* Constructor :

A constructor is a special member function that is automatically called when an object of a class is created.

→ The purpose of a constructor is to initialize the data members of the object to specific values

→ Every class will have a default constructor provided by the compiler.

→ A constructor has the same name as the class name.

→ It does not have any return type.

→ It should be public.

→ It can be declared as private also in some cases.

→ Constructor is called when object is created.

→ Constructor can be overloaded and it can take default arguments.

* Types of Constructor -

i) Non-parameterised Constructor (Default Constructor)

ii) Parameterised Constructor

iii) Copy Constructor (Special member function of a class that creates a new object as a copy of an existing obj)

Eg:-

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

class MyClass
{
  Public:
     int x;

     MyClass()    // Default Constructor
     {
        x = 0;
        cout << "Default Constructor" << x << endl;
     }

     MyClass(int value)   // Parameterised Constructor
     {
        x = value;
        cout << "Parameterized Constructor" << x << endl;
     }

     MyClass(MyClass &m)   // Copy Constructor
     {
        x = m.x;
        cout << "Copy Constructor" << x << endl;
     }
};

int main()
{
    MyClass obj;   (It will call default constructor)
    MyClass obj1(5);  (It will call parameterised constr.)
    MyClass obj2(obj1);  (It will call copy constructor)
    MyClass obj3 = obj1;  (copy constructor)
}
```

O/P- Default constructor 0
Parameterised Constructor 5
Copy constructor 5
Copy Constructor 5

Note- Here all constructor's names are same (MyClass), so these are overloaded and it is known as constructor overloading.

* There are two types of Copy Constructor -
i) Deep Copy Constructor
ii) Shallow Copy Constructor

i) Deep Copy Constructor-

A deep copy constructor creates a new obj with a seprate memory address for its newly member variables.

→ This means that changes made to the member variable of the existing object will not affect the newly created object.

→ Deep copy constructor is useful for making a copy of dynamic structures which are in heap, like dynamic array, linked list, trees etc.

ii) Shallow Copy Constructor-

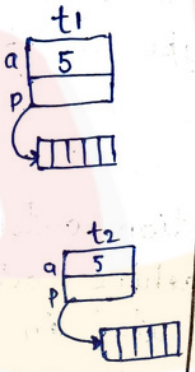
A shallow copy constructor simply copies the memory address of the existing object's member variable into the newly created object's member.

→ This means the newly created object and the existing object share the same memory address for their member variables.

→ If any changes are made to the existing object's member, those changes will also be reflected in the newly created object.

Eg:- Deep Copy

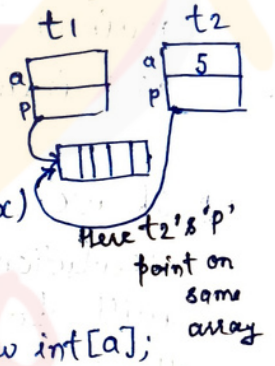
```
class Test
{
  int a;
  int *p;
  Test(int x)
  {
    a = x;
    p = new int[a];
  }
  Test (Test &t)
  {
    a = t.a;
    p = *new int[a];
  }
int main()
{
  Test t1(5);
  Test t2(t1);
}
```



Make Deep copy

Shallow Copy

```
class Test
{
  int a;
  int *p;
  Test(int x)
  {
    a = x;
    p = new int[a];
  }
  Test (Test &t)
  {
    a = t.a;
    p = t.p;
  }
int main()
{
  Test t1(5);
  Test t2(t1);
}
```



Make Shallow copy

* There are two methods of writing a function inside a class -

i) Write the prototype as well as elabrate the function inside the class.

ii) Prototype is written inside the class and its ~~elabrati~~ elaboration outside the class along with scope resolution (::) operator.

→ If we write the function inside the class itself then the machine code of that function will be replaced at the place of function call. Means that this function has not a seprate part, it is part main() only. This type of function are called as inline function.

→ If we write the function outside using scope resolution then the machine code for that function will be seprately generated in stack and when there is call, it will go to that function code and after that it will return back to the main function.

* We should write simple or ~~sign~~ single line function (get/set) as inline function but with loops should be avoided.

* Inline function will save time. Fuction call will not be made and activation record will not be created.

Eg:-

```
class Rectangle
{
  Private:
      int l;
      int b;
  Public:
      int area()   // Inline Function
      {
        return l*b;
      }
      int Perimeter();
};
int Rectangle :: Perimeter()   // Using scope resolution
{
   return  2*(l+b);
}
void main()
{
   Rectangle r(10,5);
    r.l = 10;
    r.b = 5;
    cout << r.area();
    cout << r.perimeter();
}
```

* All types of member function that are written in a class for making best class ever.-

i) Constructor
ii) Accessor
iii) Mutators
iv) Facilitator [Actual function of class (area(), perimeter())]
v) Enquiry [Used for checking if an obj satisfies some cond]
vi) Destructor [Used for releasing resources used by obj]

Write all functions for a rectangle -

```
#include <iostream>
using namespace std;
class Rectangle
{
  Private:
      int length;
      int breadth;
  Public:
      Rectangle();
      Rectangle(int l, int b);
      Rectangle(Rectangle &r);
      int getLength(){Return length;}
      int getBreadth(){Return breadth;}
      void setLength(int l);
      void setBreadth(int b);
       int area();
       int Perimeter();
       bool isSquare();
       ~Rectangle();
};
int main()
{
   Rectangle r1(10,10);
   cout<<"Area"<<r1.area()<<endl;
    if(r1.isSquare())
        cout<<"Yes"<<endl;
}
```

```
Rectangle :: Rectangle()
{
    length = 1;
    breadth = 1;
}

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

Rectangle :: Rectangle(Rectangle &r)
{
    length = r.length;
    breadth = r.breadth;
}

void Rectangle :: setLength(int l) { length = l; }

void Rectangle :: setBreadth(int b) { breadth = b; }

int Rectangle :: area() { return length * breadth; }

int Rectangle :: perimeter() { return 2*(length + breadth); }

bool Rectangle :: isSquare()
{
    return length == breadth;
}

Rectangle :: ~Rectangle()
{
    cout << "Rectangle Destroyed";
}
```

* 'this' Pointer -

If there is any name conflict b/w the parameters and property (member of the class) then to refer the properties or the data members of the class, we use 'this' keyword.

→ It is useful in avoiding variable name conflict.

```
Eg:- class Rectangle
     {
       Private :
           int length;
           int breadth;
       Public :
           Rectangle (int length, int breadth)
           {
               this → length = length;
               this → breadth = breadth;
           }
           void ~~int~~ display ()
           {
               cout << "Length = " << this → length << endl;
               cout << "Breadth = " << this → breadth << endl;
           }
     };
     int main()
     {
        Rectangle r1 (10, 5);
        r1.display()
     }
```

Operator Overloading -

If int a = 5, b = 5;

int c = a + b;

cout << c;

(Here Compiler doesn't show any error)

But if we define our own data type (User define data types) then these operators (+, -, * ...) can not be used means it doesn't work on custome data types.

Eg:- let assume obj1, obj2, obj3 are objects of any class -

And if we add two objects and stores in other object then here compiler will show error.

obj3 = obj + obj2

→ The compiler shows error because the compiler doesn't know how to add objects of an class using '+' operator.

→ It knows only how to add numbers of pre-define data types (int, float) using '+' operator.

→ To add two objects, we have to overload '+' operator.

→ And here 'operator overloading' comes in light -

"Operator overloading allow us to redefine the behaviour of an operator when used with custom data types. (User define). This means that we can create our own class and objects, and define how operators (+, - *, /) should behave when used with those objects."

* There are various operators that can be overloaded in c++, except few of them (sizeof, ::, .).

* Operators can be overloaded using member function and friend function.

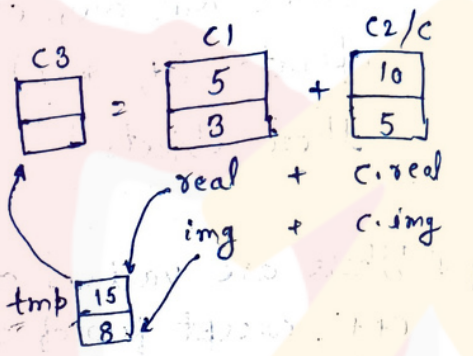
* Operator overloading using member function -

```
Class complex
{
  Public:
     int real;
     int img;

     complex(int r=0, int i=0)        // constructor with default parameters
     {
        real = r;
        img = i;
     }

     void display()
     {
        cout << real << "+i" << img << endl;
     }

     complex operator+(complex c)     // operator overloading of '+'
     {
        complex temp;
        temp.real = real + c.real;
        temp.img = img + c.img;
        return temp;
     }
};
int main()
{
   complex c1(5,3), c2(10,5), c3;
      c3 = c1.operator+(c2);
   or, c3 = c1+c2;
   c3.display();
}
```



output → 15+i8

* Friend function -

→ If we are using two or more objects in the parameter then we have to make use of friend function.

When both are of same class then we have two options -

i) make operator as member function

ii) Make it as friend function.

→ When two arguments are from different classes then there is only one option that is friend function.

* Operator overloading using friend function -

```
class complex
{
  Private:
     int real;
     int img;
  Public:
    complex (int r = 0, int i = 0);
     {
         real = r;
         img = i;
     }

    void display()
    {
       cout << real << "+i" << img << endl;
    }
  friend complex operator+ (complex c1, complex c2);
};
```

```
complex operator+ (complex c1, complex c2)
{
    complex temp;
    temp.real = c1.real + c2.real;
    temp.img = c1.img + c2.img;
    return temp;
}

int main()
{
    complex c1(5,3);
    complex c2 (10,5), c3;

    c3 = c1+c2;
or  c3 = operator+(c1,c2);
    c3.display();
}
```

Here 'operator+' function is called and two parameters are passed c1 and c2

* Insertion operator overloading (<<) -

Prototype used in insertion ope^r overloading - (change it a/c to your class)

```
ostream & operator << (ostream &os, complex & c)
```

- ostream & : return type; It must be declared as friend insid class
- operator << : function name
- (ostream &os, complex & c) : parameters (as reference); It must take two parameters.

* Eg:-

```
class complex
{
  private:
     int real;
     int img;
  public:
     complex (int r = 0, int i = 0)
      {
        real = r;
        img = i;
      }
   friend ostream &operator << (ostream & os, complex &c);
};
ostream &operator << (ostream & os, complex &c)
 {
    os << c.real << "+i" << c.img << endl;
    return out;
 }
int main()
{
    complex  c(10, 5);
     cout << c << endl;
  Or, operator << (cout, c);
```

Here "operator <<" fun. is called and two parameters are passed as cout (ostrea) and c.