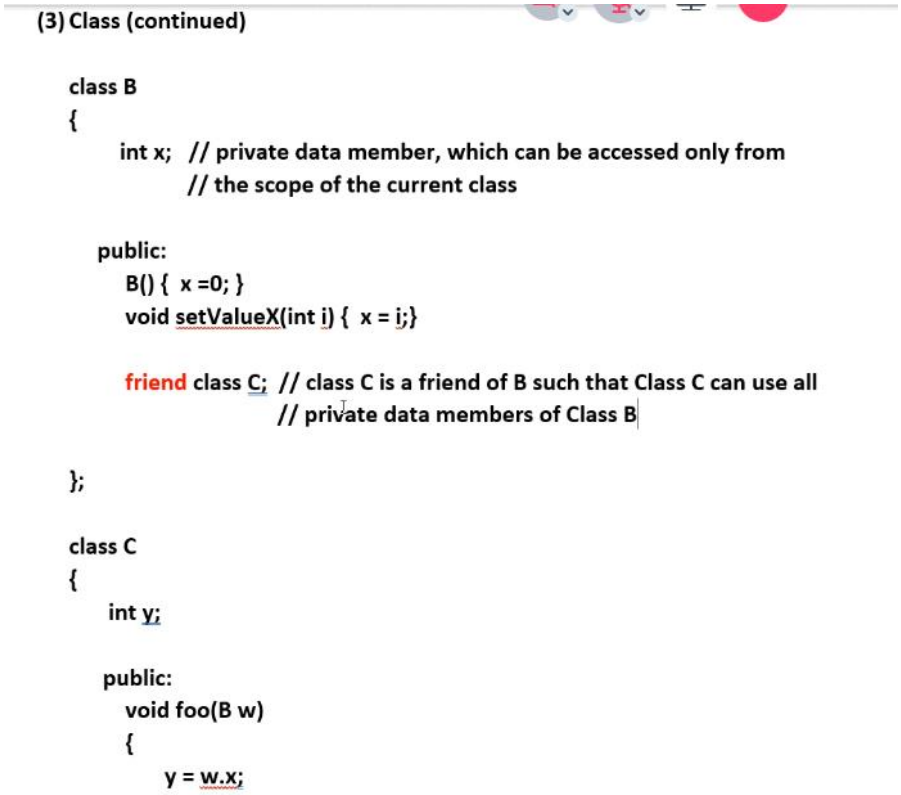
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



- Example of solving this issue using friend class:



- Remember: friendship relationships are one-way; if B is a friend of class A, does not mean class A is a friend of B; for two-way friend relationship you must define friendship in both classes



(3) Class (continued)

```
class B
{
      int x;  // private data member, which can be accessed only from
              // the scope of the current class

   public:
      B() { x =0; }
      void setValueX(int i) { x = i;}

      friend class C;  // class C is a friend of B such that Class C can use all
                       // private data members of Class B

};

class C
{
    int y;

   public:
      void foo(B w)
      {
          y = w.x;
```

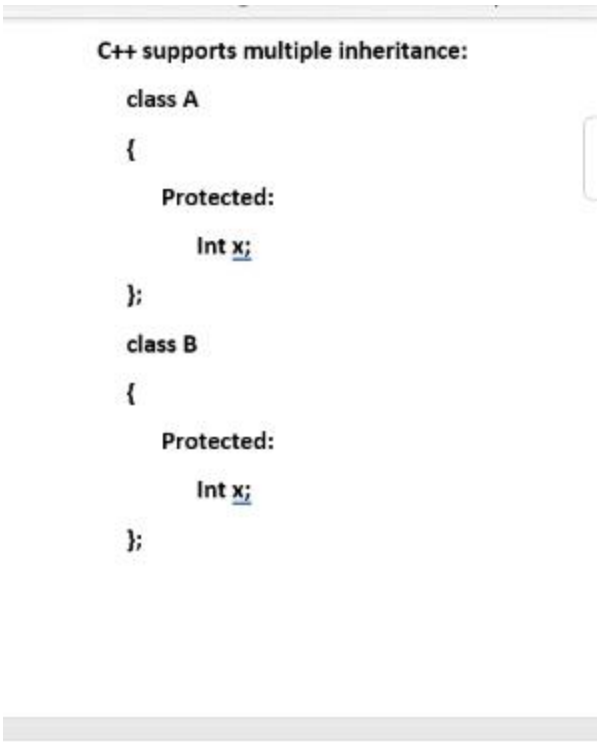
  - Above, class C is a friend of class B, but class B is not a friend of Class C
  - Above, Class C scope can access ALL members of class B, including private data members
  - Good practice to put friend definitions under Public data members just as in the example above
- Friend classes are useful for when you want one class to be able to access all members (including private) of another class, but do not necessarily want/need to use inheritance (which takes up more space whenever you declare child class objects); simply may need one class to be able to use another classes object and be able to edit some of the attributes of those objects easier without using overhead from set/get functions, while still keeping the class non-public members private to every other class respectively.
- You can also declare friend functions; works the same way as with attributes; when a friend function is declared, only the current class and all of its friend classes declared are allowed to use the friend function
  - Example: friend void foo(…){}
    - Only friend classes and current class can use the above friend-defined function

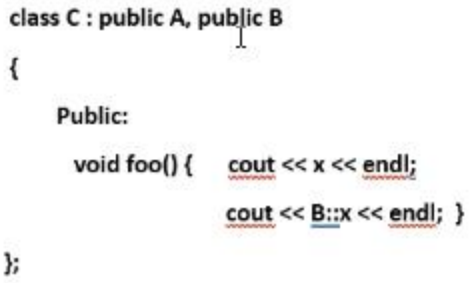
- Inheritance
  - We can create a tree-structure with parents → children → grandchildren…etc.
  - Basic ideas:

- Use a tree structure (hierarchical structure) to organize complex information
- Promote information reuse
  - We put commonality into the parent (base) class and only special things (functions or attributes) into the child class (derived class / sub-class)

- Syntax
  - Use single '":"
    - class *derived-class-name* : ***public*** *base-class-name*
    - inside derived class, you only need to provide the additional attributes/functions that you need; depending on type of inheritance needed, derived class will already contain (inherit) all attributes/functions needed from the base class
  - difference between private and protected attributes in a class: no difference in the base class; in the derived class though, you can access the protected data members of the base class but not the private data members of the base class
- C++ supports multiple inheritance
  - This means you when you create a class, it can inherit attributes from more than one class
  - Ex: class C : public A, public B
    - **But beware** if class A and class B contain an attribute with the exact same name or a function with the exact same name and parameters
    - C++ by default will use the attribute from the first class declared in the multiple inheritance statement; so above it would use class A's attribute
      - But what if you want to use the second or third class inherited? Use the scope operator; example: B::variable-name
      - See example screenshot below:

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:





- Types of attributes
  - Public
  - Private
  - protected
- types of inheritance (use the minimum access rule)
  - public
  - private
  - protected
  - use minimum access rule
    - access level in the derived class = min (access level in the base class, inheritance type) //find the minimum between these two
      - public > protected > private
      - min(public, protected) → protected in the derived class

  - min(public, private) → private data member in the derived class such that the data member can't be accessed in grandchild class
  - in order to avoid confusion, in c# they make only public inheritance the default inheritance

2-15-21 notes (mid-term exam on feb 24 based on all material up to this point, review on feb 22):

- go over lab questions and online lecture notes for the exam.
  - Will be very few questions about UNIX
  - Will test ability to read and write code
    - Don't need to make the code run as if it is real code, just finish all questions and write as much/clear as possible the solution to writing or reading the code; running-result is not necessary when writing code for the exam
  - There will be contextual questions about software engineering
    - Development cycle
    - UML
    - 3 cornerstones of object-oriented programming
    - For UNIX we will focus on the important UNIX commands and the file structure and the security for access levels
- Inheritance (continued)
  - Remember there is no way for a class to inherit another class's private variables.
  - There are three things we CANNOT inherit from base class:
    - Constructors (this is because in the derived class we add more attributes and thus there are more data members to initialize)
    - "=" operator (for similar reasoning to why constructors are not inherited)
    - Friend class attributes
      - For example if class A is a friend of B, it does not mean a subclass of A is a friend of B.
  - Operator overloading
    - Remember an array is always passed as a parameter in a function as a pointer (a pointer to the first element/index address of the array)
    - Remember to always overload copy constructor and =operator.
      - For copy constructor always use pass-by-reference parameter for the same-type object being passed into the copy constructor function
        - Remember to use this-> operator to make it easier to know the current vs the passed-in object.
    - Always return a reference when overloading the =operator so that =operator can be used in sequence
      - The code is essentially the same as the code for the copy constructor.
      - For the return, you do: return *this (returns a pointer to the current that called the function)
    - Overloading the += operator and = operator examples:

```
Row& operator = (Row &w)
{
        for(int i=0; i<DIM; i++)
            x[i] = w.x[i];
        return *this;
}
Row& operator += (Row &w)
{
            C += 1;  // C = C+1
}
```

```
Row& operator = (Row &w)
{
        for(int i=0; i<DIM; i++)
            x[i] = w.x[i];
        return *this;
}
Row& operator += (Row &w)
{
        for(int i=0; i<DIM; i++)
            x[i] += w.x[i];
        return *this;
}
```

- (overloading with an array as a data member example)

- Remember we use pass-by-reference as often as possible so that there is less overhead (functions will not allocate memory on the stack for copied values)
- Overload ==operator

```
bool operator == (Row &left, Row &right)
{
        for(int i=0; i<DIM; i++)
            if(left.x[i] != right.x[i])
                return false;
        return true;
}
```

- Prefix and post fix operators (can be tricky to overload)
  - For ++operator, do ++ first then do the assignment (=).

```
// prefix ++ operator:  do ++ first, and then =
Row& operator ++( )
{
    for(int i=0; i<DIM; i++)
        x[i]++;
    return *this;
}
```

- The tricky one to overload is the post-fix operator:
  - Do assignment (=) first, then do ++.
  - Will provide int as a parameter, but the name of the parameter is not necessary (but you can use it if you want).
  - Need to create a temporary variable
    - This is because remember: for post fix, the left-hand operand is FIRST assigned to the RH operand, and THEN the RH operand is incremented:
    - No choice but to use pass by value since for local temporary copy object created to return before the original is incremented will be destroyed at end of function:

```
Row operator ++(int )
{
    Int y[DIM];
    for(int i=0; i<DIM; i++)
        y[i]=x[i];
    Row t(y);
    for(int i=0; i<DIM; i++)
        x[i]++;
    return t;
}
```

      - Notice Row& is not the return value for this function

- Overloading the ostream operator<<

```
friend ostream operator <<(ostream &os,
        const Row &w)
{
    for(int i=0; i<DIM; i++)
        os << w.x[i] << "  " ;
}
```

- We return friend because when we use the >> or << operator we are using the object's attributes:

```
// ostream is a friend of Row such that it can
// output the private data member of Row
friend ostream operator << (ostream &os,
        const Row &w)
{
    for(int i=0; i<DIM; i++)
        os << w.x[i] << "  " ;
}
```

- Overloading << and >> operator can be a VERY POWERFUL tool to make a program easy once you write all the complexities and include using these operators.

- Lists: an important data structure in computer science

LAB 5: 02-08-2021

2-17-2021 notes:

- List: a basic data structure in computer science (one-dimensional structure)
  - Can be classified into two groups
    - Unsorted: no particular order exists in the list
    - Sorted: there can be an ascending and descending order In the list
  - Used to organize data, including tasks execution order.
  - How to find a particular element in a list?
- Search methods:
  - Linear search: search list from head to tail one-by-one in sequence (implemented by for-loop)
    - Poor time execution for large lists; Time complexity = O(n), where n for this case is the number of elements in the list.
  - Binary search: quick way to search an item from a **sorted** list.
    - Only suitable for a sorted list; unsorted list for input will not work with a binary search.
    - Time complexity = O(log n); much better than linear search.

- For example; if n = n^6, linear search has to search through 1million elements; but for Log(10^6) = 6. HUGE DIFFERENCE; much smaller execution time for searching a sorted list.

- Implementation of Linear and Binary search
  - Use array or linked list for input
  - Need three categories of functions:
    - Transformers, Observers, and Iterators
      - Iterators are used to reset a list (as in move the pointer from some class member function back to the start of the list)
      - The get-next-item function will allow you to use operators to iterate through the list
  - Unsorted list:
    - For removing an element, don't shift all previous elements after the delete forward; simply take the last element, copy and delete it, and paste it over the element in the list to be deleted; so effectively you just move the last element to the location of the element to be deleted. This is fine because the list is unsorted.
    - You can use: typdef int ItemType; //you can easily change the int to float or whatever data type desired.
    - Note: whenever an element will go unused, don't bother to delete it contents when it is not necessary; save some execution time because if the element can't be accessed because of a "delete" (length of array declared shorter after a delete operation) , then the data that it holds is irrelevant since the memory is still allocated.
  - Sorted list:
    - To add an element
      - Find location to insert
      - Shift all remaining elements downward by one position
      - Insert the item into the location
      - Increment length
    - To delete an element
      - Find location of item to be deleted
      - Move the remaining list upward by one position (which will in effect overwrite the location to be deleted)
      - Decrement the length
        - Unfortunately for add and delete with sorted list it costs a lot of overhead because you have to move all elements up
          - This can be overcome by using a sorted linked list
    -

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

2-22-2021 notes: Overview for the Midterm Exam [WED, FEB 24 11AM – 2PM; 3 HRS to complete; Prof prefers we submit one single, typed file, similar to lab report format] (6 big questions on the exam)

- Main purposes of exam questions:
    - Writing code
    - Reading the code
    - Data structures
    - Unix
    - Software engineering
- Strategy to prepare:
    - Go over all labs and assignments in order to be prepared for the exam and make sure you understand all of the code
    - Understand code from lecture notes
    - Real lecture notes
- Major topics:
    - Unix:
        - Tree structure
            - Structure is like a tree in the real world; but upside down(root node at the top)
            - 2d structure (unlike lists)
            - Hierarchical structure better to organize big data
        - Relative and absolute paths
        - File access
        - Pipe and redirect
        - Some important commands (meaning)
    - Software engineering
        - UML
        - Testing
        - Development cycle
        - Fundamental building blocks of SE
    - C++
        - Provide at least the skeleton to show the overall method of a given code
        - Know how bitwise operators work
        - Structs and classes (two major containers/bags)
            - Use a struct for simple application and need to represent basic information; everything public by default; no need for set and get functions (although you can write functions)
            - For polymorphism; go with a class: but there is extra overhead; struct is more lightweight compared to a class
            - Know how to write definition of a struct and access data members of a struct
            - Struct and Array in combination

        - You can create a struct and give it an array data member
        - Or you can have an array made of struct data members
    - …
    - Class
        - All member functions normally we put in public:
            - Should contain default, parameterized, and copy constructors
            - Overload operators
- Input and output:
    - Cout << //go to computer screen
    - Cin >> //get input from keyboard
    - #incldue<fstream> //class for file stream to output or input from a file

Statements for using file I/O

```
#include <fstream.h>

ifstream  myInfile;              // declarations
ofstream  myOutfile;

myInfile.open("A:\\myIn.dat");   // open files
myOutfile.open("A:\\myOut.dat");

myInfile.close( );               // close files
myOutfile.close( );
```

        -
        - A: is from old style hard drives, nowadays it will likely say C: for the root node (hard drive / starting location of a memory drive)
- Control: for, while, loops, switch-case statements, if-else statements
    - Don't forget in a loop statement or switch statement to make use of the 'break' or 'continue'
        - Break → exit loop
        - Continue → goes back to start of loop to start the next iteration (and exits current iteration)
- Function:
    - Pass by reference (function variable points to the same address as the passed-in variable)
    - Pass by value (copy value of passed-in variable to new variable/memory location)
- Debugging
    - Regular break point, conditional breakpoint

        - Assert.h functions (if assertion is evaluated as FALSE, program will halt at the location of the assert) //especially good for debugging logical errors (pre and post conditions)
        - Cerr // output log file
    -
    - Array:
        - How to traverse all the elements of 2-d arrays?: use two for-loops; one for the rows, another for columns
    - List:
        - For now, at least know how to **add or delete** one item for the unsorted list; remember, order doesn't matter for unsorted list, so the operation should be very simple
    - Inheritance:
        - Base class → derived class
        - Inheritance type (public, private, protected) of public/private/protected data members of base class
    - Friend:
        - Only way for one class to access another class's private data members, it must be a friend of the class of which the private members will be accessed
        - Remember: friend relationships are ONE WAY; in order for two classes to access each others private members, they must both specify in their class methods that the other class is a friend; otherwise, if only one class makes another class a friend, it still cannot access the private members of the class to which it made a friend and allow it to access its private members.
        - Declare friend classes under the public members of the class
- Data structures
    - Unsorted List //no sequential order of the data in any way
        - We can only use a linear search: use for-loop to search every element one by one
        - Time complexity for linear search: O(n), where n is the number of element in the list
        -

3-1-2021 notes:

- Remember always initialize variables inside of functions especially since uninitialized variables contain junk information; usually a really large number.
- Search
    - Unsorted list: one-way traversal (linear search) via a for-loop
    - Sorted list: can do linear, but also can use binary search
- Binary search
    - Basic idea: divide and conquer

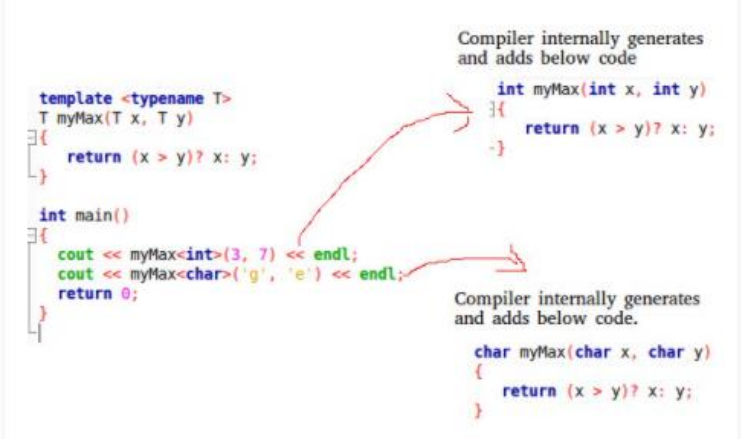
- Each step halves the number of elements to be searched
  - Start at middle element; compare this middle element to the value to be searched for
  - Because the list is sorted: if middle value is greater than search value, then we know the search value cannot be in the upper half of the list; if the middle value is smaller than the search value, then we know the search value cannot be in the lower half of the list
  - Can use a while-loop or recursive function to accomplish a binary search
  - Use 'FIRST' and 'LAST' variables to change the location of the first and last element in the list when doing binary search; this is necessary to determine the midpoint element value for each time you half the array while searching
    - Use switch case inside of while loop for less than, greater than, or element found cases
  - Recursive version of binary search

```
                                                    // Recursive definition
template<class ItemType>
bool BinarySearch ( ItemType info[ ], ItemType item ,
                    int fromLoc , int toLoc )
    //  Pre:  info [ fromLoc . . toLoc ] sorted in ascending order
    //  Post:  Function value = ( item in info [ fromLoc . . toLoc] )

{     int mid ;
      if ( fromLoc > toLoc )                    // base case -- not found
            return false ;
      else {
              mid = ( fromLoc + toLoc ) / 2 ;

           if ( info [ mid ] == item )     // base case-- found at mid

             return true ;

           else if ( item < info [ mid ] )          // search lower half
                  return BinarySearch ( info, item, fromLoc, mid-1 ) ;
             else                                   // search upper half
                  return BinarySearch( info, item, mid + 1, toLoc ) ;
        }

}
```

fig 3

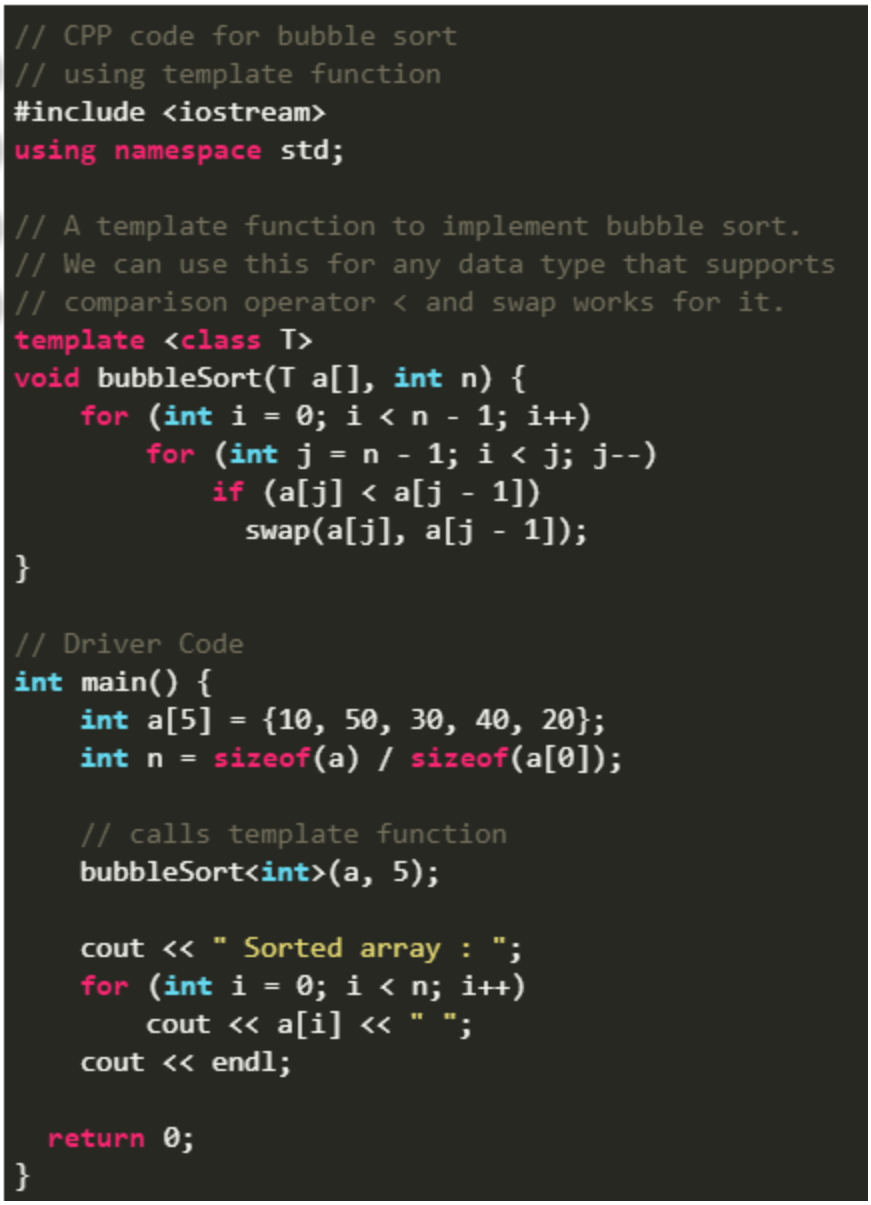
-
- Use while-loop for better efficiency/better time complexity since recursive call will have to keep calling functions on the stack

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Templates
  - 

    - C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.
  -

```
#include <iostream>
using namespace std;

// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
   return (x > y)? x: y;
}

int main()
{
  cout << myMax<int>(3, 7) << endl;  // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char

  return 0;
}
```



```cpp
// CPP code for bubble sort
// using template function
#include <iostream>
using namespace std;

// A template function to implement bubble sort.
// We can use this for any data type that supports
// comparison operator < and swap works for it.
template <class T>
void bubbleSort(T a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
              swap(a[j], a[j - 1]);
}

// Driver Code
int main() {
    int a[5] = {10, 50, 30, 40, 20};
    int n = sizeof(a) / sizeof(a[0]);

    // calls template function
    bubbleSort<int>(a, 5);

    cout << " Sorted array : ";
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;

  return 0;
}
```

- Output→ Sorted array: 10 20 30 40 50

- **What is the difference between function overloading and templates?**
  Both function overloading and templates are examples of polymorphism feature of OOP (object-oriented programming). Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations.

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Class Templates Like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

Following is a simple example of template Array class.

```
#include <iostream>
using namespace std;

template <typename T>
class Array {
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print() {
    for (int i = 0; i < size; i++)
        cout<<" "<<*(ptr + i);
    cout<<endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

Output:

```
 1 2 3 4 5
```

- **Can there be more than one arguments to templates?**
  Yes, like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same.

```
#include<iostream>
using namespace std;

template<class T, class U>
class A  {
    T x;
    U y;
public:
    A() {    cout<<"Constructor Called"<<endl;   }
};

int main()  {
   A<char, char> a;
   A<int, double> b;
   return 0;
}
```

- Remember you can define multiple functions with the same name but different interfaces:

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Template:

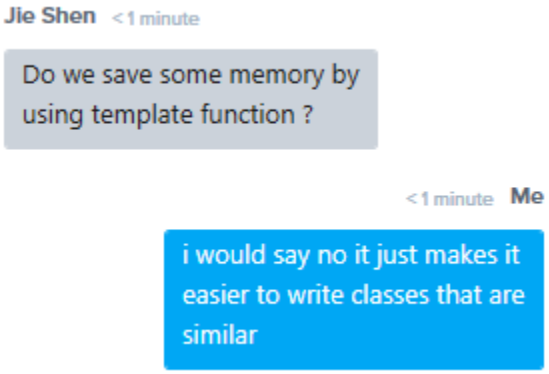
```
Void foo(int x)
{ .... }
```

```
Void foo(float y)
{
 ....
}

Void foo(Student z)
{
    ....
}
```

- Instead of writing multiple functions, a better solution for representing this polymorphism/overloading is to use a template function:

```
template <class T>    // T is a generic data type
void foo( T x)   // template function
{
    ....
}
```



- Above is correct. It does not save memory.

- C++ will create as many copies needed internally and thus allocate memory, it just makes it easier so you do not have to explicitly define each class that has the same layout for multiple classes. C++creates the classes implicitly based on the template function declared.

- We can use templates to define a class or struct, or a function
- Example template function:

Swap

```
#include <iostream.h>
template <class TParam>
void Swap( TParam & x, TParam & y )
{
  TParam temp;
  temp = x;
  x = y;
  y = temp;
}
```

Swap

```
int main()
{   int m = 10;
    int n = 20;
    Student S1( 1234 );
    Student S2( 5678 );
    cout << m<<"   "<<n<<"   "<<endl;
    Swap( m, n );     // call with integers
    cout << m<<"   "<<n<<"   "<<endl;
    cout << S1.getID()<<"   "<<S2.getID()<<"   "<<endl;
    Swap( S1, S2 );   // call with Students
    cout << S1.getID()<<"   "<<S2.getID()<<"   "<<endl;
    return 0;
}
```

Multiple parameter function template

```
template <class T1, T2>
 void ArrayInput(T1  array, T2 & count)
{
    for (T2 j= 0; j < count; j++)
    {   cout << "value:";
        cin >> array[j];
    }
 }
```

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

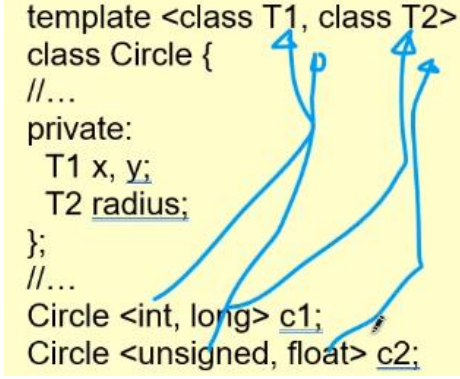
Class Template

- Declare and define an object:

```
template <class T>
 class MyClass{
 //…
 }
MyClass <int> x;
MyClass <student> aStudent;
```

Simple Example

```
template <class T1, class T2>
class Circle {
//…
private:
  T1 x, y;
  T2 radius;
};
//…
Circle <int, long> c1;
Circle <unsigned, float> c2;
```



- Must use <> brackets when declaring a class based on a template class to 'instantiate' the class template object

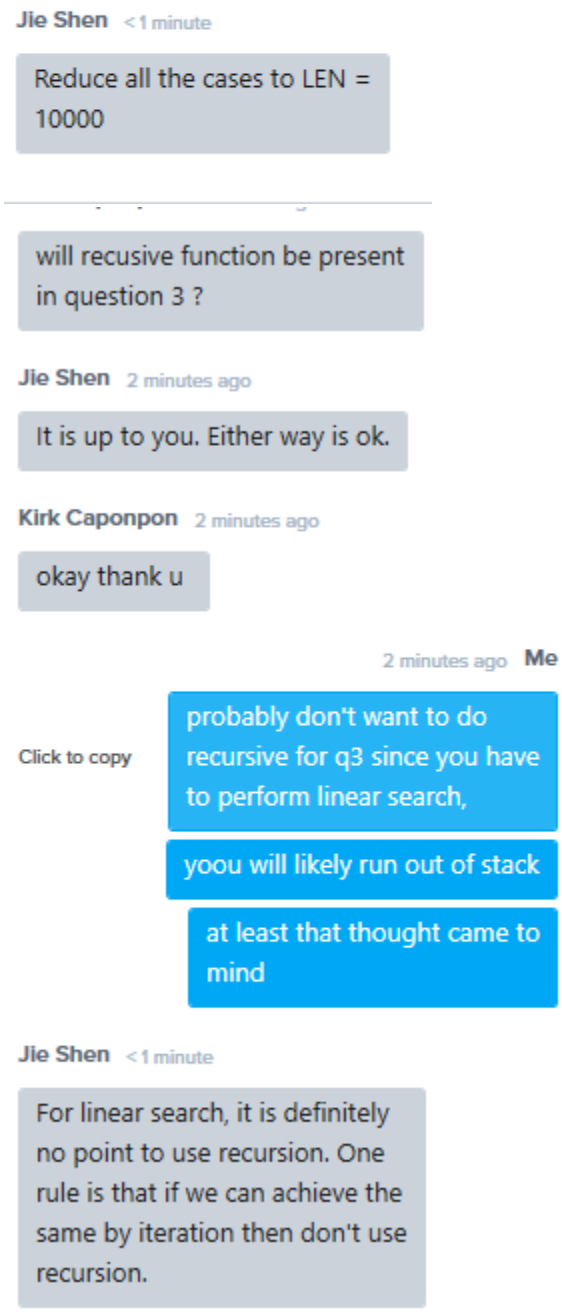
LAB 6: 3-1-21

Q1:

```
Int foo(int y, int x)
{
    If(x == 1)
        Return y;
    Elsif(x == y)
        Return 1;
    Else
       Return( foo(y-1, x-1) + 4*for(y-1, x));
```

-

- Then just write a main function to test the function
- For questions with arrays, if array is too big for program to run, simply reduce LEN and test cases by some factor of 10:



Assignment 2: 3-1-21

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

for assigment 2, question 1, do you want us to test every single function for like deleting a char on the list, or do u just want us to keep it simple to what you said on the main

like do u want the main to be like this

```
Int main( )
{
        UnsortedList x;

        // open char.dat

        ..... // set up x based on the data in char.dat


        x.printElement( );

        cout << numCharElement ( ) << endl;

        return 0;
}
```

-
    - Answer is yes; just write a main to test the functions

Assignment 3: 3-2-21

Kirk Caponpon 7 minutes ago

professor i have a question for assigment 3, wouldn't the insert () and delete() functions of both the unsorted and sorted list be the only one different, while the rest of the functions would have identical code?

Jie Shen <1 minute

Yes.

Kirk Caponpon <1 minute

okay thank you

-

3-3-2021 notes:

- Template (continued)

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Main purpose is to utilize generic data types for reducing programming time (not to reduce overhead; they do not reduce overhead; they only condense lines of code that a programmer has to write)
- Several applications: template function, template class, template struct
- It does not reduce memory usage
- Template function:

```
template <class T>

T foo(T x, int y)

{  ... }
```

  - Note: “template<class T>” is necessary when defining a template function
- Note: if you do not use the template for more than one data type, then there is no point in defining a template function

- Call a template function:

```
Int main( )
{
   Char a = 'A', c;
   Int b = 10;
   c = foo(a, b);
   …
}
```

- Swap function is a perfect application for a function template
- For functions you can define more than one generic data type (T1 and T2 for example)
- No angular brackets needed; you simply plug in data types into the function, the template will know how to handle whichever data type you input (provided it is a compatible object with the template class of course)

- Template class:
  - Need keyword/phrase: template<class T> //T is a generic data type variable, I can define it using another character if I want to
  - Then you write the class as normal, and make sure to return T, the generic data type variable:

```
template <class T>
struct E
{
    T IDnumber;
    double x;
};
```

  - Need to at least define one generic data type (T), otherwise there is no point in creating a template class

- In main, declare an instance of the template class using <>:

```
Int main( )
{
    E <int> a;
    E <Student> b;
    ...
}
```

- Class template can also have more than one template as well:

Simple Example

```
template <class T1, class T2>
class Circle {
//…
private:
  T1 x, y;
  T2 radius;
};
//…
Circle <int, long> c1;
Circle <unsigned, float> c2;
```

What is a Template?

- Templates give us the means of defining a family of functions or classes that share the same functionality but which may differ with respect to the data type used internally.
- A class template is a framework for generating the source code for any number of related classes.
- A function template is a framework for generating related functions.

- Whenever you define a function outside of a class template, you must include the <> and the scope operator:

```
Template <class T>

Array<T>::Array( unsigned sz )
{  values = new T [sz];
  size = sz;
}
```

- Note: you must also include "template<class T>" when defining a function outside a scope of the class as well:

```
template<class T>
T& Array<T>::operator[](unsigned i)
{
        return values[I];
}
```

- The above shows everything necessary when you are writing the definition for a function outside of the scope of the template class; it must be done for every function from the template class written outside of the scope of the template class

- Exceptions
  - Widely used in commercial code //not as necessary for the hw (assert is a more simple, common debugging/logic checking tools. But assert it is more limited in its functions)
  - Special, undesired cases that occur in your program that causes the program to go wrong somewhere; we want to handle/hide these.
  - Use Exception Handlers: a special function that can be used to do processing when some undesired/unusual events happen.
  - Syntax
    - Try{} block
    - catch(){}block //also a function call
    - throw action //consider this as a function call
  - Exception handling in C++ is not as good compared to Java and C#
    - Java and C# have a class that they define to incorporate polymorphism to handle each type of object thrown; they have a lot of built-in classes to handle exceptions
  - You can use a try block when you know/have an idea that over some block/section of code there is an error, but you just don't know exactly where that problem is occurring.
    - Follow a try block with a catch block (exception handler)

```
try
{
  ...  //  your segment of code that may cause
     // problems


}
catch( .. )
{
      // Exception handler


}
```

- If you use keyword “throw” it means you explicitly throw an exception.

```
if (n>9) throw "Out of range";
myarray[n]='z';
```

  - Here a string “out of range” is thrown.
  - Whatever is thrown will cause all lines of code to be skipped until the throw reaches a catch block that can receive its input:

```
try
{
  for (int n=0; n<=10; n++)
  {
    if (n>9) throw "Out of range";
    myarray[n]='z';
  }
}
catch (char * str)
{
  cout << "Exception: " << str << endl;
}
```

    - Here the catch block can catch the string since the parameter type of the catch is char* (char pointer), which is what is used to capture a string literal.

```
#include <iostream.h>    // exceptions: multiple catch blocks
int main () {
  try
  {
    char * mystring;
    mystring = new char [10];
    if (mystring == NULL) throw "Allocation failure";
    for (int n=0; n<=100; n++)
    {
      if (n>9) throw n;
      mystring[n]='z';
    }
  }
  catch (int i)
  {
    cout << "Exception: ";
    cout << "index " << i << " is out of range" << endl;
  }
  catch (char * str){
    cout << "Exception: " << str << endl;
  }
  return 0;
}
```

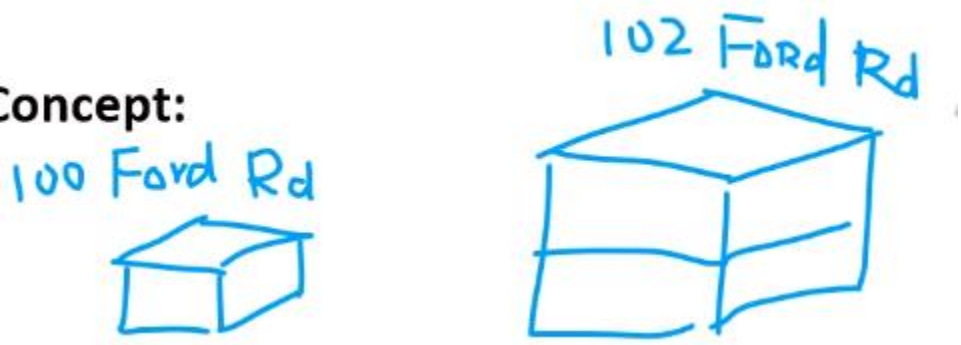
  - Notice here the first throw will go to the first catch, but it doesn’t match the parameters, so then it goes to the next catch, which is a match; this is the try block that will execute.

- After this, program continues execution on the lines **after** the catch block executed.
    - Advantageous over "break" because it will go to a catch block that can output information or make a correction to handle the exception, and then continue the program.
- After one exception handler is used, it no other exception handlers (catch blocks) will be entered until another unhandled exception is thrown.
- **Use catch(…) to catch everything**
    - This is usually placed as the last/default exception handler if all other catch blocks are not able to handle an exception thrown (i.e., an unknown exception)
- Many times, depending on type of exception you either clean up the issue and continue/or reset variables/restart program, or you can exit the program after an output message.
- You can use polymorphism for catch blocks since it is essentially a function

- Pointer
    - The most difficult, most powerful, double-edged sword of C++
    - Mistakes made with pointers can be detrimental to a program
    - Highest level of memory manipulation
    - Think of pointers as managing a building. Some buildings are 1 story, some 2, some 3…etc…but each building contains information, and the more stories, the more complex the building becomes. But they are still buildings and have the same level of base complexity.

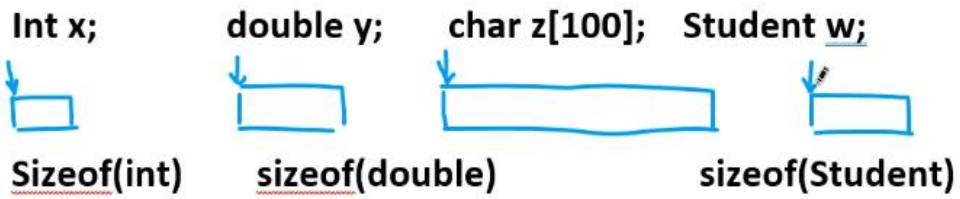
    - **Concept:**

    - **Concept:**



- Remember the “address” of the building is the entry point to all of the building and its complexity.
- Remember you can use sizeof() function to check the size of a data type (in bytes)
  - Example: sizeof(int) //output will be 4 (bytes) (4 bytes = 4*8 = 32 bits)

    - **Concept:**



  - Remember no matter the size of memory taken up, each variable has a starting memory address that can be represented by an integer (in number of bytes)

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Typdef
    - A type alias is a different name by which a type can be identified. In C++, any valid type can be aliased so that it can be referred to with a different identifier.

SPECIAL NOTES 1

- IF A FILE WONT OPEN FOR READ OR WRITE!!!!!!:
    - CHECK ITS ACTUAL FILE NAME!!!!!!!!! SOMETIMES IT COULD BE NAMED FILE.TXT.TXT OR FILE.DAT.DAT FOR EXAMPLE. GO TO THE FILE AND SELECT PROPERTIES AND CHECK ITS OFFICIAL FILE NAME!!!!! CHANGE IT TO THE PROPER NAME AND ENSURE NO DOUBLE FILE EXTENSIONS APPENDAGES OCCUR
    - ALSO MAKE SURE THAT THE FILE IS NOT BLOCKED (select UNBLOCK check box) DUE TO IT COMING FROM AN UNKNOWN SOURCE AND THAT ALL PERMISSIONS ARE ALLOWED ON THE FILE FOR ALL USERS: READ WRITE AND EXECUTE.
    - WHEN YOU COMPILE A PROGRAM USING A TEMPLATE FUNCTIPON YOU MAY GET AN ERROR ABOUT DUPLICATE FILES OR ADDITIONAL COPIES FOUND FOR A CPP FILE.
        - SOLUTION IS TO REBUILD THE PROJECT THEN RUN IT; BREAKPOINTS DON'T FUNCTION WELL WHEN YOU MAKE CHANGES TO A PROGRAM THAT USES TEMPLATES SINCE TEMPLATE FUNCTION CODE IS DEFINED BY THE COMPILER AT RUNTIME SO IF YOU CHANGE IT THEN IT DISRUPTS THINGS; HENCE, YOU MUST REBUILD IT SO THAT BREAKPOINTS FUNTION PROPERLY AND YOU NO LONGER GET THE WARNING/ERROR

3-8-2021 notes:

- Pointers

Int x; **double y;** **char z[100];** **Student w;**
&x **&y** **z, &z[0]** **&w**

Sizeof(int) **sizeof(double)** **100*sizeof(char)** **sizeof(Student)**

Rule 1: In c++, array name or function name itself represents the beginning memory address of the array or function.

Rule 2: for regular variables, we can use &var_name to retrieve the beginning memory address of the variable.

    - Rule1 in C+: array name or function name itself represents the beginning memory address of the array or function
    - Example for char array z: cout << z -----> will output beginning address of z.
    - Rule2: For regular variable to get the address: use & operator: &var_name
        - Example from above, using variable x: cout << &x ---> will output address of x.
    - A pointer variable is a variable that stores a memory address of a location in memory.

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Syntax: int * p1; → pointer p1 points to a memory location that stores an integer.
  - Int x = 5;
  - P1 = &x; → p1 stores integer memory address of location x.
  - Remember how a CPU works to retrieve a memory location from RAM (&x = memory address location) using busses and logic gates, and if necessary it will get the data stored at that RAM location (x = 5).
  - If you use dereference operator * in front of a pointer it will output the value stored at the location memory location it is storing. Ex: p1 == a memory location stored by p1; *p1 == data value stored at memory location that the pointer p1 stores; *p1 is an alias of x.

    A pointer variable is a variable whose value is memory address of a location in memory.

    Syntax:

    ```
    Int * p1;
    Int x = 5;
    P1 = &x;
    Cout << x  << endl;
    Cout << *p1 << endl;   // *p1 refers to the dereference of pointer variable p1
                           // *p1 is an alias of x
    *p1 = 10;
    Cout << x << endl;   // x = 10 at this moment
    ```

- Remember: you can reinterpret a char to point to a memory location of a different data type
- internally, No matter what, the beginning memory address of any data type, including created class..etc, points to a 4byte = 32-bit memory location (int) since no matter what the beginning memory address of any data is a 4-byte address size.
  - All pointer variables just contain a memory address, and the size of a memory address is 32-bits (or 64-bits depending on Operating system)…so you see, when a pointer points to a Char, or a Int, or createdObject from a class, all of them are stored at a memory address (or beginning memory address) of 32-bits (4 byte) memory address location. RAM memory address locations stored as 32-bits=address location.
- Note: you are not allowed to access the memory address of a pointer; so when you assign one pointer to another, you are only assigning the memory address data that one pointer holds, to the other.

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Pass by value:

```
Void foo(int x, int y) { …}
Void foo2(int &x, int &y) { …}   // pass by reference
Void foo3(int *x, int *y) // interface is the place where local
                          // declaration occurs
{
   Int temp;
```

- **Pass by value:**

```
Void foo(int x, int y) { …}
Void foo2(int &x, int &y) { …}   // pass by reference
Void foo3(int *x, int *y) // interface is the place where local
                          // declaration occurs
{
   Int temp;
   Temp = *x; *x = *y; *y=temp; // deference
}

Int main()
{
        Int a = 2, b = 3;
        Int *c, *d;
        C=&a,  d = &b;
        Foo(a, b);  // pass by value

        Foo2(a, b);  // Way 1: pass by reference

        Foo3( &a , &b    );
```

```
    Temp = *x; *x = *y; *y=temp; // deference
}

Int main()
{
        Int a = 2, b = 3;
        Int *c, *d;
        c =&a,  d = &b;
        Foo(a, b);  // pass by value

        Foo2(a, b);  // Way 1: pass by reference

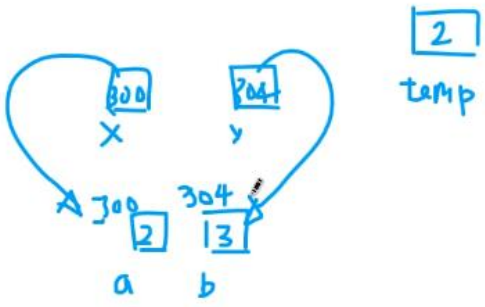
        Foo3( &a , &b    );

        Foo4(*c,   *d);  // ok?  No
        Foo4(c, d);
```

- Pass by value:

```
Void foo(int x, int y) { ...}
Void foo2(int &x, int &y) { ...}  // pass by reference
Void foo3(int *x, int *y) // interface is the place where local
                          // declaration occurs
{
   Int temp;
   Temp = *x;
   *x = *y;
   *y=temp; // deference
}

        Int a = 2, b = 3;
        Foo3(&a, &b);
```



- Example 4: pointer to a struct

```
} a, b;
a.ID = 101, a.salary=30000;
b.ID = 102, b.salary=35000;
Employee *p1, *p2;
P1 = &a;
P2 = &b;
Cout << a.ID << a.salary << endl;
Cout << p1->ID << p1->salary << endl;

Rule: regular_var.data_member
      Poiter_var->data_ member
```

```
Example 5: pointer to class
Class Employee2
```

- Example 5

```
Example 5: pointer to class
Class Employee2
{
    Int ID;
    Float salary;
  Public:
    Int getID() { return ID;}
    Float getSalary() { return salary;}
} e, f;
```

- Assume he already wrote set and get functions for int ID and float Salary

```
...
Employee2 *p3, *p4;

Cout << e.getID()  << e.getSalary() << endl;
Cout << p3->getID()  << p3->getSalary() << endl;
```

- So notice the same way you access struct data members is the same way you access member functions using the -> operator with pointers
- Not if you use derference operator on a pointer to an object, then you can must use the normal . operator, since in effect *pointer essentially is another alias for the created object.

- Example 6: dynamic array
  - We call a **static** array static because we determine the links for the array at compile time, not at runtime; **dynamic** array sizes are determined at runtime.
    - static arrays will result in faster programs since the size of the array is hardcoded

Example 6: dynamic array

```
Int x[20][30];   // static array
```

One dimensional dynamic array:

```
Float *x = new float[
```

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
One dimensional dynamic array:
Int len1;
Cout << "Input the dimension: ";
Cin >> len1;
Float *x = new float[len1];  // similar to malloc( ) in c
For(int i=0; i<len1; i++)
   x[i] = i;

// It is the programmer's responsibility to do the deallocation
delete []x;
```

- Use the delete operator to deallocate, otherwise you will have a memory leak in your program which are VERY bad and can be hard to diagnose

3-8-2021 LAB 7:

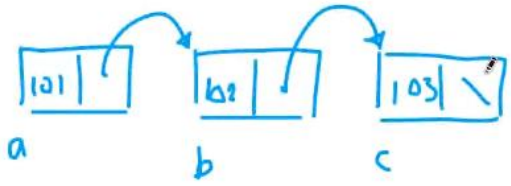
- Array disadvantage is adding in and taking out elements; so we use LINKED LISTS
- Linked list
  - Data structure to represent a list
  - Main reason: we want the flexibility to add or remove one element from the list

```
Main reason: we want the flexibility to add or remove one elem
the list.

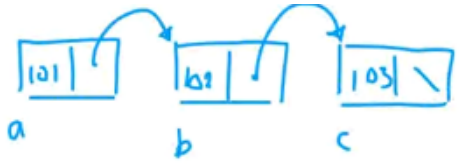
Struct Node
{
     Int id;
     Node *next;
};

Node a, b, c;

a.id=101, b.id=102, c.id=103;
```



CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:




```
Struct Node
{
    Int id;
    Node *next;
};

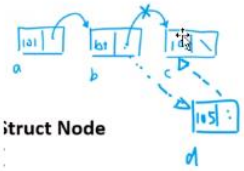
Node a, b, c;

a.id=101, b.id=102, c.id=103;
a.next=&b, b.next=&c, c.next=NULL;  // NULL
```

-
- NULL is a constant = 0; 0 represents an invalid memory address and is used to represent the end of a list
- Adding an element:

```
Node a, b, c;

a.id=101, b.id=102, c.id=103;
a.next=&b, b.next=&c, c.next=NULL;  // NULL=0
Node d;   d.id=105;
```

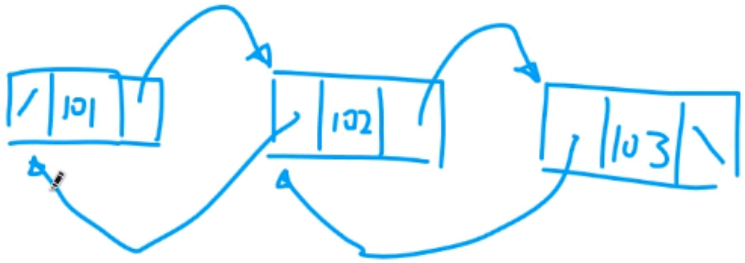


-
  - **b.next=&d, d.next=&c;**
  - The above is an example of a singly linked list
  - There is a such thing as a doubly linked list
-
-
-
-

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Doubly linked list:

```
Struct Node2
{
    Int id;
    Node2 *next;
    Node2 *prev;
};
```



- / means NULL. If it was a circularly linked list, then no node next or previous pointer contains NULL

```
Struct Node2
{
    Int id;
    Node2 *next;
    Node2 *prev;
} a , b, c;


a.id=101, b.id=102, c.id=103;
a.next=&b, b.next=&c, c.next=NULL;
a.prev=NULL, b.prev=&a, c.prev=&b;
```

- Doubly linked list example

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Dynamic Arrays
  - You need double stars ** when allocating; ** means double pointer:

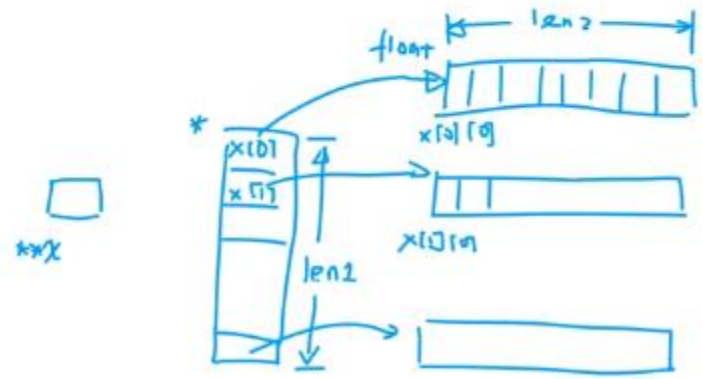
```
Int len1, len2;

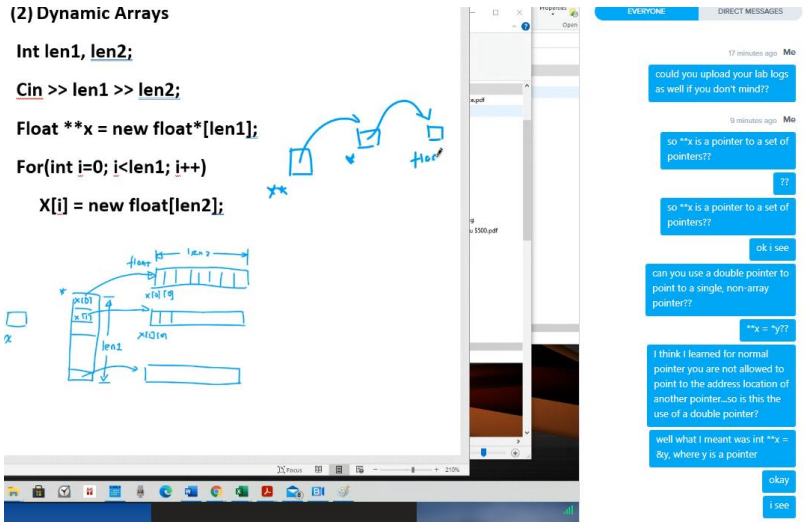
Cin >> len1 >> len2;

Float **x = new float*[len1];

For(int i=0; i<len1; i++)

    X[i] = new float[len2];
```



  -
    - **x is a double pointer variable; but remember all pointer variables are internally integers; it just contain a memory address; in this case it points to another pointer (set of pointers), which those pointers actually point to some data
    - 

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- So: double pointer x points to →address of a single pointer y which points to → address of a normal variable. **x → *y → data
- Basically a 2D matrix pointer
- So **x is a pointer to a set of float pointers (float*[len1])
- The above is how you allocate

We normally expect a double pointer variable is pointing to the begin memory address of a single point variable.

```
Double **x;
Double *y;
x = &y;
```

  - The above would not work if x was not a double pointer variable (and only a single pointer variable)

- To delete the above dynamically allocated array, you need a for loop to delete both dimensions of dynamically allocated memory:

```
// deallocation
For(int i=0; i<len1; i++)
    Delete []x[i];
```

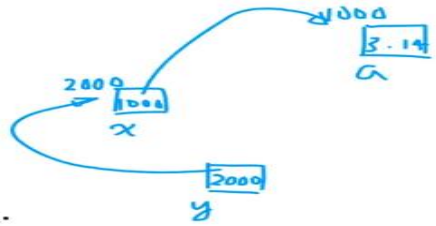
```
Delete []x;
```

- REMEMBER: ALL VARIABLES HAVE A **MEMORY ADDRESS** AND **CONTAIN DATA**; INCLUDING POINTERS (which have a memory address, and stores data of another memory address other than its own).
  - So a double pointer could be pointed to by a triple pointer. Triple → double → single → normal variable.

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

3-11-2021 notes:

- Pointers continued (double and triple pointers)



4.

```
Double *x;
Double **y;
Double ***z;
Double a = 3.14;
X = &a;
Y = &x;
```

  - Notice how pointer* points to a normal variable's address
  - Pointer** points to a pointer* address
  - Pointer *** (z) would point to pointer** address
  - NULL is a special value for pointer variables; NULL = 0; 0 is not a valid memory address. Because of this feature NULL represents the end of a linked list
  - Dynamic arrays can be referred to by pointers
  - A pointer name is the equivalent of an array name; they are both pointers.
- Dynamic array

Dynamic array:

```
Int x, y;
Cin >> x  >> y;  // the values of x and y are determined at run-time
Float a[x];   // not ok in c++
Float *a = new float[x];   // ok in c++
For(int i=0; i<x; i++)  a[i] = I; // a is a pointer name and an array name
```

  - In java and c#, this is allowed.
  - For c++ we need to create a pointer first and assign it to a dynamic array as shown above
  - Whenever we use the "new" operator, we know that we are using dynamic memory allocation; it is particular for c++; for c, the equivalent is malloc().
    - Don't forget to do "delete []var_name" for any dynamically allocated memory (new operator used) in order to deallocate it; a new should always have a corresponding delete somewhere in the program to avoid memory leak.
  -
  -
  -

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- 2 dimensional dynamic array
  - Note, you can make a vector for any dimension.
  - ```
    Float **b = new float*[x]; // each time we peel off one star

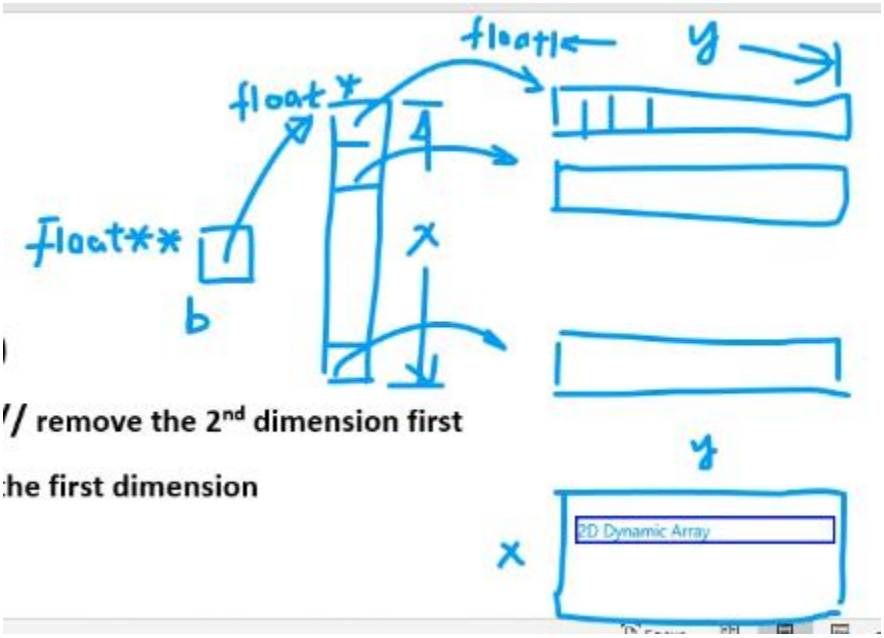
    for(int i=0; i< x; i++)

        b[i] = new float[y]; // b[i] --- float*; b[i][j] -- float
    ```
    - Notice how you have to allocate memory for the pointers* that the double pointer** points to, as well as memory for the pointer*s for the normal variables they point to.
    - Remember because we allocated memory dynamically using the “new” operator, we must use the “delete” operator to deallocate; DELETE THE LOWER DIMENSIONS FIRST, AND WORK YOUR WAY UP THE DIMENSIONS AND DELTE THEM ACCORDINGLY; otherwise if you delete upper dimensions first without deleting the lower dimensions, you will still have a memory leak.

    ```
    For(int j=0; j<y; j++)

        delete []b[j];  // remove the 2nd dimension first

    delete []b;  // delete the first dimension
    ```

    -
  - 

    - Notice how ** → * address, and * → normal var address.
    - Remember, all pointers point to the first address location in the array; this is what allows us to use array/pointer offset notation, offsetting in a direction relative to the address to which a pointer points, which is the start address of some vector/array.
    ```

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- 3-Dimensional Dynamic Array
    - Essentially the process is the same as that of 2D dynamic array; allocate memory for each dimension, starting from the first dimension and working your way down; when you delete, you start from the highest dimension and work backwards (work your way up) to deallocate so it is basically reverse of when you allocate memory.
    - so the difference between a 2D or 3D array, and a 2D or 3D Dynamic array is that you can manipulate the size of each dimension even at runtime by using the new and delete operators accordingly; for normal 2D and 3D arrays once you set the size you cannot change it during runtime…

```
// 3-D Dynamic arrays
Float ***c = new float**[x];
For(int i=0; i<x; i++)
```

```
C[i] = new float*[y];
For(int j=0; j<y; j
```

```
For(int i=0; i<x; i++)
```

```
C[i] = new float*[y];
For(int j=0; j<y; j++)
    C[i][j] = new float[z];

For(int i=0; i<x; i++)
    For(int j=0; j<y; j++)
        For(int k=0; k<z; k++)
            C[i][j][k] = i*j*k;
```

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Remember, the point of a dynamically allocated 1D, 2D, 3D…ETC…array is so that you can MANIPULATE AND CHANGE THE SIZE DURING RUNTIME. Static arrays must be known at compile time and cannot be changed once it is created.
- Remember: multi-dimensional arrays/dynamic arrays are actually stored in memory the same as a 1D array → in contiguous memory location, just separated logically into dimensions.

- VOID pointer
  - Can point to any data type; all pointers point to a 32-bit address, but for c++ when you do int * p, it is expecting p to point to a 32-bit memory address for integer data; void * pointer allows the pointer to point to the 32-bit address of any data type.

```
Void pointer:  it is a special type of pointers that point to a generic
data type.


Int *p1;  // p1 points to an integer variable

Float *p2; // p2 pints to a float variable

Void * p3;  // p3 points to any data type


Void swap3(void *p1, void *p2)

{


}
```

  -
    - We make the parameters of the function to void pointers so that the memory address of any data type can be passed in and the local function pointer will be able to point to the data (to access one address, or iterate through a passed in array of some data type)
    - We can use type_casting in conjunction with void pointers, it is a variable **powerful** tool (but can be **dangerous**):

```
Int comp_int(void *a, void *b)
{
    If( *(int*) a > *(int*) b)  // red-color: type casting of pointers
        Return 1;
    Else if ( *(int*) a < *(int*) b)
        Return -1;
    Else
        Return 0;
}
```

    -
    - The outer * represents the dereference operator (access the data stored at the given address stored by the pointer)

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

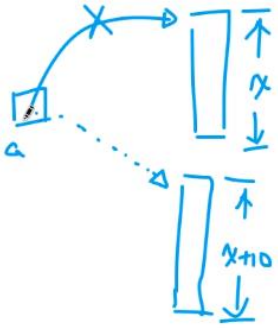
- Risks of using pointers:
  - Memory leak

```
}

Risks of using pointers:

(a)Memory leak
    Float *a = new float[x];
    a = new float[x+10];

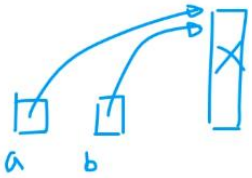

(b)      Dangling pointers
```



    - the new float[x] memory block is lost because it was not first deallocated using "delete" operator. It is a memory leak.
    - Solution: issue delete []a first, THEN you can allocate a new memory block for a to point to. (remember a pointer points to the FIRST address of an allocated memory block, but still, if you don't do "DELETE" then the entire dynamically allocated memory block will be lost → leaked.)
  - Dangling pointer

```
(b)      Dangling pointers
    Float *a = new float[x];
    Float *b = a;
```

```
Delete []a;
Cout << b[0] << endl;
```



    - B is a dangling pointer because the memory block pointed to by b is already deleted (when delete []a is issued); thus, we say that b is a dangling pointer; attempting to issue delete []b will cause an error, or attempting to access the elements of b will also cause an issue since it is a deallocated memory block.

3-15-2021 notes:

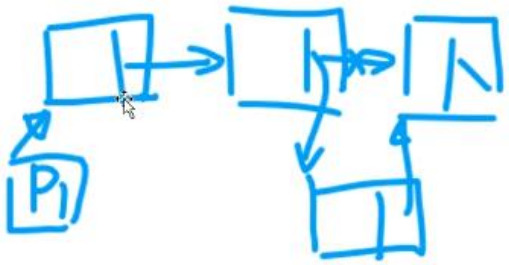
- Pointers continued:
- Linked list: no need to shift all elements when adding or deleting; simply change one of the linkages (remember we use a pointer to point to each node and access an object):

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

March 15, 2021

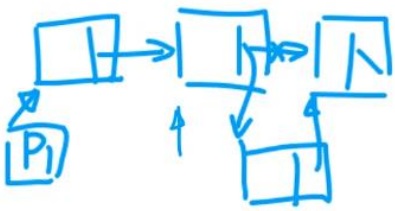
(1) Pointer (continued)

- **Linked List**




```
P1->next  = p1->next->next;
```

-
- //above statement == delete a node.
- Make sure when deleting a node, to delete the object associated to that node or else you will have memory leak.
- Weaknesses of linked lists: search time; we **must** do incremental (linear) search:




```
P1->next  = p1->next->next;  // delete a node
// traversal of a linked list
While( p1 != NULL)
{
   Cout << p1->value << endl;
   P1 = p1->next;
}
```

-
- Above is a linear search of a linked list; theoretically, you **CANNOT** do a binary search of a linked list.
    - There is a pseudo way of searching a linked list using binary search; by associating the linked list to an array if the linked list is infrequently

changed and of course if it is sorted. There are some other ways that can help you to jump around a link list, but for the most part, you can only do a linear search of a linked list.

- Abstract data types:
- Stack
  - Useful arrangement for data that we frequently use because of its practical application
  - Restriction/Pre-condition: You are only allowed to add and remove items from the **TOP**. (FILO or LIFO)
    - First in last out, or equivalently, Last in first out
  - Push and pop action at the top of the stack
  - Push: add one item to the top of the stack
  - Pop: remove one item from top of the stack
  - Examples:
    - a calling stack for recursion
    - a stack for transformation sequences (translation, rotation, scalar); used in simulation and gaming software
  - avoid naming a class with keyword stack…use stack1 or stack2 etc…visual studio uses stack as a keyword.
  - 5 major member functions:

Stack ADT Operations

- MakeEmpty -- Sets stack to an empty state.
- IsEmpty -- Determines whether the stack is currently empty.
- IsFull -- Determines whether the stack is currently full.
- Push (ItemType newItem) -- Throws exception if stack is full; otherwise adds newItem to the top of the stack.
- Pop (ItemType& item) -- Throws exception if stack is empty; otherwise removes the item at the top of the stack and returns it in item.

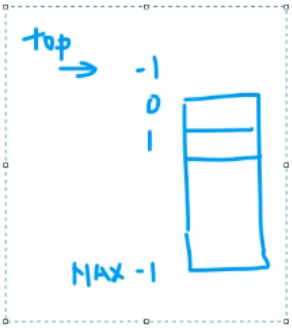
  - A stack is 1 dimensional; it is a list; so you can use an array (static or dynamic) or linked list to represent a stack

```
Structure:
Class Stack1
{
 Private:
    // static array: use an array name
    // dynamic array: use a pointer
    // linked list:  use a pointer that points to the beginning of the linked list
 Public:
     makeEmpty( )
     isFull( )
     isEmpty( )
     push(  )
     pop( )

 }
```

  - Choose between static array, dynamic array, or linked list to represent the stack

- Remember to make pointers for dynamic arrays and linked lists point to the start of the vector, meaning you have to set the pointer equal to -1 since the first element in a list is element 0.



- For push and pop functions, you don't need to return a value; simply reduce or add to the value of TOP variable, and if you need to obtain a value stored in the stack, then use pass by reference as applicable; also you do not need to clean up array elements when you use pop function unless for some reason it is necessary.
- Don't forget, you can define a template implementation of stack since stack is a ADT so frequently used

```
Template <class ItemType>
Class StackType
{
    ...
}
```

- StackType<instantiation variable> name_of_Stack
- `Template <class ItemType>  // template <typename ItemType>`
  - Remember **class** and **typename** when declaring a template class are interchangeable terms you can use as shown above
- Also remember when you use scope operator to define a template class's functions outside of the declaration of the template class, you must use the key phrase "template <class (or typename) ItemType> in front of every function definition with the scope operator.

- Using a dynamic array for a stack structure:

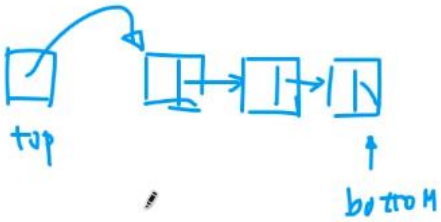
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
Template <class ItemType>  // template <typename ItemType>
Class StackType
{
    Private:
      // dynamic array
      Int top;
      ItemType *items;
    Public:
      StackType() { items = new ItemType[len1];}
```

- Of course, Declare a pointer for using a dynamic array, and then for the default constructor you must assign a block of memory for the pointer to point to.
- Don't forget to overload destructor in order to delete dynamically allocated memory when the object is destroyed (otherwise you will have a memory leak).
- You need to have an int to tell size of the array and traverse the list
- **~StackType() { delete []items;}**

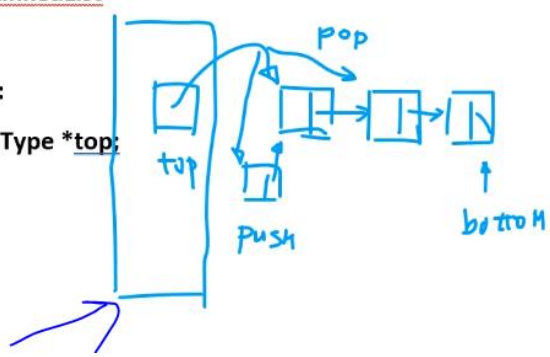
- Using a linked list for a stack ADT structure
  - Use a pointer to point to top of the stack (head node)

```
Class StackLinkedList
{
    Private:
      ItemType *top;
```



    - / = NULL; the bottom of the stack (linked list) is represented by NULL
  - If we do a push or pop, simply move the top pointer accordingly:

```
Class StackLinkedList
{
    Private:
      ItemType *top;
```

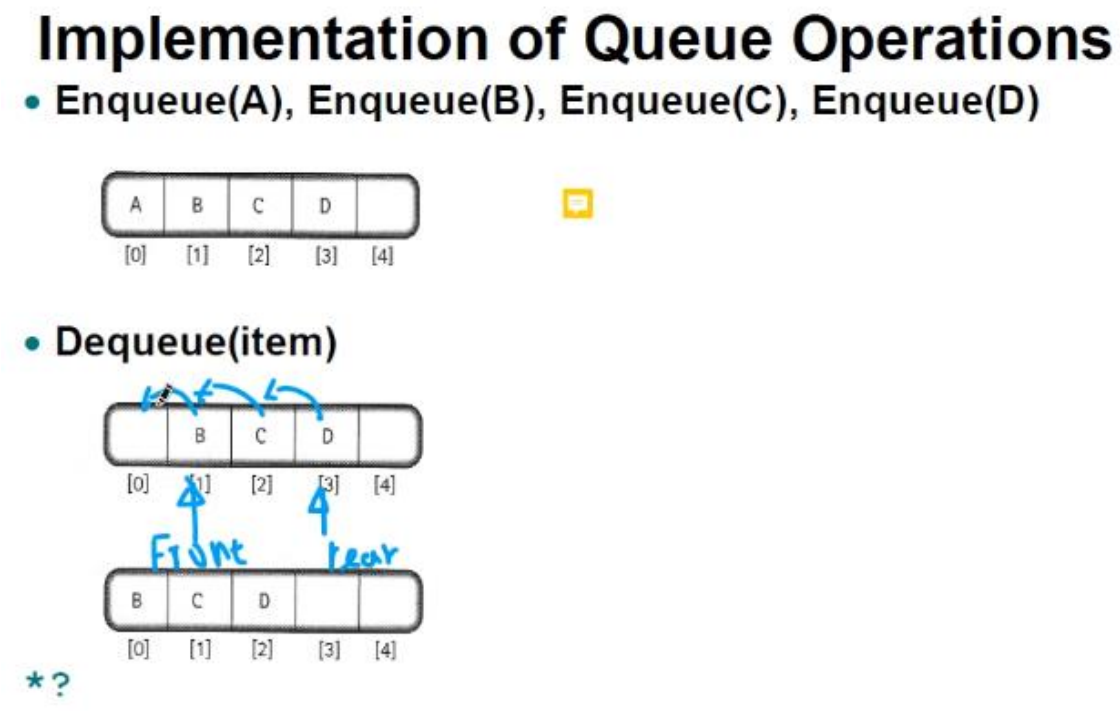


    - Arrow points to the instance of the class
  - Do not need an int to represent size of the linked list
  - But it is good to use int to track length anyways to help with efficiency

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

3-17-2021 notes:

- **Queue ADT**
- Applications:
  - Waiting queue for processes on a computer
    - OS has a queue that coordinates with the CPU to determine how to complete tasks and which tasks need to completed next (based on priority)
  - A queue in simulation
- FIRST IN FIRST OUT (FIFO) or LAST IN LAST OUT (LILO) [generally this is the case for a queue]
  - Add elements only to the back of the queue, elements leave only from the front of the queue
- Functions:
  - Transformer functions:
    - **Enqueue**: add an element to the end of the queue
    - **Dequeue**: removes an element from the front of the queue
    - Make empty
  - Observer functions:
    - isFull
    - isEmpty
- implementation of a queue
  - can use a static array, dynamic array, or a linked list pointer (head -> front, tail -> rear)
  - implementation using an array:



    - Disadvantage of using an array or a single pointer: when you dequeue because all elements have to be shifted left (or right depending on what is the front or back of the queue); remember the goal is efficiency in memory usage.

(2)Queue

Applications: waiting queue for processes on a computer, a queue in simulation

Characteristic: FIFO (First In First Out)

Implementation: static array, dynamic array, or linked list (head→ front, tail → rear)

A. Array Implementation

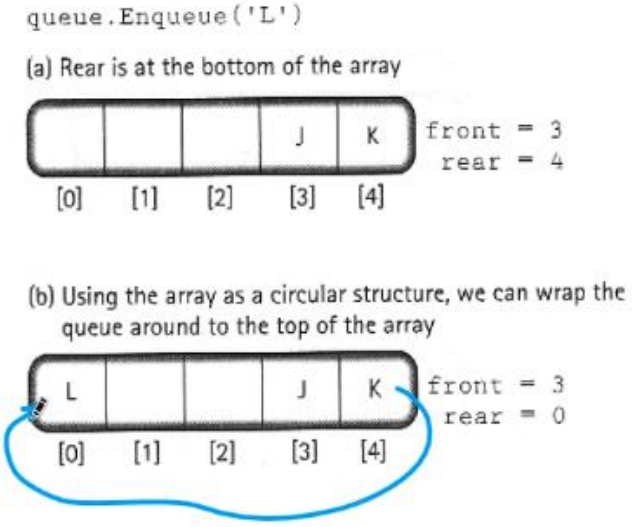
A.1 simple implementation (drawback: too many shifting operations)

A.2 second implementation (introducing two data members: front and rear)

- Solution is second implementation: use 2 variables to track the array elements; one for front, and another for pointing to back of the array.
  - You can also use wrap around for both variables
  - In this way if you enqueue using a variable where wrap around is necessary then elements can still be added to the front if there is space.

Another Queue Design

- **Wrap around**



Another Queue Design

- **Formula for Wrap-around**

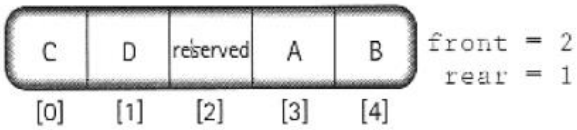
```
if (rear == maxQue - 1)
  rear = 0;
else
  rear = rear + 1;
```

- **rear = (rear + 1) % maxQue**
- Above are two ways you could implement wrap-around

- The second implementation is usually sufficient queue for all queue ADTs but there is one drawback but it is an exception case:
  - There can be ***ambiguity*** between if the status of a queue is full or empty because they can overlap in the first and second pointer values/location so that you don't know whether it is full or empty.
  - Solution is a third implementation which is exactly like the second implementation, but introducing length of queue variable or introducing a reserved element.
    - We put the reserved element in front the of the first element location of the queue.

Another Queue Design

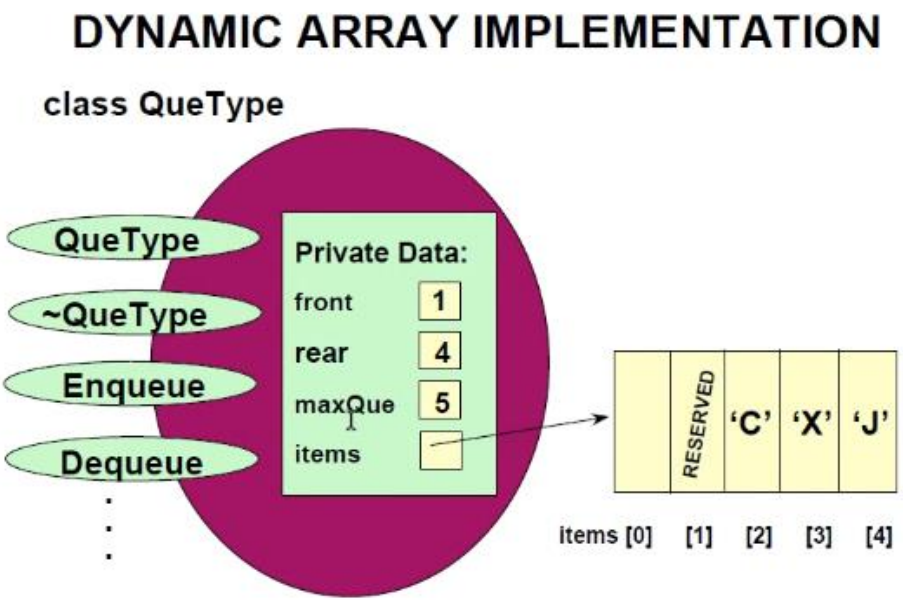
- **testing for a full queue**



- **the element preceding the front of the queue is reserved.**
- **We initialize front to maxQue – 1 and rear to front, i.e., maxQue – 1.**
- **?**
- Reserved meaning that you cannot put any data into the reserved element

- Consider true front as the location of the reserved element
- When front == rear it means queue is empty
- When front = reserved element and rear = 1 less than reserved element then the queue is full

- Dynamic array implementation



- Use a pointer to point in the dynamic array; it is exactly like the third approach/solution by introducing a reserved element or length variable; only now you have to dynamically allocate (and of course delete) memory and of course store the first address of the memory block in a pointer variable. However, you still use INT variables to move between front and rear of the dynamic array (and of course also the max-size of queue); so it is almost exactly the same as the 3rd approach from static array only you have to allocate the memory.
-

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
//--------------------------------------------------------
// CLASS TEMPLATE DEFINITION FOR CIRCULAR QUEUE
#include "ItemType.h"       // for ItemType
template<class ItemType>
class QueType
{
public:
   QueType( );
   QueType( int max );  // PARAMETERIZED CONSTRUCTOR
   ~QueType( ) ;     // DESTRUCTOR
  . . .
   bool IsFull( ) const;
   void Enqueue( ItemType item );
   void Dequeue( ItemType&  item );
private:
   int       front;
   int       rear;
   int       maxQue;
   ItemType*  items;    // DYNAMIC ARRAY IMPLEMENTATION };
   ItemType items[MAX_ITEM];    // static array implementation
```

-

```
//--------------------------------------------------------
// CLASS TEMPLATE DEFINITION FOR CIRCULAR QUEUE  cont'd
//--------------------------------------------------------
template<class ItemType>
QueType<ItemType>::~QueType( )
{
   delete [ ] items;     // deallocates array
}

.
.
.

template<class ItemType>
bool QueType<ItemType>::IsFull( )

{                        // WRAP AROUND
   return ( (rear + 1) % maxQue == front )
}
```

-
- Remember for constructor you must allocate memory for the pointer to point to (dynamic array) based on the size specified
- Make sure to use a length variable or reserved element

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
//--------------------------------------------------------
// CLASS TEMPLATE DEFINITION FOR CIRCULAR QUEUE  cont'd
//--------------------------------------------------------

template<class ItemType>
void QueType<ItemType>::Enqueue( ItemType newItem )
// Post: newItem is at the rear of the queue
{
   rear = (rear + 1) % maxQue;
   items[rear] = newItem;
}

template<class ItemType>
void QueType<ItemType>::Dequeue( ItemType &newItem )
// Post: The front of the queue has been removed and a copy
   returned in item
{
   front = (front + 1) % maxQue;
  newItem = items[front];
}
```

- If you don't include wrap-around code as done above then you will have to use an if statement; its easier to incorporate wrap-around

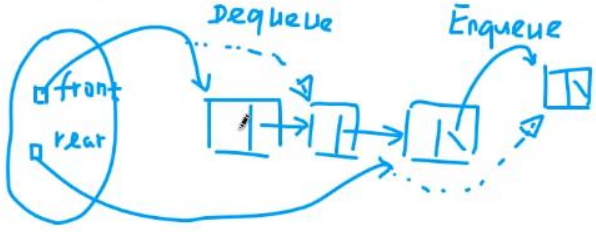
- Implementation by a linked list:
  - Use two pointer variables; front and rear to point to rear and front node of the first and last elements in the queue.

A.3 third implementation (introducing another data member: length of the queue or introducing a reserved element)

We put the reserved element in front of the first element of the queue; consider front = the location of the reserved element.

B. Implementation by Linked List

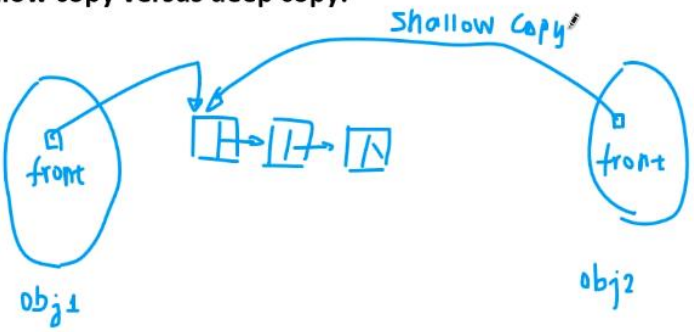
Use two pointer variables: front and rear



- Shallow copy v deep copy
  - Default copy (assign operator and copy constructor) is a shallow copy (memberwise copy).

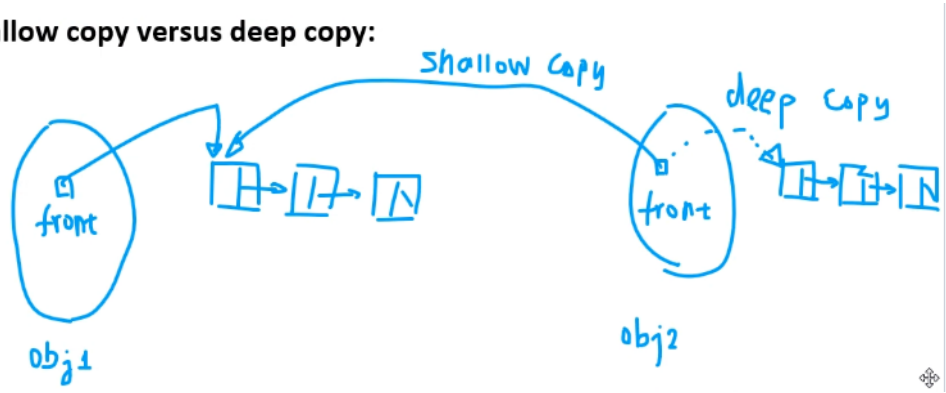
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Shallow copy versus deep copy:



- Be wary of this because if you delete obj2 pointer, then obj1 pointer will point to nothing (or vice versa) since a shallow copy was done involving pointers.
  - This will cause a dangling pointer
  - Solution is: make a **deep copy**: create an independent linked list for obj that copies all the values from the data that obj1 points to but that will point to its own separate memory address block/linked list.
  - This will allow the values of obj1 and obj2 to be equivalent but still independent of each other since they point to separate memory address space (just as it is when there are no pointers involved when performing a copy).

Shallow copy versus deep copy:



- Deep copy only needed whenever you have a linked list/pointer as a data member since a default (shallow) copy will simply cause the pointer members to point to the same address space.

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
template<class ItemType>        // COPY CONSTRUCTOR
StackType<ItemType>::
StackType( const StackType<ItemType>& anotherStack )
{  NodeType<ItemType>* ptr1 ;
   NodeType<ItemType>* ptr2 ;
   if ( anotherStack.topPtr == NULL )
    topPtr = NULL ;
   else                // allocate memory for first node
   {topPtr = new NodeType<ItemType> ;
    topPtr->info = anotherStack.topPtr->info ;
    ptr1 = anotherStack.topPtr->next ;
    ptr2 = topPtr ;
    while ( ptr1 != NULL )    // deep copy other nodes
    {   ptr2->next = new NodeType<ItemType> ;
        ptr2 = ptr2->next ;
        ptr2->info = ptr1->info ;
        ptr1 = ptr1->next ;
    }
    ptr2->next = NULL ;
   }
}
```

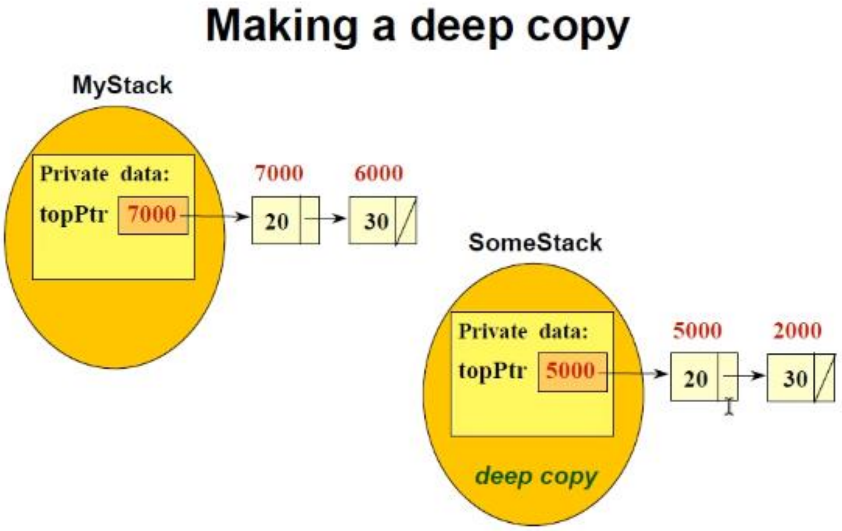
- Above is the implementation example of a deep copy; notice how ptr2 points to its own dynamically allocated memory but copies the data stored at the dynamic array of ptr1.
- Rely on ptr2 = ptr2->next and ptr1 = ptr1->next to iterate through each linked list; don't forget ptr2->next = NULL to close the list when the deep copy is finished.

Note that the deep copy is needed only when a data member is a linked list.

Deep copy: It is basically a while loop that traverses the old linked list and during each step of the traversal, we create a new node and copy the value from the old linked list to the new linked list. Repeat the process till the end of the old linked list and at the last step, set the next field of the last node of the new linked list to NULL. We rely on two pointers: p1 and p2; the first one points to the old linked list and the second one points to the new linked list.

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- How to know location in linked list?
    - There should be a top pointer (variable that points to first element address in the linked list) which is why when you search the linked list you need a **separate** pointer variable to traverse the linked list:



    -
        - Note that topPTR in above will never be changed; it should always point to the start of the linked list so that we can keep track of where the linked list begins.

new linked list. Don't use topPtr in the while because it changes its value and we will lose the track for the beginning memory

address of the linked list. You should always use p1 and p2 (temporary pointer variables in the while loop).

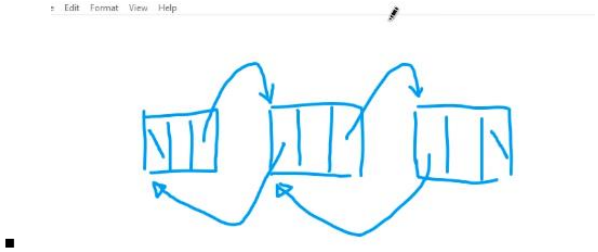
3-22-2021 notes:

- Lists by linked lists

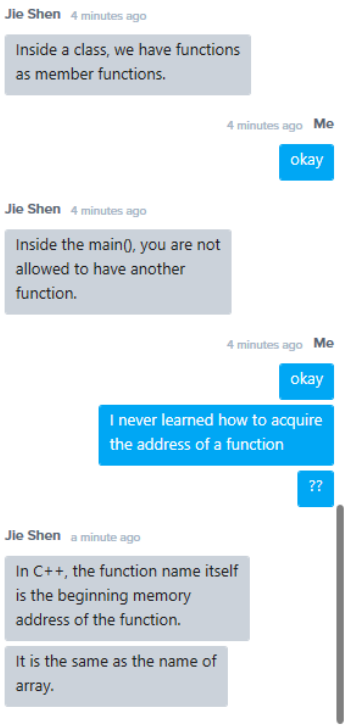
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

LAB 8 NOTES:

- Circular linked list: tail points to head
- Doubly linked list: each node contains 2 pointer: 1 points to next node, the other points to previous node
    - A doubly linked list can be circular or non circular
    - Below is a noncircular doubly-linked list:



- For c++, the function name is the beggning memory address of the functions storage location
    - Example: int func(void){…}…. → func is the beginning address of the function

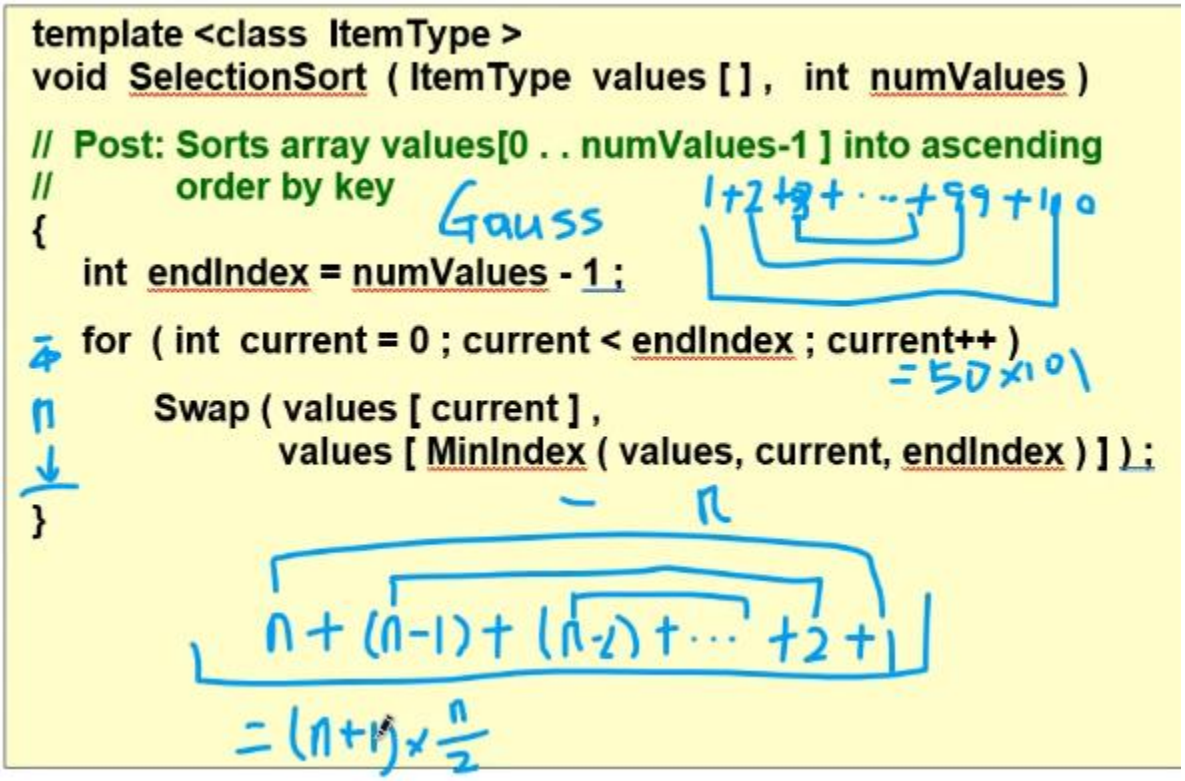


3-24-2021 notes (week 12):

- Sorting algorithms (with arrays)
- When calculating complexity, simply look at the loops and the recursive functions
- Always the largest/most noted loop will count as the complexity. For example if one nested loop has complexity n^2, and another single loop with n, then the complexity is just n^1.
    - Same goes for coefficients, if you have two nested loops with n^2, then complexity is simply 2*n^2, so essentially n^2; 2* is just a coefficient.

- So always take the highest degree term; for nested loop and single loop: n^2 + n → complexity is just n^2; for two double nested loops; 2*n^2 → complexity is just n^2.

- For sorting, make sure to use the currentPos integer to distinguish between sorted and unsorted portions of the list
- Advanced demonstration of solving for time complexity of a function:



  - Above, the complexity can be found by finding the shown sum using a Gauss Estiamtion.
    - So ignoring coefficients, you see that the above sum simplifies to n^2. Thus time complexity is Big O(N^2).
    - Notice how time complexity is larger because each time you sort one value, you still have to iterate through the remaining of the entire list (the unsorted part) using MinIndex to find the smallest value and sort it each time until the list is sorted
- There are many types of sorting algorithms: selection sort, insertion sort, bubble sort, etc..
  - Insertion sort: use two stacks (2 arrays, an old and new array), at the end of the function the old will become empty and the new will become full with the sorted list
- Often types sort algorithms will need two to three function parameters

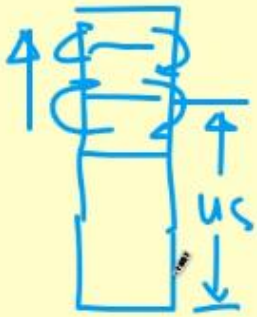
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
    template <class  ItemType >
void InsertItem  ( ItemType  values [ ] ,  int  start ,  int end )

// Post: Elements between values [start] and  values [end]
//       have been sorted into ascending order by key.
{
    bool  finished = false ;
    int    current  =  end ;
    bool  moreToSearch = ( current != start ) ;

    while  ( moreToSearch  &&  !finished )
    {

        if  (values [ current ] < values [ current - 1 ] )
        {
         Swap ( values [ current ], values [ current - 1 ] ) ;
         current-- ;
         moreToSearch = ( current != start );
      }
      else
         finished = true ;
    }
}
```



-
  - Notice how you can set a Boolean equal to some expression that will be evaluated as true or false
- Quick sort (similar idea to a binary search)

Quick Sort

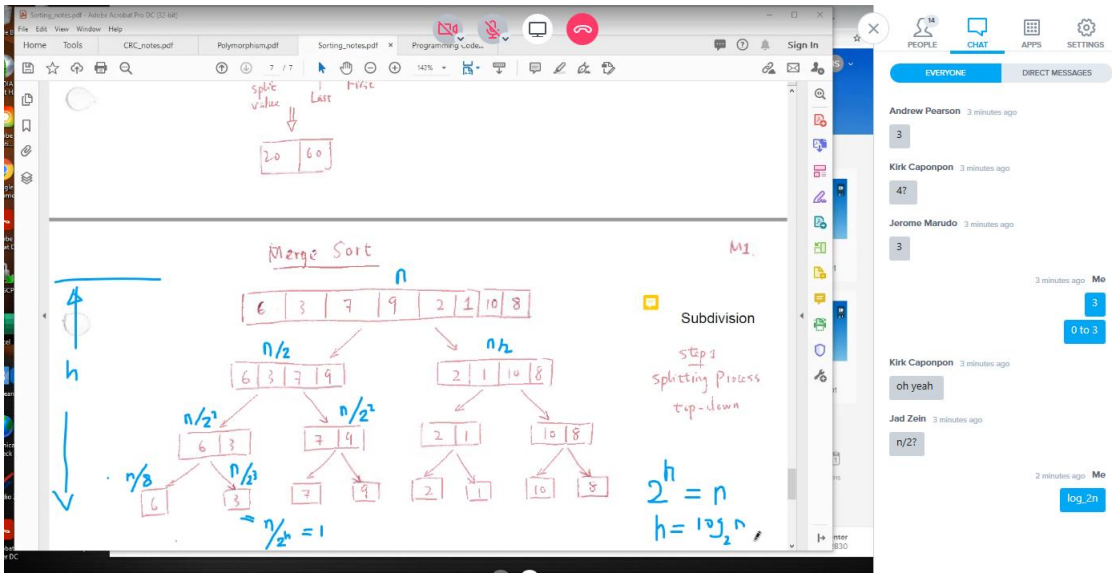
- If there is more than one item in values[first]..values[last]
  - Select splitVal
  - Split the array so that
    - values[first]..values[splitPoint-1] <= splitVal
    - values[splitPoint] = splitVal
    - values[splitPoint+1]..values[last] > splitVal
  - QuickSort the left half
  - QuickSort the right half
  -

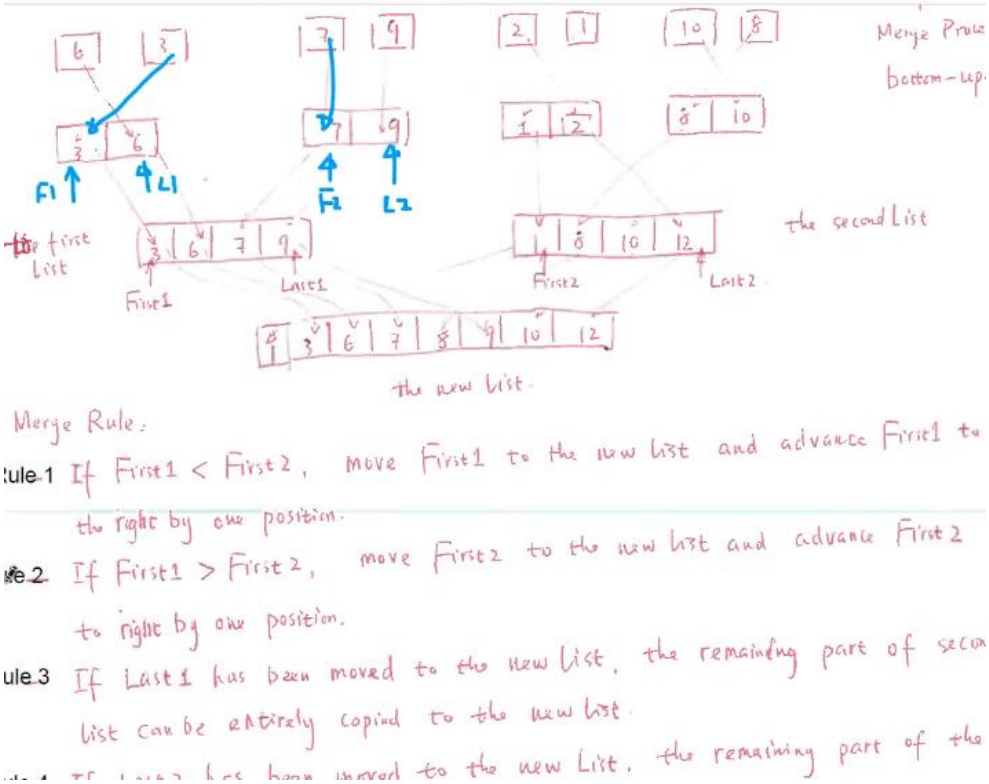
- Send all smaller values to one half (say at start of the list), and all other larger values to the other half (say, attach them to the end/, then you will have to small unsorted lists but with one list larger or smaller than the other
    - We have to make sure smaller element (the first element = split value) is smaller than the middle element; if not, then we swap.
    - Then we compare all elements in the larger half to the first element in the smaller half and make sure that it is larger than the first element from the smaller half
        - if a value from the larger half is smaller than the split element (first element in small half), then we do a swap of the element following the split value from the smaller half with the smaller element that was found in the larger half
        - once you check all values up to the middle element (which we should already know that the first (split) element in smaller half is smaller than the middle element) then you know the list is sorted properly into a larger and smaller half
        - then at this point, you sort the large and smaller halves of the list separately using swap functions
            - since we have a small and large half, then when sorting each half separately, we do not need to check at least half of the elements of the full list to sort each half
            - this function still works even with just 2 elements
                - (first and last will point to same element, element 1,
                - The split value of course will be element 0
                - Use wrap around when you move from last element forward (wrap around to first element)
- Best case is when the list has an even number of elements
- Many times you will swap first and last value

3-29-2021 notes (week 13):

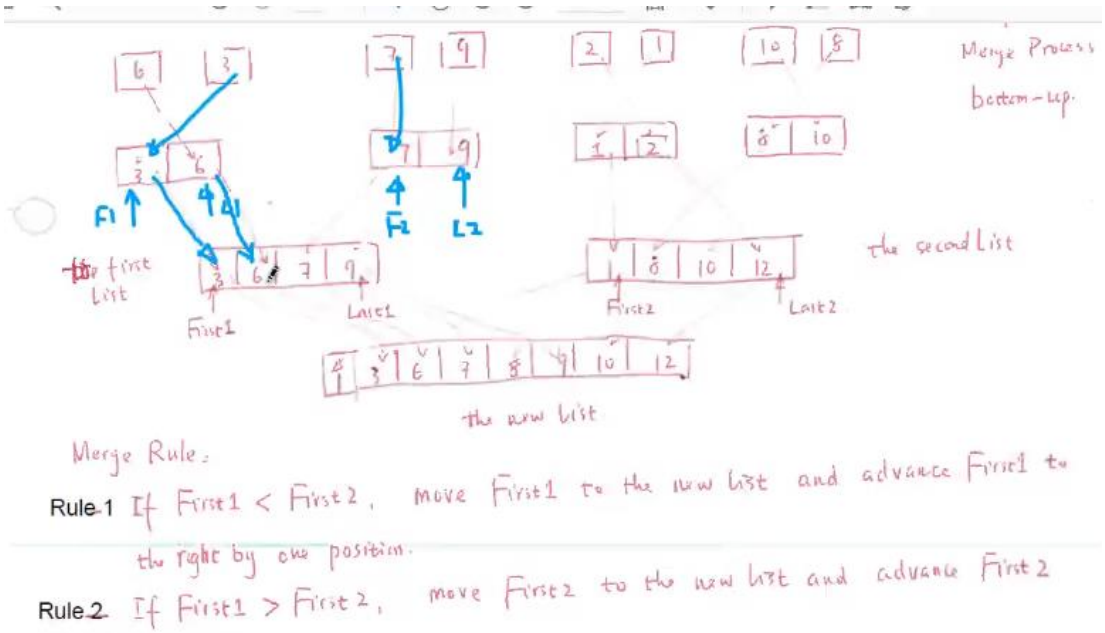
- Quick sort time complexity is normally n(logn); but worse case scenario is n^2
- Merge sort always fall in n(logn); also another divide and conquer approach (subdivision/split process)
    - For a given n (n elements) height of the tree is h = log_2(n)
    - GAURANTEES NLOGN TIME COMPLEXITY NO MATTER WHAT TYPE OF LIST is input and need to be sorted
    - How to implement merge sort with minimum memory usage??? EXAM QUESTION POSSIBILITY..which is best data structure/approach to use

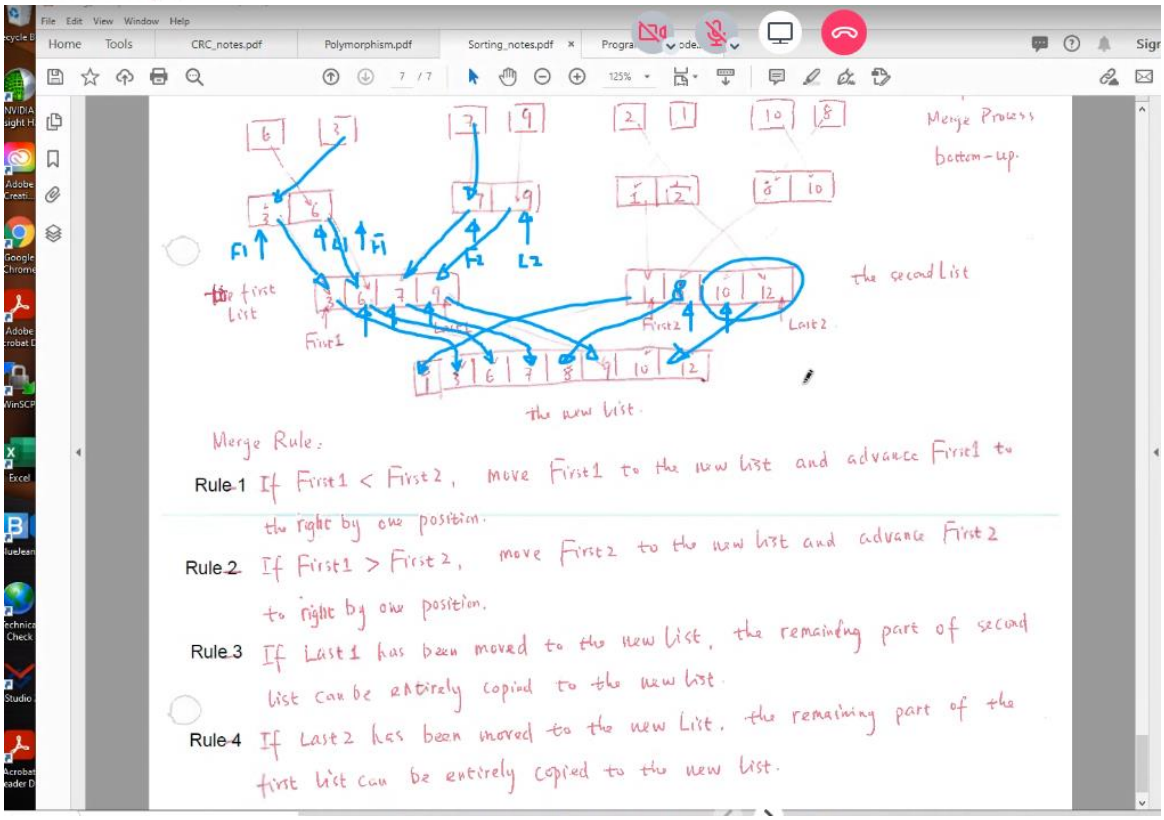
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- 

  - Also this is why O = N*LOGN
- 

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

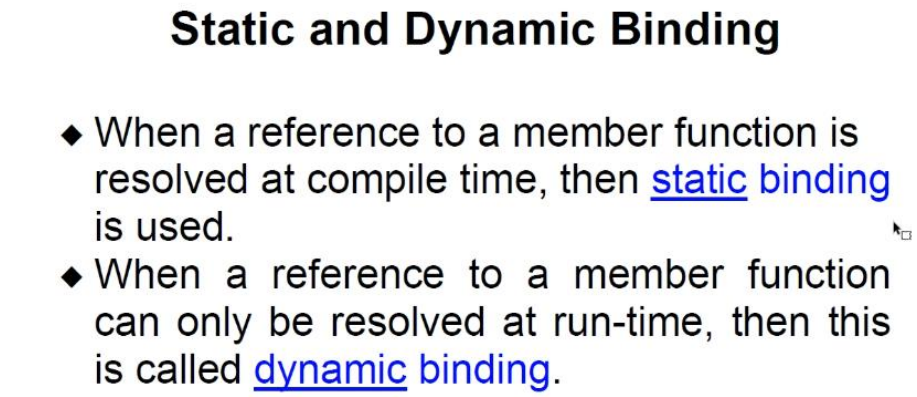
- 

- 

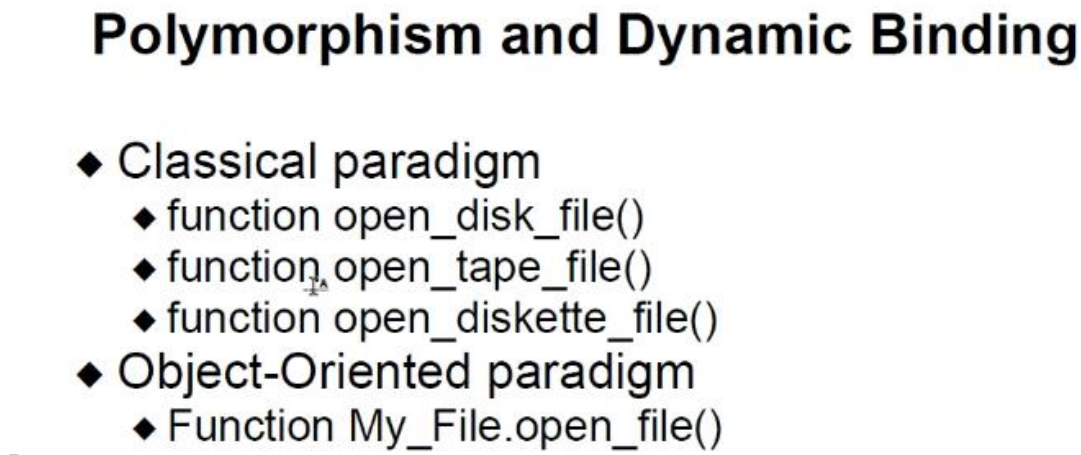
- Strong Programmer tips and program modifiability
  - Avoid global variables because their scope is too wide and so it can make it more difficult to track and debug your program
  - CRC cards: CLASS, RESPONSIBILITY, and COLLABORATION cards
    - You can use this before designing UML; it is similar to UML class diagram
    - Has a function/responsibility section, and a collaboration section
-

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Polymorphism (the 3rd cornerstone of obj-orien programming)
  - 1st cornerstone: Data abstraction
  - 2nd cornerstone: Inheritance (hierarchical class structures)
  - 3rdcornerstone: polymorphism (dynamic binding)
    - Example: a function that can handle several different types of input, like a shape function that can output info based on the shape (object type) passed in
    - Main purpose for polymorphism to make your code more elegant; instead of lots of if-else statements, we instead use one instance of the same function that takes on many forms that handles its respective input.



    - Benefit of dynamic is flexibility by ability to change depending on user input

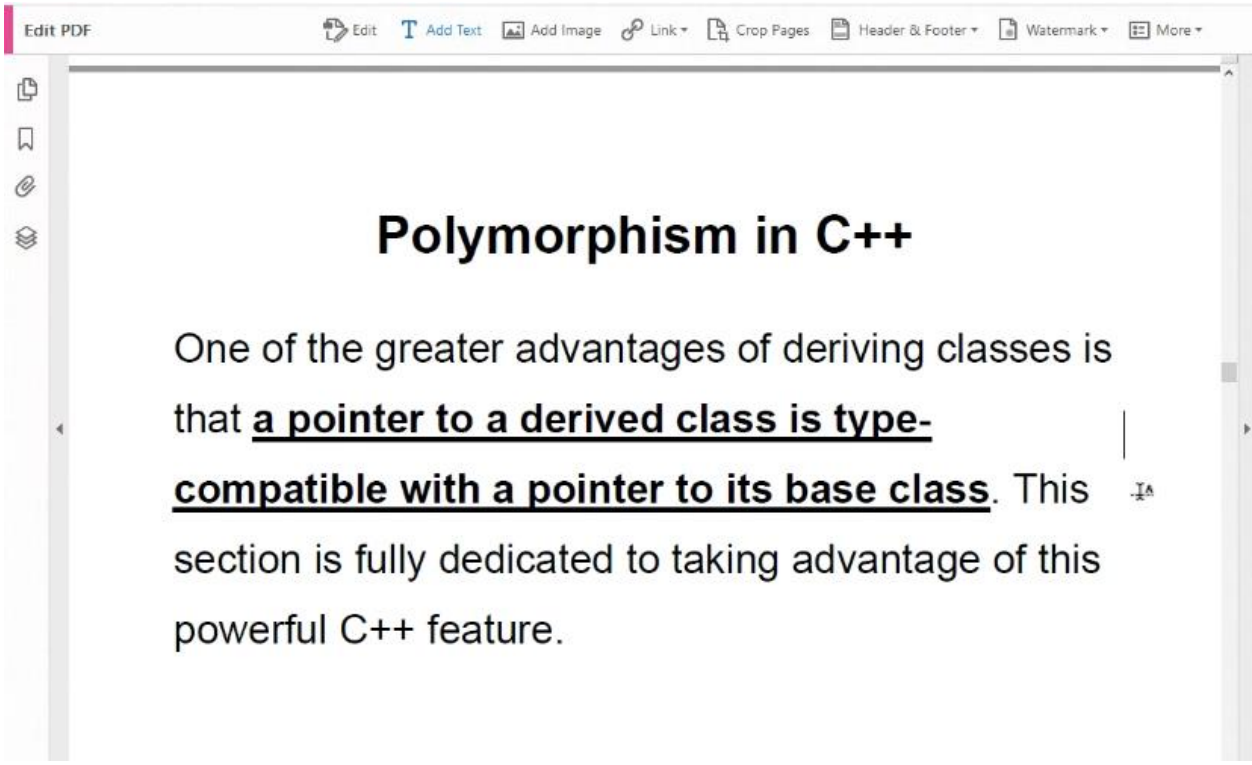


      - Above you can change those 3 functions into one function that can handle all 3 inputs
    - Drawbacks: it increases difficulty to beginners and reading others code when you use polymorphism
- Polymorphism in c++ (java and c# automatically uses polymorphism)

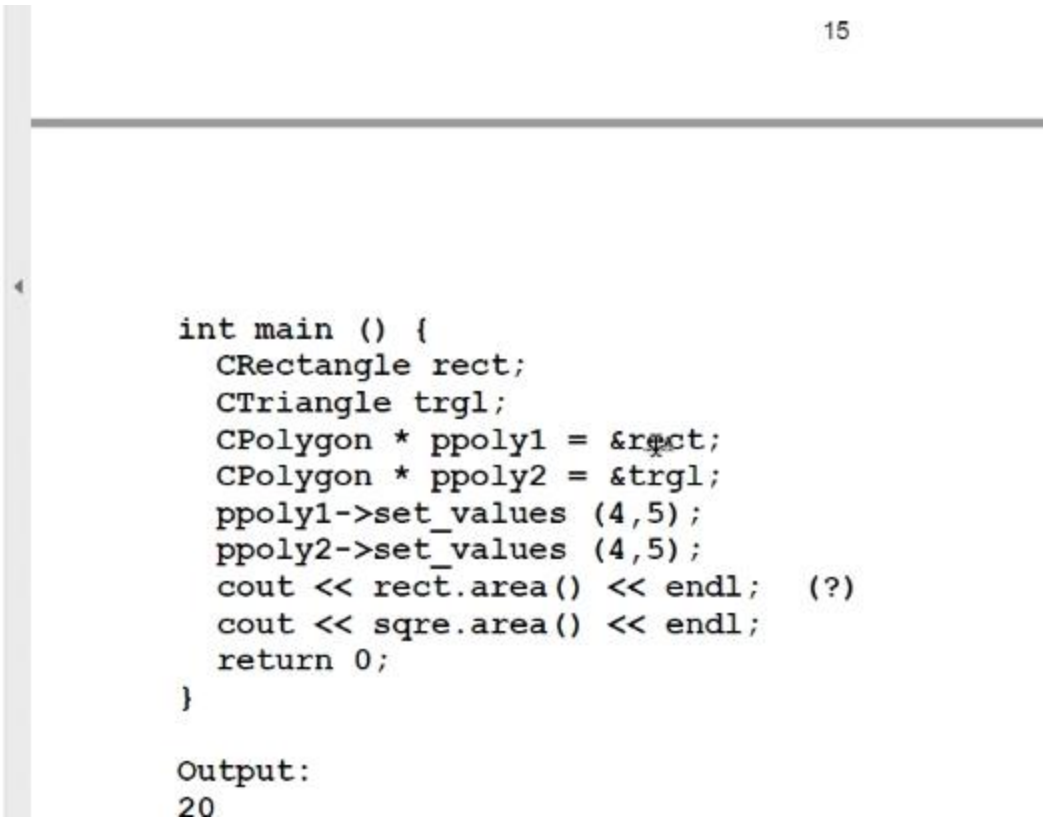
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Polymorphism in C++

- ◆ Virtual functions
- ◆ Abstract classes

-
- Need also pointers to base class
- IF YOU KNOW VIRTUAL FUNCTIONS, YOU KNOW C++; JOB INTERVIEWERS WILL OFTEN ASK YOU TO EXPLAIN VIRTUAL FUNCTIONS TO TEST KNOWLEDGE OF C++ AND POINTERS. YOU MUST KNOW THEM.
- 

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



```
int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;   (?)
  cout << sqre.area() << endl;
  return 0;
}

Output:
20
```

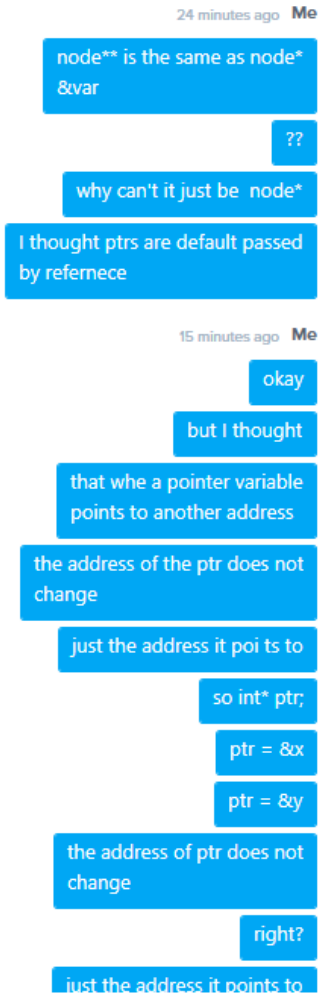
- Above, CTRAINGLE AND CRECTANLGE are derived from CPOLYGON
  - So since triangle and rectangle are derived from polygon, a polygon pointer (base class ptr) can point to derived class addresses

LAB 9 NOTES:

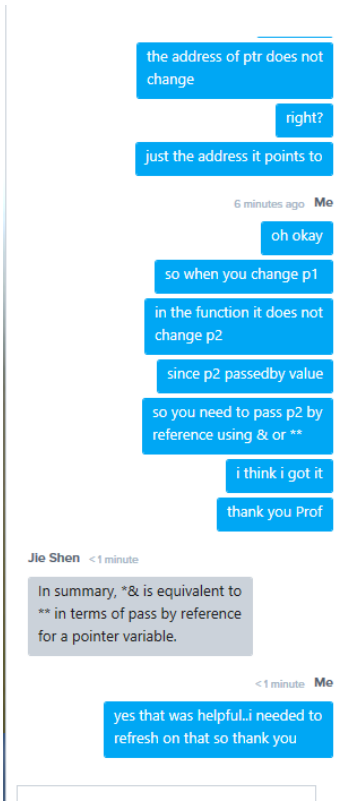
- Copy code from online lecture notes for question 1
- Question 2
  - Qsort function is available in VS
  - Use type casting with void pointers
- Question 3
  - Apply the 4 rules
  - Use linked list as one way to do merge sort
    - Put data in singly linked list
    - Do subdivision process (write a division function)
      - You can use null to separate divisions instead of creating separate linked lists
    - Then do merging process; starting from smallest divisions working to larger divisions
    - Use double ptrs to point between logical ptr subdivisions of the single list and to use pass by reference; we can do node** var….or node* &var.
      - Also need ** or *& so that the pointer passed to the function will be passed by reference and not by value; that way when the function pointer is updated so is the ptr passed to the function

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

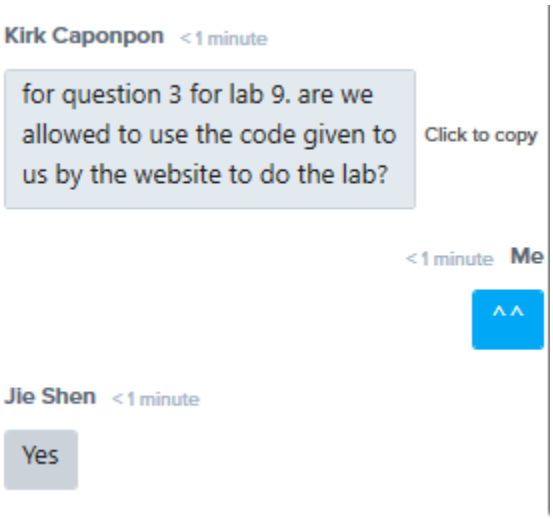
- Remember double ptr is ptr to a ptr
- In summary, *& is equivalent to ** in terms of pass by reference for a pointer variable.
- Remember, just int * x in a function parameter creates a local ptr variable that points to a normal data variables address; when the address pointed to by x is changed, the address pointed to by the passed in pointer if a ptr is passed in remains unchanged.
    - Ex:
    - func(int* x)
    - Int* y = &data_var
    - Func(y); //y is passed in by value; essentially just the address pointed to by y is assigned to the local function variable x.



CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



- Question 3:



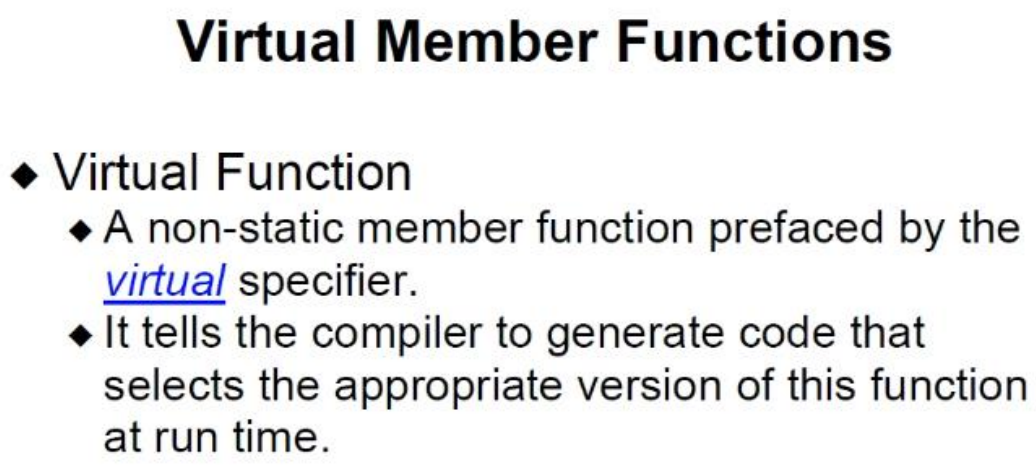
  -
  - Just make sure you understand the elegant implementation of the code to use merge sort

3-31-2021 notes (week 13):

- ***Polymorphism (continued)***
- Remember: pointer of type base-class can point also to all of its derived class-types
  - But also remember: fundamentally all pointers simply point to an integer (a 32-bit address) so virtually any pointer can point to any data type; thus a derived-class pointer

should be able to point to a base-class object too! If c++ allows it; otherwise we can use type-casting

- Virtual member functions



  - Used to **override** a function from base class with same name
    - It is a form of *dynamic binding* that is very ***similar*** to function overloading: but instead of calling the proper instance of the function based on parameters in the same class, the *virtual* keyword in the base class means that if the *virtual* function is called and it is called from a **derived** class, then go to that instance of the function which was redefined (but has same name) in the derived class and use its definition of the function to execute the function called. Otherwise, actually use the virtual function code defined in the base class if a base-class object calls the virtual function.
      - Derived class call of virtual function → go to function name from derived class (do not use the code for base-class defined class where the virtual function is created)
      - Base class call of virtual function → use the actual virtual function defined in base class
      - ***We normally put the virtual function inside the base class.**
- **Pure virtual function**

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
// virtual members
#include <iostream.h>
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void)
      { return (0); }
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };
```

- Notice above the virtual function's {} is replaced with = 0. When you set a virtual function = 0 then it is called a ***PURE VIRTUAL FUNCTION***.
  - THUS: the class becomes an ***abstract class;*** then you are **NOT ALLOWED** to have an *instance (declare an object of)* of the class; it's only purpose is to serve as a base-class to help define many derived classes that will actually be used
  - **also** if a derived class does not define (implement) the Pure Virtual Function declared from the base-class, it also becomes an ABSTRACT CLASS → no instance of this object is allowed either.

Abstract Classes

- An abstract class is a class that can only be a base class for other classes.
- Abstract classes represent concepts for which objects cannot exist.
- A class that has no instances is an abstract class
- Concrete Classes are used to instantiate objects

- Abstract classes should contain at least 1 pure virtual function (a function that cannot be used because it has no code; if a

derived class also does not define code for the virtual function, then the function remains a pure virtual function and the derived-class becomes an abstract class as well)

An Abstract Derived Class

- If in a derived class a pure virtual function is not defined, the derived class is also considered an abstract class.
- When a derived class does not provide an implementation of a virtual function the base class implementation is used.
- It is possible to declare pointer variables to abstract classes.

Abstract Classes

- In C++, an abstract class either contains or inherits at least one pure virtual function.
- A pure virtual function is a virtual function that contains a pure-specifier, designated by the "=0".

Example

```
// abstract class CPoligon
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
  };
```

```
CPolygon poly;
CPolygon * ppoly1;
CPolygon * ppoly2;
```

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

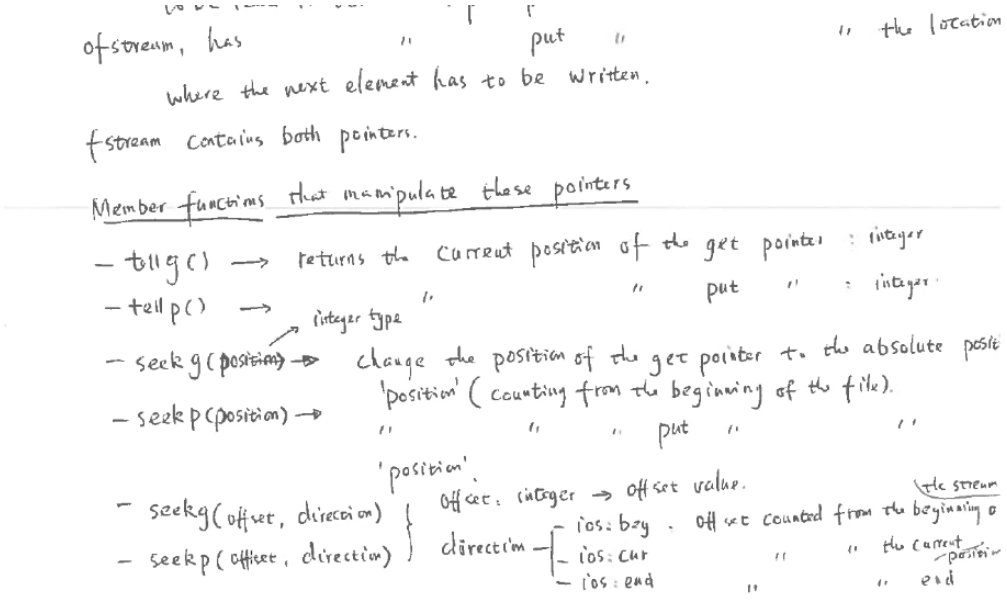
An Abstract Derived Class

- If in a derived class a pure virtual function is not defined, the derived class is also considered an abstract class.
- When a derived class does not provide an implementation of a virtual function the base class implementation is used.
- It is possible to declare pointer variables to abstract classes.

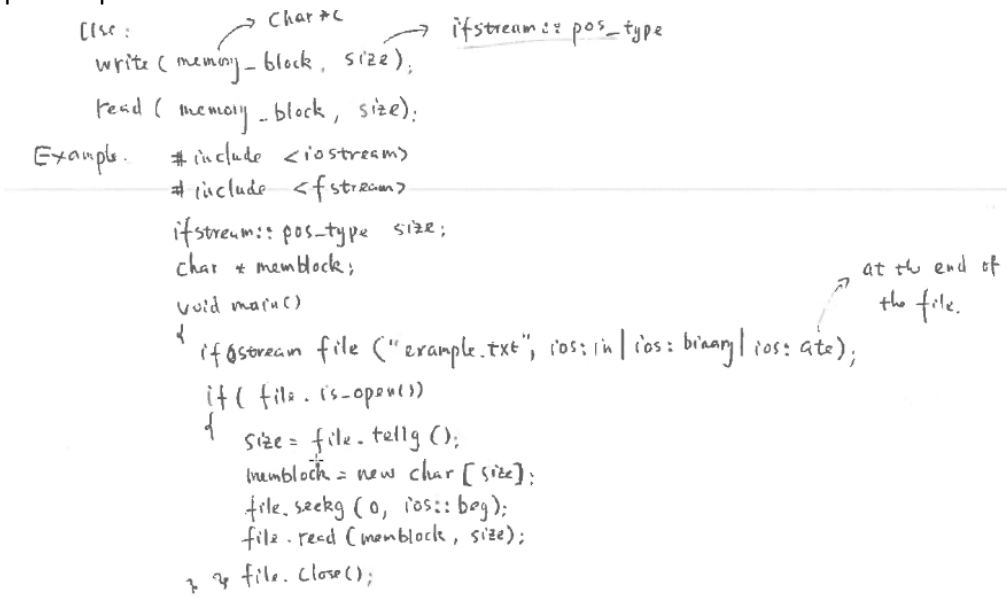
-
  - Simply declare a point of type base-class (the abstract class), then you can use it to point to derived-class address and make use of *dynamic binding* (calling the pure virtual function which in the derived class it will have an implementation of the function since the base-class pointer is pointing to an instance of derived class; we know derived class is not an abstract class like the base class is since an instance of derived-class is declared and being pointed to by the abstract-class base-lass pointer)

- Input and Output with Files
  - Include <fstream> //for files
    - Ofstream and ifstream derived from this class
      - All above classes derived from: Include <iostream> //for PC screen output and keyboard input
  - For files, we have to file pointers: <u>one for read, the other for write (get and put pointers)</u>
  - Remember that all pointers including the get and put pointers for files, POINT TO INTEGERS (32-bit memory address). So tellg() and tellp() returns an integer (32-bit address) to tell current position (in the file)
  - Use tellg and tellp as the position parameter inside of the seekg and seekp functions to change the position of the g and p pointers relative to start of file (unless you use ios flags to specify relative location to offset from)

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



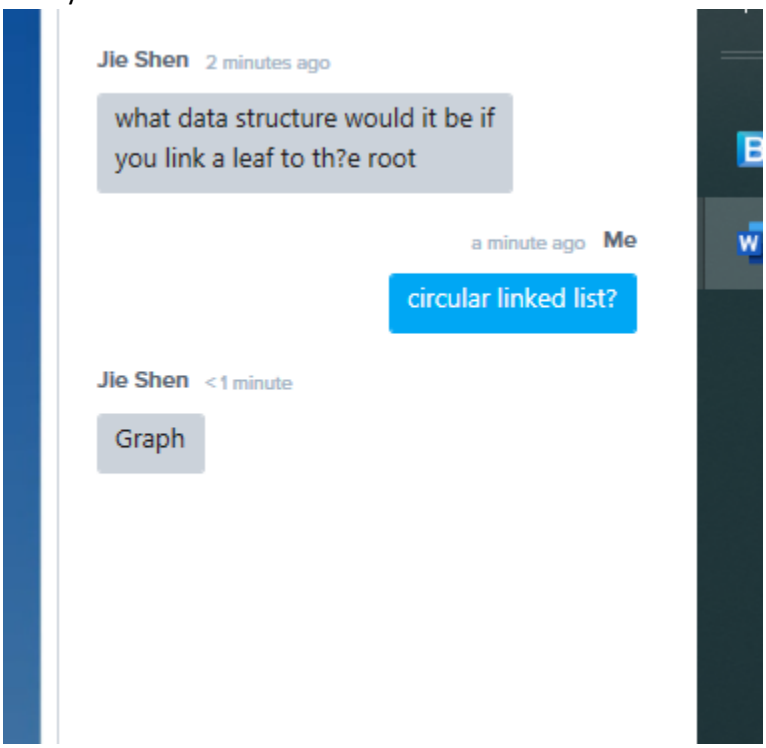
-
- Myfile << means that you are writing output to the file and thus changing the put pointer position



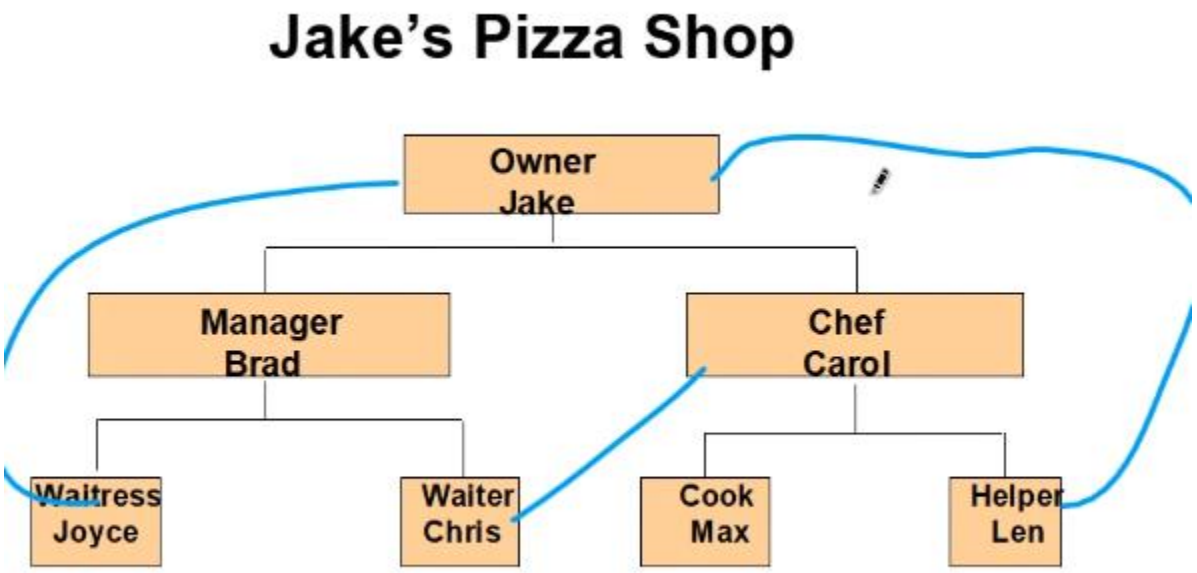
-
  - In the above example, opening a file and setting g to the end of the file (or simply opening the file ios::ate → at the end, and then doing size = tellg.() is **a trick for checking the size of the file**.
  - Then you notice memblock = new char[size]; so you know exactly the size of memory to allocate in the program efficiently to grab the data from the file and storing the data in the memblock array using file.read(memblock, size)
- Remember you can off set in the + direction and – direction
  - For example, ios::beg you can only go + direction
  - For ios::end, you can only offset in – direction
  - In between you can offset in + or - direction

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

4-05-2021 notes:

- Binary search tree
- 

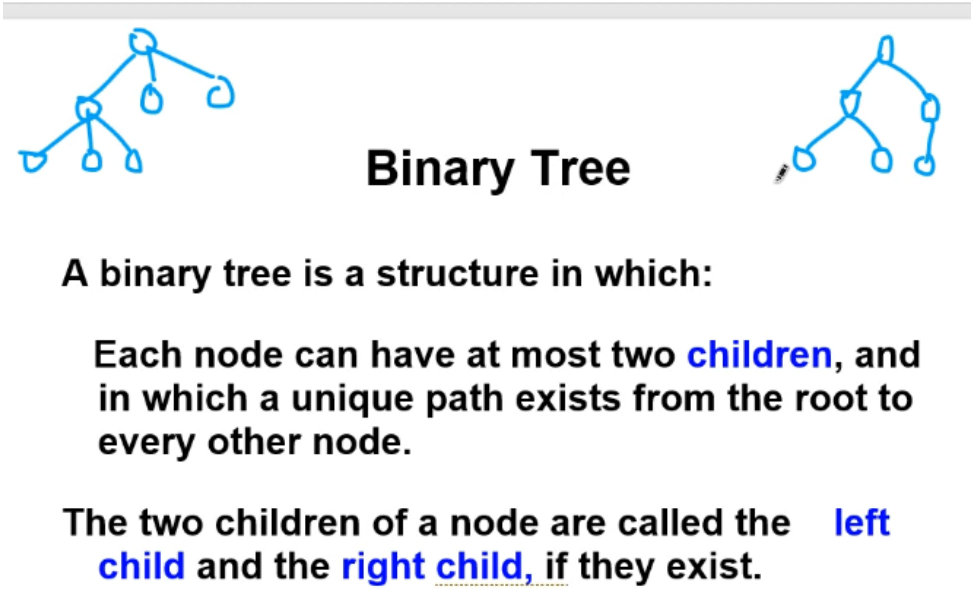
  - Answer is Graph: a tree is a subset of Graphs
  - All trees are graphs, but not all graphs are trees



  -

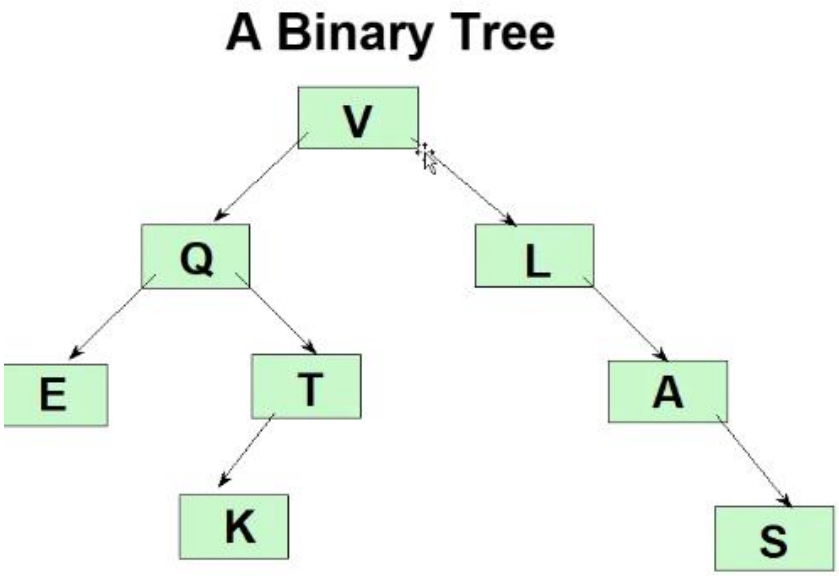
- Above, since one of the leaves is connected to the root node, we call it a **Graph**, not a Tree

- Root node begins level 0
- Remember a linked list is NOT a tree; each node in a linked list is only connected to 2 other nodes (one before, and one after the current node)
  - Binary Trees have one or 2 (2 at most) nodes that branch from the current node

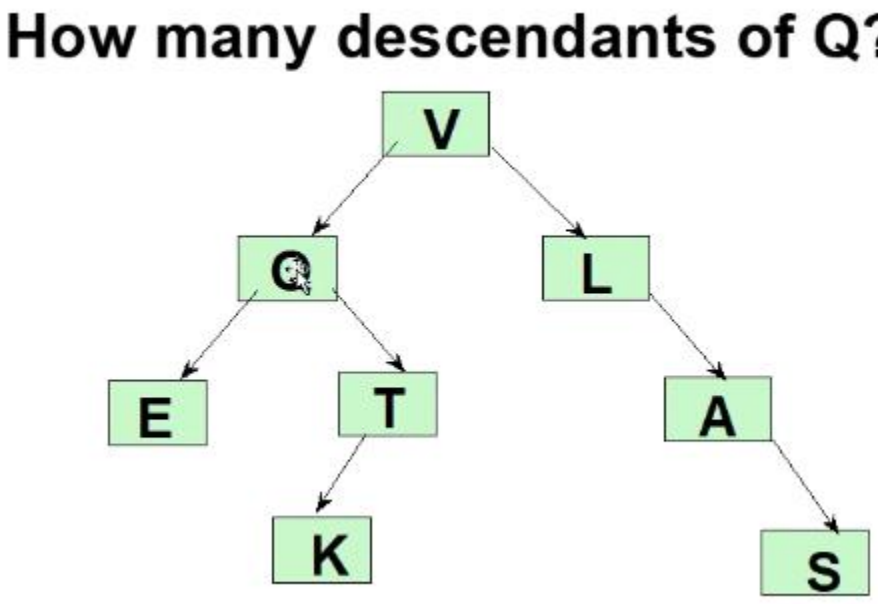


  -
  - Above, the left drawing is just a general tree
  - The right tree is an example of a binary tree
- Binary search tree must be a sorted just like a linked list
- There are right and left subtrees with respect to a reference node from which the subtrees come out of
- Binary tree exampel:

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

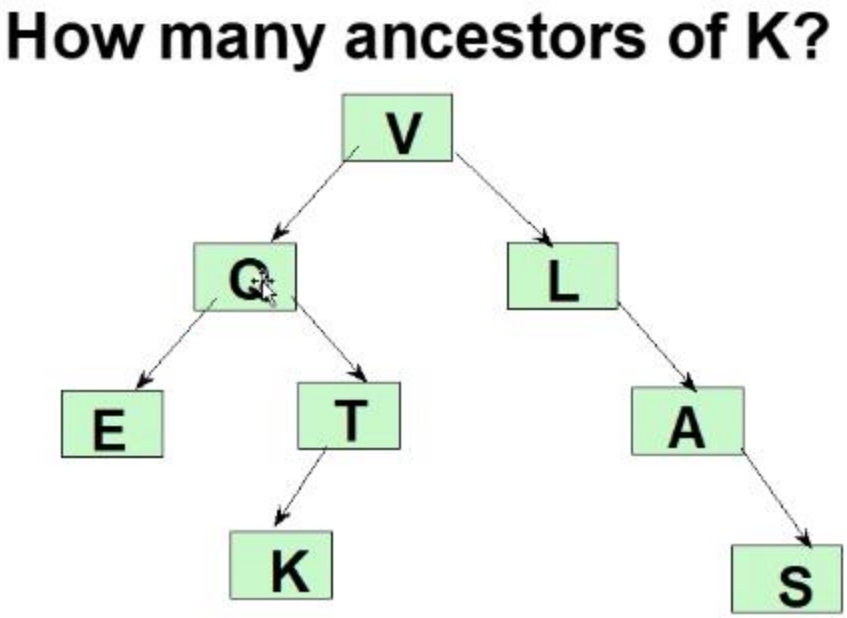


- Above, a binary tree, with THREE leaves (E K and S), root node V, and 6 parent nodes (including root node); leaves are called leaves because they have 0 children; parents are parents because they are intermediate nodes – they have at least 1 child

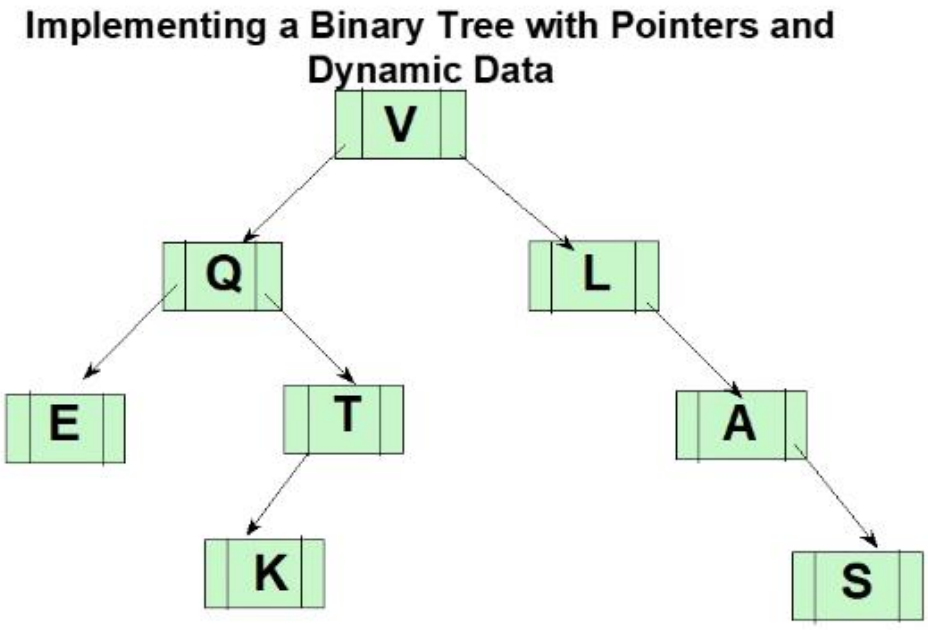


- E, T, and K are descendants of Q

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



- Ancestors of K are T, Q, and V (you travel **back up** and trace to the root to count number of ancestors)
  - Process called a back up

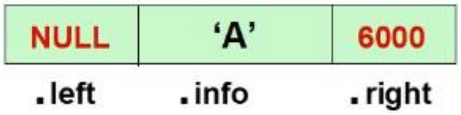


- Note: You can just use a struct for the implementation of the nodes

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Each node contains two pointers

```
template< class  ItemType >
struct   TreeNode
{
   ItemType     info;                 // Data member
   TreeNode<ItemType>* left;          // Pointer to left child
   TreeNode<ItemType>* right;         // Pointer to right child
};
```



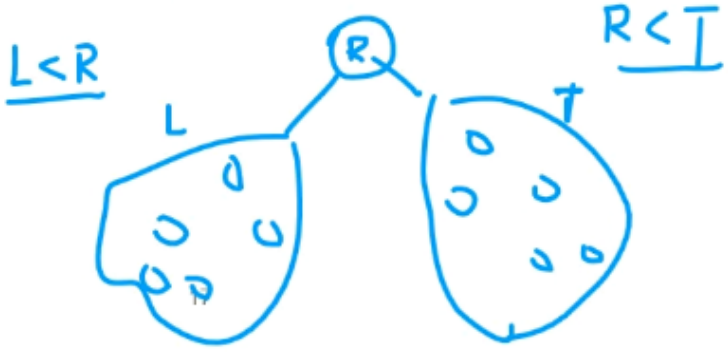
-
- NULL means there is no child for the left or right node
- Notice how the struct differs from a Linked List struct used to point through a standard list data structure

- Binary Search tree

A Binary Search Tree (BST) is . . .

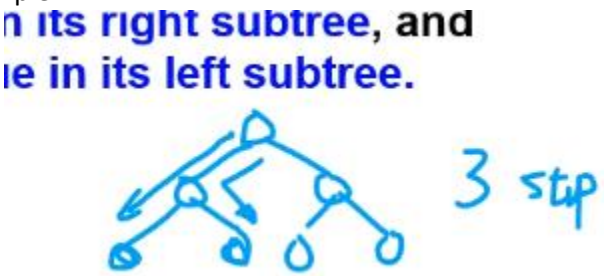
A special kind of binary tree in which:

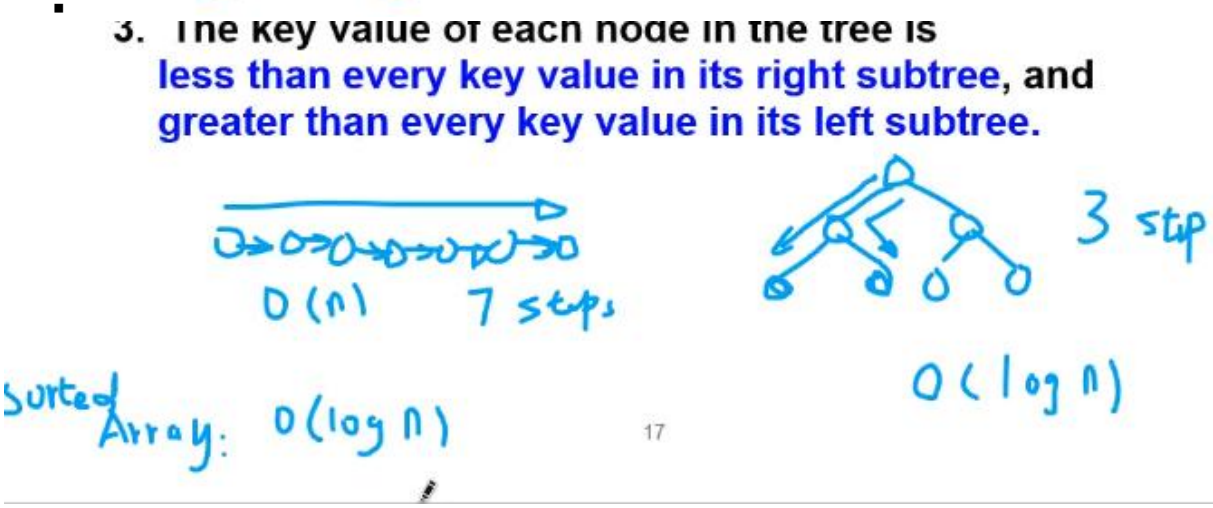
1. **Each node contains a distinct data value,**
2. **The key values in the tree can be compared using "greater than" and "less than", and**
3. **The key value of each node in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.**



  -

- Binary search tree are very fast in retrieving a node after you build the tree (especially if tree will remain the same/stay fixed)
- The total number of search steps is at **most** equal to the number of LEVELS in the tree; s for example:



- 

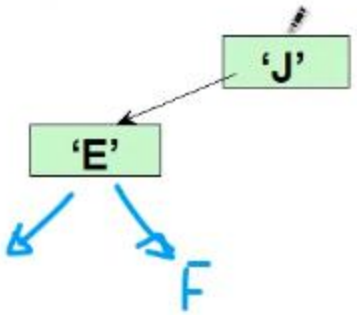
- Notice above, you have a 7-node list and a 7-node tree
  - For the linked list binary search tree, and for the sorted array using a binary search, you have the fastest search: binary search capability O(logN)
  - BUT: For unsorted array and for a singly linked list (sorted or unsorted), it is slowest O(N)
  - So that is why binary search tree using linked nodes is useful! It overcomes the inability of standard linked list to do binary search even if data is sorted

- Inserting values into BST

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Inserting ‘E’ into the BST

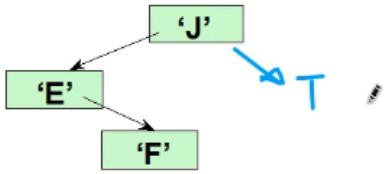
Thereafter, each value to be inserted begins by comparing itself to the value in the root node, moving left it is less, or moving right if it is greater. This continues at each level until it can be inserted as a new leaf.



-
  - Always compare with ROOT NODE when inserting a value and work from there

Inserting ‘F’ into the BST

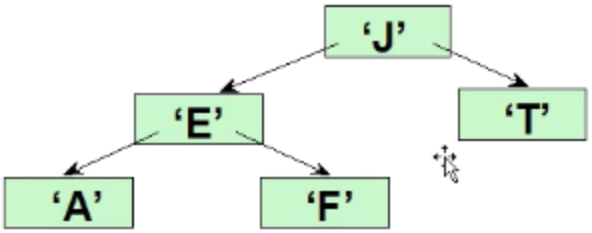
Begin by comparing ‘F’ to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



  -
    - Remember left node must always be LESS THAN its parent, right node must always be GREATER THAN its parent

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Begin by comparing 'A' to the value in the
moving left it is less, or moving right if it
This continues until it can be inserted as



-
- Input sequence is VERY IMPORTANT
- The above tree was constructed using this input order: J, E, F, T, A
- For example, if order was:

What binary search tree . . .

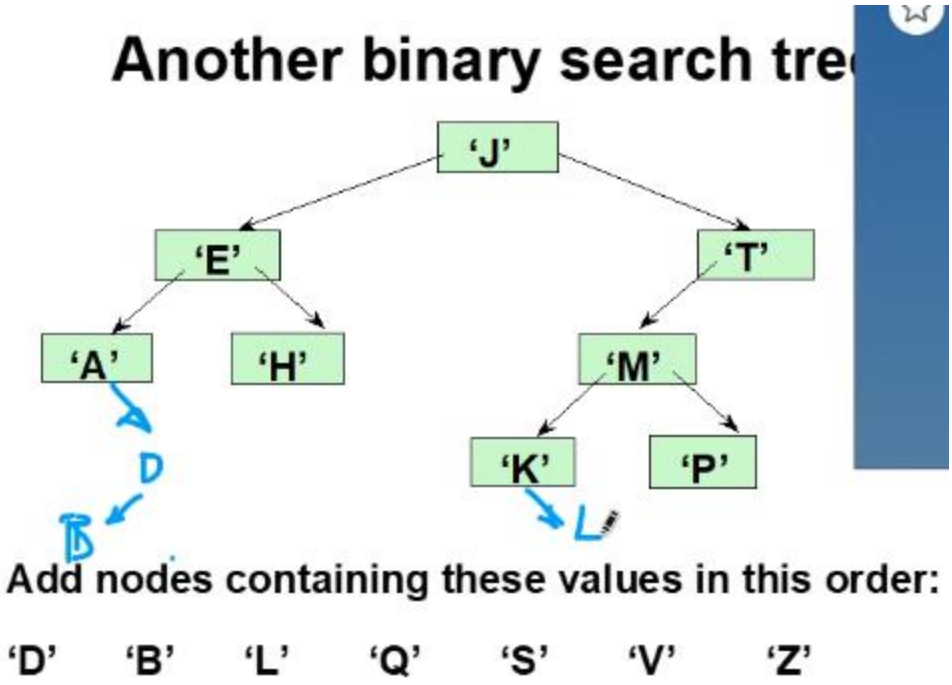
is obtained by inserting
the elements 'A' 'E' 'F' 'J' 'T' in that order?

- This is the tree you get:

-
- Above, it is bad when a built tree ends up becoming a list / incomplete binary search tree; thus the binary search would not save time
- It is optimal search when you have a COMPLETE binary search tree

- Another example of inserting:
  -
  -

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



- For the tree you can use class
- For the node you can use struct

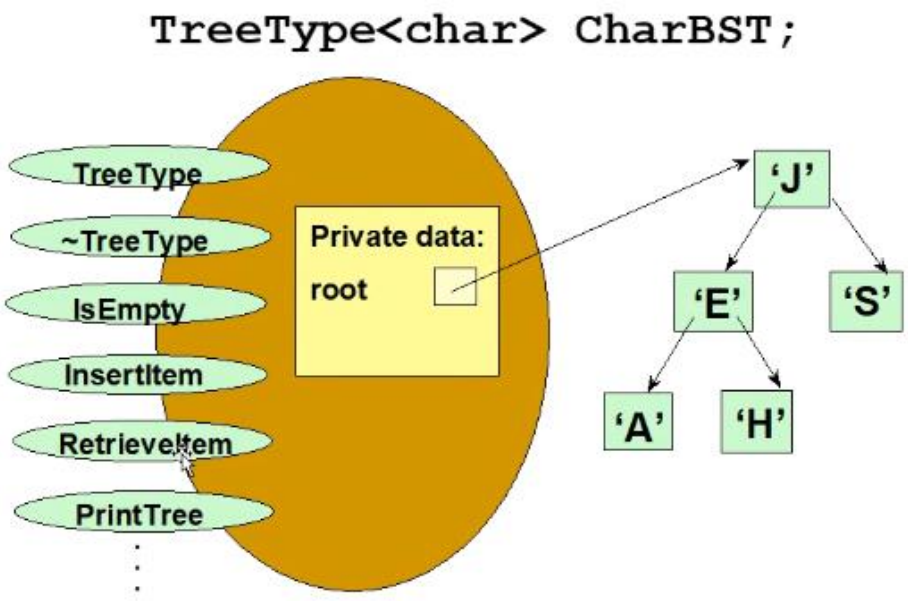
```
// BINARY SEARCH TREE SPECIFICATION

template< class  ItemType >
class   TreeType
{
public:
   TreeType ( ) ;                    // constructor
   ~TreeType ( ) ;                         // destructor
   bool   IsEmpty ( ) const ;
   bool   IsFull ( ) const ;
   int    NumberOfNodes ( ) const ;
   void   InsertItem ( ItemType  item ) ;
   void   DeleteItem (ItemType  item ) ;
   void   RetrieveItem ( ItemType&  item ,  bool&  found ) ;
   void   PrintTree (ofstream&  outFile)  const ;
    . . .

private:

   TreeNode<ItemType>*  root ;
};
```

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



-
  - Remember, the beginning memory address (ROOT NODE) must always be known and thus stored in some pointer variable
  - When tree is empty, you can set ROOT NODE = NULL
- For many of the function of the BST, you use recursive search implementations
- For insert and for finding elements in a BST, you essentially implement a binary search of the tree beginning at ROOT NODE and using a recursive implementation
- Remember to do pass by reference for even pointer variables when necessary:

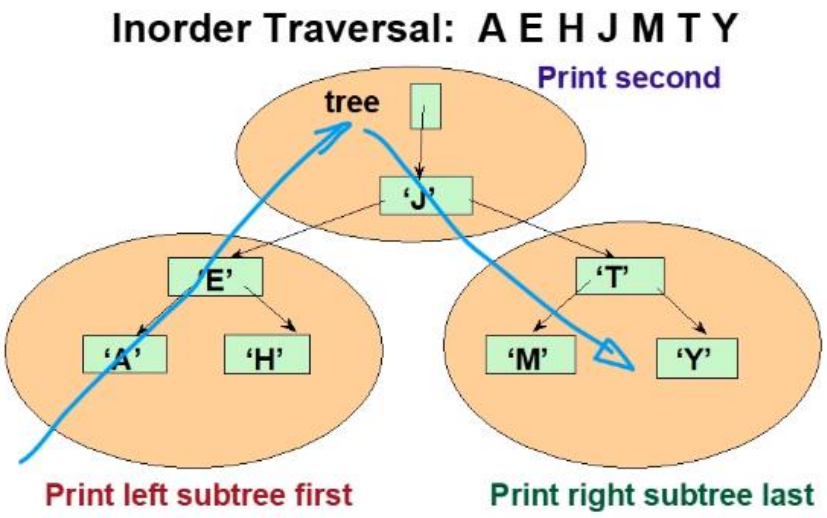
```
template< class  ItemType >
void   TreeType<ItemType> :: InsertItem ( ItemType  item )
{
       InsertHelper ( root, item ) ;
}
```

```
template< class  ItemType >
void   InsertHelper ( TreeNode<ItemType>*&  ptr,   ItemType  item )
{     if  ( ptr == NULL )
      {                              // INSERT item HERE AS LEAF
        ptr = new TreeNode<ItemType> ;
        ptr->right = NULL ;
        ptr->left = NULL ;
        ptr->info = item ;
      }
      else if  ( item  <  ptr->info )                    // GO LEFT
        InsertHelper( ptr->left , item ) ;
      else if  ( item  >  ptr->info )                  // GO RIGHT
        InsertHelper( ptr->right , item ) ;
}
```

  - For example,
  - Remember when inserting a new node, set its leaves (which are now created) to NULL

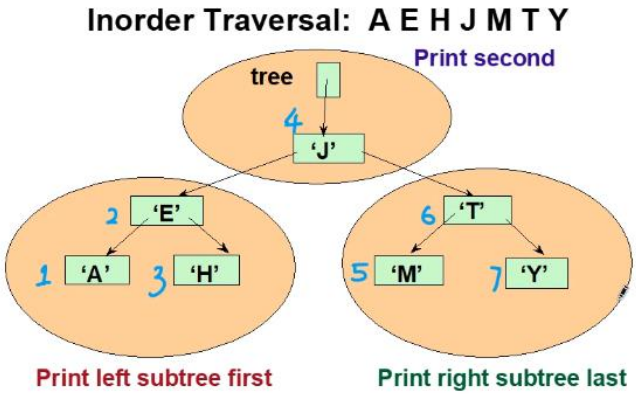
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Ways to traverse a BST tree
    - Inorder traversal



-
- Above, start rom left sub tree and move up and through upper and right trees

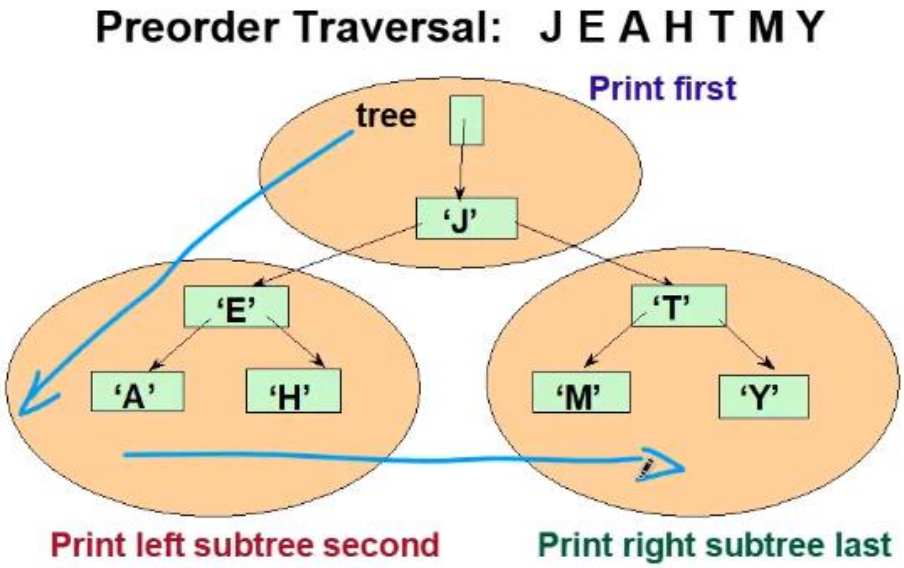
-
    - Above, the syntax means traverse Inorder, applying that formula from left most sub trees / leaf, and using that formula for any sub trees inside of subtrees



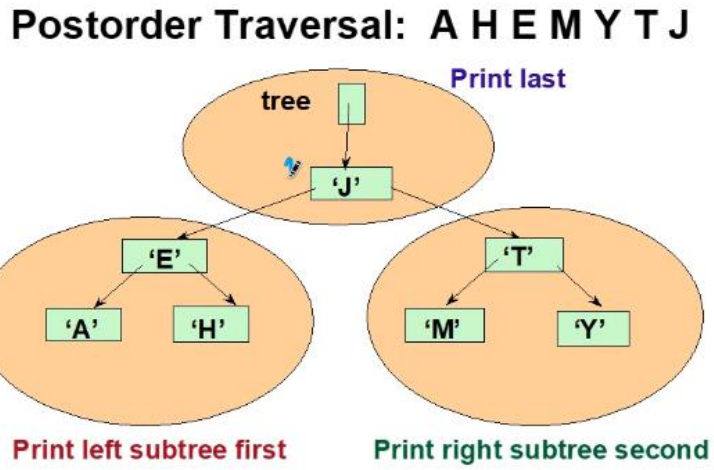
-

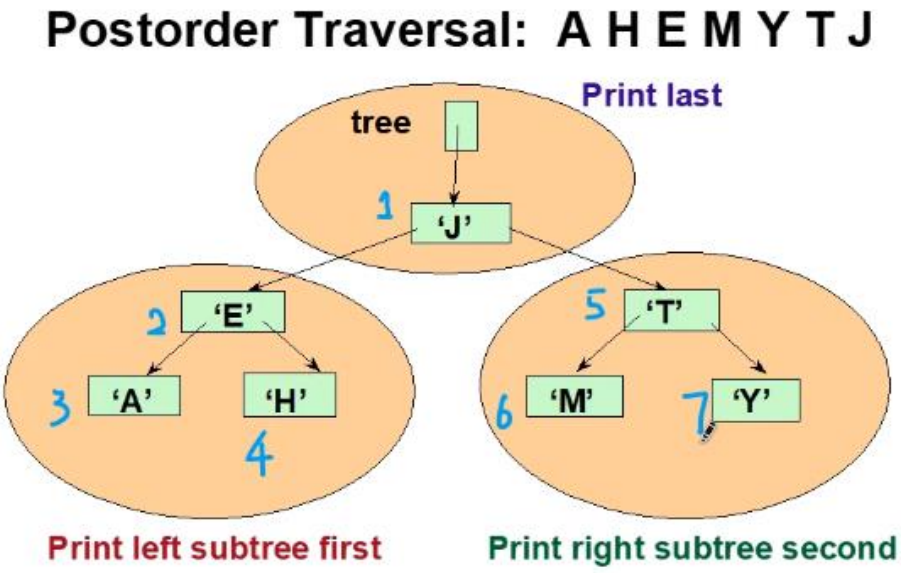
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Preorder Traversal



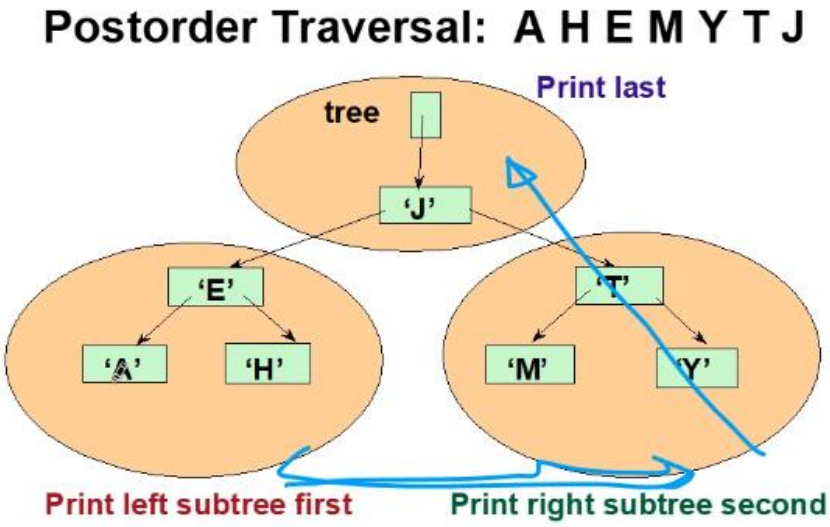
- Reference node is printed first, the its left node is printed, then lastly its right node is printed



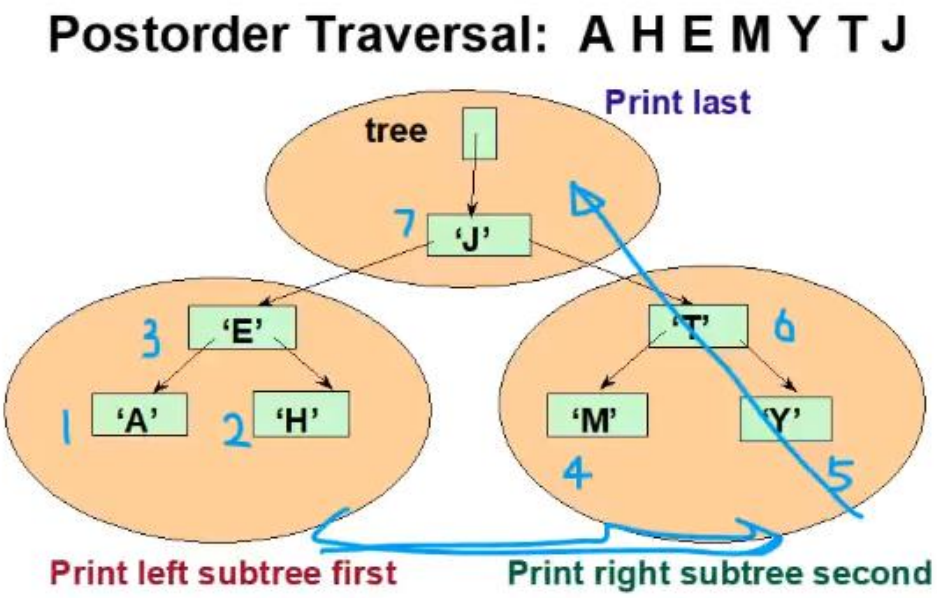


- Above shows the order of PREORDER traversal (it also shows postorder traversal) (but I can ignore the postorder for now, above the professor showed preorder traversal)

- Postorder traversal



CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

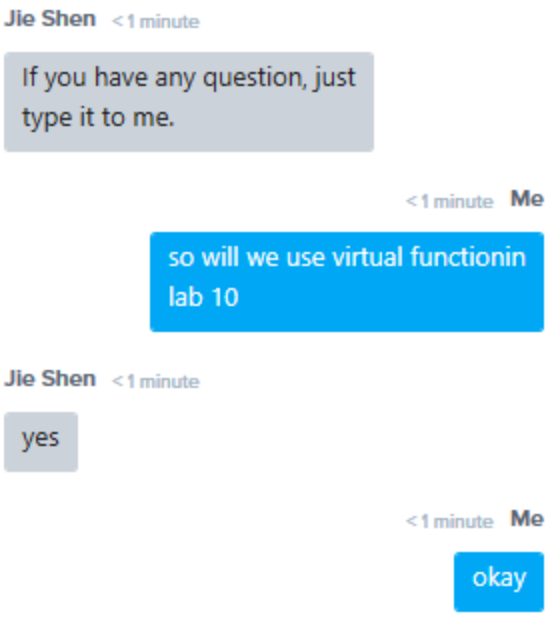


- Notice the order again for POSTORDER and the vector graph showing the formula for traversing a tree using POSTORDER traversal

LAB 10 NOTES:

- We will use virtual function in question 1

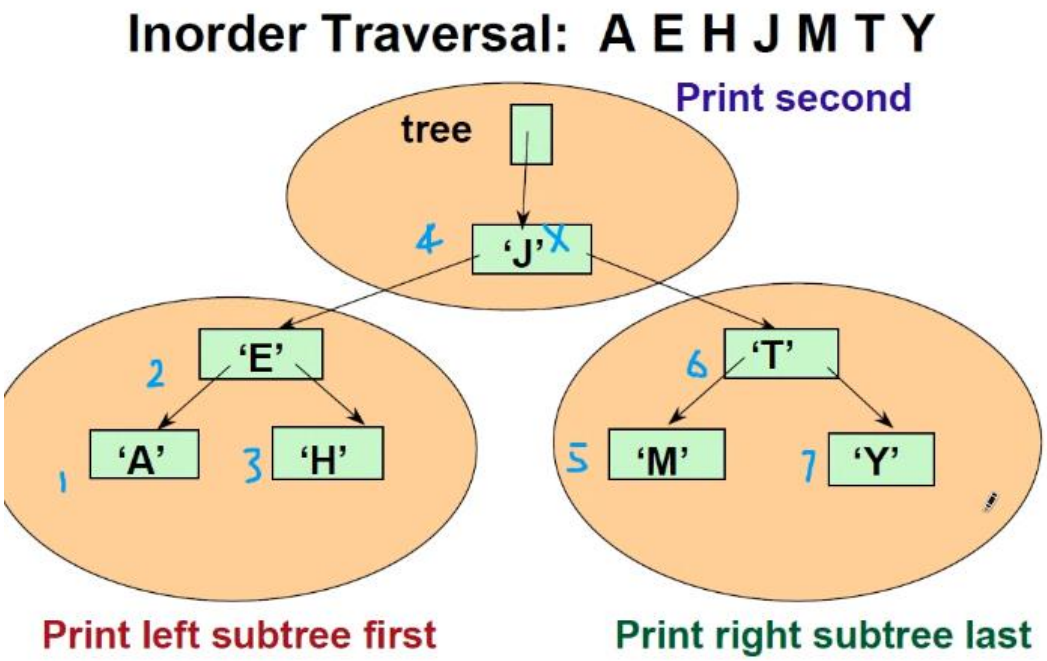
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



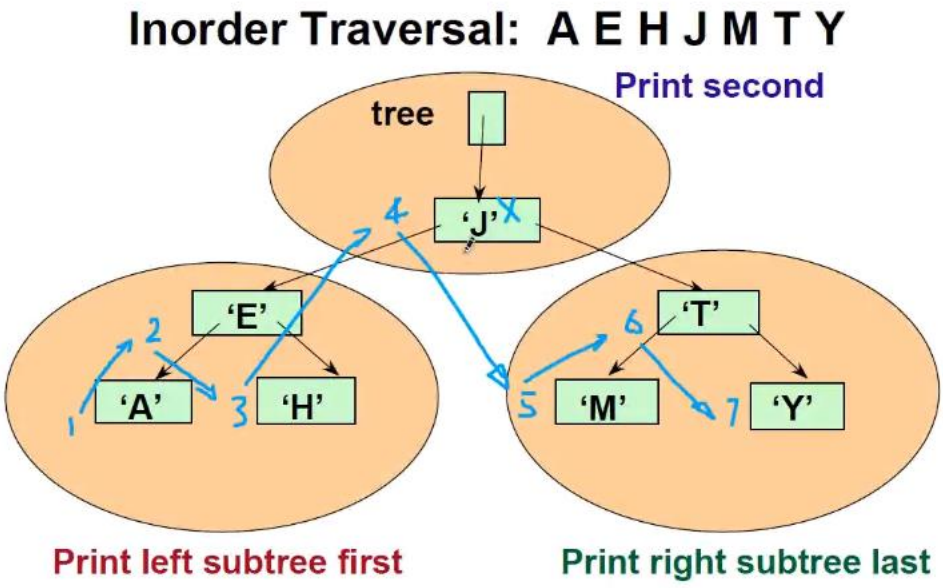
-
- Remember: write the virtual function in the base class
- Probably do QUESTION 2 FIRST then do question 1: CRC front side of card only…

4-07-2021 notes:

- Tree review:
  - Root is at top
  - Other nodes are “intermediary nodes”
  - Nodes without leaves are considered leaves
  - If leaves or nodes can move back up the tree or to nodes other than it’s own leaves, then it is not a Tree, but a more advanced version, called a GRAPH
  - Binary Search Tree (BST) is a special type of tree where all left nodes/leaves are less than the right nodes/leaves
    - For empty BST, always place first inserted element as the ROOT and then compare using the ROOT as reference for inserting subsequent nodes
  - 3 ways to traverse a tree
    - Preorder
    - Postorder
    - Inorder
  - If you delete the root node for a non BST, then just take a leaf, copy it to node, and delete the leaf that was moved in place of old ROOT NODE as the new ROOT NODE
  - For sorted list (BST) if we delete the ROOT NODE, then shift all elements to the LEFT by one position and then delete the node that was left empty after the shift
    - Use Inorder traversal

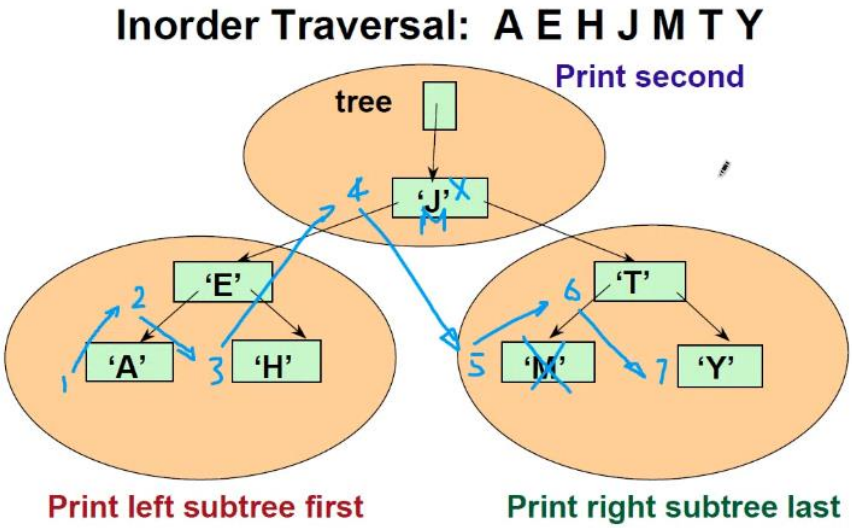


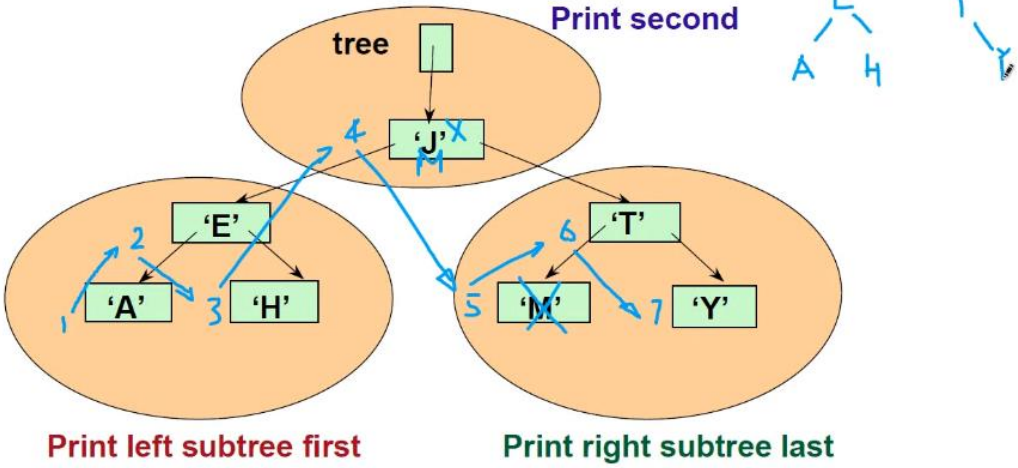
-
- Then you consider it as ONE list, the Inorder traversal list



-
- Then, there are two choices for deletion of one of the nodes:

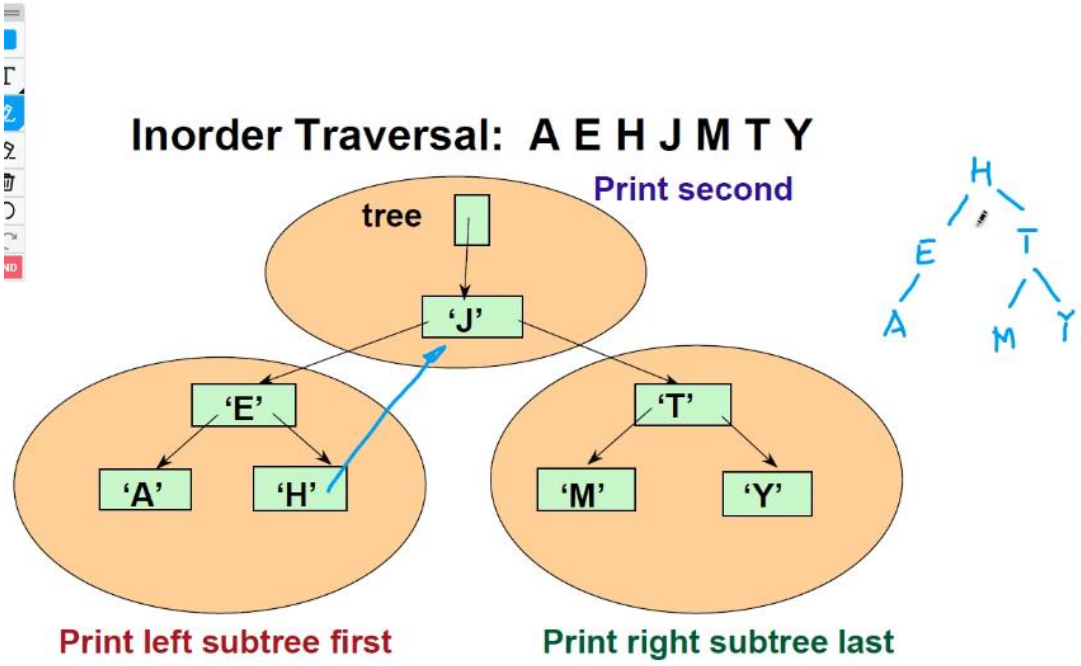
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:





- Above, notice how after deleting M and moving it in place of J so as to remove J, then it still is a BST
- The second way you could achieve the same thing is by moving a smaller leaf from left subtree :

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

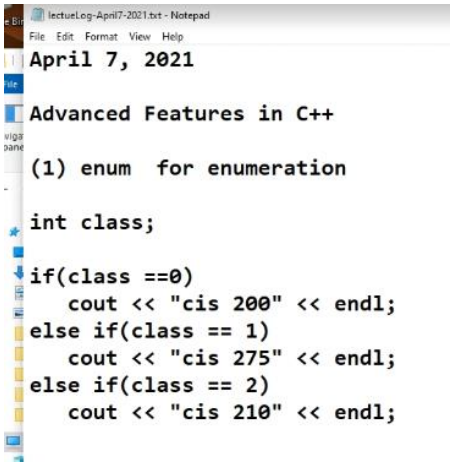


-
- Notice above, the ROOT NODE J is still removed, and the tree is still a BST
- Also notice how H is to the right of E meaning it is greater, thus it is appropriate to delete H and overwrite it to the Root and replace J, and this will still maintain BST status

  - WILL STUDY MORE THINGS ABOUT THE TREE AND GRAPHS IN CIS 350

Advanced features in C++

- Enum for enumeration:
  - Using numbers to compare values can make it easy to forget after some time away from your code what the numbers mean



  - So it is better to use strings to represent our choice; use keyword **enum**
  - Example:

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

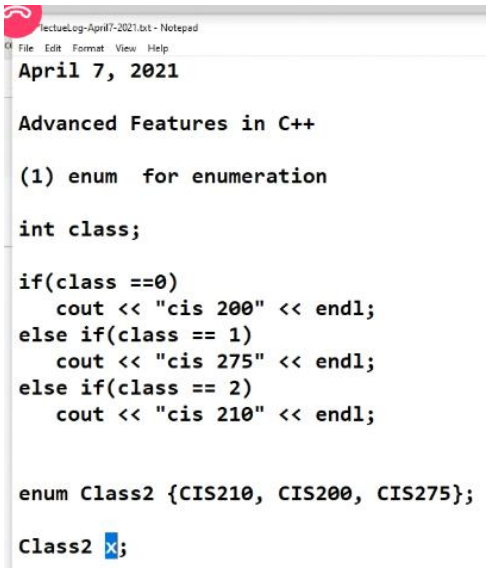
Example 1:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // Defining enum Gender
    enum Gender { Male, Female };

    // Creating Gender type variable
    Gender gender = Male;

    switch (gender)
    {
    case Male:
        cout << "Gender is Male";
        break;
```

- 

- Above, x can only be one of the three values from the enum declaration of Class2
- Using enumeration is a better solution, here is the rest of the example:

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
enum Class2 {CIS210, CIS200, CIS275};

Class2 x;

x = CIS200;

switch(x)
{
    case CIS210:   // case 0:
        ...
    case CIS200:   // case 1:
        ...
    case CIS275:   // case 2:
        ...
}
```

- Notice, case 0 1 and 2 are the old non-enumerated versions, which is not preferred; the numerical values are simply replaced by their respective word; its basically like #defining a value so you can use a word instead of a number so it is easier to keep track of meanings and associations of values

- Another example:

4/7/2021

```
#include <bits/stdc++.h>
using namespace std;

// Defining enum Year
enum year {
    Jan,
    Feb,
    Mar,
    Apr,
    May,
    Jun,
    Jul,
    Aug,
    Sep,
    Oct,
    Nov,
    Dec
};

// Driver Code
int main()
{
    int i;
```

- Internally above, each year maps to a number:

4/7/2021

```
#include <bits/stdc++.h>
using namespace std;

// Defining enum Year
enum year {
    Jan, → 0
    Feb, → 1
    Mar,
    Apr,
    May,
    Jun,
    Jul,
    Aug,
    Sep,
    Oct,
    Nov,
    Dec → 11
};

// Driver Code
int main()
```

-
- Here is the full example:

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Example 2:

CPP

```cpp
#include <bits/stdc++.h>
using namespace std;

// Defining enum Year
enum year {
    Jan,
    Feb,
    Mar,
    Apr,
    May,
    Jun,
    Jul,
    Aug,
    Sep,
    Oct,
    Nov,
    Dec
};

// Driver Code
int main()
{
    int i;

    // Traversing the year enum
    for (i = Jan; i <= Dec; i++)
        cout << i << " ";

    return 0;
}
```

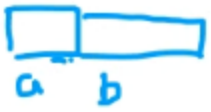
Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

- Unions → use keyword **union**

- (2) union

```
struct X
{
    int a;
    double b;
};

X v1;
v1.a = 10; v1.b=20.1;

union Y
{
    int a;
    double b;
};
```



  - Notice above, there is no overlap for a struct for the data members, but for unions there is overlap for starting address of the data members of the data structure
  - Main purpose?: it is used for when memory is limited, so for one part of the code you might use an integer, but the other part of the code might use a double; thus for when using integer you only need the first few bytes, but when using the address location for a double you will use a larger set of bytes from that same starting memory location
  - It allows you to "combine" data types using the same start memory address
  - Sizes of union of the same class of a structure will be a smaller byte size because of overlap of data members

```
// declaraing union

union union_example
{
    int integer;
    float decimal;
    char name[20];
};

void main()
{
    // creating variable for structure
    // and initializing values difference
    // six
    struct struct_example s={18,38,"geeksforgeeks"};
```

https://www.geeksforgeeks.org/difference-structure-union-c/

- Nowadays, unions are obsolete because usually we do not have as many memory limitations

- Auto → use keyword **auto**
  - **The auto data type is inferenced from the context**
  - Consider auto a data type
  - For example:

```
auto d{ 5.0 }; // 5.0 is a double literal, so d will be type double
auto i{ 1 + 2 }; // 1 + 2 evaluates to an int, so i will be type int
```

This also works with the return values from functions:

```
int add(int x, int y)
{
    return x + y;
}

int main
{
    auto
    retu
}
```

While using `auto` in place of fundamental data types only saves a few (if any) keystrokes, in future lessons we will see examples where the types get complex and lengthy. In those cases, using `auto` can save a lot of typing.

Type inference for functions

In C++14, the `auto` keyword was extended to be able to deduce a function's return type from return statements in the function body. Consider the following program:

```
auto add(int x, int y)
{
    return x + y;
}
```

  - **Best practice**

    Avoid using type inference for function return types.

    Interested readers may wonder why using `auto` when initializing variables is okay, but not recommended for function return types. A good rule of thumb is that `aut` is okay to use when defining a variable, because the object the variable is inferring a type from is visible on the right side of the statement. However, with functions, that is not the case -- there's no context to help indicate what type the function returns. A user would actually have to dig into the function body itself to determine what type the function returned. It's much less intuitive, and therefore more error prone.

Trailing return type syntax

The `auto` keyword can also be used to declare functions using a **trailing return syntax**, where the return type is specified after the rest of the function prototype.

Consider the following function:

```
int add(int x, int y)
{
  return (x + y);
}
```

Using `auto`, this could be equivalently written as:

```
auto add(
{
  return
}
```

In this case, `auto`

Why would you want to use this?

One nice thing is that it makes all of your function names line up:

```
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto printSomething() -> void;
auto generateSubstring(const std::string &s, int start, int len) -> std::string;
```

For now, we recommend the continued use of the traditional function return syntax. But we'll see th... ailing return type syntax crop up again in lesson **10.15 --**

  - The feature is ambiguous, good practice is to AVOID using auto because of potential ambiguity and confusion in your code

- **volatile** keyword

- you can set variables as volatile
- we use it because often times compilers optimize your code when it compiles it, but sometimes its optimization process can cause issues and distort some variables; so to prevent compiler from optimizing your code or a particular variable, simply use volatile keyword.
  - An example of where this might be used is for GUI interfaces, where optimization of a variable by the compiler may cause issues

Although we compile code with optimization option, value of const object will change, because variable is declared as volatile that means don't do any optimization.

```c
/* Compile code with optimization option */
#include <stdio.h>

int main(void)
{
    const volatile int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;
```

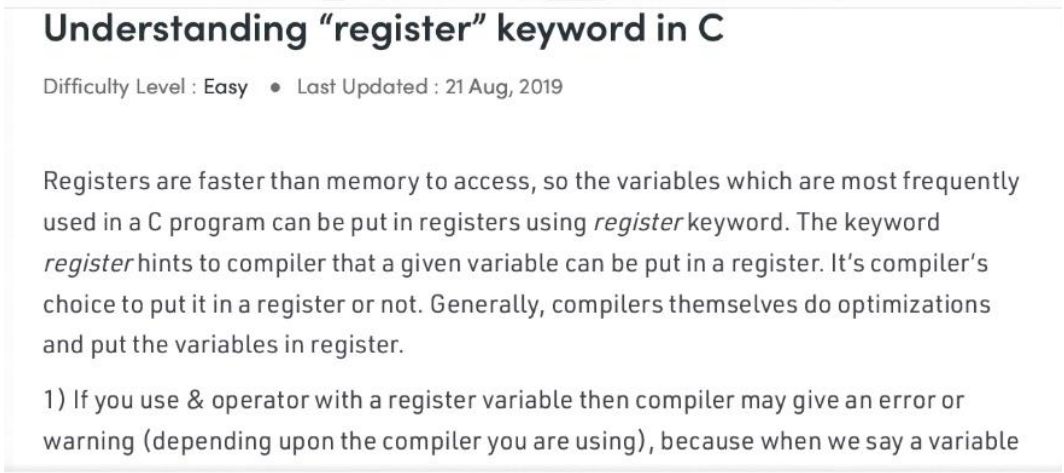
s://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/

2021 Understanding "volatile" qualifier in C | Set 2 (Examples) - GeeksforGeeks

```c
    printf("Modified value of local: %d \n", local);

    return 0;
}
```

- Keyword **register**
  - Placing this in front of a variable will cause the variable to be placed in a data register (from CPU) will simply help speed up your code; it is especially used for example when a variable will be accessed often throughout the program
  - Tells the compiler to put therelated variable into a register instead of a regular memory location (regular memory locations are not as quickly accessed; remember from MASM assembly language, and how CPU and memory works, it takes additional clock cycles to access a variable in RAM; but accessing variables in a CPU register is faster because it takes less clock cycles)

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



- Note: if you have a pointer, do not use register

```
#include<stdio.h>

int main()
{
    register int i = 10;
    int* a = &i;
    printf("%d", *a);
```

- But for this case it is okay:

```
int main()
{
    int i = 10;
    register int* a = &i;
```

  - This is okay; notice in other example I is already placed in register; but for this case I is not in register, so now you can use register with the pointer
- Cannot use other keywords with register

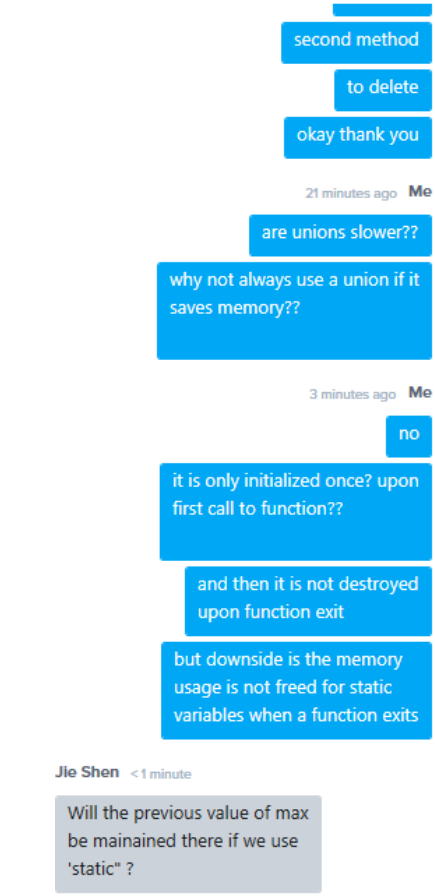
- **static** keyword

```
(6) static keyword

int getMax(int x)
{
    int max = 0;
    if(x> max)
        max = x;

    return max;
}
```

  - Notice how each time function is called, max is initialized to 0
- Now lets modify the code adding keyword static:

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

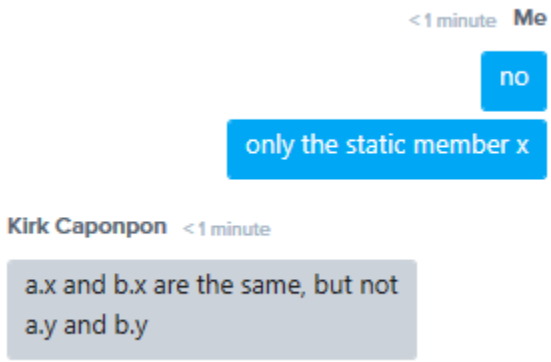
```
(6) static keyword

int getMax(int x)
{
    static int max = 0; // max becomes history-sensitive
    if(x> max)
        max = x;

    return max;
}
```

- 

  -
    - Answer to last question is also yes
    - Also unions are not slower; they are as fast as struct; but you can only use one data member at once since variables are stacked at same starting memory address: hence this is the limitation of unions; they save memory, but variables can only be accessed one at a time
- With static, now we can call the function many times and max will not keep getting reinitialized, so now we can truly find the max value every time we call the function

- But remember, the local static variables will not be freed when function exits, so static variables in function will cause less unallocated memory in your program; and also downside is you cannot reset the variable/function as easily as when you use non-static members
-

```
class Student
{
   public:
     static int x;
     int y;
     ....

};

int main()
{

    Student a, b;
    a.x     b.x
    a.y     b.y
```



-
  - Above is correct
- There is only one copy of instance x; one memory allocation for the single variable that is used by all objects of the class
- But for y, each object has its own copy of y; each y has its own memory address allocation

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
class Student
{
   public:
     static int x; // there is only one copy of x for all the
                   // objects of class Student
     int y;  // Each object of class Student has a different copy
             // variable y
     ....

};
```

- Also you can have static member functions; this means you only have 1 copy of the function for all instances of the class;
  - so it means all local variables delcared in the static function will not change; only one copy; so they will not keep getting reinitialized

```
class Student
{
   public:
     static int x; // there is only one copy of x for all the
                   // objects of class Student
     int y;  // Each object of class Student has a different copy
             // variable y

     static void foo( ) // only one copy for all the instances of
                        // class Student
     { ... }
     ....

};
```

- Inside of a static member function, you are only allowed to use LOCAL variables and other STATIC data members. You are not allowed to use non-static data members

4-12-2021 notes:

- Union
  - Struct uses at least sum of length of data members; uses contiguous blocks of memory
  - Union share beginning memory address

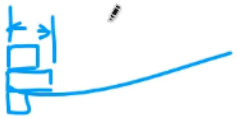
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

(2) union

A union is a special data type available in C/C++ that allows storing different data types in the same memory location. The main purpose of introducing union is to save memory in the old days. The restriction of using a union is that at any moment only one data member of the union should be used. In other words, no two or more data members should be used at any time.

Example 1: struct

```
struct X
{
  int a;
  double b;
};

X v1;
v1.a = 10; v1.b=20.1;
```



Example 2: union

- Length of union is length of longest data member; unions used originally because memory was so limited and precious

```
    int integer;
    double decimal;
    char name[20];
};

Y u={18, 38, "geeksforgeeks"};

printf("\nunion data:\n integer: %d\n "
        "decimal: %.2f\n name: %s\n",
    u.integer, u.decimal, u.name);
```

Output:

union data:

integer: 18

decimal: 0.00

name: ?

- Above, demonstrate that for unions, only 1 data member can be used at once; all 3 variables of the union cannot all be assigned a valid value at the same time;

```
Y u={18, 38, "geeksforgeeks"};  // u.integer = 18;
```

  - So notice, the above initialization of the union only actually initializes integer because it is the first one assigned a value, and not all data members of a union can be used simultaneously.
  - So if you want to use the double or the char[] array, you have to initialize them before use, and then if you want to switch to using a other member, the once again, you have to initialize it first since the initialization of another variable cancels out/voids the valid data and usage of the other members; if you need all 3 to be initialized and valid at once, then use a struct/class.

- Static keyword

- A local variable or object that is not destroyed and thus access to it is shared by all variables/objects created

```
class Student

                                                                    5

{
  public:
   static int x; // there is only one copy of x for all the
                 // objects of class Student
   int y;  // Each object of class Student has a different copy
         // variable y

   static void foo( ) // only one copy for all the instances of
                    // class Student
   {
```

- You can make a class, function, or variable STATIC.

Static data member of a class

Dear students,

Below is an example for using the static data member of a class:

```
class Student3
{
public:
static int x;
int y;
};

int Student3::x = 5;

int main()
{

Student3 w1, w2;
w1.y = 10; w2.y = 20;

cout << Student3::x << endl;
```

- Keyword: extern

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

(7) extern keyword

Case 1: extern variables

File1.cpp:

Int x;

.....

File2.cpp:

Extern int x;

.....

-
- Keyword extern means a variable or function is declared in another file; so we use extern basically like an alias; so a variable in one file is used based on declaration of a variable in another file; the variable in both files occupy the same memory address

File2.cpp:

extern int x; // it means x is declared in another file.

.....

-

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

Case 2: C functions

```
// Save file as .cpp and use C++ compiler to compile it
extern "C"
{
  int printf(const char *format,...);
}

int main()
{
  printf("GeeksforGeeks");
  return 0;
```

- Above it tells compiler to handle a C-style function declaration and convert it to C++ style upon compilation
-

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
/* Declarations of this file */
#ifdef __cplusplus
}
#endif
```

- In between should be the c function; we use the above #ifdef_cplusplus if compiling a c-style function using a cPP file (c++ compiler); if a c compiler is compiling the file then the above will be skipped of course since c compiler can compile a c-style function

- Preprocessor Directives

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

(8) Preprocessor

Preprocessors are programs that process our source code before compilation.

Case 1: macro

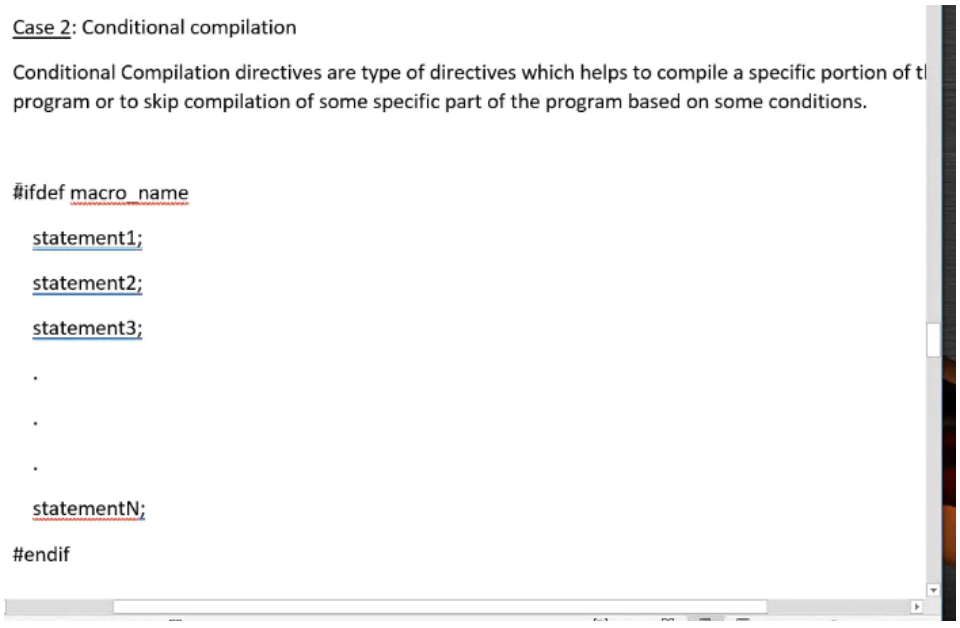
// macro with parameter

#define AREA(l, b) (l * b)

Case 2: Conditional compilation

Conditional Compilation directives are type of directives which helps to compile a specific portic program or to skip compilation of some specific part of the program based on some conditions.

- #ifdef macro_name
- Programs that process our source code before compilation
- Macro is similar to inline function



- Above, is case 2 of a preprocessor directive; with conditions included

Case 2: Conditional compilation

Conditional Compilation directives are type of directives which helps to compile a specific portion of t
program or to skip compilation of some specific part of the program based on some conditions.

```
#ifdef macro_name
   statement1;
   statement2;
   statement3;
   .
   .
   .
   statementN;
#endif
#define macro_name
#undef macro_name
```

- Now above, notice udef is used → "undefine" directive
- Some portions are for cpu, others for gpu, that is why we use a conditional directive so that it can handle multiple platforms/compilers
- It helps decide which portions of a program needs to be compiled or skipped
- Ifdef and ifndef are conditional directives

- Case 3: Once-only Header File

```
/* File foo.h  */
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

the entire file

#endif /* !FILE_FOO_SEEN */

File1.cpp:
```

- Case 3: ifndef, define, endif; this conditional set of statements is so that a file will not keep redundantly getting redefined if a file was already called upon and included/compiled; if file already included, then we skip (these preprocessor directives are assembly-language level programming directives)

File1.cpp:

#inclμde "foo.h"

...

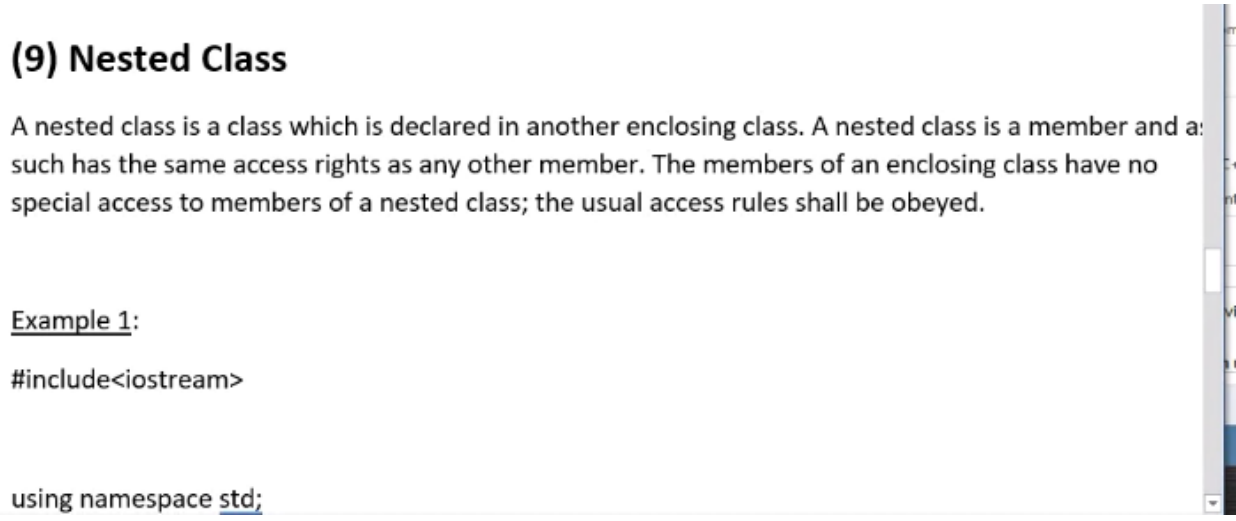
File2.cpp:

#include "foo.h"

...

- Notice above, both cpp files #inlcude the same header file; the ifndef, define, endif conditional directives will ensure the header file is included only ONCE; it executed if not yet included, upon being included next time it will skip (so file1.cpp will include it, but file2.cpp will #include, but then it will be skipped)
- Also for ifndef and define you can make it whatever name you want, but it is usually good practice to make the definition similar to the name of the file

- Nested Class



(9) Nested Class

A nested class is a class which is declared in another enclosing class. A nested class is a member and a such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed.

Example 1:

#include<iostream>

using namespace std;

- Inner class can access everything of outer class
- But outer class cannot access everything of inner class
  - This is similar idea to base and derived classes
  - Base classes cannot access members of a derived class

```
class Enclosing {     // outer class
  private:
    int x;

  /* start of Nested class declaration */
    class Nested {   // inner class
    int y;
    void NestedFun(Enclosing *e) {
     cout<<e->x;  // works fine: nested class can access
                  // private members of Enclosing class
    }
  }; // declaration Nested class ends here
}; // declaration Enclosing class ends here
```

-

```
class Enclosing {     // outer class

  int x;
```



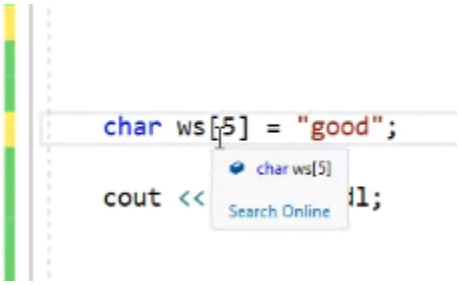
```
  /* start of Nested class declaration */
  class Nested {  // inner class
    int y;
  }; // declaration Nested class ends here

  void EnclosingFun(Nested *n) {   // a member function of outer class
     cout<<n->y;  // Compiler Error: y is private in Nested
```

-
- Above notice how an error occurs because outer class attempt to access inner class
- Nested class is like a parasite relationship; you can use base and derived classes instead, but c++ provided this nested class feature nonetheless

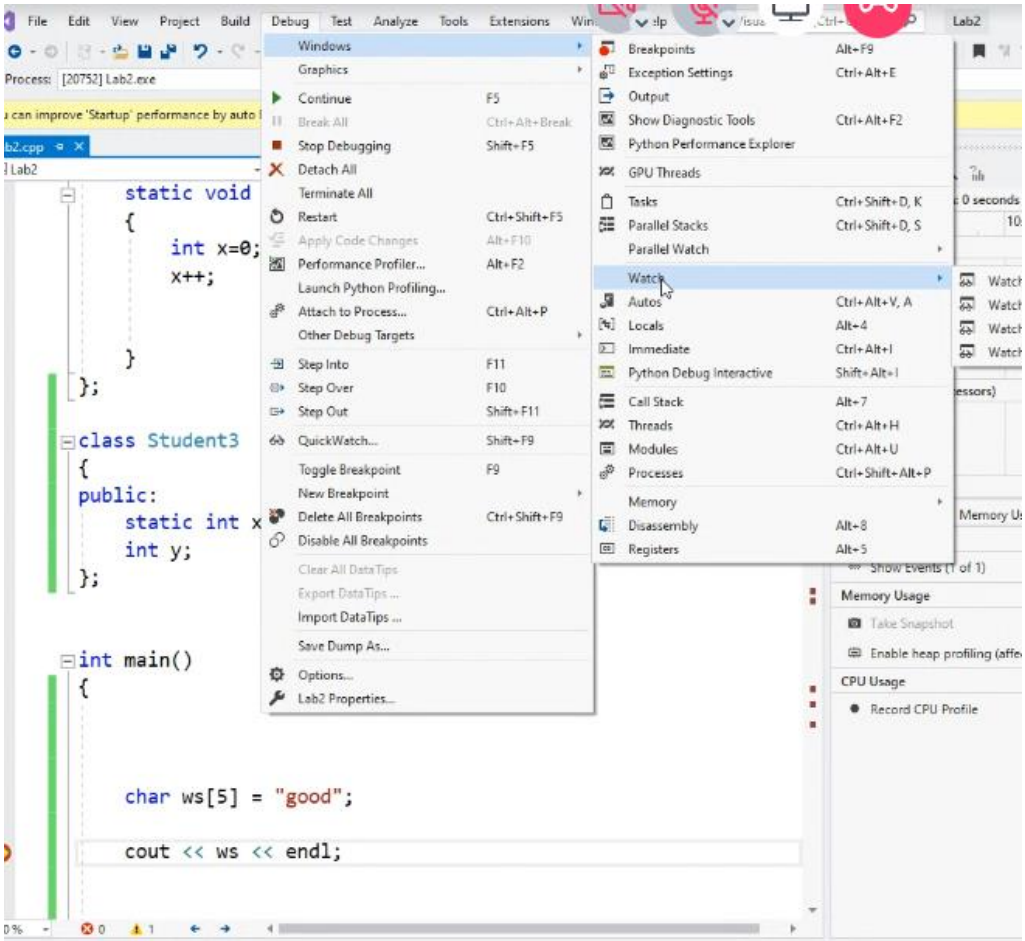
- Char arrays
  - Must be a trailing null

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
char ws[4] = "good";

cout << ws << endl;
```

-
  - Above statement is bad; need size of at least 5; and remember you can only use a string literal (which are const char arrays) to assign to a char array only at the initial declaration
  - Correct would be: char ws[5] = “good” → implicitly c++ will do this: “goodNULLCHAR”
  - 

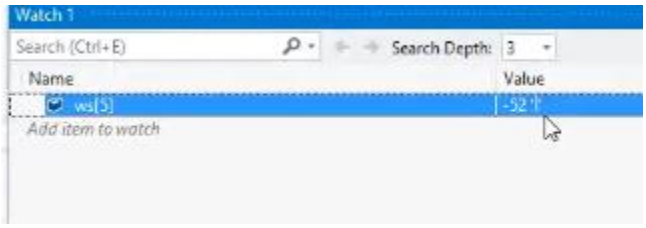
- Remember to inspect variables, use watch window:



- Above, how to open watch window

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

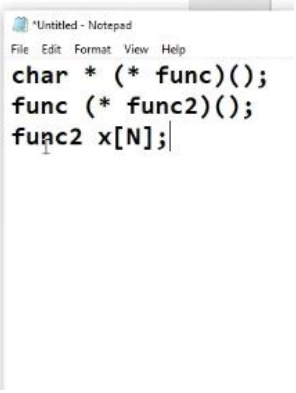
- You can also change each value of elements in an array to see the value of each element:



- Above, change the 5 inside of [] to inspect/watch the value of any data member and even each element of a data member if it is an array; so ws[4] will display under "value" the char 'd' and its numerical representation

- Question 4 from interview questions

1. How do you decide which integer type to use?
2. What does extern mean in a function declaration?
3. What's the auto keyword good for?
4. Define a linked list node which contains a pointer to itself.



  - Above is answer to question 4

LAB 11 NOTES:

- Use double pointer to construct the tree
- In a way similar to constructing a linked list, you use a double-pointer node to construct a tree.
- okay, so based on the BST, you will know that we have constructed it properly based on the output of our inorder print
  - answer: yes
-

4-14-2021 notes (final class; final exam review session/interview questions); exam is open book, open note; there is no UNIX:

- Question 18

```
int main()
{

    int i;
    char a[] = "String";
    char* p = "New Sring";
    char* Temp;
    Temp = a;
    a = malloc(strlen(p) + 1);
    strcpy(a, p); //Line number:9//
    p = (char *) malloc(strlen(Temp) + 1);
    strcpy(p, Temp);
    printf("(%s, %s)", a, p);
    free(p);
    free(a);
```

-

- Above, from interview question 18; using static cast on void* to cast it as a char*

- Question 19 – enum

```
int main()
{
    enum number { a = -1, b = 1, c, d, e };
```

- Above, c, d, and e, since they are not specified, they automatically continue numbering of the last assigned value; so above, since b = 1, then c = 2, d =3, e =4 …etc

- Use UNSIGNED CHAR to represents color (pixels In a picture or on a screen) because it will use only 1 byte
- Any value past 2 billion, consider long long integers (double long)
- Typefine you can give another alias to classes or numerical values…etc
- Understand wrap-around with matrix; use **modulo** operator
- Understand prefix and postfix ++ and –operators
- Understand inline and macro functions

A **macro** is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. (C style)

Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. (C++ style)

```
#define getmax(a,b) a>b?a:b


inline int getmax(int a, int b)
{
  return (a>b?a:b);
}
```

- Understand bitwise operators
- Know numbering systems with different bases (octal, binary, decimal, hexadecimal)

|	OR	Logic OR
^	XOR	Logic exclusive OR
~	NOT	Complement to one
<<	SHL	Shift Left
>>	SHR	Shift Right

Numbering System: binary (0, 1), octal (0, 1,…,7), decimal (0,…, 9), hexadecimal(0,…,9,A,B,C,D,E,F)

Example: 12 (octal) → decimal: ? 10

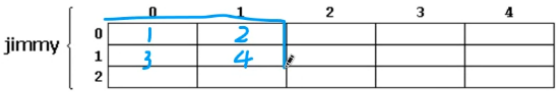
$n_2n_1n_0 = n_2 x8^2+n_1x8^1+n_0x8^0$ (octal base = 8)

$n_2n_1n_0 = n_2 x2^2+n_1x2^1+n_0x2^0$ (binary base = 2)

$n_2n_1n_0 = n_2 x16^2+n_1x16^1+n_0x16^0$ (hexidecimal base = 16)

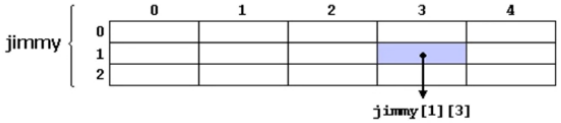
CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

- Remember how to convert from one base to another base
- Know pass by value v. pass by reference
- Know how to initialize arrays
- Know 2D and 3D arrays and how to declare
    - Remember it goes: ROW, COL, and that is how you also map a 2D array to a matrix (row-major mapping)
    - Know how to initialize a 2D array; keeping in mind row-major mapping
        - Multidimentional Arrays
            - Definition: multidimensional arrays can be described as arrays of arrays
            - Example of a 2-d array




```
int jimmy [3][5];
```

jimmy[1][3]



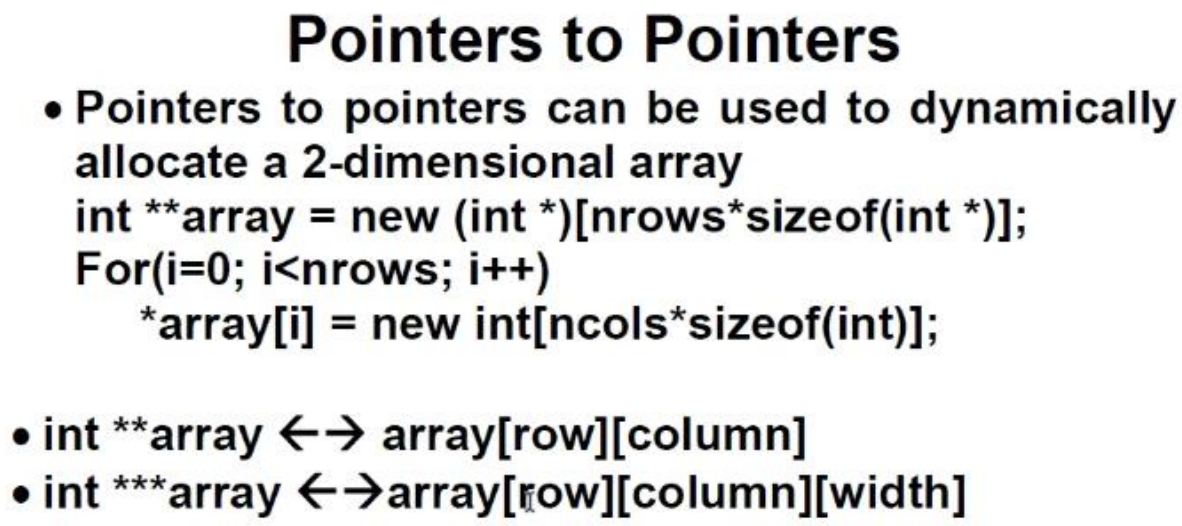
    - Above, a if a 2Darray was initialized with {1, 2, 3, 4}
- Some basic software engineering concepts may be tested
- Know object-oriented programming and design (abstract data types)
- Know the three cornerstones of object-oriented programming
- Know struct v class: struct is same except by default all data members are public; for classes all data members by default are private
- Know Try Throw Catch (exception handling)
    - Remember to have nested catch (and order of catch blocks matter); and remember catch(…) is the default catch block that catches everything
- Not on exam, but sidenote: namespace, it is another replacement implied scope for a scope (::).

```
// namespaces
#include <iostream.h>

namespace first
{
  int var = 5;
}

namespace second
{
  double var = 3.1416;
}

int main () {
  cout << first::va
  cout << second::var << endl;
  return 0;
}
```

    - So instead for example having to write std::variable, if you have the “using namespace std;”, then scope for all variables are from std scope, the you can just write the variable and the scope is implied (implicit in front of the variable, completed by compiler at compile time)

  - See week 4 notes for more details
- Know about inheritance, and the three types (public, private, protected)
  - Also know about public, private, and protected variables and their roles in the above types of inheritance
- Know about multiple inheritance; one class can inherit from more than one parent class
  -
```
class CRectangle: public CPolygon, public COutput {
class CTriangle: public CPolygon, public COutput {
```
- Know about linear v binary searches and how to implement, and when they can be applied; know time complexity of each
- Know about sorted v unsorted lists: array forms, linked list forms, and binary array and linked list forms
- Know about recursion and how to use, and when to use it (i.e., binary search)
  - Make sure to have at least one terminal condition (base case), and then also have recursive case
  - For example: use recursive function to find Factorial(n)
  - But Remember recursive calls can cause a lot of overhead; but sometimes there is no way around not using recursion for some applications
- Know about templates (function or class templates)
- Remember whenever you create a template object, you always have to use the <> brackets to specify which data type; template functions automatically will set the T generic-type parameter based on input passed through (and if input is allowed/compatible with the function)
- Remember how to write outside of scope of class for template class functions
- Know about pointers and arrays; remember a ptr variable internally is just an INTEGER that holds a 32-bit memory address (or 64-bit, depending on operating system/program type)
- Know about the nullptr/NULL and how to use it (i.e. at end of linked list so that you can use while loops until ptr == null); internally null == 0
- Know about linked lists and using ptrs
  - Using a struct for the node
- Know about an array of pointers (double or triple pointers)

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:



  -
  - Rule of thumb: * = 1D, ** = 2D, *** = 3D
    - Can use the above to allocate dynamic arrays of 1, 2 or 3 dimensions, respectively
- Remember the stack data structure: LIFO (last in, first out)
  - Know the basic functions: push, pop, etc..
- Remember the queue data structure: FIFO(first in, first out)
  - know the basic functions: enqueue, dequeue, isFull/empty, etc..
  - we can implement a queue with an array, but remember the special case for the front and rear pointers, you need to make sure to use either a 3rd pointer to the array, or introduced a reserved element
- we can also implement stack and queue via linked list
  - But: we have to a handle the shallow copy problem (for static array, you don't have to use a deep copy; shallow copy (default) is fine)
    - Thus we need to implement a deep copy for dynamic allocation to prevent memory errors
    - Need to copy VALUE ONLY; not memory address; this is why we need deep copy
  - Know/understand the deep copy function for a linked list

CIS 200 NOTES – taken based on Dr. Jie Shen W21 course– Demetrius E Johnson
*Anyone, feel free to use(:

```
template<class ItemType>        // COPY CONSTRUCTOR
StackType<ItemType>::
StackType( const StackType<ItemType>& anotherStack )
{  NodeType<ItemType>* ptr1 ;
   NodeType<ItemType>* ptr2 ;
   if ( anotherStack.topPtr == NULL )
    topPtr = NULL ;
   else                 // allocate memory for first node
   {topPtr = new NodeType<ItemType> ;
    topPtr->info = anotherStack.topPtr->info ;
    ptr1 = anotherStack.topPtr->next ;
    ptr2 = topPtr ;
    while ( ptr1 != NULL )    // deep copy other nodes
    {   ptr2->next = new NodeType<ItemType> ;
        ptr2 = ptr2->next ;
        ptr2->info = ptr1->info ;
        ptr1 = ptr1->next ;
    }
    ptr2->next = NULL ;
   }
}
```

- Know how/where to delete and insert a node in a linked list for queue or stack (or sorted/unsorted list)
- Know about polymorphism: dynamic binding
  - You have to have a pointer to the base class
  - Have to have a virtual function in the base class
  - Then use that pointer to point to derived classes and call then the appropriate virtual function will be called depending on which derived object is being pointed to
- Know about time complexity of the different sorting algorithms and when a given algorithm can be ued (sorted or unsorted tree/list)
  - Quick sort is similar to binary search (time complexity is also nlogn)
  - Merge sort: divide and conquer
    - Merge sort with linked list
  - Selection sort
- C++ has a qsort function, but you have to design your own comparison (parameter is a function pointer)
- Know about Binary Search Tree (BST)
  - Know the traversal methods: inorder, preorder, postorder
  - Know how to insert or delete items
  - Use a node class with two pointers