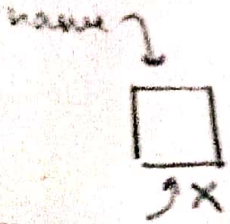
OOPs 2

```
class student {
    private:
        int age;
        char *name;
    public:
        student(int age, char *name)
        {
            this->age = age;
            this->name = name;
        }
};
```

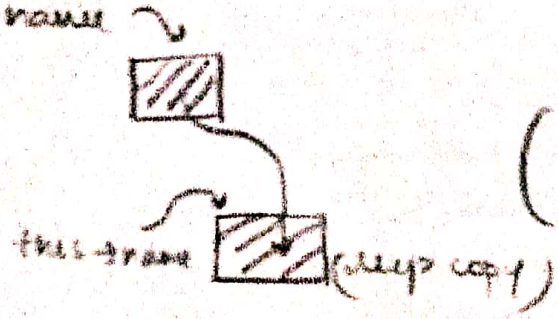
- private: we don't know the size here, hence we wait tru the constructor to actually work with or allocate memory for the pointer to point.
- this->name = name; → makes object's name pointer to pointer at the same loc pointed to by the arg. sent name pointer
- only DTh index address copied (shallow copy)



- if any changes are made to this location Through that pointer, then that would be seen in our name as well
- shallow copy
- to make sure that our name doesn't change, we should copy the contents of the arg pointer to the memory pointed by our pointer (deep copy) ←

```
* main file
    char name[] = "abcd";
    student s1(20, name);
    s1.display()
    name[3] = 'e';
    student s2(24, name);
    s2.display();
    s1.display();
```

both showcase same result for name



```
student (int age, char * name)
{
    this → age = age;
    this → name = new char [strlen (name) + 1]
    strcpy (this → name, name);
}
```

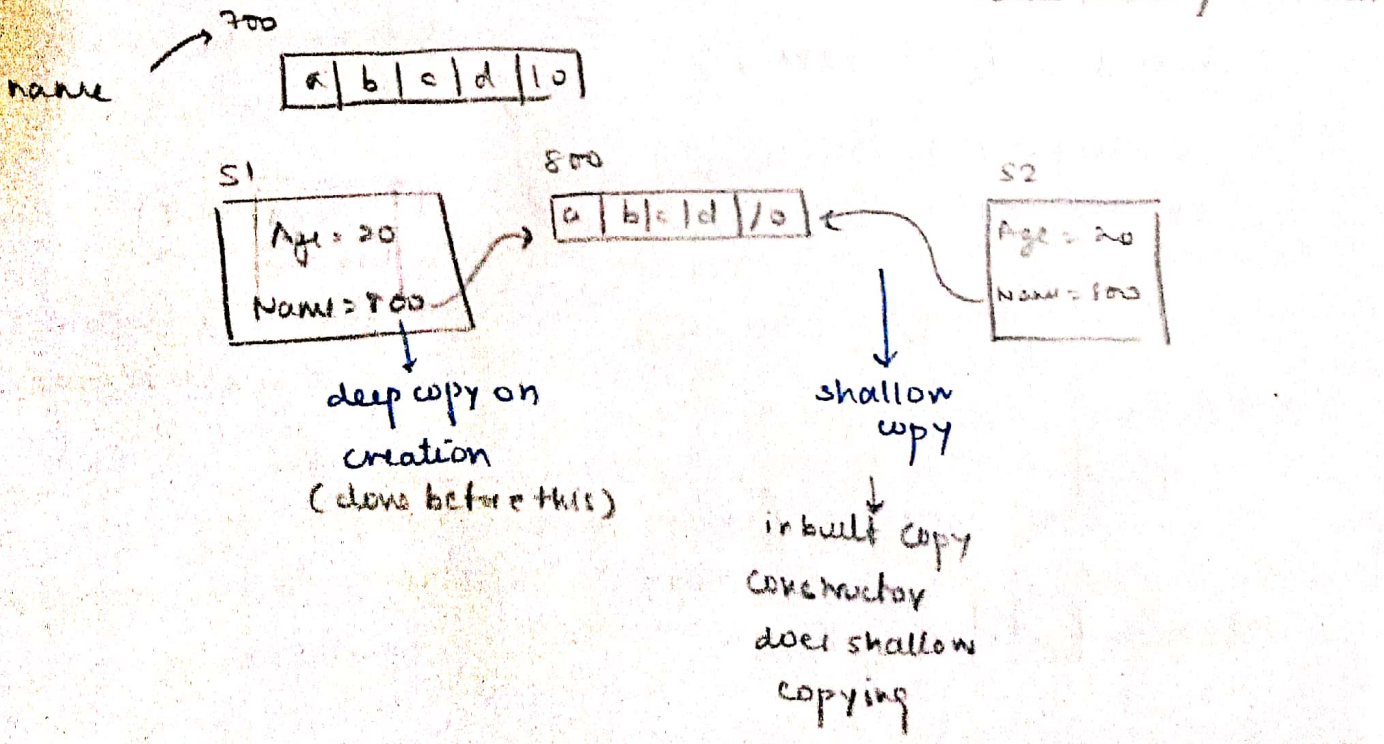
we have to insert '\0' also as well

→ deep copy

* making our own copy constructor

```
char name [] = "abcd";
student s1 (20, name);
s1.display();        // abcd
student s2 (s1);
s2.name [0] = 'x';
s2.display();        // xbcd  → shallow copy
s1.display()         // xbcd
```

↓ refer to the same memory location



```
Student (student const & s)
{
    this -> age = s.age;
    // (this -> name = name);   // shallow copy
```

deep copy → `this -> name = new char [strlen(s.name)+1];`

```
    strcpy (name, s.name);
}
```

* if we don't pass s by reference, then the copy constructor will get into an ∞ loop, as it will create a copy of the object coming from main and since copy function is itself calling a copy function without any limit, hence ∞ loop. so pass by reference.

* to stop any illegal changes, we can pass const. reference variable such that changes through that path are not allowed.

int a = 5;
int const b = a; → won't be able to change b since b is constant

↓

constant variables have to be initialized

int &y = a; → even reference variables have to be initialized

↓

handling const and reference data members in a class.

↓

Const variables of a class get initialized by default constructor by garbage and hence cannot be used after

(so it throws an error)

↓ compilation

↓

we can't assign value in class definition as well because object specific behaviour might be different

↓

even while making our own constructor and passing values as arguments, we won't be able to assign value, since we are not initializing rather assigning (variable is already created)

(memory is already allocated (so assigning))

↓

we need initialisation list to counter all these problems and work with const variables.

↓

such variable ex. can be const rollno. → since we don't want to change by mistake

this not required

(5)

```
Student (int r) : rollNumber (r), age(age)
{
}
```

rollNumber gets initialized to r

* same goes for reference variables

* construction of our own constructor is required !!

```
class student { Public :
    int age ;
    const int rollNumber ;
    int & x ;
```

→ reference variable for age

initialisation list

```
student ( int r, int age ) : rollNumber (r),
                             age(age), x(this->age)
{
}
};
```

Here this is req. to avoid confusion b/w parameter 'age' or member 'age'

* constant functions.

```
int b = 10;
int const a = b;
```

↳ similarly we can construct objects of our class

↓

Fraction f1(10, 2)

const Fraction f3

↓

we won't be able to change any property of f3

↙ can safely call constant member functions

↓

functions that don't change any property of the current object

to note keyword

↓

after declaration of function (normal)

```
int getNumerator() const {
};
```

* if we set function that mention const to functions that change values will be caught by compiler

* pay special attention to make functions that don't change any value to const

↓

so that even constant objects can access them.

static properties of a class

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

Student S1;

RN
Age

Student S2

RN
age.

any non static members get copied to these

→ same copy for the entire class

↓

seperate copies are not made for each object

↓

maintained by class itself

```
cout << student::totalstudents;
```

scope resolution operator

S1.totalstudents ⊗

↓

compiler allows

↓

logically incorrect

↓

we can even change using one object and that will be reflected for the whole class

static variables initialized outside class

```
int student::totalstudents = 0;
```

int → datatype of our variable

student → member of student class

↓

established by class name and Scope resolution operator

* regular update of static member → constructor can be used.

↓

whenever object is created → totalstudents ++;

* Static functions → only access and modify static variables
  ↓
  does not have this access

ex getter for private static member

↓

```
static int getTotalStudents{

    return totalStudents;

}
```

↓

```
student :: getTotalStudents();
```

824 | 3 23 −824 32 −4 824 435 −5775

1 sim

* operator overloading

↓

extending the functionality of pre existing operators, such that they work for user defined types as well. (classes)

↓

F3 = F1 + F2

F3 → Fraction class object

+ → won't work as it is not defined for our fraction class

↓

extending '+' functionality, such that it can work even for user made classes is called operator overloading

* the add function made for fraction class was updating the object which called the add function.

i.e. f_1.add(f_2)

↓

f2 gets added to f1

↓

we can change that and put the answer in a new variable object by changing the return type and some operations of the function

```
Fraction add(Fraction const &f2)   // const → not making any changes to the current object
{
    int lcm = denominator * f2.denominator;
    int x = lcm / denominator;
    int y = lcm / f2.denominator;

    num = x * numerator + y * f2.numerator;
    den = lcm;

    Fraction fnew(num, den);
    fnew.simplify(); return fnew; }
```

to use '+' in place of add.

↓

we need

↓

keyword

operator to overload

```
Fraction operator+ (Fraction const & f2)
{
    same definition as before
}
```

* same can be done for any operator, the logic need to be thought of. thats basically it

↓

when the two operands are hit with that operator, we get the operator+ function called and corresponding result

⇒ overloading '==' (binary operators)

not making any changes to the current object

```
bool operator == (Fraction const & f2) const
{
    return (numerator == f2.numerator &&
            denominator == f2.denominator);
}
```

0	1	2	3	4	5	6	7	8
324	3	23	-327	32	-4	324	435	-5775

* operation overloading 2


```
int a = 5;
++a
cout << a; // 6
```

```
int a = 5;
int b = ++a;
```

first update a, then assign it to b.

(++) → pre-increment
↳ post-increment

unary operator → requires only one input to work with

* pre-increment

```
++f1;
```

this → automatically gets passed to this, we donot need to specify any thing.

```
int a = 5;
++(++a)
cout << a; // 7
```

$(f_3) = ++(++f1)$ → should also work, but without returning

f_3 value shows up here

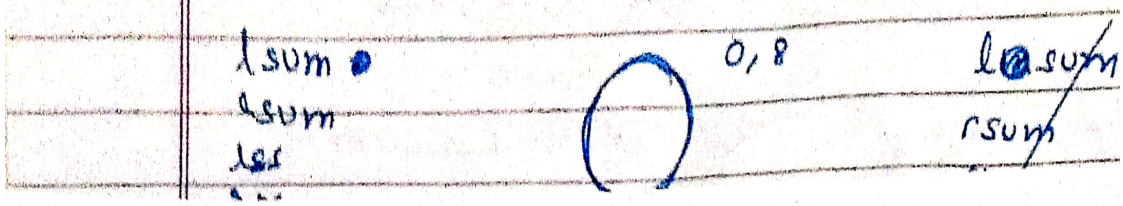
this gets only incremented once when returned an object.

↓

before getting assigned to a variable the value stays in the memory buffer

since this part is not being received anywhere, temporary memory is being created and ++ is called on it,

since we returning by value → second level of increment is happening on the system copy. and not our variable




```
Fraction & operator ++ ()
    {
        numerator = numerator + denominator;
        simplify();
        return *this;
    }
```

* post - increment

↓

first use, then increment

↓

(i++)++ ✗ → does not work in reality as well.

to indicate i++ (post increment)

```
Fraction operator ++ (int) {
    Fraction fNew (numerator, denominator);
    numerator += denominator;
    simplify();
    fNew.simplify();
    return fNew;
}
```

operator overloading :

+=

```
int i = 5, J = 3;
i+=J  →  i = i+J
```

↓

```
(i+=J)+=J;  → nesting allowed
```

ordering

```
Fraction & operator+=(Fraction const &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();
    return *this;
}
```

* Dynamic Array → i) data array
ii) addElement function
iii) nextIndex
iv) size (can be derived from nextIndex)
v) capacity

a) constructor
b) copy constructor (deep)
c) copy assignment (deep)
d) addelement
e) get element
f) print

```
class DynamicArray {

    int * data;
    int nextIndex;
    int capacity

    public:

    DynamicArray () {
        data = new int [5];
        nextIndex = 0;
        capacity = 5;
    }

    DynamicArray (DynamicArray const & d)
    {
        this -> nextIndex = d.nextIndex;
        this -> capacity = d.capacity;
        this -> data = new int [d.capacity];
        for (int i = 0; i < d.nextIndex; i++)
        {
            this -> data[i] = d.data[i];
        }
    }
```

this -> data = d.data
↓
shallow copy

deep copy

```
void operator = (DynamicArray const &d)
  {
     this -> nextIndex = d.nextIndex;
     this -> capacity = d.capacity;
     this -> data = new int [capacity];
     for (int i=0; i<d.nextIndex; i++)
       {
          this -> data[i] = d.data[i];
       }
  }
```

```
void add (int element){
   if( nextIndex == capacity )
     {
        int * temp = data;
        data = new int [capacity * 2];
        for (int i=0; i < temp -> nextIndex; i++)
          {
             data[i] = temp[i];
          }
        delete[] temp;
        capacity *= 2;
     }
   data[nextIndex] = element;
   nextIndex++;
}
```

```
void add(int i, int element){
    if(i<nextIndex)
        data[i] = element;
    else if(i>=nextIndex)
        add(element);
    else return;
}

void get(int i) const {
    if(i<nextIndex){
        return data[i];
    }
    else return -1;
}

void print() const {
    for(int i=0; i<nextIndex; i++){
        cout << data[i] << " ";
    }
    cout << endl;
}
```