Note: Struct → Members are private by default
Class → Members are public by default } only diff

"instance of struct is structure variable"

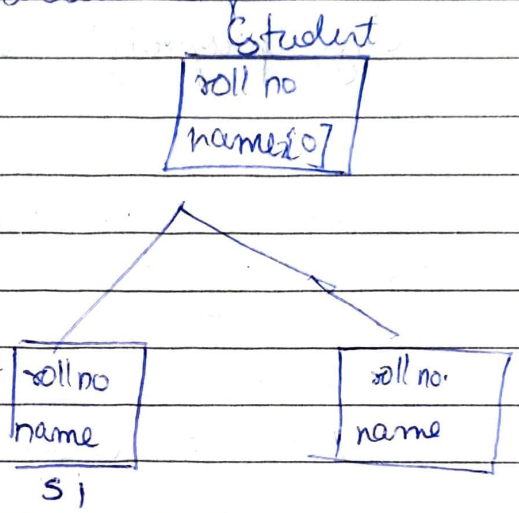
Object Oriented Programming

1. Class – blueprint that defines properties and functions of object. It is a group of similiar objects (Instance of class is object

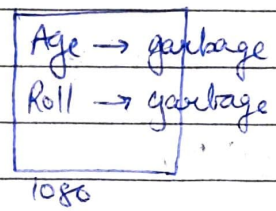
2. Object → specific properties & f^n

```
class Student {
public:
    int roll no;
    char name[20];
};
int main{
    Student S1;   // Static
    student S2;
    Student *S3 = new student;   // Dynamic
}
```

→ object / Compare with int a;



compare with int *p = new a;



S1. rollno = 12;
(*S3). rollno = 30;
// S3 → rollno = 30;

3. Syntax of Class

```
Class class_name{
    Access Modifier:
        Data members;          // data variable
        Member functions();    // fn used to manipulate
}
```

are set of O/R that may be applied to objects of that class

s1.age = 24; → set the properties

- Memory is only allocated to the variable of the when object is created.
- But memory allocated to the f^n only once when class is created

i.e. 1) Only one copy of function for all the objects of class

2) ~~Single~~ Every obj has an indival copy of all the variables of class

4. Access
private: –
public:
// member
protected:

obj. fn

5. Getters and setters

private member of class are not accessible the class.

Getters → access data members of the object

~~Setters~~ → accessor fn, read the private fn

setters → change the data members of o

mutator f^n, set the private fn

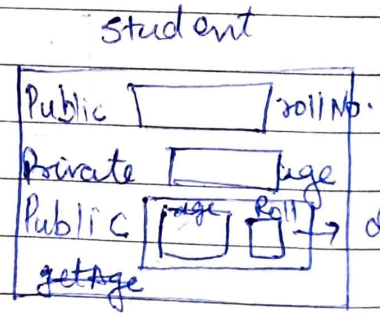
we can read & write private proper

```
class Student{
public:
int rollNo;
~~vo~~
private:
int age;

public:
void display(){
    cout<<age << " " << rollNo;
}
int getAge(){
    return age;
}
void setAge(int a){
    age = a;
}
```

→ will be returned in s1

// a=20
age → 20



int main(){
Student S1;
Student * S2 = new student;

S1 setAge(20);
S1 display();

O/P → 20, garbage-value

7 Constructors visibility and accessibility of class members (variables and functions) within a class or from outside the class
that initialize the properties with their special values.

(1) na consider student S1; → | Roll age | inside
S1 student();
↳ is call

object of student class is creat. whose name is S1 inside memory

1) no same name as class
2) No return type
3) No I/P arguments
4) Member fn
5) Invoked automatically

Default	Parameterized
- accepts no. parameter	- takes arguments
- public member	- to declaring constructor with argument like default constructor
- compiler automatically supplies a default i.e when when we make for every class constructor is automatically created at time	

2. copy constructor copy is created at time of creation
$s2 = s1$

3. copy assignment → both two objects are already created and if we have to copy values

4. Destructor
1) same as our class
2) no return type
3) No I/P argument
memory is deallocated when destructor is called

```
Class Student{

public:
int rollNumber;

private
int age;

public:
~Student(){                                  // Destructor
  cout << "Destructor called!" << endl;

Student (int rollNumber){                    // Parameterized constructor
      cout << "Constructor 2 called" << endl
        this.rollNo = rollNo;
}

Student (int a, int r){
      cout << "Const 3 is called" << endl;
        this.age = a;
        this.rollNumber = r;
}
```

```
Student() {   // Default
    cout<< "constructor 1 is called" <<endl;
}
```

```
Student s1;              // Default (const 1)
Student s2(101);         // ~~cons~~ Parameterized (const 2)
Student s3(20,102);      // Parameterized (const 3)
Student s4 (s3);         // copy const.
s1= s2;                  // Assignment o/p
                         No constructor is called
Student s5= s4;
```

→ new object is created

- When you create constructor p.i.e. parameterized constructor then the default constructor does not work. If we want a constructor for no arguments we have to create a constructor explicitly

This → holds address of current object, pointer variable

```
Student s1,(10,20);

Student(int a; int r) {
    age = a;
    rollnumo = r;
}
```

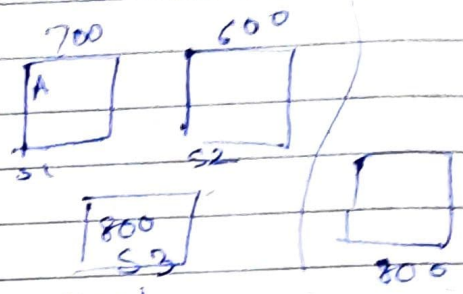
when s1 is created address of s1 is stored in memory to this,

age = a → store value of a in address of s1 in age

4) Destructor

Student s1 (10, 1001);
Student s2 (20, 2001);
Student *s3 = new Student (30, 3001);

s2 = s1;
*s3 = s1;
s2 = *s3;



O/P → Destructor called → for 1
Destructor called → s2
s3 → pointer variable, we don't destructor

Static variables → destructor is called
Dynamic → delete keyword is used

delete s3;
↳ Destructor is called.

5) Virtual Destructor
↳ gaurantees that the obj of derived class is destructed properly

→ destructor is called in reversed order

→ copy Constructor can be private

1. Using Friend function
2. Using singleton design pattern
3. Name Constructor Idom

Declare all the class constructor in private or protected section and access them by creating public static function

eg

```
Class Point{
    ~~public~~ private:
        float x1, y1;
        Point( float x, float y){
            x1=x;
            y1=y;
        };

    public:
        static Point Polar (float, float); //
                Polar Coordinates
        static Point ~~Polar~~ Rect ( float, float);
        ~~to~~ // Rectangular
};
```

write fn for both

1. Now if we had created only one constructor of the type Point P = Point (5.7, 1.2); → here there would have been ambiguity which constructor to be called
2. So to avoid this we use constructor idom

Constructors → ~~are~~

There is no concept of virtual constructor because whenever we create a object, a constructor is ~~due~~ automatically invoked.

Virtual keyword is used when we have base class pointer to derived class object.

As constructor is ~~directly~~ automatically invoked virtual constructor is not ~~used~~ present

Virtual destructor

Now consider

A
|
B
|
C

we want to delete C, but if we call destructor the base class gets deleted. Therefore the classes inherited have no use

Using virtual keyword we access the child class and delete the class that we want

OOP-2

① Shallow Copy → 1) pointer is copied but not the memory it is copying to. It means that original object and created object points to same memory

2) Since both objects will reference to the same memory location, change made by one will be reflected by the changes made in other copy. ③ Default constructor does shallow copy.

② Deep Copy

1) New copy of the array is created
2) Destructor is used to deallocate the memory
3) → new memory is created and then the contents are copied

```
int age;
char *name;
public:
Student (int age, char *name){
    this → name = name; // Shallow
    this → name = new char [strlen(name)+1];
                                    // Deep
    strcpy (this→name, name);
}
```

Shallow

Shallow Consider name → 980

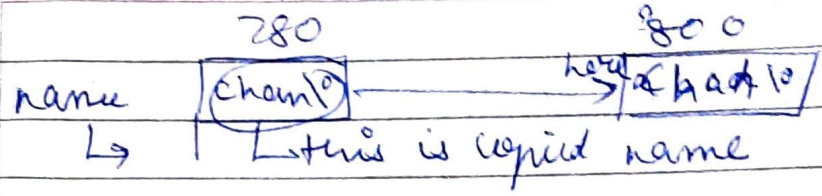
980 → is copied to this

980
name

Deep → Consider * name = "Clan" name = 980

name = new char [5] → new array is created

3) Assignment :- already initialized object is assig
new value from another existing Object



2) Copy Constructor it does shallow copy
It used to create a copy of an alread
existing Object of class type

Student & S1 (S2); → value of S1 that is passed argument will go here
↳ this will have S1

Student [Student const & S1] { → ensure that original copy is unchange
S1.age ← this→age = S.age; → S2.age
S1.name // this→name = S.name // shallow copy
S2.name

// Deep
this→name = new char [strlen (S.name)+1];
Strcpy (this→name, S.name);
}

3) Initialisation list → at time of memory alloc
it initialises the list.
Class student {
public:
int age;
const int rollNumber ; // Const type data member
once we give value it won't be abli
change
int &x; // age reference variable

Page No.

data member we need to initialise

Student (int x, int age) : rollNumber(x), age(age)
x (this → age) }

argument

normal variables can be initialised

}

This will initalise the list during memory alocation i.e when ~~so~~ rollnum block is allocated, it will intialise rollNumber to it

(3) Constant function
- They can be called through const object
constant f^n are those which don't change any property of current.

(4) Static Members (class level scope)
(1) that belongs to a class and it is
(2) not created seperably for every object.
(3) If we want to access it we can access through.
(3) Mention accessed it using (::) → scope resolution operator
(4) Its intialisation is outside class

~~St stotal stu~~

Syntax. → static class_name :: variable name;

Static f^n → is independent of any class object

eg total student we don't want total student to be declared for every object so we mark it as static

(5) Space allocated is for limited lifetime
→ Cannot be accessed ~~we~~ using scope resolution operator

(6) They are class specific and not called on object

Operator overloading

- we can specify more than one defn of an o/p in same scope, → called as o/p over

(1) Only for user defined operators (objects, structures)

(2) We can extend the functionality of any preexisting operator such that it works for our user-defined classes

(3) Nesting in post increment is not allowed

(4) Nesting is allowed in +=

Function Overload

(1) Same name, diff^n parameters i.e. different types of argument or diff^n arguments

Ambiguity

(1) Type Conversion eg add(~~int~~ float), add(int)
(2) Function with default arguments → add(int), add(int a)
(3) Function passed by reference
↳ add(int)
add(int(4)
→ add(int a, int b=2)

Page No.
Date

Static keyword

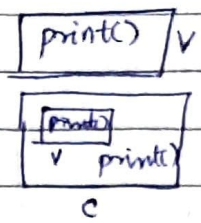
1. They are called ~~wha~~ without creating objects of class
2. Initialised once and a single copy to be shared by all instances of the class and it can be ~~ac~~ accessed directly by class name & does not need object.
3. They donot contribute in size of an object

Polymorphism

Pillar / Components of OOPs

Compile time

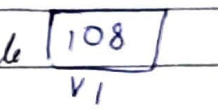
vehicle v;
car c;



v.print(); // vehicle
c.print(); // car

This will be divided at compile time

vehicle *v1 = new Vehicle; // vehicle



vehile *v2;
*v2 = &c // it will be able to access those properties which are present in v2 class

v1 → print(); // vehicle
v2 → print(); // ~~car~~ vehicle

Run Time → by virtual and pointer

- ~~Using~~ Now if we want to print ~~or~~ access the same fn from ~~base~~ ~~car~~ derived class using base class, we can do it by using virtual function
- If we use virtual keyword in vehicle class we ask compiler to execute the program at runtime.
- So during ~~com~~ runtime, compiler checks ~~wh~~ ~~to~~ that object belongs to which class. If it belongs to vehicle class, ~~vehicle gets printed~~, ~~if its base class is~~ and it points derived class so the fn of derived class is executed

② Class car : public vehicle{
void print{
cout << " car " << " ";
}
}

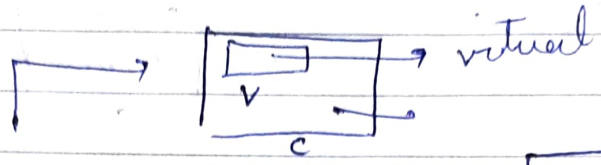
base_class_name

by

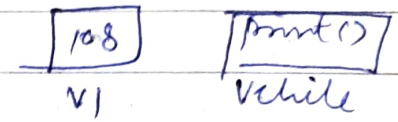
of

① Class vehicle{
virtual void print()
cout << " vehicle ";
}

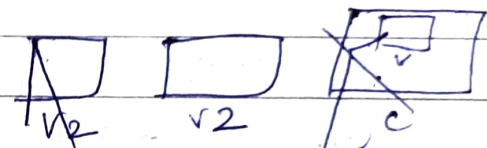
↳ ~~No car, vehicle v; car c;~~



~~vehicle * v1 = new vehicle()~~



~~vehicle * v2 = & c;~~



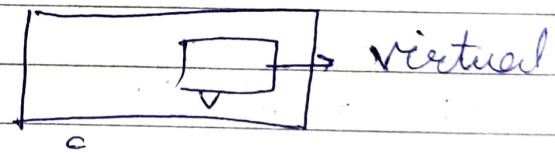
~~① virtual is ve~~

②

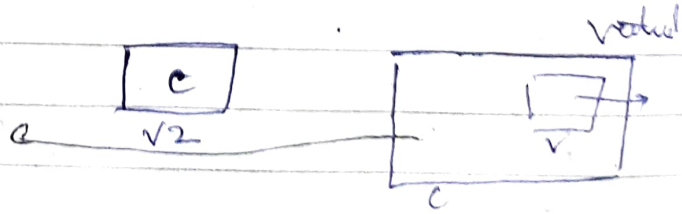
① Implementing all virtual fn

Now,

vehicle v;
car c;



Vehicle * v1 = new Vehicle();
vehicle * v2 = & c;



so, print fn of c is printed

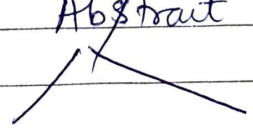
v1 → print(); // Vehicle
v2 → print(); // Car

(7) Syntx

* A bas fn can be accessed with scope resolution operator if it ~~vorches~~

(1) Pure Virtual Functions :

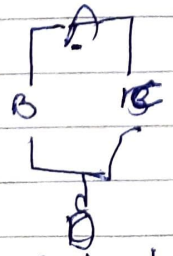
- Can ~~&~~ be ~~Emy~~ private and overridden
- Consider a class Employee, it has derived classes HR, Manager, etc. Every sector has diff^n salary pay. So we cannot write Salary fn of in employee so we declare it as ~~virtual = 0~~ pure virtual function.
- Once if we write pure virtual function, ~~abstra~~ it becomes abstract class and hence we cannot ~~a~~ create class ~~for~~ object for it
- Because when we create object we have to intialised it, as it a abstract class we cannot create object.
- ~~If we cor~~ Abstract - declared by assign 0



Vehicle
|
Car

Implement all fn by

Become abstract
↳ Comment Out / delete the same fn



(2) So here we declare B & C virtual, so ~~dec~~ A will be implemented directly

(1) here A is implemented twice for D. This creates ambiguity

Operator Overloding

↳ Compile time polymorism

1. ~~a + b~~ a + b, '+' operator is used to add two predefined datatype.
2. If we want to add / or use operators for e.g. i.e. add / sub / multiply, d 2 fraction we use operator overloading
3. used to perform operation on user defined datatypes

- Can virtual fn be inline
  Yes, only when compiler knows exact class of the object rather than pointer or ref to object

- Abstract Class
  1. Can have constructors
  2. In C++ ; an interface can be made by making all methods as pure virtual

- Pure Virtual destructor

deriving properties from another class

Page No.
Date

Inheritance

(+) Syntax class derived name : : Access specifier base_class_name

① When base class is privately inherited by derived class,
→ ① public members becomes private members of derived class
② public member of base are not accessible ~~to~~ by the objects of derived class
③ Only member functions are accessible

② When base class is publically inherited by derived class
① ~~pub~~ both ~~data~~members and member functions are accessible

③ Protected
Accessible ~~to~~ within its own class and class immediately derived after it

(+) ~~Exo~~ Example https://chat.openai.com/share/f1accc46-6eac-4ecb-a553-4f233a03115f
① Multilevel
Grand Father } gives property to father
↓
Father } father gives property to child
↓
Child

As child cannot give property to father.
So we can conclude that inheritance is one sided i.e. from base class to derived class

types CN

② Multiple

Mother Father

child

Child has properties of both mom & dad

③ Single

Father

↓

Child

④ Hybrid

Student

Exam Assignment Coding

Result

⑤ Heraclial

Human

Men Women Transgender

— Disadvantages

① Slow

② Improper use lead to wrong solution

③ Often data members are left unused → leads to memory wastage

④ change in base class affects in derived class

— Advantages

① Reusability

Page No.
Date

Sealed keyword

1) When we want that a method should only work for a particular class we use seal keyword. It & when we seal a class or function it cannot be inherited
2) It cannot be applied on ~~seal~~ ~~n~~ static members

- Function / overridding
when method of derived class is overrude

Difference betn overriding and new

① Overriding
It extends method of base class

② New
hides the method of base class

- HAS-A relationship
↳ one class defines another class as any entity reference.
i.e. one class is used as an object or member in other class

```
Class Address{
    public: string city, state;
    Address ( string city , string state){
            this → city = city
            this → state = state
    }
}
```

- Friend

If a fn is declared friend fn protected and private members can be accessed using the fn

1. Not in scope of class that is it just gives permission to access the private members of the class
2. Cannot be called from object of the class
3. Can be invoked without any object
4. Usually objects as arguments
5. Can be declared public and private members of class
6. Cannot access the members directly by name
   (argument_name . member_name)

HAS-A Contn

```
Class Employee {
    private: Address *address;
    public: int id;

    Employee (int id, Address* address) {
        this -> id = id;
        this -> address = address;
    }
```

O/P:⇒