Date: / /
Page:

Day - 99

OOPs

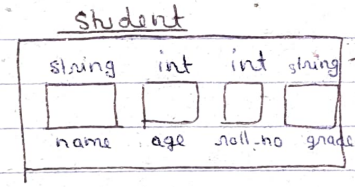
* **Object Oriented Programming:**

⇒ It is an approach or a programming pattern where the programs are structure around object rather than Function & Logic.

⇒ Sometimes, we are using a block of code again & again.

⇒ For ex: Student details.

⇒ So, to solve this problem, class & objects comes into picture.



⇒ Code

```
class student {                    → it's also like datatype (so, this is called user defined datatype)
    string name;                   → public:
    int age, roll_no;
    string grade;
};
int main(){
    student S1;
    S1.name = "Hirdesh";           → we can't access name, age, roll_no & grade directly because by default these all are private.
    S1.age = 20;
    S1.roll_no = 28;
    S1.grade = 'A+';
}
```

Date: / /
Page:

=> So, access directly, we have to make our class public.
=> To make public, we have to write public inside the class in the starting.

=> Public — is an access modifier.
=> There are three access modifiers —

(i). Public.
(ii). Private.
(iii). Protected.

* Class:
=> It is user defined data type.
=> Blueprint for creating objects.

=> To access directly or insert values directly into the class —
=> we will use functions —

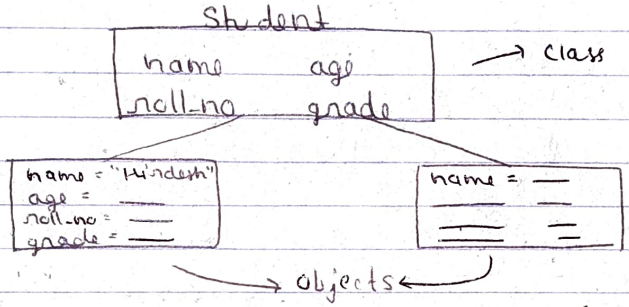
```
class Student {
    string name;
    public:
    void setname(string n){
        name = n;
    }
};
int main(){
    Student s1;
    s1.setname("Hirdesh");
}
```

Date: / /
Page:

* Object:

⇒ It is an entity that has a state & behaviour.

⇒ Anything that exist in physical world.



⇒ A class can have multiple objects.

class → class doesn't have any size.

```
class a{
    int b;
    char c;
};

int main(){
    a ob1;
    cout << sizeof(ob1);
}
```

⇒ When we only have int b then the size is 4 bytes. But if we also ~~written~~ have char c then the size is 8 bytes — why? char only takes 1 bytes.

⇒ So, our answer may be 5 bytes but in reality its 8 bytes.

⇒ So, here the concept is of Padding.

Date: / /
Page:

⇒

0	1	2	3	4	5	6	7
c	b	b	b	b			

1st 2nd

32 bit OS
64 bit OS
4 byte

⇒ So, in 32 bit OS, our compiler can only read 4 bytes at a time.

⇒ So, if they want to read b then it's in the 2nd cycle. So, it's takes some time.

⇒ So, to speed up this process, we can take extra space.

⇒

0	1	2	3	4	5	6	7
c	x	x	x	b	b	b	b

1st 2nd

This is padding

⇒ 1 byte = Multiples of 1
2 byte = Multiples of 2
3 byte = " " 3
4 byte = " " 4

⇒ Padding is used to process things faster at the cost of space.

⇒ Ex:

```
char c;
char b;
int a;
```

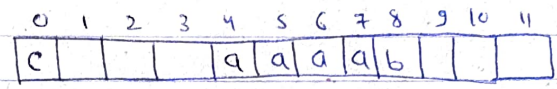
0	1	2	3	4	5	6	7	8
c	b	x	x	a	a	a	a	

∴ Padding 8 bytes

Date: / /
Page:

⇒ If we write this —

```
char c
int a
char b
```



Size = 12 bytes (not 9 bytes)

⇒ Because size should be divided by the biggest datatype size i.e. here is int.

⇒ But 9 is not divided by 4 so nearest divisible no. is 12.

⇒ That's why answer is 12.

⇒ So we have focus on alignment.

⇒ So that size should be min.

⇒ For this, we use greedy alignment —

⇒ First write the biggest datatype then small.

Ex:
```
int a
char b
char c
```

* Static vs Dyamic Memory Allocation

```
Student *S = new Student;
(*S).name = "Hirdesh";
S → name = "Hirdesh";
```