OOPs

Object Oriented Programming

Lecture 42

Class

→ user defined data type

eg: `Hero h1;`

- `Hero` → class
- `h1` → object of class Hero

Objects

→ instance of a class

Code →

```
class Hero
{
    int health;
};
int main()
{
    Hero h1;
    cout << sizeof(h1) << endl;
    return 0;
}
```

`;` → semicolon after class

`sizeof(h1)` → prints 4

Empty Class

```
class Hero
{
};
int main()
{
  Hero h1;
  cout << sizeof(h1) << endl;
  return 0;
}
```

prints 1

* 1 byte of memory is allocated for its identification and to keep track of it.

Using class of different file in the current

```
#include "Hero.cpp"
```

→ oops101.cpp

Accessing properties / data members

- using "." operator

example → hl.health;

Access Modifiers

- public
- private (by default)
- protected

Public

- public properties can be accessed even outside the class

Private

- only within the class
- by default always private

Getter/Setter

↳ if the variable is under private, we can have a public function that returns the value of thats variable.

example: → getter_setter.cpp

```
class Hero
{
  private:
    int health;
  public;
    int gethealth()
    { return health;
    }
    int sethealth(int h)
    { health = h;
    }
};
```

→ getter

→ setter

* Size of the object of the class is more than just summation of all the data members, rather its minimum sum gauranteed.

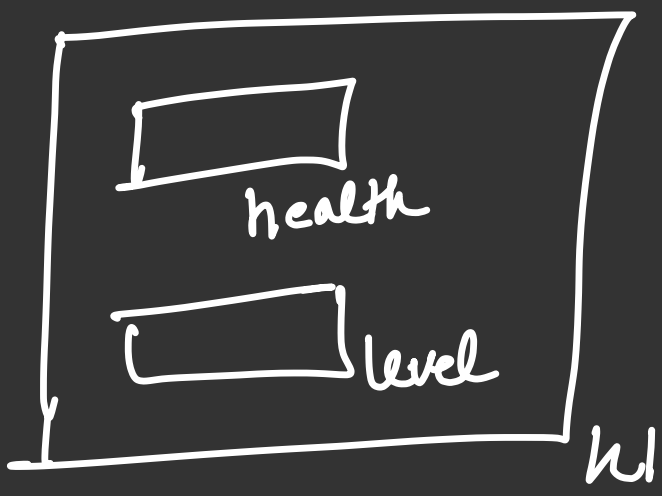
→ Compiler adds some padding for data member alignments.

Static Allocation

This,

Hero h1;

→ was static allocation



Dynamic allocation

```
Hero *h = new Hero;
```

(in heap memory)

* Now to access data members and functions, we use

→ `(*h).gethealth();`

or

`h -> gethealth();`

→ static_dynamic.cpp

Constructor

- invoke when object is created
- no return type
- no i/p parameter
- same name as class name

`Hero h1;` is same as `Hero h1();`

Parameterised Constructor

```
Hero(int h)
{
    health = h;
}
```

parameters

☆ "this" keyword

↓

our current object address is stored in this keyword

Thus,

```
Hero (int health)
{
this -> health = health;
}
```

this -> health ⟶ object.health

cout << this << endl;

↳ prints address of object

&(*h) ⟶ address of object h

when ↓

Hero *h = new Hero;

Copy Constructor

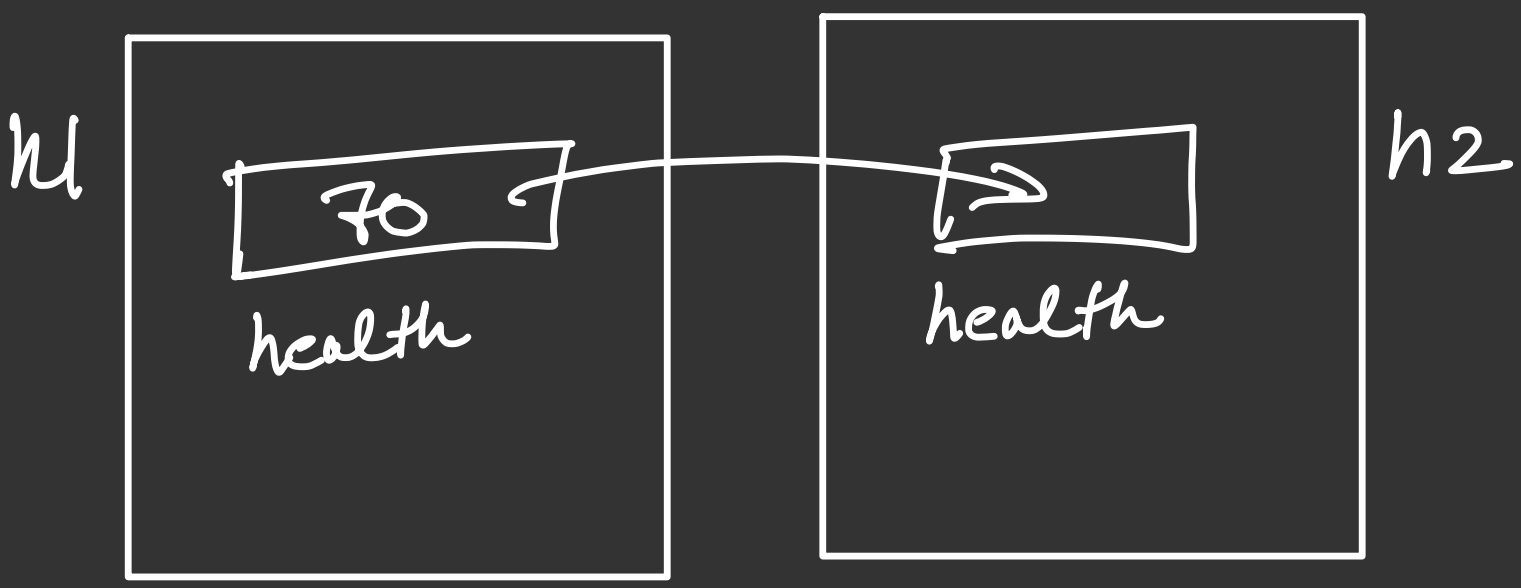
```
Hero h1;
h1.health = 70;
```

```
Hero h2(h1);
```

→ passing object as parameter

→ same as saying:

```
h2.health = h1.health;
```



✡ If we use parameterised constructors, then we need to create default constructor as well.

✡ Copy constructor requires '&'

```
Hero(Hero& temp)
{
   ==
}
```

```
Hero h1;
Hero h2(h1);
```

✡ If not used, then pass by value, copy is created temp, but in order for it to be created it calls default constructor which is the function itself.

Thus, pass by reference

If we need to use char array - we don't define it along other datas members we just define a pointer

```
char *name;
```

and in the default constructor ⟶

```
Hero()
{
    name = new char[100];
}
```

⟶ dynamically

Otherwise is a BAD PRACTICE

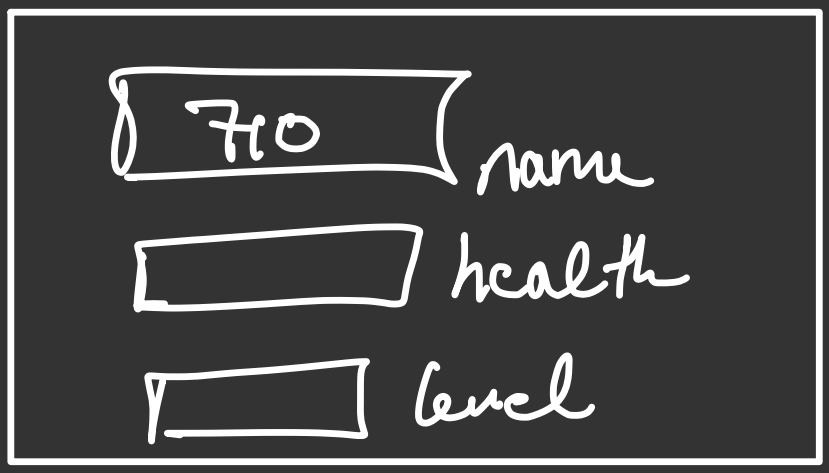
⟶ copy_constructor.cpp

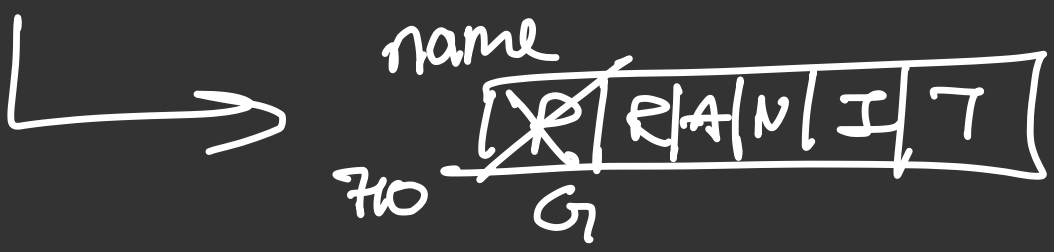
Shallow and Deep Copy

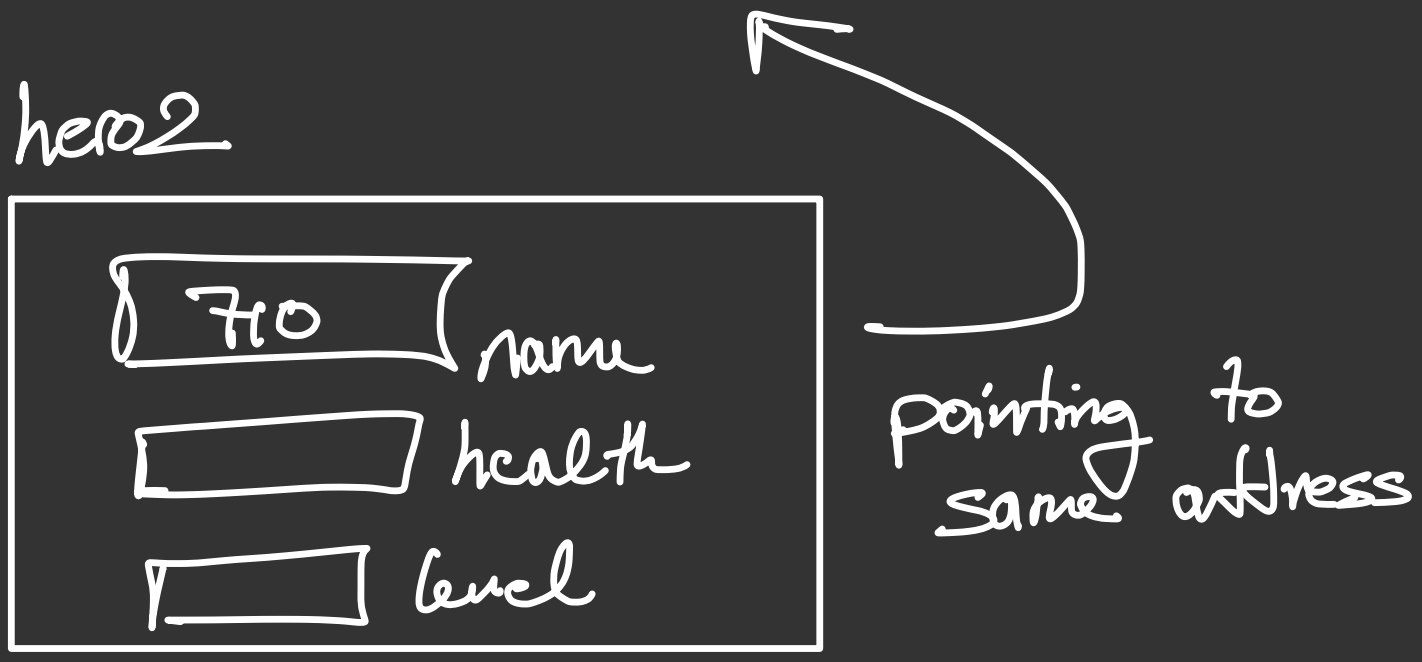
```
// defaults copy constructor
Hero hero2(hero1);
```

The problem is that when changes made to one objects name it automatically makes changes to other objects name, because ⟶

```
hero1.name[0] = 'G';
```







This is called Shallow Copy

Solution is Deep Copy

↳ making another copy of name [P|R|A|N|I|T] at a different address

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

Thus, now different addresses.

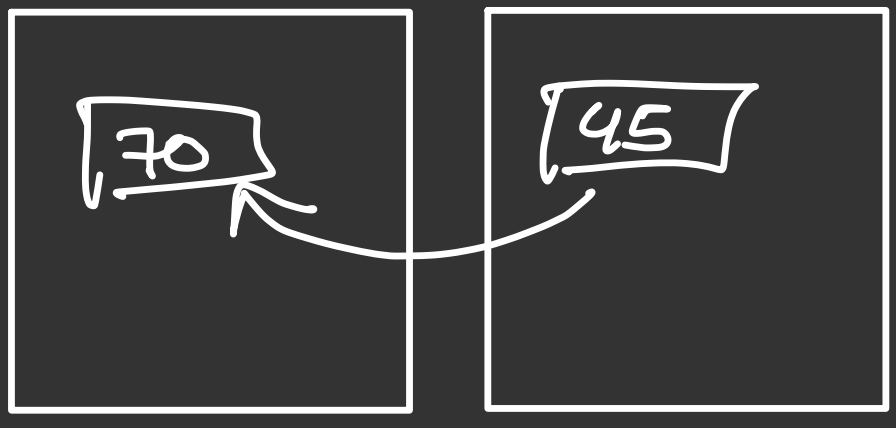
☆ PRACTICE OOPS MCQ in Code Studio, link in bookmarks.

Another way?

Copy Assignment Operator "="

↳ after both the objects have been created

```
Hero a(70);
Hero b(45);
a=b;
```



⇒ a.health = b.health;
and other data members also.

Destructor

(memory deallocation)

→ automatically created
→ name is same as class name

→ no return type
→ no input parameter

✱ Destructor is automatically called for static objects

While, for dynamically created objects you have to call destructor manually.

Hero *h1 = new Hero;

delete h1; → manually calling destructor.

Const Keyword

→ Whenever a const keyword is attached with any method, variable or the object of a class, it prevents it to modify its data item value.

example:

```
Const int var = 5;
```

* If we want to keep altering values →

```
int n = 9;
int *y = &n;
```

→ const variable
→ pointer pointing to const value
→ const pointer pointing to value
→ const pointer, const value

Initializer List

→ used in initializing the datas members of the class.

example:

```
class Point
{
    private:
        int x;
        int y;
    public:
        Point(int i=0, int j=0)
            : x(i), y(j) {}
};
```

This is same as ⟶

```
Point (int i = 0, int j = 0)
{
    x = i;
    y = j;
}
```

Static Keyword

→ we use it to create such a data member that can be directly accessed with creating the object as it belongs to the class.

Declaration →

```
static int timetoComplete;
```

Initialization after the class →

```
int Hero::timetoComplete = 5;
```

- Hero → class name
- :: → scope resolution operator
- timetoComplete → field / data member name.

Print →

cout << Hero::timetoComplete << endl;

Static Functions

→ no need to create object call it using class name

→ they dont use "this" keyword as no object

→ They can only access static member

→ static.cpp

Lecture 43

Encapsulation

→ wrapping of data members and functions into a single unit / entity.

class ←

data members → properties

functions → methods / behaviours

Fully encapsulated class ?

→ all data members have to private.

Advantages :

→ Information hiding

or Data Hiding

→ We can make class Read only
(Getter ✓, No setter)

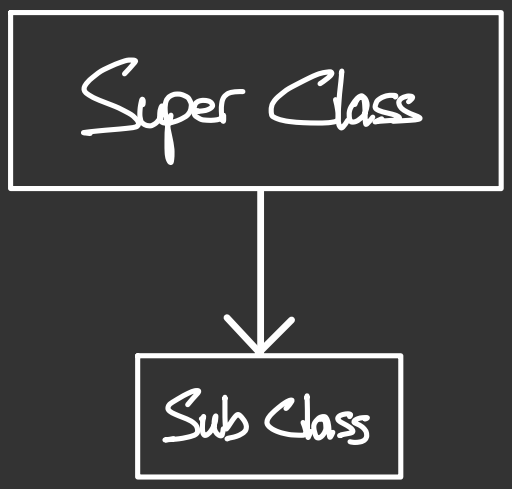
→ Code Reusability

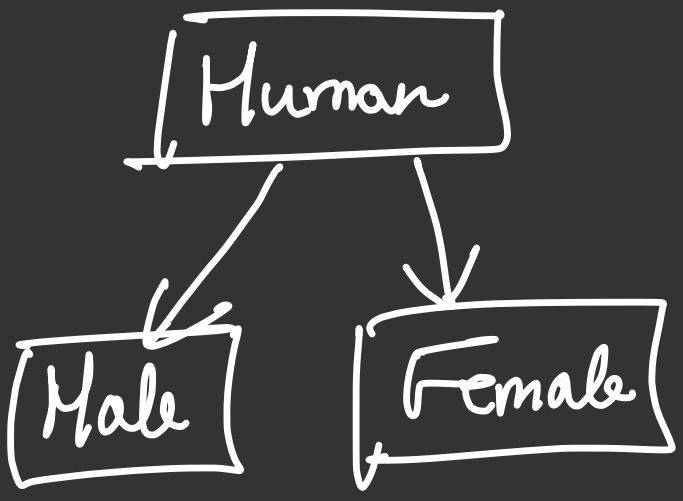
→ helps in unit testing

→ encapsulation.cpp

Inheritance

↳ The capability of a class to derive properties and characteristics from another class is called Inheritance.





⟶ parent / super class

⟶ child / sup class

Syntax:

class _____ : _____ _____

↑ child name — mode — parent class

☆ Child class will have its data members and functions along with of the parent class.

(mode of inheritance)
While declaring child class

Parent Class	Public derivation	Private derivation	Protected derivations
Public member	Public	Private	Protected
Private member	Not inherited	Not inherited	Not inherited
Protected member	Protected	Private	Protected

Super class access specifier → properties
→ data members

☆ Private data members of any class cannot be inherited.

☆ Protected

↳ can be accessed in the class, in the child class and nowhere else.

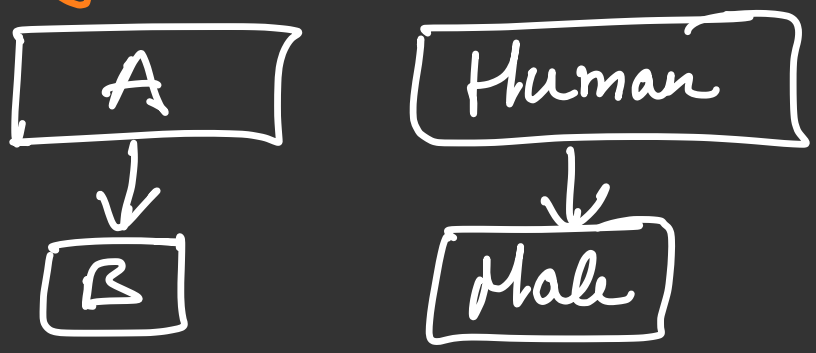
⟶ can access using getter and setter.

⟶ so cannot be accessed from main directly

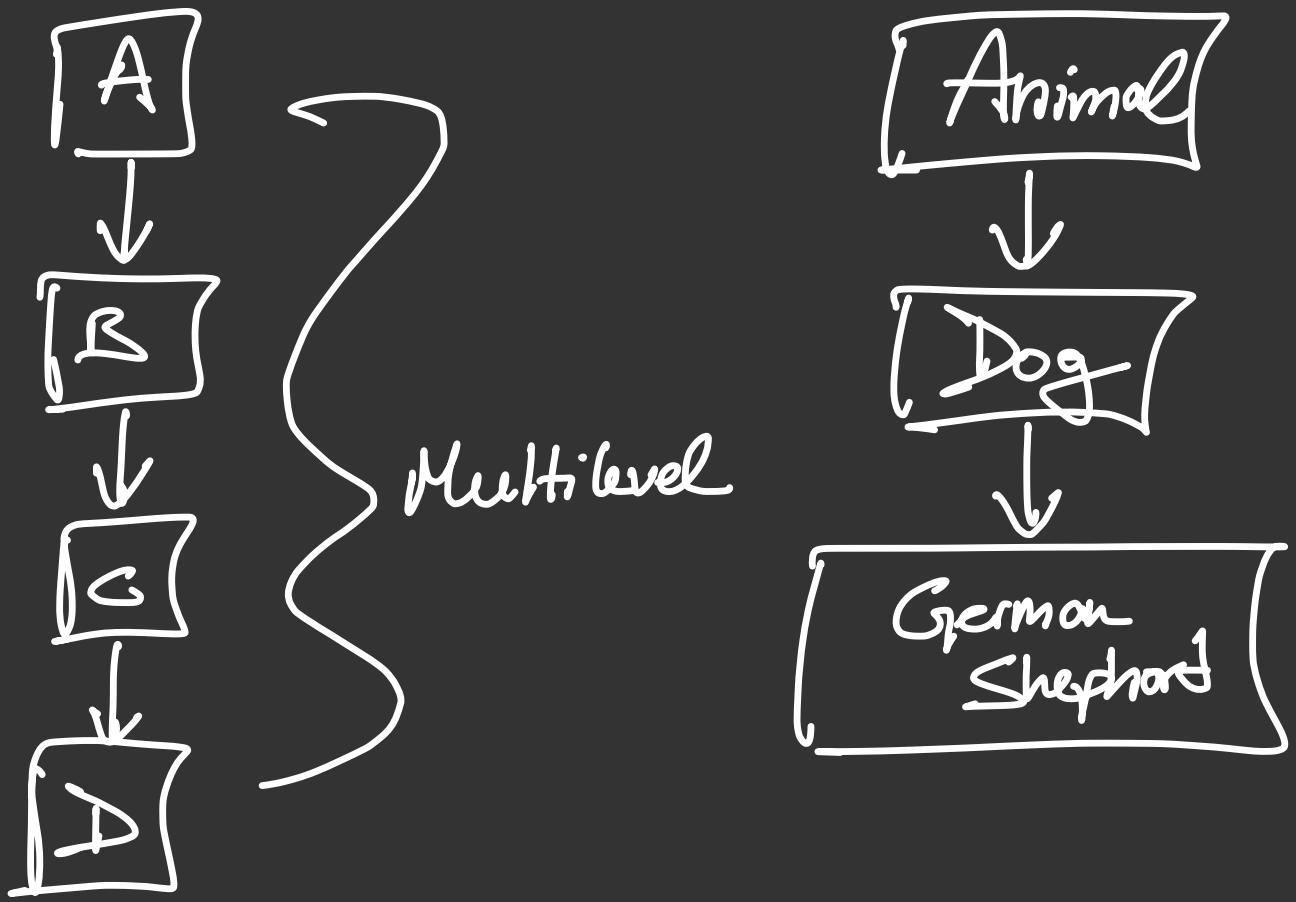
⟶ inheritance – protected.cpp

Types of Inheritance

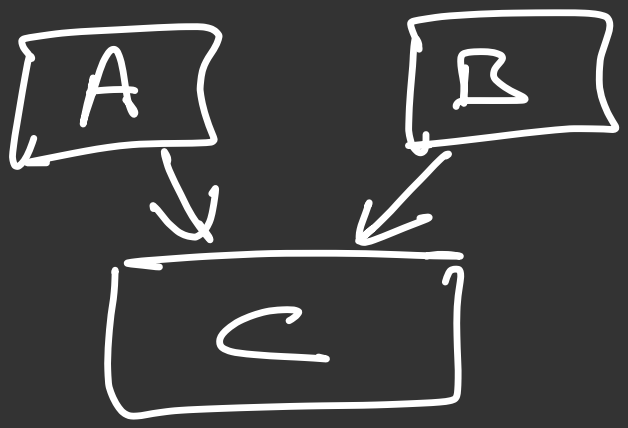
(1) Single Inheritance



(2) Multilevel Inheritance



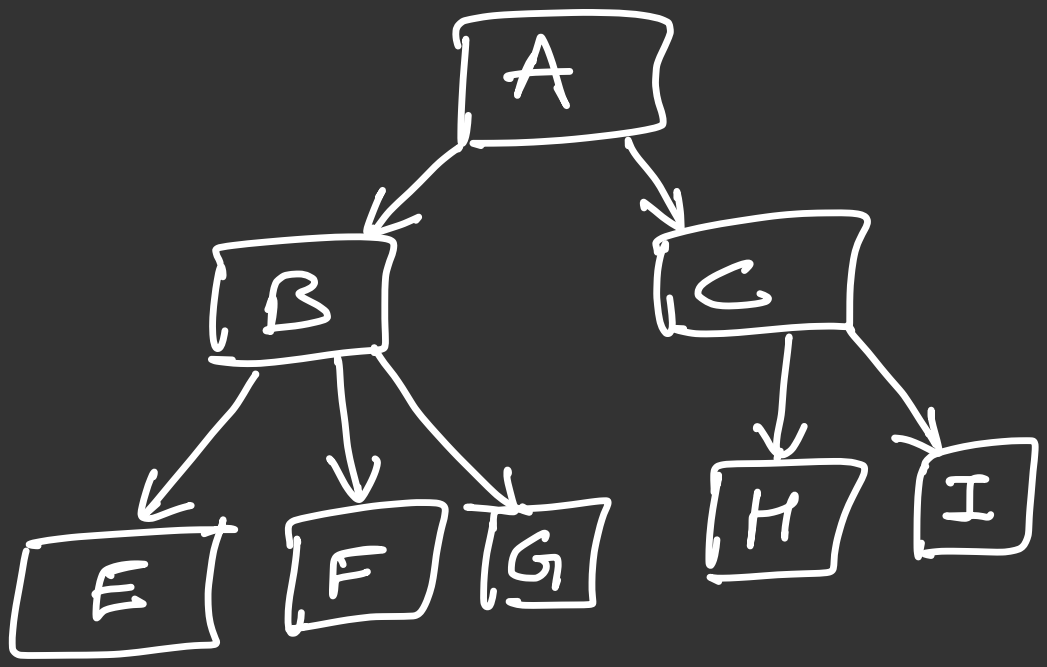
3 Multiple Inheritance



→ multiple_inheritance.cpp

4 Hierarchical Inheritance

↳ one class serve as parent class for more than one class



(5) Hybrid Inheritance

⟶ combination of more than one type of inheritance.

Inheritance Ambiguity

☆ When both class A and B let's say have a same function void abc() and class C inherits both of them, then while calling for that function via Class C object we use Scope resolution operator (::)

```
C cl;
cl.A::abc();
cl.B::abc();
```

⟶ inheritance_ambiguity.cpp

Polymorphism

- poly → many
- morphism → forms

→ existing in many forms

→ someone's
- father
- husband
- brother
- son

Two types

- Compile time polymorphism
- Run time polymorphism

Compile Time polymorphism (Static)

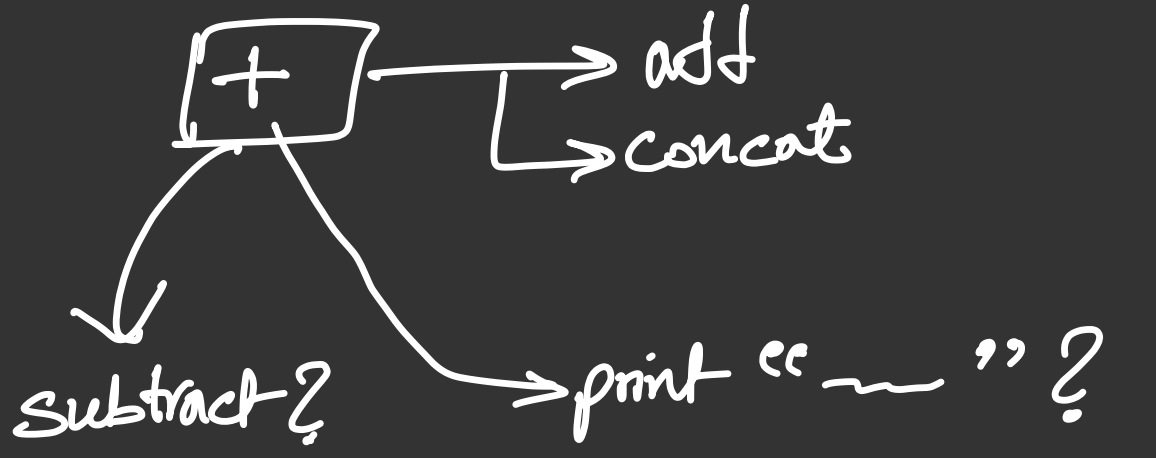
- function overloading
- operator overloading

a> Function Overloading

* for function overloading, you need to change the input parameters, just changing the return type will not work.

→ function_overloading.cpp

b> Operator overloading



Not all operators can be overloaded

Syntax ⟶

```
void operator+ ( )
{
    ==========
}
```

second port is the input parameter

So, when you pass

```
c1;
c2;
c1 + c2;
```

c1 → can be accessed using 'this' as the current object of the class

c2 → second port

Runtime polymorphism

(Dynamic)

↳ Method / Function Overriding

Method Overriding

Rules ⟶

1. Both the methods should have the same name.
2. Same parameters.
3. Possible through Inheritance only.

⟶ method_overriding.cpp

Abstraction

↳ Implementation hiding

⟶ showing only essential information

⟶ Using Access Specifiers

- public
- private
- protected

Advantages ⟶

1. Only you can make changes to your data or function, and no one else can. Thus, makes the application secure

2. Increases reusability of code. (avoids duplication)

Go through OOPs code
studio reference material
+
Questions (MCQs)
+
A cheatsheet in Bookmark
(mindmap)
especially the Misc