OOPS → programming paradigm

DATE: __/__/____
PAGE: _____

- **Classes & objects** → consumes space
  - → creating object statically & → creating object dynamically

- **Access modifiers** → Data Hiding
  - → Public ↳ accessible for only friend function, member function
  - → private ↳ accessible for all
  - → protected ↳ accessible for member func^n, friend func^n, derived class (depending upon mode of inheritance).

- **Getters & setters** → mutator func^n
  ↳ accessor func^n

- **Scope resolution operator (::)**
  - → To access global variables.
  - → To define a function outside a class.
  - → To access class's static variable.

- **Constructors ()**

  Student s1; ← constructor.

  ↳ s1. Student ()

  → Construction is only called once for every object in its lifetime.

  ① Default constructor

  ② Parameterized Constructor

  Note: Once we create our own constructor, the compiler's implicit default constructor is overshadowed.

THIS keyword

↳ It is a pointer that points to the object for which the func^n was called.

Contains address of object

this → a = a
this → b = b
→ (*this).a = a
(*this).b = b

Good Write

③ Copy constructors

can only be called at time of creation.

Student s2(s1);
↳ s2.Student(s1)

Note:
default copy constructor makes shallow copy

Copy assignment operator (=):

Student s1(10, 1001); Student s2(20, 2001)

→ shallow copy

s2 = s1;

↳ copy the value of one object to another at a later time.

Shallow & deep copy

Shallow copy

→ faster.

→ changes made to new copies are reflected to original.

→ pointer will be copied not the memory it points to.

Ex.

```
class Student{
    int age;
    char *name;
public:
    Student(int age, char* name)
    {
        this->age = age;
        this->name = name;
    }
};
```

Shallow copy

Deep copy

→ slower.

→ changes are not reflected.

→ create a copy of the memory the pointer points to

Ex.

```
Student(int age, char *name)
{
    this->age = age;
    this->name = new char[strlen(name)+1];
    strcpy(this->name, name);
}
```

Good Write

Note :- We can also use a string rather than a char, it will simply do deep copy & work fine.

DATE: __/__/____
PAGE: _____

→ Making own copy constructor

pass by reference

```
Student C Student (cons & s)
{
    this → age = age;
    this → name = new char [strlen(s.name)+1]
    strcpy (this → name, s.name);
}
```

→ Initialisation List

It enables us to set a attribute that can only be set once for a particular object & latter cannot be changed.

```
class Student
{
public:
    int age;
    const int rollNumber;
    int & x;

    Student (int r, int age): rollNumber(r), age(age),
        x (this → age) {

    }
};
```

this → age

argument

→ Constant Functions → These functions which doesn't change any property of current object.

- Only constant objects can invoke constant functions.

```
int getStudentName() const
{
    return name;
}
```

Student const S2;

↓

Can only use constant functions.

Good Write

→ Static Members & Static function

★ we can define class members static using static keyword.

★ When we declare a member of a class as static it means no matter how many objects are of the class are created, there is only one copy of static member.

★ A static member is shared by all objects of class.

Ex:-

```
class Student {
    static int totalStudent;
    :
    Static int getTotalStudent()
    {
        return totalStudent;
    }
};

int Student :: totalStudent = 0; // Initialisation.

main()
{
    :
    :
    Cout << Student :: getTotalStudent() << endl;
}
```

→ Static member function.

↳ Can only be done outside class.

★ A static function → Independent of any class object.
- ↳ Can be invoked even if no object exist.
- ↳ Can only use static properties
- Does not have (this keyword.)

Good Write

4 Pillers of OOPS

DATE: __/__/____
PAGE: _____

① Encapsulation → Implementation level process.

It refers to combining data along with the methods operating on that data into a single unit called class.

Advantages:

→ Security of data
↳ Protects an object from unwanted access by clients.
→ It allows access to a level without revealing complex details below that level.
→ Simplifies maintenance ↳ since code is reused.
↳ No redundancy.

★ Access specifiers facilitate Data hiding.
★ Implemented using access specifiers.
★ Data is hidden using methods of getters & setters.

② Abstraction → Design level process / Interface level process

↳ Hiding

↳ It refers to providing only essential info. about the data to the outside world, hiding the background details or Implementation.

★ Can implement using abstract class & interfaces.
★ Objects that help to perform abstraction are encapsulated.

Types of abstraction → Data abstraction
→ Control abstraction.

Note: One type of abstraction in C++ can be header files

Advantages:
① Avoids code duplication, & Increases reusability.
② Can change internal implementation of class independently without affecting the user.
③ Increase security
④ Increase complexity & redundancy.

Good Write

→ Reusability of code

DATE: __/__/____
PAGE: _____

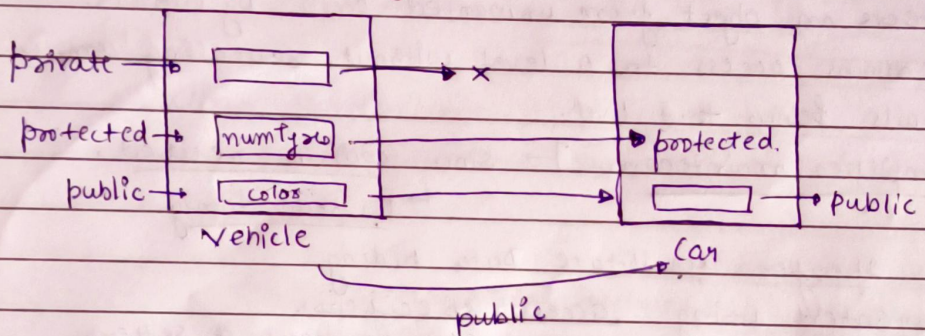
③ Inheritance → capability of a class to inherit the properties of some other class.

Example:

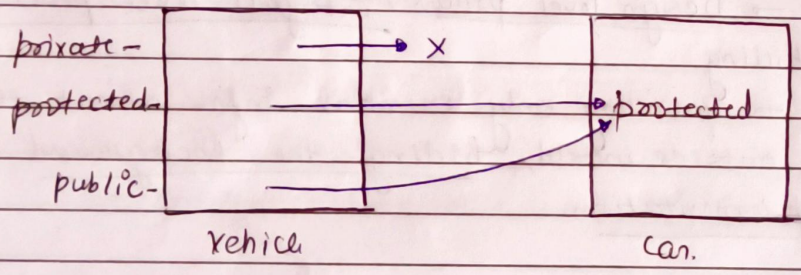
```
class Car: access_specifier Vehicle {
};
```

Car → derived class Name
Vehicle → Base class Name

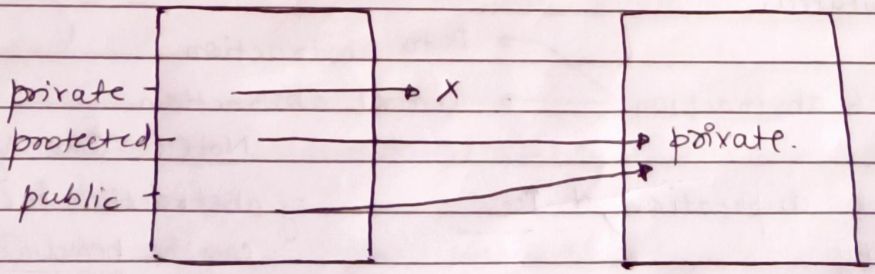
→ Inheriting publicly



→ Inheriting protected



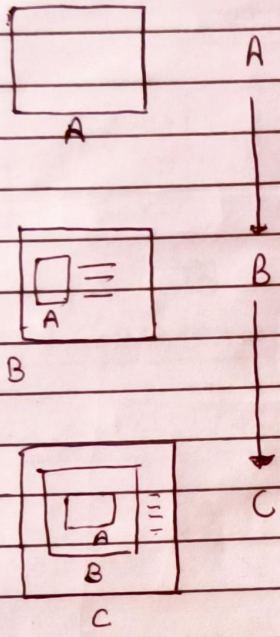
→ Inheriting private



Good Write

DATE: __/__/____
PAGE: _____

Order of constructor & destructor call



A a; → A()

B b; → A()
B()

class object
C c; → A() → B() → C()

constructor of parent class is called implicitly on base class.

Order of call.

Destructor are called in reverse order.
↳ Destructor of derived class is called first.

Note: This implicit call is always for default constructor.

* In case of calling a parameterised constructor of base class from derived class.

Ex:

```
class Vehicle
{
private:
    int maxSpeed;
public:
    Vehicle (int z)
    {
        maxSpeed = z;
    }
};
```

```
class Car : public Vehicle
{
public;
    Car (int x) : Vehicle (x){
    }
};
```

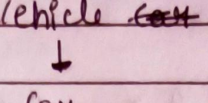
```
class Hondcity : public Car
{
public:
    Hondacity (int m) : Car (m){ .... }
};
```

```
main()
{
    Hondacity h(50);
}
```

Good Write

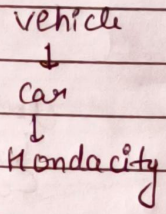
→ Types of Inheritance

① Single

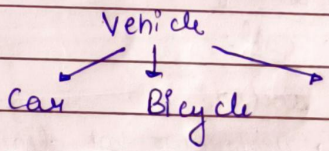



```
class Car: public Vehicle
{
};
```

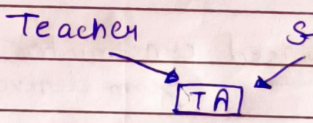
② Multilevel Inheritance



③ Hirarchical Inheritance



④ Multiple Inheritance **




```
class TA: public Teacher, public Student
{
};
```

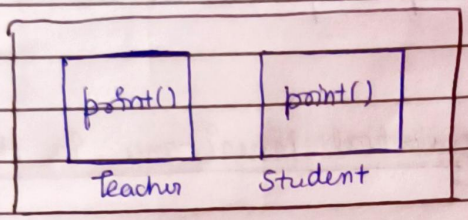
TA a ⟶ order of construction call.
Teacher()
Student()
TA()

Good Write

DATE: __/__/____
PAGE: _____

Ambiguity

Suppose both Teacher & Student have a func^n with same name.

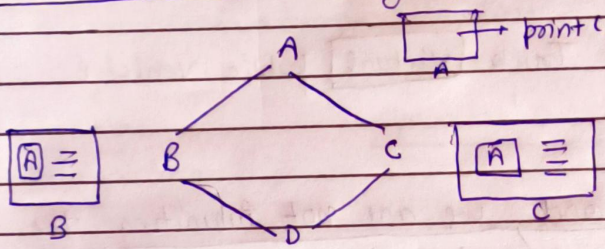


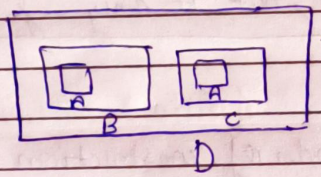
TA

```
TA a;
a.Student::point();
```
↳ calling the point func^n of Student class.

⑤ Hybrid Inheritance → More than 1 type of inheritance.

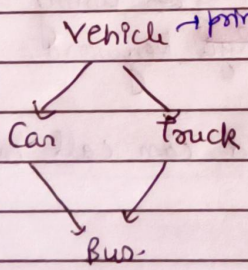
Diamond problem / Ambiguity → It occurs when two superclasses of a class have a common base class.





→ There are two instances of A in D. So ambiguity will arise while accessing.

Ex:



Order of construction call

Bus b →
1. Vehicle()
2. Car()
3. Vehicle()
4. Truck()
5. Bus()

b.print() ; → will give error.

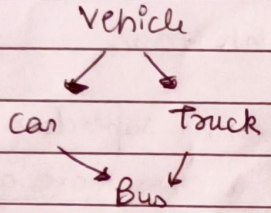
Resolving ambiguity → b.car::print()

DATE: __/__/____
PAGE: _____

→ One more issue still present

We don't require multiple copies, but we require only 1 copy.

→ We will use concept of virtual inheritance in this case:-




```
class Car: virtual public Vehicle {
};
class Truck: virtual public Vehicle {
};
```

Basically by using the virtual keyword, we are not inheriting the entire values from base class, but it only maintains a pointer to them.

Bus b → Vehicle (Order of construction call)
↳ after using virtual keyword.
Car
Truck
Bus

Note: After being virtual now child can call constructor of it's grandparent class.

Good Write

→ more than 1 form.

Polymorphism

DATE: __/__/____
PAGE: _____

↳ ability of a message to be displayed in more than 1 forms.

Behavior of the same object or function is different in different contexts.

Polymorphism

Compile time
→ Static polymorphism
→ Static Binding / Early Binding
→ A call to an overriden method is resolved at compile time

Function overloading

Operator overloading

Run time / Dynamic Binding / late Binding
→ A call to an overriden function is resolved at runtime.

method overriding

Virtual function

→ Compile time

① Function overloading

There are multiple functions in a class with same name but different parameters.
A call to an overloaded func^n is resolved at compile time.

Note: Functions having only difference in return type cannot be overloaded.

When two or more functions have same name but differ in no. of arguments, or type of argument, then it is function overloading.

Good Write

② Operator overloading

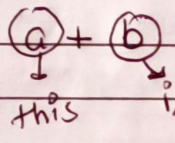
When an operator is updated to be used for user-defined data types, this is known as operator overloading.

* We assign additional tasks to operators without altering the actual meaning of operation.

* To use operator overloading → at least 1 operator must be user defined data type.

Note:- . , :: , typeid , size , .* , ?: cannot be overloaded.
= , & cannot be overloaded.

→ Overloading Binary operators




```
return-type operator + (i/p)
{
;
}
```

Example:
```
Point operator + (Point const &P2)
{
    int newx = x + P2.x;
    int newy = y + P2.y;
    int newz = z + P2.z;
    Point Pnew(newx, newy, newz);
    return Pnew;
}
```

→ Overloading Unary operators

DATE: __/__/____
PAGE: _____

(i) ++ (pre increment) → nesting is allowed.

++ P1
↳ this

Example:

```
Point & operator ++ ()
{
    x = x+1;
    y = y+1;
    z = z+1;
    return *this;
}
```

→ pass by reference
↳ for avoiding copy creation of buffer.

(ii) ++ → post increment → nesting not allowed.

P1++

```
return_type operator ++ (int)
{
    :
}
```

Example:

```
Point operator ++ (int)
{
    x = x+1;
    y = y+1;
    z = z+1;
    Point P(x, y, z);
    return P;
}
```

→ Runtime Polymorphism:

↓

Method Overriding (using Virtual functions)

DATE: __/__/____
PAGE: _____

In function overriding, we give a new definition to the base class method or function in the derived class.

Note: we can point base class pointer to child class object but vice-versa is not true.

↳ But base class pointer can only access properties of base class.

→ By default binding of function is done at compile time

Virtual function

↳ If we declare the function in base class as virtual then, the binding is done at runtime.

Example:

```
class Vehicle{
public:
void print()
{
  cout << "NA" <<endl;
}

virtual void show()
{
  cout << " This is a vehicle" <<endl;
}
};
```

```
class Car: public Vehicle
{
public:
  void print()
  {
    cout << " Car <<endl;
  }

  void show()
  {
    cout << " This is a Car" <<endl;
  }
};
```

```
main()
{
  Vehicle * v = new Car;

  v → print(); ]→ calls vehicle class print ] → calls vehicle class print

  v → show(); ]→ calls car class show ]→ Virtual func" (Binded at runtime)

  Good Write
}
```

DATE: __/__/____
PAGE: _____

Pure virtual function & Abstract classes

→ Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called Abstract class.

→ A pure virtual function is a virtual function for which we don't provide any definition in the base class.
↳ If we don't override that function in the derived class, then derived class will also become abstract class.

Example:-

```
Class Base
{
    int x;
public:
    virtual void func() = 0;
    int getX() { return x; }
};
```

```
Class Derived : public Base
{
    int y;
public:
    void func()
    {
        cout << "func() called";
    }
};
```

```
Derived d;
d.func();
```
→ func called.

Note: A class is abstract if it has at least 1 pure virtual function.

* We cannot create objects of abstract classes.

* We can have pointers & references of abstract class.

```
Base *b = new Derived();
b->func();
```

* An Abstract class can have constructors.
* An Abstract class can also be defined using static keyword

```
Static ~~Struct~~ Shape Class
{
Good Write  virtual void Draw() = 0;
}
```

→ Friend Function

DATE: __/__/____
PAGE: _____

* If a functions is defined as a friend function, in a class, then it can access the protected & private data of a class.

* It can be → member of another class
  → A global function

Example:-

```
class Rectangle
{ private:
    int length;
  public:
    Rectangle()
    {
      length = 10;
    }
  friend int printLength(Rectangle);
};

int printLength(Rectangle b)
{
  b.length += 10;
  return b.length;
}

main()
{
  Rectangle b;
  cout << printLength(b) << endl;
};
```

→ can be declared in any section of class

DATE: __/__/____
PAGE: _____

Imp points regarding friend func"

* can be a normal function → without using an object.
* It is not in the scope of class, of which it is friend.
* A friend function cannot access the private & protected data members of the class directly. It needs to make use of class object & then access the member of using dot operator.

→ Friend class

↳ A friend class can access private & protected members of other class in which it is declared as friend.

Example:-

```
Class Creeks                    class CFGs
{                                {
  friend class CFGs;               statements;
};                               }
```

* Too much use of friend functions & classes lessens the value of encapsulation.
* Friendship is not mutual.
* Friendship is not inherited.

Demerits :-

* lessens encapsulation.
* Friend func" cannot do any runtime polymorphism in its members.

Merits :-

* Can act as a bridge b/w two classes by accessing their private data

Good Write

DATE: __/__/____
PAGE: _____

Que:- Can a constructor be virtual?

~~Constructor can be virtual.~~

Constructor cannot be virtual:-

Because:

(i) There is no virtual memory table present while calling the constructor. So construction of a virtual constructor is not possible.

(ii) Because the object is not created, virtual construction is impossible.

(iii) The compiler must know the type of object before creating it.

→ Virtual destructors

→ It is used to destroy an object through a base class pointer by calling derived destructors appropriately.

Virtual table

The compiler can't know which code will be executed by the $0 \rightarrow f()$ call at the compile time. Since it doesn't know what 0 points to. Hence, it would be best if we had a "virtual table", which is a table of function pointers.

Data types & sizes

- int → 4 bytes.
- Char → 1 byte.
- boolean → 1 byte
- float → 4 bytes.
- double → 8 bytes.

Good Write