Oops

< Object Oriented >
Programming

Motivation.txt

```
class Pillars{
     public:
          Encapsulation;
          Polymorphism;
          Inheritence;
          Absttaction;
};
```

#Constructor #destructor #thispointer #overloading

A to Z Handwritten Notes

Biswarup Acharjya

28/06/2024
8:47 P.M

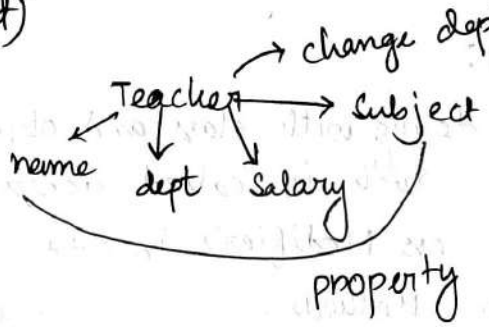
• Object Oriented Programming

Basically we talk about object. concepts related object. in c++.

OOP: The OOP is the way to write code in a better way. We convert real life scenarios in our code using OOPs.

- Object: Objects are entities in the real world
- Class: class is like a blueprint of these entities (A group of object)

entity many anything pen, laptop, bag, phone



Syntax:

Class

```
Class class_name {
    // property → attributes
    // Methods → member function
};
```

#include <bits/stdc++.h>

Example:

```
Class student {
public:
    string name;
    int age;
    int class;
    int Roll_no;
};
```

• This is how we create a student class, where name, age, class, roll_no are the property of the class, or attributes of the class

Object: We'll create object inside the class.

```
class_name object_name;
```

```
int main () {
    student stu1;
}
```

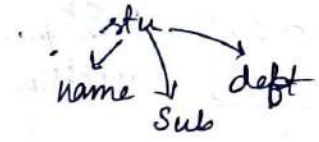
This is how we create a object (stu1) inside the main function

How to assign value to an objects

As our object has many property. Now how to assign value in it. Using 'dot' operator.

```
stu1.name = "Biswarup";
```

As we know student as so many property.




```
stu1 . name = "Biswa"
stu1 . sub = "OOP"
stu1 . dept = "CSE
stu1 . ROll = 20051
```

(We can treat as, a single variable;)

```
int main () {
  st1. name = "Biswa";
  cout << st1 . name;
}
```

- **Access Modifier:**

When we are deling with class and object. We should keep in mind one additional which is called access Modifier. It's a key words

There are 3 access Modifier's in c++

1) Public (data & Methods accessable to everyone)
2) Private (data & Methods accessable inside class)
3) Protected (data & Methods accessable inside class & to its derived class)

* by default in c++ all are private

- **Setter & Cretter:** Sometimes we try to get private data member outside the code. For that reason we can use getter and setter public funct by using get and set. We can access private data Member & memberfun-

```
Class teacher {
    Private:
        double salary;
    public:
        string name;
        int age;
        string dept;
```

Setter

```
    void setSalary (double s){
        salary = s;
    }
```

Cretter

```
    double getSalary (){
        return salary;
    }
};
```

```
int main () {
  teacher t1;
  t1. name = "Biswa";
  t1. setSalary (2500);
  t1. age = 21;

  Cout << "Name is " << t1. name;
  cout << "Salary is " << t1. getSalary ();
```

So basically private members are in retsricted mode. We can't access of their value. For that value if we try to access we have some public get and set function. It's a public function. first we have to set value then get the value.

Setter

```
Void setvalue ( int a) {
    age = a;
}
```

Getter

```
int getvalue () {
    return age;
}
```

Inside main function:

```
int main () {
    ~~struct setage~~
    st1.setvalue ( 21);
    cout<< st1.getvalue ();
    return 0;
}
```

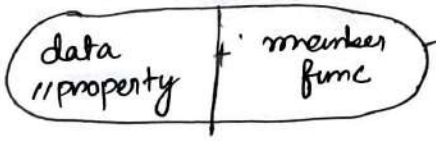
4 pilar of opps
1) Encapsulation
2) Abstraction
3) Inheritence
4) Polymorphism

■ Encapsulation:

Encapsulation is wrapping up of data & memberfunction in a single unit called class.

*data hidding

In the word encapsulation, we find a word called Capsul.



It's nothing but a class or define a class, it self a encapsulation

data hidding → hide sensitive data → private

Example: // create a bank system

```
Class Account {
    public :
        string id;
        string user_name;
    private :
        int balance;
        ~~int~~ string password;
};
```

■ In this way we write a class as an example of encapsulation in an single entity called class.

29/06/24
4:21 P.M

- **Constructor:** → default (Basic)

Special Method / function invoked automatically at time of object creation

- Same name of class
- No return type
- Only called once automatically, at object creation
- Memory allocation happens, when constructor is called.
- Constructor declear always in Public

```
Class Hero {
    public:
        int rank;
        string name;

        Hero(){
            cout << "Constructor called";
        }
};
int main (){
    cout << "Hi" << endl;
    Hero H1;
    cout << "Hello" << endl;
```

Output:

```
Hi
Constructor called
Hello
```

Types of Constructor:

i) Non Parameterized Constructor
ii) Parameterized Constructor
iii) Copy Constructor

- **Parameterized Constructor:**

In the time of object creation we pass value in the sigle line as paramter

```
Class Teacher {
        Public:
            string name;
            string subject;
            int Salary;
            string dept;

    Teacher (string n, string S, int Sal, string d){
            name = n;
            Subject = S;
            Salary = Sal;
            dept = d;
    }
};
```

```
int main(){
    Teacher t1("Biswa", "CSE", 2500, "OOP");
    cout<< t1.name;
    cout<< t1.Salary;
    return 0;
}
```

- If you make one constructor (Self), in this code the default constructor will be vanished:
- So when you write any constructor implementation, then no default constructor exist here
- In One class multiple constructor can exist. But with different no of parameter.

■ When multiple constructors are present in same code, but with different no of parameter. It is called Constructor overloading, which is an example of polymorphism.

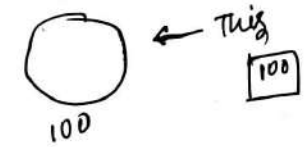
- This pointer: In c++, there is a special pointer called this pointer. that's points to the current object. It holds the address of current obj.

Example:

```
Teacher (string name, string dept, string sub, double salary){
    this → name = name;
    this → dept = dept;
    this → subject = subject;
    this → Salary = salary;
}
```

Left → object property
right → function para

How it works?



(*this).property

↓

Obj.property

int *ptr = &x;

Copy Constructor:

It helps to copy the object.

- Special constructor (default) used to copy properties of one object into another

Example:

```
int main(){
    Teacher t1("Biswa", "CSE", "C++", 25000);
    Teacher t2(t1); // default copy const invoked
    return 0;
}
```

// create a custom copy constructor:

```
class Teacher {
public:
    string name;
    string dept;
    string subject;
    int salary;
// para - constr
Teacher(string name, string dept, string subject, int salary){
    this->name = name;
    this->dept = dept;
    this->subject = subject;
    this->salary = salary;
}
// Copy constructor
Teacher(Teacher &orgObj){ → Pass by Refrence
    this->name = orgobj.name;
    this->dept = orgobj.dept;
    this->subject = orgobj.subject;
    this->salary = orgobj.salary;
}
};
int main(){
    Teacher t1("Biswa", "CSE", "OOPs", 25000);
    Teacher t2(t1);
    cout<< t2.name;
}
```

output
Biswa,

Shallow vs Deep Copy:

It's a part of copy constructure.

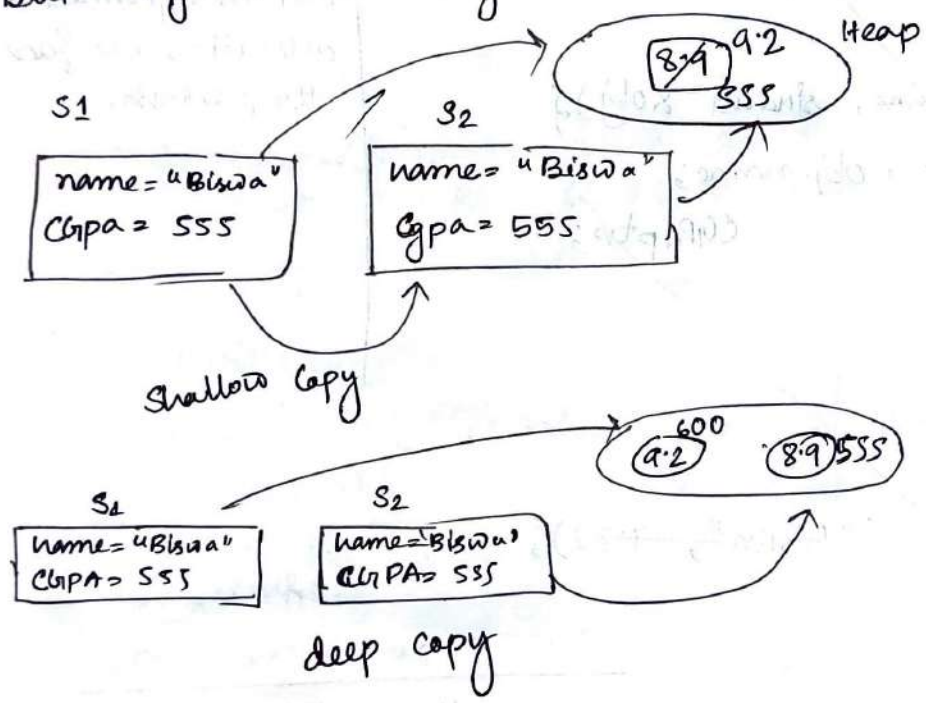
A Shallow copy of an object copies all of the member values from one object to another.

A deep copy, on the other hand, not only copies the member value but also make copies of any dynamically allocated memory that member points to.

It only faces problem in the time Dynamic Memory Allocation.

In our code, normal memory allocation happens in STACK

But Dynamic Memory allocation happens in HEAP

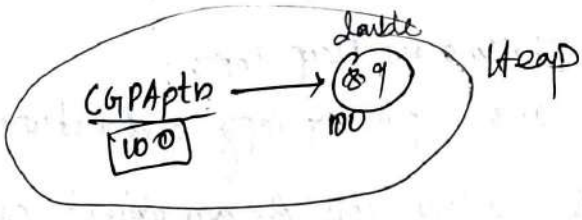


30/06/24
4:00 P.M

Example: Shallow copy:

```
Class student {
    Public:
        string name;
        double CGPA;
    student (string name, double CGPA){
        this → name = name;
        this → CGPA = CGPA;
    }
    student (student &obj){
        this → name = obj. name,
        this → CGPA = obj. CGPA;
    }
```

```
int main (){
    student S1;
    student S2(S1);
    student S1("Biswa", 9.2);
    student S2 (S1) ;
    cout << S2. name;
}
```



In normal static allocation we don't face the problem in copying value, its never reflect the other object.

But in dynamic memory allocation we face the problem.

Class

- In normal static memory allocation the original value never gets affect by the new value;

But

- In dynamic Memory allocation the original value gets affected by the new value;

Example: (Shallow copy)

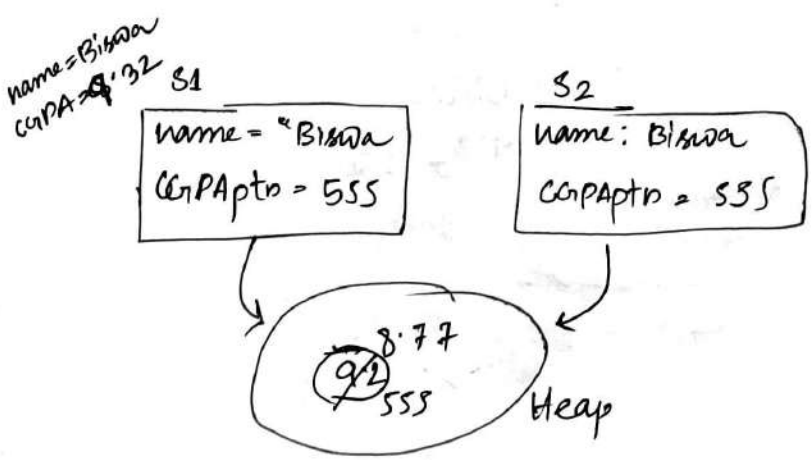
```
class Student {
    Public:
        string name;
        double * CGPAptr;
```

```
Student (string name, double CGPA) {
    this → name = name;
        CGPAptr = new double;
    * CGPAptr = CGPA;
}

// Copy constructor
Student (string name, student &obj) {
    this → name = obj. name;
    this → CGPAptr = obj. CGPAptr;
}

void getinfo() {
    cout << "name:" << name << endL;
    cout << "CGPA: " << * CGPAptr << endL;
}
};

int main () {
    Student S1 ("Biswa", 9.32);
    S1.getinfo();
    Student S2 (S1);
    S2.getinfo;
    *(S2.CGPA) = 8.77
    S2.getinfo();
    S1.getinfo()
```

Output
Biswa
9.32
Biswa
9.32
Biswa
8.77
Biswa
8.77

Here we change S2's CGPA. But it reflects S1's CGPA

This is Shallow copy



As both copies points to the same location. that's why any changes get reflected

This is shallow copy

Deep copy


```
Class Student {
    Public:
        string name;
        double * CGPAptr;

    Student (string name, double CGPA ){
        this → name = name;
        CGPAptr = new double;
        *CGPAptr = CGPA;
    }

    Student ( student & obj) {
        this → name = obj.name;
        CGPAptr = new double;
        *CGPAptr = *(obj.(GPAptr);
    }

    void getinfo () {
        cout<< "Name: "<< name<<endl;
        cout<< "CGPA: "<< CGPAptr << endL;
    }
};

int main () {
    Student S1 ("Biswa", 9.32);
    S1.getinfo()
    Student S2(S1);
    S2.name = "Aryan";
    *(S2.CGPAptr) = 8.32;
    S2.getinfo();
    S1.getinfo ();
```

create a new memory
] change here
give the same value

Output
Biswa
9.32

Aryan
8.32

Biswa
9.32

int *i = new int 01/7/24

Static and dynamic Allocation:

We can create our own object in 2 way. Statically and dynamically

Static Allocation

```
Class student {
    Public:
        string name;
        int roll;
};
int main() {
    student S1; // static allocation
}
```

dynamic allocation

```
Class student {
    Public:
        string name;
        int roll;
};
int main() {
    student *S1 = new Student;
    cout << (*S1).name;
        ↓
    cout << S1 → name;
}
```

Destructor:

Oppsite of Constructor.

- Constructor Allocate memory |: Destructor dellocate memory.

Destructor:

i) Has the same name of class
ii) No return type
iii) Dellocate the memory occuppied by object
iv) start with '~' Tilda sign.
v) Automattically called after objects work done
vi) when we create object statically, in that time destructor automatically called
vii) But when we create object dynamically, in that time we have to called destructor manually. Using "delect" word.
delect helps to make the memory free occupied by the obj;

Example:

```
class student {
  public:
        string name;
        int roll;

  student() {
      cout << "constructor called" << endl;
  }
  ~student() {
      cout << "destructor called" << endl;
  }
};
int main() {
    student S1; // static allocation
    student *S2 = new student; // dynamic allocation
    delect S2; //manually delete the memory as dynamic
}
```

02/07/2024
4:47 PM

Inheritance

Inheritance is one of the main pillar of oops.

Ex:
Parent → child . Our parent pass on our qualities to us..

When class A pass its property to same other class B.

Class A (Base class / Parent class)

↓

ClassB (inherit property from class A) (child class / Derived class

Inheritance: when property and member function of base class are passed on to the derived class.

why inheritance ?

a) code reusability

b)

Syntax:

```
class child_class : access modifier Base_class_name {
        // Body
};
```

Synax:

class child_class : access modifier name of Base class

Example:

```
class Person {          // Base class
    Public:
        string name;
        int age;
    person (string name, int age){
        this → name = name;
        this → age = age;
    }
};
class student : public person{
    Public:
        int roll;
    void get_info() {
        Cout<<"name:"<< name <<endl;
        Cout<<"Age:"<< age <<endl;
        Cout<<"Roll:"<<roll<<endl;
    }
};
```

Here we inherit the property of class Person in class student without writing extra code

```
int main(){
    Student S1;
    S1.name = "Biswa";
    S1.roll = 1234;
    S1.age = 21;

    S1.getinfo();

    return 0;
}
```

Output
Biswa : name
1234 : roll
21 : Age.

Order of Constructor Call ↓

Note: When inheritence happens one thing should in our mind

Constructor case: 1) Base class's constructor calls first 2) then child class's constructor calls. } Imp for Interview

But

Destructor case: 1) child class's destructor calls first. 2) then Base class's destructor calls.

Example: If we make our own constructor in inheritence process.

```
class Person{
    Public:
        string name;
        int age;
    Person(string name, int age){
        this->name = name;
        this->age = age;
    }
};
class Student : public Person{
    public:
        int roll;
    Student(string name, int age, int roll) : person (name, age){
            this -> roll = roll;
    }
}
int main(){
    Student S1("Biswarup", 21, 2005);

    Cout<<S1.name<<endL;
    Cout<<S1.roll<<endL;
    Cout<<S1.age<<endL;

    return 0;
}
```

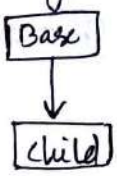
This is the concept of basic inheritance

- Mode of Inheritence:

Base class	Derived class Private Mode	Derived class Protected Mode	Derived class Public Mode
Private	Not inherited	Not inherited	Not inherited
Public	Private	Protected	Public
Protected	~~Protected~~ Private	Protected	Protected

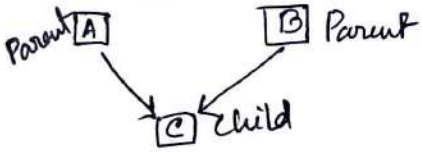
Types of Inheritance:

1) Single Inheritance: Previous example is a exp of Single Inheritance




```
class A{
  //
};
class B : public A{
  //
};
```

2) Multiple Inheritence:



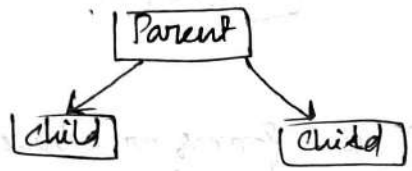

```
class Student{
    Public: string name;
            int roll;
};
class Teacher{
    Public: int salary;
            string subject;
};
class TA: public student, public Teacher{
    Public: int day;
};
int main(){
    TA T1;
    T1.name = "Biswa";
    T1.salary = 30000;
    T1.roll = 18;
    T1.subject = "OOP";
    T1.day = 2;
    cout<< T1.day <<endl;
    return 0;
}
```

3) Multi-level Inheritence

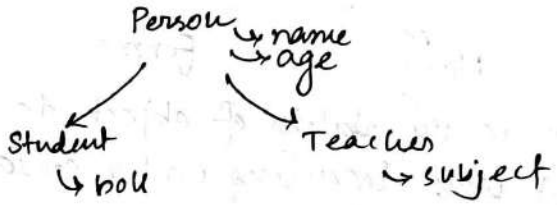
Example:

```
class myclass{
    string name;
    int age;
};
class mychild : public myclass{
    public:
        int roll;
};
class mygrndchild : public mychild{
    public:
        string research;
};
int main(){
    mygrndchild S1;
    S1.name = "Biswa";
    S1.roll = 200;
    S1.age = 21;
    S1.research = "Physics";
    cout<< S1.research <<endl;
}
```

4. Hierarchial Inheritance:



from one parent class multiple child classes inheritance

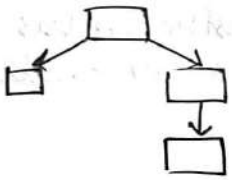


Example:

```
Class Person {
    Public:
        String name;
        int age;
};
Class Student : public Person {
    Public:
        int roll;
};
Class Teacher : public Person {
    Public:
        int salary;
        string sub;
};
```

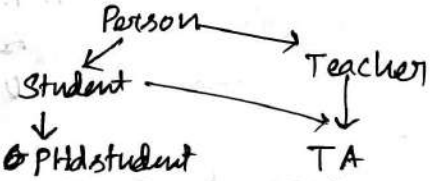
```
int main () {
    Student S1;
    S1. name = "Biswa";
    S1. age = 21;
    S1. roll = 123;
    Teacher T1;
    T1. name = "Aryan";
    T1. age = 32.
    T1. Salary = 48000;
    T1. Sub = "Math";
    return 0;
}
```

5. Hybrid Inheritence:



Mixture of all inheritence

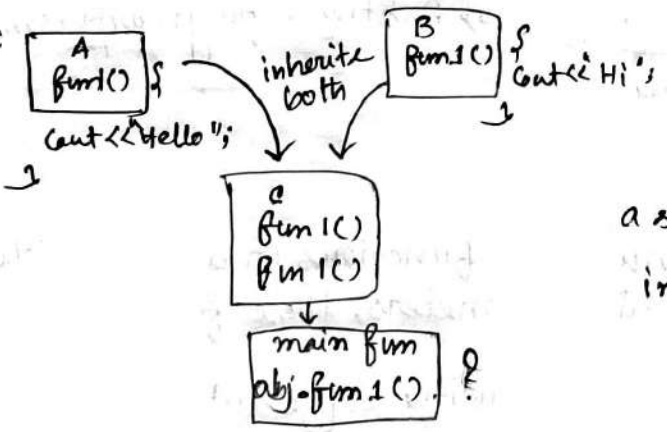
Ex:



■ Inheritence Ambiguity:

It is a worst situation when 2 base class has same name of data member and child class inheritance base class (multiple inheritence)

Example:



to solve this ambiguity we have a scope resolution operator :: in the func

```
int main () {
    c obj;
    obj. A :: fun1 ();   //obj.B::fun1();
    return 0;
}
```

Polymorphism:

04/7/2024
7:59 P.M

Poly + morphs
Many forms

Polymorphism is the ability of object to take on different forms or behave in different way depending on the context in which they are used.

Example: Best example is constructor overloading

```
class Student {
    public:
        string name;

    Student () {
        Cout<< " Non-Parametrized " ;
    }

    Student ( String name) {
        this → name = name;
        Cout<< " Parameterized ";
    }
}

int main () {
    Student S1;
    return 0;
}
```

We make 2 constructor here. and in main function we call.

Now we call the object but which function will call. As we have 2 Student() constructor

// As both constructor has same. this is call constructor overloading

Here we haven't passed any parameter so non-parameterized constructor will be call.

Types of Polymorphism:

1) Compile Time polymorphism
   - Function Overloading

2) Runtime polymorphism

- **Function Overloading:**

When there are multiple functions in a class with the same name but different parameters, these functions are overload.

Functions can be overloaded by using different types. of argument

- function with same name but different type of parameter or different type of parameter. in same class.

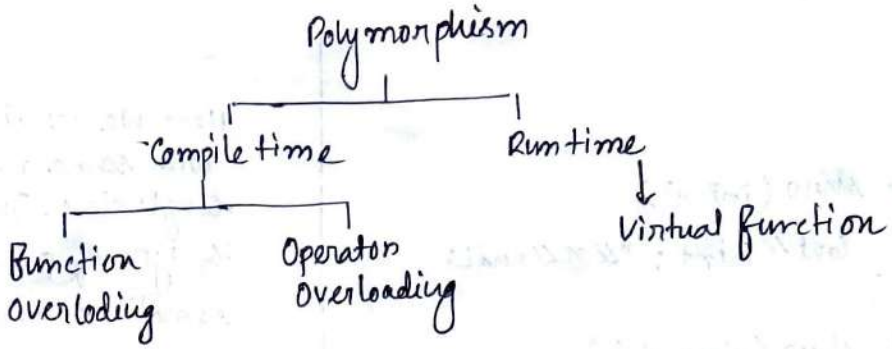
Example:

```
Class Print{
    public:
        void show (int x){
            cout<< "int : "<<x<<endl;
        }
        void show (char ch){
            cout<< "ch : "<<ch<<endl;
        }
};
int main(){
    print p1;
    p1.show(200);
}
```

Here we create 2 fun with same name in a single class. But the diff is, type of parameter/ sometimes no of paramter

Another Example of function overloading:

```
Class calculate {
    Public :
        void add(int a, int b){
            return cout<< "sum: "<<a+b<<endL;
        }
        void add (int a, int b, int c){
            Cout<<"Sum:" << a+b+c <<endL;
        }
        void add (double a, double b){
            Cout<<"Sum:" <<a+b<< endL;
        }
};
int main(){
    Calculate c;
    c.add (2,3);
    c. add (2,7,1);
    c add (2.1, 2.1);

    return 0;
}
```



* Function overloading: same name but diff no of argumnt / Parameter

Example:

```
class Area {
    Public:
        int calArea(int r) {
            return 3.14 * r * r;
        }
        int calArea ( int l, int b) {
            return l*b;
        }
};
int main() {
    Area Obj1, obj2;
    Obj1.calArea(4);
    obj2.CalArea(4,2);
}
```

In compliation time compiler know. which fu should be called

* Operator Overloading: when one operator perform different tasks then it's called operator overloading,

Simple example:

```
Int a=5;
int b=10;
cout<< a+b;  15
```

```
String a="Biswa"
String b=" Rup";
cout<< a+b;  BiswaRup
```

So '+' used to add two operands, but also used as for concatenation Simple.

Example: of a complex number:

The form of a complex number is = a+ib; Suppose 3+i2 and 4+i6 and add them

```
   3+i2  → imaginary no
4+i6
-------
7+i8
```

real no

```
class Complex {
        int real;
        int imag;
Public:
    Complex (int real, int imag) {
        this->real = real;
        this -> imag = imag;
    };
    void print () {
      cout << real << "+i" << image << endl;
    }
    Complex () {        // this is for complex ans.
    }
    Complex operator + (Complex & c) {
            Complex ans;
            ans.real = real + c.real;
            ans.imag = imag + c.imag;
            return ans;
    }
};
  int main () {
      Complex c1 (4,6);
      Complex c2 (7,1);
      Complex c3 = c1 + c2;
      c3. print ();
     return 0;
  }
```

- Runtime Polymorphism:

  which also know as dynamic polymorphism.

  Function Overriding / Method overloading is a way of runtime polymorphism. It's depending on Inheritence.

Methode overriding or func overriding is a feature that allows you to redefine the parent class method in the child class based on requirement.

Function Overriding

In the other hand, whatever method/func the parent class has by default are available in the child class. But sometimes a child class can change that method or modify the method based on the requirement. This process is called function Overriding. Only possible by inheritance.

Rules:
1) The parent class method and child class method must have same name. (Same name)
2) The parent class method/func and child class method/func must have same parameters. (Same parameters)
3) Possible only through inheritance

Example:

```
Class Parent {
public:
        void show () {
            cout<< "I am Parent class"<<endl;
        }
};
class child1: public Parent {
    public:
        void show () {
            cout<< "I am child class 1 "<<endl;
        }
};
class child2 : public Parent {
    public:
        void show () {
            cout<< "I am child class 2"<< endl;
        }
};
int main() {
    child1 cs;
    child2 cm;
    cs.show ();
    cm.show ();
}
```

Output:

```
I am child class 1
I am child class 2
```

Another example:

```
class Animal {
    public:
        void sound(){
            cout<< "Speaking" <<endl;
        }
};
class Dog : public Animal {
    public:
        void sound(){
            cout<< "Barking" <<endl;
        }
};
class Cat : public Animal {
    public:
        void sound(){
            cout<< "Meowing" <<endl;
        }
};
int main(){
    Cat Tom;
    Tom.sound();
    Dog Rocky;
    Rocky.sound();
    return 0;
}
```

output:
Meowing
Barking

10/7/24

Now we try to learn about function overriding with virtual function

- Virtual function

A virtual function is a member function that you expect to be redefined in derived classes. (during Runtime)

- Virtual function dynamic in nature
- Define ~~class~~ by the key-word "virtual" inside the base class and always declared with a base class and overridden in a child class.
- A virtual function is called during Runtime.

Simple example of a virtual function

```
Class Parent {
    public:
        void getInfo() {
            cout << "Parent class";
        }
        Virtual void hello() {
            cout << " hello from parent";
        }
};
Class child : public Parent {
    public:
    void getInfo() {
        cout << "child class";
    }
    void hello() {
        cout << " Hello from child";
    }
};
int main() {
    child c1;
    c1.hello();
}
```

output:
Hello from child.

But we'll see how it works during Runtime with a Better example.

Example:

```
Class Animal {
    public:
        void speak() {
            Cout << "Hu-Hu";
        }
};
Class Dog : public Animal {
    public:
        void speak() {
            cout << " Barking";
        }
};
int main() {
    Animal *P; // pointer P, points Animal value;
    P = new Dog (); // create a obj of Dog and store the address inside P;
    P -> speak();
}
```

Output:
Hu-Hu

Why? Hu-Hu?

In compilation time inside main fun();
P → Animal *P; / P pointer points Animal
P = new Dog() (skip)
P → speak().

■ In our Previous code why Hu-Hu is our output? * *

Ans: After writing the code inside the main function compiler see that

- there is a pointer (*P) which points Animal value. (1st line)
- In the 2nd line p = new Dog(); It is create an object of Dog and store the address inside p.

But when compiler see the "new" keyword he will skip the line, Because this allocation will be happen in "Run-time". As we used 'new' keyword. (Dynamic Memory Allocation in Heap)

- p → speak(). In third line p call speak function. Now P is Points to Animal type. So this line call void speak(){ cout<<"Hu-Hu"; }

In Compiler time he has already decided to print Hu-Hu.

This is the Reason.

Now the Question is how we'll print "Barking" as p store Dog.?

To achieve this we have to use "virtual" key-word.

Ex:

```
class Animal{
    public:
virtual void speak(){
        cout<<"Hu-Hu";
    }
};
class Dog : Public Animal {
    public:
        void speak(){
            cout<<"Bark";
        }
};
int main(){
    Animal *P;
    P= new Dog();
    P-> speak();
}
```

output:
Barking

How it works?

Inside main()
1) P pointer made, points Animal type value.
2) skip 2nd statement (As New) for dynamic Allocation
3) p→speak();

When speak() fun see "virtual" keyword. He will say whatever fun calling, it should be runtime Dont decide right now P calls 1st speak()/2nd speak(), and say p will call the function in which type of object 'p' store. And It will be decided in Runtime

As p store the address of the Dog that why virtual function print Barking.

why we are doing this? We create parent class pointer to get access the child class. if the function is override.

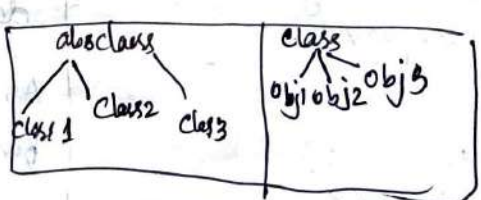
■ Pure virtual function / abstract class

```
class Animal {
    Public: virtual void speak() = 0;
};
```

• Pure virtual function:

```
Class Animal {
    public:
        virtual void speak()=0;   // abstract class  → for this class object never be created.
};
Class Dog: public Animal {
        public:
            void speak() {
                cout<< "Barking";
            }
};
Class Cat: public cat {
        public
            void speak() {
                Cout<< "Meowing";
            }
};
int main() {
    Animal * P;
    vector< Animal *> animals;
    animals.push_back(new Dog());
    animals.push_back(new cat());
    animals.push_back(new cat());
    animals.push_back(new Dog());
    animals.push_back(new cat());
    for(int i=0; i<animals.size();i++){
        p= animals[i];
        P→ speak();
    }
    return 0;
}
```

output:
Barking
Meowing
Meowing
Barking
Meowing



Abstraction

• Hiding all necessary details & showing important data. (Implementation hiding)

Access modifiers are the only way to implement Abstraction.

There is one more way to implement Abstraction using "Abstract class"

- Abstract class never create object.
- Abstract class is blueprint for derived class.
- Abstract class used to provide a Base class from which derived class can be inherited.

Example:

```
class Shape{
    public:
        virtual void draw()=0;
};
class Circle : Public Shape{
        public:
            void draw(){
                cout<<"Draw a Circle";
            }
};
class Retangle: publi Shape {
        public:
            void draw(){
                cout<<"Draw a Rectangle";
            }
};
int main(){
        circle c1;
        c1.draw();
}
```

12/07/2024
12:46 A.M

Static keyword:

Static keyword
- Static variable
  - in function
  - in class.
- Static object

- **Static variable:** variable declared as static in a function are create or initialized once for the lifetime of the program.

Example:

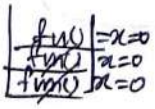
```
#include<bits/stdc++.h>
using namespace std;
void fun(){
    int x=0;
    cout<<"x:"<<x<<endl;
    x++;
}
int main(){
    fun();
    fun();
    fun();
}
```

output:
0
0
0

```
#include<bits/stdc++.h>
using namespace std;
Void fun(){
    static int x=0;
    cout<<"x:"<<x<<endl;
    x++;
}
int main(){
    func();
    fun();
    fun();
}
```

output
0
1
2

Reason:



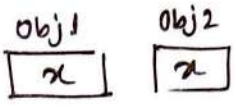
x is not in stack. It is in some other place in memory

- Static Variable (in class):

Static variables in a class are created and initialised once. They are shared by all the object of the class.

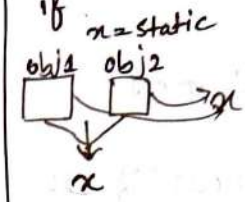
Example:

```
Class A {
    public:
        int x;
    void incre(){
        x=x+1;
    }
};
int main(){
    A obj1;
    A obj2;
    obj1.x=100;
    cout<<obj1.x;
    obj1.incre();
    cout<<obj1.x;
    obj2.x=200;
    cout<<obj2.x;
    obj2.incre();
    cout<<obj2.x;
}
```



output:

```
100
101
200
201
```




```
class A{
    public:
    static int x;
    void incre(){
        x=x+1;
    }
};
int main(){
    A obj1;
    A obj2;
    obj1.x=100;
    obj2.x=200;
    cout<<obj1.x;
    obj1.incre();
    cout<<obj1.x;
    cout<<obj2.x;
    obj2.incre();
    cout<<obj2.x;
}
```

output:

```
200
201
201
202
```

() Static keyword () (Static Data member)

Example: in a game there is Hero & Enemy. And there is a Time to Complete. But time to complete never depends on Hero and enemy. It's a independent.

Definition: It's a property. It's create a data member, which is belongs to class. No need to create object to access it.

Syntax: datatype classname :: (scope) name of the variable/data member name

```
int A :: Time to complete = 100;
```

- It belongs to class not object. So no need to create obj for it.

```
class Hero(){
    public:
        int static time to complete;
};
int Hero :: time to complete = 10;
int main(){
    cout<< time to complete;
    return 0;
}
```

Static function:

- ↳ No need to create of object / we can but no need
- ↳ Belongs to class
- ↳ This keyword it doesnot have
- ↳ Static function only access the static member function.*

```
class A {
  public:
    static int timeToComplete; // static datamember
      static int Random() {          //static function.
          return timeToComplete;
      }
};
int A :: timeToComplete = 2008 ;
int main() {
    Cout<<A:Random()<<endl;
    Cout << A :: Random() << endl; // static function only access static function
    Cout<< A :: timeToComplete <<endl; // we can access it this way also
```

Static object:

When we create an object static, it will exists lifetime in that program.

Example:

```
class A {
 public:
     A(){
       Cout<<"constructor"<<endl;
     }
   ~A(){
      Cout<<"Destructor"<< endl;
   }
};
int main() {
    if(true){
        A obj;
    }
    Cout<<"End of the Code";
 return 0;
}
```

Output
Constructor
Destructor
End of code

Destructor print last it means static object stays in the main fun() in the last moment

```
class ABC{
  public:
   ABC() {
   Cout<<"constru
  }
  ~ABC(){
    Cout<<" Destructo
  }
};
int main() {
   if(true){
   static ABC obj;
  }
Cout<<"End of code";
 return 0;
}
```

Output
constructor
End of code
Destructor

Exception Handling

20/7/2024
10:12 P.M

Definition: An exception is an unexpected problem that arises during the execution of a program and our program terminates suddenly with some error / issues.

real life example: worldcup 19 → draw → SuperOver → boundary Count
Preparation → Marketdown → again prepare

Try: It represents a block of code that may through an exception placed inside the try block:

Catch: It represents a block of code that is executed when a particular exception is thrown from try block.

Throw: An exception in c++ can be ~~through~~ thrown using the Throw keyword.

Example: We will use a bank example

```
class Customer {
    string name;
    int balance;
    int acc_number;
    public:                    // constructor
        Customer (string name, int balance, int acc_number) {
            this → name = name;
            this → balance = balance;
            this → acc_number = acc_number;
        }
    // deposit.
    void deposit (int amount) {
            if (amount > 0) {
                balance += amount;
                cout << "on balance << " is credited successfully";
                Cout << "Current balance" << balance;
            }
        else {
            Cout << "amount Should be greater then zero";
        }
}
```

```
void withdraw (int amount) {                    ← handle exception kara
    if(amount >0 && amount<=balance){
            balance -= amount;
        cout<< amount <<" is debited successfully"<<endl;
    }
    else if (amount<0){
            cout<< "Amount should greater then zero";
    }
    else{
        cout<< "Insufficient balance" <<endl;
    }
};
int main(){
    Customer c1 ("Aryan", 5000, 200134);
    c1. withdraw (300);
}
```

Here is a better way to handle exception. using throw, Try, Catch keyword.

```
//                  withdraw Section
void withdraw (int amount){
    if (amount>0 && balance >= amount){
        cout<<"amount withdraw/debited Successfuly";
          balance -= amount;
    }
    else if(amount<0){
        throw " amount should be greater than 0";
    }
    else{
        throw " Your balance is low";
    }
}
int main(){
        customer c1("Rohan", 5000, 210354);
    Try{
            c1. withdraw (6000);
        c1. Deposite (300);
    }
    Catch(const char *e){
        cout<<" Exception occured"<< e <<endl;
    }
}
```

Example: 2 To get better understanding

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

int main(){

    int a;   cin>>a>>b;
    int b;


try {

    if(b==0){
        throw "Divide by 0 is not possible";
    }
    int c = a/b;
    cout<<c<<endl;
}
catch(const char *e){
    cout<<"Exception Occure:"<<e<<endl;
}

}
```

The question comes in your mind, if something can handle by simple if/else, then why we use try, throw, catch?

When we write 10000 lines in our code it helps us to find bad allocation and runtime error very easyly.