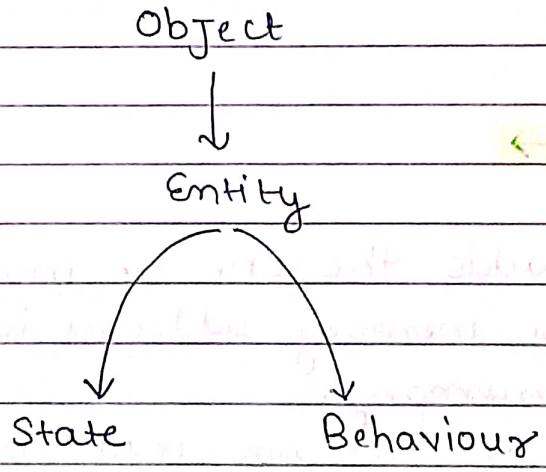
30/10/2023
Monday

* **Objected-Oriented Programming (OOPS)**

What? OOPS is a programming paradigm or technique in which things revolve around the object.

Why? OOPS relates programming with real-life applications. Increases readability, reusability, manageability.

* **Object →** Object is an entity which has state/properties and behaviour/functions.



* **Class →** Class is a user-defined or custom data type.

= Class is the blue-print of the object and object is the instance of class.

__/__/____

* **Size of empty class →**

```
class animal{

};

int main(){
    cout<<"size of empty class: "<<sizeof(animal);
}
```

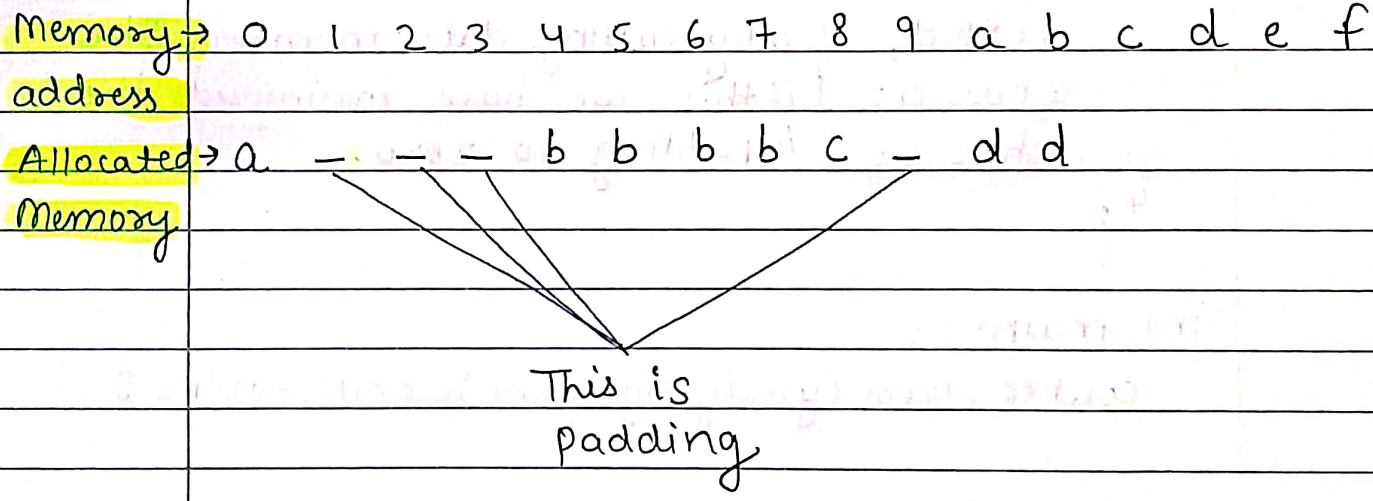
Size of empty class is 1 byte. To keep the track and to identify that something like this class exists, empty class is allocated with minimum possible space which is addressable in memory i.e. 1 byte.

* **Padding →**

Padding adds the one or more empty bytes between the memory addresses to align the data in memory.
The processor does not read 1 byte at a time. In 32-bit processor, processor reads 4 bytes at a time and in 64-bit, it reads 8 bytes.
Padding is done to minimize the CPU cycles.

For instance,

```
class padding{
    char a;
    int b;
    char c;
    short d;
};

int main(){
    cout << sizeof(padding) << endl; → O/P = 12
}
```



- Type with a size of 1 byte can be stored at multiple of one byte.
- Type with a size of 2 byte can be stored at multiple of 2 byte.
- Type with a size of 4 byte can be stored at multiple of 4 byte and so on.

__/__/____

* **Greedy Alignment →**

Greedy alignment is a technique used to minimize padding by ordering the data members of the class in decreasing order of size. Data member with largest size is placed first in the class.

For instance,

```
class greedyAlignment{
        int b;
        short d;      // By sorting data members like
        char a;       // this, we have minimized the
        char c;       // padding to zero.
};

int main(){
    cout<< sizeof(greedyAlignment)<<endl; →O/P= 8
```

Memory Address → 0 1 2 3 4 5 6 7 8 9 a b c d e f

Allocated memory → b b b b d d a c

// In short, greedy alignment reduces padding.

__/__/__

* **Access Modifiers** → They defines the scope of class attributes.

→ Public
→ Private
→ Protected

- **Public** → By making the class attributes public, we can access them inside and outside the class.

- **Private** → By making the class attributes private, we can access them inside the class only.
By default, they are marked private.

* **Class consists of →**

```
class {

    State / Properties
    int a;
    String str;          → Data members

    Behaviour / functions
    void func1() { }
    void func2() { }     → Member functions

};
```

Note → Object ki properties ko access krne ke liye dot (.) operator ka use hota hai.

__/__/____

Code →

```
class animal{

    //State
    public:
    int age;
    int weight;

    // behaviour
    void eat(){
     cout << "eating" << endl;
    }
    void sleep(){
     cout << "sleeping" << endl;
    }
};

int main(){
    //object creation
    animal ramesh;

    // static
    ramesh.age = 12;
    ramesh.weight = 45;
cout << ramesh.age << endl;
cout << ramesh.weight << endl;
    ramesh.eat();
    ramesh.sleep();
}
```

O/P →

```
12
45
eating
sleeping
```

__/__/__

* Getter and setter →

If we want to access private members outside class, we use getters and setters for that.

Getter and setter are the functions. Getter fetches the property and setter set the value of property.

Code →

```
class getSet {

    private:
    int value;

    public:
    int getValue(){
        return value;
    }
    int setValue(int v){
        value=v;
    }
};

int main(){
    //object
    getSet num;
    num.setValue(20);
    cout<< num.getValue()<<endl;
}
```

O/P → 20

__/__/____

* Dynamic object creation →

Code →

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

class animal {

    public:
    int age;
    void eat() {
        cout << "eating" << endl;
    }
};

int main() {
    // creating object dynamically
    animal *suresh = new Animal;

    // accessing object using (.) operator
    (*suresh).age = 12;
    cout << (*suresh).age << endl;
    (*suresh).eat();

    // alternative, using arrow
    suresh -> age = 12;
    cout << suresh -> age << endl;
    suresh -> eat();
}
```

O/P →

```
12
eating
12
eating
```

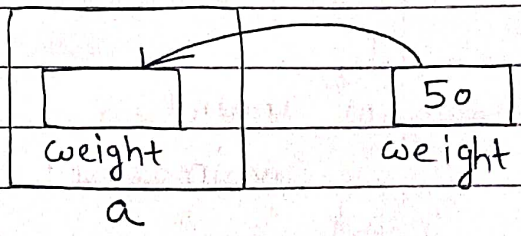
* **this keyword →** this is a pointer to the current object.

Code →

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

class animal{
    private:
    int weight;
    public:
    int getWeight(){
        return weight;
    }
    int setWeight(int weight){
        this->weight = weight; //or we can write like
        (*this).weight = weight;
    }
};

int main(){
    animal a;
    a.getWeight();
    a.setWeight(50);
    cout<<a.getweight()<<endl;
}
```

O/P → 50

Here, current object is a.



weight is copied to the weight of the object a.

Note → Jb bhi data members ko access krenge class mai, use this keyword. It is considered as good practice. __/__/____

* Constructors → Constructor is called whenever an object is created.

→ It initialises object.
→ Same name as class name.
→ Has no return type.

As constructor is called by default whenever an object is created but when we make constructor by our own, then this constructor overrides the default one.

1. Default constructor →

```
Class animal {
    public:
    string type;
    int age;
    int weight;

// default constructor
 animal ( ) {
   this → type = " ";
   this → age = 0;
   this → weight = 0;
   cout << "constructor called" << endl;
  }
};
int main ( ) {
    animal a;
  }
```

// the above attributes are // initialized for the given object.

O/P → constructor called.

Note → Object initialisation bina constructor ke bhi kar skte hai but using constructor is considered as good coding practices.

2. Parameterized constructor →

```
class animal {
    public:
    int age;
    int weight;
    string type;

// parameterized constructor
// single parameter
    animal (int age) {
        this -> age = age;
    cout << "parameterized constructor 1 called" << endl;
    }
// two parameters
    animal (int age, int weight) {
        this -> age = age;
        this -> weight = weight;
    cout << "parameterized constructor 2 called" << endl;
    }
};
int main() {
    animal a(10);
    animal b(10, 20);
    }
```

O/P → Parameterized constructor 1 called.
Parameterized constructor 2 called.

Note → Object creation mai jitne parameters pass kiye honge uske according, constructor call jayegi.

Note → Jb copy constructor create kiya to default constructor create krna hi padega nhi to error dega.

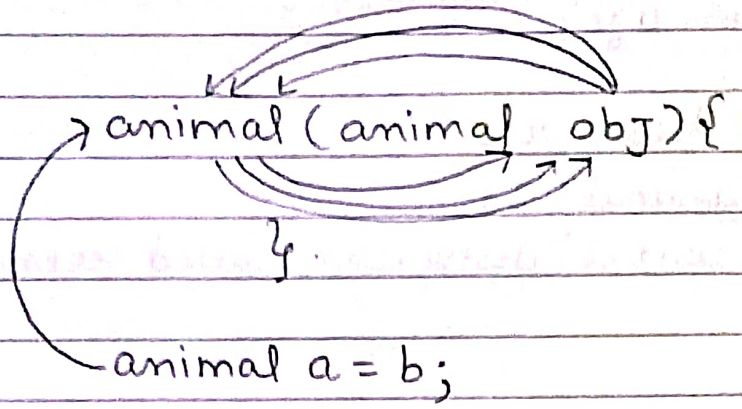
* Copy constructor →

```
class animal {
    public:
    int age;
    int weight;

// default constructor
  animal () {

  } ;

// copy constructor
    animal (animal &obj) {
        this → age = obj.age;
        this → weight = obj.weight;
  cout << " i am inside copy constructor " << endl;
  }
};

int main ( ) {
    animal a;
    animal b = a;
    animal c (b);

    animal * d = new animal (c);
  }
```

These are the two methods to copy the objects.

O/P → i am inside copy constructor
i am inside copy constructor
i am inside copy constructor

* If object is passed by value in copy constructor → Error aayega



Pass by value krne se repeatedly copy banegi. That's why copy constructor again and again call hoga and infinite loop mai fss jayega. To prevent this, pass by reference kiya object ko inside copy constructor.

* Destructor →

Constructor create krte time object initialisation ko jo memory allocation ho hua tha uska free krta h destructor.

→ In case of static, destructor is called automatically.
→ In case of dynamic allocation, destructor is to be called manually.

- It has no return type.
- No input parameters.
- It too has same name as class name.

→ tilde symbol precedes destructors.

__/__/____

Code→
```
class animal {
    public:
    int age;

    // destructor
    ~animal() {
        cout << "destructor called" << endl;
    }
};

int main() {
    animal a;
    a.age = 12;
    animal* b = new animal;
    b->age = 12;

    //manually calling destructor in case of heap
    delete b;
}
```

O/P → destructor called
destructor called

= Stack memory ke case mai jb scope khtm ho jayega to automatically call ho jayega destructor and heap memory ke case mai manually call krna pdega.

29/10/2023

* Local And Global variables →

Global variables →

(1) Written outside of function.
(2) Accessible to all functions. functions mai global variable ki copy nhi banti, actual memory location pr kaam ho rha hota hai.

Local variables →

(1) Written inside of function.
(2) Accessible inside the function scope only.

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

int x = 5; // global variable

int main(){
    x += 2;

    int x= 10; // local variable
    {
     int x = 20; // local variable
     cout << x <<endl; // 20
    }
    cout << x<<endl; // 10
    cout<< ::x <<endl; // 7
}
```

__/__/____

* **Memory layout of a program →**