Object-oriented Programming

(*) Procedural programming :- (task → Sub task)

(1) focus is on process or actions that a program takes
(2) program are typically collection of function
(3) Data is passed as an argument to fun^{n}.

Limitations :-

As programs get larger they become more:
- difficult to understand
- difficult to maintain
- difficult to extend and debug
- can't reuse the code
- fragile and easier to break.

(*) Object-oriented programming
classes and object

(1) focus is on classes that model real world domain entities
(2) Allow developers to think at a higher level of Abstraction
(3) can be used in large programs.

Page No.
Date

(*) Classes ?

- blueprint from which objects are created
- a user defined data type
- has attributes (data)
- has methods (functions)
- can hide data and methods
- Provide a public Interface.

Eg:- Account, std::vector, std::String ... etc.

(*) Objects ?

- Created from a class
- represent a specific instances of a class
- can create many, many objects
- each has its own identity
- each can use the defined class method

Eg:- frank's account is an instance of an account class

jim's account is an instance of an account class

Both are different and can be used together.

Eg:-
```
#include<iostream>
using namespace std;
class Hero{
    // Properties
    int Health{};
};
int main(){
    // creating object
    Hero h1;
    cout<<"size:"<<sizeof(h1);
    return 0;
}
```

class → object

4 byte // because it only contain single property of int type.

Note: if the class has no properties then the size of object of instance of class will be 1 byte to show its significance and.

```
Class Account{
};
Account Acc;
sizeof(Acc);
```
→ 1 byte.

(*) Note: To access the properties or the data members of the class by using object with dot operato

Eg:-
```
Hero h1;

cout << "Health is : " << h1.Health << end
```

(•) Provided that Data member is declared in public

(*) Note: To access the private data member of a class we can use getter and setter methods.

Eg:-
```
private:
    int Health{};

int getHealth(){
        return Health;
    }

void setHealth(int h){
        Health = h;
    }
```

```
int main(){

    Account Acc1;
    // Getter                    Acc1.
Cout<<" Health is " << getHealth() <<endl;

    // Setter
~~Acc1 Acc1~~. Acc1.setHealth(70);

Cout<<"Health:" << Acc1.getHealth;
                          → // 70.
```

Q what is Padding ?

Q what is greedy alignment ?

(*) Note : we can also create object of the class dynamically.

Eg:-

```
Account * acc1 = new Account;
Cout<<" Health is " <<(*acc1).Health ;
Cout<<" level is" << (* acc1).level ;
                (or)
Cout<<" level is" << (acc1)→level ;
```

(*) Creating objects :-

Eg:- Account object1;

Eg:- Account accounts[] { object1 };

Eg:- std::vector<Accounts> account1 { object1 };

Eg:- accounts1.push_back(object2);

Eg:- Account *abc = new Account();

Eg:- player players[] { frank, Hero };
↳ It is an array of player object which has two entities.

(II) Constructor:

It is a special method that is automatically called when an object of a class is created.

Note: The constructor has the same name as the class, it is always public, and it doesn't have any return value.

Eg:- Class Myclass {

public:

Myclass() { // Constructor

Note: for better understanding of constructor visit udemy c++ // constructor - Inheritance section

```
    cout<<"Hello, Constructor called";
  }
};
int main() {
    Myclass obj1;
    return 0;
}
```

Note: when the user defined constructor is called using object, default constructor has been destroyed.

(*) Parameterised constructor:

Constructor with parameters.

eg:-

```
Myclass( int x, int y, char ch){
    brand=x;
    age=y;
    name_letter = ch;
}
```

(*) "this" keyword / Pointer

'this' is a keyword that refers to the current instances of the class or can said the object of the class.

There can be 3 main usage:

(•) It can be used to pass the current object as a parameter to another method.

(•) It can be used to refer current class instance variable

(•) It can be used to declare indexers.

Eg:-
```
#include <iostream>
using namespace std;

class Employee {

public:
    int id; // data member
            (or)
            Instance variable
    String name;
    float salary;
```

```
// parametrised constructor
Employee(int id, string name, float salary)
{
    this->id = id;
    this->name = name;
    this->salary = salary;
}
void display(){
cout<<id<<" "<<name<<" "<<salary;
}
};
int main(){
Employee e1 = Employee(101, "xyz", 8900);
Employee e2 = Employee(102, "abc", 9090);
e1.display();
e2.display();
return 0;
}
```

o/p = 101 xyz 8900

(*) Copy constructor:

It is a member function that initalizes an object using another object of the same class.

A constructor which creates an object by initalizing it with an another object of same class, which has been created previously.

Eg:- Sample (Sample & obj1){

```
    this->Id = obj1.Id;
}
```

Eg:- Account obj1 (70, 'A'));

```
// Copy constructor
Account obj2(obj1);
```

(Copy all the data members of obj1 to obj2.

```
// making user defined copy constructor
Account(Account &temp){
    this->health = temp.health;
    this->level = temp.level;
}
```

Eg: Type::Type (const Type &source);

Account::Account (const Account &obj1);

Page No.
Date

Note: Every class has its predefined copy constructor but user can also defined its own. After user defined copy constructor, default constructor get destroyed.

Note: The default copy constructor always do shallow copy.

Shallow :- Accessing same memory with two different name.
Copy!

Eg!- Account obj1;

```
obj1.setHealth(12);
obj1.setLevel('A');
char name[7] = "Babbar";
obj1.setName(name);
obj1.print(); → 12, A, Babbar
```

// use default copy constructor

```
Account obj2(obj1);
obj2.print(); → 12, A, Babbar
```

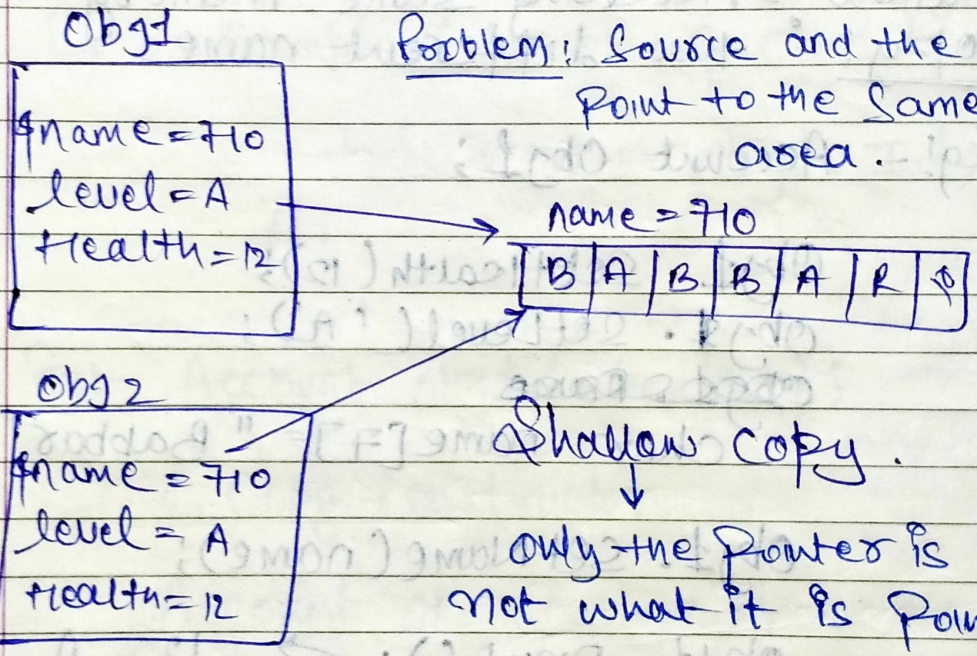
Now,

```
obj1.name[0] = 'G';
obj1.print();
```

→ 12, A, Gabbar

```
obj2.print();
```

→ 12, A, Gabbar



Problem: Source and the new obj point to the same data area.

Shallow copy.

↓

only the pointer is copied, not what it is pointing to

↔ To solve this problem, we will create our own copy constructor and use deep copy.

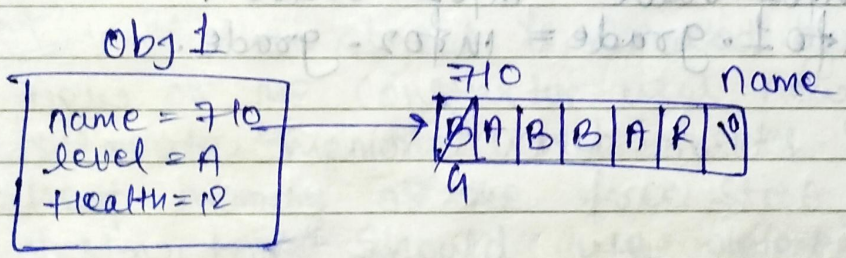
(•) use STL classes as they already provide a copy constructor

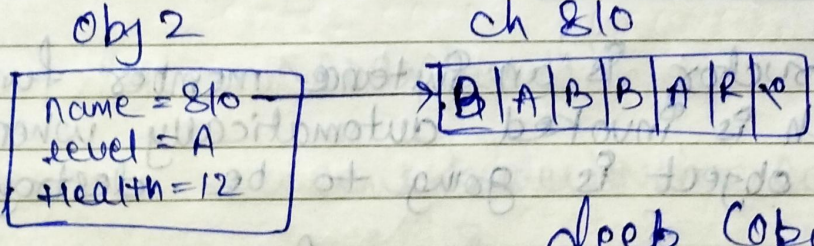
(•) Avoid using raw pointer data member

→ Create new storage and copy values

(*) deep copy.

```
// copy constructor
Account ( Account & temp){
    char *ch = new char[strlen(temp.name)+1];
    strcpy(ch, temp.name);
    this->name = ch;
}
```





deep copy.

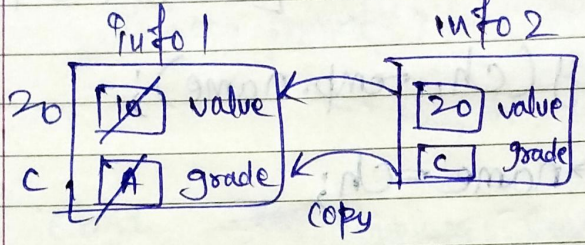
(*) Best Practise for copy constructor

(•) Provide a copy constructor when your class has raw pointer members
(•) Provide the copy constructor with a const reference parameter

(*) Copy Assignment operator (=)

Eg:-

```
Students info1 (10, 'A');
Students info2 (20, 'C');
info1 = info2;
```




```
info1.value = info2.value.
info1.grade = info2.grade.
```

(#) Destructor :-

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed.

Destructor is the last function that is going to be called.

Page No.
Date

(*) Destructor is also a special member fun^{n} like constructor used to destroy the object.

(*) It has same name as of class with preceded by a tilde (~) Symbol.

(*) Neither require any argument nor return any value.

(*) It automatically called when object goes out of space.

Note: If the object is created by using the new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

```
Eg:- ~MyClass()
     {
        cout<<"destructor called";
     }
```

Note: Destructor is automatically called only for static object, for dynamic object we have to use delete keyword.

Eg:- Student *info2;

// automatically deleted by destructor

Student *info1 = new Student();
delete info1;

// we have to use delete.

H/W

Q Initialization list ?

(11) Static keyword or Datamember :

↳ It doesn't belongs to the class and doesn't need object to be called.

Eg:-
int Account::timeToComplete = 5;

Syntax:

data-type class-name :: name = value;

↳ Scope resoultion operator

Note

declare outside the class and main.

int main() {
cout << Account::timeToComplete;
}

```
class Account {
public:  // declare
        static int timetocomplete;

};
// Initialize Static member
int Account :: timeToComplete = 5;

int main()
  {
    cout << Account :: timeTocomplete;
  }
```

→ 5

(#) Static function : It is a member function that is used to access only static data members.

(•) It can be called even if no object of class exists.

```
Eg:- class Student {
     public:
          static int random(){
          cout<<" Random static";
          }
     };
     int main(){
          Student :: Random();
```