```
int main(){
    A _a;
    B _b;
    _b.display(_a);
    return 0;
}
```

aim :- provide easier solution to real-world problems by implementing real-world entities

DELTA Date / / Pg No.

~ Object oriented Programming ~ (OOPS)

what is this ??

→ (to design a program using objects and classes) — type of example or pattern or model

A programming paradigm (technique) in which things work around an object → we need to include concept of object in our coding technique

→ A type of entity having 2 things in all
- Having state or property (existence)
- A particular behaviour

In real life, like microphone, sights, laptops etc

In short, we want our programs to be related closer to real-world

But why ??

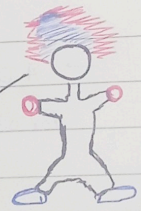
OOPS ← In order to increase readibility of our program, increase its manggibility, extensibility → extensive nature to increase

object → entity → (State/properties), (Behaviour)

For eg:-

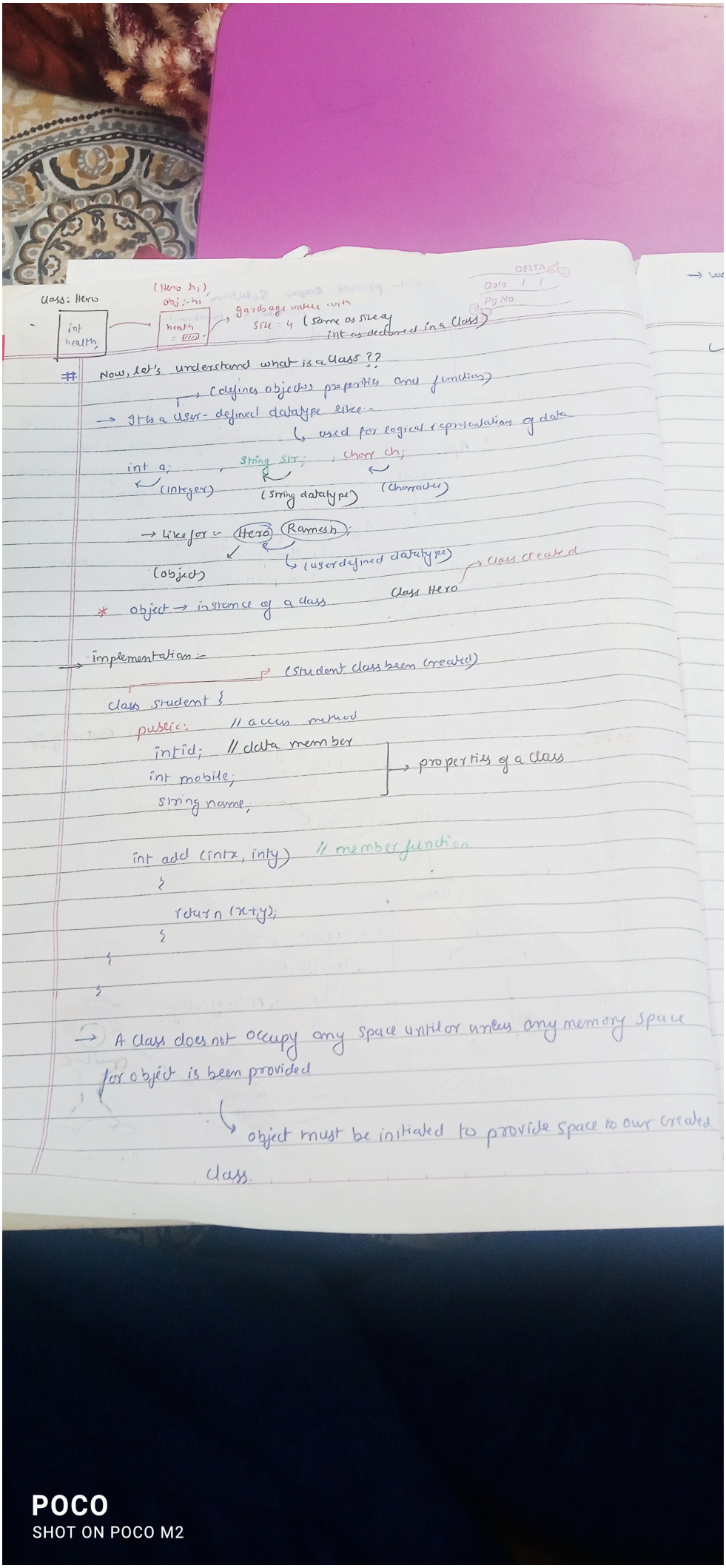
Hero
- Name → "Tekken" / Paul
- Health → 70
- Level = 'A'
- Behaviour → attack() / defensive()

↳ properties of entity



oops isn't hard my friend!!

(Random)



We can create class further in the same file or working with other file by using #include "Hero.cpp"
↳ class name

* object being a runtime entity, this an instance of class → it can represent a person, place or any other item
↳ it can operate both data members as well as member functions

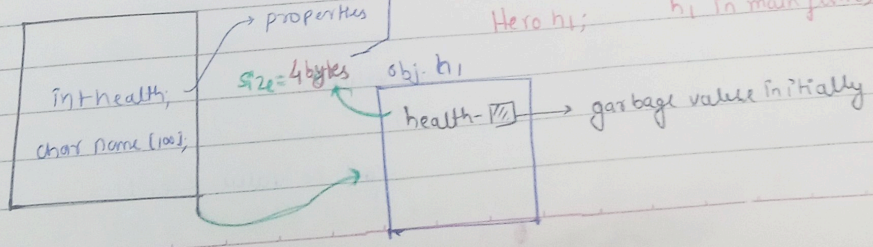
for eg. Student s = new Student();
↓
(when object is created using new keyword, the space allocated to variable is a heap memory
↳ starting address being stored in Stack memory

→ If new keyword is not been used to create an object, no space allocated in heap memory → object contains null value in Stack

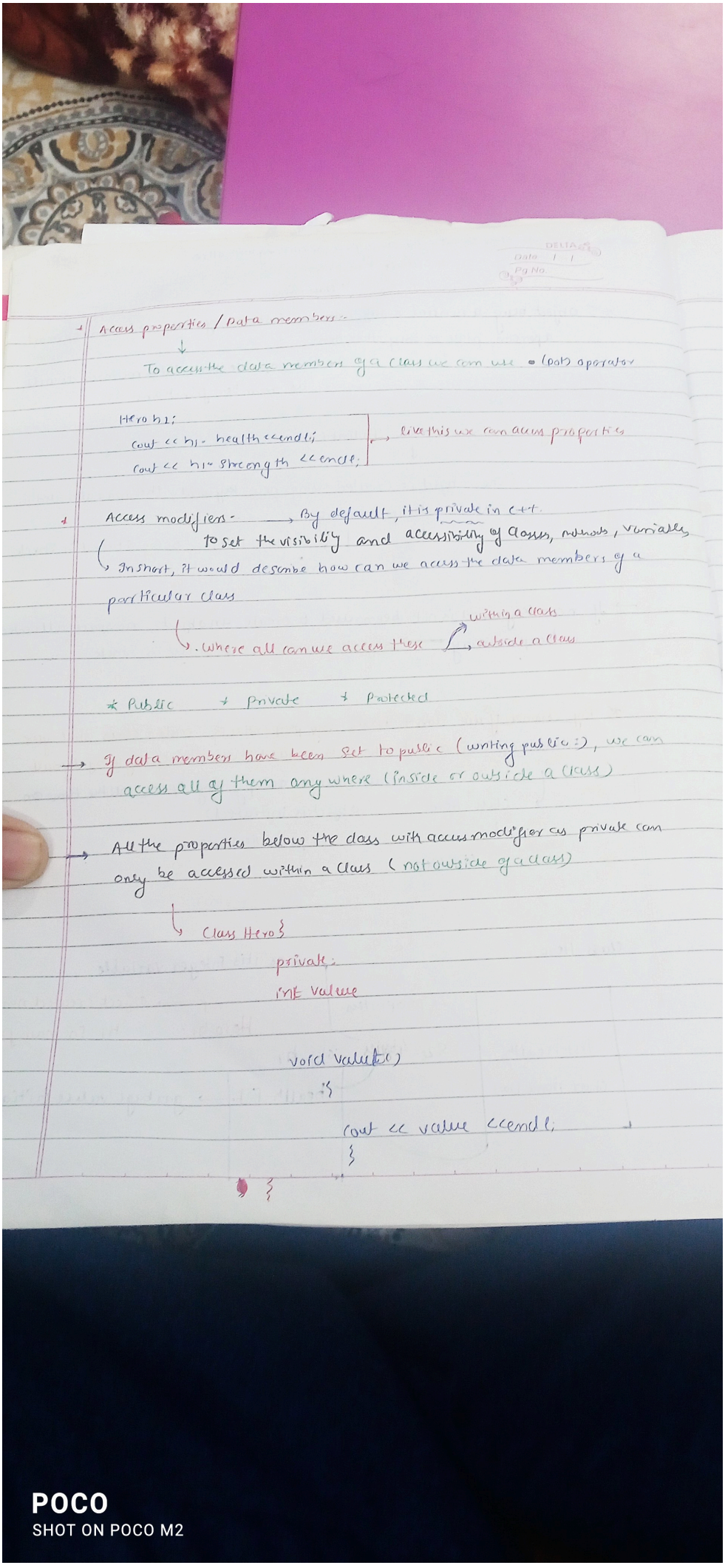
* By chance, if we don't have any properties of created class, still after initializing our object, the class would have size of 1byte by default
↳ (class doesn't accupy any space till the time an object is initiated)

class Hero
{
→ no properties used
//public:
}

class Hero



Hero h1; → object created named h1 in main func()



Access properties / Data members :-

↓

To access the data members of a class we can use . (dot) operator

```
Hero h1;
cout << h1.health << endl;
cout << h1.strength << endl;
```

→ like this we can access properties

Access modifiers :- → By default, it is private in c++

↳ To set the visibility and accessibility of classes, methods, variables

↳ In short, it would describe how can we access the data members of a particular class

↳ Where all can we access these → within a class / outside a class

* Public * Private * Protected

→ If data members have access set to public (writing public :), we can access all of them anywhere (inside or outside a class)

→ All the properties below the class with access modifier as private can only be accessed within a class (not outside of a class)

↳

```
Class Hero{
    private:
    int value

    void value()
    {
        cout << value << endl;
    }
};
```

POCO
SHOT ON POCO M2

To set values via objects -

```
main()
{ h1.sum = 70;        cout << h1.sum << endl;
}
```

→ this reduces (neglects) scope of printing garbage value

\+ Protected access modifier → during the child class concept we would understand it clearly (functions and variables cannot be accessed outside class except child class) ↓ Same class and/ Sub class

↳ forinheritance

\+ Concept of getter / setter :-

↳ To access properties (that are private) outside a class, we use these functions to access these members

```
class Hero
{
    int value;
    char ch;

    int getvalue()
    {
        return value;
    }

    char getch()
    {
        return ch;
    }
```

→ getter functions

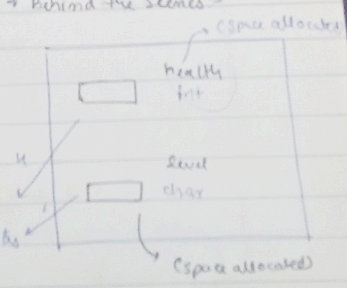
(setter function)

```
void setvalue (int value1, char name)
{
    // we can also give conditions
    if (name == "A")
    {
        value = value1;
    }
}
```

↓ value stored in a variable in private class

```
main()
{
    Hero ramesh;
    ramesh.setvalue(70, 'A');
    cout << ramesh.getvalue();
}
```

\+ Behind the scenes :-



size = (4+1) = 5 bytes

↳ but on compiler it would give = 8 bytes

↳ due to padding and greedy allignment

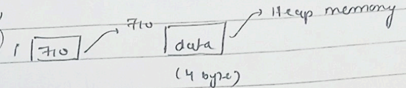
Global variables, Static variables ← memory is automatically managed
↳ allocated memory as program starts and deallocated as program ends

* Static vs dynamic allocation:-
* Static → size and type of memory is fixed and known at compile time
Allocating the data, same time as declaration and not storing any pathway (address where actual data is stored)

Stack memory used ← `int a = 10;` // static allocation ⟶ compile time

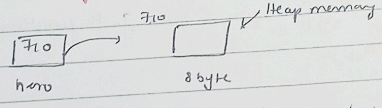
`int * i = new int;` // dynamic allocation → runtime

variable size, lifetime → memory avoids



↳ Address 710 of heap memory contains data and pointer i pointing towards address 710
Heap address (dynamically allocated)
↓
new keyword use
↳ to allocate or deallocate → depends on program
↓
potential memory leaks or overhead issues

Same we can do for classes - `Hero * h = new Hero;`



`h → set value (70);` `h → sethealth (100, 'A');`

↦ dereference operator

For output
```
cout << (*h).level << endl;
cout << (*h).gethealth() << endl;
```
↳ dereference operator

or
```
cout << h → gethealth() << endl;
cout << h → level << endl;
```

↳ for dynamic allocation ⟶ size can be determined during execution of program providing flexibility

Constructor → original class remains automatically invoked during declaration

* Concept of Constructor and Destructor -
(Invoked automatically at time of object creation)
whenever we create an object, 1st thing that is been called is "constructor"

For eg Hero ramesh,

Behind the scene a default Constructor without any parameter
ramesh. Hero() been made

- Invoked during object creation time
- No return type
- No i/p parameter (for case of default Constructor)
- Same name as the class and may be defined inside or outside class

↳ When we try to write our own Constructor, the Constructor that had already been created by system will be replaced by this

↳ no argument, no parameters
→ Constructor can be overloaded

class Hero {

// constructor (default with no parameters) created

Hero()
{
cout << "constructor made" << endl;
}
}

default Constructor →

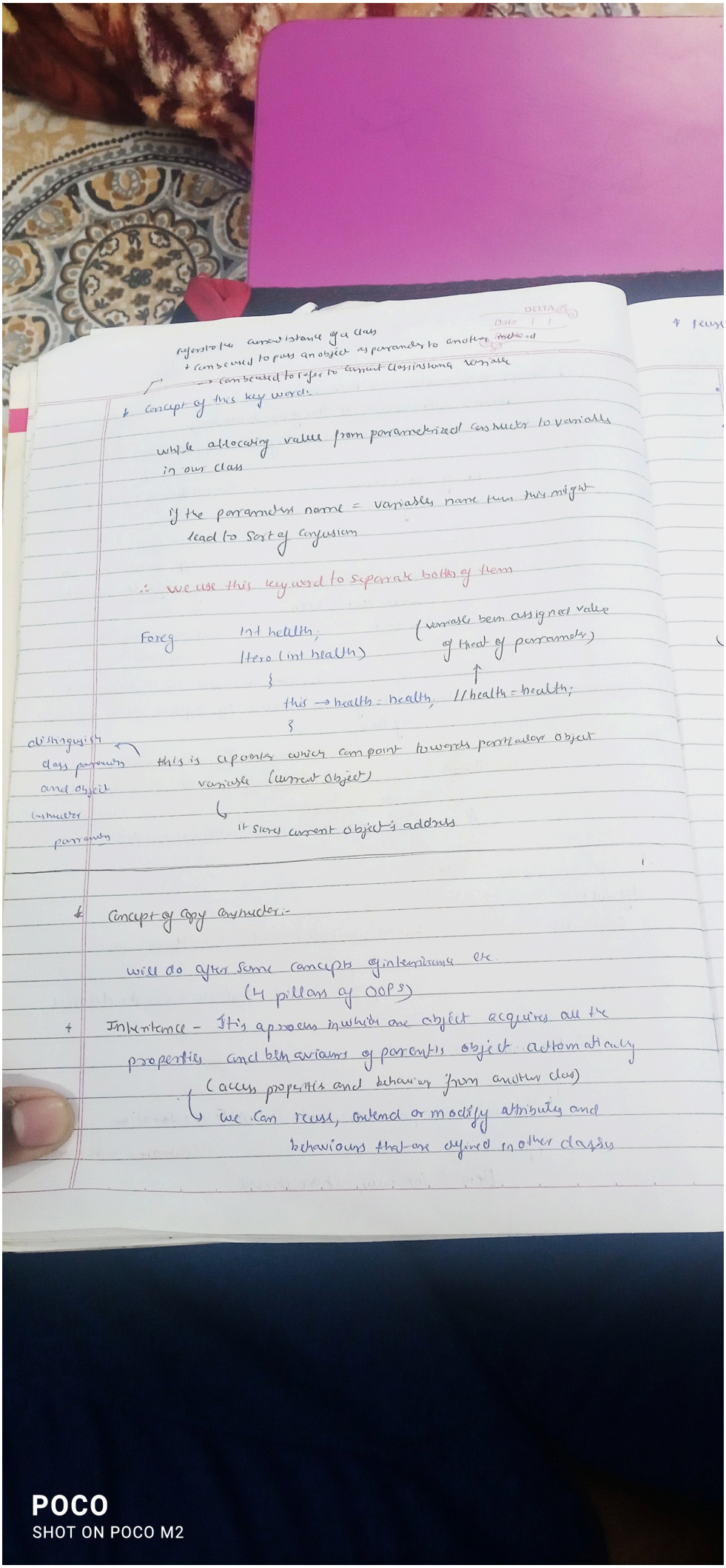
* No virtual constructor can be declared

→ Parameterized Constructor:-

↳ Constructor having parameters with it while declaration

↳ Hero (int value, char check)

Constructors with parameters



* reuse, modify the attributes

- Parent object → Base class
- child object → derived class

for their usage we need our access modifier to be public.

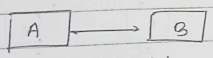
C++ Syntax:

class derived_class :: access modifier base_class
(sub class) (parent class)

↳ class Dog :: public Pet (For eg)

(i) Single inheritance:- when 1 class inherits properties from 1 base class only

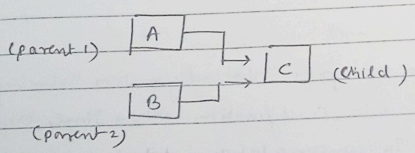
(Derived class = Base class properties + own properties)



(Base class) (derived class)

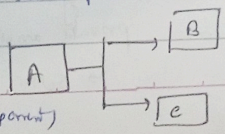
(ii) multiple inheritance - deriving a new class that inherits properties from 2 or more base class

class Pet : public Dog, public Cat



(parent 1) (parent 2) (child)

(iii) Hierarchical inheritance - A type of inheritance where a base class can produce 2 or more child classes (child classes can inherit properties from same parent)

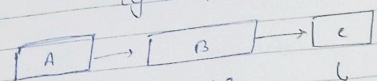


(parent) Child 1 Child 2

* Ambiguity resolution → In case of multiple inheritance, if 2 classes have same member functions (eg. void display() in both B and C)
↳ ambiguity occurs → usage of scope resolution operator ::

(iv) Multilevel inheritance — multiple inheritance is a process of deriving a class from another derived class (parent, new — parent — child concept)

C.A::display();
C.B::display();

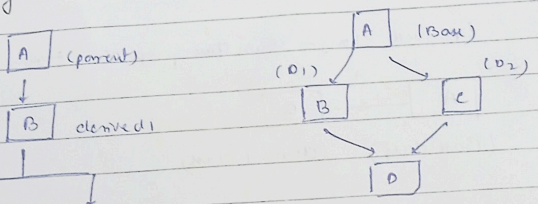


parent 1 — parent 2 / child — child

(has prop. of A) (has properties of 1 and 2)

(v) Hybrid inheritance — This a combination of various types of inheritances → simple, multiple and hierarchical

(Single + ~~multiple~~ hierarchical) or (multiple + hierarchy) (multipath inheritance)



↳ Diamond inheritance

* Encapsulation-

↗ wrapping up the data and information into single unit

process of combining data and functions into a single unit, called class. In encapsulation, data is not accessed directly → accessed through functions present inside the class

capsule

↓

attributes of class are kept private and public

↳ setter and getter methods

DELTA
Date / /
Pg No

data binding — process of binding the application UI and business logic → any changes in business logic will reflect in application UI

binding the data property and functions that manipulate data

encapsulation → responsible for hiding the data

restricting members to access objects

Securely used, reduces human error

→ only high level access, no expose reveal of below level

+ Abstraction — providing abstract detail views, model or structure of a real life problem to reduce unnecessary details

providing necessary information regarding the data only and hiding unnecessary details like (background details and informations)

+ Polymorphism — It is the ability to present the same interface for differing underlying forms (data types)

every class has different underlying data

for eg → a shape { can be a square or a circle

(many forms)

Types of polymorphism

- Compile time (Static) → implemented at compile time (eg — method overloading) function call resolved at compile time
- Runtime (Dynamic) → also called as dynamic polymorphism

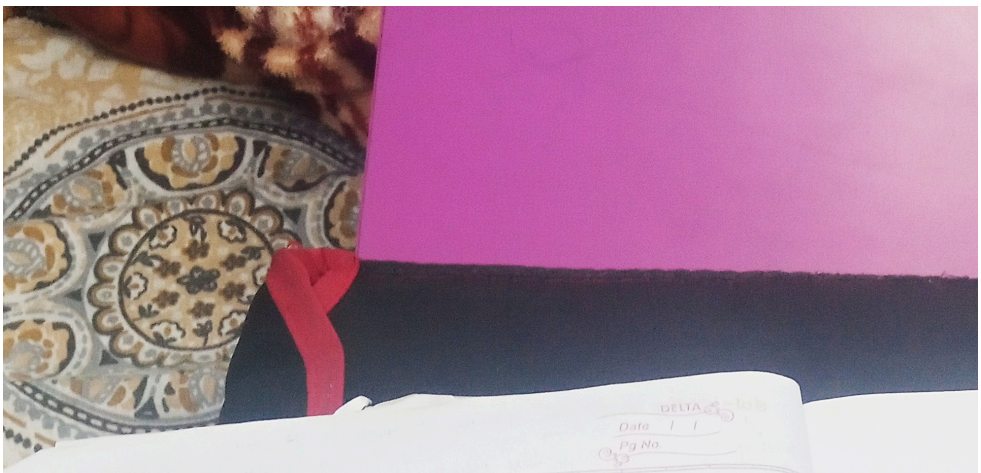
(Runtime polymorphism)

→ for eg Function overriding

↳ when the child class contains method that's already present in parent class (child class overtakes (overrides) parent class)

↳ parent and child class with same function name with different definition

call of function is at runtime



* method overloading - method overloading is a technique which allows us to have more than 1 function with same function name but different functionality

↳ how can it vary?? → return type of function
→ type of params passed to fun
→ no. of params passed to fun

```
int add (int x, int y)
{
    return (x+y);
}

int add (int x, int y, int z)
{
    return (x+y+z);
}
```

Concept of copy constructor -

A copy constructor is an overloaded constructor used to declare and initialize an object from another object.

- default copy constructor
- user defined copy constructor

↓

it is a member function which initializes an object using another object of same class

C++ Sample Code :-

```
#include <iostream>
using namespace std;
class go {
```

DELTA
Date / /
Pg No.

↗ Sample code

```
public:
    int x;

go(int a)
  {
     // parametrized
     x=a;
  }
go( go &i)
  {
     // copy constructor
     x=i.x;
  }
};
int main()
 {
   go a1 (20);
   go a2(a1);
   cout << a2.x <<endl;
 }
```

* Destructor – works opposite to constructor, destructs object of classes
  ↳ Can be defined only once in class

(an object when created ↳ constructor and destructor both created)

~ = tilde sign for destructor
  ↳ having same name as constructor

```
~A()
 {
 cout<< "Destructor in use" <<endl;
 }
```

→ to access private members of other class acting as its friend

* Friend function – Acting as a friend of the class. It can access the private and protected members of the class
  ↳ not a member of the class but must be listed in class defination
    ↳ Can access and manipulate the data

friend function – has the ability to access private data of class

→ friend function cannot access private members directly
↳ how to use an object name and dot operator with each member name
↳ uses objects as arguments

eg friend int mul(A, k)
{
return (k.a * k.b);
}

⇒ Aggregation ↳ process in which one class defines another class as entire reference
↳ another way to reuse the class

~ Virtual function in C++ ⇒

It's a member function present in base class and redefined by the derived class
↳ when we use the same function name in both base and derived class, function in base class is declared by keyword virtual

↳ Virtual functions cannot be static
→ It may have virtual destructor but cannot have virtual constructor.

DELTA
Date / /
Pg No.

// virtual function (re-defined in derived class)

```
virtual void print()
}
cout << " print base class" << endl;
{

bptr → print();   // virtual function velley printing
```

Last topics

→ Abstract function

** * Pure Virtual function - It is not used for performing any task
↳ only serves as a place holder

we can have implementation but we must try to override it

→ A function declared in base class having no defination relative to base class

↳ A class containing pure virtual function cannot be used to declare objects on its own

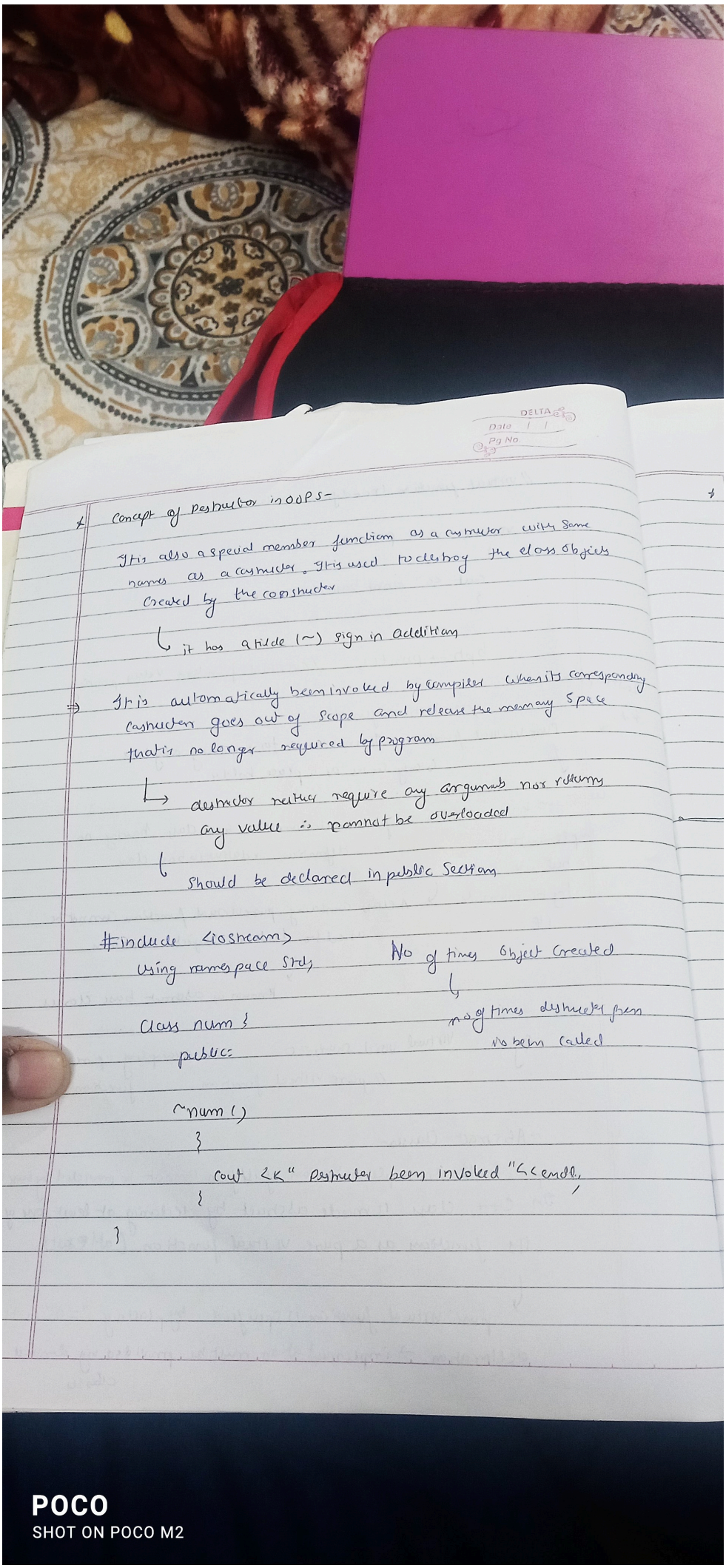
↳ Known as abstract base classes

eg - virtual void show() = 0 → example of pure virtual function
// pure virtual function

~Abstract Classes~

→ Sometimes implementation of all fun cannot be provided by base class
In C++, class is made abstract by declaring at least one of its functions as a pure virtual function (atleast)

↳ pure virtual function is specified by placing "= 0" in it's declaration. → implementation must be provided by derived classes



Concept of Destructor in OOPS-

It is also a special member function as a constructor with same name as a constructor. It is used to destroy the class objects created by the constructor

↳ it has a tilde (~) sign in addition

⇒ It is automatically been invoked by compiler when its corresponding constructor goes out of scope and release the memory space that is no longer required by program

↳ destructor neither require any arguments nor returning any value ∴ cannot be overloaded

↳ should be declared in public section

```
#include <iostream>
using namespace std;

class num {
  public:

  ~num()
  {
    cout << "destructor been invoked" << endl;
  }
};
```

No of times Object created
↳ no of times destructor been invoked

DELTA
Date / /
Pg No.

* Abstraction continued -

- Helps user to avoid writing low level code
- Avoids code duplication and increases reusability
- Helps to increase security of an application or program as only required details are provided to the user.

For eg- Bank → we just need to put details no use of internal complex functionality

// example code :-

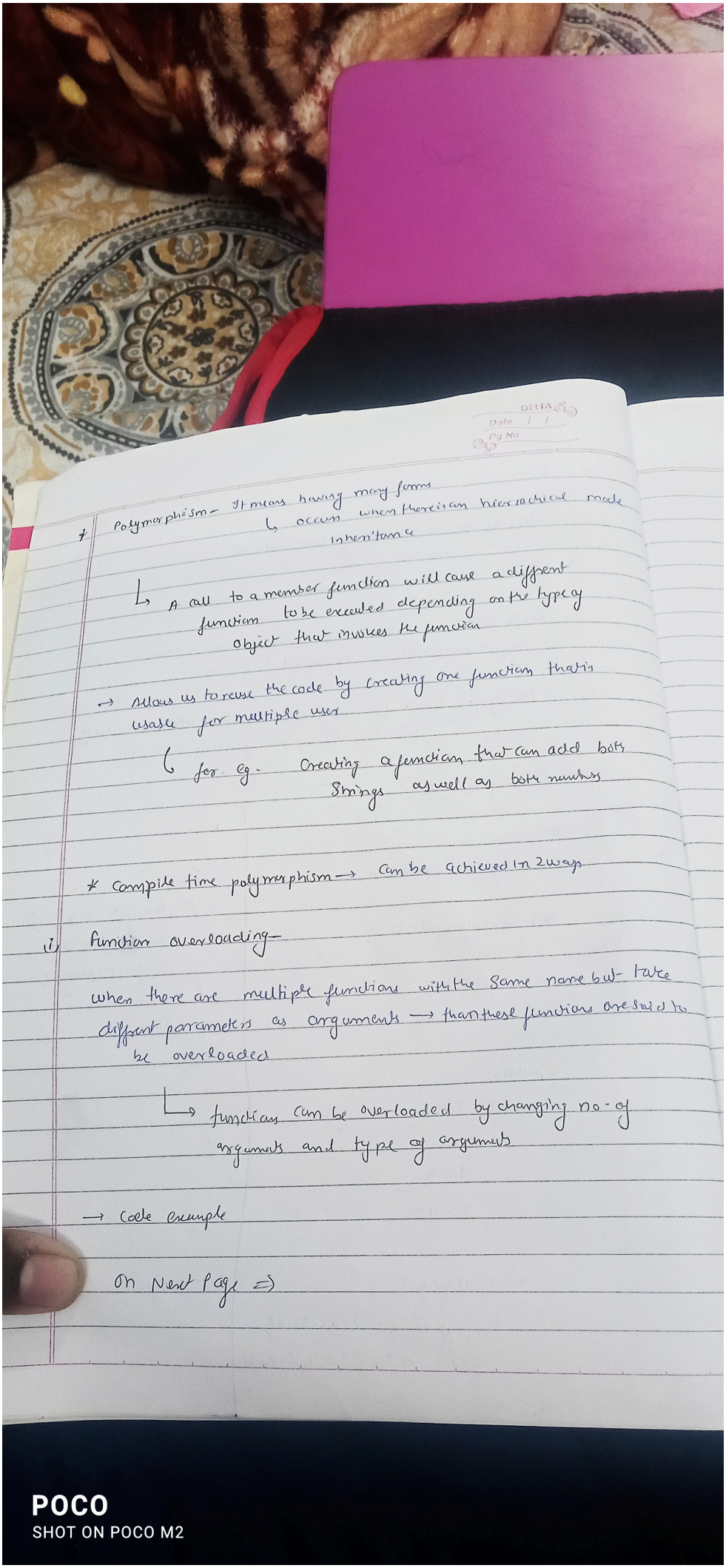
```
#include <iostream>
using namespace std;

class implementabs
{
    private:
    int a, b;

    public:
    void set (int x, int y)
    {
        a = x;
        b = y;
    }

    void display ()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};
```

```
int main()
{
    implementabs obj;
    obj.set (10, 20);
    obj.display ();
}
```



* Polymorphism - It means having many forms
  ↳ occurs when there is an hierarchical made in inheritance

↳ A call to a member function will cause a different function to be executed depending on the type of object that invokes the function

→ Allows us to reuse the code by creating one function that is usable for multiple user

↳ for eg - Creating a function that can add both Strings as well as both numbers

* Compile time polymorphism → can be achieved in 2 ways

(i) Function overloading -

When there are multiple functions with the same name but take different parameters as arguments → then these functions are said to be overloaded

↳ functions can be overloaded by changing no. of arguments and type of arguments

→ Code example

On Next Page =>

```
#include <iostream>
using namespace std;
class Check
{
    public:
    void func()
        {
        cout << "No argument" << endl;
        }

    void func(int x)
        {
        cout << "function with one argument" << endl;
        }

    void func(int x, int y)
        {
        cout << "function with 2 arguments" << endl;
        }
};
int main()
{
    Check obj;
    obj.fun();
    obj.fun(4);
    obj.fun(4,7);
}
```

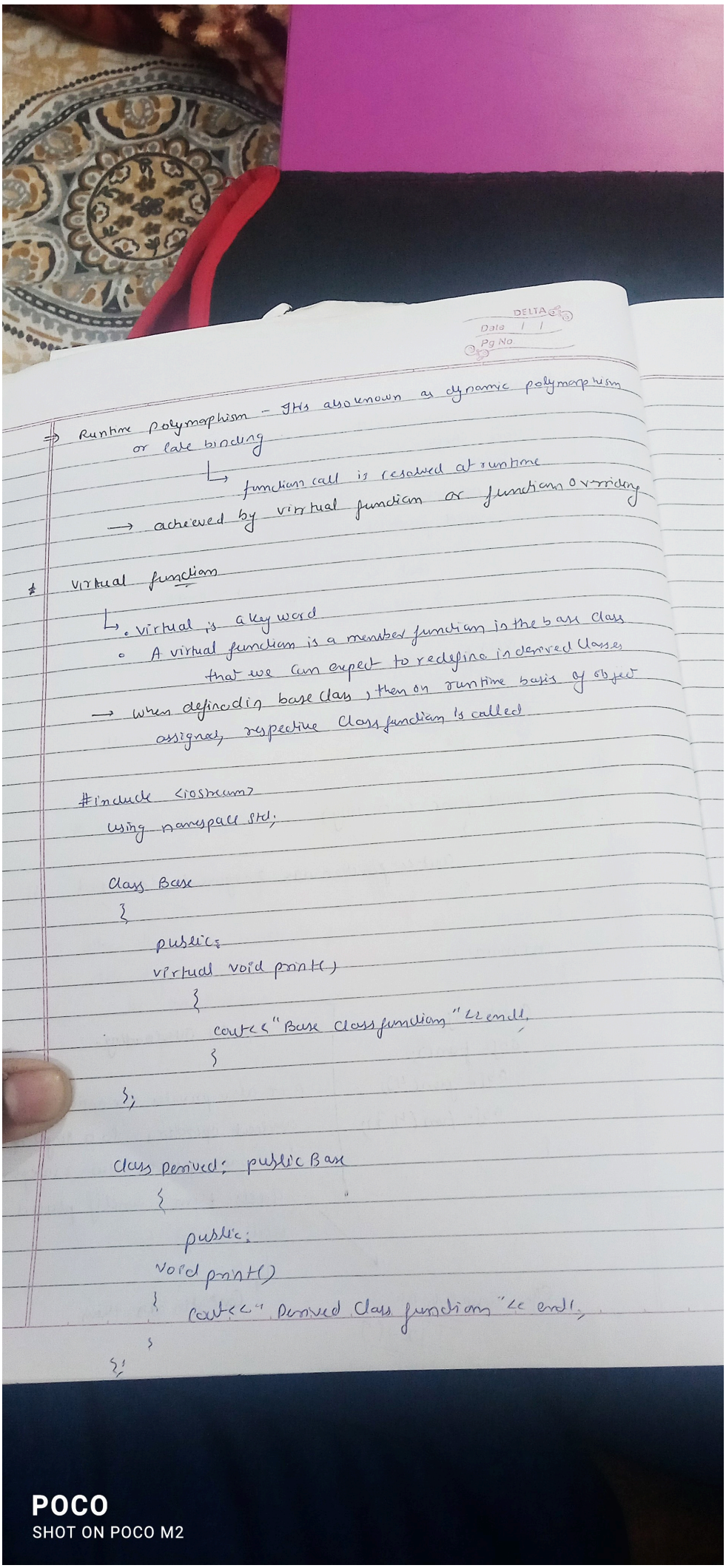
2. Operator Overloading.

C++ also provides the option to overload operators so a single operator '+' when placed b/w 2 integers adds them and if placed b/w strings

↳ Concatinates them.

Eg. Complex no.

12 + i9



⇒ Runtime polymorphism - It is also known as dynamic polymorphism or late binding
↳ function call is resolved at runtime
→ acheived by virtual function or function overriding

* Virtual function
↳ • virtual is a key word
• A virtual function is a member function in the base class that we can expect to redefine in derived classes
→ when defining base class, then on runtime basis of object assigned, respective class function is called

```
#include <iostream>
using namespace std;

class Base
{
    public:
    virtual void print()
        {
        cout<< "Base class function" << endl;
        }
};

class Derived: public Base
    {
        public:
    void print()
    {
        cout<< "Derived class function" << endl;
    }
};
```

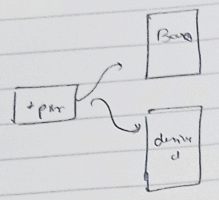
```
int main()
{
    // creating a base class pointer to point individual classes

    Base *bptr;

    Base base;
    Derived derived;

    bptr = &base;
    bptr -> print();   // prints Base class

    bptr = &derived;
    bptr -> print();   // prints derived class
}
```



→ Friend function implementation → Look at github for codes