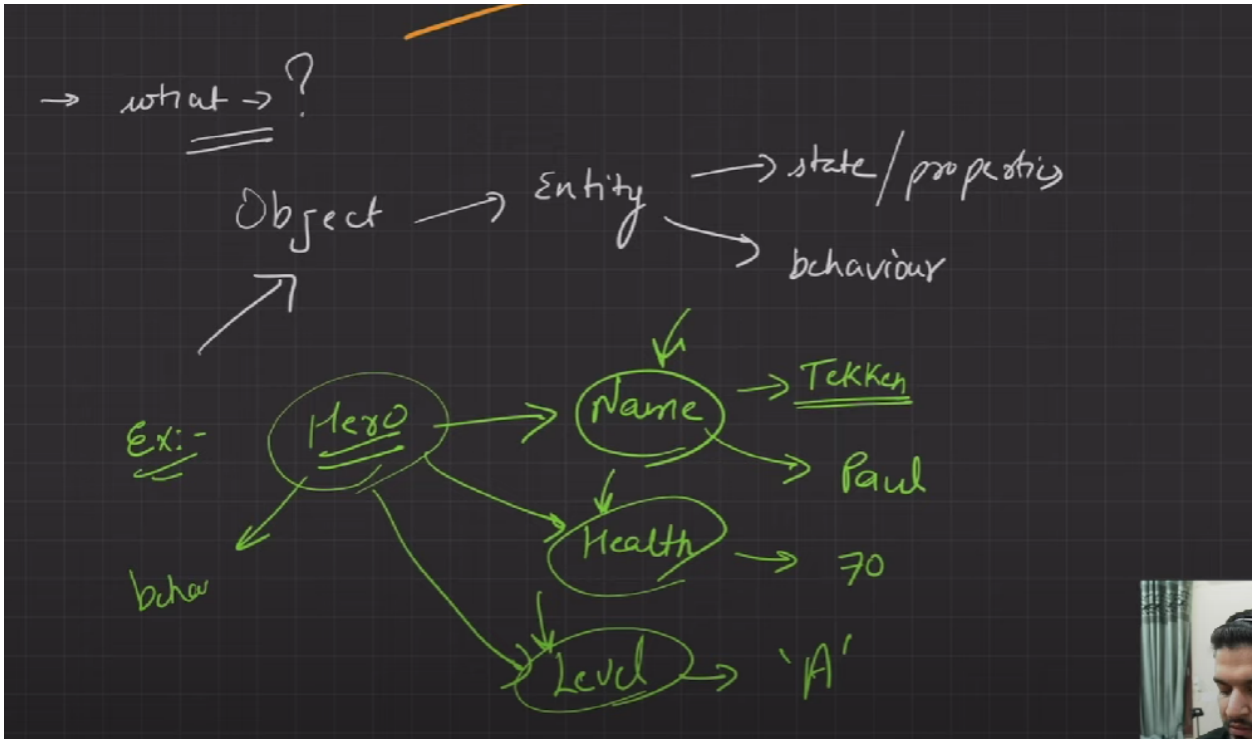
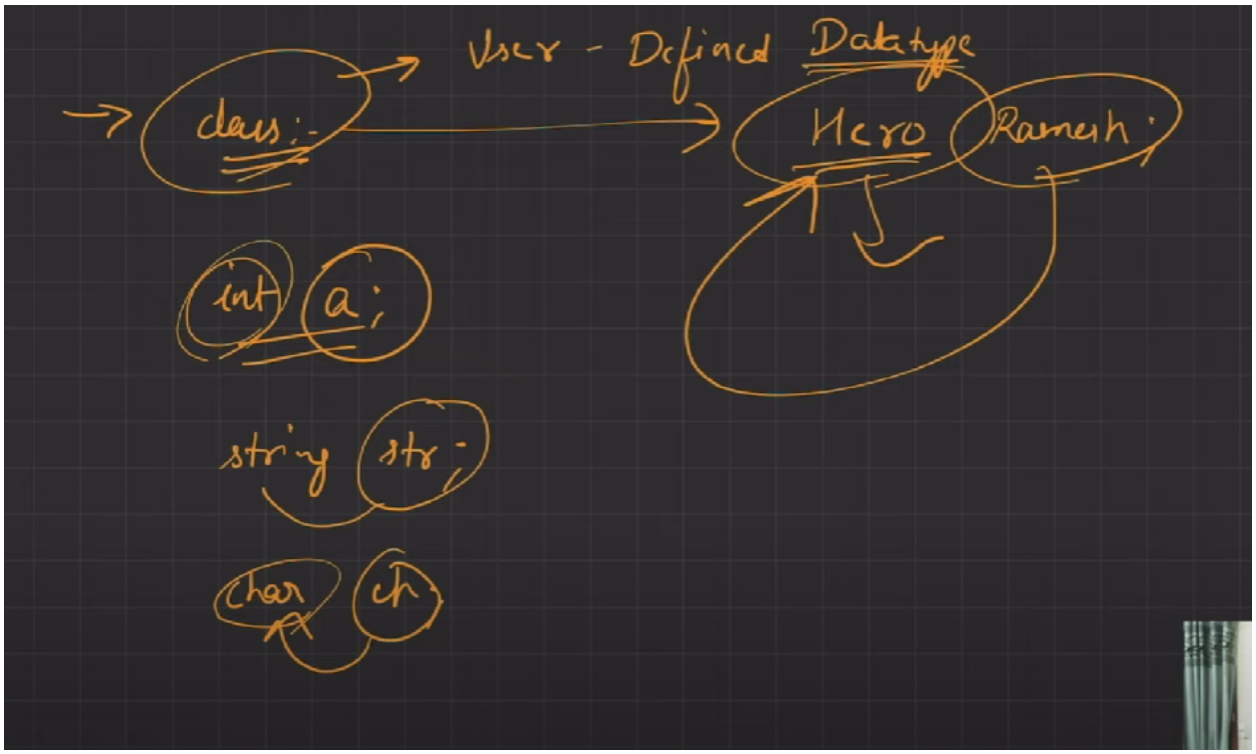
OOPs

Pura Oops ka concept revolves around *objects*.

What is an Object then?
Object is an Entity -> Any *property* , *state* or *Behaviour* of anything



What is Class?
Class is nothing but a *user-defined function or a data-type* .
Some predefined data types are like int , string , char, bool etc..But this is user-defined.



Implementation of Class :

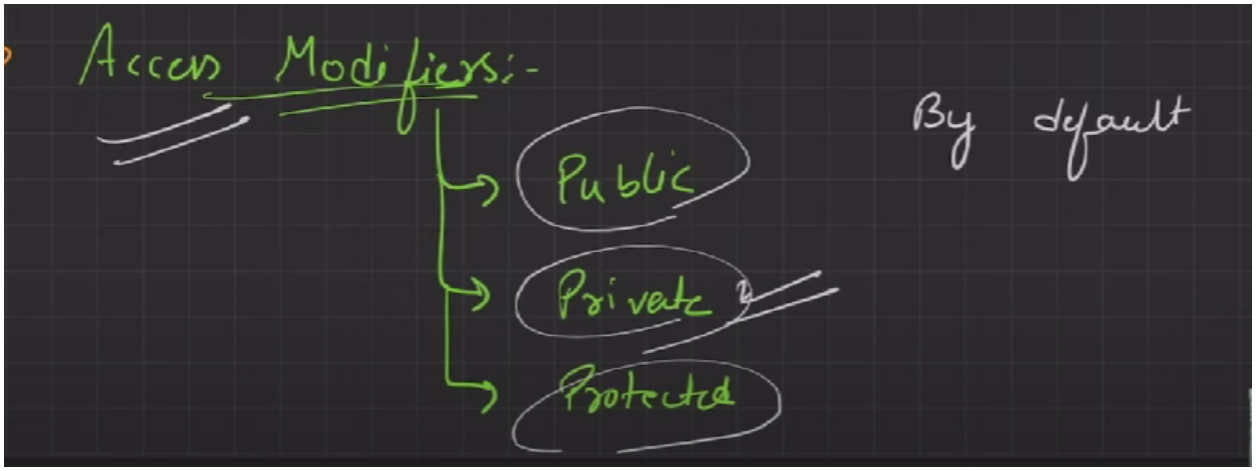
Class is like a *Template or BluePrint*.
Object is an instance of a class.

Empty Class -> Size -> Always 1 Byte -> It is assigned just for the presence of Class.

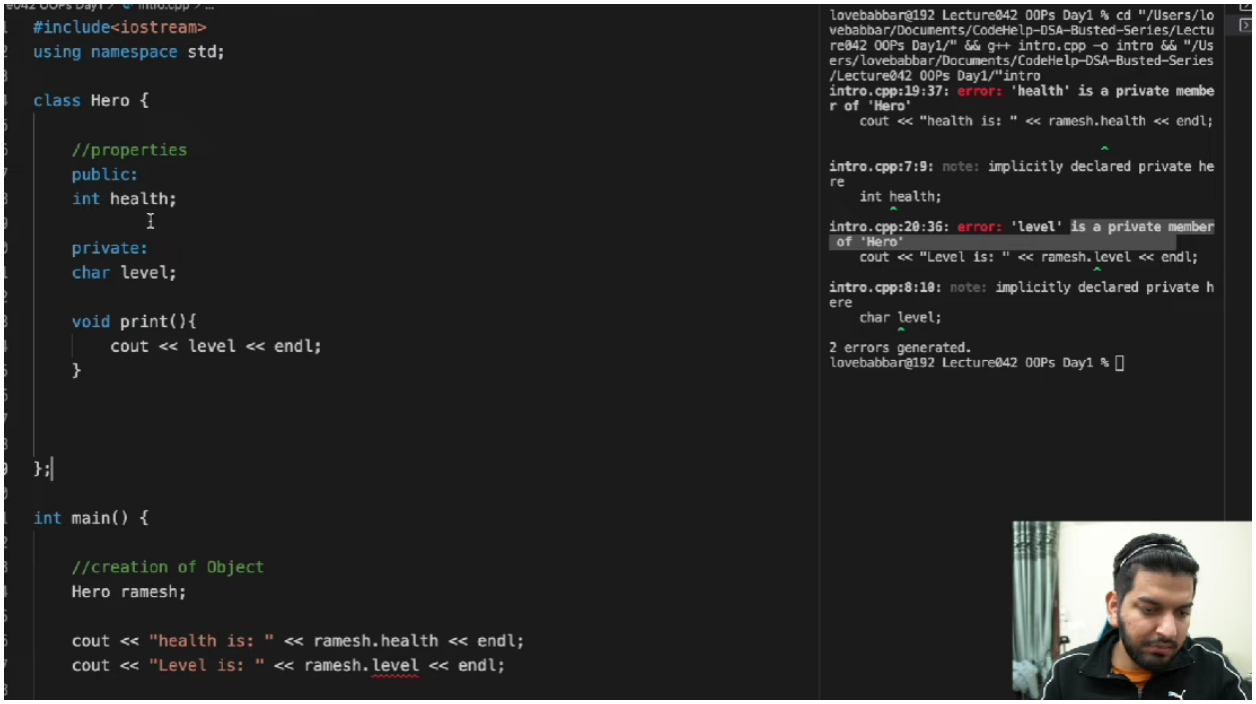
We can access any property of a Class by using a ***Dot Operator*** **“.”**

```
cout << "health is: " << ramesh.health << endl;
cout << "Level is: " << ramesh.level << endl;
```

Access Modifiers:



Public : Can access the properties anywhere
Private : Can access the properties only in the initiated Class.
Protected : ????

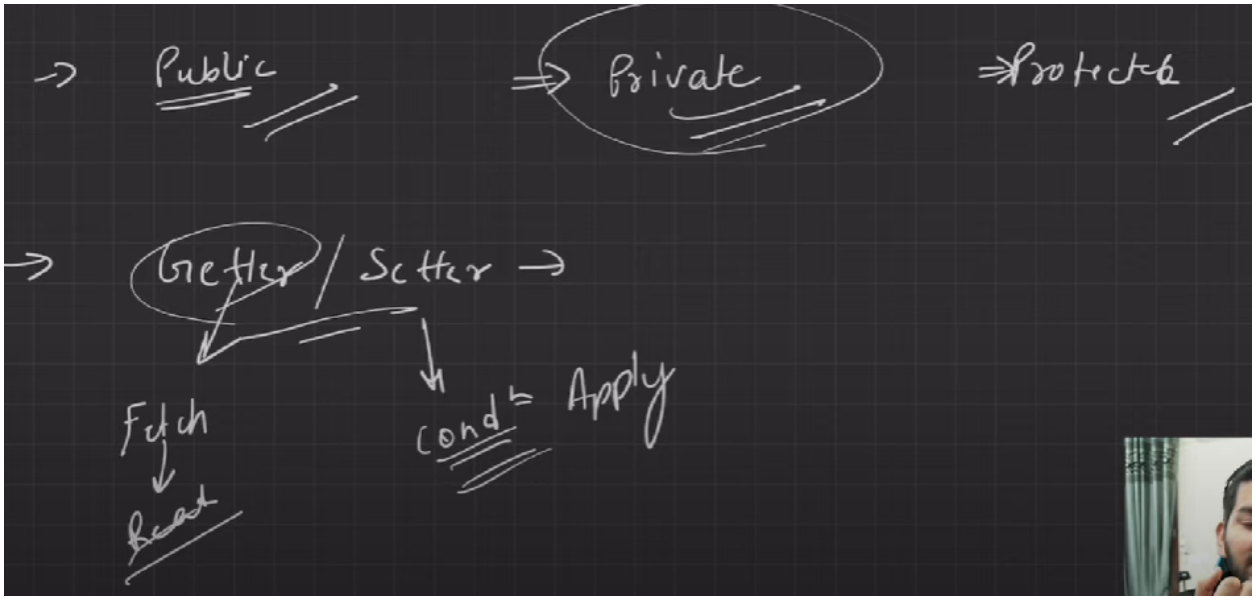


To **access** the properties which are private we can use the concept of ***getters and setters***.

Getters will **fetch** the data in private section and return it.
Setters will **set** the data in private section and return it.

Getters and Setters can be ***used for every access modifiers***.
These are nothing but **functions** which helps in accessing things.We create function in the class itself make it public and then access anywhere.We can call for it using the ***dot operator***.

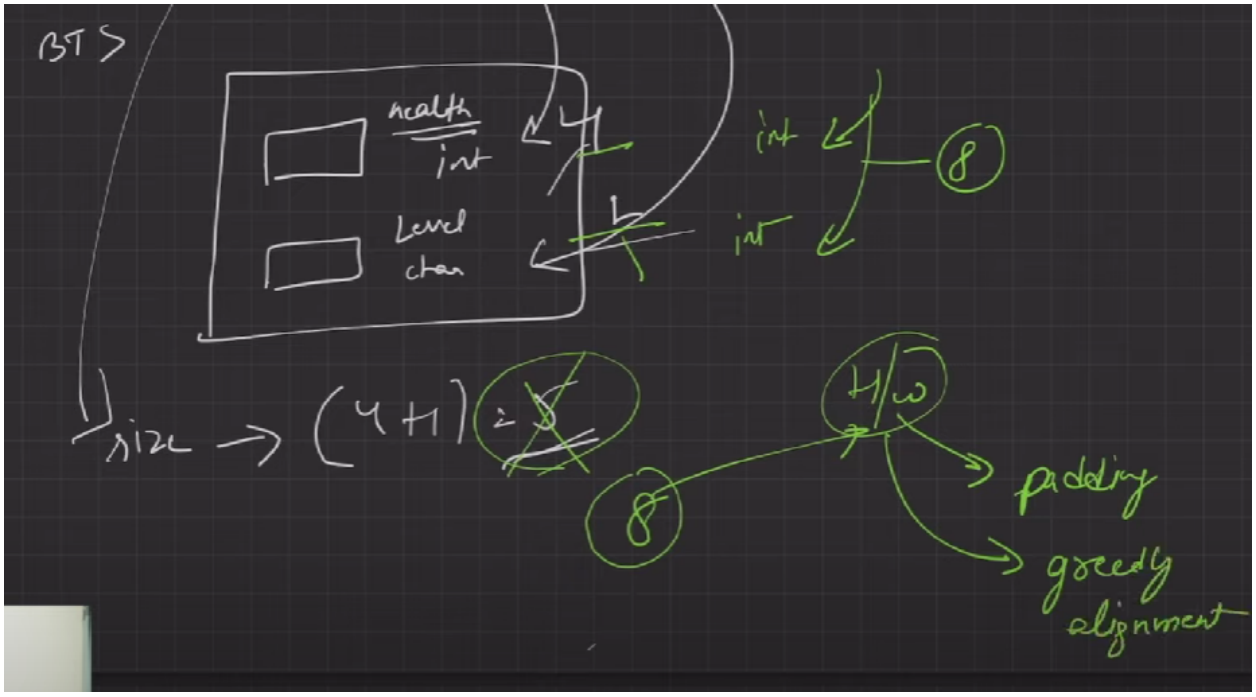
This also helps us if we want to apply any conditions before accessing.



Implementation of Getters and Setters:

```
int getHealth() {
    return health;
}

char getLevel() {
    return level;
}

void setHealth(int h) {
    health = h;
}

void setLevel(char ch) {
    level = ch;
}
```

Twist:



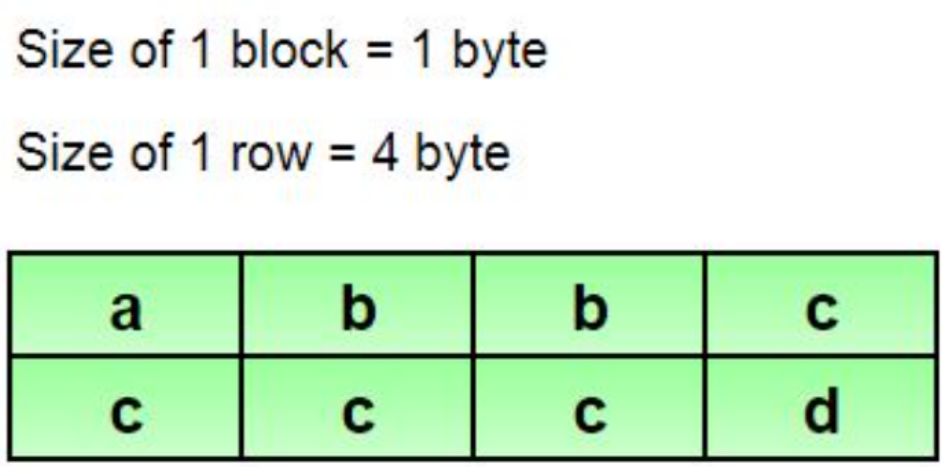
The class has one int and a char.
So the size of the class should have been 5 (4+1).
But it is 8.

What is Padding And Data Alignment?
https://www.geeksforgeeks.org/data-structure-alignment-how-data-is-arranged-and-accessed-in-computer-memory/

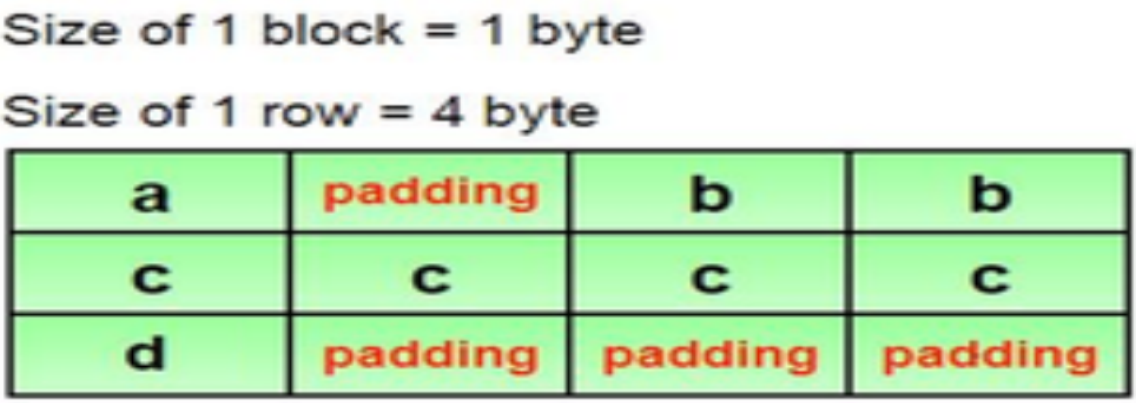
In Short,
While allocation of memory , the compiler takes the **biggest data type** as reference for Data Alignment and allocates according to that data type and if the smaller data types are to be filled they can be filled with the help of Padding .
Remember : **Memory Should be Continuous** .

```
struct
{
    char a;
    short int b;
    int c;
    char d;
}
```

Now we may think that the processor will allocate memory to this structure as shown below:

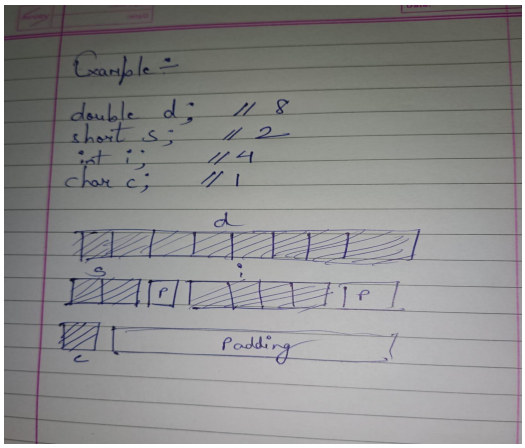


The total memory allocated in this case is 8 bytes. But this never happens as the processor can access memory with a fixed word size of 4 bytes. So, the integer variable c can not be allocated memory as shown above.



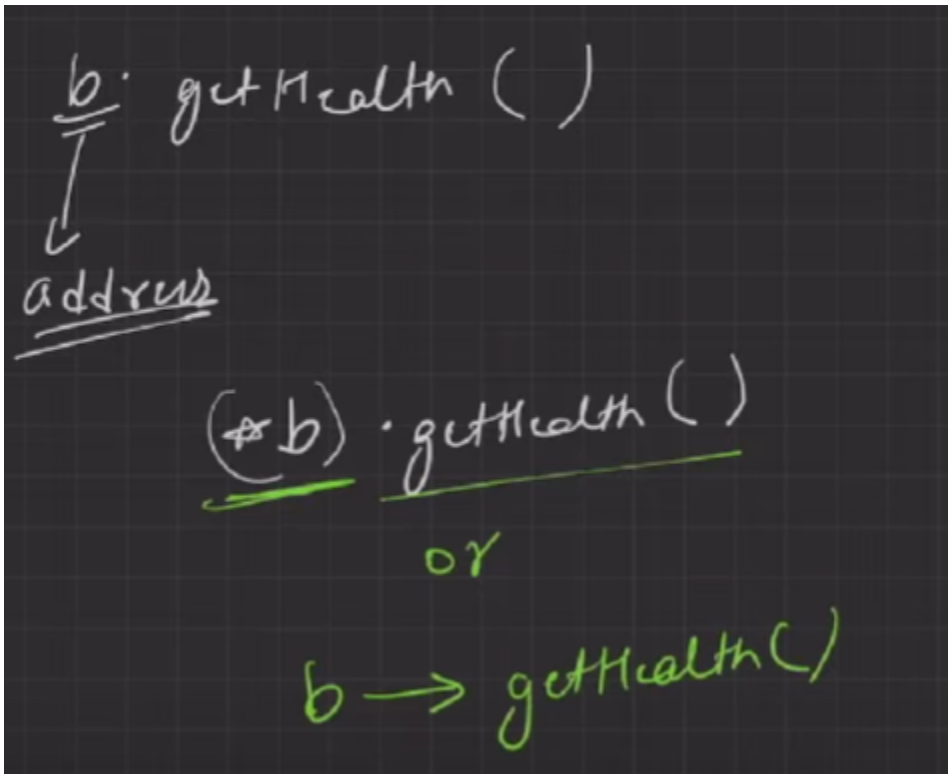
```
struct test1
{
    short s;
    // 2 bytes
    // 2 padding bytes
    int i;
    // 4 bytes
    char c;
    // 1 byte
    // 3 padding bytes
};

struct test2
{
    int i;
    // 4 bytes
    char c;
    // 1 byte
    // 1 padding byte
    short s;
    // 2 bytes
};
```

For a double data type:



Dynamic Allocation:

```
int main() {

    //static allocation
    Hero a;

    cout << "level is  " << a.level << endl;
    cout << " health is " << a.getHealth() << endl;

    //dynamicallly
    Hero *b = new Hero;
    cout << "level is  " << (*b).level << endl;
    cout << " health is " << (*b).getHealth() << endl;
```




```
cout << "level is  " << b->level << endl;
cout << " health is " << b->getHealth() << endl;
```

Note:

(*b). Is Equivalent to ***b->***

This is Dereferencing.

Constructors:

```
class Hero{
    Public:
    int health;
};

Hero h1;
```

When we create a object h1 a default constructor is called . This is mainly h1.Hero(); . The default constructor has the same name as the class.

If we create a Constructor then the default constructor gets killed , khtm , doesn't gets called.

```
class Hero{
    Public:
    int health;
    Hero(){
        cout<<"constructor called";
    }
};
```

This goes same for both static and dynamic allocation.
Dynamic Allocation : Hero* h1 = new Hero();

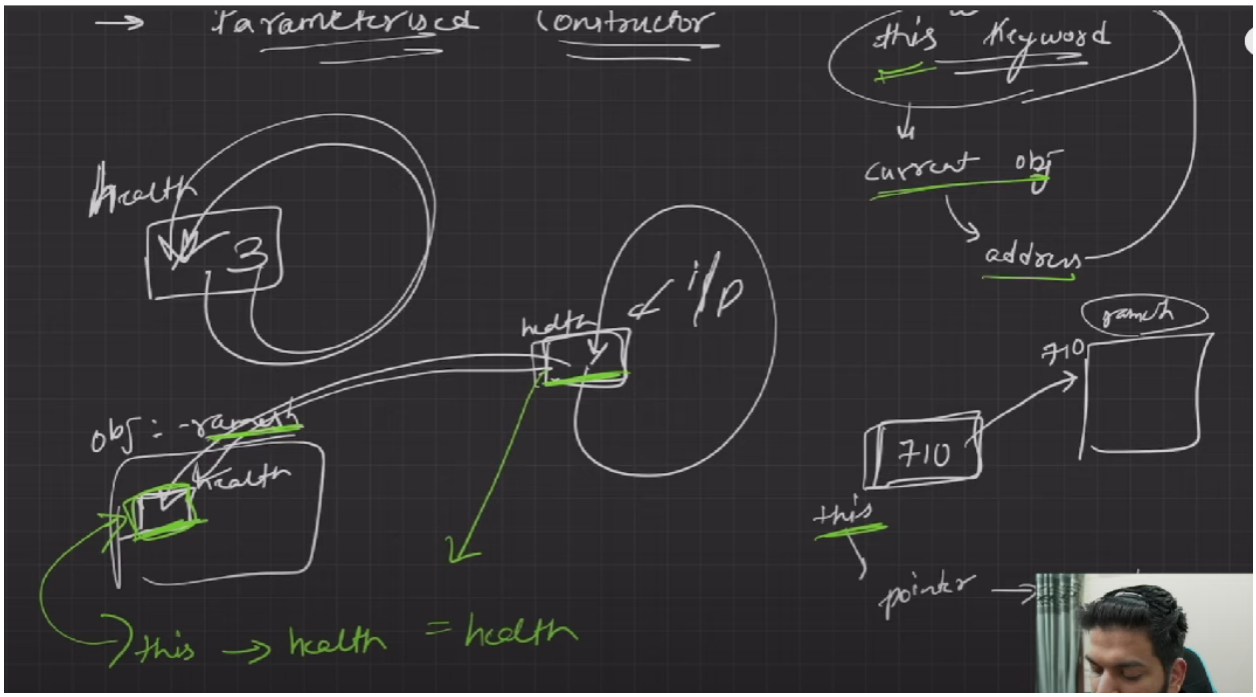
This Keyword:

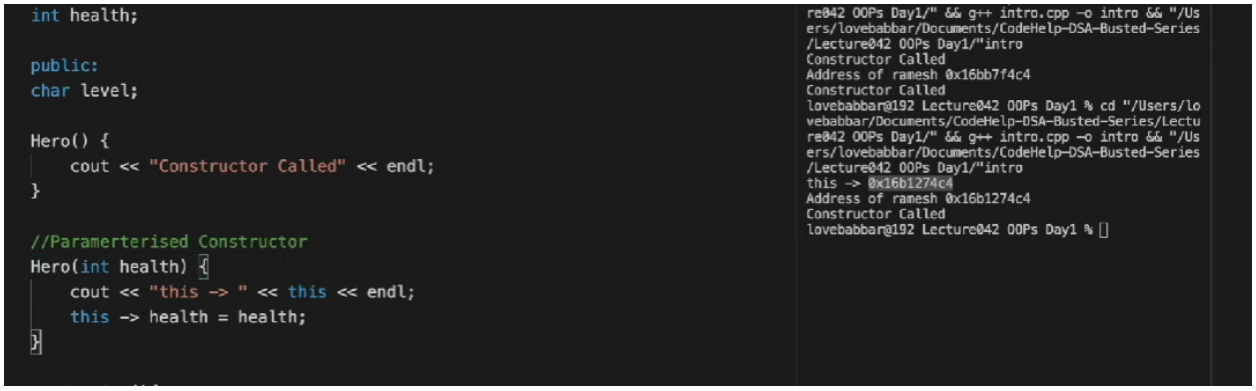
‘This’ this is a pointer which stored the address of the current object.
In our case,

Hero h1;

Here, this keyword will store the address of h1 object.

This helps in reducing the confusion between same name variables in constructors.





```
int health;

public:
char level;

Hero() {
    cout << "Constructor Called" << endl;
}

//Paramerterised Constructor
Hero(int health) {
    cout << "this -> " << this << endl;
    this -> health = health;
}
```

```
re042 OOPs Day1/" && g++ intro.cpp -o intro && "/Us
ers/lovebabbar/Documents/CodeHelp-DSA-Busted-Series
/Lecture042 OOPs Day1/"intro
Constructor Called
Address of ramesh 0x16bb7f4c4
Constructor Called
lovebabbar@192 Lecture042 OOPs Day1 % cd "/Users/lo
vebabbar/Documents/CodeHelp-DSA-Busted-Series/Lectu
re042 OOPs Day1/" && g++ intro.cpp -o intro && "/Us
ers/lovebabbar/Documents/CodeHelp-DSA-Busted-Series
/Lecture042 OOPs Day1/"intro
this -> 0x16b1274c4
Address of ramesh 0x16b1274c4
Constructor Called
lovebabbar@192 Lecture042 OOPs Day1 %
```

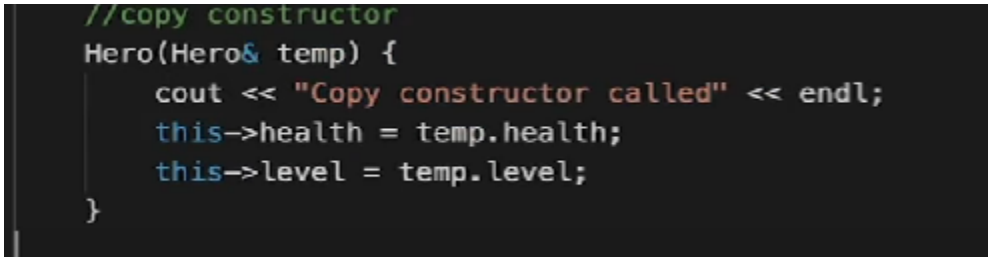
Note: Suppose we created a constructor and we know that the **default constructor** gets killed and if after some time or due to any reason we delete our defined constructor then the default constructor does not come in handy as it was killed. ***Once Killed Gone for Always.***

Copy Constructor:

Just like our default Constructor , copy constructor is also generated automatically.
To call copy constructor:

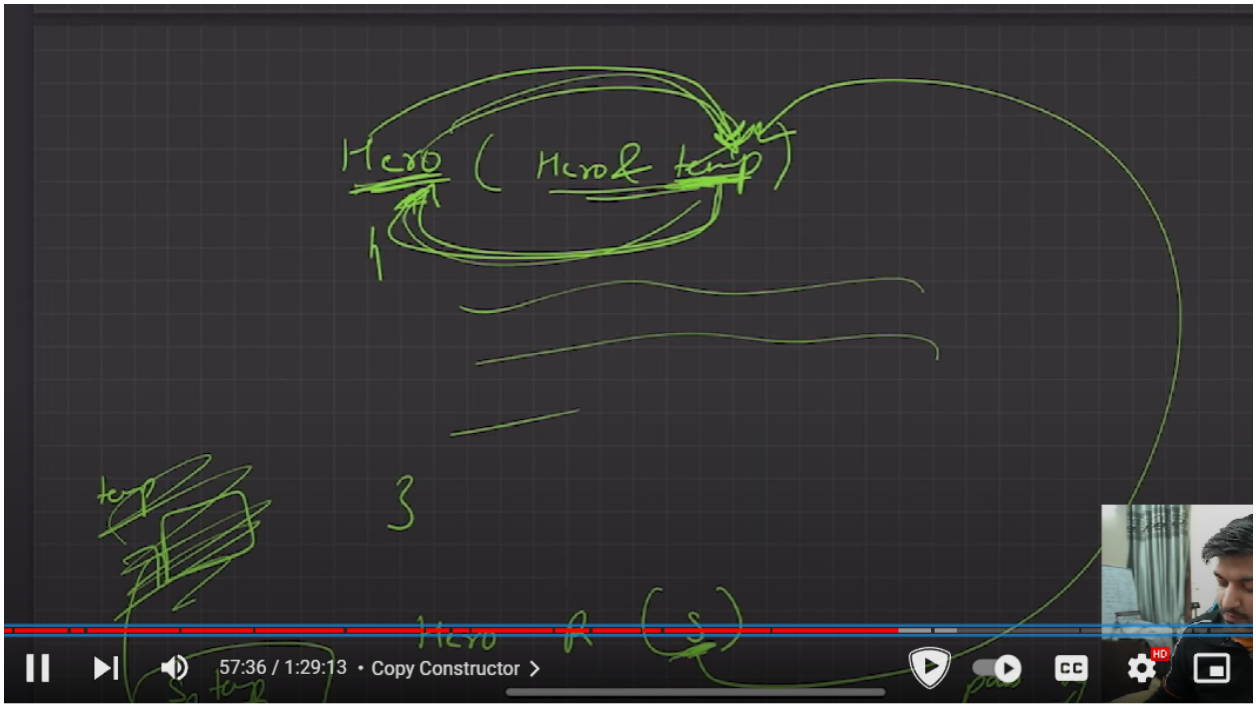
Hero ramesh;
Hero suresh(ramesh);

UserBuilt:



```
//copy constructor
Hero(Hero& temp) {
    cout << "Copy constructor called" << endl;
    this->health = temp.health;
    this->level = temp.level;
}
```

Here , while passing our object we ***pass it by reference*** or it will throw an error. Reason behind the error is that if we pass it by value , the compiler will try to make a copy of that object so the copy constructor will be called and already there is an object so it gets stuck in an ***infinite loop .*** That's why we call it by passing the reference.

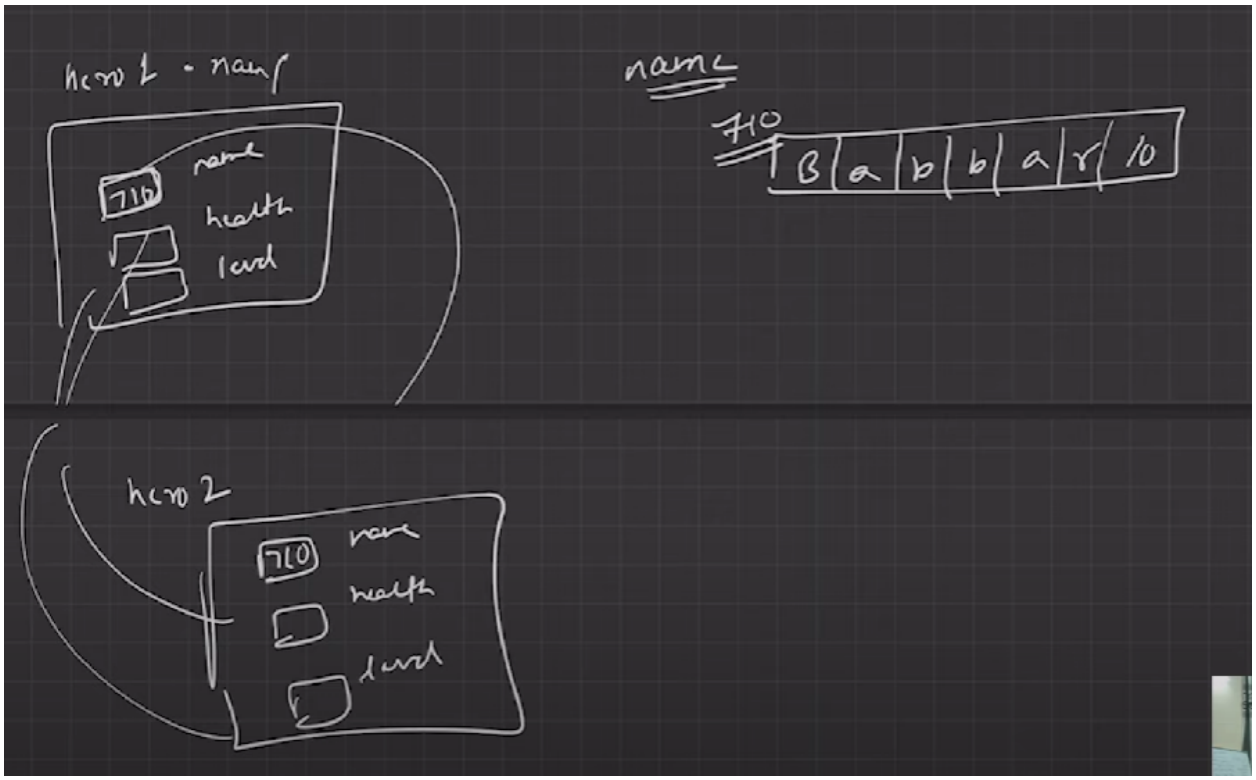


Shallow And Deep Copy :

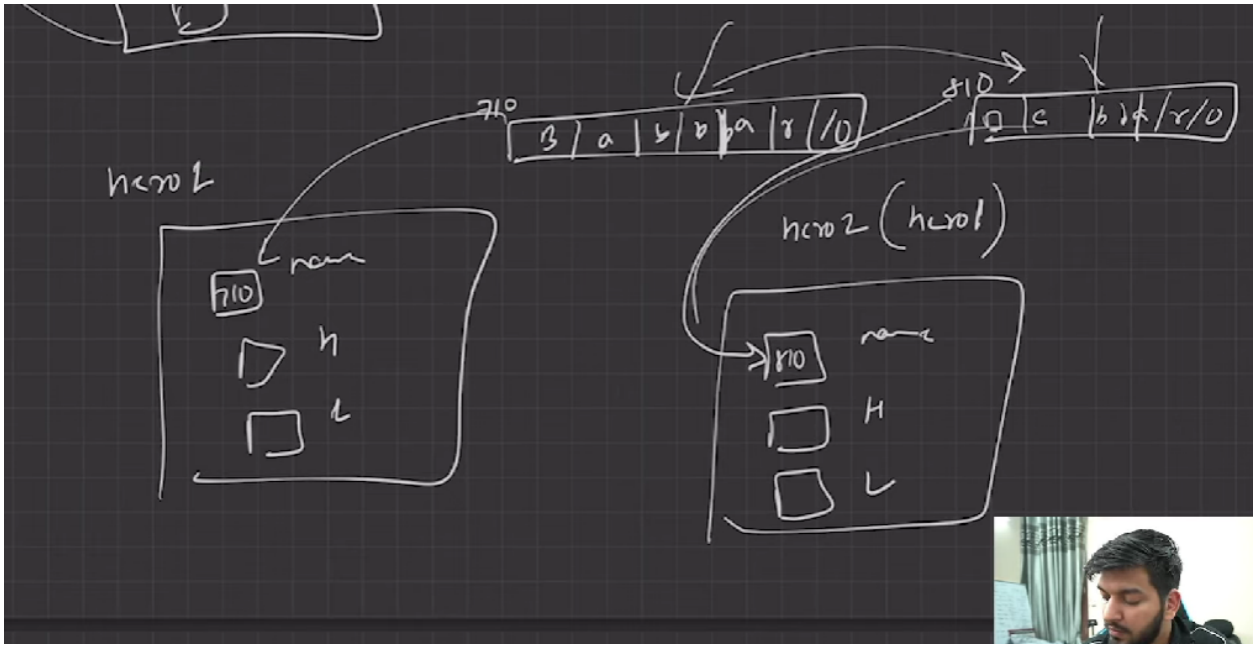
Shallow Copy : Both the objects use the ***same memory.***

Deep Copy : Both the objects use the ***different memory***.This is achieved , when copy constructor is called we *create a different memory and assign it to the new object*.

Shallow:



Deep Copy :




```
Hero(Hero& temp) {

    char *ch = new char[strlen(temp.name) + 1];
    strcpy(ch, temp.name);
    this->name = ch;

    cout << "Copy constructor called" << endl;
    this->health = temp.health;
    this->level = temp.level;
}
```

Copy Assignment Operator:

This operator is the ***same as our equal operator***.
For eg:

Hero h1 ;
Hero h2;
//Assign them values.
h1 = h2;
Here , h2 will get copied in h1.

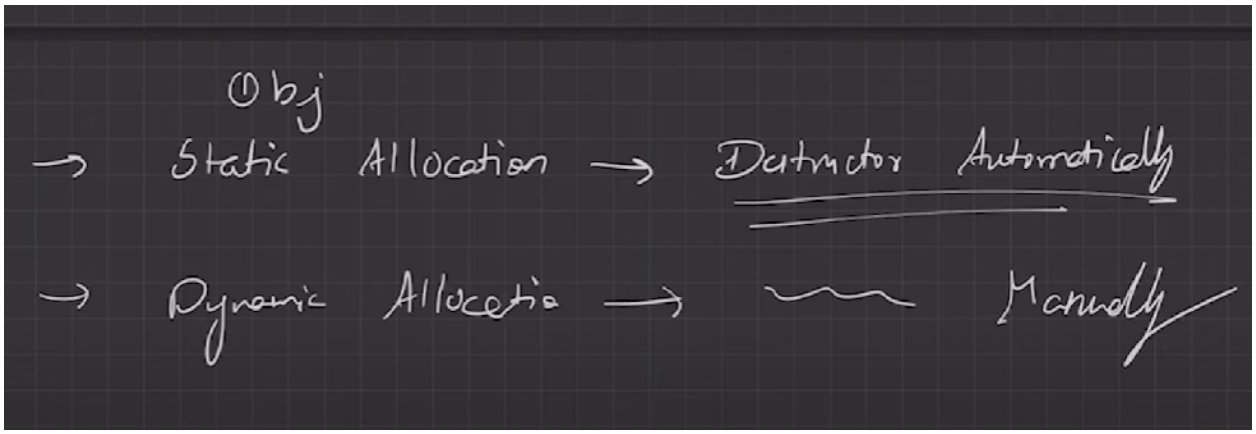
Destructor:

Just like our default constructor a default destructor is also made.
It is also generated automatically.
It ***does not take any inputs , no return type , name same as Class name***.

Destructor is used for the ***deallocation of the memory*** used in the program .
That is to free the memory at the ***end of the main function***.

Implementation:

```
~Hero(){
        cout<<"Destructor is called"<<endl;
}
```




```
Hero *b = new Hero();
delete b;
```

Static Member:

Static keyword make the data member ***independent of any other aspect***.
Static data member **belongs to the class** .
There's ***no need to initialize an object*** to access the data member.

This is how we initialize the static data member and access it in the main function.

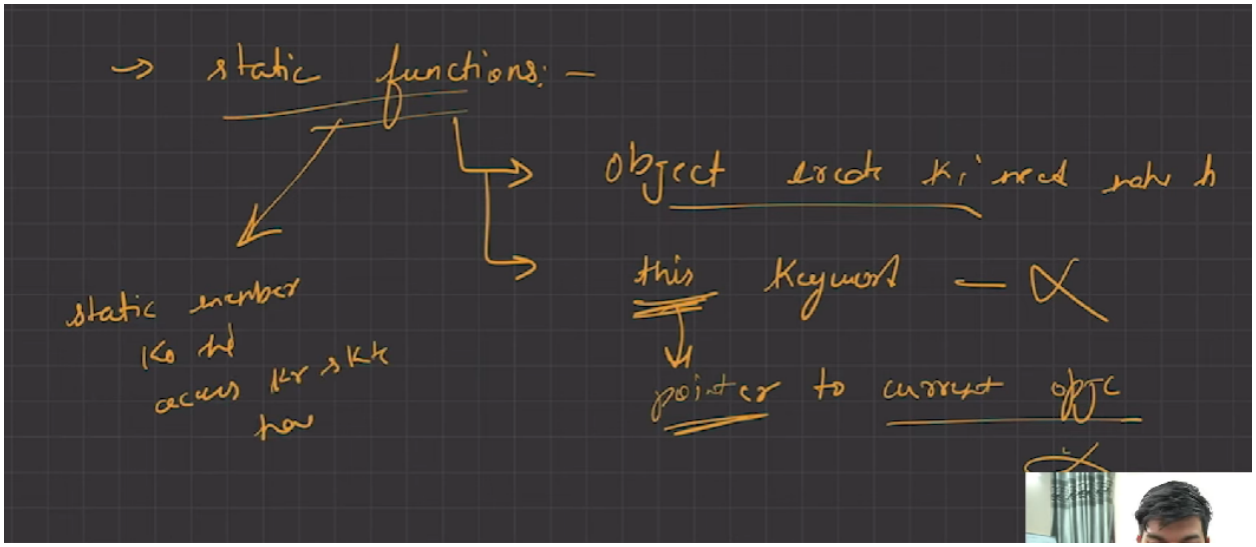
```
int Hero::timeToComplete = 5;

int main() {

    cout << Hero::timeToComplete << endl;

    Hero a;

    cout << a.timeToComplete << endl;


    Hero b;
    b.timeToComplete = 10 ;
```

Static Function:

This is the **same as the static member**.
Keypoints are that :

There is ***no need to create an object*** to access it.

As there is no object there is ***no this keyword***.




```
70      static int random() {
71          return timeToComplete ;
72      }
73
74      //Destructor
75      ~Hero() {
76          cout << "Destructor bhai called" << endl;
77      }
78
79  };
80
81  int Hero::timeToComplete = 5;
82
83  int main() {
84
85      //cout << Hero::timeToComplete << endl;
86      cout << Hero::random() << endl;
87
```

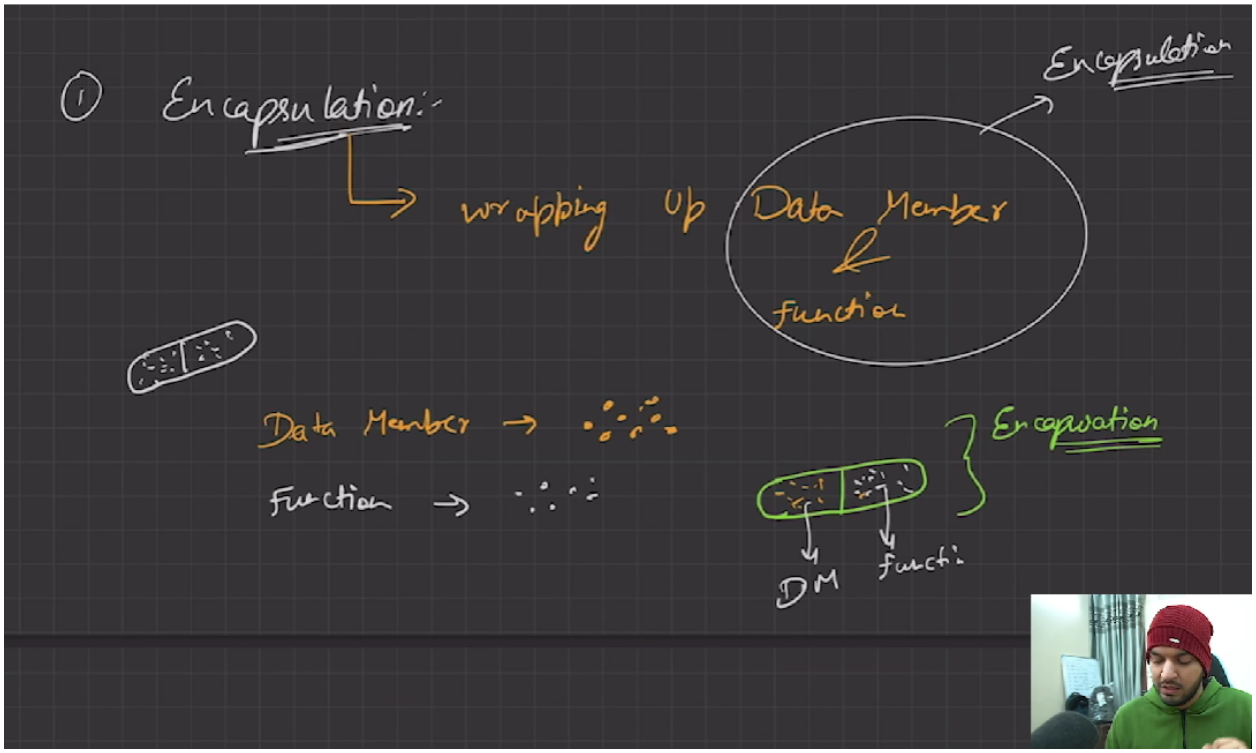
Note:

Only Static Members can only use *Static Functions.*

Pillars of OOPs :

1.Encapsulation:

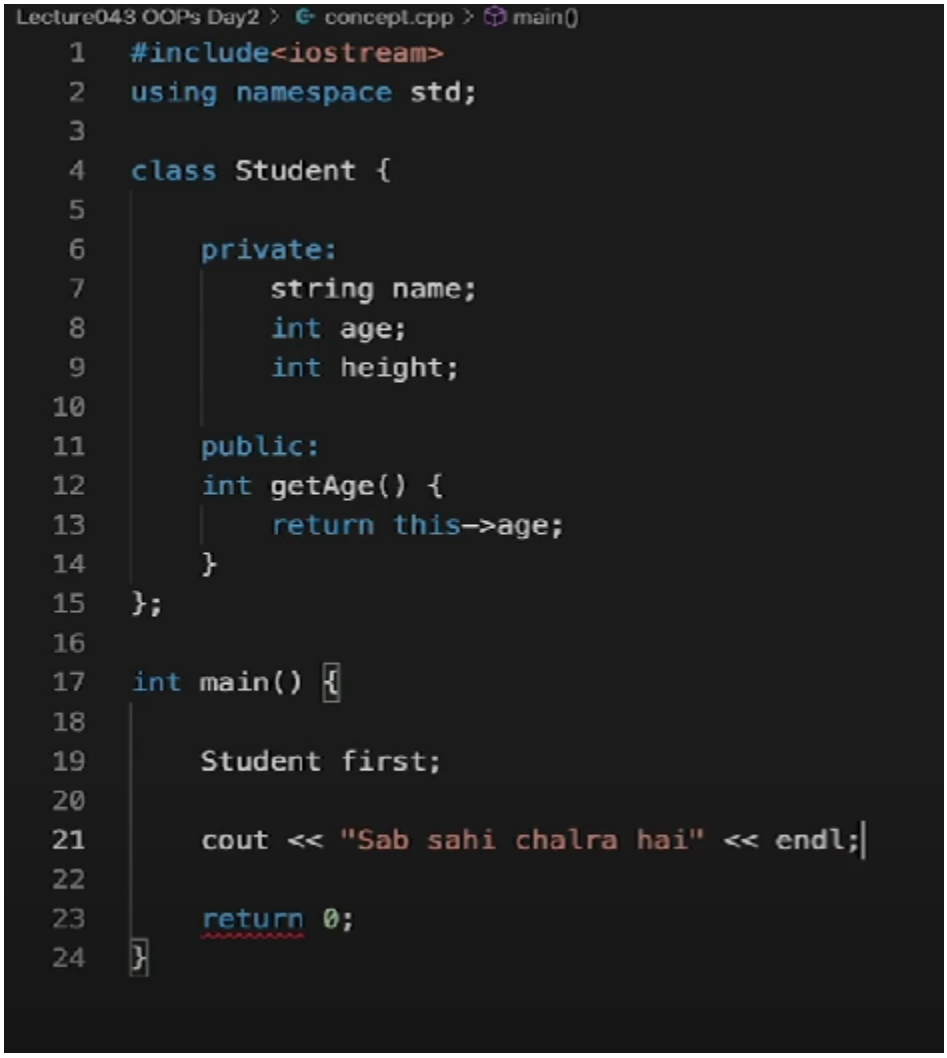
Definition : Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.



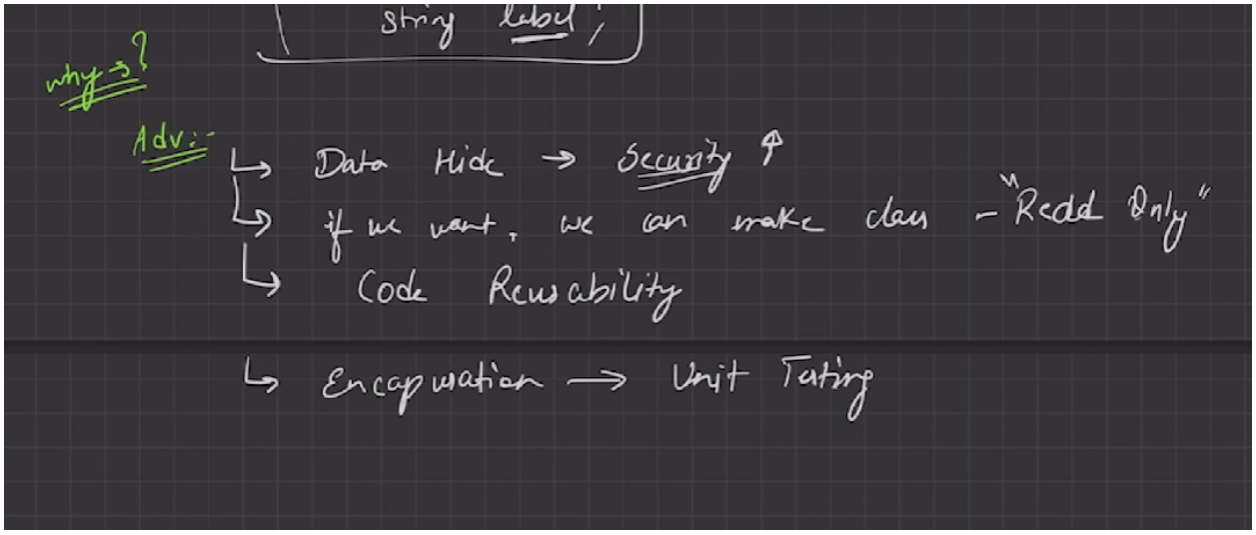
Data Members->Properties/state

Functions -> Methods/Behaviour

Implementation:



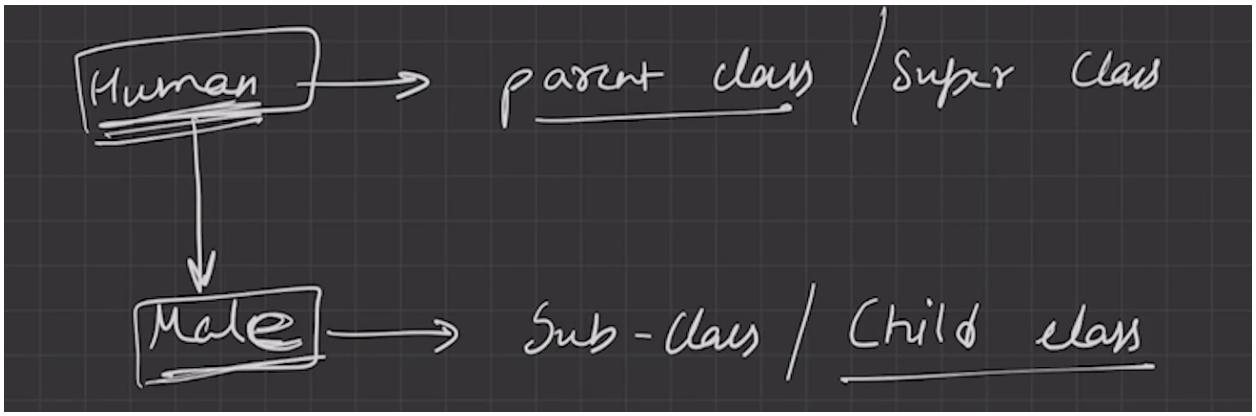
Advantages:



Encapsulation is nothing but a simple implementation of data members and functions which are private .
Encapsulation -> Data Hiding ->Information Hiding

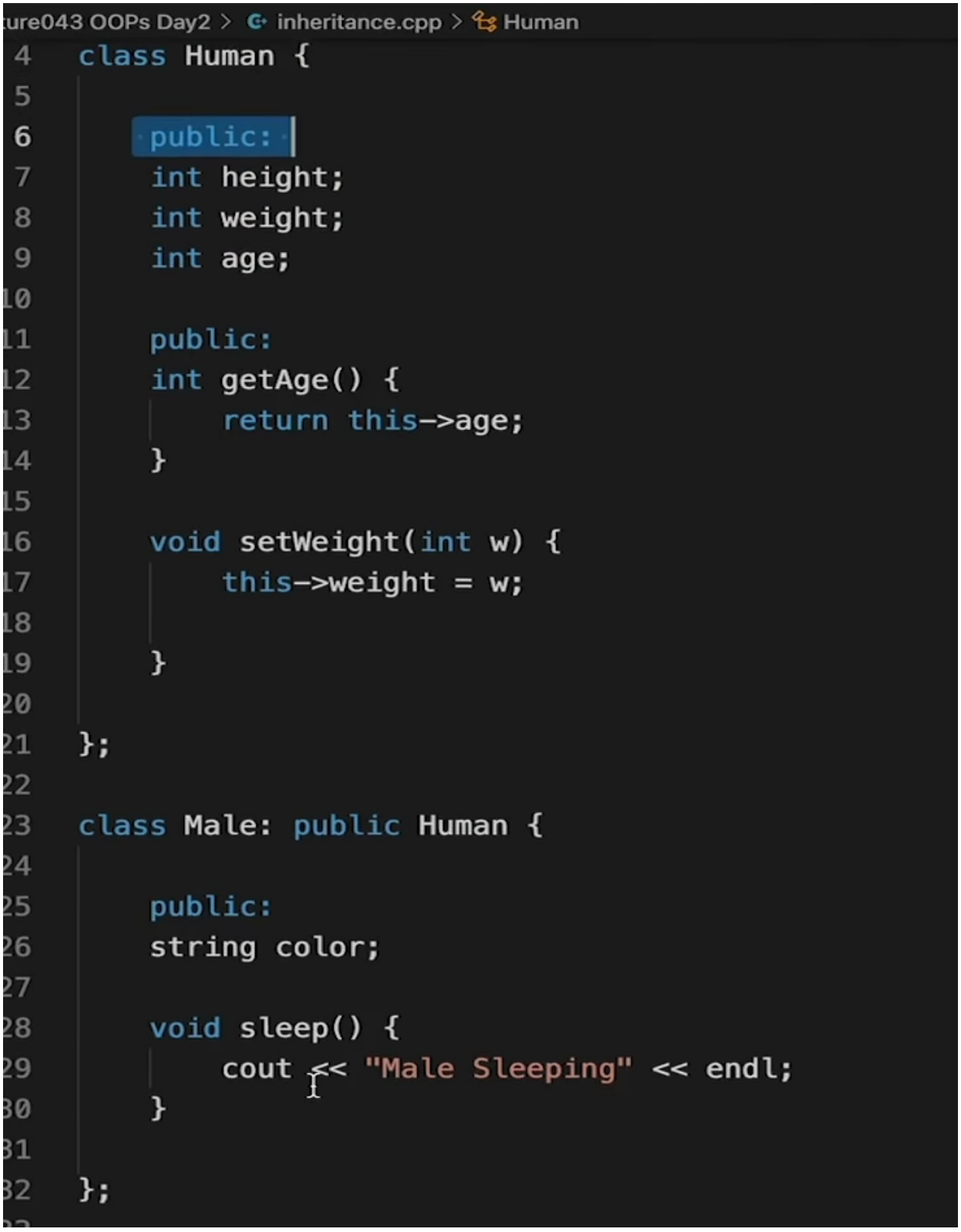
2. Inheritance:

Inheriting the properties of a class into another class.



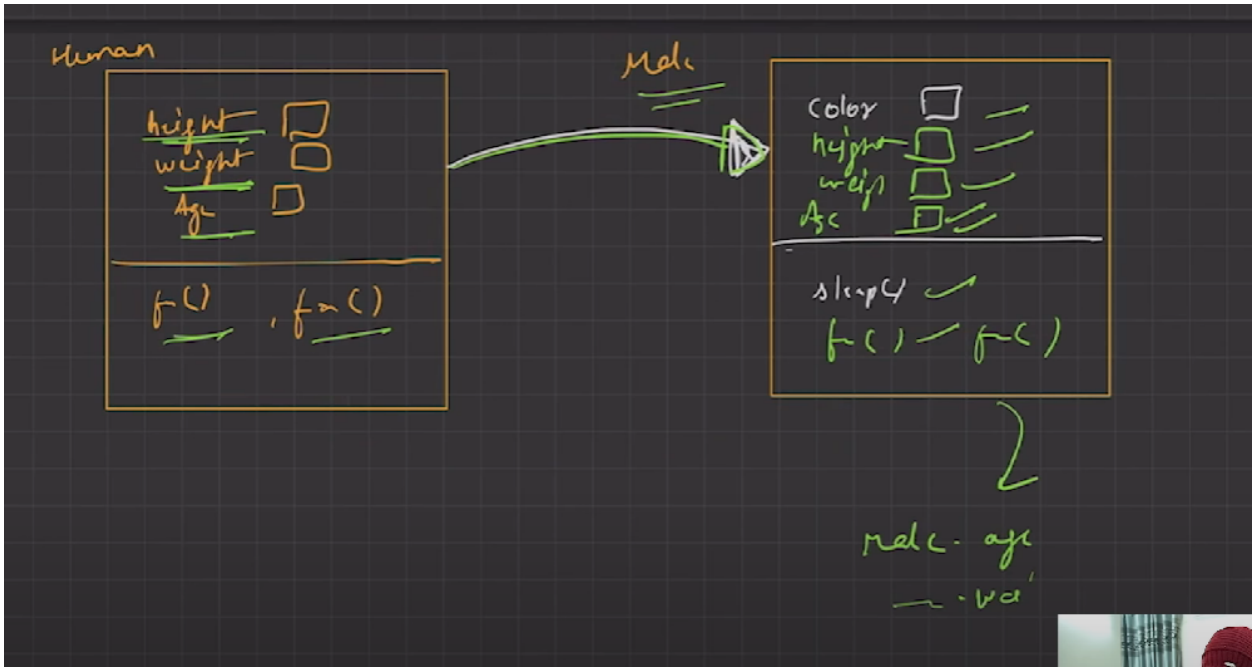
C++ inheritance syntax:

```
class parent_class {
    //Body of parent class
};
class child_class: access_modifier parent_class {
    //Body of child class
};
```


ture043 OOPs Day2 > inheritance.cpp > Human
class Human {
public:
int height;
int weight;
int age;
public:
int getAge() {
return this->age;
}
void setWeight(int w) {
this->weight = w;
}
};
class Male: public Human {
public:
string color;
void sleep() {
cout << "Male Sleeping" << endl;
}
};

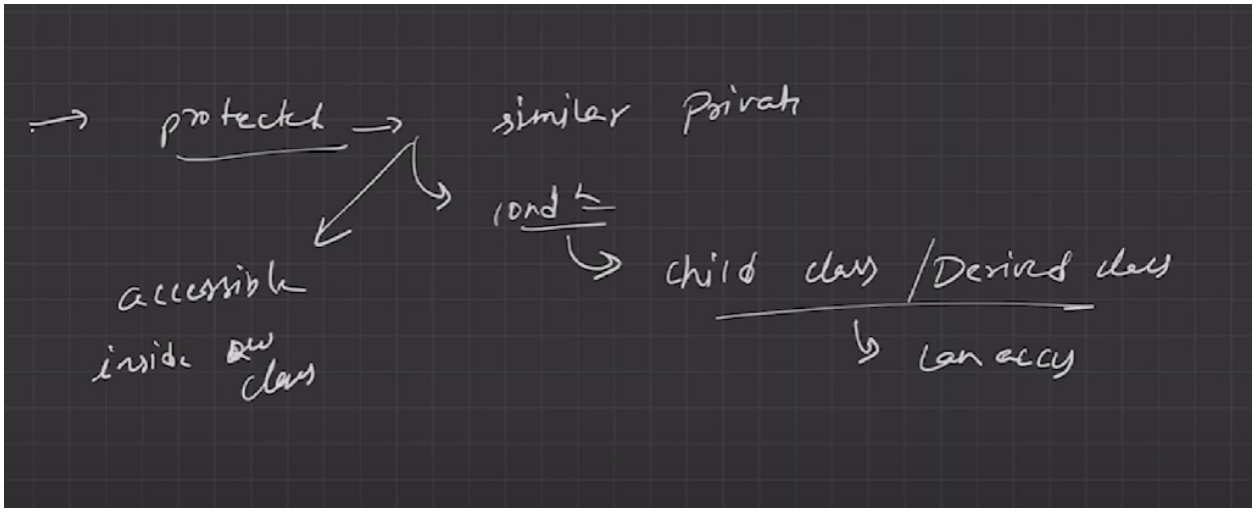
Definition:

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.



Base Class member Access Specifier	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not Accessible	Not Accessible	Not Accessible

What is Protected Access Modifier?



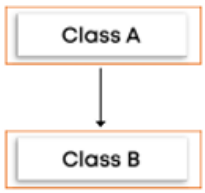
Types of Inheritance

C++ supports five types of inheritance they are as follows:

1) Single inheritance
2) Multilevel inheritance
3) Multiple inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

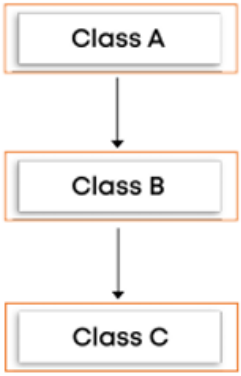
1. Single Inheritance:

In single inheritance, one class can extend the functionality of another class. In single inheritance, there is only one parent class and one child class.



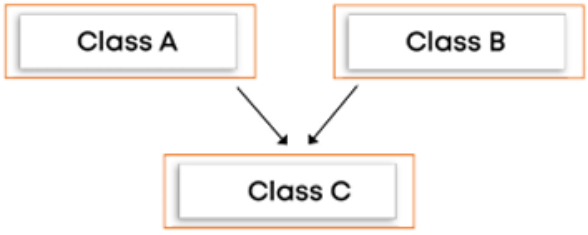
Single Inheritance

When a class inherits from a derived class, and the derived class becomes the base class of the new class, it is called multilevel inheritance. In multilevel inheritance, there is more than one level.



Multi-Level Inheritance

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance, a single child class can have multiple parent classes.

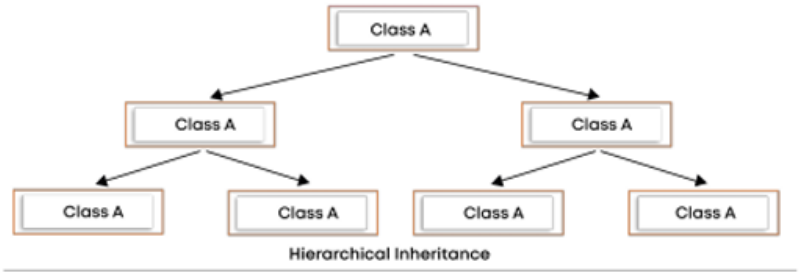


Multiple Inheritance

```
// Parent class
class Animal {
    public:
        void eat() {
            cout << "eating" << endl;
        }
};

// Parent class
class Dog {
    public: void bark() {
        cout << "barking" << endl;
    }
};

// Inherited class
class BabyDog: public Animal, public Dog {
    public: void weep() {
        cout << "weeping";
    }
};
```

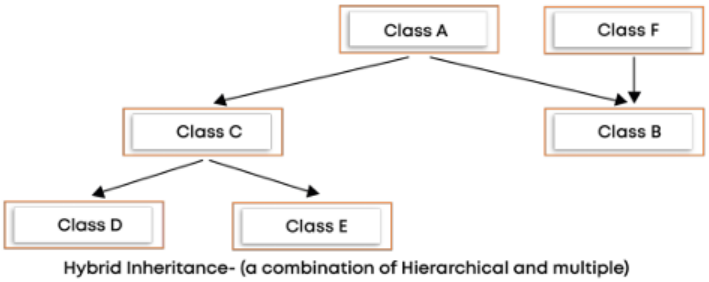
Overview

In hierarchical inheritance, one class serves as a base class for more than one derived class.



```
using namespace std;
// Parent class
class Animal {
    public:
        void eat() {
            cout << "eating" << endl;
        }
};

// child class1
class Dog: public Animal {
    public: void bark() {
        cout << "barking" << endl;
    }
};

// child class2
class Cat: public Animal {
    public: void meow() {
        cout << "meowing" << endl;
    }
};
```

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritances can be called hybrid inheritance.



Hybrid Inheritance- (a combination of Hierarchical and multiple)

```
// Parent class1
class Vehicle {
    public:
        Vehicle() {
            cout << "This is a Vehicle" << endl;
        }
};

//Parent class2
class Fare {
    public:
        Fare() {
            cout << "Fare of Vehicle\n";
        }
};

//Child class1
class Car: public Vehicle {

};

//Child class2
class Bus: public Vehicle, public Fare {

};
```

Inheritance Ambiguity

C++ Ambiguity

There may be a possibility that a class may inherit member functions with the same name from two or more base classes, and the derived class may not have functions with the same name as those of its base classes. If the derived class object needs to access one of the same-named member functions of the base classes, it results in ambiguity as it is not clear to the compiler which base's class member function should be invoked.

Avoid ambiguity using scope resolution operator

The ambiguity can be resolved by using the **scope resolution operator** by specifying the class in which the member function lies as given below:

Syntax:

```
object.class_name::method_name();
```

```
class A {
    public:
        void abc() {
            cout << "Class A";
        }
};

class B {
    public:
        void abc() {
            cout << "Class B";
        }
};

class C: public A, public B {
    public:
};

//Main Code
int main() {
    C obj;
    obj.A :: abc();
}
```

3. Polymorphism:

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism is a concept that allows you to perform a single action in different ways. Polymorphism is the combination of two Greek words. The poly means many, and morphs means forms. So polymorphism means many forms. Let's understand polymorphism with a real-life example.

Real-life example: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

Types of polymorphism

There are two types of polymorphism in C++.

- Compile-time polymorphism
- Runtime polymorphism

a) Function overloading: When there are multiple functions in a class with the same name but different parameters, these functions are overloaded. The main advantage of function overloading is it increases the readability of the program. Functions can be overloaded by using different numbers of arguments and by using different types of arguments.

b) Operator Overloading: C++ also provides options to overload operators. For example, we can make the operator ('+') for the string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. A single operator, '+,' when placed between integer operands, adds them and concatenates them when placed between string operands.

Points to remember while overloading an operator:

- It can be used only for user-defined operators(objects, structures) but cannot be used for in-built operators(int, char, float, etc.).

List of operators that can be overloaded in C++:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

List of operators that cannot be overloaded in C++:

::	.*	.	?:

2.Runtime polymorphism

Runtime polymorphism is also known as dynamic polymorphism. Method overriding is a way to implement runtime polymorphism.

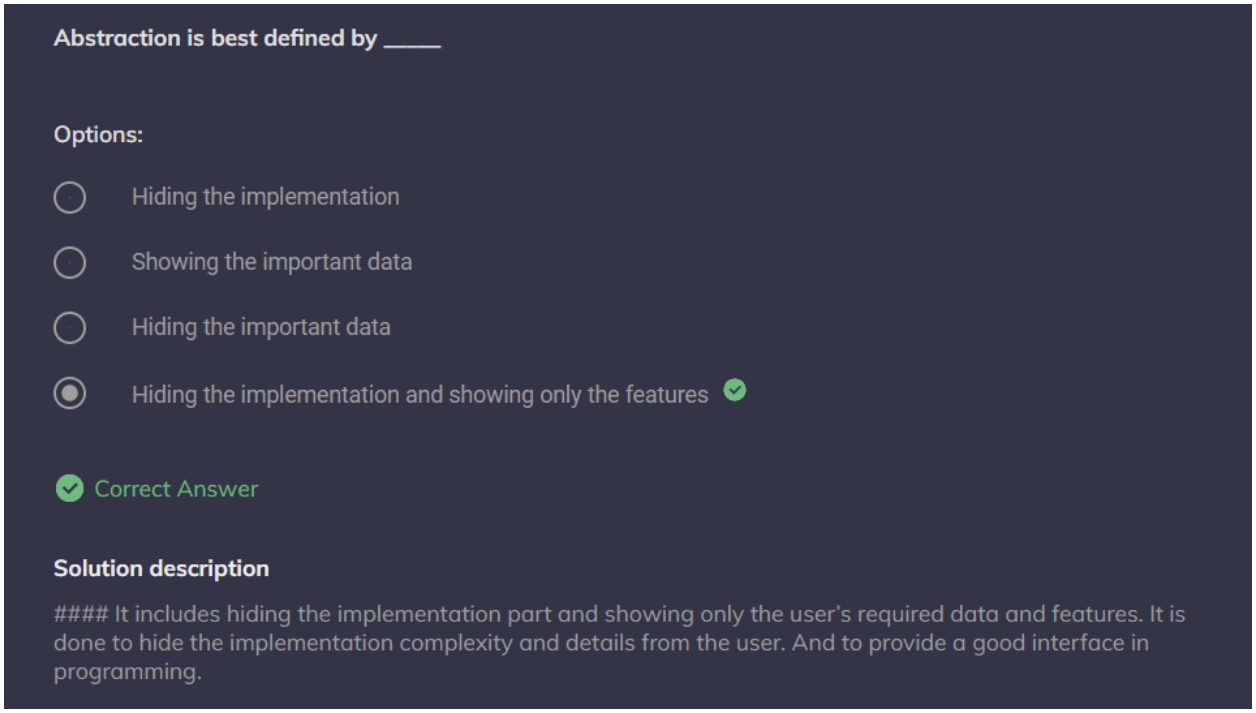
a) Method overriding:
Method overriding is a feature that allows you to redefine the parent class method in the child class based on its requirement. In other words, whatever methods the parent class has by default are available in the child class. But, sometimes, a child class may not be satisfied with parent class method implementation. The child class is allowed to redefine that method based on its requirement. This process is called method overriding.

Rules for method overriding:

- The method of the parent class and the method of the child class must have the same name.
- The method of the parent class and the method of the child class must have the same parameters.
- It is possible through inheritance only.

```
#include<iostream>
using namespace std;
class Parent {
    public:
        void show() {
            cout << "Inside parent class" << endl;
        }
};
class subclass1: public Parent {
    public: void show() {
        cout << "Inside subclass1" << endl;
    }
};
class subclass2: public Parent {
    public: void show() {
        cout << "Inside subclass2";
    }
};
int main() {
    subclass1 o1;
    subclass2 o2;
    o1.show();
    o2.show();
}
Output:
Inside subclass1
Inside subclass2
```

4. Abstraction:



Many answers and their examples are misleading.
Encapsulation is the packing of *"data"* and *"functions operating on that data"* into a single component and restricting the access to some of the object's components.
Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.
Abstraction is a mechanism which represent the essential features without including implementation details.
Encapsulation:-- **Information hiding**.
Abstraction:-- **Implementation hiding**.
Example (in C++):

```
class foo{
    private:
        int a, b;
    public:
        foo(int x=0, int y=0): a(x), b(y) {}

        int add(){
            return a+b;
        }
}
```

Internal representation of any object of foo class is hidden outside of this class. **--> Encapsulation**.
Any accessible member (data/function) of an object of foo is restricted and can only be accessed by that object only.

```
foo foo_obj(3, 4);
int sum = foo_obj.add();
```

Implementation of method add is hidden. **--> Abstraction**.