C:\vs code\GitHub\Basic-roadmap\10 - OOP as it Should Be (Concepts)\courses\res.cpp

```cpp

#include<iostream>
using namespace std;
// class  : ( blue print of object  )

class className{

 // attributes :

 // functions :

};
// accessibility
/*
✨ access specifiers(modifiers)
   public :   visible for  all program
   private :  visible into the class
   protected : visible into the  parent class and derived class (child)
*/

/*

 data members : any variable declared inside the class
  members  methods(functions) :  any functions or procedure
  declared inside the class
 class members = data + methods

*/

// properties : get/set
/*
 one of the most  useful functions that allow us access (read | update )
 the  private members into a class

   ✨get : read
   ✨set : update
*/
// example :
class clsPerson
{

protected:
    int v1 = 5;
    int f1()
    {
        return 1;
    }

private:
    string cFirstName;
    string cLastName;

public:
```

```
    string getFullName()
    {
        return cFirstName + " " + cLastName;
    }

    //   getters :
    string getFirstName() { return cFirstName; }
    string getLastName() { return cLastName; }

    // setters :
    void setFirstName(string firstName) { cFirstName = firstName; }
    void setLastName(string lastName) { cLastName = lastName; }
};

// properties get and set through '=' (just for microsoft environment )
class className{
__declspec(property(get=getFunction,put=setFunction)) datatype varNameToShowUser;
};
// ✨Encapsulation
/*
In normal terms Encapsulation is defined as wrapping up of data and
information under a single unit.
In Object Oriented Programming, Encapsulation is defined as binding
together the data and the functions that manipulates them
*/

// ✨Abstraction
/*
In simple terms, abstraction “displays” only the relevant attributes
of objects and “hides” the unnecessary details.
*/

// ✨ constructor  :
class className{
 className(){
   // code

 }
};
   // A constructor is a special type of member function that is called
   //automatically when an object is created
   /* types
     empty : no parameters
     parametrized  with parameters
     copy  : used to initialize the members of a newly created object
      by copying the members of an already existing object.
   */

// ✨ destructor :
class className{
 ~className(){
   // code

 }
};
/*
```

```
  Destructor is an instance member function which is invoked automatically
   whenever an object is going
  to be destroyed. Meaning, a destructor is the last function that is going
  to be called before an object is destroyed.
*/
//example :
class cRectangle
{
private:
    string firstName;
    string lastName;

public:
    // empty constructor :
    cRectangle()
    {
        firstName = "";
        lastName = "";
    }

    // parametrized constructor :
    cRectangle(string firstName, string lastName)
    {
        this->firstName = firstName;
        this->lastName = lastName;
    }

    // copy constructor  :
    cRectangle(cRectangle &copy)
    {
        firstName = copy.firstName;
        lastName = copy.lastName;
    }

    string getFirstName()
    {
        return firstName;
    }
    string getLastName()
    {
        return lastName;
    }

    // destructor :
    ~cRectangle()
    {

        cout << "good night : " << firstName << endl;
    }
};


// ✨static members :
  class className{
    static varName;

```

```
  };
// initialize a static variable :
  type className varName =value;
   /*
   Static Member is a variable that is shared for all objects, any object modifies it
   it get modified for all other objects.
   */

//example :
class cA
{
private:
    int var;
    static int counter;

public:
    cA()
    {
        var = 0;
        counter++;
    }

    void print()
    {
        cout << "\n var = " << var << "\n";
        cout << "counter = " << counter << "\n";
    }
};

int cA::counter = 0;


//✨ static functions :
  class className{
    static functionName(){
        //code
    }

  };
/*
  ✔Static function  is a function  that is shared for all objects
  ✔Static Functions can be called at class level without a need to have an object.
  ✔No, Static methods can only access static members , because static methods can be called at
    class level without objects, and non static members you cannot access them without having
    object first.
*/

// access to  a static function  :
int main(){
 className::functionName();
}

// example :
class cA
{
```

```
private:
    static int counter;

public:
    cA()
    {
        counter++;
    }
    static int getCounter()
    {
        return counter;
    }
};


//✨  Inheritance :
/*
     Inheritance: Inheritance is one in which a new class is created that
     inherits the properties
     of the already exist class. It supports the concept of code
     reusability and reduces the length
     of the code in object-oriented programming.
*/
// base class / super class / parent class
class baseClass{

};
// sub class / derived class / child class
class derivedClass : modifiers className{

};

// access to  function  from the base class :
class baseClass {

 public :
 void functionExample(){
    // code
 }
};
class derivedClass : public baseClass {
    void functionExample(){
 baseClass::functionExample();
   // added code
    }
};

//example
class cPerson
{

    int id;
    string firstName;
    string lastName;
    string email;
    string phone;
```

```
277
278 public:
279     // empty constructor :
280     cPerson()
281     {
282         id = 0;
283         firstName = "";
284         lastName = "";
285         email = "";
286         phone = "";
287     }
288     // parametrized constructor :
289     cPerson(int id, string firstName, string lastName, string email, string phone)
290     {
291
292         this->id = id;
293         this->firstName = firstName;
294         this->lastName = lastName;
295         this->email = email;
296         this->phone = phone;
297     }
298
299     // print function :
300     void print(bool isBaseClass = true)
301     {
302
303         cout << "\n_____\n";
304         cout << "the id        : " << id << "\n";
305         cout << "the firstName : " << firstName << "\n";
306         cout << "the lastName  : " << lastName << "\n";
307         cout << "the email     : " << email << "\n";
308         cout << "the phone     : " << phone << endl;
309         if (isBaseClass)
310             cout << "_____\n";
311     }
312
313     void sendEmail(string subject, string body)
314     {
315         cout << "\nThe following message sent successfully to email:" << email << "\n";
316         cout << "subject : " << subject << "\n";
317         cout << "boyd : " << body << "\n";
318     }
319
320     void sendSms(string sms)
321     {
322         cout << "\nThe following SMS sent successfully to phone:" << phone << "\n";
323         cout << sms << "\n";
324     }
325
326     int getId() { return id; }
327     string getFirstName() { return firstName; }
328     string getLastName() { return lastName; }
329     string getEmail() { return email; }
330     string getPhone() { return phone; }
331
332     // setters :
```

```
    void setFirstName(string firstName) { this->firstName = firstName; }
    void setLastName(string lastName) { this->lastName = lastName; }
    void setEmail(string email) { this->email = email; }
    void setPhone(string phone) { this->phone = phone; }
};

class cEmployee : public cPerson
{
    string title;
    string department;
    float salary;

public:
    void print(bool isBaseClass = true)
    {
        cPerson::print(false);
        cout << "the title      : " << title << "\n";
        cout << "the department : " << department << "\n";
        cout << "the salary     : " << salary << "\n";
        if (isBaseClass)
            cout << "_____________________________________\n";
    }

    // setters :
    void setTitle(string title) { this->title = title; }
    void setDepartment(string department) { this->department = department; }
    void setSalary(float salary) { this->salary = salary; }

    // getters :
    string getTitle() { return title; }
    string getDepartment() { return department; }
    float getSalary() { return salary; }

    cEmployee(int id, string firstName, string lastName, string email, string phone, string
title, string department, float salary)
        : cPerson(id, firstName, lastName, email, phone)
    {

        this->title = title;
        this->department = department;
        this->salary = salary;
    }
};

// multi level inheritance :  class1 inherited from class2 and class2 inherited from class1
...

// -- inheritance visibility modes

 public :  public  keep public , protected keep protected
 private :   -- public && protected will be private (you can access them within the base and
the derivedClass )
 protected :  -- public && protected will be protected (you can access them within the base
and derivedClass and all nextLevelDerivedClass)

//  type of inheritance :

```

```
// single   : class inherit one class
// multi-level : class inherit a class and the class inherited by another one ..
// hierarchal  :  one class inherited by multiple classes

// -----special type --------
// multiple : one class inherit multiple classes  (not recommended supported by cpp )

// -----special type --------
// hybrid : one class inherit multiple classes that also inherit  a class    (not recommended supported by cpp )


// up casting vs down casting

// up casting : convert from a derived class to base class  (using pointers )

// down casting : convert from a base class  to  a derived class  (you can't convert it )

// example :
class cPerson
{
public:
    string name = "ayoub";
};

class cEmployee : public cPerson
{
public:
    string title = "nice";
};
int main()
{

    cEmployee e1;
    // up casting :
    cPerson *p1 = &e1;

    cout << p1->name << endl;

    cPerson p2;
    // down casting :
    cEmployee *e2 = &p2;
    cout << e2->name << endl;

    return 0;
}

In C++, the virtual keyword is used to declare a member function in a base class
that can be overridden by a function with the same signature in a derived class.
 This concept is a fundamental aspect of polymorphism in object-oriented programming.
  Here are some key points about the base usage of virtual:

Polymorphism:
Virtual functions enable polymorphism, allowing different objects to be treated
 as instances of a common base class.
Polymorphism allows you to write code that can work with objects of
```

different derived classes through a common interface.
Function Overriding:

When a **function** is declared **as virtual in** a base **class**, it can be overridden **in** derived classes.
Function overriding allows derived classes **to** provide their own implementation **of** the **virtual function**.
Late Binding (Dynamic Binding):

The decision **of** which **function to** call is made at runtime rather than compile time.
This is achieved through the use **of** a **virtual function** table (vtable) **or** similar mechanism, which maintains a mapping
of virtual functions **to** their actual implementations **in** derived classes.
Base Class Pointers **and** Derived Class Objects:

Virtual functions are particularly useful **when** dealing **with** base **class** pointers pointing **to** objects **of** derived classes.
When a **virtual function** is called through a base **class** pointer,
the appropriate version **of** the **function in** the derived **class** is invoked.

```
// example :
class cPerson
{
public:
    string name = "ayoub";

    virtual void print()
    {
        cout << "HI, i'm person \n";
    }
};

class cEmployee : public cPerson
{
public:
    void print() override
    {
        cout << "HI, i'm an employee \n";
    }
};
class cStudent : public cPerson
{
public:
    void print() override
    {
        cout << "HI, i'm a student  \n";
    }
};
int main()
{

    cEmployee e1;
    cStudent s1;

    cPerson *p1 = &e1;
    cPerson *p2 = &s1;
```

```
    p1->print();
    p2->print();

    return 0;
}


/*
static/Early binding

      vs

dynamic/late binding

*/

// static/Early binding
Static Binding: The binding which can be resolved at compile time by the
compiler is known as static or early binding.


//✨ polymorphism :
/*
Polymorphism is one of the important features/principles/concepts of OOP,
 word Ploy means "Many" and word
Morphism means "Form" so it means "Many Forms", the ability to take more than one form.
*/
// examples :
/*
1- function overloading
2- function overwriting
3- operator overloading
4- virtual functions
*/
In C++, an abstract class is a class that cannot be instantiated on its
own and is meant to serve as
a base class for other classes. It may contain abstract methods, which are declared
but not defined
in the abstract class. The derived classes must provide concrete implementations for these
abstract
 methods. Abstract classes are used to define an interface or a common set of features that
derived classes must implement.

Here are the key features and concepts related to abstract classes in C++:

1. **Abstract Class Declaration:**
   - An abstract class is declared using the `class` keyword, along with the `virtual`
   keyword for abstract methods.
   - It may contain both concrete (implemented) and abstract (unimplemented) methods.
   - Abstract methods are declared with the `virtual` keyword and are followed by `= 0`
   to indicate that they have no implementation in the abstract class.

   ```cpp
   class AbstractClass {
   public:
       // Concrete method
       void concreteMethod() {
```
```

```
            // Implementation
        }

        // Abstract method
        virtual void abstractMethod() = 0;
    };
    ```

2. **Cannot be Instantiated:**
    - Objects of an abstract class cannot be created directly. It is meant to be used as a blueprint for other classes.

    ```cpp
    // Cannot do this - results in a compilation error
    // AbstractClass obj;
    ```

3. **Derived Classes Implementation:**
    - Any class that inherits from an abstract class must provide concrete implementations for all the pure virtual (abstract) methods declared in the abstract class.

    ```cpp
    class DerivedClass : public AbstractClass {
    public:
        // Concrete implementation for the abstract method
        void abstractMethod() override {
            // Implementation
        }
    };
    ```

4. **Abstract Class as Interface:**
    - Abstract classes are often used to define interfaces, where the derived classes provide specific implementations for the methods declared in the interface.

    ```cpp
    class Interface {
    public:
        virtual void method1() = 0;
        virtual void method2() = 0;
    };

    class ConcreteClass : public Interface {
    public:
        void method1() override {
            // Implementation for method1
        }

        void method2() override {
            // Implementation for method2
        }
    };
    ```

5. **Destructor in Abstract Class:**
```
```

```
   - An abstract class can have a virtual destructor, and it's a good practice to provide
   a virtual destructor to ensure proper cleanup when objects of derived classes are deleted.

   ```cpp
   class AbstractClass {
   public:
       virtual ~AbstractClass() {}
   };
   ```

Abstract classes provide a way to achieve abstraction and polymorphism in C++ by
 defining a common
interface that derived classes must adhere to. They are an essential part of
object-oriented programming
 and are widely used in designing class hierarchies.

 // INFO :
 An abstract class in C++ has at least one pure virtual function
 by definition. In other words, a function that has no definition.

The C++ interfaces are implemented using abstract classes and these abstract classes
should not be confused with data abstraction which is a concept of keeping implementation
 details separate from associated data.

 // friend class :
 A friend class can access both private and protected members of the class
 in which it has been declared as friend.

 class className{

 //
friend className2;

 };
 class className2{

 };

 // friend function :
 A friend function in C++ is a function that is declared outside a class but is capable of
 accessing the private and protected members of the class. There could be situations
 in programming wherein we want two classes to share their members. These members may be
 data members, class functions or function templates. In such cases, we make the desired
 function, a friend to both these classes which will allow accessing private and
 protected data of members of the class.

 class className{

friend datatype functionName(arg);

 };
 datatype functionName(arg){

    //code
 }
```

```

 // using strcut with  classes :
class className{

 struct structName{
 att1;
 att2;
 };
public :
structName ob1;

};

// example :
#include <iostream>
#include "./input.h"

using namespace std;

class cPerson
{

private:
    struct stAddress
    {
        string city;
        string street;
    };

    string fullName;
    stAddress add;

public:
    friend istream &operator>>(istream &inp, cPerson &person);

    void printINfo()
    {
        cout << "\n-------- Person info -----------\n";
        cout << "the full name : " << fullName << "\n";
        cout << "the city : " << add.city << "\n";
        cout << "the street  : " << add.street << endl;
        cout << "\n-------- --=---- -----------";
    }
};

//  output stream operator :
istream &operator>>(istream &inp, cPerson &person)
{

    person.fullName = input::readString("enter the  full name : ");
    person.add.city = input::readString("enter the  city  : ");
    person.add.street = input::readString("enter the street  : ");

    return inp;
}
```

```
int main()
{
    cPerson p1;

    cout << "the person info :\n";

    cin >> p1;
    p1.printINfo();
    return 0;
}

// nested classes :
    Nested or Inner Classes : A class can also contain another class definition
    inside itself, which is called “Inner Class” in C++.

// enclosing /containing  class    :
class className{

 // inner /nested class :
// code
 class className{
 // code
};
// code
};
// example :
#include <iostream>
using namespace std;

class person
{

protected:
  class c
  {

    string name;
    string lastName;

  public:
    void print()
    {
      cout << "the name : " << name << "\n";
      cout << "the last name  : " << lastName << "\n";
    }
  };

public:
  c a;
  string km;
};

class e : public person
{

  c e2;
```

```
};

int main()
{

  e p1;
  p1.a.print();

  return 0;
}

// separate class in library :
Separating Code and Classes in Libraries will make our life easier
and we can control our code and organize it better.

We must user "#pragma once" in each header file to prevent the complier
from loading the library more than one time and have repeated code included.

/*
1- create new file  header file with extension .h
2- include the  included it in your main file :
3- add #pragma once  to the header file to included one time
*/

// example cEmployee.h
#pragma once
#include<iostream>
using namespace std ;
#include "cPerson.h"
class cEmployee : public cPerson
{
    string title;
    string department;
    float salary;

public:
    void print(bool isBaseClass = true)
    {
        cPerson::print(false);
        cout << "the title      : " << title << "\n";
        cout << "the department : " << department << "\n";
        cout << "the salary     : " << salary << "\n";
        if (isBaseClass)
            cout << "_____________________________________\n";
    }

    // setters :
    void setTitle(string title) { this->title = title; }
    void setDepartment(string department) { this->department = department; }
    void setSalary(float salary) { this->salary = salary; }

    // getters :
    string getTitle() { return title; }
    string getDepartment() { return department; }
    float getSalary() { return salary; }

```

```
    cEmployee(int id, string firstName, string lastName, string email, string phone, string title, string department, float salary)
        : cPerson(id, firstName, lastName, email, phone)
    {

        this->title = title;
        this->department = department;
        this->salary = salary;
    }
};


//✨ objects with vectors :
int main(){

vector <clsA> v1;
short NumberOfobjects=5;

// inserting object at the end of vector
for (int i = 0; i < NumberOfObjects; i++)
v1.push_back(clsA(i));

//✨ printing object content
for (int i = 0; i < NumberOfObjects; i++)
v1[i].Print();

return 0;
}
//Objects and Dynamic Array
int main(){

short NumberOfobjects = 5;

//✨ allocating dynamic array
// of Size NumberOfObjects using new keyword

clsA * arrA = new clsA[NumberOfobjects];

// calling constructor
// for each index of array
for (int i = 0; i < NumberOfObjects; i++)
arrA[i] = clsA(i);

// printing contents of array
for (int i = 0; i < NumberOfObjects; i++)
arrA[i]. Print();

return 0;
}

//✨ Objects with Parameterized Constructor and Array
int main(){

// Initializing 3 array Objects with function calls of
// parameterized constructor as elements of that array
clsA obj[] = { clsA(10), clsA(20), clsA(30) };
```

```
886
887 // using print method for each of three elements.
888 for (int i = 0; i < 3; i++)
889 obj[i]. Print();
890
891 return 0;
892 }
```