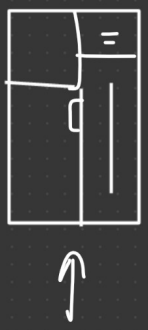
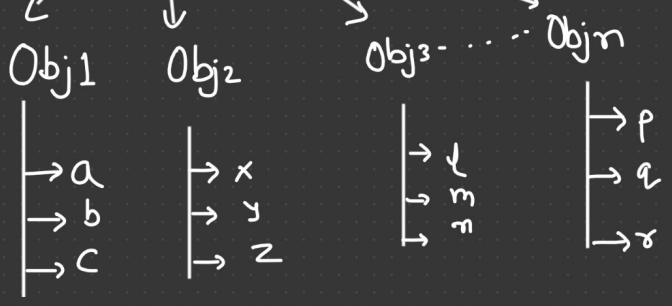
Objects and classes



9555031137

Real entity

Blue print of Object



Class

Data member → set of attributes (height, weight, length, breadth, colour...)

+

member function → set of operation (addition(), running(), eating().....)

```
class Student
{

   Char name[20];
   int age;
   int rollno;

};
```

By default Access Modifier is private

```
struct Student
{
   char name[20];
   int age ;
   int rollno;
};
```

Structure of C.

User defined Data Type

```
struct Student s;
```

s.age
s.rollno

```
int x;
```

```
class Addition
{
 public :
   int x;
   int y;
   int add()
   {
     return x+y;
   }
};
```

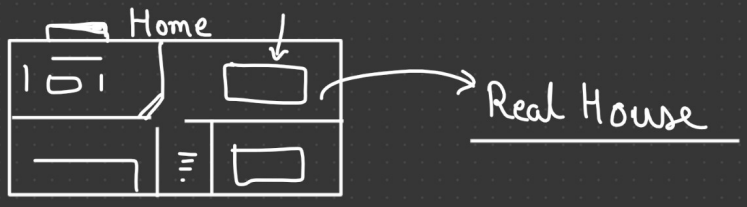
Data member + Member function ⇓ Class

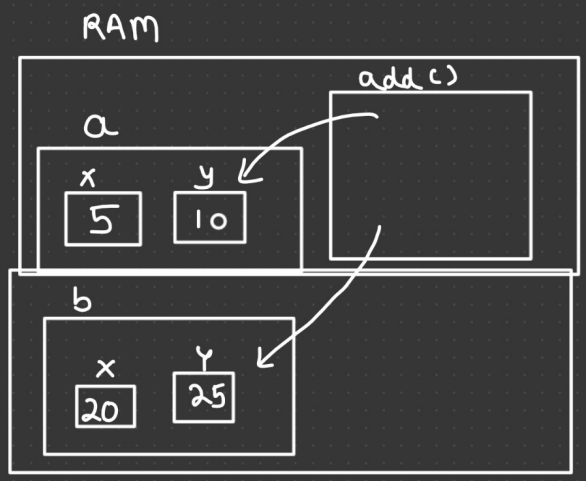
C Lang:-

```
int add(int x, int y)
 {
    return x+y;
 }
int main()
{
  int z = add(2, 3);
  printf("%d", z);
}
```

& No memory has been allocated at the time of Class Creation.

```
int main()
{
  Addition a, b;
  a.x = 5;
  a.y = 10;
  b.x = 20;
  b.y = 25;
  int z = a.add();
  int p = b.add();
  cout<<"Addition is"<<z<<endl;
  Cout<<"Addition is"<<p;
  return 0;
}
```

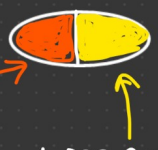




printf("Addition is %d", z);

OOPs features :-

- ✓ ✱ Abstraction ⟶ Data hiding
- ✓ ✱ Encapsulation ⟶ class = Data member + member function



- ✓ ✱ Polymorphism ⟶ Poly = many (EK naam, Anek Kaam)

<< , >>

- ✓ ✱ Inheritance
  - ✱ Reusability of code ⟶ with the help of inheritance.

```
Class  Addition
{
  public :
        int x,y;
        int add();
        int sub();
};
```

Class = 10 x 50 = 500 line.

= 20 x 100 = 2000 line

↓

20 - 30 line

```
int  Addition :: add()
{
  return x+y;
}

int Addition::Sub()
{
  return x-y;
}
```

```
Class A
{
    int add();

};
int A::add()
{
}
```

Constructors

```
int x;      → declare (Garbage value)
x = 5;      → Assign
```

```
int x = 5;  → Initialization
```

```
Addition a;  → declare
a.x = 5;
a.y = 10;    } → Assign
```

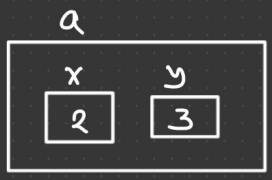
- We can initialize our object using Constructors in C++.
- It a special function whose name is same as the name of our class.
- It does not have any return type.

```
Class Addition
{
public :
        int x, y;        // Instance Variable

        Addition (int a, int b)
        {
            x = a;
            y = b;
        }

        int add()
        {
          return x+y;
        }
};
```

```
int main()
{
   Addition a = Addition(2,3);
   int z = a.add();
   Cout<<z;
   return 0;
}
```



- Access Modifier of the constructor must be public.
- If we want that no one can create object of my class then I need to make my constructor private.
- we can make more than one constructors in our class.
- If we do not make any constructor in our class then Compiler itself creates a default constructor for our class.
- If you create any constructor then compiler will not create any default constructor. The responsibility of creating default constructor is passing to you now.

we can call the constructor implicitly & explicitly:-

① Addition a = Addition(2,3); ⟶ Explicit call

② Addition a(2,3); ⟶ Implicit call.

③ Addition a = 2; ✓

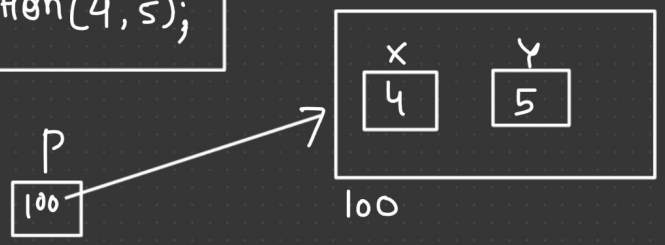
Addition a = (2,3); then only one value will get copied using one parameter constructor, which is 3.

Addition a = 2,3; // Error

DMA using Constructor :-

"new" Keyword is used to create dynamic memory in c++.

```
Addition *P = new Addition(4,5);
```




```
P -> x = 5;
P -> y = 10;
P -> add();
```

```
(*P).x = 5;
(*P).y = 10;
(*P).add();
```

Default argument Constructor:-

```
Addition (int a, int b=0)
{
    x = a;
    y = b;
}
```

⟶ So it will work for one parameter & two parameter both.

Note :- Default value will be assigned from right to left only.

Default argument Method :-

```
int add(int x, int y, int z=0)
{
    return x+y+z;
}
```

```
add(2,3); // output = 5
```

```
add(2,3,4); // output = 9
```

Getter & Setter :-

```
class Addition
{
    int x;
    int y;
public:
    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};
```

By default it is private we can't access private member Outside the class.

→ getter function for x

Read Access Only

a.x → error

a.y → error

a.getX(); ✓

a.getY(); ✓

Cout << a.x; → Output

Cin >> a.x; → input.

```
Class Addition
{
    int x, y;
public:
    void setX(int a)
    {
        x = a;
    }
};
```

→ setter

Addition a;

a.x → error

a.y → error

a.setX(3);

↓

write access only

we can also restrict our input variable using this concept.

For ex:-

```
class Addition
{
  public : int x,y;
};
```

```
int main()
{
  Addition a;
  a.x = -5;   ✓
  a.y = 105;  ✓
  return 0;
}
```

if I want $0 \leq x, y \leq 100$, i.e $x, y \in [0, 100]$

```
class Addition
{
  int x, y;
  public :
    void setX(int a)
    {
      if(a<0 || a>100)
        x = 0;
      else
        x = a;
    }
    void setY(int b)
    {
      if(b<0 || b>100)
        Y=0;
      else Y=b;  }
```

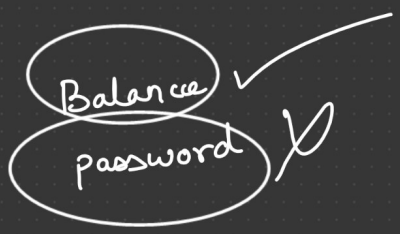
```
int main()
{
Addition a;
a.x = -5; → error
a.setx(-5);
a.setY(105);
```

X = 0 , Y = 0

```
cout<< a.getx();  ✓
cout<< a.Y;       X
```

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

};
```

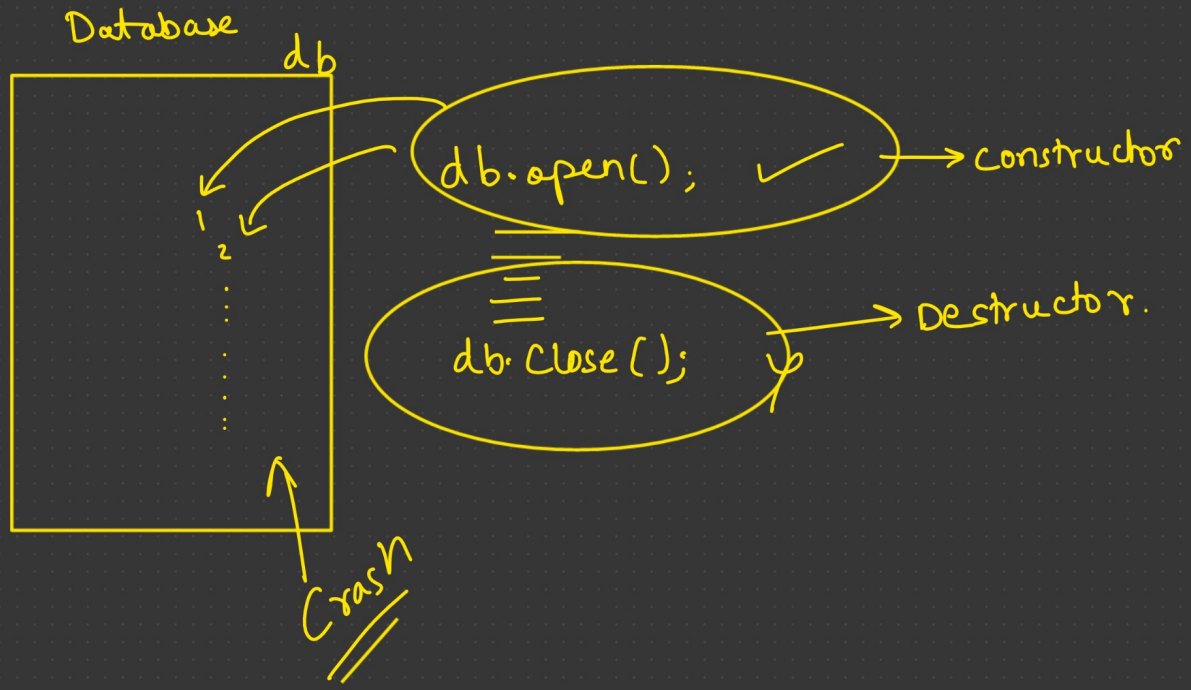


Destructor

- The destructor should clean up any garbage value, so it actually serves a useful purpose.
- Compiler called it automatically.
- Compiler implicitly create and call it.
- Its name is same as class name preceded by tilde(~).

```
~ Addition( ) {        }
```

- Destructor does not take any parameter. It means only one destructor exist for one class.



If we use "new" to allocate memory using constructor, then destructor should use "delete" to free that memory.

For ex:- `Addition *p = new Addition();`

`delete p;`