OOPs I

1

~~[illegible]~~ a programming style that revolves around objects

↓

basic entities which possess basic properties and functions

↓

involving these objects (eg. laptop, pen, students, teacher, etc) in our code

↓

object - oriented programming can be said to be oriented to the real world.

← the way objects behave or interact is how we want them to behave in code

↓

IMS eg. student, teacher, classroom, admin, etc objects. code revolves around the behaviour of these individual objects and their interaction.

* student :

$\left.\begin{array}{l}\text{name}\\ \text{Roll no}\\ \text{Age}\\ \text{Address}\\ \text{contact no.}\end{array}\right\}$ properties

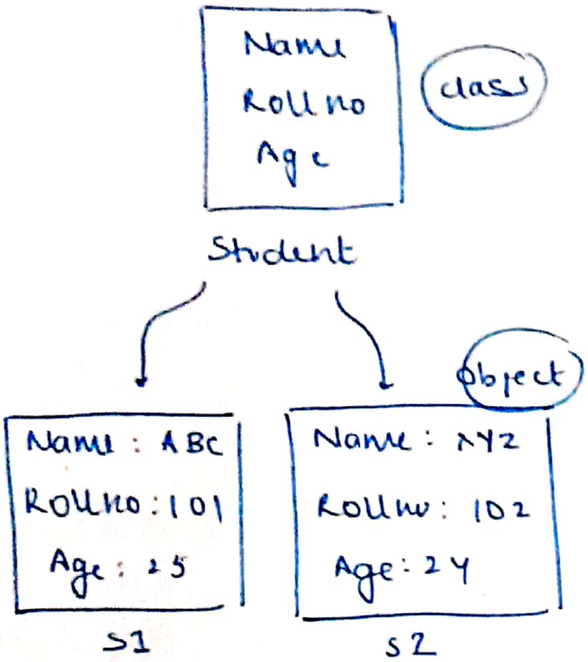
→ change Address
→ Set Roll No.

} functions

(properties + functions) } object

Student : name
Rollno
Age

wanting to create 20 students
↓
defining all properties from scratch is a tedious task for 20 students
↓
so we create a blueprint of these properties with no values, and whenever req we create its copy and put in values to create an object
↓
the blueprint : class
the copy with values : object



hence it gives us info what exactly the object comprises of in terms of properties and functions.

OOPs
- class
- object

* class in code :

```
class Student {
    int rollNo;
    int age;
    char name [100];
};
```

class name

keyword

error prone

→ the properties of the student class are now specified and need not be specified again and again
↓
We can make its copy and it will already have these values and we need to just put values in.

* creating objects of a class (any entity)

→ user defined datatypes

true for all instances

```
class student {

    int rollNumber;
    int age;

};
```

→ student s1;

allocation of almost 8 bytes with two properties garbage value

static allocation
(object of student class created)

↓

like int a

↓

4 bytes of memory with name a and random value

student * s2 = new student;

↓

dynamic allocation (heap)

int * a = new int;

* including our files in other C++ files

↓

#include "students.cpp"

→ for the compiler to recognise references we make

→ file name ↓ should be in same directory

int * a = new int
*a = 5

int a;
a = 5;

needs to be dereferenced

↑

a pointer

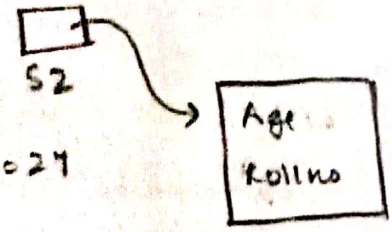
↑

s2.age = 24

↓

(*s2).age = 24 ✓

s2 → age = 24 ✓ (shortcut)



s1

method to access member

↑

s1.age = 25
s1.rollno = 101

s1 = 24 ✗ → we need to specify which prop to put value in

* Access modifiers → which properties and methods are visible and useable outside
  - i) private
  - ii) public
  - iii) protected → later

* private : properties and methods visible only inside
  ↓
  accessed outside will fetch an error.

* public : properties and methods are visible outside and can be accessed outside the class

(•) by default : private

```
class student {
    public:
        int rollNumber;
        int age;
};
```

```
void setAge (int a)
{
    age = a;
}
```

```
class student {
    public:
        int rollNumber;
    private:
        int age;
    public:
        void display ()
        {
            cout << age << " " << rollNumber;
        }
```

```
int getAge()
{
    return age;
}
```

class methods can operate on private data

5

* accessing methods

↓

same way as attributes are

↓

s1. display()

s2 → display() / (*s2) .display()

(•) making properties private is to control access of the data from wrongful or errorful setting

↓

using getters and setters makes sure that a specified format is used to work on private data.

↓

ex. age is never -ve

↓

we can add constraints for -ve values to not be able to update and throw errors

← or password protect it, like ask for password, send in arguments and check if passwords match, then change value, else throw error.

↑

(giving certain users access)

* constructors :

(Default const.)

student() {

}

student s1;

↓

s1. student ();

↓

is created by default with every class.

→ whenever this is written

↓

a special function with name same as class is called and initializes them with garbage value or default value

↓

if nothing specified

→ called once in its lifetype for each object

↙

special function

↓

constructor (default)

i) same name

ii) no return type

iii) no input arg

student * s3 = new student ;

↓

(*s3) . student()

↓

s3 → student()

student() {

cout << "constructor called" << endl;

}

→ once user defined → default is not called or referred

↓

replaced by our function (our constructor)

↓

function overloading is allowed.

↓

thus many different constructors can be created to handle different situations

↓

only 1 will be called.

parameterized constructor ← student (int r) {

cout << "constructor 2 called" << endl;

rollNumber = r;

}

otherwise optional

↑

only req. when confusion in scope is present for members

inserting its value in itself

↑

closer scope

↑

* Student (int rollNumber) {

this → rollNumber = rollNumber;

}

→ both variables same ?

↓

this keyword to refer to current object of the class

↓

holds address of current object

↓

the address of the memory block created for the object implicitly sent.

→ refers to current object which has been created or is being initialized

↳ pointer

↳ special keyword

* There are other functions that we get when we create an object of a class

↓

i) copy constructor → creates an object that is copy of another object

a constructor at the end of the day

↓

called initially when object is created.

Student s1(s2);

↓

s1 is a copy of s2

↓

s1.student(s2);

↓

student * s3 = new Student(20, 2001);

↓

s4.student(*s3)

↓

student * s5 = new student(*s3);

(*) stay careful using dynamic allocation

ii) copy assignment operator

student s1(10, 1001);
student s2(20, 2001);

if we want s1 to be copied to s2 after creation, then

s2 = s1

↓

copy assignment operator

iii) Destructor → same name as class

No return type

No input arguments

↓

object memory deallocation

↓

can be only 1 → no arguments

~Student()

~ introduced for destructor

statically allocate memory gets lost quickly

delete s3

↓

dynamic should be manually removed by calling delete

0,4 5,8

student s5 = s4; → copy constructor called

↓

rather than copy assignment operator

* fraction class : i) numerator
  ii) denominator } private.

iii) constructor, with values for num and den.

↓

to stop garbage value from getting inside our numbers.

iv) printing in fraction format

↓

num / den;

v) add 2 ~~...~~ fraction.

↓

f1.add(f2) → update in f1 only.

↓

return type void only.

↓

simplify the fraction after addition (gcd)

↓

make seperate function and work accordingly

9

```
void add (Fraction & f2)
{
    int lcm = denominator * F2.denominator;
    int x = lcm / denominator;
    int y = lcm / F2.denominator;

    int num = x * numerator + y * F2.numerator;

    numerator = num;
    denominator = lcm;

    simplify();
}
```

Fraction f2 = main.f2 → copy constructor called ↓ new object formation ↓ to avoid ↓ reference variables ↓ Fraction & F2 = main.f2 ↓ F2 and main.f2 point to the same memory block ↓ Now since F2 reference variable points to the location, the main object can also be changed (call by reference) ↓ constant reference to be passed ↓

```
void add (Fraction const & F2)
```

```
int i = 5;
int & k = i;
int const & k = i;
```

$\boxed{5}$ i, k

↑ we want that changes are not allowed through the reference variable ←

vi) multiply function
(simplify)

* Complex Numbers class : (3 + 4i) :

i) real

ii) imaginary

iii) constructor

iv) addition (c1.plus(c2))

v) multiply (c1.mult(c2))

↓

put in c1 and let c2 remain unchanged

vi) print
(proper format)

take choice input

↓

1. for plus
2. for multiply

Scanned by CamScanner