res\res_OOP.cpp

```cpp

#include<iostream>
using namespace std;
// class : ------------------------------------[-]
/*
    Classes in C++ are a fundamental feature of object-oriented programming (OOP). They provide a way to define custom data types that encapsulate
    data and functions into a single entity. Here's a comprehensive overview of classes in C++
*/
    // basic syntax
        class className{
    [modifier:]
        // attributes : (data members)
        datatype varName;

    [modifier:]
        // functions :  (methods)
        returnType functionName(args){
            //code
        }
        };

    // accessibility : ------------------------------------[-
    /*
    ✨ access specifiers(modifiers)
        public :   visible for  all program
        private :  visible into the class
        protected : visible into the  parent class and derived class (child)   (we will talk about derived class later on )
    */
    // elements : ------------------------------------[-
    /*
        -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  (getters)
   ✨set : update  (setters)
*/
    //* 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{
/*
    A constructor is a special type of member function that is called
    automatically when an object is created
*/
    //syntax :
        className(){
        // code

        }
        };
    // types of constructor :
    /*
```

```
            1-empty : no parameters
            2-parametrized  with parameters
            3-copy  : used to initialize the members of a newly created object by copying the members of an already existing object.
    */

// ✨ destructor :  ------------------------------------[-
/*
    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.
*/
    //syntax :
        class className{
        ~className(){
        // code

        }
        };

    //* 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 : ------------------------------------[-]
/*
   Static Member is a variable that is shared for all objects, any object modifies it
   it get modified for all other objects.
*/
    // syntax :
        class className{
            static varName;

        };
    // initialize a static variable :
        type className::varName =value;

    //* 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 :  ------------------------------------[-
 /*
    ✔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.
*/
    // syntax :
```

```
        class className{
            static functionName(){
                //code
            }

        };

    // 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;

        public:
            // empty constructor :
                cPerson()
                {
                    id = 0;
                    firstName = "";
                    lastName = "";
                    email = "";
                    phone = "";
                }
            // parametrized constructor :
                cPerson(int id, string firstName, string lastName, string email, string phone)
                {

                    this->id = id;
                    this->firstName = firstName;
                    this->lastName = lastName;
                    this->email = email;
                    this->phone = phone;
                }

            // print function :
                void print(bool isBaseClass = true)
                {

                    cout << "\n_____\n";
                    cout << "the id        : " << id << "\n";
                    cout << "the firstName : " << firstName << "\n";
                    cout << "the lastName  : " << lastName << "\n";
                    cout << "the email     : " << email << "\n";
                    cout << "the phone     : " << phone << endl;
                    if (isBaseClass)
                        cout << "_____\n";
                }

            void sendEmail(string subject, string body)
            {
                cout << "\nThe following message sent successfully to email:" << email << "\n";
                cout << "subject : " << subject << "\n";
```

```
328                 cout << "boyd : " << body << "\n";
329             }
330 
331             void sendSms(string sms)
332             {
333                 cout << "\nThe following SMS sent successfully to phone:" << phone << "\n";
334                 cout << sms << "\n";
335             }
336 
337             int getId() { return id; }
338             string getFirstName() { return firstName; }
339             string getLastName() { return lastName; }
340             string getEmail() { return email; }
341             string getPhone() { return phone; }
342 
343             // setters :
344             void setFirstName(string firstName) { this->firstName = firstName; }
345             void setLastName(string lastName) { this->lastName = lastName; }
346             void setEmail(string email) { this->email = email; }
347             void setPhone(string phone) { this->phone = phone; }
348         };
349 
350         class cEmployee : public cPerson
351         {
352             string title;
353             string department;
354             float salary;
355 
356         public:
357             void print(bool isBaseClass = true)
358             {
359                 cPerson::print(false);
360                 cout << "the title      : " << title << "\n";
361                 cout << "the department : " << department << "\n";
362                 cout << "the salary     : " << salary << "\n";
363                 if (isBaseClass)
364                     cout << "_____________________________________\n";
365             }
366 
367             // setters :
368                 void setTitle(string title) { this->title = title; }
369                 void setDepartment(string department) { this->department = department; }
370                 void setSalary(float salary) { this->salary = salary; }
371 
372             // getters :
373                 string getTitle() { return title; }
374                 string getDepartment() { return department; }
375                 float getSalary() { return salary; }
376 
377             cEmployee(int id, string firstName, string lastName, string email, string phone,
    string title, string department, float salary)
378                 : cPerson(id, firstName, lastName, email, phone)
379             {
380 
381                 this->title = title;
382                 this->department = department;
```

```
                this->salary = salary;
            }
        };

    // multi level inheritance :  class1 inherited from class2 and class2 inherited from class3   ...

    // -- inheritance visibility modes  ------------------------------------[-]
    /*
        public :  public  keep public , protected keep protected
        private :   -- public && protected will be private in the derived class  (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 1 inherited  by class2 and class2 itself inherited by class3 .. classN ..
        -hierarchal  :  one class inherited by multiple classes

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

        -----special type --------
            -hybrid : one class inherit from  multiple classes  that also inherit  from another  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;
```

```
434
435         cPerson p2;
436         // down casting :
437         cEmployee *e2 = &p2;
438         cout << e2->name << endl;
439
440         return 0;
441     }
442 // friend class  : -------------------------------------[-]
443  /*
444     A friend class can access both private and protected members of the class
445     in which it has been declared as friend.
446  */
447     //syntax
448         class className{
449
450             friend className2;
451
452         };
453         class className2{
454
455         };
456
457  // friend function : -------------------------------------[-]
458  /*
459     A friend function in C++ is a function that is declared outside a class but is capable
    of
460     accessing the private and protected members of the class. There could be situations
461     in programming wherein we want two classes to share their members. These members may be
462     data members, class functions or function templates. In such cases, we make the desired
463     function, a friend to both these classes which will allow accessing private and
464     protected data of members of the class.
465  */
466
467     //syntax :
468         class className{
469
470         friend  datatype functionName(arg);
471
472         };
473         datatype functionName(arg){
474
475             //code
476         }
477
478 //✨ objects with vectors :  -------------------------------------[-]
479     int main(){
480
481     vector <clsA> v1;
482     short NumberOfobjects=5;
483
484     // inserting object at the end of vector
485         for (int i = 0; i < NumberOfObjects; i++)
486         v1.push_back(clsA(i));
487
488     //✨ 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) };

    // using print method for each of three elements.
    for (int i = 0; i < 3; i++)
    obj[i]. Print();

    return 0;
    }

// operator overloading : ------------------------------------[-]
/*
Operator overloading in C++ allows you to define custom behaviors for operators when they
are used
with user-defined types (classes and structures). It enables you to extend the functionality
of operators
beyond their predefined meanings for built-in types. Here's an in-depth look at operator
overloading in C++:
*/


    1. **Syntax of Operator Overloading:**
    /*
    - Operator overloading is achieved by defining a member function or a friend function
for the operator with the keyword
        `operator` followed by the operator symbol.
```

```
    - The overloaded operator function typically takes one or more parameters representing the operands of the operator.
    - The return type and behavior of the operator function depend on the specific operator being overloaded.
    */
        // Example:
            ```cpp
            class MyClass {
            public:
                MyClass operator+(const MyClass& other) const {
                    MyClass result;
                    // Define addition behavior
                    return result;
                }
            };
            ```

    2. **Types of Operators that Can Be Overloaded:**
    /*
    - Most operators in C++ can be overloaded, including arithmetic, relational, logical, bitwise, assignment, and others.
    - Some operators, such as member access (`.`) and the scope resolution operator (`::`), cannot be overloaded.
    - Unary operators, binary operators, and ternary operators can all be overloaded.
    */
    3. **Member Functions vs. Friend Functions:**
    /*
    - Operator overloading can be implemented as a member function or a friend function.
    - Member functions are part of the class definition and have access to the private members of the class.
    - Friend functions are declared outside the class but have access to its private members if declared as a friend.
    - The choice between member functions and friend functions depends on the specific requirements of the operator and its operands.
    */
    4. **Implicit vs. Explicit Overloading:**
    /*
    - Operator overloading can be performed implicitly or explicitly.
    - Implicit overloading occurs when the operator is used with objects of the class, and the compiler automatically
        calls the overloaded operator function.
    - Explicit overloading involves explicitly calling the overloaded operator function using function syntax.
    */

        //Example (Implicit Overloading):
            ```cpp
            MyClass obj1, obj2;
            MyClass result = obj1 + obj2; // Implicit call to overloaded operator+
            ```

    5. **Rules and Best Practices:**
    /*
    - Operator overloading should adhere to the principle of least surprise, maintaining intuitive behavior similar to built-in types.
    - Overloaded operators should respect their conventional meanings to avoid confusion and maintain code readability.
    - Overloaded operators should be implemented symmetrically when applicable (e.g., `+` and `-` should be consistent with each other).
```

```
    - Overloaded operators should be used judiciously to enhance code clarity and maintainability, avoiding excessive or obscure overloading.
    */

    //example :
        //  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;
        }
/*
    Operator overloading provides a powerful mechanism for extending the expressive capabilities of C++ classes and enabling more
    natural and intuitive syntax for user-defined types. When used appropriately, operator overloading can improve code readability
    and maintainability, making C++ programs more concise and expressive. However, it should be used judiciously and with care to ensure
    that the behavior of overloaded operators remains consistent and intuitive.
*/


//✨ 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.
*/
// type of implementation  :-------------------------------------[-
/*
    1- function overloading
    2- function overwriting
    3- operator overloading
    4- virtual functions
*/

//virtual functions  ------------------------------------[-]
/*
    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;
        // up casting  convert from employee (derived class) to person (base class)
            cPerson *p1 = &e1;
            cPerson *p2 = &s1;
            // due to print it's a virtual function so cPerson->print will print the print function of cEmployee class
            p1->print();
            p2->print();

            return 0;
        }

// static/Early binding  vs  dynamic/late binding   ------------------------------------[-]
/*
Static (or Early) Binding and Dynamic (or Late) Binding are two different mechanisms used
for resolving function calls in object-oriented programming languages like C++.
*/

1. **Static Binding (Early Binding):**
/*
   - Static binding refers to the process of linking a function call to its definition at compile-time.
   - In static binding, the decision about which function to call is made by the compiler based on the declared type of the object or pointer.
   - The compiler determines the function to call by examining the static (compile-time) type of the object or pointer, not its runtime (actual) type.
   - Static binding is efficient but less flexible because the function call is resolved at compile-time, making it suitable for performance-critical
     scenarios where compile-time optimization is essential.
*/

        //Example (C++):
        ```cpp
        class Base {
        public:
            void display() {
                std::cout << "Base display" << std::endl;
            }
        };

        class Derived : public Base {
        public:
            void display() {
                std::cout << "Derived display" << std::endl;
            }
        };

        int main() {
            Base* ptr = new Derived();
            ptr->display(); // Calls Base::display() due to static binding
            delete ptr;
            return 0;
        }
        ```

    2. **Dynamic Binding (Late Binding):**
```

```
    /*
    - Dynamic binding refers to the process of linking a function call to its definition at runtime.
    - In dynamic binding, the decision about which function to call is deferred until runtime and is based on the actual type of the object.
    - Dynamic binding allows for polymorphic behavior, where a function call can be resolved to different implementations based on the runtime type of the object.
    - Dynamic binding is achieved through the use of virtual functions and inheritance hierarchies in object-oriented programming languages like C++.
    - It provides greater flexibility and enables features such as polymorphism, runtime polymorphic behavior, and dynamic dispatch.
    */

        //Example :
        ```cpp
        class Base {
        public:
            virtual void display() {
                std::cout << "Base display" << std::endl;
            }
        };

        class Derived : public Base {
        public:
            void display() override {
                std::cout << "Derived display" << std::endl;
            }
        };

        int main() {
            Base* ptr = new Derived();
            ptr->display(); // Calls Derived::display() due to dynamic binding
            delete ptr;
            return 0;
        }
        ```
/*
    In summary, static binding resolves function calls at compile-time based on the declared type of the object or
    pointer, while dynamic binding defers the decision until runtime, based on the actual type of the object.
    Dynamic binding enables polymorphism and runtime polymorphic behavior, making it a powerful mechanism for
    designing flexible and extensible software systems.
*/

// abstract class ------------------------------------[-]
/*
    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.
*/
/*
```

```
    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.
 */

    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.
    */
        // syntax :
            ```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() {}
        };
        ```

// 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{

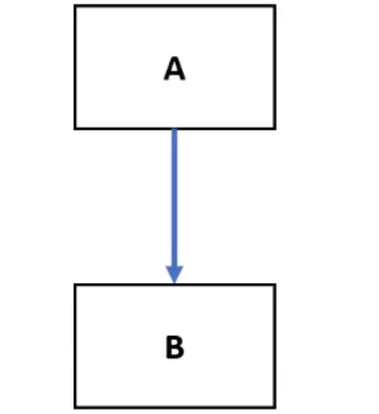
```
```

```
            // 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;
        }

// using strcut with  classes : ------------------------------------[-]
    class className{

    struct structName{
    att1;
    att2;
    };
```

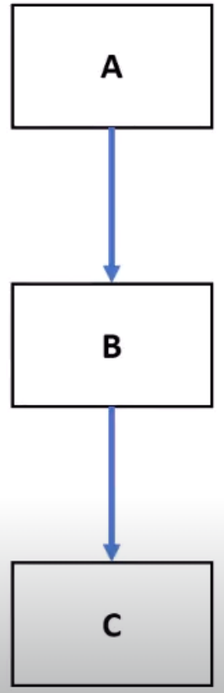
```
    public :
    structName ob1;

    };

    //* 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-------- --=---- -----------";
            }
        };
// 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
        {
```

```
1017            string title;
1018            string department;
1019            float salary;
1020
1021        public:
1022            void print(bool isBaseClass = true)
1023            {
1024                cPerson::print(false);
1025                cout << "the title      : " << title << "\n";
1026                cout << "the department : " << department << "\n";
1027                cout << "the salary     : " << salary << "\n";
1028                if (isBaseClass)
1029                    cout << "_____________________________________\n";
1030            }
1031
1032            // setters :
1033            void setTitle(string title) { this->title = title; }
1034            void setDepartment(string department) { this->department = department; }
1035            void setSalary(float salary) { this->salary = salary; }
1036
1037            // getters :
1038            string getTitle() { return title; }
1039            string getDepartment() { return department; }
1040            float getSalary() { return salary; }
1041
1042            cEmployee(int id, string firstName, string lastName, string email, string phone,
    string title, string department, float salary)
1043                : cPerson(id, firstName, lastName, email, phone)
1044            {
1045
1046                this->title = title;
1047                this->department = department;
1048                this->salary = salary;
1049            }
1050        };
1051
1052
```

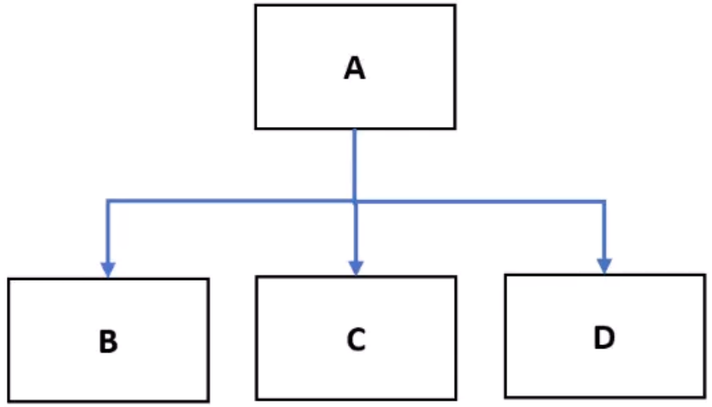
Inheritance Types



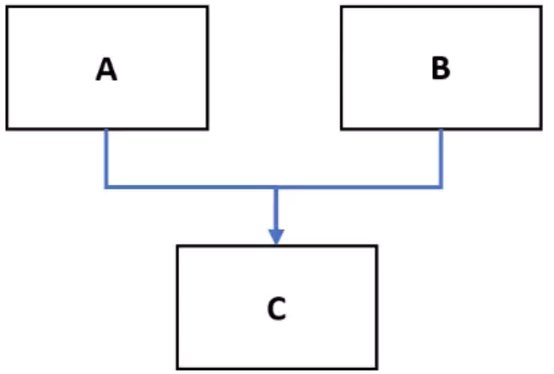
Single Inheritance



Multi Level Inheritance



Hierarchal Inheritance



Multiple Inheritance