**Part – 1 (A few Obvious Things)**

* Constructor sets garbage values by default
* Constructor is called only once for an object (during its creation)
* When we create our own constructor, the inbuilt one is not valid anymore
* We can have multiple constructors and pass parameters accordingly to call different constructors
* ‘this’ keyword serves as a pointer to the current object, (at times it is implicitly implied, but when we have identical variable names, ‘this’ must be used.)
* **Inbuilt constructors –**

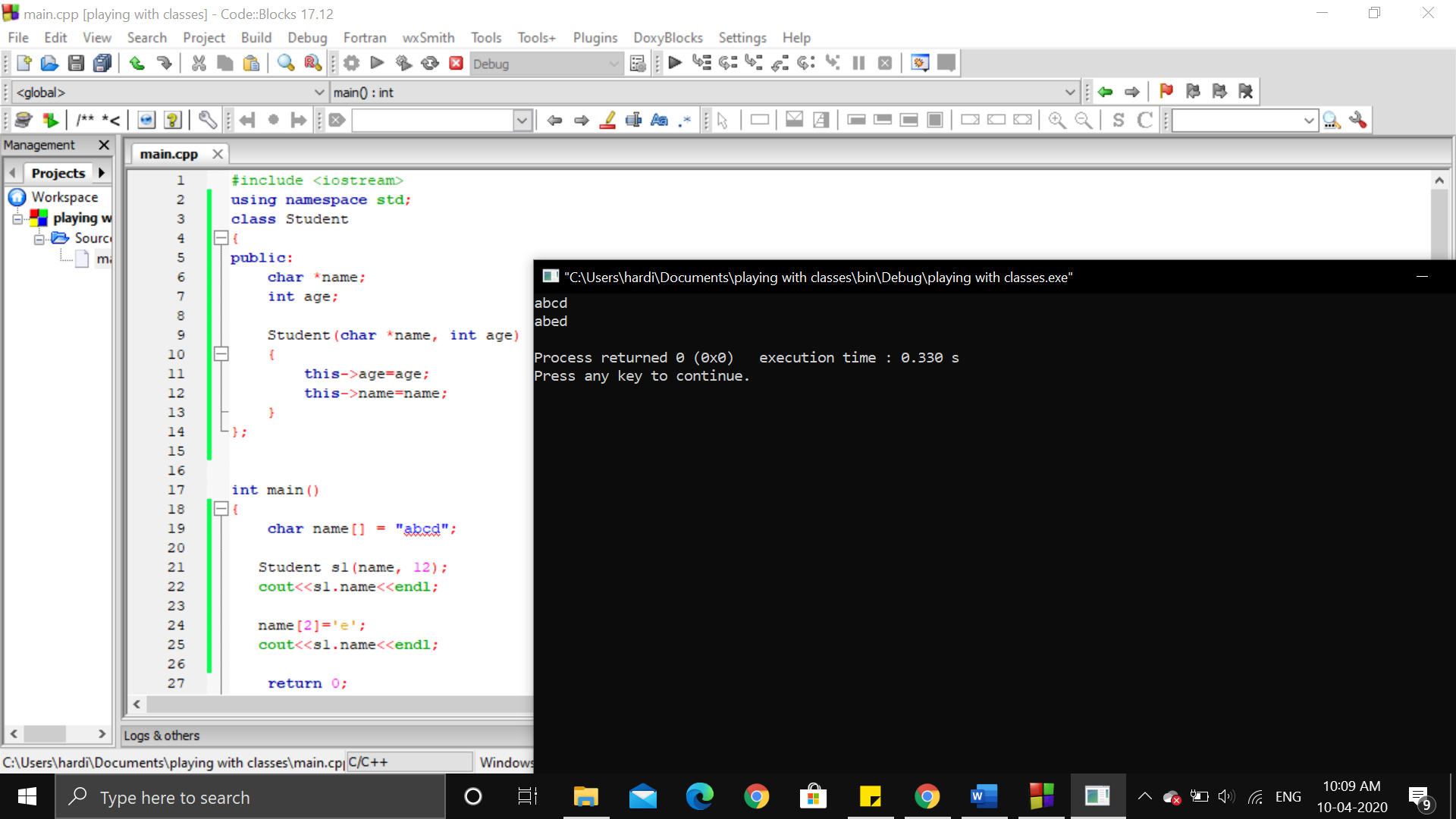
1. **Default Constructor**
2. **Copy Constructor –** usage goes like -> Student s2(s1); //s1 is already an object.

**NOTE THE FOLLOWING -**> Student s1=s2; (it is a copy constructor and NOT an assignment operator because s1 is being created!!!). (Assume s2 is already created). It is internally understood as -> **Student s1(s2).**

1. **Copy Assignment operator (NOT A CONSTRUCTOR, BUT AN OPERATOR) –** usage goes like -> // working is same as copy constructor, the difference is just that copy assignment is used when both the objects have already been created.
2. **Destructor –** No return type, No input arguments, Same name as our class, Helps in de-allocation of memory, usage goes like -> ~Student(){}. Called only once for an object.

**Part – 2(Important Portion)**

* If we have a pointer as a data member in our class, for example – a character pointer, things work a bit differently and we need to be a bit careful.
* To illustrate the above, consider the following screenshot –



* the pointer remains same but the content ahead changes, thereby producing a change. This is called **shallow-copy,** when we just pass the reference to initial index (arises in case of arrays)
* **Observation –** If we use strings in case of character arrays, such a situation is avoided (maybe because deep-copy takes place)
* **Deep-Copy ->** we copy the entire thing, instead of just the initial reference. For example, we change the constructor itself, inside it, we create a new char array and copy the whole character array into it.

**Example ->** this->name = new char[strlen(name)+1]; strcpy(this->name, name); //1 extra length for null character

* **ALWAYS REMEMBER THIS!! -> INBUILT *copy constructor uses SHALLOW COPY***
* To create a copy constructor with deep-copy feature, do exactly what we did above.

Example -> this->name = new char[strlen(name)+1]; strcpy(this->name, name);

* However, there are a few things we have to take care of when we create our own copy constructor –

1. When we pass **Student s** as parameter, it again calls the C++ copy constructor (the one created by us), but it will keep calling itself again and again and again (Copy Constructor : Infinity Loop). Now, we have to calling the copy constructor like this. For this, **we pass the object by reference -> “Student &s”**
2. Another issue arises now, since **“s”** is passed by reference, it can be changed inside the copy constructor, therefore to avoid such a scenario, we also declare it as a constant parameter. Therefore we write it as -> “**Student const &s”.** This works perfectly now!

**About Initialization Lists**

* These basically help us in initializing constant data members.
* Example – Student(int r) : roll { roll = r;} // data member -> const int roll;
* Basically, you **put a colon and then the data member which is declared as constant**
* After this, the constant data member can be initialized through the use of constructor (only).
* We can also initialize non constant variables using initialization list

**Example ->** *Student(int r, int age) : roll(r), age(age), x(this->age){} //body is empty. x is declared as int &x, note that x is initialized with the value of data member age and not parameter age (****hence we need this keyword)****! While data member age is initialized with the parameter age (****and we don’t need the this keyword)***

**About Constant Functions**

* Note that constant objects can call only constant functions (the ones we have declared as constant).
* **Required Syntax example ->** int get() const {} //just add const keyword after function name
* **A constant function doesn’t make changes in the object it is called by (**this)!

**About Things of Static**

* The key idea here is that static members are common across all objects, that is, they are a common data member accessible to all objects.
* **For example –** Total number of students – this data member should be a static one.
* **To access static properties, we need to use** SCOPE RESOLUTION OP.
* **Usage ->** cout << **Student :: totalStudents** <<endl;
* **Initialization** of static properties takes place outside the class
* **Example ->** int Student :: totalStudents = 0; //outside the class definition.
* **Note that,** if we access static properties using object as well, it still gives us an answer (whether we read or modify it), but it is logically incorrect and we should refrain from such practices.
* **Usage idea ->** for a property like totalStudents, we can set it on increment inside the constructors, to make things convenient.
* **IDEA OF STATIC FUNCTION** -> just write **“static”** before function name
* **Way to access ->** Student :: getTotalStudents();
* **Remember to use scope resolution for static properties.**
* **Static functions can access only static functions/data members/ properties.**
* **Static functions don’t have** ‘this’ keyword, because they are simply independent of objects.

**About Operator Overloading**

* **Basic Syntax ->** Fraction **operator+**(Fraction const &f2){}
* **Examples ->** DMAS operators, ++, --, ==, +=
* **Pre increment -> operator++**
* **See the overloading of ++ operator (pre increment) in video, especially when we pass by Reference (as Fraction&) and (return \*this) in order to make the buffer point to f1 itself and not create a copy. Basically, this REFERENCE IDEA,** comes into play during NESTING

**Abstraction and Encapsulation**

* **Encapsulation –** ‘clubbing together’, clubbing data and functions of an entity, i.e., make a class (all data members and functions are said to be ‘encapsulated’)
* **Abstraction – ‘**to hide (the unnecessary details)’. Example – operating TV with remote, we just press buttons on the surface, but do we know what exactly goes on inside the TV and remote? So such details are hidden and this is called **Abstraction. Example – C++ inbuilt sort is also an example of abstraction, STL etc are all having hidden implementation which we don’t care about!**
* **Now,** to achieve **abstraction** in our very own class, we use **ACCESS MODIFIER.**
* **Reasons to use abstraction –** 1. To do internal changes that don’t concern the user, 2. To avoid errors, that is user making unnecessary mistakes.

**Inheritance**

* To inherit properties/characteristics.
* Promotes code re-usability.
* **ACCESS MODIFIER -> Private > Protected > Public (**protected is the middle ground, the property is accessible to only the child classes)
* **Parent = Base class and, Child = Derived Class**
* **Syntax -> be careful here!**
* **Example -> class Car : public Vehicle{}; //default is private**
* If this access modifier is public, then protected of parent is inherited as protected and public as public
* If this access modifier is protected, then both protected and public of parent are inherited as protected.
* If this access modifier is private, then both protected and public of parent are inherited as private members. (**DO note that private members of parent class can never be inherited).**
* **Memory wise working –** If a class has inherited something, then when we create an object of the child class, the constructor for parent class is also called by default. The order of constructor call is -> if the inheritance order is -> A is parent of B, then while creating B, first the constructor of A is called, then constructor of B is called. **The destructors are called in the opposite order as the constructors.**
* **By default,** even if there are multiple constructors of the parent class, the default one is called unless stated otherwise in the code. **BUT NOTE HERE THAT** if we remove the default constructor by adding our own constructor, we must make a call to the appropriate constructor, since the default one can no more be used.
* **Types of INHERITANCE –**

1. **Single Inheritance –** One parent and one child class
2. **Multi-Level Inheritance –** Multiple levels are present. (Many parents, many children)
3. **Hierarchical Inheritance –** One parent, many children classes
4. **Multiple Inheritance –** Many parents, one child class.

**Example -** Syntax -> *class TA : public Teacher, public Student* (Teacher constructor is called first, then student, then TA)

**Note –** If the parent classes have same data members or any of the functions, **SCOPE RESOLUTION** is used to specify which class we are talking about.

**Example –** a.Student :: print(); (calling Student class’s print function here).

**But note that if our current class (like TA) also has the same data member / function (as parents), then when we access that data member/function, and it is present in the current class, we need not use any scope resolution and that member/function is accessed normally.** Example -> a.print(); //if TA class has print function, it is called normally.

1. **Hybrid Inheritance –** Combination of above types to make a new structure.

**About ‘virtual’ keyword –** When added before a class during inheritance (like -> class Truck : virtual public Vehicle), we basically ensure that only the bottommost class calls the constructor for the class declared as virtual, and that too, only once. For example-

Vehicle

Car

Truck

Bus

Consider the above inheritance diagram. When we add the ‘virtual’ keyword while creating Car and Truck classes, the Bus class calls the constructor for Vehicle class directly for itself, and that too, only once, while, Car and Truck don’t call the constructor for vehicle class for themselves. Car and Truck have a pointer kind of thing to vehicle class while Bus class has an actual object.

Also note, that after a class is declared virtual like here, when we need to use parametrized constructor for Vehicle class, we need to pass the parameter for the same inside Bus class as well (along with Car and Truck). If it weren’t a virtual class, parameter had to be passed through Car and Truck only. Again note that in case of ‘virtual’ keyword, only the bottommost class calls the base class (**when we have a DIAMOND structure)**

This phenomenon is also called **Virtual Inheritance.**

**About Polymorphism**

* It means many forms. A set of code behaves differently in different scenarios.
* Two types – Compile Time and Run Time
* **Compile Time –** Two ways-

1. **Use Function Overloading/ Operator Overloading** -(same name, different functionalities, like parameters and ~~return type~~ (**THIS IS WRONG! WE CAN’T DIFFERENTIATE FUNCTIONS ON THE BASIS OF RETURN TYPE)** for functions). Observe how everything is determined during compile time.
2. **Method / Function Overriding –** like parent class and child class have a function with same name, so, the function in child class overrides the function in parent class. To access function of parent class, we use SCOPE RESOLUTION.

**Note that** *-> Parent class pointer can point to Child class object (vice versa not allowed). Also note that, using such a pointer, we can access only the properties which are present in parent class*

* **Run Time –** to achieve Run Time polymorphism, we use ‘virtual’ keyword (this time for functions). **(Example -> parent and child have same function, function in parent is declared ‘virtual’, we have a pointer of parent class pointing to an object of child class).** Now, the decision is taken at runtime and the working is same as above cases. When we call print function, then if child class has it, we call that one itself, and if child doesn’t have it, parent class’s function is called. **But if child has it, and parent doesn’t, then we get an error! Example -> calculating salary of different kinds of employees. (see video)**

**About Virtual Functions and Abstract Classes**

* **Pure Virtual Functions ->** Example -> virtual void print() = 0; //basically =0 .
* **Any** class which has **even one pure virtual function,** then that class becomes **an abstract / incomplete class.** Basically, we **cannot create an object of such a class.**
* If a child class inherits from an abstract class, we have two choices –

1. Either complete the definition of pure virtual functions of the parent class
2. Stay as an abstract class itself.

* **Example –** Again, same as above, employee example
* **Basically, whenever and wherever you think or want to force** the child class to define a function, create its parent class as a class made of pure virtual functions ( basically make the parent class an abstract class ).

**About Friendship in OOPS**

* ‘friend’ functions allow the access of private and protected properties.
* **Example ->** friend void Bus :: print(); //do this inside the class whose private property you want to access.
* **C++ goes line by line.** We need to define things above the lines where they are used (unlike JAVA)
* **The sequence goes like -**> The class which needs to access private properties, the class which shares its private properties, function definition of first class which needed the private property (note the function is defined using SCOPE RESOLUTION)
* **To make** a GLOBAL function as a friend of a class, the syntax goes like -> friend void test(); //inside the class whose private property you want to access.
* **Again in the above case,** the order goes like -> global function name, define it as a friend in the required class, add global function definition.
* **friends of a class don’t have access to the ‘this’ keyword**
* friend keyword can be added under any access modifier of a class, it doesn’t matter.
* To make a whole class as a friend of a class, do this -> friend class Bus; -> now Bus class is a friend of our current class. (one way friendship)
* We can also define a two way friendship ( a good thing to do ).