OOPs

Polymorphism

compile time

fn overloading | operator overloading

↳ static binding.

Run time polymorphism

↓

virtual fn

↳ dynamic binding

→ by method overriding

Diff b/w

compile time poly	Run time poly
→ f(n) to be invoked is known at compile time.	→ f(n) to be invoked is known at run-time.
→ overloading, early / static binding.	→ overriding, late / dynamic binding.
→ Overloading :- f(n)'s with same name but diff arguments (diff number & type).	→ overriding :- fn's with same name and same no. of same arguments & type.
→ by fn / operator overloading	→ by virtual fn / pointers
→ fast execution as it is known at the compile time.	→ slow exectn as it is known at run time.
→ less flexible	→ more flexible

eg class shape {
public:
virtual void draw() {
cout << drawing
}
class Rect :: public shape {
void draw() {
cout << drawrect;
}
void main() { shape *s, sh;
s = &sh; s->draw (from
rect r;
s = r;
s->draw() -> (Derived)

Operator Overloading

to study.

Benefits of interfaces

- consistency & contract
- Modularity.
- ~~Poly~~ polymorphism.
- code Reusability.
- code Documentation :- gives the expected behaviour of classes.

e.g → click function in simple, toggle, radio buttons

→ In c++ there is no keyword "interface", so to declare an interface we use abstract classes with pure virtual functions.

→ Interface should not contain member data, attributes

→ should be declared public

→ use virtual when overriding & override to indicate explicit overriding

→ all methods must be implemented.

Abstraction

- implementation details are protected.
- programmer does not need to write low level code.
- ease & proper partitioning of the code

What is object oriented Programming.

↳ It is about creating objects that contain both data & functions.

while, procedural programming is about writing procedures or f(ns) that perform operations on the data

↳ OOP is faster & easier to execute.

↳ provides clear structure for the programs.

↳ helps to keep code "DRY", Don't repeat yourself, makes code easier to maintain modify & debug.

⇒ Class is a template for the object & object is an instance of a class.

When interface & when abstract class.

↳ when we have multiple inheritance & require a clear blueprint that has only design & no implementation then we go for interface.

↳ when we don't need multiple inheritance & require a mix of design plan & pre-implementation → abstract class

OOPs relating to the real world e.gs

→ Abstraction :→ sending data from bluetooth, we don't know how it connect with other phones or devices

→ Making a phone call.

→ Encapsulation :- In real-time → used for security purpose

→ e.g when we connect from blue-tooth we can access to send files to other phones but we cannot send SMS or call from their phone.

→ this is achieved by access specifiers

→ **Polymorphism** :- e.g we can turn on the phone by some button & we can turn off the phone by same button.

→ **Inheritance** :-

- Mobile → Samsung
  single level

- Mobile → Samsung → Samsung S5
  multi level

- Mobile → Samsung, apple

Why we need OOPs in programming language ?

1) Duplicate code is Bad.
2) code / requirement will always change.

advantage of OOPs

→ improved s/w development productivity.
→ improved s/w maintainability
→ faster development
→ lower cost development
→ higher quality software.

Disadvantage of OOPs

→ steep learning curve :- it is complex to create programs based on the interaction of objects.
↳ inheritance / polymorphism can be challenging to comprehend initially.
→ larger program size
→ Slower programs.
→ not suitable for all types of problems.

Q→ When we say that "x" language is OOP, then what is meant by that?

↳ It means that the language supports & facilitate the concept of OOPs, i.e objects are used to design applications & programs.

→ language which ~~have~~ are considered OOP, typically have built in support for these concepts.

→ Java :- strongly OOP.

→ C++ :- adds OOP support to the procedural programming language.

Class

→ members are by default private

→ instance is called object

→ declared using class keyword

→ NULL values are possible

→ inheritance is possible

Structure

→ members are by default public.

→ instance is called structure variable

→ declared using struct keyword

→ Null values are not possible

→ inheritance is possible

Similarities

⇒

→ Encapsulation :- Data & methods together.

→ Access modifiers :- public, pvt etc.

→ Data mems & mem fn.

→ constructors & Destructors

→ inheritance

Access Modifiers :- specifies the scope of a field, method

- **Private** :- only within the class.
- **Default** :- only within the package. if no access modifier is by default it is default.
- **Protected** :- with the package & outside the package only using child class.
- **Public** :- everywhere, within or outside package / class.

- If we make a constructor private, then we cannot make instance of a class from outside the class.

- If you are overriding any method, overridden method must not be more restrictive.

e.g.
```
class A {
    protected void Msg() { sout("Hello"); }
}

public class Simple extends A {
    void Msg() { sout("Helloworld"); }   C.T. error
    public static void Main(String args[]) {
        Simple obj = new Simple();
        obj.Msg();
    }
}
```

by default "default"

default > protected.

→ private > default > protected > public {restrictions}

→ ~~friend~~ friend class | methods.
→ can access pvt & protected member of a class ~~which~~ to which it is declared as a friend.

→ always declared inside a class.

Types of member functions

1) Simple :- which do not have any special keyword like static etc as prefix.

2) static :- f(n) is made static using static keyword.

→ It can be called using the object and the direct member access operator.

→ It can be called directly without the object using classname & scope resolution operator

A::f();

→ It can access only static data & static member f(n)

3) **Const** :- when used with a f(n), such mem f(n) can never modify the object or its related data members

4) **Inline** :- don't make f(n) call, expand code here.

5) **friend** :- made to get private access of class members

↳ If a f(n) is declared inline the a compiler may choose to expand the f(n) where ~~the~~ rather than choosing to make a f(n) call → (saving on stack operations & context switching).

↳ generally used for small f(n)

↳ its a request not a command.

↳ can increase compile time as inline f(n)s are processed multiple times.

Scope resolution :- → Define member f(n) of class outside the class

↳ void classname :: myclass() {

}

→ access static members of class

int a = classname :: var.name;

↳ used for name conflict resolution.

Deep and shallow copy

→ creating a copy of an object means to create an exact replica of the object having same literal values, data type & resources

↳ Depending upon the ~~copy~~ resources held by the ~~obj~~ object, either we need to perform shallow copy or Deep copy.

Shallow Copy :- In this object is created by simply copying the data of all the variables of original object.

→ this works well if none of the variables are defined in the heap section of the memory.

→ If some variables are dynamically allocated in heap section then the copied object variable will also refer the same memory location.

↳ This will create ambiguity & run-time errors, dangling pointers, since both objs will reference to the same memory location, then change made by one will reflect those change in another object as well.

Deep Copy :- object is created by copying data of all variables. & it also allocates, similar memory resources with same values to the object.

⇒ copy can be done through copy constructor and assignment operator.

Copy constructor	Assignment operator
→ called when a new obj is created from an existing obj, as a copy of a existing obj.	→ this operator is called when an already initialized obj is assigned a new value from another existing obj
→ It creates a separate memory block for the new object	→ does not create a separate memory block or new memory space.
→ It is an overloaded constructor	→ bit wise operator.

Constructor different from normal member f(n)

→ constructor have same name as that of class

→ constructor don't have return type.

→ constructor is automatically called when object is created.

→ if no constructor defined, compiler generates a default constructor for us.

Virtual constructor :- The concept is related to the idea of creating objects based on runtime information and polymorphism

- you have a base class & multiple derived class
- instead of calling a constructor directly, you call a factory method that decides which derived class to instantiate based on some runtime information.

Virtual copy constructor

Private Destructor :- Destructors with access modifier as private are known as Private Destructors.

→ used whenever we want to prevent the destruction of the object.

→ when a class have a private destructor, only dynamic objects of that class can be created.

Virtual Destructor :- When we need to delete a Derived class object using base class pointer. we Make Destructor of Base class as virtual.

e.g

```
class base {
  public:
    ~base() {
      }
  };
class Derived : public base {
  public:
    ~Derived() { }
  };
int main() {
  derived *d = new derived()
  base *b = d;
  delete b;
  return 0;
}
```

does not delete derived class ~~pointer~~ obj rather delete base class object

but if base destructor is defined as

```
virtual ~base() {
  }
```

now on delete b; both the derived & base class objs are deleted.

how destructor differs from normal member fn :

→ called when obj goes out of scope (automatically)
↳ fn ends
↳ program ends
↳ delete variable is called.

↳ don't take any arg & don't return anything.

↳ have same name as their class & name preceded by tilde (~).

↳ deletes an obj.

↳ destructors doesn't have parameters, so there can only be one. since we can't overload it.
↳ there is nothing optional when you are destroying the object.

Yes destructors can be virtual :- and are used when we want to delete obj of derived class using the pointer to a base class.

user-defined destructor is needed when an object requires custom cleanup logic the is not handled by default destructor.

e.g <u>Resource Management</u>
↳ if your class allocates memory dynamically, you need to free that memory using delete or delete[] in the destructor to avoid memory leaks.

↳ if files are opened or sockets the we need to close them in destructors

↳ if class opens a connection to a database, it should close the connection in the destructor.

# Pure virtual destructors :- is possible virtual ~base()=0; but we have to define its body after words.
~~class Derived~~ : Base :: ~Base() {

}

Objects	class
→ instance of class	→ blueprint / template of obj.
→ real world entity e.g pen, laptop etc.	→ group of similar objs.
→ physical entity	→ logical entity.
→ created through new keyword.	→ declared using class keyword.
→ created many time as per requirements	→ created once.
→ allocates memory when it is created.	→ no memory allocation.

e.g :- class :- human, obj :- Man, women.

Object :→ State
→ behaviour
→ identity
→ responsibility.

Keywords

→ static :- used for a constant variable or a method that is same for every instance of a class
→ for memory management, every obj of the class will share a same variable.

→ virtual :- used with method declaration to allow for it to be overridden in a derived class.

→ abstract :- used to define abstract classes & methods
↳ abstract classes may or may not have abstract methods
↳ abstract methods contains a method signature, but no method body.
↳ abstract class cannot be instantiated.

→ **final** :- If any variable is declared as final, we cannot change the value of final variable.

↳ ~~can be~~ blank final variable can be initialized only in constructor.

↳ If any method is declared as final, you can inherit it cannot overside it.

↳ If we make class as final, we cannot extend it.

↳ constructor's cannot be final → redundant as they are not inherited.

→ **explicit** :- It is used to prevent implicit conversions by the compiler, that could lead to unexpected behaviour

→ **This** :- It is a reference variable that refers to the current object pointer.

↳ can be used to access current class variable, method or constructor.

↳ can be used to return the current class instance from the method.

→ **new** :- It is used to create an instance of the class

→ It allocates the memory for new object and return a reference to that memory (allocates memory at runtime in heap)

→ **Const** :- used to define immutable variables, constant pointers & member fn's that do not modify object state in c++

↳ in java rather final keyword is used.

→ **super** :- reference variable which is used to refer immediate parent class object.

↳ when we create a instance of subclass, an instance of parent class is created implicitly which is referred by super ref variable.

Binding :- connecting method call to the method body

Static Binding

- when the type of obj is determined at the compile time
- if there is any private, final or static method, there is static binding

Dynamic binding

- when the type of obj is determined at run time
-
```
class Animal {
    void eat() { cout eating }
}
class Dog :: public Animal {
    void eat() { cout dog eating }
}
void main() {
    Animal a = new Dog();
    a.eat(); // dog eating
}
```
- since when instance of Dog is created, an instance of animal is also created so compiler ~~comp~~ doesn't know its type.

Message-passing :- refers to the process of objects communicating with each other by sending & receiving messages.

- A message is a request for an object to invoke one of its methods.

e.g
```
class Dog {
    void Bark() { cout("barking"); }
}
public class main {
    psvm (string arg[]) {
        Dog mydog = new Dog();
        Mydog.Bark();
    }
}
```
→ sending mess to mydog object to invoke the Bark method.

C	C++	Java
→ Procedural lang	→ Procedural + OOP lang	→ Pure OOP lang
→ compiler only	→ compiler only	→ interpreted lang (compiler + interpreter)
→ direct code execⁿ	→ direct code execⁿ	→ Execⁿ by JVM
→ .exe files	→ .exe files	→ .class files
→ support header files	→ support header files	→ use packages.
→ 32 Keywords	→ 63	→ 50

Procedural Prog

- → program is divided into small parts called f(n).
- → no access specifiers in this
- → adding new data and f(n) is not easy
- → no proper way of hiding the data, less secure
- → overloading is not possible
- → Inheritance X
- → function is more important than Data
- → Based on unreal world
- → code reusability is absent
- → e.g → C, FORTRAN, Pascal

OOP programming

- → program is divided into small parts called objects.
- → public, private, protected
- → easy.
- → provides data hiding, so it is more secure.
- → overloading ✓
- → inheritance ✓
- → Data is more imp than f(n).
- → real world based.
- → code reusability ✓
- → C++, Java, Python, JS

to be a full OOP language.

1) → encapsulation
2) → inheritance
3) → polymorphism
4) → abstraction
5) → all predefined types are objs
6) → all user defined " " "
7) → all operatn performed on objs must be only through methods exposed at the objs.

↳ Java fails to satisfy (5) & (7) hence not pure.
↳ its supports primitive data types like int, byte, float, long etc which are not objs
↳ if we create a static fn then we can call it without the instance of that class, defying OOPs nature.

array in java is an object, we can create the array by using "new" operator. hence array is also an object. (Dynamically created).

↳ there is a class for every array type

int[] → [I
int[][] → [[I
double[] → [D

↳ arrays can be assigned to variables of type object.

int[] arr = new int[2];

↳ array class have length variable which stores the length of the array.

Garbage collection :- process by which java prog performs automatic memory management.

Java programme → bytecode → run of JVM → objs are created in heap

garbage collector find these objs and free up their memory ← obj out of scope

steps :- unreferenced objs are identified & marked ready for garbage collection.
↳ marked objs are deleted.

benefit :- automatic garbage collector, unlike other lang where we have to manually free up the memory

System.gc() method.

Manipulators :- (>>, <<), special functions used to modify input and output operations.
↳ can be used to format data, control spacing etc.

Java finally block :-> used to execute important code
↳ finally block is always executed whether the excep^n is handled or not.
↳ finally block follows the try-catch block.
↳ It can be used to put "cleanup" code such as closing a file, closing connection etc. or print impo statements.
↳ finally block will not execute if program exits or by fatal error due to which prog abort.

⇒ for each try block → 0, or many catch block
→ only one final block.

exception :- it is an event that occurs during the execution of a program that disrupts the normal flow of execn of program.

⇒ exception occurs → object is created & is handed to the runtime system to handle it.
↳ obj is called exception obj → contains info about the error, its type & locatn.
↳ Creating excep^n & handling it to runtime system is known as throwing an exception

↳ the runtime sys finds / searches the list of methods to handle it (call stack)

↳ call stack selected contain a block of code to handle exp (exception handler)

↳ catches the exception.

↳ if no handler → program terminates

Errors :- mistake / fault by a programmer the prevents it from running correctly.

↳ Syntax :- voilating the syntax rule of prog language.

↳ Semantic :- ~~make~~ logic errors, syntatically correct, but does not produces desired output.

Exception handling

Try :- contains set of statements where an exception can occur., always followed by catch / finally block.

↳ if exception occurs then the rest of the statements in the try block are skipped.

Catch :- block where you handle the exceptions, must placed immediately after a try block.

generic exception handler (Exception e) :- in the catch block should be placed at the end of all the catch block.

finalize method :- This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

↳ defined in java.lang.object

↳ protected void finalize () throws Throwable {

// cleanup code.

}

Token :- smallest individual element in C programming lang that is meaningful to the compiler. It is the basic component of a C program.

6 types in C based on for they are used to perform:

- Keywords :- else, double, int, struct, etc
- Identifiers :- general terminology for the naming of variables, fns & array, e.g int kartik = 0;
- constants :- variables with fixed values, const
- strings :- array of char, end :- '\0'
- special symbols :- [ ], ( ), :, (,) (;) etc
- Operators :-
  - ↳ Unary :- ++, --, ~~&~~ req single operand
  - ↳ Binary :- +, -, *, /, >=, <=, req 2 operand
  - ↳ Ternary :- ? 3 operands
    - ↳ comparison ? true : false

when to use struct vs class.

Cohesion :- refers to how closely related and focused the responsibilities of a single module or class are

↳ high cohesion :- well defined class with its method and properties strongly related to the purpose.

↳ high desirable.

Types

- ↳ functional
- ↳ communicational
- ↳ Procedural
- ↳ Temporal
- ↳ logical
- ↳ coincidental

Coupling :- refers to the degree of interdependence b/w classes or modules.

↳ Lower coupling :- less dependent ⇒ class / modules are
↳ less desired :- change in one module mean it req less change in related modules.
↳ better modularity & easier maintenance

Types
↳ content (highest form) → highly undesired
↳ common
↳ control
↳ stamp
↳ Data
↳ Message (lowest form) → highly desired

Singleton design pattern :- a class must ensure that only single instance should be created & single object can be used by all other classes.
↳ early instantiation :- creation of instance at load time
↳ lazy instantiation :- creation of instance when required

Enumeration | enum :→ It is a user defined data type in C
↳ mainly used to assign names to integral constants.

```
enum week { Mon, Tue, Wed, Thur, Fri, Sat, Sun };
            0    1    2    3    4    5    6
int main()
{
    enum week day;
    day = Wed;
    printf("%d", day);
    return 0;
}
```

↳ by default
↳ we can change it.
↳ value can be Min val of row
Max val of row

// output → 2

↳ make code more readable, error-prone