Structural programming -

Code is exec. line by line and no func. (time consuming)

Procedural programming -

Divided into no. of func that perform tasks.

eg. main → func 1 → func 2 → return

- less complex
- executable

[C, FORTAN, COBOL]

Top → bottom approach

Disadvantages →

- No data hiding
- Global variable are used (more prone to bugs)
- not based on real world programming.

Object Oriented programming -

- Data is treated as critical element and is tied to the functions (cannot be manipulated by ext. func)
- Bottom → Top approach

[C++, C#, Python]

* Bind together data & fn that operate on them so that no other part of code can access this data except fn.

Object → real world identity

class → blueprint of object to be created. user defined data type with data members & functions.

[Govt.]

Electricity

newcon()
close()
bill()

Water

Education

Object

Static Allocation
(stack)
(compile time)

eg int a;

Dynamic Allocation
(heap)
(run time)

eg. int *a = new int();

Class

- Instance of class is object
- Members are private by default.
- NULL values are possible

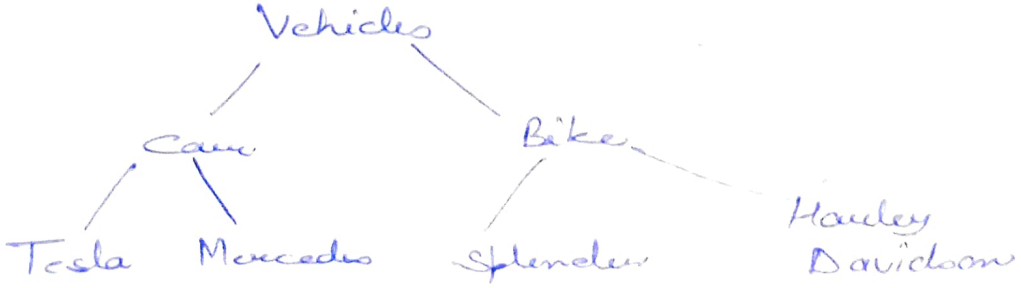
Structure

- Instance of structure is structure variable.
- Members are public by default.
- Null values not possible

- **Abstraction**
  Providing only necessary information to end user.

- **Encapsulation**
  Wrapping up of data & function in a single unit.
  Data is not directly accessible to outside function, any modification of data can be done by member fn.
  → data hiding
  → Avoid mishandling
  eg. buttons of TV

Inheritance
Object of one class can acquire prop. of another class
→ Reusability



- Many forms

Polymorphism

When some fn passes diff. behaviours in diff. situations.

Compile Time

- **Function Overloading** ⇓ Multiple func. with same name but diff. parameters
  - no. of arg. diff.
  - diff. type of arg.
- **Operator Overloading** ⇓ Operators with special meaning.

Run Time

- **Function Overriding** ⇓ Derived class has def. for one of member func. of base class.
- **Virtual Function** ⇓ Member func. i.e. declared in base class using keyword 'virtual' & is re-defined (overridden) in the derived class.

Function overloading –

```
class Display{
  public:
    void func(int x)
        cout << "x = " << x;
    void func(double z)
        cout << "z = " << z;
    Void func(int x, int y)
        cout << "x and y = " << x << y;
};

int main(){
   Display obj;
   obj.func(7);
   obj.func(8.316);
   obj.func(3,4);
   return 0;
}
```

Output –

```
x = 7
z = 8.316
x and y = 3, 4
```

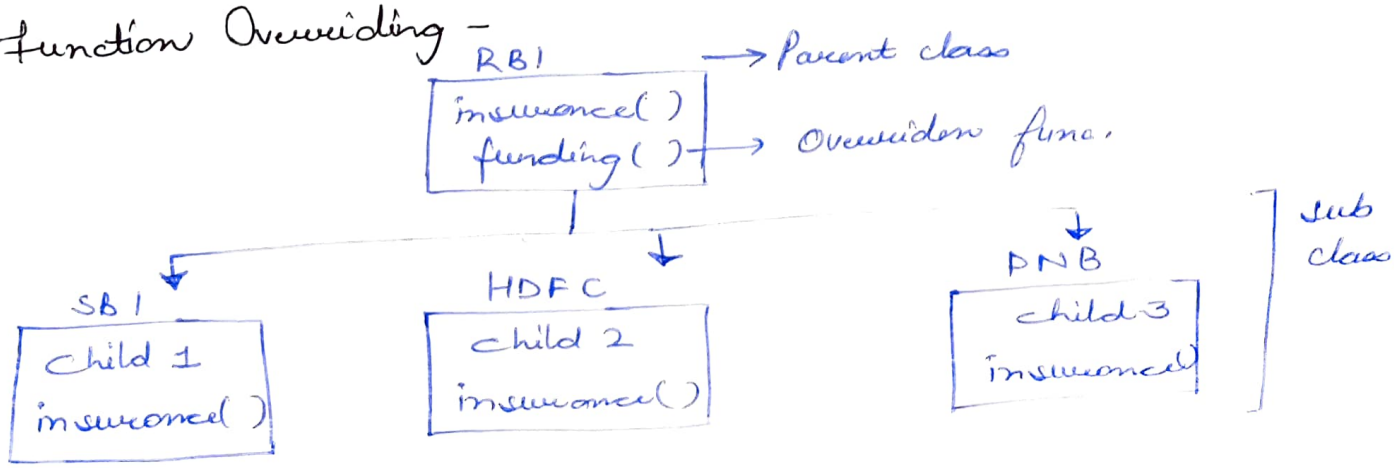
Operator Overloading -

Output -
12 + i9

```
class complex{
  private:
    int real, img;
  public:
    Complex(int r=0 , int i=0){
        real = r ;
        img = i ;
    }
    Complex operator + (Complex const & obj){
        Complex res;
        res.real = real + obj.real;
        res.img = img + obj.img;
        return res;
    }
    void print(){
        cout << real << "+ i" << img;
    }
};

int main(){
    Complex c1(10,5), c2(2,4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Function Overriding -



```
class Parent {
  public:
    void fun() {
        cout << "Base";
    }
};
```

```
class Child : public Parent {
  public:
    void fun() {
        cout << "Derived";
    }
};
```

```
Child child_derived;
child_derived.fun();
```
⟶ Derived

```
child_derived.Parent::fun();
```
⟶ Base

```
Parent *ptr = &Child_derived;
ptr->fun();
```
⟶ Base

Virtual function —
Member function that is declared within a base class and is redefined (overridden) by derived class.

- Virtual func cannot be static
- " " can be friend func. of another class.
- " " should be accessed using pointers for runtime polymorphism.
- Virtual constructor ✗, Virtual destructor ✓

```
class base {
  public:
    virtual void print() {
        cout << "print base";
    }
    void show() {
        cout << "show base";
    }
};
```

```
class derived : public base {
  public:
    void print() {
        cout << "print der";
    }
    void show() {
        cout << "show der";
    }
};
```

Virtual function -

```
base* bp;
derived d;
bp = &d;
bp -> print();
bp -> show();
```

Virtual fn binded at runtime

Non-virtual fn binded at compile time

Output -
print derived
show base

- Late binding (Runtime) → only through pointers of base class
  → acc. to content of pointer
- Early binding (Compile time) → acc. to type of pointer.

Working — If object of class is created Vptr is inserted as data member of class to point to Vtable of that class.

Limitations → slow
→ difficult to debug

Pure virtual functions -

If function is redefined in derived class then base class function rarely does any task
'Do-nothing' function may be defined as —

```
virtual void display() = 0;
```

* Class containing virtual fun (pure) cannot be used to create objects of its own and serve to provide some traits to derived class
  ↳ Abstract Base Class

Access Specifiers

- Private
- Public
- Protected

```
class base {
    private:
        int a;
    protected:
        int b;
    public:
        int c;
        void funbase() {
            a = 1
            b = 2
            c = 3
        }
};

class derived : base {
    public:
        void funder() {
            b = 20
            c = 30
        }
};

int main() {
    base x;
    x.c = 30
    return 0;
}
```

→ private not accessible in derive class

→ private & protected inaccessible in class object

Getters & Setters (Mutator) (Accessor)

```
class student {
    private:
        int rollno;
    public:
        int age;
        void setrollno(int r){
            rollno = r;
        }
        void getrollno(){
            return rollno;
        }
};
```

```
student s;
s.setrollno(55);   ✓
s.rollno = 55;     ✗
s.getrollno();
```

Constructor

- Initialise data members of class (provides data for object)
- Involved at time of object creation
- No return type & value
- Name same as class name

Non-parameterised (Default) | Parameterised | Copy

Default Constructor –

- No argument

Even if we do not define a constructor, compiler supplies default cons.

- No return type
- Cannot be virtual

```
class student{
    public:
        int age, rollno;
        student(){
            cout << "Cons";
        }
};
```

```
int main(){
    student s1;
}
```

Output – Cons

* By default constructor is public, if cons is private we can initiate using friend class.

```
class A{
    private:
        A(){
            cout<<"a";
        }
        friend class B.
};
```

```
class B{
    public:
        B(){
            A aa;
            cout<<"b";
        }
};
```

```
int main(){
    B b;
    return 0;
}
```

output -
a
b

Parameterised cons -

- takes arguments

```
student(int a, int r){
    age = a
    rollno = r
}
```

```
int main(){
    student s(20, 55);
    return 0;
};
```

Objects can be created in 2 ways -

- Explicit

```
student s = student(20,55)
```

- Implicit

```
student s(20, 55)
```

* outside class

```
class student{
    ---
};
student::student(){
}
```

Copy Constructor –

```
student (int age, int rollno){
    this -> age = age;
    this -> rollno = rollno;
}
```

this → special keyword that holds address of current object

Copy Assignment operator –

```
student & operator = (student &s){
    age = this -> age
    rollno = this -> rollno
}
```

~~Test ts~~

```
student s1, s2;
s2 = s1;
```

Destructor

- never takes argument nor does it returns any value.
- To deallocate memory for object.

```
~ student(){
}
```

- static ✗, const ✗, virtual ✓
- cannot be overloaded
- public / private

```
class {
    constructor
    mutators  → setlength(), setbreadth()
    accessor  → getlength(), getbreadth()
    facilitators → area()
    Inspectors → issquare()
    destructor
}
```

Inheritance

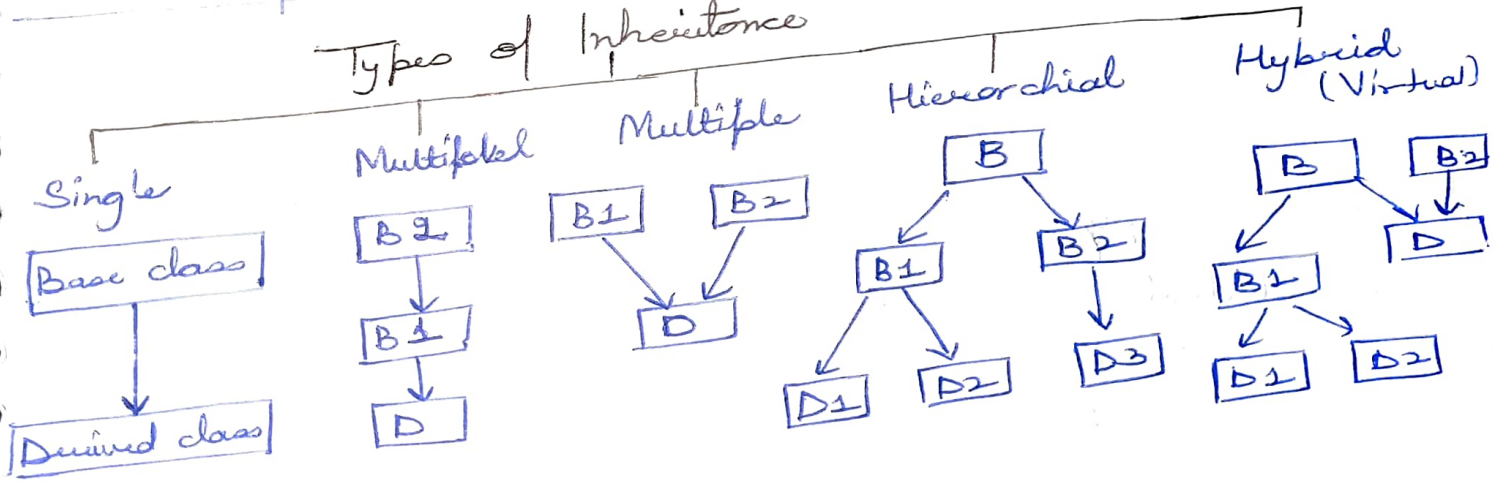
class subclass : access mode base class {

}

If access mode = private →
public member of base class → private of derived

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible	Not acc.	Not acc.

Types of Inheritance




```
class rectangle {
    private:
        int length, breadth;
    public:
        rectangle (int l=1, int b=1) {
            length = l
            breadth = b
        }
        void setlength (int l)
            length = l
```

```
    void setbreadth (int b)
        breadth = b;
    int area () {
        return getlength() * getbreadth();
    }
};

class cuboid : public rectangle {
    private:
        int height;
    public:
        cuboid (int h)
            height = h;
        int volume () {
            return getlength() * getbreadth() * getheight();
        }
};
```

☆ Shallow Copy

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

        void Display() {
            cout << age << "  " << name;
        }
};
```

```
char name = "abcd"
int a = 19
student s1 (a, name);
s1. Display();

name[0] = 'x'
a = 20
student s2 (a, name)
s2. Display();
s1. Display();
```

Output -

```
19  abcd
20  xbcd
19  xbcd
```

Rather than copying the entire name array, its just copying 0th index of array

* Deep Copy (preferred)

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

Output -

```
19  abcd
20  xbcd
19  abcd
```

Rather than copying through address, new array should have been created & then submitted.

Shallow Copy	Deep Copy
• Stores ref. of objects to original memory address	• stores copies of objects value,
• Reflects changes made to new/copied object in original object.	• doesn't reflect changes made to new/ copied object in original object.
• Stores copy of original object & points ref. to objects.	• Stores copy of original object & recursively copies objects as well.
• Fast	• Slower.

Copy Constructor

- Inbuilt → shallow copy → if we do not want to alter 't'

```
Student (student const &t) {
    cout << "Cons called";
    rno = t.rno;
}
```

→ otherwise goes in infinite loop as everytime in para it creates an object of t then goes in cons.

```
student z(40), z1;
z1 = z;        → Copies but cons. is not called
student z2 = z; → Cons. called.
```

- Binding – Converting identifiers into address

Dynamic binding – compiler adds code and address at run time.

include → way of including standard or user defined file in a program. (file inclusion)

eg # include <iostream>
include "abc.c"

Namespace – Instead of writing func. declaration in header file we create a namespace in that.

- If in a prog. we include 2 header files & both contain a func. of same name, it will lead to contradiction. To avoid conflict, namespace is used.

```
# include <iostream>
using namespace std; ——→ Global scope

namespace f1 {
    int x = 10;
    void fun() {
        cout << "f of f1";
    }
}

namespace f2 {
    int x = 7;
    void fun() {
        cout << "f of f2";
    }
}
```

```
int main(){
    cout << f1::x;
    f1:: fun();
    cout << f2: x;
    f2: fun();
    return 0;
}
```

Output -
10
f of f1
7
f of f2

* namespace as = peaul space;
  peaul space :: x = 5;

cout & cin are part of std namespace ↴
  std:: cout << - - -
  std:: cin >> - - -

Generic pointer -
- pointer of type void void * gp;
- cannot be dereferenced.

typecasting -
  reinterpret - cast < int * > (p);

Problem with normal pointers

- Memory Leaks
  When memory is repeatedly allocated but never freed
  Excessive memory usage & system crash;
- Dangling Pointer
  When object is deallocated without modifying value of pointer
- Wild Pointer
  pointer declared & allocated memory but never initialised to point to any valid address.
  eg. a pointer is pointing an object (later deallocated) but this pointer is still in memory & pointing.

~~void fun() {~~ Memory Limit Exceed.

```
class Rectangle {
    private:
        int l, b;
};

Void fun(){
    Rectangle *p = new Rectangle();
}

int main(){
    while(1)
        fun();
}
```

→ Creates a pointer pointing Rectangle object. When fun() ends p will be destroyed (local variable) but memory it consumed won't be deallocated as we didn't used delete p;

Smart Pointer —

```
class SmartPtr {
    int *ptr;
public:
    explicit SmartPtr (int *p = Null){
        ptr = p;
    }
    ~SmartPtr() {
        delete (ptr);
    }
    int & operator *() {
        return *ptr;
    }
};

int main(){
    SmartPtr (new int());
    *ptr = 20;
    cout << *ptr;
}
```

Pointer

* Variable that maintains a memory address and data type information about memory location.
* Not destroyed in any form when goes out of scope.

Smart Pointer

* It's a pointer-wrapping stack-allocated object.
* It destroys itself when goes out of its scope.

Member functions

Simple | Const | Static | Inline | Friend

declared inside class & defined inside/outside class

Inline –

- request not command
- expands in line when called, whole code is inserted at pt. of function call.

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

int main(){
    cout << add(2,3);
}
```

A lot of overhead tasks are performed like saving registers, pushing arg. to stack & returning to calling func. ∴ Time consuming. Inline func. is used to solve these overheads

Const

- cannot modify object

```
int fun(a) const {
    ++x
}
```

(int x)

Error

```
int fun() {
    return x;
}
```

(const x)

Error

Static

- once declared is allocated with memory that can't be changed

```
Static void fun() {
}
```

- cannot access ordinary data members/ func.
- can access static data inside/ outside class.

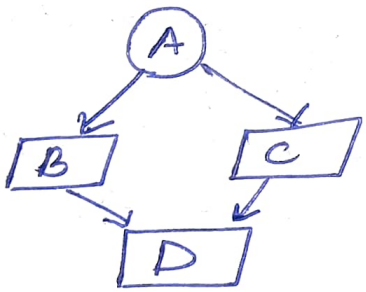
Friend :-

- access all private & protected members of class.
- non-member func.

```
class withfriend {
    int i;
  public:
    friend void fun();
};
```

```
void fun() {
    withfriend w;
    w = 10;
}
```

Diamond Problem (Multiple Inheritance)



B, C have 2 copies of member variable of A as both as inh. from A. Now D is inh. from B, C thus causing ambiguities.

Solution - Virtual Inheritance.

```
class Person {
  public:
    Person() {
        cout << "Person()";
    }
    Person (int x) {
        cout << "Person (int)";
    }
};
```

```
class Father : public Person {
                ↳ virtual
  public:
    Father() {
        cout << "Father()";
    }
    Father (int x) {
        cout << "Father (int)";
    }
};

class Mother : public Mother {
                ↳ virtual
  public:
    Mother() {
        cout << "Mother()";
    }
    Mother (int x) {
        cout << "Mother (int)"
    }
};

class child : public Father, public Mother {
  public:
    child () {
        cout << "Child ()";
    }
    child (int x) {
        cout << "Child (int)";
    }
};
Child child(30);
```

Output 1 -

Person()
Father()
Person()
Mother()
Child (int)

output 2 - (Virtual)

Person()
Father()
Mother()
child (int)

eg2

```
class base{
    public:
        int salary = 900;
};

class derived1: virtual public base{
    public:
        int base = 100;
};

class derived2: virtual public base{
    public:
        int inc = 400;
};

class deived3: public d1, public d2{
    public:
    void sum(){
        cout<< salary + bonus + inc;
    }
};

int main(){
    derived3 x;
    cout<< x.salary << x.bonus << x.inc;
    x.sum();
    return 0;
}
```

Output –

900 100 400

1400