**NYUTB II (Prof Katz) (07/09/20)**

**LECTURE (10/09/20)**

**M15 OOP**

* “private” is the default access modifier for classes in C++ (if not given, assumed as private).
* If no constructors given, the immediate values of mem vars once obj is created will be garbage values.
* Constructor initialisation ( : mem\_var(value); ) is also called as member initialisation list.
* You can have any no of constructors for a class with diff params, but during an object creation, only one constructor of the matching params will be called.
* "**this**” is a pointer which points to the same object in which it is being called (though not necessary, you can always use **this**)
* Operator overloading (other oop languages like java, python got rid of it)
* =, [] can be overloaded only if it’s a member of a class.
* <<, >> cannot almost be overloaded as a member
* Choosing member vs non member - rewatch (All overloading rewatch)
* Return by reference if the op is \*= (we need to make the change and return the same obj)
* Pre increment op : Date& operator++(); same as above
* But Post inc op: Date operator++(int);
  + Int to diff between pre and post
  + Return by value because post inc returns the copy and does the add only on original
* Classes that Contain Dynamic Memory
  + Shallow copy issue
    - All classes have built in = and cons which can copy an existing obj
    - But when pointers invoked, the built in ops copy the pointers and not the value the pointer is pointing to.
    - This is called shallow copy issue
  + Big 3 is the solution
    - Big 3 functions - if you need them, you need them all.
    - Copy Constructor:, dest, overloaded = op
    - Include all the 3 funcs in class def if the class contains dynamic mem
* Inheritance
  + Its a **“is-a”** relationship.
  + Constructors of base class can be called from derived class cons but only through member initialisation method (given as : … {})
* Polymorphism
  + Pet = Cat  is always allowed.
  + Cat = Pet can be done only if = is overloaded op with param taken as pet.
  + Pet pointer = Cat  is always allowed.
  + Cat pointer = pet  is **never allowed**.
* Virtual Function
  + The version of the func called depends on type of the obj (RHS) and not type of the pointer (LHS) - *CHECK AGAIN*
  + virtual func () = 0; (makes it a pure virtual function)
* Abstract Class
  + If a class contains any pure virtual functions (even 1), it becomes an abstract class.
  + Abstract class cannot be instantiated.
  + Pointers of type abs class can be created and make it point to derived class obj.
  + The above pointer can be used to call the pure virtual func in the derived class.

**KEY NOTES SUMMARY**:

1. Objects are not passed by reference by default.
2. Default cons is available by default, but if you create at least one cons, then you need to also create default constructor else it won’t be available.
3. (\*b1).func() is the same as b1->func(); (\* lower order of precedence over . op)

**LECTURE W9**

**Intro Katz**

* **No math from now on (even on exams) :D**
* Finish reading and modules by wednesday, formulate questions and visit TAs by wed.
* If not attending webinar, send him mail of the code work that is part of lecture.
* Every Thursday mornings is webinar.

**OOP INTRO**

* Data and functions that work on this data both encapsulated as an object (models real life objects).
* Encapsulation to make sure user (programmer) doesn’t worry about inner implementations (ex: odometer in car)
* Anything that needs to be done, should be handled by internal mechanism of the obj than us doing using a manual. Thats the idea of OOP. Internal mechanism should be automated and need not be a worry of users (programmers).
* We need to protect the data (And give some condition)
  + This can be done through making it private.
  + Default is private
  + Read and write comes together, cannot do restriction only for one case
* Function should be inline only if it’s 1 line of code, literally (func def inside class def is called inline func)
* Getters can be constantified always
* Objects can be considerably large, so objs should be passed by reference (to not make a copy of it every time)
* Const reference should be used while passing to func, to ensure the func will not change the obj’s value.
  + If a constified obj is used as a param in a func, all functions of that obj used inside the func (that shouldn’t be changed any value) should be a constified func.
* A class cannot have it’s own class directly, but you can use a pointer of it’s own class type **XXX**
  + Ex: Person has a person is not possible
* “this" is a pointer which always points to the calling object.
* Private refers to CLASS restriction not object restriction **XXX**
  + Spouse of other object can be accessed through this object because both are of the same class.

**OVERLOADING**

* Operators are just functions with strange syntax for function call.
* Diff types of operators are
  + Unary
  + Binary
  + Ternary
* =, [ ] can only be overloaded as a member func of that class.
* Some almost cannot be overloaded as a member <<, >>
* Post inc vs pre inc
  + ++++x is valid (++(++x))
  + x++++ is not valid
  + Pre
  + Post

**INHERITANCE**

* Creates a is - a relationship.

**OOP KEY-NOTES**

1. C++ allows us to drop const in a mem function when we redefine it in derived class.
2. No need to include keyword “virtual” in function definition (only req in declaration).

***DOUBTS - PROGRAMMING***

1. Comparable?
2. && when?
3. Do we need the Big 3 for DS classes (ideally when should I include those?)
4. Do we need to study splay trees, B-trees, sets and maps also?
5. Height diff > 1 doesn’t matter in RBT - balanced?
6. Creating an object locally inside a loop (static memory) and adding it to LL doesn’t it go out of scope?

**W12 Lecture**

1. Apple changes kernel more than Microsoft

**W13 Lecture**

1. Using thread and sharing resource - create asynchrony problem - use critical section.
2. Java - synchronize keyword creates somophore.
3. Every body in java is a somophore.
4. Java makes it easy to synchronize and make it thread safe.

**10.1 STRUCTURES**

***Introduction***

* Sometimes it’s useful to have a collection of different types of values together treated as a single item.
* It can be done through structs (or even classes - advanced)
* Structs are simplified kind of classes.
* A structure can be thought of as an object without any member functions.
* Syntax
  + struct Struct\_name { member functions; };
  + Ends with a ; **XXX**
* The name of a structure is called **structure tag.**
  + Should be a mix of upper & lowercase beginning with Upper.
* The name of vars used inside are called **member names.**
* **Location:** Placed outside of any function, generally above all functions.
* Structure Tag can be used to define variables of struct type inside functions.
* **Structure variable (account)** - Variable created using the structure as its type.
* **Member variable (balance) -** struct\_variable.member\_name
* **Member value (1000)** - value assigned to a member variable.
* A structure’s value is a collection of all its member values.
* Member values are stored in **member variables.**
* Two structs can have the same member names.
* Structure value of one struct var can be assigned as struct value of another struct var if both of the struct vars are of the same stuct type.
  + Example
    - Fruits apples, oranges;
    - apples = oranges; (same as assigning all member values to each other)
* PITFALL: Leaving semicolon at the end of struct
  + We have ; after struct def because we can def struct vars immediately after definition.
  + Example : struct Fruits { member vars; } apples, oranges;
  + But, always we try to declare the struct vars separately later.
* Dot Operator (.) is used to specify a member variable of a structure variable.

***Structures as Function Arguments***

* Structures can be used as args and be passed as value or reference.
* Structures can also be returned from a function (func can be used like to fill the mem vars)

***Structures Within Structures (has - a relationship)***

* Structures can have its members as other smaller structures (Example : Person -> Birthday)
* To access the member var of inner structure —> person1.birthday.year (2 dot operators).     **XX**

***Initializing Structures***

* Structs can also be initialised during declaration.
* Ex: Date myDate = {12, 06, 20}
* The values should be given in order of the member vars (here, (date, month, year))
* If more values are given than members - it throws an error
* If less values are given than members - other values are initialised to 0 of the given type.
* PS: Structs can have access modifiers and also member functions (But forget hearing this)

**10 INTRODUCTION**

* A class is a user-defined data type.
* Can be used the same way as pre-defined data types like int, char, double, etc.
* Things to know: What makes a good class definition, techniques to define classes that is in accordance with the modern programming practices.

**INTRODUCTION TO OBJECTS AND CLASSES**

* An object is a variable of the type class that has both data and functions that work on its data.
* Function that is associated with the object is called **member function** and the data that is associated with the object is called **data members.**
* The type of the objects determines its member functions.
* Two objects of the same type can have different values but same member functions.
* Data Type whose variables are objects are called classes. We generally call it just “type” in the context of object.
* Calling a member function - Syntax:
  + calling\_object.member\_func\_name(arguments);
* Definition of a class should involve two things
  + What kinds of values the variables (objects) can hold.
  + What the member functions are.

**10.2 CLASSES**

* Class is a user defined data type that defines what kind of values and the member functions the variables (objects) can hold.
* Structures + Member Functions = Class
* Syntax:
  + class Class\_Name {
  + public/private/protected:
    - mem\_vars;
    - mem\_func; };
* Inside the class - always define the access restriction : public, protected, private (details later)
* Member function and member vars ordering can be inter-mixed.
  + Style here: mem funcs 1st and then mem vars.
* Member func can also have it’s func only declared and defined later (outside classes as well).
* **Encapsulation:** Combing mem vars and funcs into a single package, such as an obj of a class is called encapsulation.
* If mem func is defined outside the class - syntax will be
  + void Class\_name::func\_name()
  + **::** **scope resolution operator** —> same as dot operator (purpose : to tell what a member func is a member of) but dot op is used with objects and scope res op is used with classes.
  + The class name that precedes SRO is called **type qualifier.**
  + Inside func definition, you can use all the members (vars and funcs) of the qualifier class wo DO.
* You can build a library of your own class type def and use it as predefined types.
* You can place class def into a separate file and use it in a program that uses this type.

***Public and Private Members***

* Class def should separate the rules for using the class vs details of implementation of class as strong as predefined types.
* With an **ideal class def,** you should be able to change the details of implementation of the class (mem vars and mem funcs --> Ex: int month vs char m1,m2,m3) and the program that uses this class should still work wo any changes in the main program.
* To make that happen, you need to have enough member funcs (**setters and getters)** so that you don’t need to access the mem vars directly, instead only through member funcs (ex: using input() func to get inputs for mem vars instead of getting inputs for those mem vars directly in main)
* **Foolproofing this using private member restriction**
  + Member variables of the class are declared as private members (private: )
  + If a member var from a class is private, it can only be accessed by member functions of the same class and no where else.
  + It’s a severe but wise restriction to impose.
  + Makes the code easier to und, read and update if all mem vars are private.
  + All the members of the class can be made private members (both mem vars and mem funcs)
  + They will be called **“private member variables”** and **“private member functions”.**
  + **Private mem vars** enable us to set some conditions on initialising them (Ex: check date() func inside input() and set() to see if the values given are within range)
  + **Private mem funcs** enable us to use that func only inside other mem func of the same class (as helper functions, ex: checkdate()) and cannot be accessed anywhere else.
* **Private Members Uses Summary**
  + Disables access from main.
  + Add conditions over initialisation.
* **Public Members** (vars and func) have public access so they can be accessed from main func and even from non member funcs.
* Any no of occurrences of public and private can be there in a class (ex: public: , private:,  public: , etc)
* It is always good to label all the members in one of the access restriction category.
* **PROGRAMMING TIP**
  + Always make all member variables private (only mem func access).
  + Define accessor and mutator functions
    - == op cannot be used to check equality of two obj unless overloaded.
    - Therefore, getters are used to access each mem var and they are compared individually to compare 2 objects.
    - These getters are called **accessor functions.**
    - Setters are called **mutator functions.**
    - Always have accessor and mutator funcs to all vars in a class.
  + You can use = operator with objects
    - You can assign one object to another (today = birthday) even though their member vars are private.
    - For this to happen make sure the object birthday has their mem vars assigned with values already.
* **Properties of a class - Summary**
  + Class have both member variables and member functions.
  + All the members should be either public or private.
  + All the member variables should be private.
  + Private members can be accessed only within member functions of same class.
  + A member function can be overloaded (multiple functions with the same name)
  + A class can have other classes as the type for the mem variables (**has a** relationship).
  + A function can have formal parameters whose type are classes.
  + A function can return objects of some class type (return type can be a class).

***Constructors for Initialisation***

* A special type of member function is called **constructor.**
* Constructors are used to initialise member variables when objects are declared.
* Constructors are automatically called when an object of that class type is declared.
* Constructors are used for both init of mem vars and also “other sorts” of initialisation.
* Syntax Notes
  + Constructor should have the same name as that of class.
  + Cannot return any value and also no void - so return type should be mentioned at all.
  + Syntax: **Class\_Name (param 1, param 2, …);**
  + Object initialisation : **Class\_Type Obj1(arg1, arg2,…)**
* Constructors access modifier should only be **public XXX.**
* Constructors can be overloaded as well to initialise objects in diff ways.
* If you have a good set of constructors, mutator funcs can be omitted (but you should have it if you want to edit the mem var values later).
* **Empty / Default Constructors**
  + If constructors are declared with no params  **(Class\_Name (); )** - object declaration will have 0 as values initialised to the mem vars (This is the same as default constructor).
  + Definition: *Class\_Name***::***Class\_Name***() :** *mem\_var1***(0),** *mem\_var2***(0.0), … {}  XXX**
  + : mem\_var1(0), mem\_var2(0.0), …
    - This segment is called member initialisation list.
    - Syntax -> : followed by a list of all or some mem vars with init value in ().
    - Body need not be empty (you can have other conditions and checks as well)
    - For mem vars given in initialisation section, values can be formal params as well (for constructors with params)
* Constructors can be called explicitly from main
  + Ex: account1 = BankAccount(500, 99, 5.5)
* Initialisers can also be given when an obj is created as a dynamic variable.
  + Ex: BankAccount \*myAcc = new BankAccount(100, 99, 4.8); (ideally you should have it inside class def)
* **PROGRAMMING TIP** (Always include a default constructor)
  + A constructor that can be called without any args is called a default / empty constructor.
  + Compilers create default constructors automatically if there are no other cons defined in the class.
  + But if there are cons defined and there are no default cons, compiler will throw an error.
  + Therefore, always good to define a default constructor.
* **PITFALL** (Constructors with no args)
  + If you want to call constructors with no args, just simply create obj with no ()
  + Ex: BankAccount myAccount;
  + But if you want to call constructors with no args in explicit constructor calls, you need to include ().
  + Ex: myAccount = BankAccount();
* **Member initialisation while declaration:** initialise mem vars in the class itself to let the objects have that value by default if no args are mentioned during declaration: Ex: BankAccount myaccount;
* **Constructor delegation:** Same thing can be achieved by including another constructor with values inside default constructor so that init will be by default the given values even if not mentioned.

**10.3 ABSTRACT DATA TYPES**

* Data type is not only about the collection of values but also the type of operations defined on them.
* A data type is called ADT if the programmers who use them does not have access to the details of how the values and operations are implemented Ex: int (you do not know anything about implementation)
* Classes and structs (programmer defined types) are not automatically ADTs, but should be implemented like ADTs.
* To make a class ADT, you should define it in a certain way.

***Classes to Produce ADTs***

* A value for a class type is the set of values of all it’s mem vars.
* A class can be made ADT if it’s **implementation details** are **separated** **from usage** for a programmer.
* It means that any change in the implementation should not affect the usage for the programmer.
* Ex: In BankAccount class, mem var “balance” can be implemented as 1 double or 2 ints (dollars and cents), and corresponding changes can be made to cons and mem funcs, so that it’s usage in main will not get affected (aka the details of imp and its change doesn’t affect the programmer who uses it).
* One way to ensure this “**separation**”  is to follow these rules: **Class Design as ADT (XXX)**
  + Make all mem vars private members.
  + Make all required basic ops as public mem funcs and specify how to use them in comments.
  + Make any helping funcs private.
* To check if ADT is implemented correctly - change the implementation of ADT (things in private mem and mem funcs /cons def) and nothing in the main - the program should work fine still.
* **Benefits of making a class ADT**
  + Separate imp vs usage
    - Only interface will be available for programmers.
    - Changes in imp will not affect usage.
  + Delegate imp and usage to diff programmers
  + Divides big task into 2 small tasks so it becomes easier to read, implement, debug and maintain.
* Here information hiding is applied to both data values and functions (private members) as opposition to information hiding seen in chap 3 (only functions as a black box)

***Interface***

* **Interface of ADT** (= things in public members in a class)
  + The interface of ADT tells you how to use the ADT class in a program.
  + If the ADT is a class, interface consists of public member functions (only declaration) of that class along with comments that tells you how to use them.
  + Interface of ADT is all you should know to use it in a program.
* **Implementation of ADT** (= things in private members in a class + mem func / constructors def)
  + This tells you how the interface of ADT is realised as a C++ code.
  + Implementation of ADT consists of private members and the definitions of both public and private mem funcs of the class.
  + Although you need implementation to use the ADT in a program, you should not worry about it’s details while using the ADT in the main program and in ordinary functions.
  + You should think of the imp of ADT as a black box (like the one in procedural abstraction Chap 4)
* Interface and Implementation of ADT should be placed in separate files, also separate from the main program which will use the ADT (3 files - inter, impl, main).

**10.4 INHERITANCE**

* One of the most powerful features of OOP is the use of **derived classes** (inheritance).
* Inheritance allows you to define a **general class** and later design some **specialised classes** that adds some more details to the existing general class.
* If we say class B is a derived class of class A, class B has all the features of class A and also some extra features.
* Convention to draw (flow chart) —> Arrow from derived to base class.
* Metaphor used - family hierarchy and inheritance
  + Derived class - Child of base class
  + Base class - Parent of derived class
  + Child class inherits members of parent class
* Syntax: **Class** *Derived\_Class\_Name* **: public** *Base\_Class\_Name* **{};**
* Benefits
  + Code reuse
  + Simplification (no need to make changes at multiple places)
  + Code easier to maintain and write
* Takes time and effort to get well versed with designing classes using inheritance.

**CHAPTER 11**

**11.1 FRIEND FUNCTIONS**

* We can define “operations” on objects as non member (ordinary) functions.
* Ex: Equality Function (takes 2 date objects and check if their months and days are same using accessor methods)
* A friend function gives a non member function (ordinary functions) access privileges as that of a member function.
* A friend function of a class can **directly read or write the values** of the private member variables of that class without using accessor and mutator functions (also direct access to private mem funcs)
* An ordinary non member func can be made a friend function by declaring it as a friend function inside the class definition.
* Syntax: (Inside the class def under public access) (can be placed in private access but still will be public)
  + **friend** function\_declaration();
* Friend function behaves just like normal functions
  + Func def does not include class and access operator (BankName::)
  + Func call does not include object and dot operator (obj.equal())
* You can achieve the task without having it as a friend func by using accessors and mutator functions.
* But the only reason to make a normal function as friend function is to simplify the def of the function and to make it more efficient (Reasonable point).
* **PROGRAMMING TIPS**
  + Define both accessor / mutator and friend functions (A & M would be used for other purposes as well).
  + Use both member and non member functions -  A simple rule to decide between making a function member func or non member (friend) func
    - Make it a member func if the task being performed involves only one object, else make it non member (friend) func if it involves more than 1 object.
* Example Program : Money Class
* **Const Parameter Modifier (XXXXX)**
  + Call by reference is more efficient than call by value (since CBV creates 2 copies).
  + This efficiency is more important while having objects in formal params.
  + Thus it makes sense to use a call by ref for formal params with objects even though the function doesn’t change the value of the param.
  + If you know that the value shouldn’t be changed, and want to make it efficient by making it call by ref, then couple the call by ref with const modifier (Ex: **const** **string&  str** —> one you’ve seen in your IDE)
  + Such parameter is called a **Constant Parameter.**
  + Members functions called by objects are call by reference by default (since value of mem vars of the obj gets changed by the mem func)  **XX**
  + To ensure the mem func should not change the value of the mem vars of the object, you can make the params in the mem func a constant parameter.
    - Syntax: return\_type mem\_func\_name (params) **const**; (Here const comes at the end after parentheses) **XXX**
  + **PITFALL** (inconsistent use of const) **XXX**
    - const modifier is an all or nothing proposition.
    - If it is used for one param of a particular type, then it should be used for every other param of that type for which the value will not be changed.
    - If the type is a class type and if you use const, then you should use const for all the member functions of that class type used inside that func which doesn’t change the value of it’s calling object.

**11.2 OVERLOADING OPERATORS**

* Operators are nothing but functions defined for that type but with a little syntax variation (on how you use it)
* An operator (+,-,\*, etc) can be overloaded the same way as functions.
* Operators can be overloaded to accept arguments of type  - “classes”
* You can overload most, but not all operators.
* The operator need not be a friend of the class but often you want it to be.
* Syntax
  + Return\_type **operator +**(param1, param2){} (User “**operator**”  keyword and the type of operator instead of the function name)
* Rules on Overloading Operators
  + At least one of the operand should be of the class type.
  + Overloading operator func can be a friend or ordinary func or even member function.
  + You cannot create a new operator.
  + You cannot change the no of operands the op can take (usual nos only allowed ex for +, only 2 ops)
  + You cannot change the precedence of the operator.
  + Following operators cannot be overloaded ( . , :: , : ,  .\* , ?: )
  + Some operators are overloaded in a diff way ( =, [], —>)

***Constructors for Automatic Type Conversion***

* The system will perform certain type conversions automatically.
* These automatic type conversions are produced by constructors (1st initialise them to a matchable temp obj and then does the task)
* Ex: If + overloaded with 2 operands both as classes and if the overloaded + is used like

full\_amount = base\_amount + 25 (one of type Money class and one of type int), then the compiler checks of overloaded + with 1 class and 1 int, if not there, the compiler checks for constructor with 1 arg as int, if present, it converts int 25 to Money 25 and does the addition.

* So single arg constructor is used to convert the int 25 first into a temp Money obj, before adding them both.
* If the single arg constructor is not present or if double 25.0 is given and no single arg constructor takes double value, then the compiler throws an error.

***Overloading Unary Operators***

* Unary operators like (- (neg), ++x, --x) can be overloaded the same way as above.
* But Unary operators like x++, x-- (postfix) are overloaded differently (covered in modules / lecture).
* << >> operators are binary operators with 1op stream and 1op string to be considered.
* << >> can also be overloaded the same way as above but there are few concerns such as what should be returned from the overloaded operator?
  + Looking at the chain of couts (((cout << “I have”) << amount) << “dollars\n”); - each sub expression should return “cout” which is of type ostream.
  + So the overloaded << should return the 1op (cout).
  + Therefore overloading op func syntax is
    - **ostream& operator <<(ostream& outs, const Money& amount);**
    - Yes, the operator (or function) should return a reference (ostream**&**) instead of value.
    - This is because ostream is a stream, when it comes to stream, you need to return the stream (reference object) itself instead of the actual values like entire file / keyboard / screen.

<<<< DO THE MONEY PROBLEM GIVEN IN DISPLAY 11.8 >>>>

**11.3 ARRAYS AND CLASSES**

* Arrays can be combined with classes to form array of classes (objects) or classes with arrays as member variables.
* An array who’s base type is a class is treated the same way as arrays with primitive data types.
* Array with classes usage syntax
  + arr[ i ].mem\_var1; arr[ i ].mem\_var2;
* If the array is a member variable it can be accessed as obj.arr[ i ]

**11.4 CLASSES AND DYNAMIC ARRAYS (Reinforcement Needed)**

* A dynamic array can have a base type as a class.
* A class can have a dynamic array as a member variable.
* Dynamic variables / arrays stays even if the pointer is a local variable and goes away at the end of a function call.

***Destructors***

* Destructor is a special member function which deletes the object, when it goes out of scope.
* A destructor is called automatically when an object goes out of scope such as
  + Function ends.
  + Program ends.
  + Block with local variable as object ends.
  + Delete operator is called.
* For dynamic variables / arrays used in the imp of class, they cannot be deleted by the programmer (doesn’t even know), hence there’s a special kind of member function called **Destructor** to do this job.
* A destructor is a member function that is called automatically when an object goes out of scope.
* The destructor contains delete to destroy all the dynamic vars/arrays used by the object.
* Destructor can do other clean up jobs as well (details not given)
* Syntax: **~Class\_Name(){}** (Just like constructor but with an added ~)
* A destructor cannot have any params.
* There can be only one destructor for a class.
* Destructors cannot be overloaded (since only one destructor per class)

***Copy Constructor***

* Pointers used as call by value parameters also change the actual value pointed by the pointers.
* In order to avoid this for class types, copy constructors are used.
* Copy constructor creates new dynamic array for the object’s value to be copied so that change made here will not affect the actual copy (2 pointers are not created and pointed to the same array, 2 sep dynamic arrays are here)
* Syntactic Notes
  + A copy cons is a cons that has 1 param of the same class type.
  + This 1 param must be a constant param and call by ref (const Class\_Name& param)
* Whenever C++ needs to make a copy of an object, it automatically calls the copy constructor. In particular 3 circumstances
  + When a class object is declared and initialised by another object of same class.
  + When a function returns a value of the class type.
  + Whenever an argument of the class type is “plugged in” for a call-by-value parameter.
* **When you need a copy constructor?**
  + If a class definition involves pointers and dynamically allocated memory using the *new* operator, then you need to include a copy constructor.
  + Classes that do not involve pointers or dynamically allocated memory do not need a copy constructor.
* PS: Copy constructor will not be invoked when you set one object = another using = operator.
* **THE BIG THREE**
  + Copy constructor, overloaded = operator and destructor are called the BIG THREE, because when you define any one of them, you need to define all the three.
  + Compiler can auto generate these but won’t behave as you wish (especially when mem vars have pointers, dynamic arr or other classes)

**Copy Constructor (Outside resources)**

* Copy constructor initialises another object using an object of the same class.
* Syntax: Class\_Name (const Class\_Name& old\_obj)
* Copy constructors can be made **private** to make the object of the class ‘**non-copyable’**
* Why param inside copy constructor is call by reference?
  + Because copy constructors are called when an object is passed by value.
  + If we pass by value inside copy constructor it will again call a copy constructor to pass by value, which results in infinite chain of copy constructor calls.

**When is Copy Constructor Called?**

1. When an object of the class is ‘returned by value’ (Returns the created copy)
2. When an object is ‘passed by value’ (to a function) as an argument (creates a copy for the param)
3. When an object is initialised from an object of the same class (Class\_Name newObj = oldObj)
4. When the compiler creates a temporary object (wherever applicable)

**Why User-Defined Copy Constructor Needed?**

* Compiler creates it automatically.
* But when an object has pointers or any dynamic allocation of resources (memory, file handle, network connection, etc), we need t create copy constructors (to enable deep copy for pointers)

**Copy Constructor Vs Assignment Operator**

* MyClass t1, t2;
* MyClass t3 = t1; —> Copy Constructor (because new obj is created based on existing obj of same class)
* MyClass t2 = t1; —> Assignment operator (because t2 already initialised using default constructor)

**15 INHERITANCE**

* Inheritance provides a new dimension of abstraction.
* This enables us to create a general form of a class, and later create specialised classes that can inherit all the features of general class and also have their own set of additional features.

**15.1 INHERITANCE BASICS**

* Inheritance is the process by which a new derived class (child) can be created from another class called base class (parent).
* Derived class can have all the members of base class and also it’s own members (vars / funcs).
* If we create objects of derived class, we can use that object to access mem vars and mem funcs of the base class.
* If we define same function in both derived and base class, we will have options to either “*re-define”* or “*override”* the function in derived class (details later).
* Even when an obj of derived class is passed to a func with param as base class, the mem func with same names will try to invoke the func of the derived class (details later).
* Only **public** mem funs and mem vars can be **inherited** by the derived class. **XXX**
* You only list the declaration of the inherited mem func in the derived class - def if you want to change the def of the func.
* Inheritance syntax: **class** *Derived\_Class\_Name* **: public** *Base\_Class\_Name* {}
* We use the keyword public to mention that all public members of the base class will be inherited.
* The definition of a mem func from base class can be changed in the derived class so that it has diff meaning in the derived class. To do that,
  1. Include the func declaration in the derived class def
  2. Include the func definition (re-defined) in the derived class imp
* This is call **re-defining** the inherited member function.
* **ADV** : Inheritance allows you to use the code written in the base class (Employee) in the 2 derived classes (HourlyEmployee and SalariedEmployee) - thus enabling reuse of code and easy maintanence.
* **Family Terminologies**
  1. Base class - Parent class
  2. Derived class - Child class
  3. Parent of base class - Ancestor class for derived class
  4. Child of base class - Descendant class for ancestor class
* Specialised member functions (like constructors) are not inherited from base class by default.

***Constructors in Derived Classes***

* Constructors are not inherited from base class, but you can invoke a cons of base class from cons of derived class (normal flow).
* All derived class constructors should have calls to base class cons to initialise the inherited mem vars and to allot them memory.
* If there is no call to the base class cons, the default cons (0 arg cons) of base class will be invoked automatically (so explicit call to def cons of base class can be omitted, however the preference is to call) - if there is no def arg given in base class, it leads to an error condition.
* If multiple levels of inheritance happened (C from B, B from A), the cons of C should have call to cons of A 1st, then call to cons of B 2nd and then the setup should happen for C.
* **An object of a derived class has more than one type (XXX)**
  + An object of a derived class can be used anywhere that an object of it’s base class is allowed.
  + Ex1 : Obj of type HourlyEmployee can be passed as args to a func with formal param as type Employee.
  + Ex2: Obj of type HourlyEmployee can be assigned to a var (obj) of type Employee.
  + The vice versa is not possible and not allowed.
  + More generally, an obj of descendant class can be used anywhere an object of it’s ancestors are allowed.
* **PITFALL**
  + Use of private member variables from the base class
    1. Private member vars from base class can be directly accessed only within the def of the mem func of the base class.
    2. A mem func / mem var of the base class cannot be directly accessed by any other funcs, not even the mem func of the derived class (even though the mem func is a redefinition of the base class’s mem function).
    3. It can only be accessed using the public accessor and mutator functions of the base class.
    4. This is because,
       1. We cannot compromise the variables just by letting the access by creating derived classes
       2. Also this helps us to avoid inappropriate changes of the values of the mem vars of base class by mistake (remember accessor and mutator funcs can have conditions to check for wrong values)
  + Private member functions are effectively not inherited.
    1. Remember private mem vars and funcs are not directly accessible outside interface and imp of base class.
    2. In case of private mem vars, it can be accessed through accessor and mutator funcs. In case of private mem funcs, it’s simply like, the func is not accessible (take is as not inherited).

***Protected Qualifer (Access Modifier)***

* If mem vars and mem funcs are marked protected, instead of private in a base class, it can be accessed directly by it’s derived class (and also derived classes of the derived class) but everywhere else, it’s like private.
* Many argue, it’s use is bad style, since it compromises the principle of hiding the class implementation.
* Therefore you need to make your own decision whether or not to use the protected qualifier.

***Redefinition of Member Functions***

* If a derived class requires a different implementation for an inherited mem func from base class, it can be **re-defined** in the derived class.
* To re-define an inherited mem func from base class, it’s fun declaration should be listed in the derived class def (despite no changes) and it’s redefined def should be made in the derived class implementation.

***Redefining vs Overloading***

* In the derived class, When a func has the same name as that of a func from base class but has diff type params or  diff no of params, then it is called **function overloading.**
* In derived class, If both functions have same name and type and no of params, if only the def of them varies, then the function has been **re-defined**
* **Signature**
  + A func’s signature is it’s func name with sequence of types in the params list (excluding const and &).
  + When you overload a function, then the overloaded function will have different signature than the original function.
  + PS: Some compilers consider const vs no const as overloaded, ie considers const as part of the func signature, but that is best avoided.

***Access to a redefined base function***

* You can access the base class version of a redefined mem func using an object of the derived class by using the scope resolution operator
* Syntax:  *derived\_class\_obj***.***Base\_Class\_Name***::***redefinedFunc*(); —> instead of accessing the function directly as *derived\_class\_obj***.***redefinedFunc*();

**15.2 INHERITANCE DETAILS (Reinforcement Needed)**

* Inheritance with classes that use dynamic arrays or pointers
* Special member functions like constructors, destructors and copy constructors and private member functions are not inherited by derived class.
* Overloaded = operator won’t not be inherited. So when you use = in derived class, it will be the default C++ = operator.
* It’s because these functions need extra details about mem vars of derived class for them to function.
* = & copy constructors should be declared as member functions in derived class and their definitions should invoke the base class’s = & copy cons first to make them work for inherited mem vars of derived class and then the setup goes on.
* Destructors in Derived Class
  + Base class destructors are invoked automatically all the time when a derived class’s destructor is called.
  + Therefore, no need to invoke a call to base class destructor explicitly inside derived class’s destructor
  + Order in which destructors are called are reverse of the order in which constructors are called

**15.3 POLYMORPHISM**

* Polymorphism is the ability to associate multiple meanings to one function name.
* Refers to a very particular way of associating multiple meanings to a single function name.
* Polymorphism refers to the ability to associate multiple meanings to one function name by means of a special mechanism known as **late binding.**
* 1 of the key components of OOP

***Late Binding***

* A **Virtual Function** is one in some sense used before it is defined.
* When you make a function virtual, you are basically telling the compiler that, at this point of time I don’t know how the function is implemented, wait until it is used in a program and then get the implementation from the object instance.
* Ex : Center func from base class Figure which uses draw function which has diff def for diff Figures like Triangle which while writing center for base class Figure which uses draw, at that time you do not even know (written) the def of draw as required for Triangle - Make it virtual.
* The technique for waiting until run-time to determine the implmentation of a function is called **Dynamic / Late Binding.**
* Some technical details you need to know in order to use a virtual function
  1. If a func will have diff imp in derived class and you want it to be a virtual function, you add the keyword “virtual” to it in the base class (you also need to add it in derived class but that’s only style pref, by default it will be virtual)
  2. Keyword “virtual” is added to the function declaration and not to the function definition. **XX**
* You can make all functions virtual, but the only reason not to do so is **“efficiency” -** Compiler and run time environment does lot of work for virtual functions.
* **Overriding  vs Redefining**
  1. When you change def of a virtual func in derived class its called function **overriding.**
  2. When you change def of a non virtual func in a derived class its called function **redefined.**
  3. From a programmer’s perspective, you do the same thing for both (changing the def of mem func) but the compiler does diff job so diff names.
* Polymorphism, Late Binding, Virtual function are all really the same topic (inter-connected), to achieve one goal —> Associating multiple meanings to one function.

***Virtual Functions and Extended Type Compatibility***

* C++ is a strongly typed language - types of items are always checked and compiler throws an error if there is type mismatch such as between args vs params (cases when no auto conversion can be invoked).
* In few cases C++ does coercion (auto type case) --> Int to Double Types (char, int to float / double) but not Double to Int types (vice versa)
* But strong typing interferes with idea of inheritance in OOP
  + Pet objP (Base) Dog objD (Derived), you can do this assignment —> objP = objD but not the vice versa.
  + objP loses the breed field (mem var of objD which is not in objP) which is called **slicing problem.**
  + However,  if you don’t want the pet obj to lose the breed value which is a dog mem var, you can do that using pointers to dynamic object instances.
    - Pet \*ppet; Dog \*pdog;
    - pdog  = new Dog;
    - pdog->name = “Tiny”; pdog->breed = “Labrador”;
    - ppet = pdog;
    - Here since ppet is a pointer to a pet obj, making it point to dog obj will enable it to access both mem vars *name* and *breed.*
    - But this breed mem var can be accessed only inside virtual mem func and not directly.
    - Ex: cout << ppet->breed;    // will produce an error

**PITFALLS**

* **Slicing Problem**: Assigning derived class obj to base class obj will make the base class obj lose the mem vars and mem funcs of the derived class obj.
* **Not using Virtual Member Functions:** If you don’t use virtual mem functions when using pointer vars to assign objs the above way, you cannot access the mem vars and mem funcs of the derived class obj.
* For all this craziness, keep two simple rules in mind
  1. If the domain type of the pointer p\_ancestor is a base class for the domain type of the pointer p\_descendant, then the following assignment of pointers is allowed:

p\_ancestor = p\_descendant;

    Moreover, none of the data members or member functions of the dynamic variable being pointed to by      p\_descendant will be lost.

2. Although all the extra fields of the dynamic variable are there, you will need virtual member functions to     access them.

* If you do incremental development (code a little, test, code a little, test, etc), if you code a class with virtual function and if you have not coded the definition of that virtual func, then the compiler will throw some hard to understand errors.

**PROGRAMMING TIP (Make Destructors Virtual)**

* Base \*pB = new Derived; delete pB;
* In the above case, when delete is called, the destructor of Base will be invoked and ends and the dynamic variable of derived obj remains.
* If the destructor is marked virtual, the above delete call will call the destructor of the derived class first (due to virtual, it takes derived class’s dest) which deletes the derived class dynamic obj and then invokes the base class’s destructor which will delete the base class dynamic obj pB.
* Thus it is always better to make destructors virtual.

**OOPS SUMMARY** [Link](http://cs.stmarys.ca/~porter/csc/common_341_342/notes/oop_3pillars.html)

**C++ POINTERS SUMMARY** [**Link**](http://cs.stmarys.ca/~porter/csc/common_341_342/notes/pointers.html)

***Oops***

* The type of the objects are called **classes** which have the blueprint for the object’s member variables and member functions.
* The classes are basically **user-defined** data types which can have different primitive data types or other class objects (has-a relationship) as members inside them.
* Each instant of the class is an object (**instantiation**).
* Each object of same class will have its own data but same member functions.
* **Access Modifiers**:
  + Modifies the access restriction of the members of the class.
  + This is done to avoid **unauthorized modifications** to the data of an object from outside, provide **conditions** on data modifications and also to enable **abstraction** to make the class **ADT** (interface vs implementation).
  + **private**: members are only accessible by its class member functions and not outside the class.
  + **protected**: members are only accessible the class and the derived classes.
  + **public**: members are accessible to everyone.
  + **Tip**: Always make all member variables private.

***Constructors***

* Special kind of member functions are constructors.
* Constructors are used to **initialize** member variables during object instantiation along with conditions.
* Constructors can be **overloaded** for different types of input during object instantiation.
* If you include a constructor, default constructor (empty constructor) should also be provided.
* Constructors can be called explicitly to assign values after object instantiation.
* Types
  + Empty (default) constructors
  + Arg constructors
  + Copy constructors
  + Destructors

***Encapsualtion***

* Data members and functions that operate on data are bundled together into a **single** **unit** called objects. This is called encapsulation.
* Also, encapsulation refers to information hiding.
* Process or method of **hiding data** in a single entity like object from outside world (information hiding).
* Can be done using access modifiers.
* Disables access of data of a class from code outside the class.

***Data Abstraction***

* Hiding the class implementation details from the user in order to avoid complexity, so that the user should only know how to use the class, rather than how it is implemented.
* Process or method of hiding **unwanted information** from outside world (implementations).
* Relies on separation of **interface** from **implementation.**
* We define classes as ADT classes to provide data abstraction.

***Friend Functions***

* Friend function gives a non-member ordinary function – member function level privileges.
* A function or class can be made a friend.
* Mostly done to avoid the use of accessor and mutator functions.
* When Member Function vs Friend Function?
  + Member function when the operation is performed only on 1 object (**this**).
  + Friend function when the operation is performed on 2 objects (this and another object) – generally operator overloadings.

***Constant Param Modifiers***

* Param as **const** type **&**
* Make it only **&,** if object has to be passed by reference and changes has to be made to that actual argument.
* Make it **const** and **&,** if passed by reference and safety ensure that no changes has to be made to that argument (just done to save memory by not creating a copy of that object).
* Make the member function as **const** (in the end) if that member function should not change the values of the member variables of this object.
* Always constantify (**const &**) if objects are passed as params.
* It’s an all or nothing proposition – if const, other inner functions should be const if no edits.

***Operator Overloading***

* Done to support basic operations on objects as operands.
* Some operators are overloaded by C++ compilers **by default** (**=** and **&**)
* Some operators cannot be overloaded (**:: .** **.\*** **?:**)
* Some operators can only be overloaded as member functions (**=** and **[]**)
* Some operators almost cannot be overloaded as mem func (**<<** and **>>**)
* Operator overloading can be done in two ways
  + operator+(a,b) – **non member**
  + a.operator+(b) – **member**
* Some operators change the data in an object, some return a new object.
  + a + b returns a new object.
  + a += b changes the calling object a.
* Procedure
  + Create a temp object.
  + Do the corresponding operation for each member vars of temp object.
  + Return the temp object.
* Constructors will enable automatic type conversion (if provided) if one of the operands is of non object type.
* When to choose member vs friend for overloading? **[IMP]**
  + a + b - - works on both.
  + a + 5 - - works on both (auto type conversion given above).
  + 5 + a - - works only if friend (cos this call will look like 5.add(a) and 5 isnt an object)
* What to Return? **[IMP]**
  + The **value** returned depends on what operator should do.
  + The return **type** depends on what is being returned.
    - If item returned is created as **local obj** inside, should return by **value**.
    - If item was a passed in **param** or **this**, should return by **reference** (whenever **\*this** return by &)
  + Return by reference is generally preferred.
* Pre vs Post Inc / Dec Operator
  + ++i : change existing object value and then return a & to the existing object.
  + i++: copy the existing obj, change existing obj value and return the copy (int).

***Big Three***

* Copy constructor, overloaded=, destructor.
* a = b (in normal vars, a copies b data and any change doesn’t affect b data)
* a = b (in pointers, a copies the pointer add, so any change in \*a will affect \*b as well)
* **Shallow** copy happens when we copy (a = b) an object which has pointers.
* Can do **deep copy** via manually written copy constructor.
* What each one do?
  + Copy constructor:
    - constructs an object based on existing one.
    - Used in new object initialization with old object
  + operator=
    - Used in object assignments with old objects.
    - copies one object to another (deep copy).
    - Always check for self assignment (if (this == &rhs)) to avoid it.
* Copy constructor identity : Constructor with **const** **&** to **same class type** param.
* It is always good to avoid copying at most times since its costly (space and time).
* Copy constructor() **= delete**; will disable copying for that class totally (use for safety)
* Good to include big three if the objects have dynamic variables / arrays.
* Why copy constructor param is &
  + Because copy constructors are only called when passed by value (without &) leads to infinite triggers.
* Copy constructor is triggered
  + When Class\_Type a = b (one obj is assigned to another during declaration)
  + When an object is returned from a function (return by value creates copy)
  + When an object is passed to a function as value (pass by value)

***Inheritance***

* Mechanism by which one class (derived class) **inherits** properties (member variables and member functions) of another class (base class).
* Inherits member variables and public member functions
* Will **not inherit** constructors, copy constructors, destructors, overloaded =, and private member functions.
* Private member variables of base class can be accessed inside derived class only using accessor functions, else make the derived class friend in base class for direct access **[IMP]**
* Same functions in both classes
  + **Redefine**: changing definition
  + **Override:** changing definition and virtual
  + **Overloading:** Different signatures
* Derived class **should manually invoke** constructor of the base class to initialize base class’s member variables
  + Can only be done through constructor initialization list
  + During multi level inheritance, happens in asc order (A, B, C)
* Object of derived class can be assigned to base class variables (enables polymorphism).
* Destructors of derived class **auto invoke** destructors of base class in **dsc order**.

***Polymorphism***

1. Using the same instance of something in many forms (to do many actions).
2. In OOPs, using different versions of the same function and choosing the appropriate one based on context.
3. Polymorphism can be done in 3 ways.
   1. Function / Operator Overloading.
   2. Function Overriding via virtual functions.
   3. Templates (it’s actually part of overloading only)
4. Function Overloading examples
   1. Multiple functions with same name but different args.
   2. Default initialization in function parameters.
   3. Templates
5. Function Overriding **[IMP]**
   1. When a derived class object is referenced by a base class pointer, calling an overridden function will call the derived class version if that function is made “virtual”.
   2. If the function is non-virtual, base class version will be called, **despite overridden**.
   3. If the function is made virtual but not overridden in derived class, again base class version will be called.
   4. If the function is made virtual and overridden, but if you want to call the base class version, then use access resolution operator (::) with the base class name.
   5. A class must have virtual destructors but cannot have virtual constructors.
   6. Polymorphism is done through **Late binding** (Dynamic binding / Run-time binding)
   7. Run-time binding 🡪 Content of pointer; Compile-time binding 🡪 type of pointer.
   8. If a function is made virtual in base class, “virtual” keyword is not necessary in the function declaration in derived class (but preferred).
6. Slicing problem
   1. Base class bObj;
   2. Derived class dObj;
   3. bObj = dObj (bObj will lose dObj’s members which is called slicing problem)
7. Varations Allowed [Pet p; Pet\* pptr; Cat c; Cat\* cptr;]
   1. p = c (always allowed)
   2. c = p (allowed if Cat& operator(const Pet&) is overloaded)
   3. pptr = &c (always allowed – polymorphism)
   4. cptr = &p (never)
8. Make desructors virtual.
9. Pure Virtual
   1. When base class contains a function that doesn’t know what to do which can only be implemented in the derived class, make it pure virtual **“= 0”**;
   2. If a base class have pure virtual function, it becomes **abstract class**.
   3. Abstract class can **never instantiate** an object (cos that object’s virtual function method will be not defined)
   4. However you can create base class **pointers** and have them point to derived class objects (this is allowed because, now when that virtual function is called from base class pointer, the derived class version will be called due to polymorphism)

**Summary**

1. Always include copy constructor with = delete; as safety check if the class have pointers.
2. ++, --, = always overload as member functions.
3. Others overload as friend functions.
4. In operator=, check if = happening for same object and also clear the current object before assigning.

**W10: FILE PROCESSING**

**IO STREAMS (SAVITCH CHAPTER 6)** (20/09/20)

* Input can come from a keyboard or file. Output can go to a screen or file.
* IOs are delivered via **pathways** called **streams**.
* Streams are **objects**.

**6.1 STREAMS AND BASIC FILE I/O**

* You are already using files to store your program, but you can use files to store input and output of a program as well.

***Stream***

* A stream is a **flow of** characters or other kind of **data**.
* If the flow is into your program, it’s called **input stream**.
* If the flow is out of your program, it’s called **output stream**.
* **cin** is an input stream connected to **keyboard** and **cout** is an output stream connected to the **screen.**
* You can define your own stream (objects) to get input and save output to files and use it with a variable in a program.
* Input Ex:
  + int the\_number;
  + in\_stream >> the\_number; (in\_stream – stream object we created for file)
* Output Ex:
  + out\_stream << “The number is “ << the\_number << endl*;*
* The following out\_stream object will save the output to a file that we connected to that stream.
* Once the streams are connected to files, IO can be done on them, same way we do to keyboard and screen.

***Why Use Files for I/O?***

* Files provide a way to store data (I&O) permanently.
* We can read and use these files whenever necessary (as opposed to availability only during run-time)
* Provides convenience to use large data of input and output as well.

***File I/O***

* You can read input from a file beginning to end (or you can start it over again - reread) to use in your program, but you cannot backup in the middle and start there.
* You can write output to a file beginning to end but you cannot edit the written output (you can write extra outputs to the file).
* For this to happen, connect the program to the file via stream.
* Stream is an **object variable** that is declared and connected to the file.
* The **file** is the **value** of the stream variable.
* The value of the stream variable can be changed (changes the file) but using special member functions and not = operator.
* The types (classes) for IO stream (obj) variables are **ifstream** and **ofstream** respectively.
* Stream Declaration Ex:
  + **ifstream** in\_stream;
  + **ofstream** out\_stream;
* These stream types are included in the directive **<fstream>**
* These IO stream variables should be **connected** to a file before using it. This is called **opening** a file.
  + Input syntax: in\_stream.open(“file\_name.dat”)*;*
  + Output syntax: out\_stream.open(“file\_name.dat”);
  + The entire path\_name should be included if the file is not in the same directory.
  + Syntax: in\_stream.open(“”);
* Now the program can take input from the file using the >> operator.
* Ex (reads 2 input numbers from the file and save it in the 2 int variables):

int num\_1, num\_2;

in\_stream >> num\_1 >> num\_2;

* The open function of ofstream will **create a new file** if the file doesn’t exist or **discards** the previous contents of the file if it already exists.
* << sends the output to the file.
* Ex: (sends the string and the numbers to the file)
  + out\_stream << “The numbers are “ << num\_1 << “ “ << num\_2;
* The file has two types of names while dealing with it in a program.
* One the original file name called **external file name** and the other is the **stream** **name** which is connected to the file which will serve as the file name inside the program.
* You need to disconnect the file from the stream before your program ends.
* A file is closed with a call to function **close**.
* Ex:
  + in\_stream.close();
  + out\_stream.close();
* **2 reasons to CLOSE** the file
  + Close immediately after usage to avoid corruption of file, when the program ends abruptly.
  + To re-use the output file as an input again in the program.
* A file can be used for both input and output but the method is different (not discussed)
* Any combinations of input and output streams are allowed 🡪 Can take inputs from file and/or keyboard and send outputs to file and/or screen.

***Classes and Objects in Streams***

* cin, cout, in\_stream, out\_stream discussed above are all stream **objects.**
* The type for these objects are called classes.
* Every objects can have different data members but same member functions.
* For objects of 2 different classes that have the same function name (ex: open for ifstream, ofstream), the member function is decided based on the type of the object.

***Programming Tip*** *–* “Check whether a file was opened successfully”.

* There are a number of reasons that can make the file not be opened successfully (ex: file name was wrong).
* It is always good practice to test if the file was opened (connected) successfully.
* To do the test, use **stream.fail()**function which returns a bool value.
* Ex:

if (in\_stream.fail()) {

cout << “Input file opening failed\n”;

exit(1);

}

* **exit(int)** is a pre-defined function from the directive **<cstdlib>** which will end the program when executed.
* int value 1 is given for error cases and 0 for success cases.

***Appending to a File***

* To append the output to a file with contents, use the following method

ofstream fout;

fout.open(“file.txt”, **ios::app**);

* If the file already exists with data, it appends the output, else it creates a new file.
* To include the constant ‘**app**’, you need to include **<iostream>** directive.

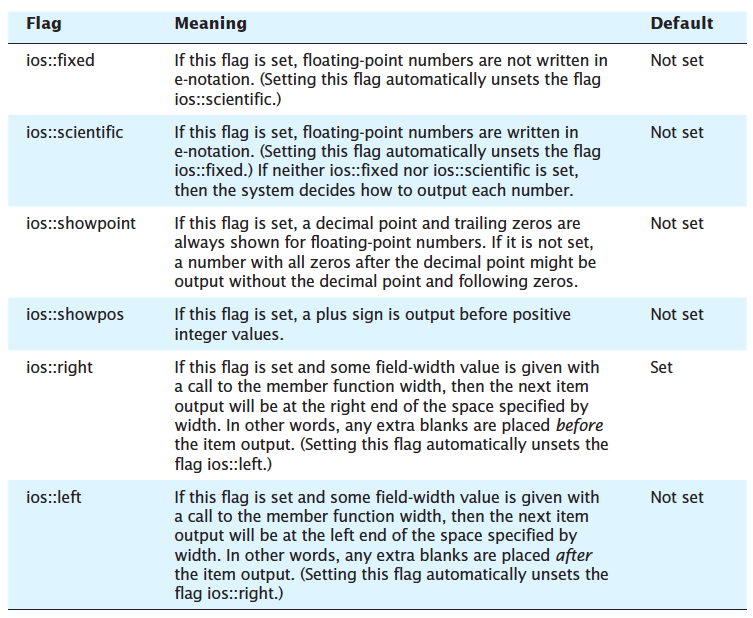
***File Names as Input***

* You can get the filename from the user everytime you need to read from or write to different file, instead of changing the literal in the open function’s argument.
* To achieve that
  + #include <string >
  + string fileName;
  + fin.open(fileName);
* The same above can be done for output stream as well.

**6.2 TOOLS FOR STREAM I/O**

***Formatting Output With Stream Functions***

* The layout of a program’s output is called the **format of the output.**
* Magic formula is one such formmating command used to format the output (decimal pts).
* Formatting commands can be used on any output streams (file or screen).
* Magic Formula
  + **cout.setf(ios::fixed);** 🡪 sets the flag setf to show double type numbers as fixed point notation (the usual way of writing decimal numbers), instead of e-notation.
  + **cout.setf(ios::showpoint);** 🡪 sets the flag setf to always include decimal points.
  + **cout.precision(2);** 🡪 sets the precision (no of decimal points)
  + Some other useful flags



* Another very common formatting function is **width** which produces the output in the given width.
* Syntax:
  + cout.width(4);
  + cout << ”7”;
  + produces the output 7 with 3 spaces before that (total width = 4).
  + If the no of characters is equal or more than width, it will produce all the output with no spaces.
* Any flag that is set can be unset using **cout.unsetf(argument);**

***Manipulators***

* A manipulator is a member function that is called in a different way.
* These are placed after <<
* Ex: cout << endl;
* Manipulator **setw(no)** does the same as **cout.width(no)**.
* Ex: cout << Numbers << setw(4) << 10 << setw(4) << 20 << setw(6) << 30.
* Manipulator **setprecision(no)** does the same as **cout.precision(no).**
* Ex: cout << $ << setprecision(2) << 20.5 << “\t” << $ << 10.5 gives the output $20.50 $10.50
* In order to include these 2 manipulators, you need to include the **<iomanip>** directive.

***Others***

1. **Streams as Arguments to Functions**
   1. Stream parameters must be call by reference.
   2. Stream objects returned should also be by reference.
2. **Checking for the ‘End Of File’**
   1. Since we won’t be knowing how many data are there, it’s always good to check and read the file until all input data are collected.
   2. This can be done using a loop in the following way. **[IMP]**

double next;

**while (fin >> next)** {sum = sum + next;}

* 1. **>> is overloaded** both to get input and to check if data is read, therefore both actions will be done in one fin >> next statement.
  2. The above can also be done for other types of data like char, string or int.

**6.3 CHARACTER I/O**

* All data is input and output as character data.
* But C++ does automatic type conversions from char to whatever type you want.
* In order to make use of actual char and bypass auto typecast C++ gives you some facilities.

***Member Functions Get, Put and PutBack***

* **get** is a member function of every input stream (istream or fstream).
* get reads **one character** of input and includes all the characters – blank spaces, \n, etc.
* Syntax: **cin.get(char\_var);**
* **put** is same as get but is a member of every output stream.
* put writes one character to the output stream.
* Syntax: **cout.put(‘a’);**
* **putback** allows you to do reading until a character is encountered, which will be putback into the input stream.
* Syntax: **cin.putback(next);**
* In the above, if next has a character, it will be putback into the input stream.
* When using get, get rid of the remaining input characters or the **unexpected ‘\n’** or **‘\r’** in input using user-defined **newline()** **[IMP]**

***Making Stream Parameters Versatile***

* A function with parameter type stream can be made to accept both types of streams

istream / ifstream or ostream / ofstream by making it only **istream** or **ostream.**

* User-defined **newline()** function can be redefined to make it accept both the streams
* Ex:
  + Function declaration: **void newline(istream& ins)** [accepts both istream / ifstream]
  + Function call: **newline(cin)** or **newline(fin)**

***EOF Member Function* [DONOT USE THIS]**

* EOF is a member function that comes with input stream ifstream.
* Syntax: **fin.eof();**
* **eof()** is a Boolean member function that returns true if the entire file is read, else false.
* Usage: if (!fin.eof()) or while (!fin.eof()) [Not end of file]
* Note: eof() does not become true until it tries to read one more character beyond the last character in the file which is the **eof marker character**
* Deciding when to use eof vs (while (fin >> next)) **[IMP]**
  + Use **eof** when you’re treating the inputs as text (**characters**) by using get.
  + Use other method when you’re treating the inputs as numeric data.

***Predefined Character Functions***

* Some of the predefined character functions are
  + toupper(char\_var) – Converts to uppercase.
  + tolower(char\_var) – Convers to lowercase.
  + isupper(char\_var) – Checks if uppercase.
  + islower(char\_var) – Checks if lowercase.
  + isalpha(char\_var) – Checks if it’s an alphabet.
  + isdigit(char\_var) – Checks if it’s a digit.
  + isspace(char\_var) – Checks if it’s a space character.
* These functions are in the library with the header file **<cctype>**
* **PITFALL**
  + char and int are used closely here as char values are stored as numbers (num that is associated with the given char) than actual chars.
  + toupper(c) or tolower(c) returns int value, hence used in cout will print that number rather than the actual char.

***Introduction***

* We gonna use files to bring in data into the program and use files to store data output from the program.
* Data cannot be typed in at everytime (since it will be large at most cases).
* There needs to be a way to access data on the file system.
* Input – Reading data into your program.
* Output – Writing data out from your program.
* ostream class doesn’t .open function but can do everything else if you passed ofstream object into a function with ostream base class formal param.
* infile.clear() 🡪 clear the flag if opening has failed once (**recommended** but might not be required in the latest C++ compilers)

***Files and Locks***

* To open a file, you should know it’s “**full name**”.
* A file’s full name consists of it’s **‘name’** and **‘extension’.**
* A file exists in a directory.
* If you don’t specify a directory, it will look for the current dir (as where the program lies).
* Files can also be **locked** sometimes – if it’s in use for writing already.

***Objects***

* C++ has an internal representation (as **object**) for a file.
* The data type of this object varies depending on if it’s an input file or an output file.
* Input files are represented by the type **ifstream.**
* Output files are represented by the type **ofstream.**
* cin and cout are also objects.
* cin’s data type is **istream** and cout’s data type is **ostream.**

***Steps to File Processing***

* You must include the directive **<fstream>**
* You must create stream objects based on the type of activity (I or O)
  + ifstream fin;
  + ofstream fout;
* You must ‘**open’** the connection to the file with the corresponding stream.
  + fin.open(file); / fout.open(file);
  + Opening fails due to many reasons, so it’s important to check it always.
  + fin.fail()
* You can also use constructors to open a file.
  + ifstream fin(“filename.ext”);
  + ofstream fout(“filename.ext”);
* Once opened, the stream object can be used just like how we use cin and cout.
  + fin >>; / fout <<;

***Passing to a Function***

* ifstream and ofstream objects can be **passed to** or **returned from** a function.
* When passing or returning from a function, it must be **always** **done as reference** **(&)**.
  + This is because the object representing the file gets changed everytime we operate on the file (object has a pointer which points to the current position of the file we are reading or writing in, which gets changed everytime we operate (read or write) on them)
* ifstream and ofstream are **derived classes** of the base class **istream** and **ostream** (which are used by cins and couts) – hence they can be the data type for the params in the function, thus enabling us to **pass both types of streams** into it.

***Output***

* Rarely opening a file for output fails
  + If you don’t have write permissions.
  + If your storage doesn’t have enough space.
  + So it is always good to check the opening.
* When opening a file,
  + If it exists, the file will be overwritten (old data lost).
  + If it doesn’t, new file in the name will be created.
* It’s really important to **CLOSE the connection** once done,
  + Realize theres a buffer for output, so the actual writing to the file may not happen until the connection is closed or ofstream obj is destroyed or the program ends.
  + It’s imp if you want to work on that file after writing (because the file will be locked in until the connection is closed)
  + Once the file is used, the file connection should be closed to reuse the same stream object again inside the program.

***Input***

* Input files are more likely to fail when opening.
  + Usually due to wrong file name or file doesn’t exist.
  + It is vital to check that a file is opened successfully – fin.fail() test case.
* ifstream includes a bool member to check validity of file – while(fin).
* If you are going to try to open the file again (after one failure) you must clear the flag of the ifstream object using **fin.clear() [IMP]**

***Reading in Data***

* Many books use EOF but we recommend **against it**, because when you open an empty file, EOF is never reached so that input will have garbage data – so doesn’t work always.
* It’s always better to use **while (fin >> temp)** which will both get the input data and also checks if the read was successful.
* Using the >> operatorc c++ will,
  + Skip the leading whitespace characters (blank, empty, \n)
  + Reads in the ‘valid’ characters.
  + Stops when it reaches trailing whitespace or invalid characters.
* What’s a Valid Character? **[IMP]**
  + What constitutes valid character depends on the data type.
  + **String** – Anything is valid – Stops at 1st occurrence of whitespace char.
  + **Int** – Whole numbers – Stops at 1st occurrence of any invalid char (letter, . etc)
  + **Double –** Real numbers – Stops at 2nd period or whitespace char.
  + **Char** – Anything is valid – Stops at 2nd occurrence of any character.
* Reading a line of data
  + getline(fin, string\_var)
  + Starts reading at the current position in fin and ends at 1st occurrence of \n.
  + Ignores the \n character and returns everything else.
* Ignore Function
  + **streamObj.ignore()** is used to skip characters.
  + We specify how many characters to skip and which position to stop at.
  + Common Usage: **fin.ignore(9999,‘\n’);**
    - Skips upto 9999 characters or the 1st return character.
  + Commonly used to ignore few characters in the middle of a line.
* Moving the position pointer in the file
  + **seekg** moves the position pointer in the file.
  + **positive** value moves it **forward** and **negative** value moves it **backward**.
  + Caution: If EOF is reached and if you try to move the pointer back to the beginning again, you must clear the flags (use fin.clear()) without fail.

**Reading and Writing?**

* Suprisingly it’s very strange to read and edit a file in place.
* Even in text editors you read in completely, edit in memory and save it again in the file.
* So we are not doing reading and editing at the same time in a file.
* Usally, we will read in an entire file, make the changes in the memory and write out the entire file to the disk.

***Appending***

* **ios::app** should be passed as a 2nd param to the fout.open function to append data to an already existing file with contents.
* Syntax: fout.open(filename, ios::app).
* If the file doesn’t exist, it creates a new file and starts from the beginning as usual.

**DATA STRUCTURES**

***Popular Data Structures***

1. Arrays
2. Linked Lists (Lists)
3. Stacks
4. Queues
5. Hash Tables
6. Trees
7. Heaps
8. Graphs

***Which DS to choose?***

1. Depends on the type of data.
2. Cost of operations.
3. Memory consumption.
4. Ease of implementation (not always).

***Linked Lists***

* One of the fundamental data structures in computer science.
* A linked list is made up of **nodes** which has 2 sections, a **data** section and **pointers** to other nodes (links).
* The pointers (links) are connected to form a chain of nodes.
* The list is recorded by **head**, which is a **pointer** to the **first node**.
* The wonderful thing about linked lists is no matter how complicated the data in each node is, you can always draw a picture of the linked list (useful to design).

***Why Linked Lists?***

* **Pros**: Better Insertion / Deletion **O(1)** vs Arrays **O(n)**
* **Cons**: Access time O(n) vs Arrays O(1)
* Linked Lists can be reorganized, merged, etc easily.
* Linked List donot require any overhead (SC = O(1)) apart from pointers (4 bytes).

***PITFALL: Losing Nodes***

* If you don’t create a local pointer and directly set the head of the linked list to the new Node, the old nodes will be lost.
* Losing nodes leads to memory leak which will crash the program, sometimes crash the OS.
* To avoid such lost nodes, a pointer variable (head) should always point to the head node of the linked list.

***Searching a Linked List***

* In order to search the nodes in a linked list, we need to iterate the nodes.
* The only way to iterate through the nodes is to create a local iterator pointer and move through the pointers of the nodes.

***Pointers as Iterators***

* An iterator is a construct that allows you to cycle through the data items stored in a data structure.
* An iterator can be an object of some iterator class or an array index or a pointer.
* General Outline:

NodeType\* iter;

for (iter = head; iter != NULL; iter = iter->link) {do something;}

***Inserting and Removing Nodes Inside a List***

* When a afterMe node pointer is given, new node can be inserted after the afterMe node, despite that node being in the middle or the last.
* Function declaration: void insert(NodePtr afterMe, int data);
* You can always “**squeeze in**” a new node into a linked List, just by adjusting **two pointers** no matter how long the linked list is or where you want to squeeze the node in.
* **Comparision to Array**: Inserting an element into a linked list is more efficient than inserting an element into an array. **[IMP]**
* Removing a node is similar to this, if you know the ‘before’ Node and ‘discard’ Node.

before->link = discard->link;

* However after deleting the node, you need to free up it’s memory

delete discard;

* **PITFALL**: Using = operator with Dynamic Data Structures.
  + If you do this head2 = head1, head2 pointer of some linked list will point to the linked list pointed by head1 and head2 linked list will be lost.
  + Now there is only one linked list and any change made to head1 linked list will have effects over head2 as well.
* Alternatively you can overload the = operator to copy linkedlists (node by node) when it is used.

***Variations on Linked Lists***

* Many data structures can be created using nodes and pointers.
* Some of them are doubly linked list, stacks, queues, trees and graphs.

***Doubly Linked Lists***

* A doubly linked list allows you to move through the list in both directions.
* It has two links inside the node, one to next node and one to previous node.
* Rather than just head pointer, a doubly linked list has pointers to both ends (**prev and next)**

***Notes***

* Going off the end
  + Cross all nodes even the end node.
  + while (tempPtr != null)
  + Used in situations where the list can be empty.
* Stopping at the end
  + Going to the end node and stop inside it.
  + while (temp->next != null)
  + Used when we know the list is not empty and some action has to be done in the end node.
* Recursion in Linked Lists
  + Recursion is often used in linkedlists because a sub list looks the same as a larger list.
  + Ex: recusiveCopy function copies the given linked list into another list.
* What are Linked Lists Useful for?
  + Anywhere we need storage with **constant time insertion** and **no overhead**, but don’t need anything other than **linear access**.
  + Also used to implement other common data structures like **Stacks** and **Queues.**
  + Used in **FAT32** file system to save and access data on a hard drive - Each block contains a pointer to the next block; FAT (head) contains pointer only to the first block – **Cons:** Every random access needs to go through all the blocks till that block.

***Advantages of Linked Lists***

* There are dynamic in nature, which allocates memory as and when required.
* Insertion and Deletion operations can be done in constant time.
* Elements can be inserted anywhere on the list.
* Elements can be accessed in linear time.
* Are used in implementations of Stacks, Queues, Trees and Graphs.

***Types of Linked Lists***

* Singly linked list.
* Doubly linked list.
* Circular linked list.

***Array vs Linked Lists***

* Both are used to store **linear** data (Linear data structure).
* Array elements are stored in **contagious** memory locations, but LL nodes are stored at **random** memory locations connected via links / pointers.
* For static arrays memory is allocated at **compile-time**, but for LLs memory is allocated when new nodes are added (**run-time**).
* Arrays have **random** access O(1) and LLs have **sequential** access O(n).
* Arrays have **linear** insertion and deletion and LLs have **constant** insertion and deletion.

***Working with Templates***

* Functions and Classes can be templated to create a generic version which accepts any data type.
* Syntax:
  + Before each function or class we put **template <class T>**
  + We can also use the following syntax **template <typename T>**
  + T represents the place holder for the data type.
  + Compiler replaces T with the actual data type when it knows it.
  + Should be included infront of every member function definition if a class is templated.
* **Templated Class**
  + When a class is templated, its name includes <T>
  + Creation Example:
    - SomeVal<int> value1;
    - SomeVal<char> value2;
  + Function Declaration Example:
    - void SomeVal<T>::setVal(T newValue);

***Stacks***

* LIFO data structure.
* Stack is a list with insertions and deletions only from one end (top/ head).
* Operations are **push** (insert), **pop** (remove) and **peek** (return top value).
* Retrieves data in the reverse order in which it is stored.
* Implementations :
  + Linked List: head as top and insertions / deletions from head RT: **O(1)**
  + Vector: back, pushback, popback (vector’s methods), **theArray** (var to access the stack), **topOfStack** (index, -1 if stack is empty) RT: **O(1)**
* Other members are
  + isEmpty() – checks if the stack is empty.
  + Constructor
  + Copy Constructor
  + Destructor

***Expressions***

* Only for binary operators.
* **Infix**: <operand> <**operator**> <operand>
  + Precedence: PEMDAS (Exponents R to L, others L to R)
* **Prefix**: <**operator**> <operand> <operand>
* **Postfix:** <operand> <operand> <**operator**>
* Infix to others conversion
  + Do the conversion based on the order of precedence.

***Applications***

* Balancing symbols
  + Compilers use this to check syntax errors.
  + **ALG:** Last unclosed should be closed first.
  + **RT:** O(n)
* Postfix expressions evaluation
  + 4.99 1.06 ∗ 5.99 + 6.99 1.06 ∗ +
  + **ALG**: When a no is seen, push into stack, when an operator is seen, pop 2 nos from stack, do the operation and push no onto the stack.
  + **RT**: O(n)
* Infix to Postfix conversion
  + Standard form is infix which can be converted to postfix using stack.
  + **ALG**
    - Push symbols to stack, output operands.
    - While stack previous symbol has higher / equal precedence than current symbol, pop them and print. Push input symbol onto stack.
    - If (, don’t do anything until ). If ) remove until (.
    - If input empty, pop all pending operators from stack.
* Function Frames
  + Stacking new frames for each function call.
  + Saves local variables (so new function vars doesn’t overwrite) and calling function location (to know where to come back)
  + **ALG**
    - Save current function vars and func address and push onto a stack.
    - Do this repeatedly for each function calls (each data stacked is that function’s vars and mem address – **stack frame**).
    - Once curr function is done, pop the stack to return to that function.

***Queues***

* FIFO data structure.
* Operations at both the ends (front and back).
* Queues are implemented as lists.
* Operations
  + enqueue – insert element at the back of the list.
  + dequeue – remove and return the element from the front of the list.
* Implementations
  + Arrays: **theArray** (var to access the queue), **front**, **back**, **size** (no of elements) RT: **O(1)**
  + Linked Lists: Two pointers to front and back of the queue with items being added to the back and removed from the front RT: **O(1)**
* Applications
  + Process Scheduling (OS)
  + Computer Networks.
  + Simulating wait.

**W10: SEARCH & SORT ALGORITHMS**

**SEARCH ALGORITHMS**

* Linear Search O(n)
* Binary Search O(log n)

***Linear Search***

* Searches for the value one element at a time.
* Running Time = **O(n)** – Since it searches all the element in worst case.
* Implementation – Iteration / Recursive.

***Binary Search***

* Works only on the sorted list.
* Searches for a value by repeatedly dividing the list into halves.
* In every iteration, searches for the value in the left half or right half based on the value being less than mid or greater than mid value.
* Running Time = **O(log n)** – Since the list gets halved during every iteration, so the total number of iterations is log n.
* Implementation – Iteration / Recursive.

**SORT ALGORITHMS**

1. Selection Sort O(n2)
2. Insertion Sort O(n2)
3. Bubble Sort O(n2)
4. Merge Sort O(nlogn)
5. Quick Sort O(nlogn)

* Before choosing the algorithm for sorting, have a basic idea on the data to figure out which algorithm will be the best. XXX

***Selection Sort***

* Selecting the minimum element and swapping it on and on.
* Running Time: **T(n) = θ(n2)**
* Implementation 🡪 Looping find min function and swap.

***Insertion Sort***

* + Takes each item and places at a sorted position towards it’s left, so when it moves from left to right, everything on the left to current position is sorted.
  + Average RT = θ(n2)
  + Best case RT = θ(n)
  + Useful if the entire array is almost sorted (only 1 need to be sorted) - Best case. XXX
  + Not useful in most of the cases.

***Merge Sort***

* Dividing and sorting each half and then merging them into one sorted sequence.
* Three step process using recursion.
* Step 1: Recursively sort the 1st half of the arr.
* Step 2: Recursively sort the 2nd half of the arr.
* Step 3: Merge the 2 halves into one sorted sequence.
* Implementation
  + Base case: low == high
  + Recursive step: sort 1st half, sort 2nd half, merge(arr, lLow, lHigh, rightH);
* Running Time: T(n) = **θ(nlogn)**
* Space Complexity = O(n)
* Caveat: We need to create one more array to save the merged array (in-place sorting is not possible) – Requires twice the amount of memory.

***Quick Sort***

* + Worst case RT = θ(n2)
  + Average case RT = θ(nlogn)
  + Best case RT = θ(nlogn)
  + Worst case happens when the pivot element is the min el or max el in the list.
  + Do median of 3 partitioning to randomize the pivot selection.
  + Benefit: Doesn’t use extra memory (no extra array required).
  + Built in sorting functions (from <algorithm> library) use quicksort.
  + Generally considered the **fastest sorting algorithm** because it has the best performance in the avg case which occurs for most inputs.
  + Space Complexity = O(nlogn)

***Some Important Info On Arrays***

1. You should understand these algorithms in such a way that you can reproduce it yourself in quite a while (or be able to figure out the modifications in them) – memorizing is not required and not expected.
2. Quicksort is the most preferred sorting algorithm and preferred over mergesort ([link](https://www.geeksforgeeks.org/quicksort-better-mergesort/))
   1. **Worst Case:** Its worst TC O(n2) can be avoided by using a randomized version of quicksort which is choosing the right pivot element, which improvises it’s performance and makes it as efficient as merge sort O(nlogn).
   2. If randomized pivot is chosen, practically on most of the cases the running time will be O(n\*logn).
   3. **Auxillary Space:** It’s better than merge sort because it doesn’t use extra space like mergesort which uses auxillary array while merging the divided elements.
   4. Allocating and Deallocating this extra array consumes some time which can be avoided in quicksort, thus making both space and time complexity better than merge sort (on most of the practical use cases).
   5. **Cache Friendly:** It is cache friendly because it has good locality of reference when used with arrays.
3. Where Quicksort vs Mergesort is used?
   1. Quicksort – Arrays.
   2. Mergesort – Linked lists and large lists of data from external storage.
4. Merge sort is better for linked lists. Quick sort is better for arrays ([link](https://www.geeksforgeeks.org/why-quick-sort-preferred-for-arrays-and-merge-sort-for-linked-lists/))
   1. Merge sort is better for LL because we can insert nodes in the middle in LL with constant space and time, hence the merge operation of merge sort can be implemented without extra space O(1) needed for LL.
   2. Arrays can randomly access elements in constant time which is essential for quick sort because they need a lot of random access, for example to randomly choose the pivot element.

**TREES DATA STRUCTURE**

***Trees***

* Arrays, Linked Lists, Stacks, Queues are **linear** **data** **structures**. It has a logical start and a logical end. Each element has a next and previous element.
* Trees are data structures used to represent naturally **hierarchical** data.
* Often used for **searching**.
* A tree is a collection of nodes to represent hierarchical structure.
* The top most node is called **root node**.
* Each node contains data and references / pointers to other nodes (childs).
* Some terminologies – **parent**, **child**, **siblings**, **grandparent**, **grandchild**, **leaf** nodes.
* Root node doesn’t have any parent.
* A node with no child is called **leaf node**.
* Tree node traversal is **one** **directional**.
* **Ancestor** node: The node that we came from (previous node) -- A is ancestor of B.
* **Descendent** node: The node that we came to (reached node) -- B is descendent of A.
* **Cousin** nodes: Node that doesn’t have same parent but have same grandparent.

***Tree Properties***

* Trees are called recursive data structure.
* Each child node itself looks like a sub-tree (recursive structure).
* A tree with **‘n’** nodes will have **‘n – 1’** edges.
* **Depth of a node**
  + No of edges in the path from root node to that node.
  + Depth of **root** **node = 0.**
* **Height of a node**
  + No of edges in the longest path from that node to a leaf node.
  + Height of **leaf** **nodes = 0.**
  + Height of a tree with **1 node = 0.**
  + Height of an empty tree with **no nodes = – 1**
* Heigh of a tree = Height of root node or max depth of the tree.
* Height and Depth of a node can be same for some nodes (central level node in the tree).
* Most common way to create trees is **dynamically** created nodes with **links** / pointers.
* Binary Tree
  + Most common and famous tree is **binary tree.**
  + In BT, all the nodes will have **at most 2** children.
  + A BT node has **3 members** – Data, &left child, &right child.

***Tree Applications***

* Store naturally **hierarchical** data -- Ex: File system.
* Organize data for **quick** search, insertion and deletion **O(Logn)** -- Ex: BST.
* Compiler design -- Expression trees.
* Trie -- Dictionary for spell checks.
* Network routing algorithm.

***Tree Storage – Nodes***

* For trees with unlimited children, the child pointers should either be arrays or linked lists.
* Parent – Multichild -- Parent node has an array of child nodes (**vectors**).
* Parent – Child – Sibling -- Parent node has a pointer to favourite child and favourite child forms a link with its siblings (**linked list**).
* If we have a limited no of children (like 2 for BT) we can then setup that no of pointers directly (**most common trees**)

***Binary Tree***

* Each node can have **0** or **1** or **2** children.
* **Strict / Proper** BT: Each node should have 0 or 2 children.
* **Level** is the depth of a node at each level (0,1,2,3,…)
* **Complete** BT: All levels **except last** are completely filled and all nodes are as **left** as possible.
* **Perfect** BT: All levels of the tree are complete.
* Maximum level of a tree = Height of a tree.
* No of levels = Height of a tree + 1 (including 0 level)

*Formulas*

* The maximum no of nodes at level **i = 2i**
* Maximum no of nodes in a BT with height h**, n =** 20 + 21 … + 2h = **2h+1 – 1 = 2no of levels – 1**
* Height of perfect BT with n nodes, **h = log2(n+1) – 1**
* Minimum height possible with BT (when it’s perfect BT) **h =** **⌊log2 n⌋**
* Maximum height possible with BT **h = n – 1**

*BT Continuation*

* All above formulas are helpful to analyse cost of various operations on BT.
* **Cost** of operations is **proportional** to the **height** of the tree.
* Cost of a tree will be reduced if the **tree is balanced.**
* In general a tree is **balanced** if the max depth / height of the tree is of the order (**logn**).
* **Balanced Binary Tree (AVL)**
  + For each node, difference between the **height** of left and right sub-tree is not more than **1 (k)**
  + Difference **k =** **|hleft – hright|**
  + For leaf nodes, k = |–1 – (– 1)| **= 0**
  + For all nodes in a perfect BT, **k = 0**
* We can implement BT using
  + Nodes (like linked lists)
  + Arrays
    - Used only for complete BTs.
    - Nodes are indexed from left to right in each level.
    - for a node at index i,
      * left child index = 2i + 1
      * right child index = 2i + 2
    - **Array** implementation is used for **heaps** (special kind of BT)

***Binary Search Tree* [IMP]**

* Special kind of BT where data is efficiently organized for quick search and quick update.
* BST is a BT in which values are sorted in the order **left <= root < right**.
* Very common not to have duplicate (=) values.
* Best and Average RT for BST for all the three operations is **O(log n).**
* Worst case RT for BST is **O(n)** when the BST is unbalanced (only 1 node at each level).
* To achieve the best RT always, the BST should be **balanced**.
* Uses
  + Useful when you need all the opertions (search, insert and delete) to be done efficiently.
  + Very efficient for in-order storage of any items which can be compared using a overloaded < operator.
* BST Node Impl
  + Variables: data, parentPtr (optional), leftPtr, rightPtr.
  + Functions: constructors, friend class BST, getSize().

***BST Tree Traversals***

1. Tree Traversal: Process of visiting (reading / processing) each node in the tree exactly once, in some order.
2. **In Order**
   1. Process the left nodes first recursively, then this node, then right nodes recursively.
   2. **L-RN-R**
   3. Visually processes all the left nodes first in a tree.
   4. In BST, this results in **‘normal’** order.
   5. Depth-First Traversal
   6. Use:
      1. Gets the nodes in ascending order (sorted) in a BST.
      2. Can get the nodes in descending order by reversing in-order traversal.
3. **Pre Order**
   1. Process this node first, then left nodes recursively, then right nodes recursively.
   2. **RN-L-R**
   3. Depth-First Traversal || Polish Notation
   4. Use:
      1. Used to create a copy of the tree.
      2. Also used to get prefix expression of an expression tree.
4. **Post Order**
   1. Process left nodes recursively, right nodes recursively, then this node.
   2. **L-R-RN**
   3. Depth-First Traversal || Reverse-Polish Notation
   4. Use:
      1. Used to delete the nodes of a tree (complete tree).
      2. Also used to get postfix expression of an expression tree.
5. **Level Order**
   1. Process the nodes based on **depth,** the nodes closest to root first, then successively lower order nodes.
   2. Breadth-First Traversal
   3. (Visually) process all the nodes at each level first.
   4. Use:
      1. Useful while doing operations on large trees where nodes which are important gets operated upon first.
6. Impl: Check if the given node is not null and recursively process it according to the traversal.
7. TC = **O(n)**; SC = **O(n)** for all the above traversals.

**Common Operations in BST**

* insert(node, data);
* contains(node, target);
* remove(node);
* inOrderPrint(node);
* preOrderPrint(node);
* postOrderPrint(node);
* levelOrderPrint(node);
* findMin(node);
* findMax(node);
* findHeight(node);
* isBST(node);
* getSuccessor(node, data);
* makeEmpty(node)
* clone(node)

***Node Insertion***

* Iterative
  + Params: theData.
  + If root is empty, insert directly.
  + If root is not empty use 2 pointers (temp and prev), temp to iterate it to null and prev to access the previous to null and create a node as left or right child to prev based on the value.
* Recursive
  + if reached null 🡪 insert the node.
  + else if data < this node data 🡪 go left.
  + else if data > this node data 🡪 go right.
  + else 🡪 do nothing (duplicate).

***Node Removal***

* Leaf Node Removal
  + Update the parent’s corresponding ptr (left, right or root) to null.
  + Remove the node.
* Node with one child
  + Promote the child’s data above to the node to be deleted.
  + Make the child as to be deleted.
  + Delete the child.
* Node with two children
  + Choose the max child of left sub-tree or min child of right sub-tree – Our decision but min of right sub tree is easy)
  + Loop through the nodes and find the min or max node.
  + Promote that node’s data.
  + Delete that selected node duplicate copy by calling remove as recursion.

***BT – Expression Tree***

* The leaves of an expression tree are **operands.**
* Other nodes contain operators.
* If binary operators 🡪 binary tree.
* Evaluation
  + Infix expression (regular exp) can be produced by in-order traversal.
  + Postfix expression can be produced by post-order traversal.
  + Prefix expression (less useful) can be produced by pre-order traversal.
* Constructing an expression tree – Postfix expression to an expression tree.
  + Create a stack and read each symbol at a time.
  + If the symbol is operand -- New one node tree is created with that operand and a pointer to that tree is pushed onto the stack.
  + If the symbol is operator -- New tree is created with the operator as the root and two operands popped from the stack as childs to that root and a pointer to this tree is pushed onto the stack.
  + This is done repeatedly until all symbols are read.

***Balanced BST***

* BST fails when insertions are already in order (inserting a sorted data) -- Makes it a tree with height **n – 1** (1 node at each level) which makes it a linked list with search time **O(n)**.
* To guarantee **O(log n)** search time, we need to **balance** the tree during insertions and removal.
* Due to these balancing, insertions and removal will take a **bit longer** (but still closer to O(logn) 🡪 2\*logn).
* Two popular balanced BSTs are available
  + AVL -- Easy to understand, but performance is not all that great **(2\*O(logn))**
  + Red-Black Trees -- Hard to understand, but performance is fabulous.
  + STL’s **sets and map** are implemented using Red-Black trees.

***AVL Trees***

* It’s a BST with balance condition.
* Named after its creaters and easier to understand.
* It ensures the depth of the tree to be **O(Logn)** and thus its operations.
* Each node records its own **height**.
* Finding height formula (here) : **max(hleft , hright) + 1**
* The minimum no of nodes in an AVL tree of height h S(h) = **S(h – 1) + S(h – 2) + 1**
* Condition: The difference of the height of left and right sub-tree of every node should not differ by more than **1**.
* If the heights differ by more than 1, a **rotation** is necessary to **rebalance** the tree.
* **Balance information** should be updated **only** to the **nodes in the path** from the insertion node to the root (only these nodes will have their balance altered due to an insertion).
* While balance information update, a node (**α**) might violate the AVL condition which has to be rebalanced.
* A violation might occur in in **four** cases,
  + An insertion into the left sub-tree of the left child of **α**
  + An insertion into the right sub-tree of the left child of **α.**
  + An insertion into the left sub-tree of the right child of **α**.
  + An insertion into the right sub-tree of the right child of **α.**
* Case 1-4 and 2-3 are themselves mirror images respectively – only rotation direction varies.
* Outside (1-4) insertion can be fixed by **single** rotation.
* Inside (2-3) insertion can be fixed by **double** rotation.

***Rotation Solutions***

* Given a grandparent, parent, child situation which is unbalanced,
  + Single clockwise rotation – Make the parent the new grandparent. Now grandparent and child become the new children.
  + Works if the nodes with higher heights are outwards (left-left or right-right).
  + But if there is a right child for left parent, then while making left parent the new grandparent, the previous grandparent’s left child will be the previous parents right child. (Think visually)
  + Rotations: **α** and **α’s** child (visualize)
  + If left – left : clockwise.
  + If right – right : counter-clockwise.
* If the node with higher heights are inwards (left – right / right – left) then do Double rotation.
  + Rotations: **α’s** child (k3) and grandchild(k2), then **α**(k1)and **α’s** new child(k2).
  + Perform two single rotations (visualize with the aid of k1 – k2 – k3)
    - If left – right : 1 counter-clock, 1 clock.
    - If right – left : 1 clock, 1 counter-clock.

***Red-Black Trees***

* One caveat with AVL is **calculating heights** whenever insertions, deletion or rebalancing happens which takes **O(logn)** so total TC is **2\*Olog(n)**
* RBT avoids it by having **4 laws,** (Purpose of these laws is to determine when a rotation is gonna be necessary)
  + All nodes are coloured **red or black** (null is black)
  + The **root** is always **black**.
  + A **red** **node** cannot have a **red child.**
  + Every path from **node to null** must contain the same number of **black** nodes.
* Whenever we insert a new node, we colour it red node (not law but implementation factor)

***Insertions Strategy***

* Insert a node Z and colour it **red** (else property 4 will be broken).
* If the **parent** is **black**, we are done, else we need to fix the violations.
* Recoloring and rotations can fix the violations.
* Every insert to **red parent** can create 4 scenarios.
  + **Z = root**: color Z black.
  + **Z uncle = red**
    - Change color of parent and uncle as black.
    - Change color of grand parent as red.
    - Make grand parent as new Z and repeat.
  + **Z uncle = black** (line – outwards) – 2 cases
    - Rotate grand parent to Z’s opposite direction.
    - Recolor parent and grand parent.
  + **Z uncle = black** (triangle – inwards) – 2 cases
    - Rotate parent to Z’s opposite direction.
    - Make parent as the new Z.
    - Do the outward line rotation.

**IMPORTANT NOTES**

1. Always include using namespace std **inside the function definition** where it is needed.
   1. Donot include that at the top of the program just like that.
   2. This will eliminate the purpose of namespace which is to use common names from different namespaces.
2. Return with **error code 11** – **Segmentation fault**
   1. Occurs when program tries to access a memory location which is not allowed to access.
   2. Access of members of a **nullptr**.
3. Tail recursions (recursion at last line) can be easily replaced by while loops.
4. Amortozied constant time: