Low Level Design

- Kapil Yadav

- **OOPs concepts**
- **Design Principles**
- **Design Patterns**

Kapil Yadav

INDEX

→ Run Time polymorphism
- Virtual Functions

12. Encapsulation

13. Data Abstraction

1. OOPS DESIGN PRINCIPLES
- DRY
- KISS
- YAGNI

2. COHESION & COUPLING
3. CQS
4. SOLID

OOPS DESIGN PATTERN

1) Factory Method Design Pattern :-
2. SINGLETON DESIGN PATTERN :-
3. Builder Design Pattern
4. Observer Design Pattern
5. Abstract Factory Design

class :- Blueprint

↓

design / type (template) → state / property / field / data member
→ behaviour / function / data function.

Ex- Humans → 2 hand, 2 leg, 2 eyes → state / properties
↓
behaviour → walk, speak, eat.

→ Architect gives Blueprint is class, using the Blueprint, we make the house is object.

→ Object takes memory / space, class doesn't.
Object is an implementation / real entity / instance of class.

→ class is a logical component.

→ Object takes memory from heap.

```cpp
class Human {
    public:
    //properties
    int age;
    int weight;

    //behaviour
    void sleep()
    {
        cout<<"He is sleeping"<<endl;
    }

    void eat()
    {
        cout<<"He is eating"<<endl;
    }
};


int main()
{
    Human chitti;
    chitti.age = 25;
    chitti.weight = 40;
    chitti.sleep();
    chitti.eat();
}
```

Access Modifiers in C++:-

One of the main features of OOPs is **Data Hiding.**

Data Hiding :- Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

→ The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.

→ There are 3 types of access modifiers available in C++.

1. Public
2. Private
3. Protected

→ By default, the access modifier of the members will be Private.

1. Public :- The public keyword is used to create public members (data and functions).
   - The public members are accessible from any part of the program.
   - The public members can be accessed from anywhere in the program using direct memory access operator (.) with the object of that class.

```cpp
class Rectangle{

    public:
     int Length,Breadth;

    Rectangle()
    {
        Length=5;
        Breadth=5;
    }



    void Display()
    {
        cout<<"\n Length  "<<Length;
        cout<<"\n Breadth  "<<Breadth;
    }
};

int main()
    {   Rectangle R;
        cout<<R.Length<<" "<<R.Breadth;
        R.Display();
        return 0;
    }
```

Private Access Modifier:-

- The private keyword is used to create private members (data and functions).
- The private members can only be accessed from within the class.
- However, friend classes and friend functions can access private members.

```
1 class Rectangle{
2
3      int Length,Breadth;
4     public:
5
6
7     void Area(int L, int B)
8     {   Length = L; Breadth = B;
9         int area = Length * Breadth;
10        cout<<"Area of the Rectangle is : "<<area<<endl;
11    }
12 };
13
14
15 int main()
16    {   Rectangle R;
17        R.Area(10,20);
18        return 0;
19    }
```

- We can access the private data members of a class indirectly using the public member functions of the class.

Friend class :- A friend class can access private and protected members of other class in which it is defined as friend. It is sometimes used to allow a particular class to access private members of other class.

```
1 class Animal{
2     int age;
3     int weight;
4     public:
5     Animal()
6     {
7         age = 25;
8         weight = 70;
9     }
10    friend class Dog;
11 };
12
13 class Dog {
14    public:
15    void display(Animal &t)
16        {
17            cout<<endl<<"Age "<<t.age;
18            cout<<endl<<"Weight "<<t.weight;
19        }
20 };
21
22 int main()
23    {   Animal A;
24        Dog D;
25        D.display(A);
26        return 0;
27    }
```

Friend Function:-

Friend Function Like friend class, a friend function can be given a special grant to access private and protected members. A friend function can be:
a) A member of another class
b) A global function

- A friend function is a special function in C++ which in-spite of not being member function of a class has privilege to access private and protected data of a class.
- A friend function is a non member function or ordinary function of a class, which is declared as a friend using the keyword "friend" inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- The keyword "friend" is placed only in the function declaration of the friend function and not in the function definition.
- When friend function is called neither name of object nor dot operator is used. However it may accept the object as argument whose value it want to access.
- Friend function can be declared in any section of the class i.e. public or private or protected.

Syntax:-

```
Syntax :

class <class_name>
{
    friend  <return_type>  <function_name>(argument/s);
};
```

```cpp
class Largest
{
    int first, second, maxi;
    public:
        void set_data(int f, int s)
        {
            first = f;
           second = s;
        }
        friend void find_max(Largest);
};

void find_max(Largest t)
{
    if(t.first>t.second)
        t.maxi=t.first;
    else
        t.maxi=t.second;

        cout<<"Maximum Number is\t"<<t.maxi;
}

int main()
{
    Largest l;
    l.set_data(10,20);
    find_max(l);
    return 0;
}
```

Protected Access Modifier :-

→ Protected Access Modifier is similar to private access modifier,
→ But the difference is protected members can be access within the class and from the derived class as well.

```cpp
class Animal{
    protected:
    int age;
    int weight;
    public:
    Animal()
    {
        age = 25;
        weight = 70;
    }
};

class Dog : public Animal{
    public:
    void display()
    {
    cout<<"Age  "<<age<<" " <<"Weight  "<<weight<<endl;
  }
};

    int main()
    {   Dog D;
        D.display();
        return 0;
    }
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Base class member access specifier	Type of inheritance		
	public inheritance	**protected** inheritance	**private** inheritance
public	**public** in derived class Can be accessed directly by member, friend and nonmember functions	**protected** in derived class Can be accessed directly by member and friend functions	**private** in derived class Can be accessed directly by member and friend functions
protected	**protected** in derived class Can be accessed directly by member and friend functions	**protected** in derived class Can be accessed directly by member and friend functions	**private** in derived class Can be accessed directly by member and friend functions
private	Hidden in derived class Can be accessed by member and friend functions through **public** or **protected** member of the base class	Hidden in derived class Can be accessed by member and friend functions through **public** or **protected** member functions of the base class	Hidden in derived class Can be accessed by member and friend functions through **public** or **protected** member functions of the base class

CONSTRUCTOR :-

Constructor is a special method that is invoked automatically at the time of object creation.

→ It is used to initialize the data members of new objects.

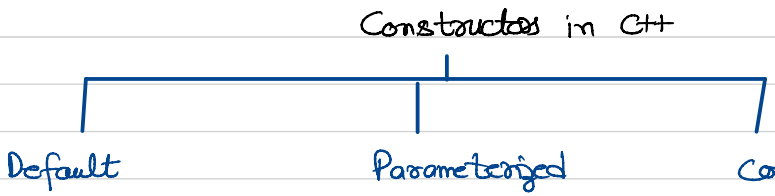
→ Constructors don't have any return type.

→ Constructor have same name as the class itself.

→ If we don't specify a constructor, C++ compiler generates a default constructor for object.

→ It should be placed in public section of class.

→ It can be overloaded.



→ We can do constructor private. But what will happen ?

```cpp
class Human {
    public:
    //properties
    int age;
    int weight;

    //default constructor
    Human()
    {
        cout<<"I am inside the default construtor - creating object"<<endl;
    }

    //parameterised constructor
    Human(int age, int weight)
    {   this->age = age;
        this->weight = weight;
        cout<<"I am inside the parameterised construtor by passing both age and weight  "<<age<<" "<<weight<<endl;
    }
    //parameterised constructor
    //constructor overloading
    Human(int age)
    {   this->age = age;
    cout<<"I am inside the parameterised construtor by only passing age "<<age<<" "<<weight<<endl;
    }
    //copy constructor
    Human(const Human &h)
    {
        age = h.age;
       weight = h.weight;
       cout<<"I am inside the copy construtor by only passing age "<<age<<" "<<weight<<endl;
    }

    //behaviour
    void sleep()
    {
        cout<<"He is sleeping"<<endl;
    }

    void eat()
    {
        cout<<"He is eating"<<endl;
    }
};


int main()
{
    Human chitti;
    cout<<" I came outside the constructor"<<endl;
    chitti.age = 25;
    chitti.weight = 40;
    Human(22,70);
    Human(22);
    Human kapil = chitti;
    chitti.sleep();
    chitti.eat();
}
```

→ Do constructor returns any value ?

No, constructor does not return any value.

In C++, there are several ways to call the copy constructor of a class:

1. Direct Initialization:
   You can use direct initialization to call the copy constructor when creating a new object of the same class, passing another object of the same class as an argument.

```cpp
MyClass obj1; // Creating an object using the default constructor
MyClass obj2 = obj1; // Calling the copy constructor explicitly using direct initialization
```

2. Function Parameter Passing:
   When passing an object of a class as a function parameter by value, the copy constructor is automatically called to create a copy of the original object inside the function.

```cpp
void someFunction(MyClass paramObj) {
    // Copy constructor is called here to create a copy of the object passed as a parameter
    // ...
}

MyClass obj;
someFunction(obj); // Copy constructor is called when passing 'obj' as a parameter
```

3. Returning Objects from Functions:
   When a function returns an object of a class by value, the copy constructor is called to create a copy of the object that will be returned.

```cpp
MyClass createObject() {
    MyClass obj;
    // ...
    return obj; // Copy constructor is called here to create a copy that will be returned
}

MyClass newObj = createObject(); // Copy constructor is called when returning the object
```

4. Explicit Copy Construction:
   You can explicitly call the copy constructor using the object of the same class and the copy constructor syntax.

```cpp
MyClass obj1; // Creating an object using the default constructor
MyClass obj2(obj1); // Explicitly calling the copy constructor
```

These are the common methods of calling the copy constructor in C++. Remember that if you don't define a copy constructor explicitly, the compiler will provide a default copy constructor for you, which performs a member-wise shallow copy. If your class has dynamically allocated resources or pointers, you may need to define a custom copy constructor to ensure a deep copy is performed to avoid issues with shallow copies.

STATIC KEYWORD :-

↳ Static variables : variables in a function, variables in a class

Static Members of class :- class objects and functions in a class.

→ In class, static member is accessed using class, not object.

```
class Human {

    static int count;

    Human()
    { count++;
    }
};      // initialize static member of class Human
int Human::count = 0;

int main()
{   cout << Human::count << endl;
}
```

Q:) Can I access static data member without creating an object?

Ans. Yes.

Static function → can change static Data Member.
Cannot change the value of non-static Data Member.

```cpp
class Human {
    public:
    //properties
    int age;
    int weight;


    //static data member
    static int count;

    //default constructor
    Human()
    {  count++;
        cout<<"I am inside the default construtor - creating object"<<endl;

    }

   static void update()
        {   count++;
        }

    void sleep()
    {
        cout<<"He is sleeping"<<endl;
    }

    void eat()
    {
        cout<<"He is eating"<<endl;
    }
};

int Human::count =0;

int main()
{
    Human chitti;
    cout<<" I came outside the default constructor"<<endl;
    chitti.age = 25;
    chitti.weight = 40;
    chitti.update();

    cout<<chitti.age<<" "<<chitti.weight<<endl;
    cout<<"Total count Calls  "<<Human::count<<endl;

    chitti.sleep();
    chitti.eat();
}
```

NOTE :- A non-static member function can modify a static data member as long as the data member's visibility allows it.

→ we cannot use this keyword in static method.

Q) Why main function in Java is static?
Ans.)

THIS Keyword :-

→ Object Pointer – A pointer contains address of an object is called object pointer.

→ this is a local object pointer in every instance member function containing address of the caller object.

→ this pointer cannot be modify.

→ It is used to refer caller object in member function.

→ Friend functions do not have this pointer, because friends are not members of a class. Only member functions have a this pointer.

```cpp
#include <iostream>
using namespace std;
class Employee {
   public:
       int id;
       string name;
       float salary;
       Employee(int id, string name, float salary)
        {
            this->id = id;
            this->name = name;
            this->salary = salary;
        }
       void display()
        {
            cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1 =Employee(101, "Kapil", 120000);
    Employee e2=Employee(102, "Deepankar", 80000);
    e1.display();
    e2.display();
    return 0;
}
```

NEW KEYWORD :—

→ The new is a memory allocation operator, which is used to allocate memory at the runtime.

→ The memory initialized by the new operator is allocated in a heap.

→ It returns the starting address of the memory, which get assigned to the variable.

Syntax — type variable = new type(parameter_list);

- Type – datatype of a variable.
- variable – name of the variable.
- parameter_list – is the list of values that are initialized to a variable.

int *ptr = new int;

WHAT IS MALLOC ?

→ Malloc() is a function that allocates memory at the runtime.

→ Syntax — type variable_name = (type*) malloc(sizeof(type));

- type : it is the datatype of the variable.
- variable_name : it defines the name of the variable that points to the memory.
- (type*) : It is used for typecasting so that we can get the pointer of a specified type that points to the memory.

NOTE :- The malloc() function returns the void pointer, so typecasting is required to assign a different type to the pointer. The sizeof() operator is required in the malloc() function as the malloc() function returns the raw memory, so the sizeof() operator will tell the malloc() function, how much memory is required for the allocation.

→ Memory Allocation using New

```cpp
#include<iostream>
using namespace std;
class car
{   string name;
    int num;

    public:
        car(string a, int n){
            cout << "Constructor called" << endl;
            this ->name = a;
            this ->num = n;
        }

        void enter() {
            cin>>name;
            cin>>num;
        }

        void display() {
            cout << "Name: " << name << endl;
            cout << "Num: " << num << endl;
        }
};

int main(){
    car *p = new car("Innova", 2012);
    p->display(); delete p;
}
```

Memory Allocation using malloc :-

```cpp
#include <iostream>
#include<stdlib.h>
using namespace std;

int main()  {
  int len;
  cout << "Enter the count of numbers :" <<endl;
  cin >> len;
  int *ptr;
  ptr=(int*) malloc(sizeof(int)*len);
  for(int i=0;i<len;i++)  {
     cout << "Enter a number : " <<endl;
      cin >> *(ptr+i);
  }
  cout << "Entered elements are : " <<endl;
   for(int i=0;i<len;i++)  {
     cout << *(ptr+i) <<endl;
  }
    free(ptr);
    return 0;
}
```

→ If the sufficient memory is not available, malloc() function returns the NULL pointer.

→ Allocated memory can be deallocate using free() function.

Difference between New and Malloc

NEW	MALLOC
1. New operator construct an object. (it calls the constructor to initialize an object.	1. Malloc is a function, it does not call the constructor.
2. delete operator to destroy the object.	2. free() function to deallocate the memory.
3. new is an operator	3. It is a predefined function in stdlib.h header file.
4. new can be overloaded.	4. malloc() cannot be overloaded.
5. If sufficient memory is not available, new will throw an exception	5. malloc() will return a NULL pointer.
6. We need to specify number of objects.	6. We need to specify number of bytes to be allocated.
7. memory allocated by new cannot be resized.	7. It can be reallocated using realloc() function.
8. Execution time of new is less than malloc()	8. E.T is more than new.

FINAL Keyword :-

→ C++11 allows built-in facility to prevent overriding of virtual function using final specifier.

```cpp
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void myfun() final{
        cout << "myfun() in Base";
    }
};
class Derived : public Base{
    void myfun(){
        cout << "myfun() in Derived\n";
    }
};

int main(){
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}
```

error - 'overriding final functions'

```cpp
#include <iostream>
class Base final{
};

class Derived : public Base
{
};

int main(){
    Derived d;
    return 0;
}

```

error :- 'Cannot derive from 'final' base 'Base' in derived type 'Derived'

CONST KEYWORD :-

→ Const Keywords used to define the constant value that cannot change during program execution.

Use of const Keyword with different parameters :-

- Use const variable
- Use const with pointers
- Use const pointer with variables
- Use const with function arguments
- Use const with class member functions
- Use const with class data members.
- Use const with class objects.

1. Const variable

```
const int a = 20;
a = a + 10;
```

OUTPUT :- Error.

2. Const pointer

→ We cannot change the address of the const pointer after its initialization, which means the pointer will always point to the same address once the pointer is initialized as the const pointer.

```cpp
int main ()
{
int x = 10, y = 20;

// const integer ptr variable point address to the variable x
int* const ptr = &x;

//ptr = &y; // now ptr cannot changed their address
*ptr = 15; // ptr can only change the value
cout << " The value of x: " << x << endl;
cout << " The value of ptr: " << *ptr << endl;
return 0;
}


OUTPUT:
        The value of x: 15
        The value of ptr: 15
```

3. Pointer to constant variable

→ It means pointer points to the value of a const variable that cannot change.

const int* x;
char const* y; } → both are pointer to constant variable

```cpp
int main ()
{
int x = 7, y = 10;

 // here x become constant variable
const int *ptr = &x;
cout << " \n The initial value of ptr:" << *ptr;
cout << " \n The value of x: " <<x;

// *ptr = 15; It is invalid; we cannot directly assign a value to the ptr variable
ptr = &y; //  here ptr variable pointing to the non const address 'y'

cout << " \n The value of y: " <<y;
cout << " \n The value of ptr:" << *ptr;
return 0;
}
```

4. Constant function Arguments

```cpp
int Test (const int num)
{
// if we change the value of the const argument, it thwrows an error.  num = num + 10;
cout << " The value of num: " << num << endl;
return 0;
}
int main ()
{
Test(5);
}
```

5. Const pointer pointing to a Const variable:-

Syntax - const datatype* const varname;

```
int main()
{
    int x = 5;
    int m =10;
     const int* const i = &x;
    //i = &m; error
    //*i=10;
    // The above statement will give CTE
    // Once Ptr(*i) value is
    // assigned, later it can't be modified(Error)

    char y = 'A';
    const char* const j = &y;

    // *j='B';
    // The above statement will give CTE
    // Once Ptr(*j) value is
    // assigned, later it can't be modified(Error)

    cout << *i << " and " << *j;
    return 0;
}
```

6. Pass const-argument value to a non-const parameter of a function cause error:-

```
1 int foo(int* y){
2     return *y;
3 }
4
5 int main(){
6     int k = 8;
7     const int* x = &k;
8     cout << foo(x);
9     return 0;
10 }
11
```

→ error :- invalid conversion from 'const int*' to 'int*'

7. For const return type :- the return type of the function is const and so it returns a const integer value to us.

```
1 const int foo(int y)
2 {
3     y--;
4     return y;
5 }
6
7 int main()
8 {
9     int x = 9;
10    const int z = 10;
11    cout << foo(x) << '\n'<< foo(z);
12
13    return 0;
14 }
```

→ There will be no issue whether we pass const or non-const variable to the function because the value will be returned by the function will be constant automatically. As the argument of the function is non-const.

8. For const return type and const parameter :-

```cpp
const int foo(const int y)
{
    //y--; it will give an error
    return y;
}

int main()
{
    int x = 9;
    const int z = 10;
    cout << foo(x) << '\n'<< foo(z);

    return 0;
}
```

→ Here, both const and non-const values can be passed as the const parameter to the function, but we are not allowed to then change the value of a passed variable because the parameter is const.

Otherwise, we will face the error.

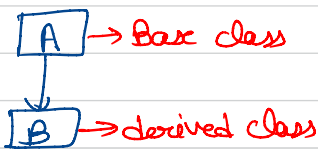
" y is a const var its value can't be changed.

Inheritance?- In C++, there are 5 types of Inheritance.

1. Single
2. Multilevel
3. Hierarchical
4. Multiple
5. Hybrid

} Not in Java

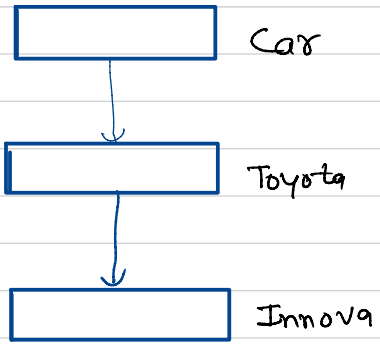
1) Single :-




```cpp
class Human {
    public:
    //properties
    int age ;
    int weight;

};

class Male : public Human
{  public:
};

int main()
{   Male Me;
    Me.age = 35;
    Me.weight = 60;
    cout<<Me.age<<" "<<Me.weight<<endl;
}
```

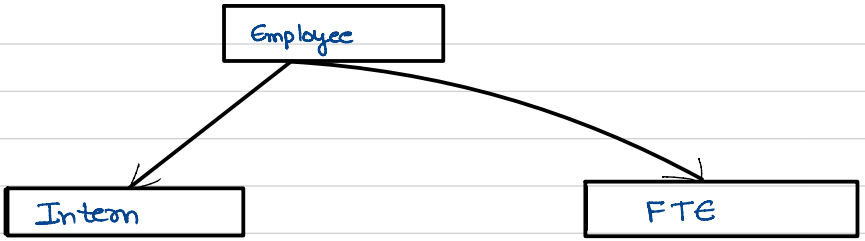
output : 35 60

2.) Multilevel :-



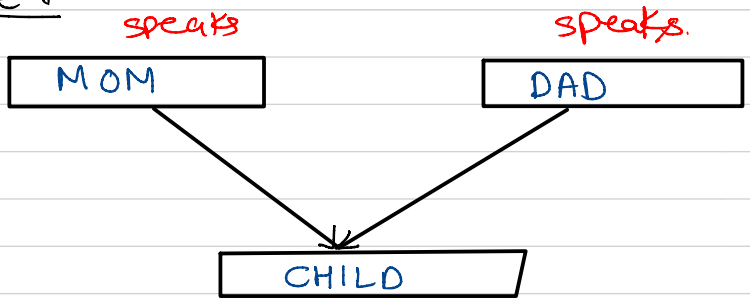

```cpp
class Car {
    public:
    string cartype;

};

class Toyota : public Car
{  public:
    string country;

};

class Innova : public Toyota{
     public:
     int price;
};

int main()
{   Innova In;
    In.cartype = "Petrol";
    In.country = "Japan";
    In.price = 300000;
    cout<<In.cartype<<" "<<In.country<<" "<<In.price<<endl;
}
```

3) Hierarchical :




```cpp
class Employee{
    public:
    string Profile;

};

class Intern : public Employee
{  public:
    int salary;

};

class Fulltime : public Employee{
    public:
    int salary;
};

int main()
{   Fulltime FTE;
    FTE.salary = 1000000;
    FTE.Profile = "SDE";
    Intern ITE;
    ITE.salary = 500000;
    ITE.Profile = "SDE";
    cout<<FTE.Profile<<" "<<FTE.salary<<" "<<endl;
    cout<<ITE.Profile<<" "<<ITE.salary<<" "<<endl;
}
```

Multiple :

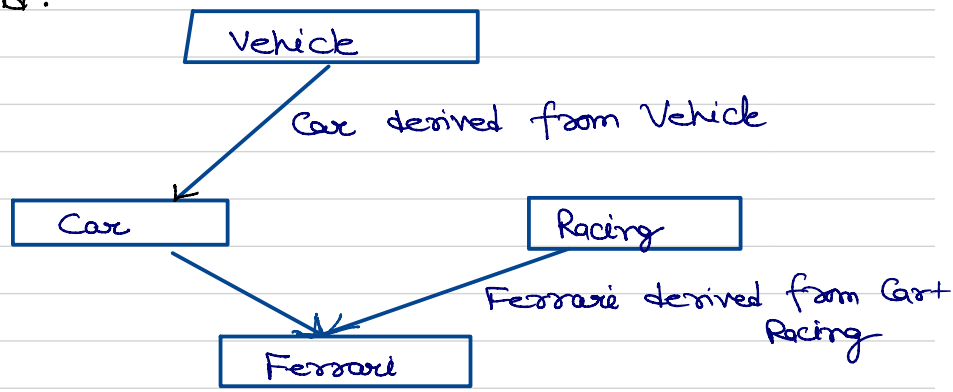


→ There can be ambiguity, when the Base classes functions have same name, to resolve this:- use Scope resolution operator.

```
1  class Mom{
2          public:
3          void display()
4          {  cout<<"Class MOM  "<<endl;  }
5      };
6
7      class Dad{
8          public:
9          void display()
10         {  cout<<"Class DAD  "<<endl; }
11     };
12
13     class child : public Mom, public Dad{
14         public:
15         void display()
16         {   cout<<"Class CHILD  "<<endl; }
17     };
18
19     int main()
20     {   child ch;
21         ch.display();
22         ch.Mom::display();
23         ch.Dad::display();
24     }
```

scope resolution operator

Hybrid :-




```cpp
#include <iostream>
using namespace std;
class vehicle  {
public:
vehicle()  {
    cout<< "This is a vehicle\n";  }
};
class Car: public vehicle  {
public:
Car()  {
    cout<< "This is a car\n"; }
};
class Racing  {
public:
Racing()  {
    cout<< "This is for Racing\n";  }
};
class Ferrari: public Car, public Racing  {
public:
Ferrari() {
cout<< "Ferrari is a Racing Car\n"; }
};
int main() {
    Ferrari f;
    return 0;
}
```

ADVANTAGES :-

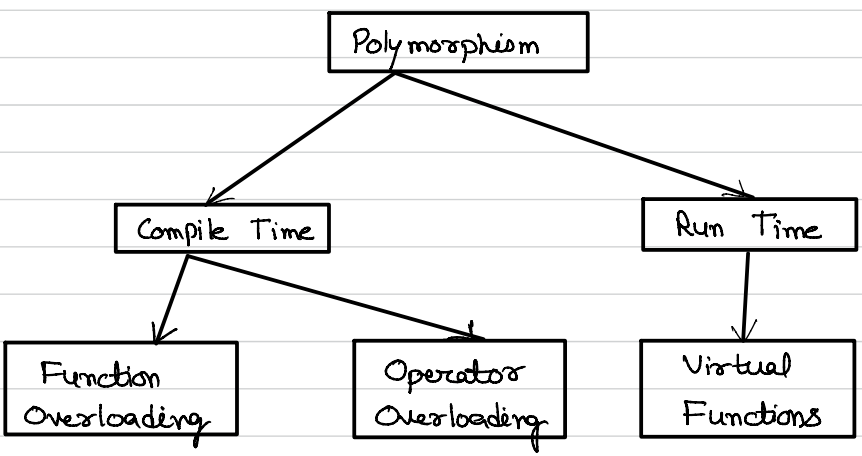
- → Code Reusability.
- → Improves code readability as we don't need to rewrite the same code again, code looks cleaner and scalable.
- → Inheritence supports extensibility as new classes can be easily added to existing classes.

POLYMORPHISM :-

POLY → Many

MORPH → Forms.

→ In C++, polymorphism is mainly divided into two types:-
  → Compile-time polymorphism.
  → Runtime Polymorphism.



1. Compile-time polymorphism :- This type of polymorphism is acheived by function overloading or operator overloading.

- Function Overloading :-

→ When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded.

→ Functions can be overloaded by :
  - changing the number of arguments
  OR / AND

- changing the type of arguments.

```
class Calculator{
    public:
    int add(int a, int b)
    {
        return a+b;
    }

    int add(int a, int b, int c)
    { return a+b+c;
    }

    float add(float a, int b)
    { return a+b;
    }

    float add(int a, float b)
    { return a+b;
    }

    float add(float a, float b)
    { return a+b;
    }

      };


int main()
{   Calculator Calc;
     float a,b;
    cout<<"Add three integer numbers "<<" "<<Calc.add(2,3,5)<<endl;
    cout<<"Add two integer numbers "<<Calc.add(10,30)<<endl;
    a = 11.5;
    int c = 20;
    cout<<"Add one integer and one float number "<<Calc.add(a,c)<<endl;
    b = 39.60;
    cout<<"Add one float and one integer number "<<Calc.add(15,b)<<endl;
    a = 19.4;
    b = 18.6;
    cout<<"Add two float numbers "<<Calc.add(a,b)<<endl;
}
```

Q.) Can we overload main method ?

To overload main() function in C++, it is necessary to use class and declare the main as member function. Note that main is not reserved word in programming languages like C, C++, Java and C#. For example, we can declare a variable whose name is main, try below example:

```cpp
#include <iostream>
using namespace std;
class Test
{
public:
    int main(int s)
    {
        cout << s << "\n";
        return 0;
    }
    int main(char *s)
    {
        cout << s << endl;
        return 0;
    }
    int main(int s ,int m)
    {
        cout << s << " " << m;
        return 0;
    }
};
int main()
{
    Test obj;
    obj.main(3);
    obj.main("I love C++");
    obj.main(9, 6);
    return 0;
}
```

The outcome of program is:

```
3
I love C++
9 6
```

```cpp
#include <iostream>
int main()
{
    int main = 10;
    std::cout << main;
    return 0;
}
```

Output:

```
10
```

Operator Overloading :-

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

How to overload the operator :-

→ To overload an operator, a operator function is defined inside a class.

Syntax

```
class class_name
{
    public :
        return type operator(args)
        {
        }
};
```

```cpp
class Rectangle{
    int Length,Breadth;
    public:
    Rectangle()
    {
        Length=0;
        Breadth=0;
    }

    void operator ++()
    {
        Length+=2;
        Breadth+=2;
    }

    void Display()
    {
        cout<<"\n Length  "<<Length;
        cout<<"\n Breadth  "<<Breadth;
    }
};


    int main()
    {   Rectangle R;
        cout<<"\n Length and Breadth before increament  ";
        R.Display();
        ++R;
        cout<<"\n Length and Breadth after increament ";
        R.Display();
        return 0;
    }
```

But, among them, there are some operators that cannot be overloaded. They are

- Scope resolution operator : :
- Member selection operator
- Member selection through *

Pointer to member variable

- Conditional operator ? :
- Sizeof operator sizeof()

Operators that can be overloaded

1. Binary Arithmetic -> +, -, *, /, %
2. Unary Arithmetic -> +, -, ++, –
3. Assignment -> =, +=,*=, /=,-=, %=
4. Bit- wise -> & , | , << , >> , ~ , ^
5. De-referencing -> (->)
6. Dynamic memory allocation and De-allocation -> New, delete
7. Subscript -> [ ]
8. Function call -> ()
9. Logical -> &, | |, !
10. Relational -> >, < , = =, <=, >=

Why can't the above-stated operators be overloaded?

1. sizeof – This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

2. typeid: This provides a CPP program with the ability to recover the actual derived type of the object referred to by a pointer or reference. For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type 'look' like another type, polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

3. Scope resolution (::): This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this operator.

4. Class member access operators (.(dot), .* (pointer to member operator)): The importance and implicit use of class member access operators can be understood through the following example:

Important points about operator overloading

1) For operator overloading to work, at least one of the operands must be a user-defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behaviour is the same as the copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

Runtime Polymorphism :-

→ It is achieved by function overriding.

- Function Overriding :-

→ Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overriden.

```
1 class Animal{
2     public:
3     int age;
4     int weight;
5
6     void speak()
7     {
8         cout<<"HIIII "<<endl;
9     }
10 };
11
12 class Dog : public Animal{
13     public:
14     void speak()
15     {
16         cout<<" WOOF "<<endl;
17     }
18 };
19
20 int main()
21     {   Dog D;
22         D.speak();
23         return 0;
24     }
```

Rules → Must do inheritance.

→ same function name / same parameter.

Q:) Can we override static method ?

linkedin.com/in/kapilyadav22

Virtual Function:- A virtual function is a member function, which is declare within a base class and is overridden by a derived class.

→ When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class's version of the function.

→ They are mainly use to achieve Run-time polymorphism. The resolving of function call is done at runtime.

→ Functions are declared with a virtual keyword in the base class.

Points to remember :-

1. virtual functions cannot be static.
2. Virtual function can be a friend function of another class.
3. They are always defined in the base class and overridden in a derived class. (we may / maynot override in the derived class, optional).
4. A class may have virtual destructor, but it cannot have a virtual constructor.

```cpp
#include <iostream>
using namespace std;

class Shape {
 public:
    int get_Area(){
        cout << "This is call to parent class area\n";
        return 1;
    }
};

class Square : public Shape {
int area=0;
public:
    Square(int l, int b)
    { area = l*b;
    }
    int get_Area(){   cout << "Square area: " << area << '\n';
        return area;
    }
};
int main()
{
    Shape* s;
    Square sq(5, 5);
    s = &sq;
    s->get_Area();
    return 0;
}
```

OUTPUT - This is call to parent class area

```cpp
#include <iostream>
using namespace std;

class Shape {
 public:
    virtual int get_Area(){
        cout << "This is call to parent class area\n";
        return 1;
    }
};

class Square : public Shape {
int area=0;
public:
    Square(int l, int b)
    { area = l*b;
    }
    int get_Area(){   cout << "Square area: " << area << '\n';
        return area;
    }
};
int main()
{
    Shape* s;
    Square sq(5, 5);
    s = &sq;
    s->get_Area();
    return 0;
}
```

OUTPUT :- Square area: 25.

Use of Virtual Function?

→ It allows us to create a list of base class pointers and call methods of any derived class without even knowing the kind of derived class object.

→ Working of virtual functions (concepts of VTABLE and VPTR).

If a class contains a virtual function then the compiler itself does two things.

1. If object of the class is created then a virtual pointer (VPTR) is inserted as a data member of the class to point to VTABLE of that class.
   For each new object created, a new virtual pointer is inserted as a data member of that class. Virtual pointer is inherited by derived classes.

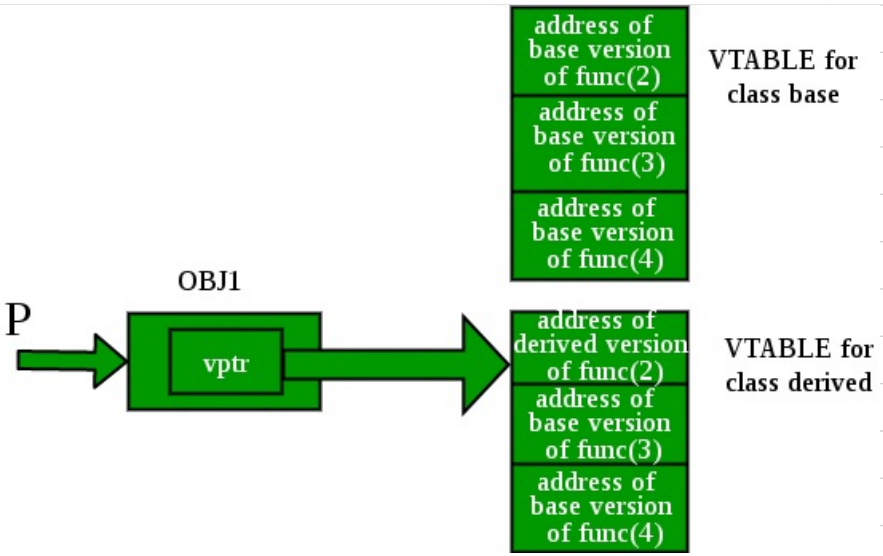
2. Irrespective of object is created or not, class contains as a member a static array of function pointers called VTABLE.

   Virtual Table — A table created at compile time for every single class containing the most derived versions of virtual function only.

   → A virtual table contains one entry for each virtual function that can be

called by objects of the class.

```cpp
#include<iostream>
using namespace std;

class base {
public:
    void fun_1()
    { cout << "base-1\n"; }

    virtual void fun_2() {
         cout << "base-2\n"; }

    virtual void fun_3() {
        cout << "base-3\n"; }

    virtual void fun_4()
    { cout << "base-4\n"; }
};

class derived : public base {
public:
    void fun_1()
    { cout << "derived-1\n"; }

    void fun_2()
    { cout << "derived-2\n"; }

    void fun_4(int x)
    { cout << "derived-4\n"; }
};

int main()
{
    base *p;
    derived obj1;
    p = &obj1;

    p->fun_1();
    p->fun_2();
    p->fun_3();
    p->fun_4();

    return 0;
}

```


address of base version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class base
OBJ1
P
vptr
address of derived version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class derived

Virtual Destructor :- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behaviour.

```cpp
#include <iostream>
using namespace std;

class base {
public:
    base()
    { cout << "Constructing base\n"; }
    ~base()
    { cout<< "Destructing base\n"; }
};

class derived: public base {
public:
    derived()
    { cout << "Constructing derived\n"; }
    ~derived()
    { cout << "Destructing derived\n"; }
};

int main(){
base *b = new derived();
delete b;
getchar();
return 0;
}

```

OUTPUT :-

```
Constructing base
Constructing derived
Destructing base
```

NOTE :- The base class pointer was used to make the derived class object.
Destructor has to be made virtual, otherwise Derive Destructor will not call.

```cpp
#include <iostream>
using namespace std;

class base {
public:
     base()
    { cout << "Constructing base\n"; }
    virtual ~base()
    { cout<< "Destructing base\n"; }
};

class derived: public base {
public:
    derived()
    { cout << "Constructing derived\n"; }
     ~derived()
    { cout << "Destructing derived\n"; }
};

int main(){
base *b = new derived();
delete b;
getchar();
return 0;
}

```

OUTPUT :- Constructing base
Constructing derived
Destructing derived
Destructing base

Q:) What about Virtual Constructor ?

It is not possible. Because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet., So constructor should always be non-virtual.

→ Because the object is not created, virtual construction is impossible.

→ The compiler must know the type of object before creating it.

Error – constructors cannot be declared 'virtual'.

Encapsulation :- Encapsulation is defined as wrapping up of data and information under a single unit.

```cpp
class Addition{
   public:
      Addition(int t = 0) {
         total = t;
      }

      void addNum(int number) {
         total += number;
      }

      int getTotal() {
         return total;
      };

   private:
      int total;
};

int main() {
   Addition a;

   a.addNum(3);
   a.addNum(2);
   a.addNum(8);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Role of access specifiers in encapsulation :-

The process of implementing encapsulation can be sub-divided into two steps :-

1. The data members should be labeled as private using the private access specifiers.

2. The member function which manipulates the data members should be labeled as public using the public access specifier.

→ Data encapsulation led to the important OOP concept of data hiding.

⇒ We can write "read only" or "write only" methods to implement data hiding.

Abstraction:- Data abstraction is a process of providing only the essential details to the outside world and hiding the internal details

- C++ provides a great level of abstraction. for ex- pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

Data Abstraction can be achieved in two ways:-

- Abstraction using classes.
- Abstraction in header files.

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

```cpp
#include <iostream>
using namespace std;
 class Sum
{
private: int x, y, z; // private variables
public:
void add()    {
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
int main()    {
Sum sm;
sm.add();
return 0;
}
```

→ we are not allowed to access x, y, z directly however, we can access them using member function of the class.

Advantages of Data Abstraction :—

→ A programmer does not need to write the low level code.

→ Data abstraction avoids code duplication.

→ Increases reusability.

→ Implementation details of the class are protected from the inadvertent user level errors.

OOPS DESIGN PRINCIPLES

Clean Code ? ⇒ easily readable by other developers.

Why ?

How ?

- → meaningful variable name, function, class etc.
- → Comments.
- → Modular
- → functions
- → short,
- → structured, readable, simple.

DESIGN PRINCIPLES :-

- → DRY : Don't repeat yourself.
- → KISS : Keep it simple stupid.
- → Abstraction :
- → Curly's law :
- → Boy - Scout law :- Do better code quality than what it was

Features of Good Design :-

1. Code Reuse :-
2. Extensibility :- 'change is the only constant thing in programmer's life'.

Naming → Intension - revealing names.
→ descriptive names.

```
int getSum();        ✓        int a  ✗
int getAnswer();     ✓        int b  ✗
```

→ Function Name - Verb, short [20 lines], dedicate to single task.
- should have fewer arguments (Use Helper function).
- sort, reverse, swap, pow.

→ Word's principle -

Class Name - Noun
→ descriptive
→ meaningful
→ intent - revealing

R. C. Martin

1. DRY PRINCIPLE (Don't repeat Yourself)

Curly's law → A variable should mean one thing, and one thing only.

2. KISS (Keep it Simple Stupid)

```
if (a)
 { if(b)
    { if(c)
       { if(d)
          { }
          else { }
       }
    }
 }
```

→ `if(a && b && c && d)` is simple.

Benefits :-

- easy to read/understand
- less time to code.
- Bugs chances are less.
- debug / modify / update.

③ YAGNI :- You ain't gonna need it.

"Always implement things which are actually needed, not the ones you just foresee".

→ Don't Do overengineering.

Benefits :-

- Save time.
- Concise code.

→ Preoptimisation is the root of all evils.

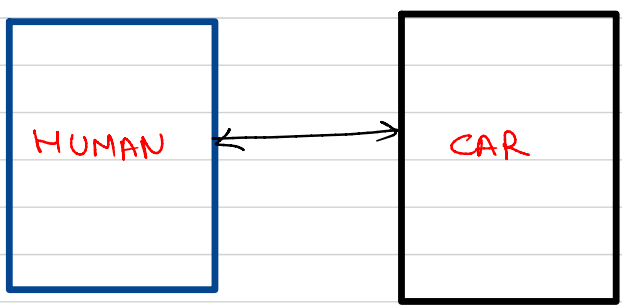
Boy - Scout Law :- The code quality tends to degrade with each change. - (tech debt).

→ 'Always leave the code behind in a better state than you found it".

COHESION & COUPLING :-

Cohesion :- It is the degree of how strongly related & focused are the various responsibilities of a module.
→ Maximum cohesion.

Coupling :- It is the degree to which each module depend on other modules.



→ Required low coupling.

Command – Query Separation : (CQS)

Command → Changes the state but doesn't return any value.

Query → return the state without changing the state.

CQS → It states that every method should be a command that performs an action, or a query that returns data to the caller, but not both.

Is there any exception of CQS?

SOLID - a set of principles.
S - single responsibility principle
O - Open - closed principle.
L - Liskov Substitution Principle
I - Interface Segregation Principle
D - Dependency Inversion Principle.

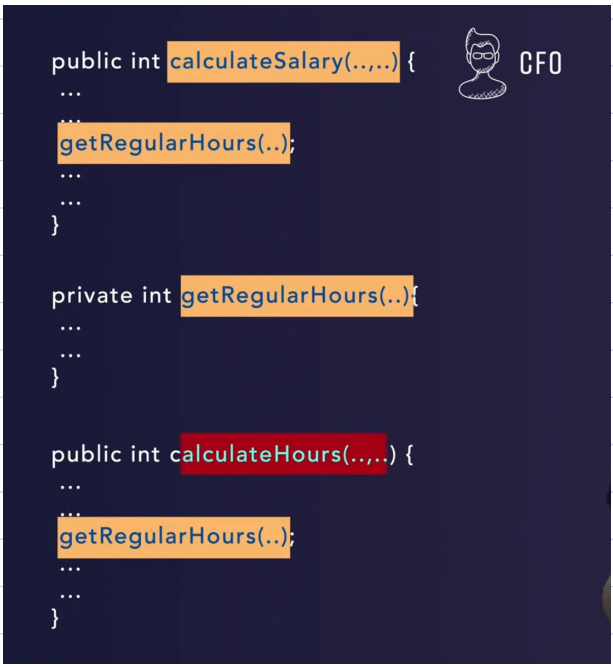
Single Responsibility principle :-

~~A class should have just one reason to change.~~

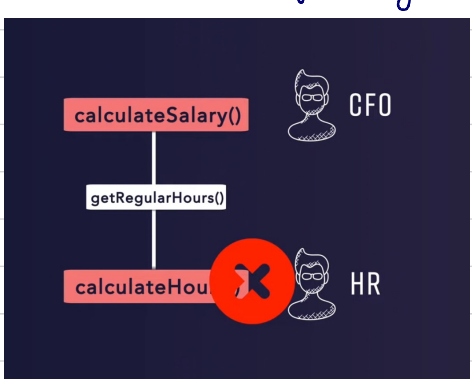
→ Any module (module means a set of functions, class, package, source code) should have a reason to change by only one actor.

Let's say we have a class Employee, have some function calculateSalary(), calculateHours(), saveEmpData().

calculateSalary() — CFO
calculateHours() — Technical
saveEmpData() — HR

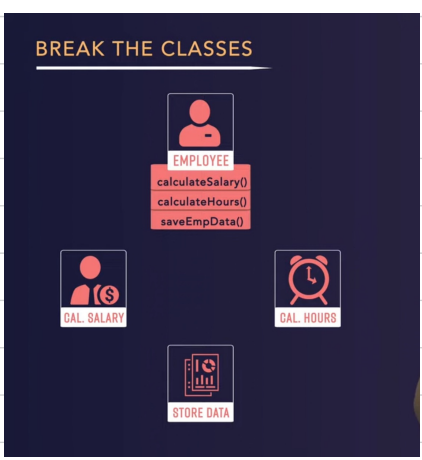


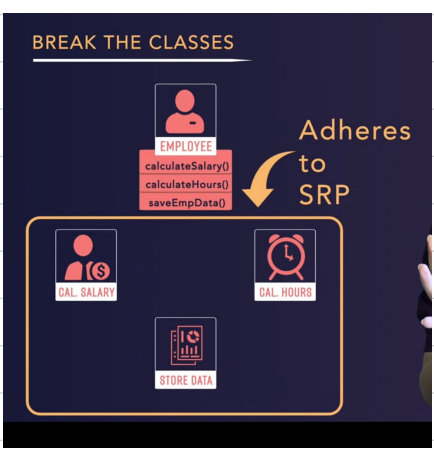
→ If calculateSalary() is used by CFO and it required a method getRegularHours() and make a change in getRegularHours(), it will also change in calculate Hours(), which is managed by someone else.



→ What happened here was, one class exposed two or three different methods, which were corresponding to different stakeholders of the software.
One Actor doesn't have to know about the other actor, but still change in the one of them is reflected the other. It is voilating the Singk Responsibility Principle.

→ Singk Responsibility principle means, that the code you are writing, if that code requested change in one stakeholder, which fullfill one business Requirements till then it is fine.
That means our class can have more than one public method as long as the change in those public method is requested by one stakeholder/group of stakeholders.





→ Create 3 different class, the calculation of salary doesn't need to know about / doesn't need, depend on any method which is used in calculate hours.

→ By decomposing one class into multiple classes, we can actually adhere single responsibility, where the business requirement sits in one particular class, and request to change those logics comes from one actor only.

OPEN - CLOSED PRINCIPLE

→ "Software entities such as classes, modules, functions, etc. should be open for extension, but closed for modification.

→ Any new functionality should be implemented by adding new classes, attributes and methods, instead of changing the current ones or existing ones.

→ Bertrand Meyer originated the term OCP.
→ Robert C. Martin considered this as most important principle.

Implementation Guidelines :-

→ The simplest way to apply OCP is to implement the new functionality on new derived classes.

→ Allow clients to access the original class with abstract interface.

WHY OCP?

→ If Not followed
  → End up testing the entire functionality
  → QA Team need to test the entire flow.

linkedin.com/in/kapilyadav22

→ Costly Process for the Organization
→ Breaks the Single responsibility as well.
→ Maintenance overheads increase on the classes.

```
enum class SensorModel {
    Good,
    Better
};

struct DistanceSensor {
    DistanceSensor(SensorModel model) : mModel{model} {}
    int getDistance() {
        switch (mModel) {
            case SensorModel::Good :
                // Business logic for "Good" model
            case SensorModel::Better :
                // Business logic for "Better" model
        }
    }
};
```

→ It's design doesn't allow to extend the behaviour unless we change the code. for ex- if we want to add new sensormodel, we have to change distanceSensor class to accomodate the new functionality,

→ We start by defining what we want a distanceSensor to do in general. In below code, DistanceSensor() is generic, it doesn't have a model, therefore it doesn't make any sense to a concrete instance of it.
Therefore we defined, DistanceSensor as a pure abstract class or in java we can say 'interface'.

```
struct DistanceSensor {
    virtual ~DistanceSensor() = default;
    virtual int getDistance() = 0;
};

struct GoodDistanceSensor : public DistanceSensor {
    int getDistance() override {
        // Business logic for "Good" model
    }
};

struct BetterDistanceSensor : public DistanceSensor {
    int getDistance() override {
        // Business logic for "Better" model
    }
};
```

→ When we create specializations of the interface, for each of the DistanceSensor model that we need, everytime we want to add new sensor, we can create a new child class, that implements the distanceSensor() interface without changing any existing code.

LISKOV SUBSTITUTION PRINCIPLE (LSP)

→ "Subtypes must be substitutable for their base types".

→ "S is a Subtype of T, then objects of type T may be replaced with objects of type S"

→ Derived types must be completely substitutable for their base types.

→ LSP is a particular definition of a subtyping relation, called (strong) behavioral subtyping.

→ Introduced by Barbara LisKov.

→ Extension of the Open close Principle.

Implementation Guidelines

→ No new exceptions can be thrown by the subtype.

→ clients should not know which specific subtype they are calling.

→ New derived classes just extend without replacing the functionality of old classes.

linkedin.com/in/kapilyadav22

Bad example

```
public class Bird{
    public void fly(){}
}
public class Duck extends Bird{}
```

The duck can fly because it is a bird, but what about this:

```
public class Ostrich extends Bird{}
```

Ostrich is a bird, but it can't fly, Ostrich class is a subtype of class Bird, but it shouldn't be able to use the fly method, that means we are breaking the LSP principle.

Good example

```
public class Bird{}
public class FlyingBirds extends Bird{
    public void fly(){}
}
public class Duck extends FlyingBirds{}
public class Ostrich extends Bird{}
```

ISP - Interface Segregation Principle

"Dependency of one class to another should be on smallest possible interface".

→ clients should not be forced to implement interface they don't use.

→ One fat interface need to be split to many smaller and relevant interfaces so that clients can know about the interfaces that are relevant to them.

→ The ISP was first used and formulated by Robert C. Martin while consulting for Xerox

Case Study

Problem :-

→ Xerox had created a new printer system that could perform a variety of tasks such as stapling and faxing along with the regular printing task.

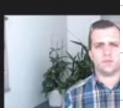
→ The software for this system was created from the ground up.

→ Modifications and Deployment to the system became more complex.

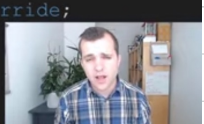
SOLUTION

→ One large Job class is segregated to multiple interfaces depending on the requirement.

```
struct SerialManager {
  virtual ~Manager() = default;
  virtual void registerReceiver(function<void(string)> receiver) = 0;
  virtual void send(string message) = 0;
  virtual void readLine() = 0;
};

struct MySerialManager : public SerialManager {
  void registerReceiver(function<void(string)> receiver) override;
  void send(string message) override;
  void readLine() override;
};
```



```
struct SerialClient {
   virtual ~SerialClient() = default;
  virtual void registerReceiver(function<void(string)> receiver) = 0;
  virtual void send(string message) = 0;
}

struct SerialReader {
  virtual ~SerialReader() = default;
  virtual void readLine() = 0;
};

struct MySerialManager : public SerialClient, public SerialReader {
  void registerReceiver(function<void(string)> receiver) override;
  void send(string message) override;
  void readLine() override;
};
```



```
Animal
{ feed();
  pet();
};
```

```
Dog : public Animal
{ feed();
  pet();
}
```

```
Lion : public Animal
{ feed();
  pet();
}
```

→ Pet cannot be used for Lion, so don't add pet function in base class.

linkedin.com/in/kapilyadav22

D- Dependency Inversion Principle.

"Depend on Abstraction (Interfaces), not on concrete classes."

→ High-level modules should not depend on low-level modules. Both should depend on abstractions.

→ Abstractions should not depend on details. Details should depend on abstractions → interfaces

NOTE:- The interaction between high level and low level modules should be thought of as an abstract interaction between them.

OOPS DESIGN PATTERN

→ formalised Best practices to write clean code.

- Creational – Factory, Abstract Factory, Singleton
- Structural – Bridge, Adaptive, Composite.
- Behavioural – Interpreter, Strategy, observer.

1) Factory Method Design Pattern :- Factory Method design pattern says that just define an interface or abstract class for creating an object but let the subclass decide which class to instantiate.

→ (Subclasses are responsible to create the instances of the class)

→ This pattern is also known as Virtual Constructor.

```cpp
#include<bits/stdc++.h>
using namespace std;

class Vehicle{
    public:
    virtual void createvehicle() = 0;
};

class Bike : public Vehicle{
    public:
    void createvehicle()
    {
        cout<<"Creating Bike"<<endl;
    }
};

class Car : public Vehicle{
    public:
    void createvehicle()
    {
        cout<<"Creating Car"<<endl;
    }
};


int main()
{
    string VehicleType;
    cin>>VehicleType;
    Vehicle* vobj;
    if(VehicleType=="Bike")
       vobj = new Bike();
    else if(VehicleType=="Car")
       vobj = new Car();
    vobj->createvehicle();
    return 0;
}

```

→ In above code, if user wants to add Tempo, bus, metro info, he/she needs to add it in multiple if else,

our user doesn't want to do it.

→ Now, we are going to create a factory, it will take care of entire creation logic, without exposing it to client/user, so client will not bother.

```cpp
#include<bits/stdc++.h>
using namespace std;

class Vehicle{
    public:
    virtual void createvehicle() = 0;
};

class Bike : public Vehicle{
    public:
    void createvehicle()
    {
        cout<<"Creating Bike"<<endl;
    }
};

class Car : public Vehicle{
    public:
    void createvehicle()
    {
        cout<<"Creating Car"<<endl;
    }
};

class Vehiclefactory : public Vehicle{
    public:
    static Vehicle* getVehicle(string VehicleType)
        { Vehicle* vobj;
        if(VehicleType=="Bike")
        vobj = new Bike();
        else if(VehicleType=="Car")
        vobj = new Car();
        return vobj;
    }
};

//client code
int main()
{
    string VehicleType;
    cin>>VehicleType;
    Vehicle* vobj = Vehiclefactory::getVehicle(VehicleType);
    vobj->createvehicle();
    return 0;
}
```

→ We created the function as static, so that we can access the function without the object of class.

→ Now, we will write our logic in factory, and if any change required, client will not bothered, we only need to do change in factory.

→ Library should be responsible to decide which obj type to create based on input.

→ Client should just call library's factory and pass type without worrying about actual implementation of creation of object.

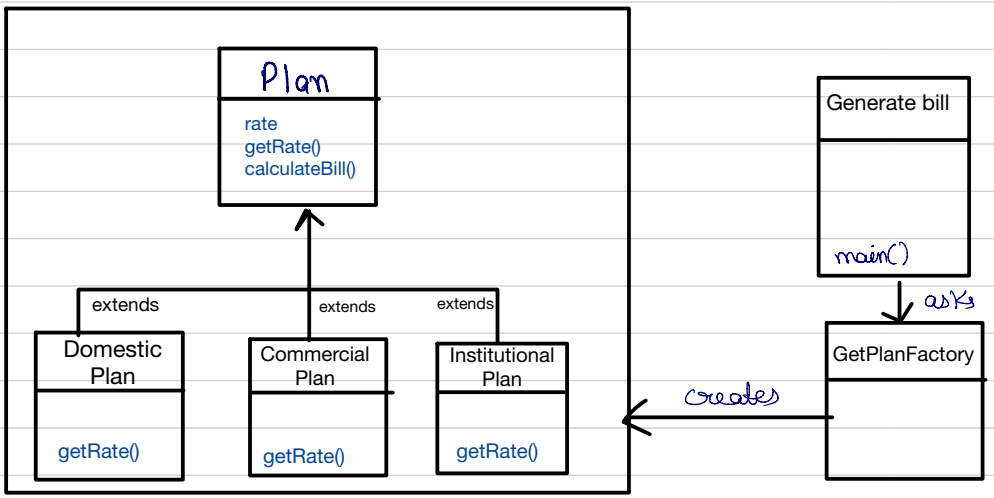
Advantages of Factory Design Pattern :-

- Factory Design Pattern allows the sub-class to choose the type of objects to create.
- It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code.

Usage of Factory Design Pattern :-

- When a class doesn't know what sub-classes will be required to create.
- When a class wants that its sub-classes specify the objects to be created.

Ex - 2 :-


Plan
rate
getRate()
calculateBill()
Generate bill
main()
asks
extends
extends
extends
Domestic Plan
getRate()
Commercial Plan
getRate()
Institutional Plan
getRate()
creates
GetPlanFactory

```cpp
#include <iostream>
using namespace std;

 class Plan{
         protected:
          double rate;
          public:
         virtual void getRate()=0;
         void calculateBill(int units){
             cout<<units*rate<<endl;
          }
};

      class  DomesticPlan : public Plan{
      public :
      void getRate(){
             rate=3.50;
        }
      };

    class  CommercialPlan : public Plan{
        public:
        void getRate(){
        rate=7.50;
   }
        };

   class  InstitutionalPlan : public Plan{
    public:
     void getRate(){
        rate=5.50;
   }
   };


```

```cpp
class GetPlanFactory : public Plan{
       public:
       static Plan* getPlan(string planType){
           Plan* pl;
            if(planType==""){
             return NULL;
            }
          if(planType=="DOMESTICPLAN") {
                pl = new DomesticPlan();
              }
           else if(planType=="COMMERCIALPLAN"){
                pl = new CommercialPlan();
            }
          else if(planType=="INSTITUTIONALPLAN") {
                pl =  new InstitutionalPlan();
          }
      return pl;
   }
};


int main(){
    string planType;
    int units;
    cout<<"Enter the name of plan ";
    cin>>planType;
    cout<<"Enter the number of units ";
    cin>>units;
    cout<<"Bill amount for "<<planType<<" of "<<units<<"units is: "<<endl;
    Plan* pl=  GetPlanFactory::getPlan(planType);
    pl->getRate();
    pl->calculateBill(units);
    return 0;
    }
```

2. SINGLETON DESIGN PATTERN :-

```
lass Logger{
    static int ctr;
    public:
    Logger()
    {  ctr++;
        cout<<" New Instance Created. No of instances are "<<ctr<<endl;
  }
    void Log(string msg)
    { cout<<msg<<endl; }
};
int Logger::ctr = 0;


int main()
{
    Logger* logger1 = new Logger();
    logger1->Log("this msg is from user 1 ");

    Logger* logger2 = new Logger();
    logger2->Log("this msg is from user 2");

    return 0;
}

```

→ New Instance Created. No. of instances are 1
this message is from user 1.

→ New Instance created. No. of instances are 2
this message is from user 2.

→ I don't want my users to be able to create logger object.

→ I want to restrict them to able to access logger constructor.

1. Restrict users from creating object of class.

→ By making the constructor private, we are restricting users from creating object or calling the constructor itself.

```cpp
class Logger{
    static int ctr;
    //it is the single instance, which we will use for all users
    static Logger *loggerinstance;
    Logger()
    {   ctr++;
        cout<<" New Instance Created. No of instances are "<<ctr<<endl; }

    public:
    static Logger *getlogger()
    {   if(loggerinstance==nullptr)
        {  loggerinstance = new Logger(); }
        return loggerinstance;
    }

    void Log(string msg)
    { cout<<msg<<endl; }

};
    Logger *Logger :: loggerinstance = nullptr;
    int Logger::ctr = 0;

int main()
{
    Logger* logger1 = Logger::getlogger();
    logger1->Log("this msg is from user 1 ");

    Logger* logger2 = Logger::getlogger();
    logger2->Log("this msg is from user 2");

    return 0;
}
```

l Yadav

⇒ Code fails in Multithreading Case.

Suppose there are two threads working parallely and they try to create a logger instance, the first will create a loggerinstance because it is null, same second thread will also do the same, so total 2 instances will be created.

Use -Mutex,

Kapil Yadav

```cpp
#include<bits/stdc++.h>
#include <thread>
#include<mutex>

using namespace std;

class Logger{
    static int ctr;
    //it is the single instance, which we will use for all users
    static Logger *loggerinstance;
    static mutex mtx;

    //constructor private
    Logger()
    {   ctr++;
        cout<<" New Instance Created. No of instances are "<<ctr<<endl; }

    public:
    static Logger *getlogger()
    {   mtx.lock();
        if(loggerinstance==nullptr)
        {  loggerinstance = new Logger(); }
        mtx.unlock();
        return loggerinstance;
    }

    void Log(string msg)
    { cout<<msg<<endl; }

};
    Logger *Logger :: loggerinstance = nullptr;
    int Logger::ctr = 0;
    mutex Logger::mtx;

void user1logs()
{
     Logger* logger1 = Logger::getlogger();
    logger1->Log("this msg is from user 1 ");
}

void user2logs()
{
    Logger* logger2 = Logger::getlogger();
    logger2->Log("this msg is from user 2");
}

int main()
{
   thread t1(user1logs);
   thread t2(user2logs);

    t1.join();
    t2.join();
    return 0;
}
```

Our code is now safe on multithreading but, we don't need mutex all the time, because first time, when loggerinstance = NULL, and multiple threads trying to access loggerinstance, we need lock, but once loggerinstance is created, we don't need lock, so put double check if (loggerinstance == nullptr) then only use lock,

Kapil Yadav

Double-checked Locking :-

→ Check for loggerInstance == nullptr, if we only need a lock, when loggerInstance == null, so check this condition, because locks are expensive.

```cpp
#include<bits/stdc++.h>
#include <thread>
#include<mutex>

using namespace std;

class Logger{
    static int ctr;
    //it is the single instance, which we will use for all users
    static Logger *loggerinstance;
    static mutex mtx;

    //constructor private
    Logger()
    {   ctr++;
        cout<<" New Instance Created. No of instances are "<<ctr<<endl; }

    public:
    static Logger *getlogger()
    {   if(loggerinstance==nullptr)
        { mtx.lock();
        if(loggerinstance==nullptr)
        {  loggerinstance = new Logger(); }
        mtx.unlock();
        }
        return loggerinstance;
    }

    void Log(string msg)
    { cout<<msg<<endl; }

};
    Logger *Logger :: loggerinstance = nullptr;
    int Logger::ctr = 0;
    mutex Logger::mtx;

void user1logs()
{
     Logger* logger1 = Logger::getlogger();
    logger1->Log("this msg is from user 1 ");
}

void user2logs()
{
    Logger* logger2 = Logger::getlogger();
    logger2->Log("this msg is from user 2");
}

int main()
{
   thread t1(user1logs);
   thread t2(user2logs);

    t1.join();
    t2.join();
    return 0;
}
```

POINTS TO KEEP IN MIND:-

→ We want to restrict users to access the constructor but there are multiple ways to access the private constructor

1.) Make copy constructor as private.

2.) Make equal to operator, overloading as private.

From C++11, we also need to use =delete to restrict user to copy operations.

```cpp
#include<bits/stdc++.h>
#include <thread>
#include<mutex>

using namespace std;

class Logger{
    static int ctr;
    //it is the single instance, which we will use for all users
    static Logger *loggerinstance;
    static mutex mtx;

    //constructor private
    Logger()
    {   ctr++;
        cout<<" New Instance Created. No of instances are "<<ctr<<endl; }
    Logger(const Logger&);
    Logger operator = (const Logger&);


    public:
    static Logger *getlogger()
    {   if(loggerinstance==nullptr)
        { mtx.lock();
        if(loggerinstance==nullptr)
        {  loggerinstance = new Logger(); }
        mtx.unlock();
        }
        return loggerinstance;
    }

    void Log(string msg)
    { cout<<msg<<endl; }

};
    Logger *Logger :: loggerinstance = nullptr;
    int Logger::ctr = 0;
    mutex Logger::mtx;

void user1logs()
{
     Logger* logger1 = Logger::getlogger();
    logger1->Log("this msg is from user 1 ");
}

void user2logs()
{
    Logger* logger2 = Logger::getlogger();
    logger2->Log("this msg is from user 2");
}

int main()
{
   thread t1(user1logs);
   thread t2(user2logs);

    t1.join();
    t2.join();
    return 0;
}
```

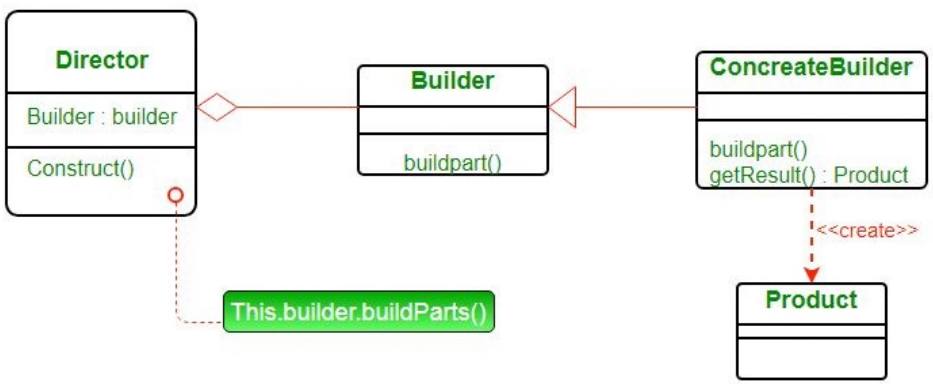
restrict users

3 BUILDER DESIGN PATTERN:-

→ Whenever we are building very complex structure which has a lot of configurations in it.

Builder Pattern says that "construct a complex object from simple objects using step-by-step approach".

UML diagram of Builder Design pattern



- **Product –** The product class defines the type of the complex object that is to be generated by the builder pattern.
- **Builder –** This abstract base class defines all of the steps that must be taken in order to correctly create a product. Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses. The GetProduct method is used to return the final product. The builder class is often replaced with a simple interface.
- **ConcreteBuilder –** There may be any number of concrete builder classes inheriting from Builder. These classes contain the functionality to create a particular complex product.
- **Director –** The director-class controls the algorithm that generates the final product object. A director object is instantiated and its Construct method is called. The method includes a parameter to capture the specific concrete builder object that is to be used to generate the product. The director then calls methods of the concrete builder in the correct order to generate the product object. On completion of the process, the GetProduct method of the builder object can be used to return the product.

Advantages of Builder Design Pattern :-

- It provides clear separation between the construction and representation of an object.
- It provides better control over construction process.
- It supports to change the internal representation of objects.

Advantages of Builder Design Pattern

- The parameters to the constructor are reduced and are provided in highly readable method calls.
- Builder design pattern also helps in minimizing the number of parameters in the constructor and thus there is no need to pass in null for optional parameters to the constructor.
- Object is always instantiated in a complete state
- Immutable objects can be built without much complex logic in the object building process.

Disadvantages of Builder Design Pattern

- The number of lines of code increases at least to double in builder pattern, but the effort pays off in terms of design flexibility and much more readable code.
- Requires creating a separate ConcreteBuilder for each different type of Product.

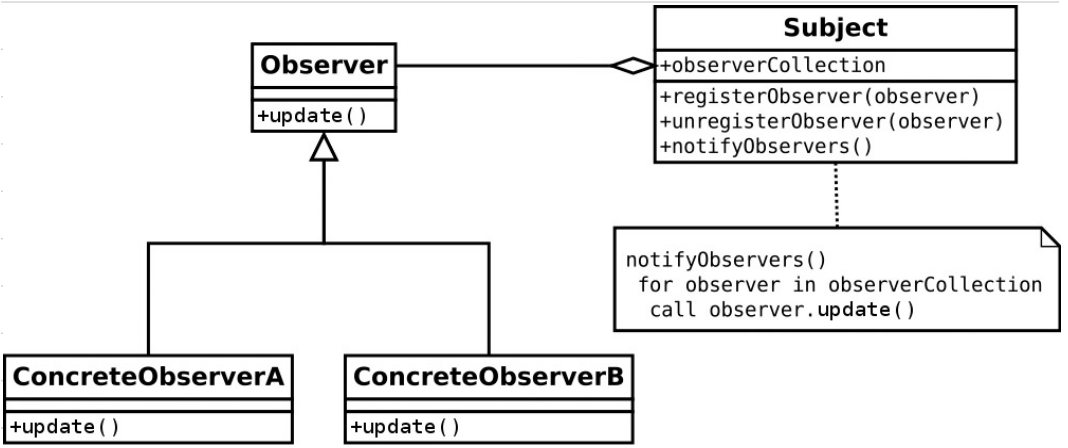
```cpp
#include<bits/stdc++.h>
using namespace std;

class Desktop{
   string monitor;
   string keyboard;
   string mouse;
   string speaker;
   string ram;
   string processor;
   string motherboard;

   public:
   void setMonitor(string pMonitor){
      monitor = pMonitor; }
   void setKeyboard(string pKeyboard){
     keyboard= pKeyboard;
   }
   void setMouse(string pMouse){
     mouse = pMouse;
   }
   void setSpeaker(string pSpeaker){
     speaker=pSpeaker;
   }
   void setRam(string pRam) {
     ram = pRam;
   }
   void setProcessor(string pProcessor){
     processor = pProcessor;
   }
   void setMotherboard(string pMotherboard){
     motherboard = pMotherboard;
   }
   void showSpecs()
   {
     cout<<"------------------------------------------------------------
  --------------------------"<<endl;
     cout<<" Monitor "<<monitor<<endl;
     cout<<" Keyboard "<<keyboard<<endl;
     cout<<" Mouse "<<mouse<<endl;
     cout<<" Speaker "<<speaker<<endl;
     cout<<" RAM "<<ram<<endl;
     cout<<" Processor"<<processor<<endl;
     cout<<" Motherboard"<<motherboard<<endl;
     cout<<"------------------------------------------------------------
  --------------------------"<<endl;
   }
};
```

```cpp
class DesktopBuilder{
   protected:
   Desktop* desktop;
   public:
   // In the constructor, we are initialising desktop, as soon as the builder is created, the product is
   // created correspondingly.
   DesktopBuilder()
   {
      desktop = new Desktop();
   }
   virtual void buildMonitor() =0;
   virtual void buildKeyboard() =0;
   virtual void buildMouse() =0;
   virtual void buildSpeaker() =0;
   virtual void buildRam() =0;
   virtual void buildProcessor() =0;
   virtual void buildMotherboard() =0;
   virtual Desktop* getDesktop(){
      return desktop;
   }
};

class DellDesktopbuilder : public DesktopBuilder{

void buildMonitor(){
   desktop->setMonitor("Dell Monitor");
}

void buildKeyboard(){
   desktop->setKeyboard("Dell Keyboard");
}
void buildMouse(){
   desktop->setMouse("Dell Mouse");
}
void buildSpeaker(){
   desktop->setSpeaker("Dell Speaker");
}
void buildRam(){
   desktop->setRam("Dell Ram");
}
void buildMotherboard(){
   desktop->setMotherboard("Dell MotherBoard");
}
void buildProcessor(){
   desktop->setProcessor("Dell Processor");
}
};
```

```
class HpDesktopbuilder : public DesktopBuilder{

void buildMonitor(){
   desktop->setMonitor("Hp Monitor");
}

void buildKeyboard(){
   desktop->setKeyboard("Hp Keyboard");
}
void buildMouse(){
   desktop->setMouse("Hp Mouse");
}
void buildSpeaker(){
   desktop->setSpeaker("Hp Speaker");
}
void buildRam(){
   desktop->setRam("Hp Ram");
}
void buildMotherboard(){
   desktop->setMotherboard("Hp MotherBoard");
}
void buildProcessor(){
   desktop->setProcessor("Hp Processor");
}
};

class DesktopDirector{
   private:
   DesktopBuilder* desktopBuilder;
   public:
   DesktopDirector(DesktopBuilder* pDesktopBuilder){
        desktopBuilder = pDesktopBuilder;
   }
   Desktop* BuildDesktop(){
      desktopBuilder->buildMonitor();
      desktopBuilder->buildKeyboard();
      desktopBuilder->buildMotherboard();
      desktopBuilder->buildMouse();
      desktopBuilder->buildProcessor();
      desktopBuilder->buildRam();
      desktopBuilder->buildSpeaker();
      return desktopBuilder->getDesktop();
   }
};
```

```
int main()
{
   HpDesktopbuilder* hpdesk = new HpDesktopbuilder();
   DellDesktopbuilder* DellDesktop = new DellDesktopbuilder();

   DesktopDirector* director1 = new DesktopDirector(hpdesk);
   DesktopDirector* director2 = new DesktopDirector(DellDesktop);

   Desktop* desktop1 = director1->BuildDesktop();
   Desktop* desktop2 = director2->BuildDesktop();
   desktop1->showSpecs();
   desktop2->showSpecs();

   return 0;

}

```

4. OBSERVER DESIGN PATTERN:-



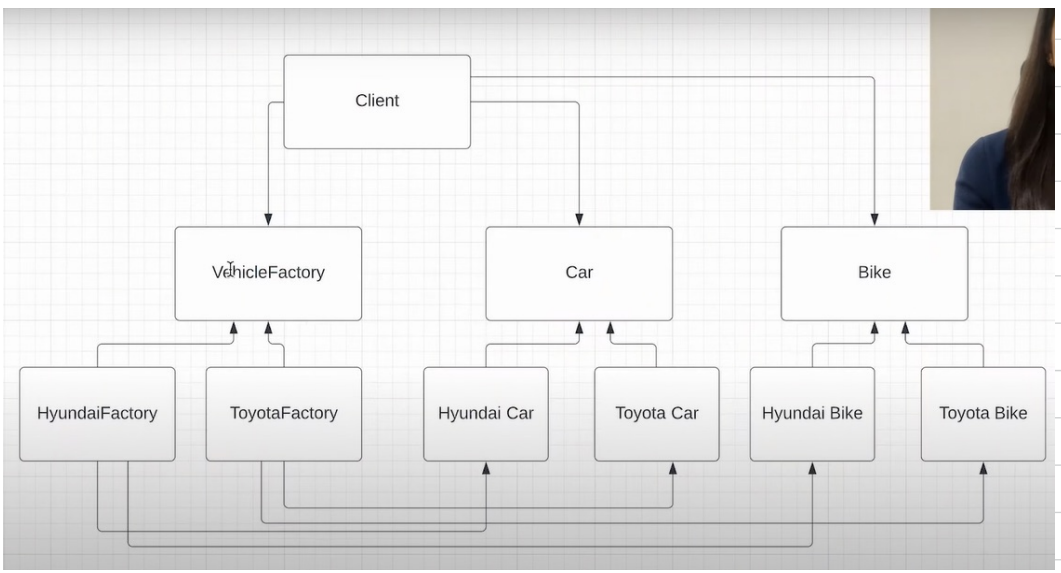
→ An observer Pattern says that "just define one-to-one dependency so that when one object changes states, all its dependents are notified and updated automatically.
Ex- group notifications.

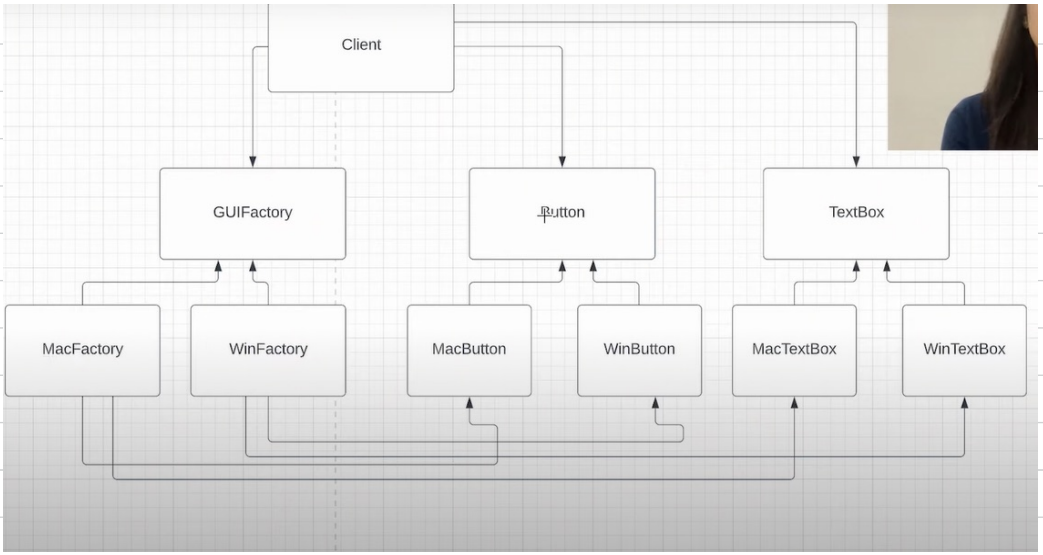
```cpp
#include<iostream>
#include<list>
using namespace std;

class ISubscriber{
    public:
      virtual void notify(string msg)=0;
};

class User: public ISubscriber{
    private:
        int userId;
    public:
        User(int userId){
            this->userId = userId;
        }
        void notify(string msg){
            cout<<" User  "<<userId<< " received message "<<msg<<endl;
        }
};

class Group{
    private:
        list<ISubscriber*> users;
    public:
        void subscribe(ISubscriber* user)
             {   users.push_back(user);
             }
        void unsubscribe(ISubscriber* user)
           { users.remove(user);
           }
        void notify(string msg)
        {
            for(auto user: users)
            user->notify(msg);
        }
};


int main()
{
    Group* group = new Group;
    User* user1 = new User(1);
    User* user2 = new User(2);
    User* user3 = new User(3);

    group->subscribe(user1);
    group->subscribe(user2);
    group->subscribe(user3);

    group->notify("new msg");

    group->unsubscribe(user1);
    group->notify("New New Message");

    return 0;
}

```

5. Abstract Factory Design:

→ Factory design pattern was creating concrete classes or object.

→ Abstract factory design pattern is going to create factories that is going to create object.




Client
GUIFactory
Button
TextBox
MacFactory
WinFactory
MacButton
WinButton
MacTextBox
WinTextBox

```cpp
#include<bits/stdc++.h>
using namespace std;

class IButton{
    public:
    virtual void press() = 0;
};

class Macbutton : public IButton{
    public:
        void press(){
            cout<<" Mac Button Pressed"<<endl;
        }
};
class Winbutton : public IButton{
    public:
        void press(){
            cout<<" Windows Button Pressed"<<endl;
        }
};


class ITextBox{
    public:
    virtual void showtext() = 0;
};

class MacTextBox : public ITextBox{
    public:
        void showtext(){
            cout<<" Showing Mac Textbox"<<endl;
        }
};
class WinTextBox : public ITextBox{
    public:
        void showtext(){
            cout<<" Showing Mac Textbox"<<endl;
        }
};

```

```
class IFactory{
    public:
      virtual IButton* CreateButton( ) = 0;
      virtual ITextBox* CreateTextBox( ) = 0;

};

class MacFactory: public IFactory{
    public:
    IButton* CreateButton( ){
        return new Macbutton( );
    }
    ITextBox* CreateTextBox( ) {
        return new MacTextBox( );
    }
};

class WinFactory: public IFactory{
    public:
    IButton* CreateButton( ){
        return new Winbutton( );
    }
    ITextBox* CreateTextBox( ){
        return new WinTextBox( );
    }
};

class GUIAbstractFactory{
    public:
        static IFactory* CreateFactory(string osType){
            if(osType=="Windows")
              { return new WinFactory( );
              }
            else if(osType=="mac")
               { return new MacFactory( );
               }
            return new MacFactory( );
        }
};
```

```cpp
int main()
{
    cout<<"Enter your machine OS"<<endl;
    string osType;
    cin>>osType;

    IFactory* fact = GUIAbstractFactory::CreateFactory(osType);
    IButton* button = fact->CreateButton();
    button->press();

    ITextBox* textbox = fact->CreateTextBox();
    textbox->showtext();

    return 0;
}
```