PAGE No.
DATE / /

Constructors

→ A constructor is a special member function whose task is to initialize the object of its class

→ It is special because its name is same as the class name

→ The constructor is invoked whenever an object of its associated class is created

→ It is called constructor because it constructs the values of data members of the class

→ There is no need to write any statement to invoke the constructor function

→ If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately

→ Constructor don't have return type

NOTE: The compiler generates a default constructor for every class, but if you define your own constructor for that class, then compiler does not generate a default constructor by itself

Characteristics of constructor

→ They should be declared in public section
→ They are invoked automatically when the objects are created
→ No return type, not even void and they cannot contain return values
→ They cannot be inherited, through a derived class can call the base class constructor
→ Constructors cannot be virtual

Eg:

```
Class add
{
    int m, n;
  public :
      add(void)
    {
      m=0;
      n=0;
    }
};
```

⇒ Types of Constructor

→ Default
→ Parameterized
→ Copy

Default Constructor

Default Constructor is the constructor which doesn't take any arguements. It has no parameters

Eg:

```
Class Construct {
public:
    int a, b;
    Construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    Construct c;
    cout << " a: " << c.a << endl
         << " b: " << c.b;
    return 1;
}
```

PAGE No.
DATE / /

Parameterized Constructors

→ The constructors that can take arguments are called parameterized constructors

→ It may be necessary to initialize the various data elements of different objects with different values when they created

→ This is achieved by passing arguements to the constructor function when the objects are created

Eg:

```
Class add
{
    int m, n;
  public :
    add(int, int);
    ....
};
add :: add(int x, int y)
{
    m=x; n=y;
}
```

→ Two ways of Calling

- → Explicit eg: add Sum = add(2,3);
- → Implicit eg: add sum(2,3)

Destructors

→ A destructor is used to destroy the objects that have been created by a constructor

→ Like constructor, the destructor is a member function whose name is same as the class name but is preceded by tildle

Eg:
```
-integer(){}
```

→ A ~~constuct~~ destructor destroys the pointer of object and releases the memory

→ ~~Eg~~ A destructor function never takes any arguement & it does not have return value

→ It will be invoked implicitly by the compiler upon exit of the program

Eg:
```
class add
{
  int m, n;
  public:

  add(){
  cout<<" I am constructor"; }

  -add(){
   cout<<" I am destructor"; }
};
```

ENCAPSULATION

→ Defined as wrapping up of ~~data~~ data members and function ~~information~~ under a single unit

→ Variables
Methods / Functions
→ CLASS

→ It means that some or all of an object's internal structure is "hidden" from the outside world

⇒ ACCESS SPECIFIERS

TYPES	Within Same Class	In Derived class	Outside the class
PRIVATE	✓	✗	✗
PROTECTED	✓	✓	✗
PUBLIC	✓	✓	✓

→ Two steps
- → data members → Private
- → member function → public

→ Eg:

```
#include <iostream>
using namespace std;
class Encapsulation
{
```

```
private:
    int x;
public:
    void set(int a)
    {
        x = a;
    }
    int get() {
    [return x;]
};
```

```
int main()
{
Encapsulation obj;
obj.set(5)
cout<<obj.get();
return 0;
}
```

OUTPUT:
5

⇒ Advantages of Encapsulation

→ It is more flexible and easy to change with new requirements

→ Encapsulation makes unit testing easy

→ It allows you to control who can access what

→ It reduces coupling of modules and increase cohesion inside a module because all piece of one thing are encapsulated in one place

→ Encapsulation allows you to change one part of code without affecting other part of code

ABSTRACTION

→ Data Abstraction or information hiding refers to providing only essential information to the outside world and hiding their background (internal) details

→ Depends on separation of the interface & implementation of details of the program

→ Two WAYS ⟶ Abstraction using Classes
⟶ Abstraction in header files

→ Eg:

```
# include <iostream>
# include <math.h>
Using namespace std.
int main ()
{
    int n=4;
    int power = 3;
    int result = pow (n, power)
    cout << " Cube of n is : " << result;
    return 0;
}
```

OUTPUT:

Cube of n is : 64

INHERITENCE

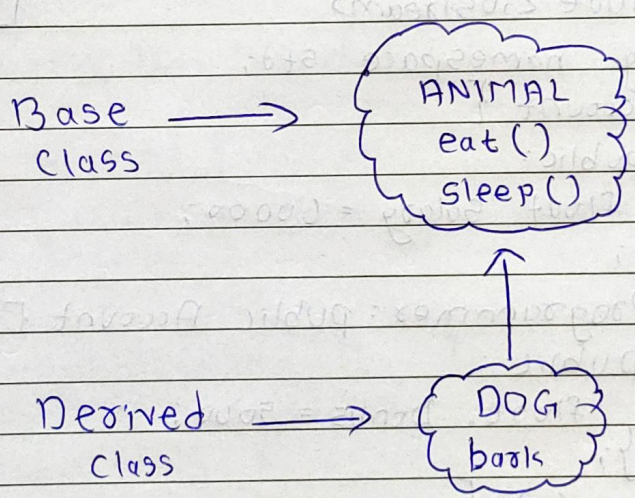
→ The Capability of a class to derive properties and characteristics from another class is called Inheritence

→ It is used in acheiving software/code reusability

→ CLASS

BASE Class
Super Class
⇓
The class whose properties are inherited by other class or subclass

DERIVED Class
Subclass
⇓
The class that inherits properties from another class



→ We use inheritence only if an is-a relationship is present between two classes
Eg: Car is a Vehicle
Orange is a Fruit

PAGE No.
DATE / /

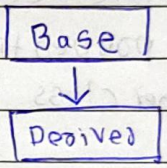
⇒ Types of Inheritence

- Single Inheritence
- Multiple Inheritence
- Multi-level Inheritence
- Heirarchical Inheritence
- Hybrid Inheritence

* Single Inheritence

The Inheritence in which a derived class is inherited from only one base class

Eg:




```
#include <iostream>
  using namespace std;
class Account {
      public:
        float salary = 60000;
      };
class Programmer: public Account {
      public:
        float bonus = 5000;
      };
int main() {
Programmer p1;
      cout << " Salary :" << p1.salary << endl;
      cout << " Bonus :" << p1.bonus << endl;
    return 0;
}
```

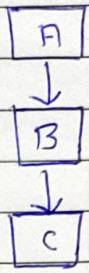
OUTPUT: Salary: 60000
Bonus: 5000

PAGE No.
DATE / /

* Multilevel Inheritence

→ It is a process of deriving a class from another derived class

→ Last derived class acquire all the members of all its base classes



→ Eg:

```
#include <iostream>
using namespace std;
class Animal {
    public:
    void eat() {
        cout << "Eating..." << endl;
    }
};
class Dog: public Animal {
    public:
    void bark() {
        cout << "Barking..." << endl;
    }
};
Class BabyDog: public Dog {
    public:
    Void weep() {
        cout << "Weeping...";
    }
};
```

```
int main() {
BabyDog d1;
d1.eat();
d1.bark();
d1.weep();
return 0;
}
```

Output: Eating...
Barking...
Weeping...

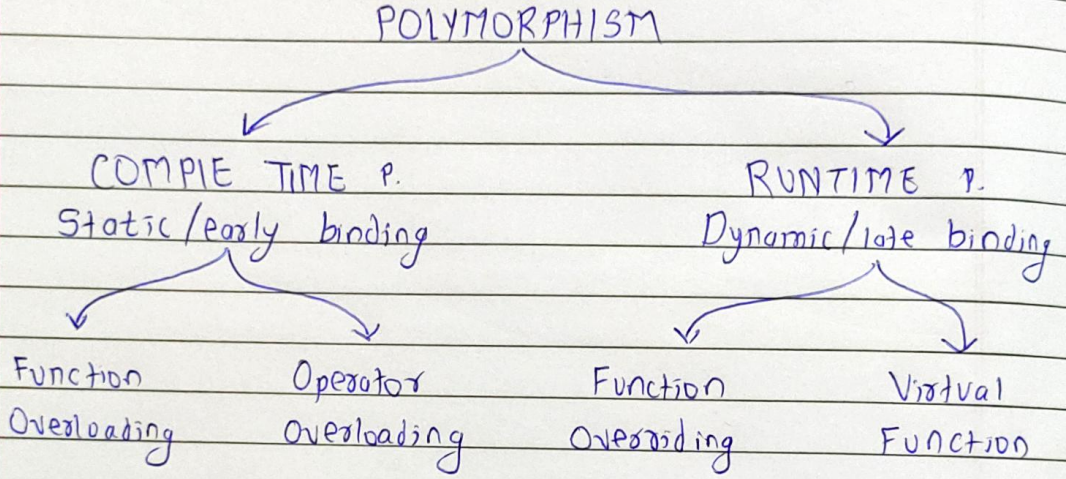
PAGE No.
DATE / /

POLYMORPHISM

POLY → many
MORPHISM → forms

→ It is a feature of OOPs that allows the object to behave differently in different conditions

→



Scope of Variable

→ A block or a region where a variable is declared, defined and used and when a block or a region ends, variable is automatically destroyed

→ Types → Local → Declared inside a block/function
→ Global → Declared outside all function/block

Eg:

```
#include<stdio.h>
int fun();

int var = 10;                          → Global Variable

int main() {
    int var = 3;                       → Local Variable
    printf("%d\n", var);
    fun();
    return 0;
}

int fun()
{
    printf("%d", var)
}
```

OUTPUT:

```
3
10
```

Overloading

→ Function Overloading (FO)
→ Operator Overloading (OO)

→ Defination: The process of having two or more function within the same name but different parameter is known as Function Overloading (FO)

Eg:

```
#include <iostream>
using namespace std;
public
Class A
{
    int n1=10, n2=20;
    public
        void fun()
        {
            int sum = n1-n2;
            cout << "Addition: " << sum << endl;
        }
        Void fun(int a, int b)
        {
            int sub = a-b;
            cout << "Subtraction: " << sub << endl;
        }
};
int main()
{ A obj;
    obj.fun(); obj.fun(100, 50);
    return 0;
}
```

PAGE No.
DATE / /

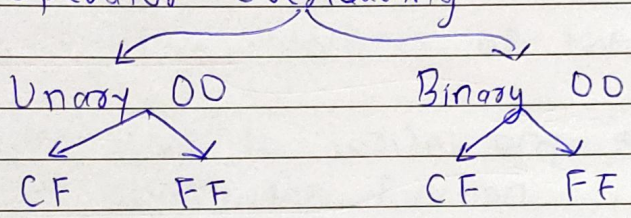
⇒ Operator Overloading (OO)

→ The process of Assigning two or more operation on the same operator is known as operator overloading

→ To acheive operator overloading we have to write a special function kown as operator()

→ 2 Ways → Class Function (CF)
→ Friend Function (FF)

→ 2 Types : Operator Overloading



⇒ Operators which can't be Overloaded
- Scope operator (::)
- Size of
- member selector (.)
- member pointer selector (*)
- ternary operator (?:)