Section 12 : Introduction to OOPS

15 April 2022 16:49

© Rajat Kumar
https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

Section 12 : Introduction to OOPS

Principles of Object Orientation :

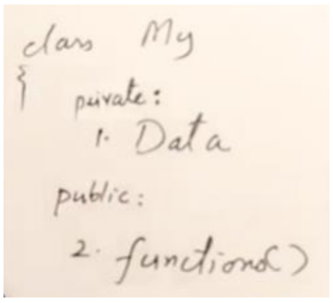
- Abstraction
- Encapsulation/Data Hiding
- Inheritance
- Polymorphism

Abstraction :

- When we don't know the internal details. That is nothing but abstraction.
- We just need names of functions, we don't see the implementation of functions as well as data (data is hidden).
- We only know functions details when we writing them not when we are using them.
- Example : Without knowing how the printf is working we have used it many times so that this abstraction for us.
- Class helps us achieving abstraction.

Encapsulation :

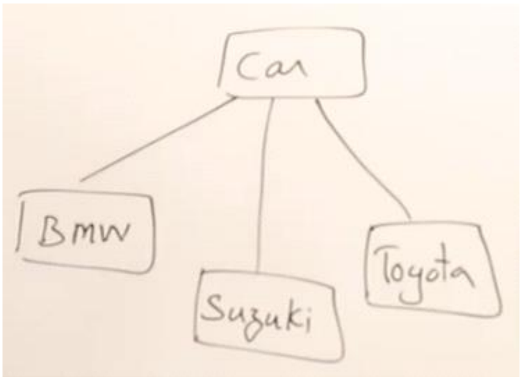
- We hide the data and make the functions visible and we put the things together at one place.
- Data and the functions together so that's it a class helps the data and the functions together. That is encapsulation and along with this, in classes we make data as private.
- This is not for security, It's for avoiding mishandling. The mishandling of the data so we make it as private and we make functions as public.
- We hide the data and show functions.
- Data hiding comes as a part of encapsulation.



Just like Abstraction and Encapsulation are interrelated, Inheritance and Polymorphism are also interrelated.

Inheritance :

- Suppose I want another class in which I want all these features plus extra features so I should be able to inherit, borrow all these features from existed class : This is inheritance.

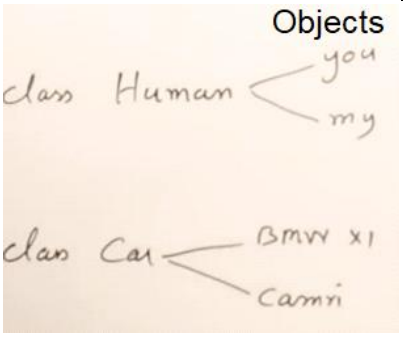


Polymorphism :

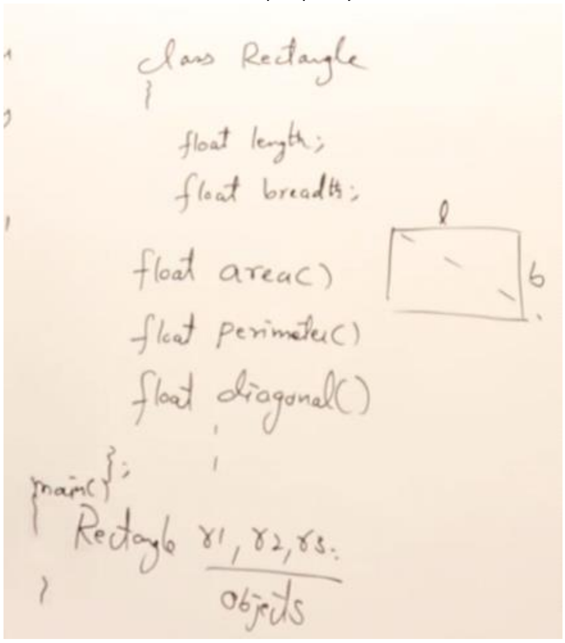
- For example : if we want to learn driving, we will learn to drive a car not bmw or suzuki or toyato. If we learn one we can drive all cars.
- The way it runs is different, the way it drives is different. That's polymorphism. So with the help of inheritance we achieve polymorphism.

Class :

- Class is basically a classification:
- Students, employees, cars, etc are all classification or a class.
- Classification is based on some criteria/property or the common things that we find in them.



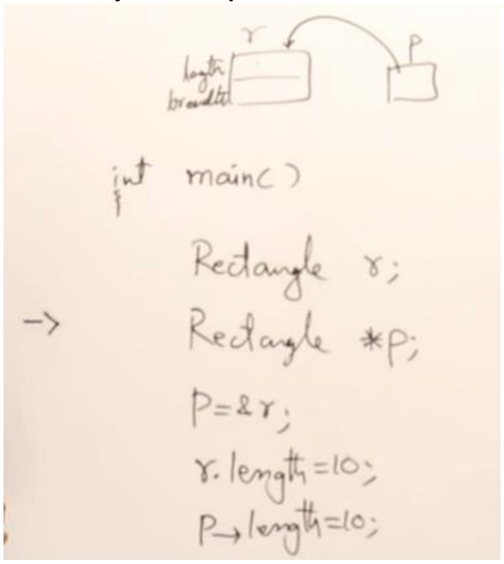
- Class is a definition/blueprint/design and object is an instance.
- Our class will contain data and functions.
    - So data is called as property and function is called as behavior.



```
class Rectangle
{
    float length;
    float breadth;

    float area()
    float perimeter()
    float diagonal()
};
main()
{
    Rectangle r1, r2, r3;
}
```

- Defining a class is different and creating its object and using it as different.
- Class are used for defining user defined data type so that we can declare the variables of that class type.
- Functions will not occupy any memory space, only data members occupy.
- Whatever we write inside the class by default it becomes private.
- Dot operators is used for accessing members of an object, we can access data members as well our member functions.

Pointer to an Object in Heap :

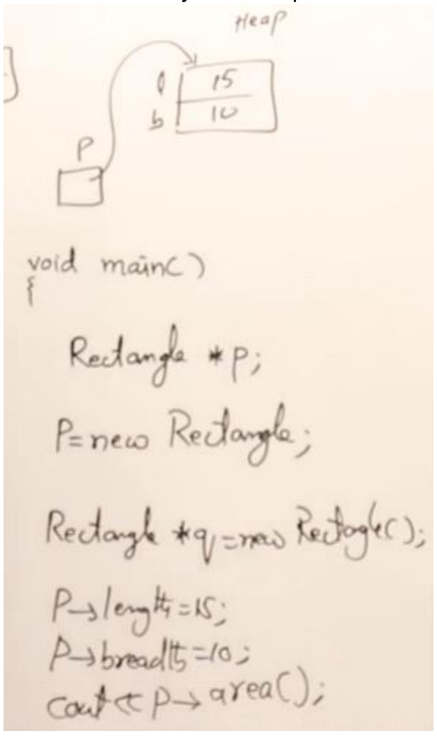


```
int main()
{
    Rectangle r;
    Rectangle *p;
    p=&r;
    r.length=10;
    p->length=10;
```

r
p
length
breadth
->

- WE are using pointer in above example but memory is still in stack not in heap.
- Dot operator . is used for accessing the members of an object using variable name. And arrow operate -> is used for accessing the members of an object using a pointer on an object

- This arrow -> is de-referencing operator instead of using star we can use this.
- Every pointer take either 2 or 4 bytes depending on the compiler.
- How to create object in Heap :

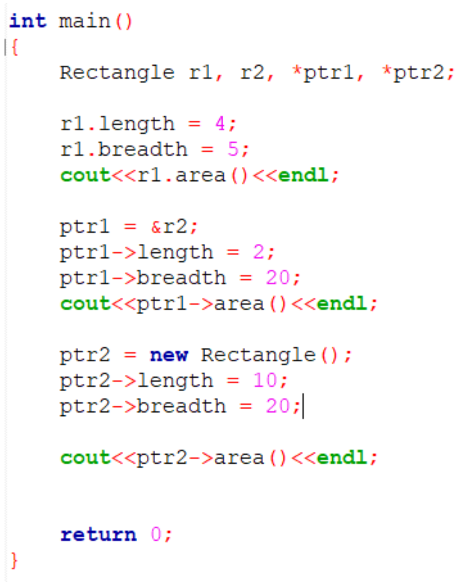

Heap
a
15
b
10
P

```
void main()
{
    Rectangle *P;
    P=new Rectangle;
    Rectangle *q=new Rectangle();
    P->length=15;
    P->breadth=10;
    cout<<P->area();
```

- There is no name to the object but pointer is pointing onto that one.

- Create object in Stack : Rectangle r;
- Create object in Heap : Rectangle *p = new Rectangle();

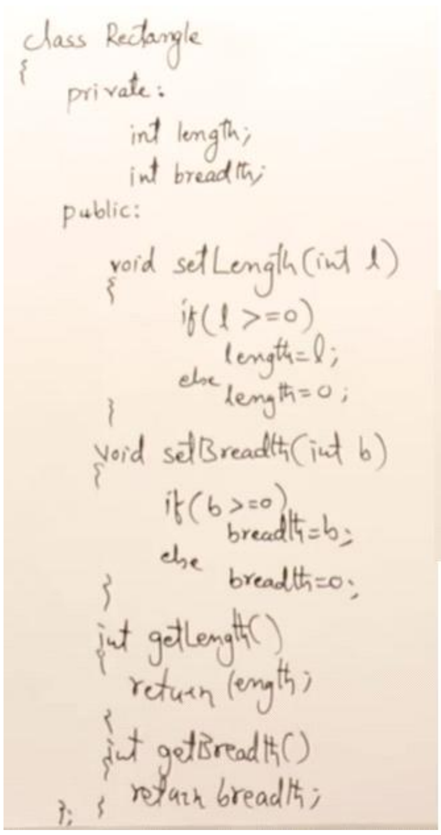
- In Java we cannot create an object in stack, always objects are created in a heap only using new but C++ gives us an option whether we want it in stack or whether we want it in heap.



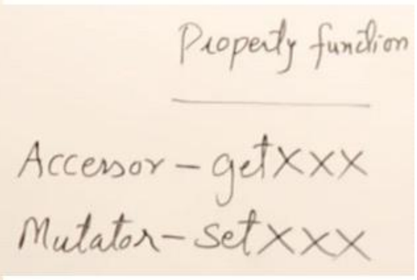
```
int main()
{
    Rectangle r1, r2, *ptr1, *ptr2;

    r1.length = 4;
    r1.breadth = 5;
    cout<<r1.area()<<endl;

    ptr1 = &r2;
    ptr1->length = 2;
    ptr1->breadth = 20;
    cout<<ptr1->area()<<endl;

    ptr2 = new Rectangle();
    ptr2->length = 10;
    ptr2->breadth = 20;

    cout<<ptr2->area()<<endl;


    return 0;
}
```

Data Hiding :

- Only the functions should be made as public, data members should be made as private.
- If that are made as public there are chances that they may be mishandled here. If mishandling is done the functions of a class may not give right results and we cannot rely on such classes.
- So we make data members as private and function as public.
- As now data members are private, we can't read or change their values. So we make functions to do so. Getters (Accessor) and Setters (Mutator).



```
class Rectangle
{
  private:
      int length;
      int breadth;
  public:
     void setLength(int l)
     {
         if(l >=0)
            length=l;
         else
            length=0;
     }
    void setBreadth(int b)
    {
         if(b >=0)
            breadth=b;
         else
            breadth=0;
    }
    int getLength()
    {
      return length;
    }
    int getBreadth()
    {
      return breadth;
    }
};
```



Property function

Accessor – getXXX
Mutator – setXXX

- If we make the data hidden then we face a problem, we cannot access the data, so we create get functions and set functions for reading and writing the data for each data member.
- We have written get and set functions. And those are called as property functions and get functions are accessor and set function are mutator.

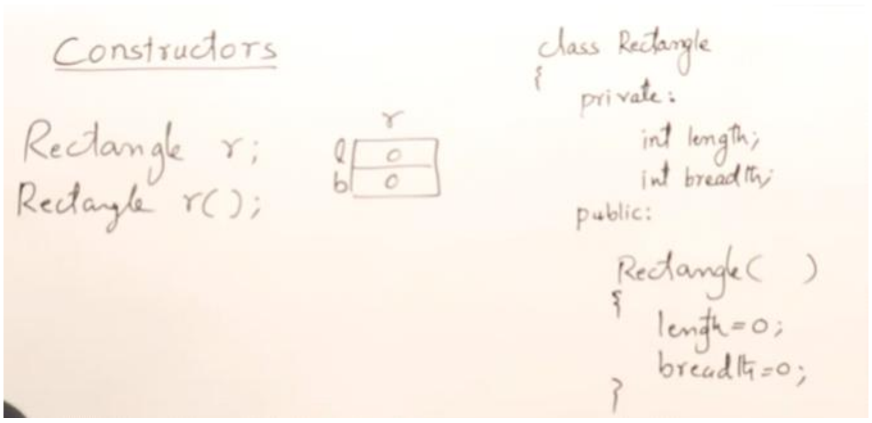
Constructor :
- It's philosophically wrong when we create and rectangle object, it's length and breadth were garbage/not define (we set letter). While creating or buying any rectangle object we also tell what should be its length and breadth.
- So we want the length and breadth to be set at the time of construction of that object. So we should have a function which should be automatically called when this object is constructed so that it take these parameters and assign these values.

Different types of Constructors :
- Default Constructor (sometimes called as build-in/compiler provide constructor)
- Non Parameterized Constructor (sometimes called as default also)
- Parameterized Constructor
- Copy Constructor

What is a Constructor :
- A constructor is a function which will have the same name as class name.
- The constructor will not have any return type but it will have same name as class name exactly same.

Default Constructor :
- Every class will have some constructor. So that compiler provided build-in constructor, called as default constructor.
- If we don't write any then a default constructor is called/provided by the compiler.
- It automatically called when we create an object.

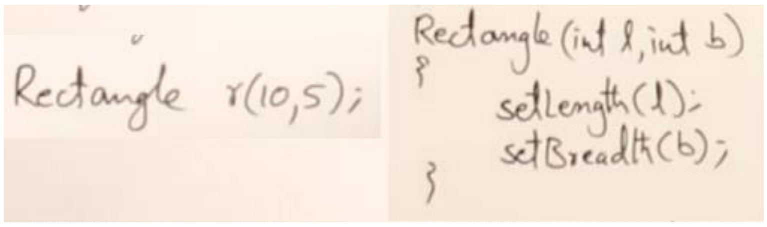
Non Parameterized Constructor : Default Constructor or User defined Constructor


Constructors

```
Rectangle r;
Rectangle r();
```

```
class Rectangle
{
  private:
     int length;
     int breadth;
  public:
    Rectangle( )
    {
      length=0;
      breadth=0;
    }
```

Parameterized Constructor :

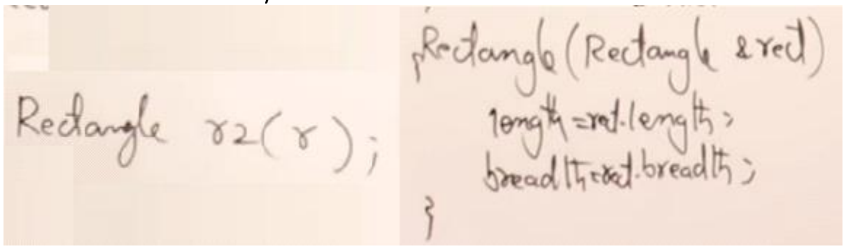


```
Rectangle r(10,5);
```

```
Rectangle(int l,int b)
{
    setLength(l);
    setBreadth(b);
}
```

Copy Constructor :

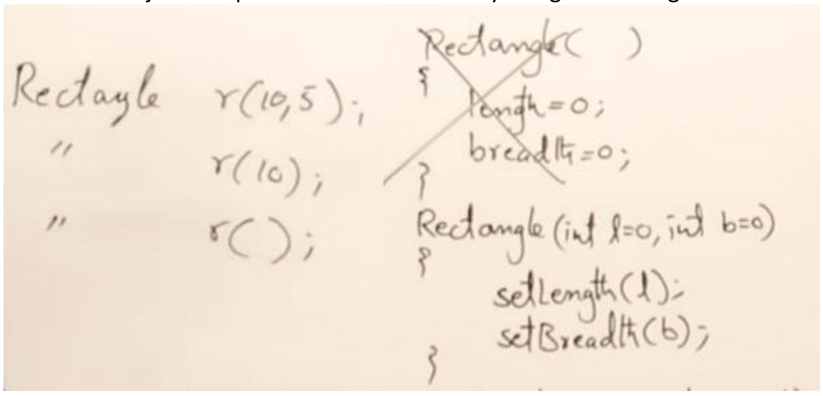
- Usually we take this as by reference and not by value. So that a new rectangle is not created again when we are calling a constructor. So that's why we take it as a reference.



```
Rectangle r2(r);
```

```
Rectangle(Rectangle &rect)
{
    length=rect.length;
    breadth=rect.breadth;
}
```

- The above all constructors are overloaded.
- We can write just one parameter constructor by using default arguments which will act as many different constructors.



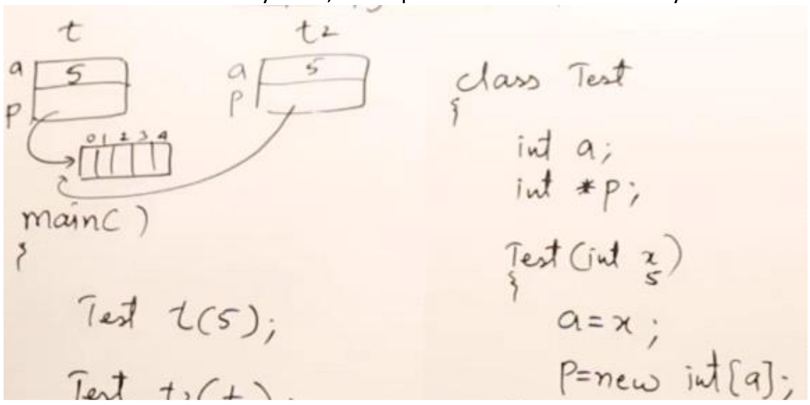
```
Rectangle r(10,5);
  "       r(10);
  "       r();
```

~~Rectangle( )~~
~~{~~
~~length=0;~~
~~breadth=0;~~
~~}~~

```
Rectangle(int l=0, int b=0)
{
    setLength(l);
    setBreadth(b);
}
```

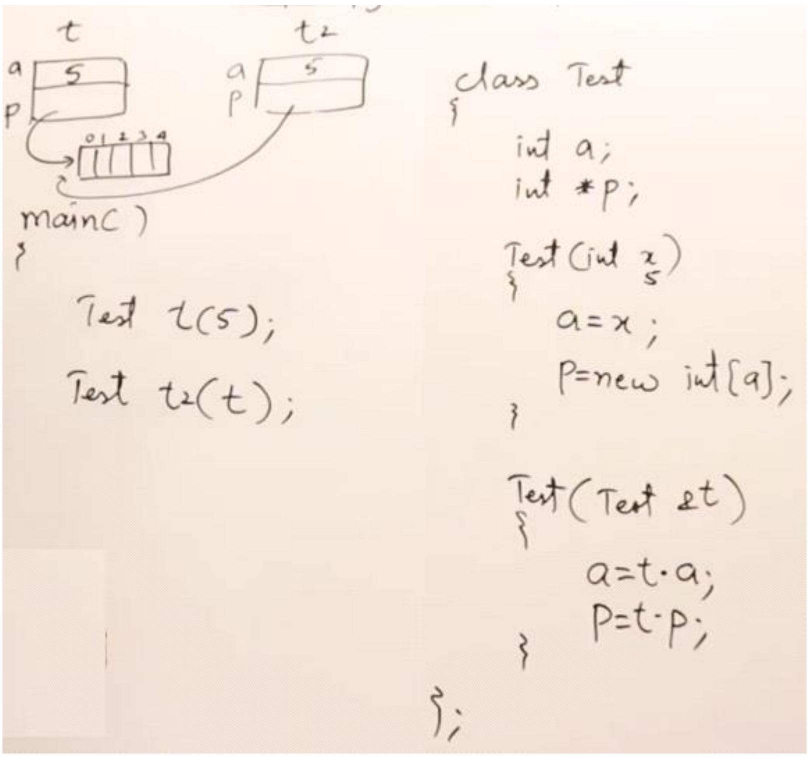
Problem with Copy Constructor :

- The problem with the copy constructor is if there is a direct memory allocation done by an object then the copy constructor may not create a new memory for it, it will point on the same memory. So we have to be careful with this type of thing.



```
main( )
{
    Test t(5);

    Test t2(t);
```

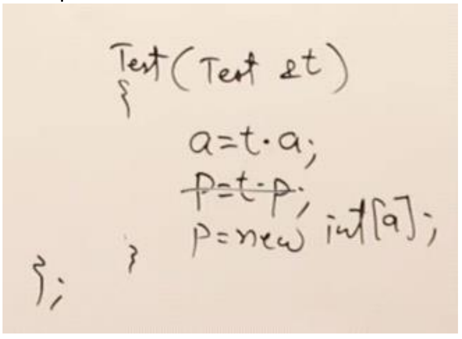
```
class Test
{
    int a;
    int *p;

    Test(int x)
    {           5
      a=x;
      P=new int[a];
```



```
class Test
{
    int a;
    int *p;
    Test(int x)
    {
        a=x;
        p=new int[a];
    }

    Test(Test &t)
    {
        a=t.a;
        p=t.p;
    }
};

main()
{
    Test t(5);

    Test t2(t);
```

Deep Copy Constructor :

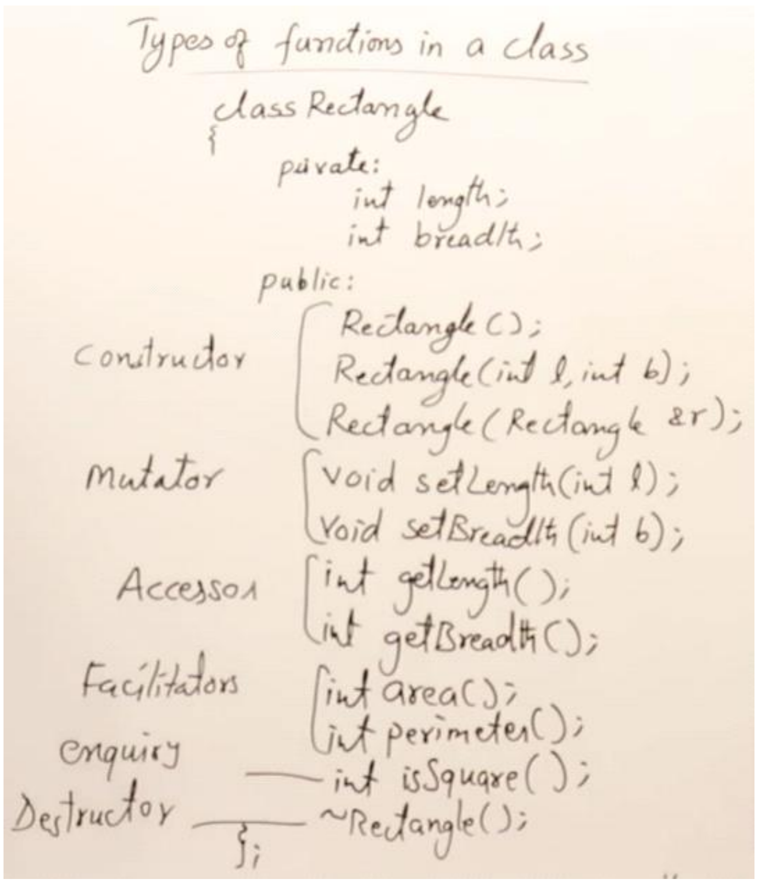
- We need to copy everything.
- Our copy constructor should create a memory and point it, instead of just pointing it to the memory of object that passed. For example :



```
Test(Test &t)
{
    a=t.a;
    ~~p=t.p;~~
    p=new int[a];
}
};
```

Types of Functions in a Class :

- Constructor
  - Non Parameterized
  - Parameterized
  - Copy
- Mutators (setLength and setBreadth)
- Accessors (getLength and getBreadth)
- Facilitators (area and parameter)
- Inspector Function (Enquiry - which will check whether it is a square or not. So this returns int value or else it is boolean).
- Destructor



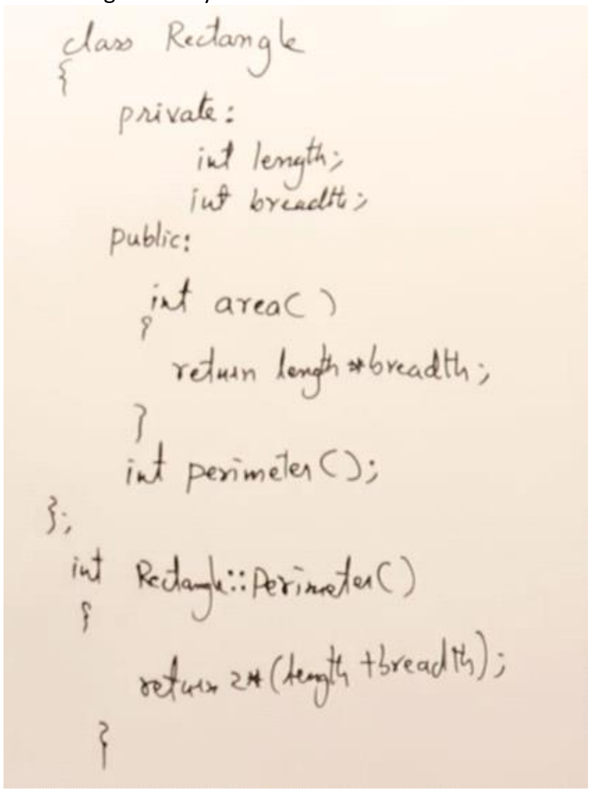
Types of functions in a class

```
                class Rectangle
                {
                    private:
                        int length;
                        int breadth;
                    public:
Constructor         [ Rectangle();
                    [ Rectangle(int l, int b);
                    [ Rectangle(Rectangle &r);
Mutator             [ void setLength(int l);
                    [ void setBreadth(int b);
Accessor            [ int getLength();
                    [ int getBreadth();
Facilitators        [ int area();
                    [ int perimeter();
enquiry        ----- int isSquare();
Destructor     ----- ~Rectangle();
                };
```

- The class is written like this only we don't write the functions inside. We don't elaborate them. We just write the header or the prototype type of the function and the functions are elaborated outside the class by using scoop resolution operator.

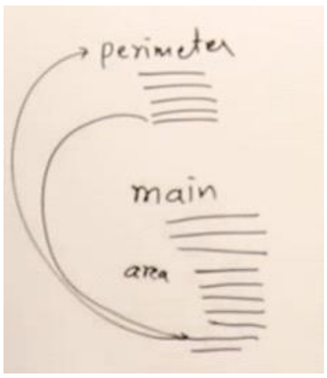
Scope Resolution Operator : (::)

- The scope resolution shows that the scope of this function is within this class that is already given. So this is same function, we are writing the body outside.



```
class Rectangle
{
    private:
        int length;
        int breadth;
    public:
        int area()
        {
            return length*breadth;
        }
        int perimeter();
};
int Rectangle::Perimeter()
{
    return 2*(length+breadth);
}
```

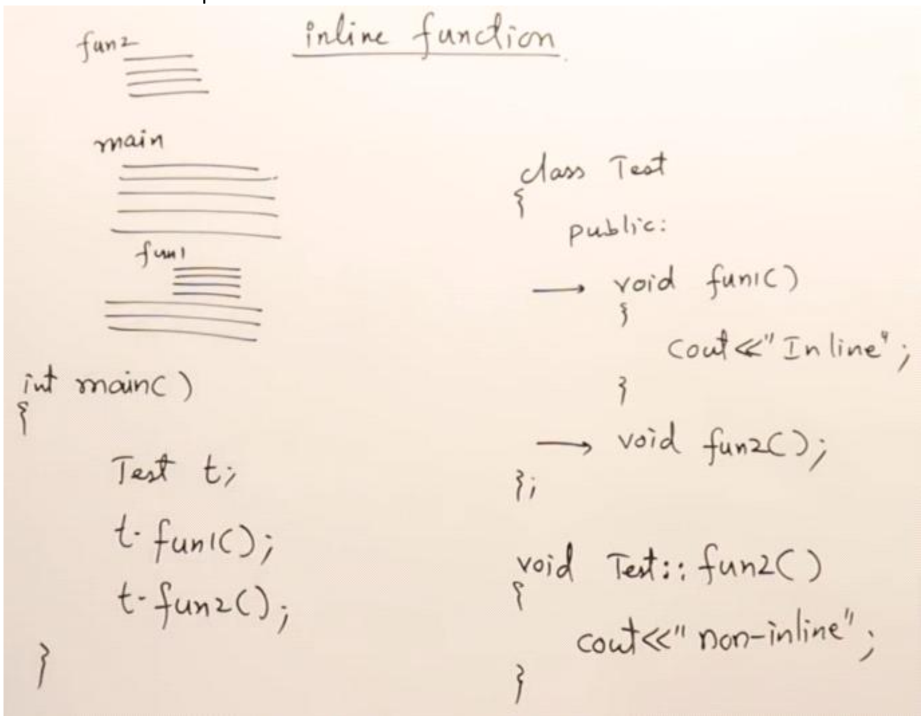
- If we write the function outside using scope resolution then the machine code for that function will be separately generated and when there is a call it will go to that function and after the function it will be return back to the main function.



- If we are writing the function inside the class itself then the machine code of that function will be replaced at the place of function column wherever the function is at that place only.
- If we write down the functions inside the class only, automatically such functions are called as **inline function**.
- So these functions automatically become inline their machine code will be replaced wherever the function is called.
- So in C++ it's a good practice to write the function outside using scope resolution.
- Otherwise if we would like them inside they will become inline function. So we should be careful when writing functions inside the class, inline functions should not have any complex logic.
- Writing inside the class is **inline**, writing outside is a separate function.

Inline Functions :

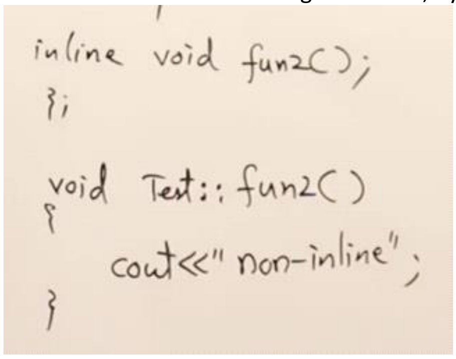
- The functions which expand in the same line where they are called.
- So there is more separate block for that function.



```
int main()
{
    Test t;
    t.fun1();
    t.fun2();
}

class Test
{
  public:
  → void fun1()
    {
       cout<<"Inline";
    }
  → void fun2();
};

void Test::fun2()
{
    cout<<"non-inline";
}
```

- Their machine code will be pasted or copied at the place or function.
- If we define a function inside a class automatically it is a **inline**.
- If we write a function outside the class then automatically **non inline** function.

- One more method for making it as inline, by adding inline before function declaration.

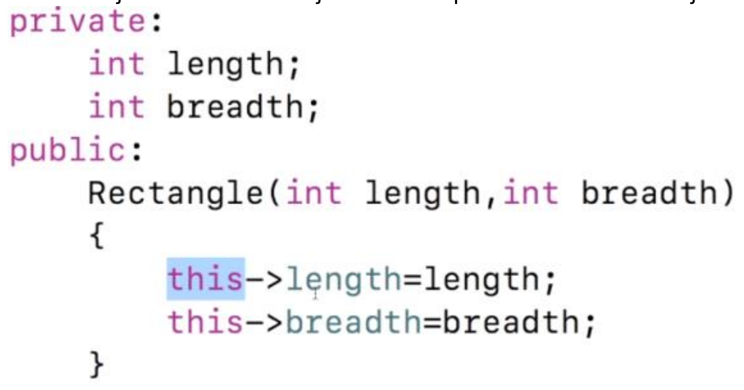


```
inline void fun2();
};

void Test::fun2()
{
    cout<<"non-inline";
}
```

- Now the code will not be separately generated but it is copied inside the main function only.

This Pointer :

- How do you refer to the data members of the same class or the same object?
- To avoid the name ambiguity and to make the statement more clear we can say this arrow (this->). So this is a pointer to the current object or the same object. This is a pointer to the same object.

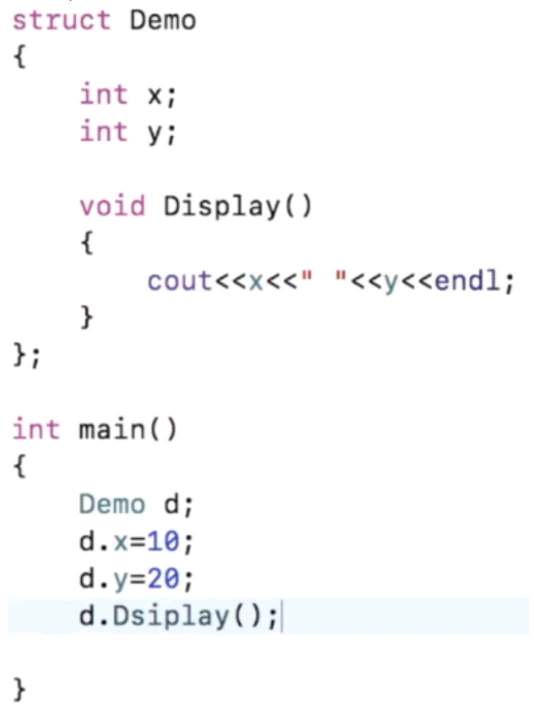


```
private:
    int length;
    int breadth;
public:
    Rectangle(int length,int breadth)
    {
        this->length=length;
        this->breadth=breadth;
    }
```

- So this is a pointer use for removing the ambiguity in between the parameters of a function with the data member of a class to refer the data members of a class of a current object.
- Using this pointer that you can refer to the members of a current object.

Structure :

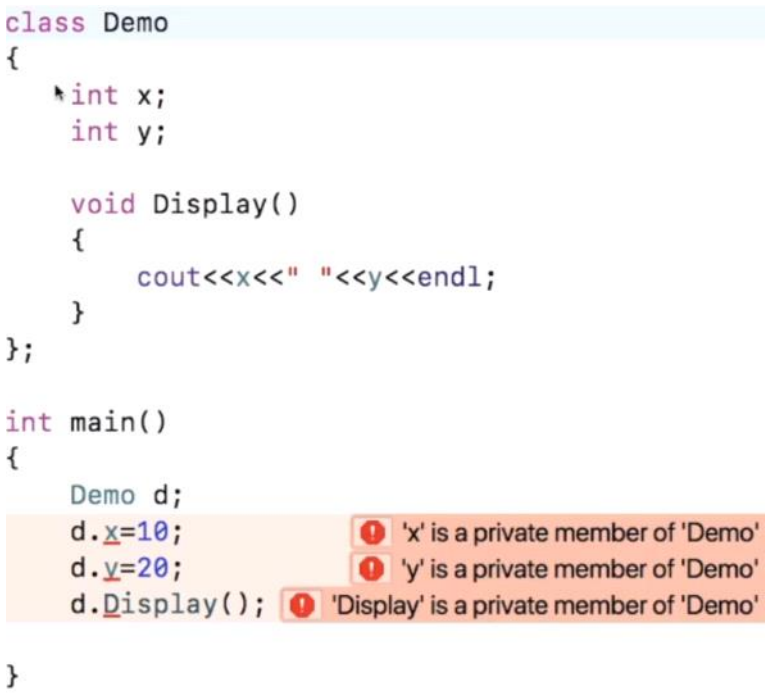
- In c language we can only have data members in structures, we cannot have functions inside a structure.
- In c++, structure can have data members as well functions inside the structure.
- In c++, structure is much similar to a class.



```
struct Demo
{
    int x;
    int y;

    void Display()
    {
        cout<<x<<" "<<y<<endl;
    }
};

int main()
{
    Demo d;
    d.x=10;
    d.y=20;
    d.Dsiplay();

}
```

Struct vs Class :

- In class, by default all data members and functions are private. To make anything public, we have to write public.
- In structure, by default all data members and functions are public, they are accessible from outside.



```
class Demo
{
    int x;
    int y;

    void Display()
    {
        cout<<x<<" "<<y<<endl;
    }
};

int main()
{
    Demo d;
    d.x=10;             'x' is a private member of 'Demo'
    d.y=20;             'y' is a private member of 'Demo'
    d.Display();    'Display' is a private member of 'Demo'

}
```

- Object will consume memory once for data members. Functions will be common for all the objects in memory.