Const Member function

```
const int x = 5;
x = 10; → error
```

```
const Addition a(2,3);
a.add();
```

```
class Addition
{
  public:
    int x, y;
    Addition(int a, int b)
    {
      x = a;
      y = b;
    }
    int add() const
    {
      return x+y;
    }
```

Array of Objects

```
Addition a, b, c;

Addition a[100];
```

```
for(int i = 0; i<100; i++)
{
   cout<<a[i].x<<a[i].y;
}
```

Addition a[5]; ⟶ Default constructor needed to make array unintialized object.

Addition a[5] = { Addition(3), Addition(4,5), Addition(6)};

1) X = 3, Y = 0
2) X = 4, Y = 5
3) X = 6, Y = 0
4) X = 0, Y = 0
5) X = 0, Y = 0

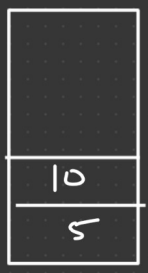
↳ we can intialize array object using multiple constructors. and remaining object will call default constructor.

Abstract Data Type (ADT)

```
class stack
{
    int s[100];
  public:
    void push(int x)
    {
      =
    }
    int pop()
    {
    } =
    int isEmpty()
    {
    } =
    int TOS()
    {
    } =
};
```

ADT

```
#include<stack>
int main()
{
  stack a;
  a.push(5);
  a.push(10);
  a.pop()
  if(a.isEmpty())
  {
  }
  return 0;
}
```



Operator Overloading

2 + 2 = int + int

2.5 + 3 = float + int

3 + 2.8 = int + float

2.5 + 2.7 = float + float

Time + Time

↓

Operator Overloading

```
class Time
{
    int hour, min;
    ==
    ==
};

int main()
{
    Time t1, t2, t3;
    t1 = t2 + t3;   ⟶ will it work ?
}
```

```
Time operator + (Time t)
{
    Time temp;
    int m = min + t.min;
    temp.hour = hour + t.hour + m/60;
    temp.min = (min + t.min) % 60;
    return temp;
}
```

T1 = t2 + t3; ✓

t1 = t2 + t3 + t4; ✓

* The overloaded operator must have atleast one operand that is user-defined type. This prevents you from overloading operators for standard types.

You can't use an operator in a manner that violates the syntax rules for the original operator.

For ex:- You can't overload the % operator, so that it can be used with single operand.

%x , x% ✗ x%y ✓