

# Functional (FP) vs Object Oriented (OOP)

## University System Example (FP)

- AddStudent
  - UpdateStudent
  - DeleteStudent
  - CalculateAverage
  - AddCourse
  - UpdateCourse
  - DeleteCourse
  - EnrollStudentInCourse
  - UnEnrollStudentFromCourse
  - HowManyStudentsInCourse
  - Doctor (Add, Edit, Delete)
  - AssignCourseToDoctor
- 
- SendEmailToStudent
  - SendTextMessageToStudent
  - SendEmailToDoctor
  - SendTextMessageToDoctor
  - CallStudent
  - CallDoctor
  - AddEmployee
  - UpdateEmployee
  - DeleteEmployee
  - CalculateSalary
  - PaySalary
  - .....



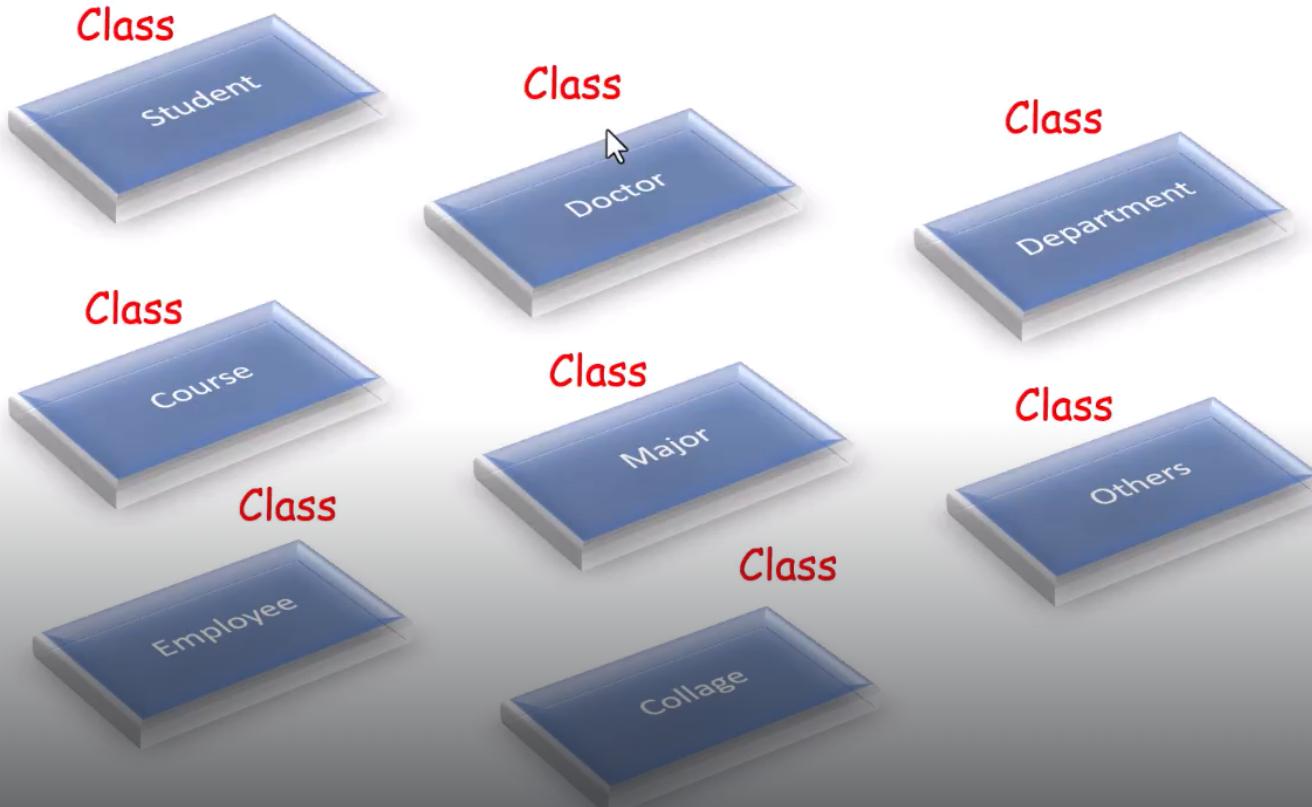
Simply

You can have thousands of functions in your System!!!!



# What is Class? Why we call it Class?

## University System Example (OOP)



Simply  
Class came from Classification



# Class Members



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

class clsPerson
{
public:

    string FirstName;
    string LastName;

    string FullName()
    {
        return FirstName + " " + LastName;
    }
};

int main()
{
    clsPerson Person1;

    Person1.FirstName = "Mohammed";
    Person1.LastName = "Abu-Hadhoud";

    cout << "Person1: " << Person1.FullName()
    << endl;
}
```

## Members



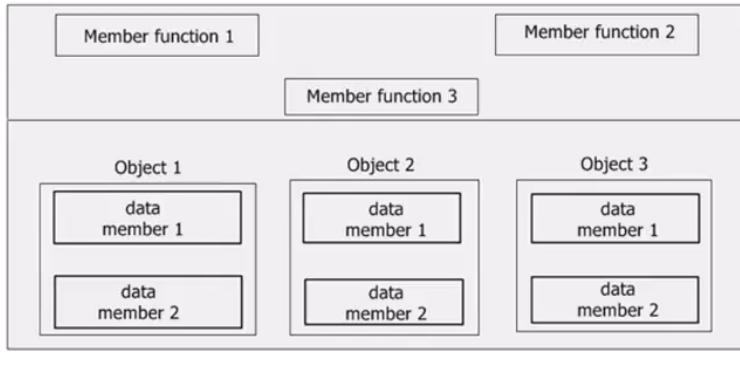
**Data Members:** Any variable declared inside the class that holds Data.

**In Our Case:**  
FirstName, LastName  
Are Data Members

**Member Methods(Functions):** Any Function Or Procedure declared inside the class.

**In Our Case:**  
FullName()  
Is a Member Function/Method

# Each instance has its own space in memory, Only Member functions are shared among all objects



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

class clsPerson
{
public:

    string FirstName;
    string LastName;

    string FullName()
    {
        return FirstName + " " + LastName;
    }
};

int main()
{
    clsPerson Person1,Person2;

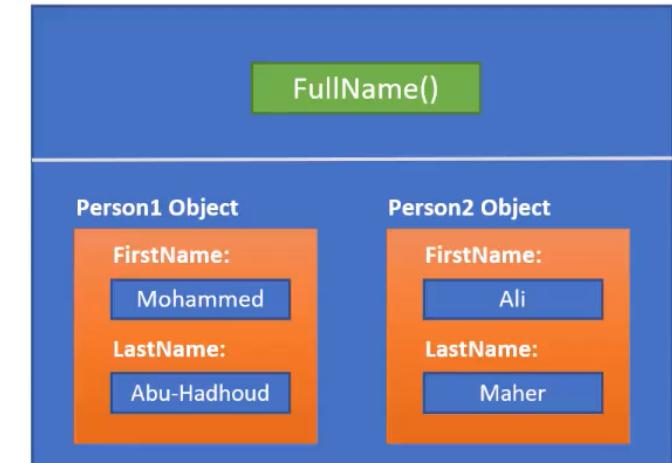
    Person1.FirstName = "Mohammed";
    Person1.LastName = "Abu-Hadhoud";

    Person2.FirstName = "Ali";
    Person2.LastName = "Maher";

    cout << "Person1: " << Person1.FullName() << endl;
    cout << "Person2: " << Person2.FullName() << endl;
}
```



Copyright © 2022



Mohammed Abu-Hadhoud

MBA, PMOC, PgMP®, PMP®, PMI-RMP®, CM, ITILF, MCPD, MCSD  
26+ years of experience

The `sizeof` operator in C++ calculates the size of objects and types based on their memory representation. However, it does not include the size of member functions or member function pointers in its calculation.

Member functions in a class are not part of the object's memory representation. They are typically shared among all instances of the class and exist only once in memory, regardless of the number of objects created. Each object only stores the data members (variables) of the class, not the member functions.

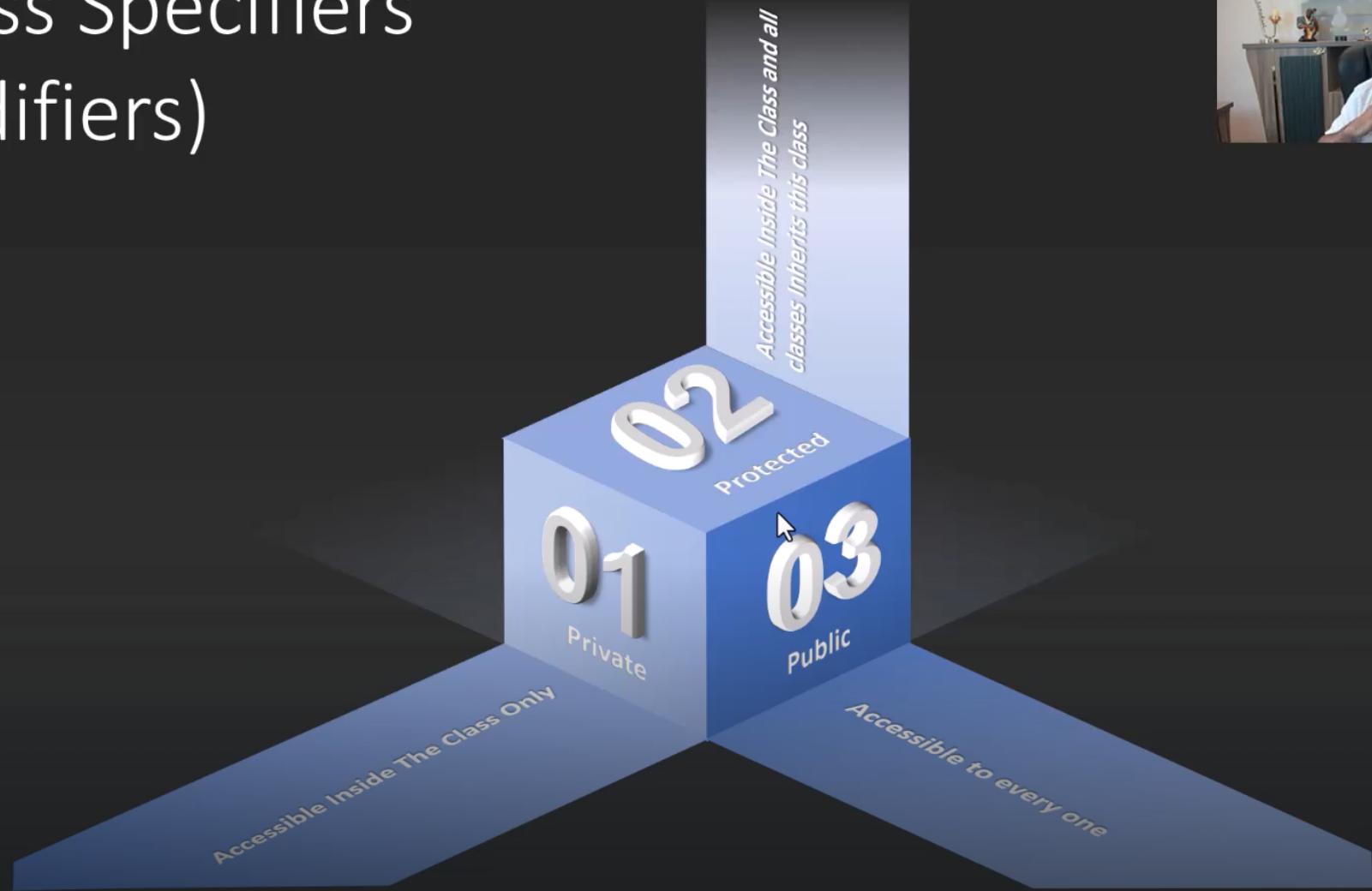
The size of a class object, as calculated by `sizeof`, includes only the memory required to store the data members of the class and any additional padding or alignment requirements. It does not include the memory required for member functions.

Therefore, when you use `sizeof` with a class or an object, it provides the size of the data members and any additional memory requirements imposed by the implementation and platform, but it does not include the size of the member functions themselves.

## Size of Classes and Objects in Memory

- The size of a class in memory is determined by the memory layout and alignment requirements of its member variables. It can be affected by factors such as padding and memory alignment.
- The size of an object of a class includes the memory required to store its data members, any additional padding or alignment requirements, and potentially the memory needed for virtual function tables (vtables) if the class has virtual functions.
- The sizeof operator in C++ can be used to calculate the size of an object or class. It returns the size in bytes and provides an estimation of the memory required by the object or class.
- The sizeof operator calculates the size of the data members within the class. It does not include the size of member functions or virtual function tables (vtables) in its calculation.
- The actual size of an object or class may be larger than the sum of its data members due to padding and alignment requirements imposed by the compiler and platform. Padding is added to ensure proper alignment of the data members in memory.
- The size of an object or class can be influenced by various factors, including the size of its data members, alignment requirements, padding, compiler optimization settings, and platform-specific considerations.
- The sizeof operator provides a way to estimate the memory footprint of an object or class, but the actual memory usage may vary depending on the specific implementation, compiler, and platform.
- It is important to note that the sizeof operator does not account for dynamic memory allocation or additional memory overhead caused by virtual inheritance or virtual base classes.
- To accurately measure the memory usage of a program or analyze the memory layout of a class, specialized profiling tools and memory analyzers can be used.

# Access Specifiers (Modifiers)



10:07

12:48

```

#include <iostream>
using namespace std;
class clsPerson
{
private:
    // only accessible inside this class
    int Variable1 = 5;

    int Function1()
    {
        return 40;
    }

protected:
    // only accessible inside this class and all classes inherits this class
    int Variable2 = 100;
    int Function2()
    {
        return 50;
    }

public:
    // accessible for everyone outside/inside/and classes inherits this class
    string FirstName;
    string LastName;

    string FullName()
    {
        return FirstName + " " + LastName;
    }

    float Function3()
    {
        return Function1() * Variable1 * Variable2;
    }
};

int main()
{
    clsPerson Person1;
    Person1.FirstName = "Mahmoud";
    Person1.LastName = "Mattar";

    cout << "Person1: " << Person1.FullName() << endl;
    cout << Person1.Function3();
}
*/

```

### Access Specifiers:

In C++, access specifiers control the visibility and accessibility of class members. They determine which parts of a class are accessible from different contexts.

- private: The private access specifier restricts access to class members to be only within the class itself. They are not accessible from outside the class or from derived classes. Private members are typically used for internal implementation details that are not intended to be accessed directly.

- protected: The protected access specifier allows access to class members within the class itself and any derived classes.

They are not accessible from outside the class hierarchy. Protected members are often used to provide access to derived classes while still restricting access from other parts of the program.

- public: The public access specifier provides unrestricted access to class members from any part of the program.

Public members can be accessed directly from outside the class and are inherited by derived classes. They are typically used for interface and functionality that needs to be accessible to other parts of the program.

By default, if no access specifier is specified, members are considered private.

It's important to carefully choose the appropriate access specifier for class members based on the desired level of encapsulation and the intended usage of the class.

Note: The access specifiers only affect the visibility of class members; they do not impact the memory layout or size of the class or object in memory.

\*/



```
1 #include <iostream>
2 using namespace std;
3
4 class clsPerson
5 {
6 private:
7     string _FirstName;
8     string _LastName;
9
10 public:
11     // Property Set: Sets the value of the first name
12     void setFirstName(string FirstName)
13     {
14         _FirstName = FirstName;
15     }
16
17     // Property Get: Retrieves the value of the first name
18     string FirstName()
19     {
20         return _FirstName;
21     }
22
23     // Property Set: Sets the value of the last name
24     void setLastName(string LastName)
25     {
26         _LastName = LastName;
27     }
28
29     // Property Get: Retrieves the value of the last name
30     string LastName()
31     {
32         return _LastName;
33     }
34
35     // Returns the full name by concatenating the first and last names
36     string FullName()
37     {
38         return _FirstName + " " + _LastName;
39     }
40 };
41
42 int main()
43 {
44     clsPerson Person1;
45     Person1.setFirstName("Mahmoud");
46     Person1.setLastName("Mattar");
47
48     cout << "First Name: " << Person1.FirstName() << endl;
49     cout << "Last Name: " << Person1.LastName() << endl;
50     cout << "Full Name: " << Person1.FullName() << endl;
51
52     system("pause > nul");
53     return 0;
54 }
```



```
1 In object-oriented programming, Property Set and Get methods are used to encapsulate the access and manipulation of private member variables (properties) of a class.
2 They provide a controlled way to set and retrieve the values of these variables.
3
4 explanation of Property Set and Get:
5
6 Property Set (Setter):
7
8 A Property Set method is responsible for setting the value of a private member variable.
9 It usually takes a parameter that represents the new value to be assigned.
10 Inside the setter method, you can perform any necessary validation or additional logic before assigning the value to the member variable.
11
12 Property Get (Getter):
13
14 A Property Get method is used to retrieve the value of a private member variable.
15 It does not take any parameters.
16 The getter method returns the current value of the member variable.
17
18 By using Property Set and Get methods, you can enforce encapsulation and control the access to the internal state of an object
19 This approach allows you to protect the integrity of the object's data and provide a standardized way for other code to interact
with the object's properties.
20
21 In the provided code, the Property Set methods (setFirstName and setLastName)
22 are used to set the values of the private member variables _FirstName and _LastName, respectively.
23 The Property Get methods (FirstName and LastName) are used to retrieve the values of these member variables.
24
25 Using Property Set and Get methods helps promote encapsulation and maintain the principle of information hiding within a class.
```



```
1 #include <iostream>
2 using namespace std;
3
4 class clsPerson {
5 private:
6     string _FirstName;
7     string _LastName;
8 public:
9     void setFirstName(string firstName)
10    {
11        _FirstName = firstName;
12    }
13    string getFirstName()
14    {
15        return _FirstName;
16    }
17    void setLastName(string lastName)
18    {
19        _LastName = lastName;
20    }
21    string getLastName()
22    {
23        return _LastName;
24    }
25    string getFullName()
26    {
27        return _FirstName + " " + _LastName;
28    }
29
30 // declaration specification class
31 __declspec(property(get = getFirstName, put = setFirstName)) string FirstName;
32 __declspec(property(get = getLastname, put = setLastName)) string LastName;
33 __declspec(property(get = getFullName)) string FullName;
34 };
35
36 int main()
37 {
38     clsPerson person1;
39
40     person1.setFirstName("M7");
41     person1.setLastName("Trojan");
42
43     cout << "first name: " << person1.getFirstName() << endl;
44     cout << "last name: " << person1.getLastName() << endl;
45     cout << "full name: " << person1.getFullName() << endl;
46
47     cout << "\n=====\\n\\n";
48
49     person1.FirstName = "Mahmoud";
50     person1.LastName = "Mattar";
51
52     cout << "first name: " << person1.FirstName << endl;
53     cout << "last name: " << person1.LastName << endl;
54     cout << "full name: " << person1.FullName << endl;
55 }
56
57 /*
58 - __declspec(property) is a Microsoft-specific extension in C++.
59 - It allows you to define properties for a class, encapsulating data with getter and setter methods.
60 - Properties provide a convenient way to access and modify class member variables.
61 - In this code, the properties FirstName, LastName, and FullName are defined using __declspec(property).
62 - The get and put attributes within __declspec(property) specify the associated getter and setter methods.
63 - The properties can be accessed and modified directly as if they were regular data members.
64 - This extension simplifies the creation of properties and improves code readability, especially in Microsoft Visual C++.
65 - Please note that __declspec(property) is specific to Microsoft's C++ compiler and may not be supported by other compilers like
g++ and clang.
66 */
67
```

The lines of code you mentioned are using a declaration specification in C++ to create properties for the `clsPerson` class. The `\_\_declspec(property)` is a Microsoft-specific extension that allows you to define properties for a class without explicitly writing separate getter and setter methods.

Here's a breakdown of the code:

cpp

 Copy code

```
__declspec(property(get = getFirstName, put = setFirstName)) string FirstName;
```

This line declares a property named `FirstName` of type `string`. The `get` attribute specifies that the `getFirstName()` method should be used as the getter for this property, and the `put` attribute specifies that the `setFirstName()` method should be used as the setter.

Similarly, the next line declares a property named `LastName` using the `getLastName()` and `setLastName()` methods as the getter and setter, respectively.

cpp

 Copy code

```
__declspec(property(get = getFullName)) string FullName;
```

cpp

 Copy code

```
__declspec(property(get = getFullName)) string FullName;
```

This line declares a property named `FullName` that only has a getter, as there is no corresponding setter specified. It uses the `getFullName()` method as the getter.

By using the declaration specification, you can access and modify the properties directly, as shown in the `main()` function:

cpp

 Copy code

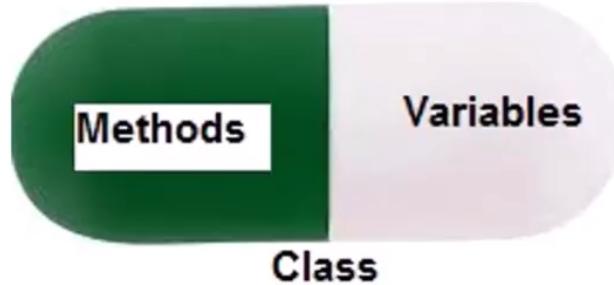
```
person1.FirstName = "Mahmoud";
person1.LastName = "Mattar";

cout << "first name: " << person1.FirstName << endl;
cout << "last name: " << person1.LastName << endl;
cout << "full name: " << person1.FullName << endl;
```

These lines demonstrate how you can assign values to the `FirstName` and `LastName` properties using the `=` operator, and how you can access the property values directly without explicitly calling the getter methods.

It's worth noting that the `\_\_declspec(property)` syntax is a Microsoft-specific extension and may not be supported by all C++ compilers.

# Encapsulation in Object Oriented (**OOP**)



**StudentObject:**

**Student1.Name**

**Student1.Email**

**Student1.GetEnrolledCourses()**

**Student1.CalcuateAverage()**

**Student1.SyndEmail(Subject, Body)**

**Student1.EnrollInCourse(10)**

**Student1.PayFees**

**Student1.UnEnrolFromCourse(10)**

You can do whatever you want  
with a certain student only  
through its object



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.

في المصطلحات العادية ، يُعرف التعريف بأنه التغليف للبيانات والمعلومات في وحدة واحدة

في البرمجة الشيئية ، يُعرف التعريف بأنه ربط البيانات والوظائف التي تتعامل معها معاً



1 In simple terms, abstraction “displays” only the relevant attributes of objects and “hides” the unnecessary details.

• عبارات بسيطة ، فإن التجريد "يعرض" فقط الفانكشن والمتغيرات بقاعد الاوبراكت المهمة بس و "بيخفي" التفاصيل الي مش ضروريه

3

4

A constructor is a special type of member function that is called automatically when an object is created.

True

False

In C++, a constructor has the same name as that of the class and it does not have a return type.

True

False

# Constructor should:

correct

Have the same name of the class.



correct

Should not return type.



correct

Should be Public.



**Default Constructor:** A constructor with no parameters is known as a default constructor.

correct

True



False

# You should always write Default Constructor.

correct

No, because if you don't write it the compiler will write it for you.



Yes, you should write it always.

Parameterized Constructor: a constructor with parameters is known as a parameterized constructor. This is the preferred method to initialize member data.

correct

True



False

When you have a parameterized constructor it will override the default constructor.

correct

True



False

A constructor is primarily used to initialize objects. They are also used to run a default code when an object is created.

correct

True



False

The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

True

False

The process of initializing members of an object through a copy constructor is known as copy initialization.

correct

True



False

It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.

correct

True



False

The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

correct

True



False

You should always implement a copy constructor in your code.

Yes , you should.

correct

No, because the compiler will do it for you.



## What are the types of constructors:

correct

Default Constructor



correct

Parameterized Constructor



correct

Copy Constructor



None of the above

< Back

Continue >

# Can you have more than one constructor in a class?

No, You cannot

Yes, You can using function overloading, and this is called "Constructor Overloading"



```
1
2 #include <iostream>
3 using namespace std;
4 class clsPerson
5 {
6 public:
7     string FullName;
8     // This is constructor will be called when object is built.
9     clsPerson()
10    {
11         FullName = "Mahmoud Mattar";
12         cout << "\nHi, I'm Constructor";
13     }
14     // This is destructor will be called when object is destroyed.
15     ~clsPerson()
16    {
17         cout << "\nHi, I'm Destructor";
18     }
19 };
20 void Fun1()
21 {
22     clsPerson Person1;
23     // after exiting from function, person1 will be destroyed and destructor will be called.
24 }
25 void Fun2()
26 {
27     clsPerson *Person2 = new clsPerson;
28     // always use delete whenever you use new, otherwise object will remain in memory
29     delete Person2;
30 }
31
32 int main()
33 {
34     Fun1();
35     Fun2();
36
37     system("pause > nul");
38     return 0;
39 }
```

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.

correct

True



False

Destructor has the same name as their class name preceded by a tilde (~) symbol.

correct

True



False

It is not possible to define more than one destructor.

correct

True



False

The destructor is only one way to destroy the object created by constructor. Hence destructor can-not be overloaded.

correct

True



False

Destructor neither requires any argument nor returns any value.

correct

True



False

It is automatically called when object goes out of scope.

True

False



```
1 // Static Members
2
3 #include <iostream>
4 using namespace std;
5 class clsA
6 {
7 public:
8     int var;
9     static int counter;
10    clsA()
11    {
12        counter++;
13    }
14    void Print()
15    {
16        cout << "\nvar = " << var << endl;
17        cout << "counter = " << counter << endl;
18    }
19};
20
21 int clsA::counter = 0; // static variable initialisation outside the class
22 int main()
23 {
24
25     clsA A1, A2, A3;
26     A1.var = 10;
27     A2.var = 20;
28     A3.var = 30;
29     A1.Print();
30     A2.Print();
31     A3.Print();
32
33     A1.counter = 500;
34     cout << "\nafter changing the static member counter in one object:\n";
35     A1.Print();
36     A2.Print();
37     A3.Print();
38 }
39
40 /*
41 الستاتيك ميمبر بيتمسي برضو شيرد ميمبر
42 اي تغير في الستاتيك داتا ميمبر بيقدر اي اويجكت يع
43 دل عليه
44 زي الس متجوز اربع نسوان كل واحدة ليها بيت خاص بيها الي هي الميموري الوكيشن بس لو واحدة قتلت جوزها بيخرسروه الكل
45 */
```

Static Member is a variable that is shared for all objects, any object  
modifies it it get modified for all other objects.

correct

True



Flase

Each Object has it's own static members.

True

correct

False



Static members are on the class level not for each object.

correct

True



False

Static Members are accessible from all objects.

correct

True



False



```
1 //  Static Methods (Functions)
2
3 #include <iostream>
4 using namespace std;
5 class clsA
6 {
7 public:
8     static int Function1()
9     {
10         return 10;
11     }
12     int Function2()
13     {
14         return 20;
15     }
16 };
17 int main()
18 {
19
20     // The following line calls static function directly via class not through the object
21     // At class level you can call only static methods and static members
22     cout << clsA::Function1() << endl;
23
24     // static methods can also be called through the object.
25     clsA A1, A2;
26     cout << A1.Function1() << endl;
27     cout << A1.Function2() << endl;
28     cout << A2.Function1() << endl;
29 }
```

Static Functions can be called at class level without a need to have an object.

correct

True



Flase

Static Functions can not be called through object.

True

correct

False



Static Functions can be called through any object as well as through the class itself.

correct

True



False

If you have a static function, can you access a non-static members of the class from inside that function?

Yes

correct

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.





```
1 #include <iostream>
2 using namespace std;
3
4 class MyClass {
5 public:
6     int x;           // Instance-specific member variable
7     static int count; // Static member variable shared among all instances
8
9     MyClass() {
10         x = 0;        // Initialize instance-specific variable
11         count++;    // Increment the static counter for each object created
12     }
13
14     static void displayCount() {
15         // Static member function can only access static member variables
16         // It cannot access instance-specific member variables like 'x'
17         cout << "Total objects created: " << count << endl;
18     }
19 };
20
21 int MyClass::count = 0; // Initialize the static member variable outside the class
22
23 int main() {
24     MyClass obj1;
25     MyClass obj2;
26     MyClass obj3;
27
28     obj1.x = 10; // Set instance-specific variable for obj1
29     obj2.x = 20; // Set instance-specific variable for obj2
30
31     // Access static member variable directly using the class name
32     cout << "Static member count: " << MyClass::count << endl;
33
34     // Access static member function using the class name
35     MyClass::displayCount();
36
37     // Access static member variable using an object (not recommended)
38     // obj3.displayCount(); // Avoid accessing static members using objects
39
40     // Access instance-specific variable using objects
41     cout << "obj1.x: " << obj1.x << endl;
42     cout << "obj2.x: " << obj2.x << endl;
43     cout << "obj3.x: " << obj3.x << endl;
44
45     return 0;
46 }
47
48 /*
49 Explanation:
50
51 1. We define a class 'MyClass' with two member variables: 'x' and 'count'.
52     The 'x' is an instance-specific member variable, and 'count' is a static member variable shared among all instances.
53
54 2. The constructor of the 'MyClass' class initializes the 'x' variable to '0' for each object created and increments the 'count' static variable.
55
56 3. We have a static member function 'displayCount' that can only access static member variables.
57     It cannot access instance-specific member variables like 'x'.
58
59 4. In the 'main' function, we create three objects of the 'MyClass' class: 'obj1', 'obj2', and 'obj3'.
60
61 5. We set instance-specific variables 'x' for 'obj1' and 'obj2'.
62
63 6. We can access the static member variable 'count' directly using the class name 'MyClass::count'.
64
65 7. We can call the static member function 'displayCount' using the class name 'MyClass::displayCount'.
66
67 8. It's not recommended to access static member variables using objects (e.g., 'obj3.displayCount()') as they belong to the entire class, not individual objects.
68
69 9. We can access instance-specific member variables using objects (e.g., 'obj1.x', 'obj2.x', and 'obj3.x').
70
71 When you run this code, it will display the total count of objects created and the values of instance-specific variables for 'obj1', 'obj2', and 'obj3'.
72 The static member count will be the same for all objects, indicating that it is shared among them.
73 */
```



```
1 #include <iostream>
2 using namespace std;
3
4 class MyClass {
5 public:
6     // Static method that returns a constant value of 10.
7     static int staticMethod() {
8         // Static method does not have access to instance-specific member variables (non-static variables).
9         // It can only access static member variables (class-level variables).
10        // It does not have access to the 'this' pointer since there is no specific instance associated with it.
11        return 10;
12    }
13
14    // Non-static method that returns a constant value of 20.
15    int nonStaticMethod() {
16        // Non-static method (instance-specific method) has access to both static and non-static member variables.
17        // It can access the 'this' pointer, which points to the specific instance calling the method.
18        return 20;
19    }
20 };
21
22 int main() {
23     // Call the static method directly through the class name (no object creation required).
24     cout << "Static method result: " << MyClass::staticMethod() << endl;
25
26     // Static methods can also be called through objects, but it's not recommended, as they are not instance-specific.
27     MyClass obj1;
28     cout << "Static method result through object: " << obj1.staticMethod() << endl;
29
30     // Call the non-static method through an object.
31     cout << "Non-static method result: " << obj1.nonStaticMethod() << endl;
32
33     return 0;
34 }
35
36 /*
37 Explanation:
38
39 1. In the example, we have a class 'MyClass' with two methods: 'staticMethod()' and 'nonStaticMethod()'.
```

40

```
41 2. The 'staticMethod()' is a static method, indicated by the 'static' keyword in its declaration. It does not operate on any specific object of the class  
42 and can be called directly through the class name 'MyClass::staticMethod()'.
```

43

```
44 3. Static methods cannot access instance-specific member variables (non-static variables) or the 'this' pointer since they are not associated with any specific instance.
```

45

```
46 4. The 'nonStaticMethod()' is a non-static method, also known as an instance-specific method.  
47 It can be called through objects of the class, such as 'obj1.nonStaticMethod()'.
```

48

```
49 5. Non-static methods can access both static and non-static member variables and have access to the 'this' pointer, which points to the specific instance calling the method.
```

50

```
51 6. In the 'main()' function, we demonstrate calling the static method both directly through the class name 'MyClass::staticMethod()' and through an object 'obj1.staticMethod()'.
```

52 However, it's important to note that calling static methods through objects is not recommended since they are not instance-specific.

53

```
54 When to use static methods:  
55 - Use static methods when the functionality does not depend on any specific object's state and can be defined at the class level.  
56 - Use static methods for utility functions or operations that are relevant to the entire class and do not require access to instance-specific data.
```

57

```
58 When not to use static methods:  
59 - Avoid using static methods when the functionality requires access to instance-specific data or when the method's behavior may vary for different instances of the class.
```

60

```
61 In such cases, use non-static methods.
```

62 \*/

```

1 #include <iostream>
2 using namespace std;
3
4 // Woman class representing different women with their information
5 class woman {
6 private:
7     string _Name;
8     int _Age;
9     string _CarModel;
10 public:
11     // Constant static member representing the husband's name
12     const static string _HusbandName;
13
14     // Static member indicating whether the husband is alive or not
15     static bool ISHusbandAlive;
16
17     // Constructor to initialize woman's information
18     woman() {
19         _Name = "Unknown";
20         _Age = 0;
21         _CarModel = "Unknown";
22     }
23
24     // Setter for woman's name
25     void setWomanName(string name) {
26         _Name = name;
27     }
28
29     // Getter for woman's name
30     string getWomanName() {
31         return _Name;
32     }
33
34     // Setter for woman's age
35     void setWomanAge(int age) {
36         _Age = age;
37     }
38
39     // Getter for woman's age
40     int getWomanAge() {
41         return _Age;
42     }
43
44     // Setter for woman's car model
45     void setCarModel(string Model) {
46         _CarModel = Model;
47     }
48
49     // Getter for woman's car model
50     string getCarModel() {
51         return _CarModel;
52     }
53
54     // Static method to get the husband's name
55     static string getHusbandName() {
56         return _HusbandName;
57     }
58
59     // Print woman's information
60     void print() {
61         cout << "\nWoman Info:\n";
62         cout << "*****\n";
63         cout << "Name: " << _Name << endl;
64         cout << "Age: " << _Age << endl;
65         cout << "Car Model: " << _CarModel << endl;
66         cout << "Husband Name: " << _HusbandName << endl;
67         cout << "Is The Husband Alive?: " << (ISHusbandAlive ? "Yes alhamdalalah" : "No, one of his wives killed him") << endl;
68     }
69
70     // Property to allow direct access to woman's name as if it were a regular variable
71     _declspec(property(get = getWomanName, put = setWomanName)) string Name;
72
73     // Property to allow direct access to woman's age as if it were a regular variable
74     _declspec(property(get = getWomanAge, put = setWomanAge)) int Age;
75
76     // Property to allow direct access to woman's car model as if it were a regular variable
77     _declspec(property(get = getCarModel, put = setCarModel)) string CarModel;
78 };
79
80
81 // Static member initialization outside the class
82 const string woman::_HusbandName = "Mahmoud Mattar";
83
84 // Initialize the static member to true (Husband is alive) initially
85 bool woman::ISHusbandAlive = true;
86
87 int main() {
88     // Create four instances of the woman class
89     woman woman1, woman2, woman3, woman4;
90
91     // Set information for woman1 and display her details
92     woman1.setWomanName("Rahma");
93     woman1.setWomanAge(22);
94     woman1.setCarModel("Kia");
95     woman1.print();
96
97     // Set information for woman2 using properties and display her details
98     woman2.Name = "Asmaa";
99     woman2.Age = 25;
100    woman2.CarModel = "BMW";
101    cout << "\nWoman Info:\n";
102    cout << "*****\n";
103    cout << "Name: " << woman2.Name << endl;
104    cout << "Age: " << woman2.Age << endl;
105    cout << "Car Model: " << woman2.CarModel << endl;
106    cout << "Husband Name: " << woman2._HusbandName << endl;
107    cout << "Is The Husband Alive?: " << (woman::ISHusbandAlive ? "Yes alhamdalalah" : "No, one of his wives killed him") << endl;
108
109    // Set information for woman3 and display her details
110    woman3.setWomanName("Bassma");
111    woman3.setWomanAge(27);
112    woman3.setCarModel("Mercedes");
113    cout << "\nWoman Info:\n";
114    cout << "*****\n";
115    cout << "Name: " << woman3.Name << endl;
116    cout << "Age: " << woman3.Age << endl;
117    cout << "Car Model: " << woman3.CarModel << endl;
118    cout << "Husband Name: " << woman3._HusbandName << endl;
119    cout << "Is The Husband Alive?: " << (woman::ISHusbandAlive ? "Yes alhamdalalah" : "No, one of his wives killed him") << endl;
120
121    // Set information for woman4 and display her details
122    woman4.setWomanName("Khulood");
123    woman4.setWomanAge(25);
124    woman4.setCarModel("BMW");
125    woman4.print();
126
127    // Changing the static member 'ISHusbandAlive' to false (Husband is no longer alive).
128    woman::ISHusbandAlive = false;
129
130    // Displaying the information again after the change.
131    woman1.print();
132    woman2.print();
133    woman3.print();
134    woman4.print();
135 }

```



```
1 Explanation:  
2  
3 1- In the given code, we have a class woman representing different women with their information, including name, age, car model,  
4 and a static member representing the husband's name and another static member indicating whether the husband is alive or not.  
5  
6 2- The woman class has several member functions for setting and getting information, as well as a print() method to display the w  
oman's information.  
7  
8 3- The static members _HusbandName and ISHusbandAlive are declared as const static string and static bool, respectively.  
9 The const keyword indicates that _HusbandName is constant and cannot be modified, while static means these members are shared amo  
ng all instances of the woman class.  
10  
11 4- In the main() function, we create four instances of the woman class, named woman1, woman2, woman3, and woman4.  
12  
13 5- We set the information for each woman using the setter methods (setWomanName(), setWomanAge(), and setCarModel()),  
14 and then call the print() method to display their information, including the husband's name and whether the husband is alive.  
15  
16 6- We can access the static members (_HusbandName and ISHusbandAlive) directly through the class name (woman::), as demonstrated  
in the print() method.  
17 Alternatively, we can access static members through any instance of the class (woman2._HusbandName)  
18 or directly using the class name without an instance (woman::ISHusbandAlive).  
19  
20 7- After setting the ISHusbandAlive static member to false, it reflects in all instances of the woman class,  
21 and the print() method shows that the husband is no longer alive for all instances.  
22  
23 8- The code uses properties to allow direct access to member variables (Name, Age, CarModel)  
24 as if they were regular variables, making the code more concise and readable.  
25  
26 Overall, the example demonstrates the usage of static members in a class, how they are shared among instances,  
27 and how they can be accessed directly through the class name or through an instance of the class.  
28 The use of properties simplifies the code by providing easy access to member variables.  
29 */
```



```
1 // Constructor and Destructor Example
2
3 #include <iostream>
4 using namespace std;
5
6 class Rectangle {
7 private:
8     int* _Length; // Pointer to store the length of the rectangle
9     int* _Width; // Pointer to store the width of the rectangle
10
11 public:
12     // Constructor for the Rectangle class that takes the length and width as arguments
13     Rectangle(int length, int width) {
14         _Length = new int; // Dynamically allocate memory for _Length on the heap
15         _Width = new int; // Dynamically allocate memory for _Width on the heap
16
17         *_Length = length; // Assign the value of 'length' to the memory location pointed by _Length
18         *_Width = width; // Assign the value of 'width' to the memory location pointed by _Width
19         cout << "Hi I'm Constructor Object has Created.\n"; // Display a message indicating that the object has been created
20     }
21
22     // Method to calculate the area of the rectangle
23     double getArea() {
24         return (*_Length) * (*_Width); // Calculate and return the area by dereferencing _Length and _Width pointers
25     }
26
27     // Destructor for the Rectangle class
28     ~Rectangle() {
29         delete _Length; // Deallocate the memory used for _Length
30         delete _Width; // Deallocate the memory used for _Width
31         cout << "Hi I'm Destructor Object has destroyed.\n"; // Display a message indicating that the object has been destroyed
32     }
33 };
34
35 int main() {
36     // Create an instance of the Rectangle class named 'rec' with length 4 and width 7
37     Rectangle rec(4, 7);
38
39     // Calculate the area of the rectangle using the getArea() method and display the result
40     cout << "Area: " << rec.getArea() << endl;
41 }
```

```
1 #include <iostream>
2 using namespace std;
3
4 // Definition of the Person class
5 class Person {
6 private:
7     int _ID;           // Private member variable to store the person's ID
8     string _FirstName; // Private member variable to store the person's first name
9     string _LastName;  // Private member variable to store the person's last name
10    string _Email;     // Private member variable to store the person's email
11    string _Phone;     // Private member variable to store the person's phone number
12
13 public:
14     // Constructor to initialize the Person object with provided information
15     Person(int id, string firstName, string lastName, string email, string phone) {
16         _ID = id;
17         _FirstName = firstName;
18         _LastName = lastName;
19         _Email = email;
20         _Phone = phone;
21     }
22
23     // Setter and Getter methods for modifying and accessing the first name
24     void SetFirstName(string fName) {
25         _FirstName = fName;
26     }
27     string GetFirstName() {
28         return _FirstName;
29     }
30
31     // Setter and Getter methods for modifying and accessing the last name
32     void SetLastName(string lName) {
33         _LastName = lName;
34     }
35     string GetLastName() {
36         return _LastName;
37     }
38
39     // Setter and Getter methods for modifying and accessing the email
40     void SetEmail(string email) {
41         _Email = email;
42     }
43     string GetEmail() {
44         return _Email;
45     }
46
47     // Method to get the full name of the person by concatenating first and last name
48     string GetFullName() {
49         return _FirstName + " " + _LastName;
50     }
51
52     // Getter method to access the person's ID
53     int GetID() {
54         return _ID;
55     }
56
57     // Setter and Getter methods for modifying and accessing the phone number
58     void SetPhone(string phone) {
59         _Phone = phone;
60     }
61     string GetPhone() {
62         return _Phone;
63     }
64
65     // Method to print the person's information
66     void printInfo() {
67         cout << "Info:\n";
68         cout << "=====*\n";
69         cout << "FirstName: " << _FirstName << endl;
70         cout << "LastName: " << _LastName << endl;
71         cout << "FullName: " << GetFullName() << endl;
72         cout << "Email: " << _Email << endl;
73         cout << "Phone: " << _Phone << endl;
74         cout << "=====*\n";
75     }
76
77     // Method to send an email message
78     void SendMessage(string Subject, string Body) {
79         cout << "The Following Message Sent Successfully To Email: " << _Email << endl;
80         cout << "Subject: " << Subject << "\n";
81         cout << "Body: " << Body << endl;
82     }
83
84     // Method to send an SMS message
85     void SendSMS(string Message) {
86         cout << "The Following SMS Sent Successfully To Phone: " << _Phone << endl;
87         cout << Message << "\n";
88     }
89 };
90
91 // Main function
92 int main() {
93     // Create an instance of the Person class named 'per1' with provided information
94     Person per1(10, "Mahmoud", "Mattar", "Mahmoud@gmail.com", "01018040635");
95
96     // Call the printInfo() method to display the person's information
97     per1.printInfo();
98
99     // Call the SendMessage() method to send an email message
100    per1.SendMessage("Hi", "How Are You?");
101
102    // Call the SendSMS() method to send an SMS message
103    per1.SendSMS("How Are You?");
104
105    // Pause the program to view the output
106    system("pause > nul");
107 }
```

## Encapsulation:

Encapsulation is one of the four fundamental principles of object-oriented programming (OOP). It refers to the bundling of data (attributes) and methods (functions or procedures) that operate on that data within a single unit called a class. The main idea behind encapsulation is to hide the internal implementation details of an object and provide a clean and controlled interface for interacting with that object.

In encapsulation, the internal data (attributes) of a class is kept private, meaning it cannot be accessed directly from outside the class. Instead, access to the data is controlled through public methods (also known as accessors and mutators) provided by the class. These methods allow other parts of the program to read and modify the data in a controlled manner, ensuring that the data remains consistent and valid.

Encapsulation offers several benefits in OOP, including:

1. Data Hiding: Encapsulation allows the internal data of a class to be hidden from the outside world, preventing unauthorized access and manipulation. This helps in maintaining the integrity of the data and prevents potential bugs and errors.
2. Abstraction: Encapsulation provides a higher level of abstraction by exposing only the essential features and behaviors of an object, while hiding the implementation details. This makes it easier for other parts of the program to interact with the object without needing to know its internal workings.
3. Modularity: Encapsulation allows the implementation details of a class to be isolated and self-contained. This promotes modularity, making it easier to maintain and update the codebase without affecting other parts of the program.
4. Code Reusability: By providing a well-defined interface for interacting with objects, encapsulation promotes code reusability. Once a class is encapsulated, it can be easily used in different parts of the program without the need to rewrite or modify its internal code.

In summary, encapsulation is a fundamental concept in OOP that allows for better organization, security, and abstraction of code. By encapsulating the internal data and methods of a class, it ensures that the class can be used effectively and safely in various parts of the program, promoting good software design and development practices.

1. Encapsulation: Encapsulation is the process of bundling data (attributes) and methods (functions) that operate on that data within a single unit, known as a class. The class hides the internal implementation details of its data and methods from the outside, providing only the essential interface for interacting with the object. This helps achieve data protection and information hiding, preventing unauthorized access and modification of the object's state.

2. Abstraction: Abstraction allows programmers to represent real-world objects in their software by focusing on the essential characteristics and behaviors, while omitting unnecessary details. It involves defining a class with its attributes and methods, creating a blueprint for objects. Abstraction allows for the creation of complex systems by breaking them down into manageable, abstract classes, making the code more modular, maintainable, and reusable.

3. Inheritance: Inheritance is a mechanism that allows a class (called the subclass or derived class) to inherit properties and behaviors from another class (called the superclass or base class). The subclass can extend or modify the attributes and methods inherited from the superclass and also add its own unique attributes and methods. Inheritance promotes code reuse, as common features can be defined in a base class and shared among multiple subclasses.

4. Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface (method or operator) to be used to represent different data types or classes. There are two types of polymorphism: compile-time (static) polymorphism and runtime (dynamic) polymorphism. Compile-time polymorphism is achieved through function overloading and operator overloading, where different methods or operators can have the same name but different parameter lists. Runtime polymorphism is achieved through virtual functions, allowing the correct method to be called at runtime based on the object's actual type.

These four principles form the foundation of OOP and help in creating well-structured, modular, and maintainable code. They provide a powerful paradigm for modeling real-world entities, promoting code reuse, and enhancing software design and development.

1- Encapsulation: Encapsulation is like putting all the related information and actions about something into a box, so it's protected and hidden from the outside world. It helps keep data safe and organized, and only shows the necessary things others need to interact with.

2- Abstraction: Abstraction is like making a simple model of something complex, focusing only on the important parts. It lets you use a basic outline to represent real things and ignore the small details. It makes your code easier to understand and manage.

3- Inheritance: Inheritance is like passing down traits from parents to children. In OOP, it means creating new classes that can reuse the properties and behaviors of existing classes. It helps save time by using what already exists and building upon it.

4- Polymorphism: Polymorphism is like using the same word for different meanings. In OOP, it means one method can work with different types of objects. It makes your code more flexible and allows you to handle different objects in a unified way.

These principles are the building blocks of OOP and help you create organized, flexible, and easy-to-understand programs. They make programming more approachable and efficient by using real-life concepts to model software.



```

1 #include <iostream>
2 using namespace std;
3
4 class clsEmployee
5 {
6 private:
7     int _ID;
8     string _FirstName;
9     string _LastName;
10    string _Title;
11    string _Email;
12    string _Phone;
13    string _Department;
14    double _Salary;
15
16 public:
17     clsEmployee(int id, string fName, string lName, string title, string email, string phone, double salary, string department)
18     {
19         _ID = id;
20         _FirstName = fName;
21         _LastName = lName;
22         _Title = title;
23         _Email = email;
24         _Phone = phone;
25         _Salary = salary;
26         _Department = department;
27     }
28
29     // Getter method to access the person's ID
30     int GetID() {
31         return _ID;
32     }
33
34     // Setter and Getter methods for modifying and accessing the first name
35     void SetFirstName(string fName) {
36         _FirstName = fName;
37     }
38     string GetFirstName() {
39         return _FirstName;
40     }
41
42     // Setter and Getter methods for modifying and accessing the last name
43     void SetLastName(string lName) {
44         _LastName = lName;
45     }
46     string GetLastName() {
47         return _LastName;
48     }
49
50     // Setter and Getter methods for modifying and accessing the title
51     void SetTitle(string title) {
52         _Title = title;
53     }
54     string GetTitle() {
55         return _Title;
56     }
57
58     // Setter and Getter methods for modifying and accessing the email
59     void SetEmail(string email) {
60         _Email = email;
61     }
62     string GetEmail() {
63         return _Email;
64     }
65
66     // Setter and Getter methods for modifying and accessing the phone number
67     void SetPhone(string phone) {
68         _Phone = phone;
69     }
70     string GetPhone() {
71         return _Phone;
72     }
73
74     // Setter and Getter methods for modifying and accessing the salary
75     void SetSalary(double salary) {
76         _Salary = salary;
77     }
78     double GetSalary() {
79         return _Salary;
80     }
81
82     // Setter and Getter methods for modifying and accessing the Department
83     void SetDepartment(string department) {
84         _Department = department;
85     }
86     string GetDepartment() {
87         return _Department;
88     }
89
90     // Method to get the full name of the person by concatenating first and last name
91     string GetFullName() {
92         return _FirstName + " " + _LastName;
93     }
94
95     // Method to send an email message
96     void SendMessage(string Subject, string Body) {
97         cout << "The Following Message Sent Successfully To Email: " << _Email << endl;
98         cout << "Subject: " << Subject << "\n";
99         cout << "Body: " << Body << endl;
100    }
101
102    // Method to send an SMS message
103    void SendSMS(string TextMessage) {
104        cout << "The Following SMS Sent Successfully To Phone: " << _Phone << endl;
105        cout << TextMessage << "\n";
106    }
107
108    // Method to print the Employee's information
109    void printInfo() {
110        cout << "Info:\n";
111        cout << "=====\n";
112        cout << "ID: " << _ID << endl;
113        cout << "FirstName: " << _FirstName << endl;
114        cout << "LastName: " << _LastName << endl;
115        cout << "FullName: " << GetFullName() << endl;
116        cout << "Title: " << _Title << endl;
117        cout << "Salary: $" << _Salary << endl;
118        cout << "Department: " << _Department << endl;
119        cout << "Email: " << _Email << endl;
120        cout << "Phone: " << _Phone << endl;
121        cout << "=====\n";
122    }
123 };
124
125 int main()
126 {
127     // Create an instance of the clsEmployee class named 'emp1' with provided information
128     clsEmployee emp1(10, "Mahmoud", "Matar", "Software Engineer", "mahmoud@MTTrojan.com", "0101406", 15000, "Development");
129
130     // Call the printInfo() method to display the Employee's information
131     emp1.printInfo();
132
133     // Call the SendMessage() method to send an email message
134     emp1.SendMessage("Hi", "How Are You?");
135
136     // Call the SendSMS() method to send an SMS message
137     emp1.SendSMS("How Are You?");
138
139     // Pause the program to view the output
140     system("pause > nul");
141
142 }

```



```
1 #include <iostream>
2 #include <string>
3
4 class clsEmployee {
5 private:
6     int _ID = 0;
7     std::string _FirstName = "";
8     std::string _LastName = "";
9     std::string _Title = "";
10    std::string _Email = "";
11    std::string _Phone = "";
12    std::string _Department = "";
13    double _Salary = 0.0;
14
15 public:
16     clsEmployee() = default; // Default constructor
17
18     clsEmployee(int id, std::string fName, std::string lName, std::string title,
19                 std::string email, std::string phone, double salary, std::string department)
20         : _ID(id), _FirstName(std::move(fName)), _LastName(std::move(lName)),
21           _Title(std::move(title)), _Email(std::move(email)), _Phone(std::move(phone)),
22           _Salary(salary), _Department(std::move(department)) {}
23
24     // Getter methods are made const to ensure they don't modify the object's state
25     int GetID() const {
26         return _ID;
27     }
28
29     std::string GetFirstName() const {
30         return _FirstName;
31     }
32
33     std::string GetLastName() const {
34         return _LastName;
35     }
36
37     std::string GetTitle() const {
38         return _Title;
39     }
40
41     std::string GetEmail() const {
42         return _Email;
43     }
44
45     std::string GetPhone() const {
46         return _Phone;
47     }
48
49     double GetSalary() const {
50         return _Salary;
51     }
52
53     std::string GetDepartment() const {
54         return _Department;
55     }
56
57     std::string GetFullName() const {
58         return _FirstName + " " + _LastName;
59     }
60
61     void SendMessage(const std::string& subject, const std::string& body) const {
62         std::cout << "The Following Message Sent Successfully To Email: " << _Email << std::endl;
63         std::cout << "Subject: " << subject << std::endl;
64         std::cout << "Body: " << body << std::endl;
65     }
66
67     void SendSMS(const std::string& textMessage) const {
68         std::cout << "The Following SMS Sent Successfully To Phone: " << _Phone << std::endl;
69         std::cout << textMessage << std::endl;
70     }
71
72     void printInfo() const {
73         std::cout << "Info:" << std::endl;
74         std::cout << "======" << std::endl;
75         std::cout << "ID: " << _ID << std::endl;
76         std::cout << "FirstName: " << _FirstName << std::endl;
77         std::cout << "LastName: " << _LastName << std::endl;
78         std::cout << "FullName: " << GetFullName() << std::endl;
79         std::cout << "Title: " << _Title << std::endl;
80         std::cout << "Salary: $" << _Salary << std::endl;
81         std::cout << "Department: " << _Department << std::endl;
82         std::cout << "Email: " << _Email << std::endl;
83         std::cout << "Phone: " << _Phone << std::endl;
84         std::cout << "======" << std::endl;
85     }
86 };
87
88 int main() {
89     // Create an instance of the clsEmployee class named 'emp1' with provided information
90     clsEmployee emp1(10, "Mahmoud", "Matar", "Software Engineer", "mahmoud@M7Trojan.com",
91                      "0101406", 15000, "Development");
92
93     // Call the printInfo() method to display the Employee's information
94     emp1.printInfo();
95
96     // Call the SendMessage() method to send an email message
97     emp1.SendMessage("Hi", "How Are You?");
98
99     // Call the SendSMS() method to send an SMS message
100    emp1.SendSMS("How Are You?");
101
102    // Pause the program to view the output
103    system("pause > nul");
104 }
```



# Last Homework - Employee



## Person:

ID  
First name  
Last name  
FullName()  
Email  
Phone  
SendEmail(..)  
SendSMS(...)  
Print()

## Employee:

ID  
First name  
Last name  
FullName()  
**Title**  
Email  
Phone  
**Salary**  
**Department**  
SendEmail(..)  
SendSMS(...)  
Print()

# Last Homework - Employee



Super Class/Base Class



**Person:**

- ID
- First name
- Last name
- FullName()
- Email
- Phone
- SendEmail(..)
- SendSMS(...)
- Print()

Sub Class/Derived Class

**Employee:**

- Title
- Salary
- Department



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.

correct

True



False



The class that inherits properties from another class is called Subclass or  
Derived Class

correct

True



False

The class whose properties are inherited by a subclass is called Base Class or Superclass.

correct

True



Flase

# Derived Class and Sub Class are the same.

correct

True



False

# Base Class and Super Class are the same

correct

True



False

You can inherit only public and protected members, private members  
are not inherited.

correct

True



False

```

1 #include <iostream>
2 using namespace std;
3
4 // Definition of the Person class
5 class clsPerson {
6 private:
7     int _ID;           // Private member variable to store the person's ID
8     string _firstName; // Private member variable to store the person's first name
9     string _lastName;  // Private member variable to store the person's last name
10    string _Email;    // Private member variable to store the person's email
11    string _Phone;    // Private member variable to store the person's phone number
12
13 public:
14
15     // default Constructor
16     // we put the default constructor here temporarily because inheritance will use it,
17     // in the coming lectures, we will solve the parameterized constructor with inheritance.
18     clsPerson(){}
19
20     // Constructor to initialize the Person object with provided information
21     clsPerson(int id, string firstName, string lastName, string email, string phone) {
22         _ID = id;
23         _FirstName = firstName;
24         _LastName = lastName;
25         _Email = email;
26         _Phone = phone;
27     }
28
29     // Setter and Getter methods for modifying and accessing the first name
30     void SetFirstName(string fName) {
31         _FirstName = fName;
32     }
33     string GetFirstName() {
34         return _FirstName;
35     }
36
37     // Setter and Getter methods for modifying and accessing the last name
38     void SetLastName(string lName) {
39         _LastName = lName;
40     }
41     string GetLastName() {
42         return _LastName;
43     }
44
45     // Setter and Getter methods for modifying and accessing the email
46     void SetEmail(string email) {
47         _Email = email;
48     }
49     string GetEmail() {
50         return _Email;
51     }
52
53     // Method to get the full name of the person by concatenating first and last name
54     string GetFullName() {
55         return _FirstName + " " + _LastName;
56     }
57
58     // Getter method to access the person's ID (Read Only Property)
59     int GetID() {
60         return _ID;
61     }
62
63     // Setter and Getter methods for modifying and accessing the phone number
64     void SetPhone(string phone) {
65         _Phone = phone;
66     }
67     string GetPhone() {
68         return _Phone;
69     }
70
71     // Method to print the person's information
72     void printInfo() {
73         cout << "Info:\n";
74         cout << "=====*\n";
75         cout << "ID: " << _ID << endl;
76         cout << "FirstName: " << _FirstName << endl;
77         cout << "LastName: " << _LastName << endl;
78         cout << "FullName: " << GetFullName() << endl;
79         cout << "Email: " << _Email << endl;
80         cout << "Phone: " << _Phone << endl;
81         cout << "=====*\n";
82     }
83
84     // Method to send an email message
85     void SendEmail(string Subject, string Body) {
86         cout << "The Following Message Sent Successfully To Email: " << _Email << endl;
87         cout << "Subject: " << Subject << "\n";
88         cout << "Body: " << Body << endl;
89     }
90
91     // Method to send an SMS message
92     void SendsSMS(string TextMessage) {
93         cout << "The Following SMS Sent Successfully To Phone: " << _Phone << endl;
94         cout << TextMessage << "\n";
95     }
96 };
97 */
98 /* Sub Class/Derived Class Super Class/Base Class */
99 class clsEmployee : public clsPerson{
100
101    double _Salary;
102    string _Department;
103    string _Title;
104
105 public:
106     // Setter and Getter methods for modifying and accessing the salary
107     void SetSalary(double salary)
108     {
109         _Salary = salary;
110     }
111     double GetSalary()
112     {
113         return _Salary;
114     }
115
116     // Setter and Getter methods for modifying and accessing the Department
117     void SetDepartment(string department)
118     {
119         _Department = department;
120     }
121     string GetDepartment()
122     {
123         return _Department;
124     }
125
126     // Setter and Getter methods for modifying and accessing the title
127     void SetTitle(string title)
128     {
129         _Title = title;
130     }
131     string GetTitle()
132     {
133         return _Title;
134     }
135 };
136
137 int main()
138 {
139
140     clsEmployee employee;
141     employee.SetFirstName("Mahmoud");
142     employee.SetLastName("Mattar");
143     employee.setEmail("matter@gmail.com");
144     employee.setPhone("01045236");
145     employee.printInfo();
146     employee.SendEmail("Hi", "How are you?");
147     employee.SendsSMS("Hi");
148
149     employee.setTitle("Software Engineer");
150     employee.setDepartment("Development");
151     employee.setSalary(15000);
152
153     cout << "Salary: " << employee.GetSalary() << "\n";
154     cout << "Title: " << employee.GetTitle() << "\n";
155     cout << "Department: " << employee.GetDepartment() << "\n";
156
157     // Calling the print will not print anything from derived class, only base class
158     // therefore the print method will not serve me here, this is a problem will be solved in the next lecture.
159     employee.printInfo();
160
161     system("pause > nul");
162
163 }
164
165 // derived => مُنْسَلِّط
166 // derived class => فَرَعَةٌ لِّلَّا
167 // sub => فَرِعَةٌ
168 // Sub Class => فَرَعَةٌ فَرِعَةٌ
169
170 // Base => اسْبَاسٌ / اسْبَاسٌ
171 // Base Class => اسْبَاسٌ اسْبَاسٌ
172

```

```
/*
```

## Explanation of Inheritance:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (known as the derived class) to inherit properties and behaviors from another class (known as the base class). The derived class can then extend or override the functionality of the base class.

In the provided code, we have two classes: `clsPerson` and `clsEmployee`. `clsEmployee` is an example of inheritance as it inherits from the `clsPerson` class. This means that the `clsEmployee` class inherits all the attributes and member functions of the `clsPerson` class, such as `ID`, `FirstName`, `LastName`, `Email`, `Phone`, `Print()`, `SendEmail()`, and `SendSMS()`.

The syntax to define inheritance is:

```
class DerivedClassName : public BaseClassName
{
    // Derived class members and methods
};
```

In this code, `clsEmployee` inherits from `clsPerson` using public inheritance. This means that all public members of `clsPerson` are accessible as public members of `clsEmployee`.

The `clsEmployee` class introduces additional attributes specific to employees, such as `_Title`, `_Department`, and `_Salary`. These attributes are unique to employees and are not part of the base class `clsPerson`.

By using inheritance, we avoid code duplication, as the common attributes and behaviors are defined in the base class, and the derived class only needs to focus on its specific attributes and behaviors.

In the `main()` function, we create an instance of `clsEmployee` named `Employee1`. Since `clsEmployee` is derived from `clsPerson`, we can access the attributes and methods of both classes using `Employee1`.

For example, we set the first name, last name, and email for `Employee1` using the `setFirstName()`, `setLastName()`, and `setEmail()` methods inherited from the `clsPerson` class. We also call the `Print()` method, which is a member function of `clsPerson`, to display the person's information.

Additionally, we call the `setSalary()` and `Salary()` methods specific to the `clsEmployee` class to set and retrieve the employee's salary.

Please note that in this example, the `Print()` method only displays the information from the `clsPerson` class because we have not overridden it in the `clsEmployee` class. In the next lecture, the code will be improved to handle this limitation and show information from both the base and derived classes.

```
*/
```

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

// Define the clsPerson class
class clsPerson
{
    // ... (Rest of the code for clsPerson class)
};

// Define the clsEmployee class inheriting from clsPerson
class clsEmployee : public clsPerson
{
    // ... (Rest of the code for clsEmployee class)
```

};

```
// The main() function
int main()
{
    // ... (Rest of the code in the main function)
    return 0;
}
```



```
1 In C++, when you initialize a member variable in a constructor, you have different options to pass the value. Let's compare the two options you mentioned:  
2  
3 1. `FirstName(std::move(fName))`:  
4 Here, `std::move(fName)` is used to perform a move operation on the `fName` object.  
5 The `std::move` function is part of the C++ Standard Library and is used to transfer the ownership of a resource from one object to another.  
6 By using `std::move`, we are essentially telling the compiler that we no longer need the value of `fName` after this point,  
7 and it's safe to move its contents to `_FirstName`. This can be more efficient than copying the contents if `fName` contains a resource that is expensive to copy,  
8 such as a large string or a dynamically allocated memory block.  
9  
10 2. `FirstName(fName)`:  
11 Here, we are simply copying the contents of `fName` to `_FirstName`.  
12 This is the regular copy operation, where the value of `fName` is duplicated, and both `fName` and `_FirstName` will have their own separate copies of the string.  
13 This is the default behavior when you pass a parameter to a constructor or any other function in C++.  
14  
15 In summary, the main difference between the two is the method of assignment:  
16  
17 - `FirstName(std::move(fName))`:  
18 Moves the contents of `fName` to `_FirstName`, and `fName` becomes empty after the move.  
19 This is useful for optimizing expensive copies when the original value of `fName` is no longer needed.  
20  
21 - `FirstName(fName)`:  
22 Performs a regular copy of the contents of `fName` to `_FirstName`. Both `fName` and `_FirstName` will have their own separate copies of the data.  
23  
24 The choice between the two depends on whether you want to move or copy the data and whether you need to keep the original value of `fName` after the assignment.  
25 If you no longer need the original value of `fName`, using `std::move` can be more efficient. Otherwise, a regular copy should be sufficient.  
26
```

In C++, when you initialize a member variable in a constructor, you have different options to pass the value. Let's compare the two options you mentioned:

#### 1. 'FirstName(std::move(fName))':

Here, 'std::move(fName)' is used to perform a move operation on the 'fName' object. The 'std::move' function is part of the C++ Standard Library and is used to transfer the ownership of a resource from one object to another. By using 'std::move', we are essentially telling the compiler that we no longer need the value of 'fName' after this point, and it's safe to move its contents to '\_FirstName'. This can be more efficient than copying the contents if 'fName' contains a resource that is expensive to copy, such as a large string or a dynamically allocated memory block.

#### 2. 'FirstName(fName)':

Here, we are simply copying the contents of 'fName' to '\_FirstName'. This is the regular copy operation, where the value of 'fName' is duplicated, and both 'fName' and '\_FirstName' will have their own separate copies of the string. This is the default behavior when you pass a parameter to a constructor or any other function in C++.

In summary, the main difference between the two is the method of assignment:

- 'FirstName(std::move(fName))': Moves the contents of 'fName' to 'FirstName', and 'fName' becomes empty after the move. This is useful for optimizing expensive copies when the original value of 'fName' is no longer needed.
- '\_FirstName(fName)': Performs a regular copy of the contents of 'fName' to '\_FirstName'. Both 'fName' and '\_FirstName' will have their own separate copies of the data.

The choice between the two depends on whether you want to move or copy the data and whether you need to keep the original value of 'fName' after the assignment. If you no longer need the original value of 'fName', using 'std::move' can be more efficient. Otherwise, a regular copy should be sufficient.

```

1 #include <iostream>
2 using namespace std;
3 class clsPerson
4 {
5 private:
6     int _ID;           // Private member variable to store the person's ID
7     string _FirstName; // Private member variable to store the person's first name
8     string _LastName;  // Private member variable to store the person's last name
9     string _Email;    // Private member variable to store the person's email
10    string _Phone;   // Private member variable to store the person's phone number
11
12 public:
13     // default Constructor
14     // we put the default constructor here temporarily because inheritance will use it,
15     // in the coming lectures, we will solve the parameterized constructor with inheritance.
16     clsPerson() {}
17
18     // Constructor to initialize the Person object with provided information
19     clsPerson(int id, string firstName, string lastName, string email, string phone)
20     {
21         _ID = id;
22         _FirstName = firstName;
23         _LastName = lastName;
24         _Email = email;
25         _Phone = phone;
26     }
27
28     // Setter and Getter methods for modifying and accessing the first name
29     void SetFirstName(string fName)
30     {
31         _FirstName = fName;
32     }
33     string GetFirstName()
34     {
35         return _FirstName;
36     }
37
38     // Setter and Getter methods for modifying and accessing the last name
39     void SetLastName(string lName)
40     {
41         _LastName = lName;
42     }
43     string GetLastName()
44     {
45         return _LastName;
46     }
47
48     // Setter and Getter methods for modifying and accessing the email
49     void SetEmail(string email)
50     {
51         _Email = email;
52     }
53     string GetEmail()
54     {
55         return _Email;
56     }
57
58     // Method to get the full name of the person by concatenating first and last name
59     string GetFullName()
60     {
61         return _FirstName + " " + _LastName;
62     }
63
64     // Getter method to access the person's ID (Read Only Property)
65     int GetID()
66     {
67         return _ID;
68     }
69
70     // Setter and Getter methods for modifying and accessing the phone number
71     void SetPhone(string phone)
72     {
73         _Phone = phone;
74     }
75     string GetPhone()
76     {
77         return _Phone;
78     }
79
80     // Method to print the person's information
81     void printInfo()
82     {
83         cout << "Info:\n";
84         cout << "=====\n";
85         cout << "ID: " << _ID << endl;
86         cout << "FirstName: " << _FirstName << endl;
87         cout << "LastName: " << _LastName << endl;
88         cout << "FullName: " << GetFullName() << endl;
89         cout << "Email: " << _Email << endl;
90         cout << "Phone: " << _Phone << endl;
91         cout << "=====\n";
92     }
93
94     // Method to send an email message
95     void SendEmail(string Subject, string Body)
96     {
97         cout << "The Following Message Sent Successfully To Email: " << _Email << endl;
98         cout << "Subject: " << Subject << "\n";
99         cout << "Body: " << Body << endl;
100    }
101
102    // Method to send an SMS message
103    void SendSMS(string TextMessage)
104    {
105        cout << "The Following SMS Sent Successfully To Phone: " << _Phone << endl;
106        cout << TextMessage << "\n";
107    }
108 };
109
110 class clsEmployee : public clsPerson
111 {
112 private:
113     double _Salary;
114     string _Department;
115     string _Title;
116
117 public:
118     // Constructor for clsEmployee class
119     // The constructor takes all the parameters for the clsPerson class (id, fName, lName, email, phone),
120     // and additional parameters for the clsEmployee class (title, department, salary).
121     // The constructor uses the member initializer list to initialize the base class (clsPerson) and derived class (clsEmployee)
122     // members.
123     clsEmployee(int id, string fName, string lName, string email, string phone, string title, string department, double salary)
124     : clsPerson(id, fName, lName, email, phone), // Call the base class constructor (clsPerson) with provided parameters.
125       _Salary(std::move(salary)),                // Move the value of 'salary' to '_Salary' member.
126       _Department(std::move(department)),          // Move the value of 'department' to '_Department' member.
127       _Title(std::move(title))                   // Move the value of 'title' to '_Title' member.
128     {
129         // The body of the constructor is empty, as all member variables are already initialized in the member initializer list.
130     }
131
132     // Setter and Getter methods for modifying and accessing the salary
133     void SetSalary(double salary)
134     {
135         _Salary = salary;
136     }
137     double GetSalary()
138     {
139         return _Salary;
140     }
141
142     // Setter and Getter methods for modifying and accessing the Department
143     void SetDepartment(string department)
144     {
145         _Department = department;
146     }
147     string GetDepartment()
148     {
149         return _Department;
150     }
151
152     // Setter and Getter methods for modifying and accessing the title
153     void SetTitle(string title)
154     {
155         _Title = title;
156     }
157     string GetTitle()
158     {
159         return _Title;
160     }
161
162 int main(void)
163 {
164     clsEmployee Employee1(10, "Mahmoud", "Mattar", "mattar@gmail.com", "0101906523", "Software Engineer", "Development", 15000);
165     Employee1.printInfo();
166
167     cout << "\nTitle: " << Employee1.GetTitle() << endl;
168     cout << "Department: " << Employee1.GetDepartment() << endl;
169     cout << "Salary: " << Employee1.GetSalary() << endl;
170 }

```



```
1 overriding → تجاوز
2 Function overriding is a concept in object-oriented programming (OOP) where a subclass provides a specific implementation for a method
3 that is already defined in its superclass. When a subclass defines a method with the same name and function signature
4 (i.e., the same return type and parameters) as a method in its superclass, it overrides the behavior of the superclass method.
5
6 In other words, function overriding allows a subclass to provide its own specialized implementation of a method, which is different from the implementation in the superclass.
7 When an object of the subclass calls the overridden method, it will execute the subclass's implementation instead of the superclass's implementation.
8
9 Overriding is commonly used in inheritance scenarios, where a subclass inherits properties and methods from its superclass.
10 By overriding methods, the subclass can customize or extend the behavior of its parent class, while still retaining the other features of the superclass.
11
12 when we say "overriding something" in the context of function overriding, it does not necessarily mean canceling or deleting the original method.
13 Instead, it means providing a new implementation that takes precedence over the original implementation in certain situations.
14 The original method still exists in the superclass, but when called on an object of the subclass, the overridden method in the subclass will be executed instead.
15 */
```

```

1 #include <iostream>
2 using namespace std;
3 class clsPerson
4 {
5 private:
6     int _ID;           // Private member variable to store the person's ID
7     string _FirstName; // Private member variable to store the person's first name
8     string _LastName; // Private member variable to store the person's last name
9     string _Email;    // Private member variable to store the person's email
10    string _Phone;   // Private member variable to store the person's phone number
11
12 public:
13     // Default constructor
14     // we put the default constructor here temporarily because inheritance will use it,
15     // in the coming lectures, we will solve the parameterized constructor with inheritance.
16     clsPerson() {}
17
18     // Constructor to initialize the Person object with provided information
19     clsPerson(int id, string firstName, string lastName, string email, string phone)
20     {
21         _ID = id;
22         _FirstName = firstName;
23         _LastName = lastName;
24         _Email = email;
25         _Phone = phone;
26     }
27
28     // Setter and Getter methods for modifying and accessing the first name
29     void SetFirstName(string fName)
30     {
31         _FirstName = fName;
32     }
33     string GetFirstName()
34     {
35         return _FirstName;
36     }
37
38     // Setter and Getter methods for modifying and accessing the last name
39     void SetLastName(string lName)
40     {
41         _LastName = lName;
42     }
43     string GetLastName()
44     {
45         return _LastName;
46     }
47
48     // Setter and Getter methods for modifying and accessing the email
49     void SetEmail(string email)
50     {
51         _Email = email;
52     }
53     string GetEmail()
54     {
55         return _Email;
56     }
57
58     // Method to get the full name of the person by concatenating first and last name
59     string GetFullName()
60     {
61         return _FirstName + " " + _LastName;
62     }
63
64     // Getter method to access the person's ID (Read Only Property)
65     int GetID()
66     {
67         return _ID;
68     }
69
70     // Setter and Getter methods for modifying and accessing the phone number
71     void SetPhone(string phone)
72     {
73         _Phone = phone;
74     }
75     string GetPhone()
76     {
77         return _Phone;
78     }
79
80     // Method to print the person's information
81     void printInfo()
82     {
83         cout << "Info:\n";
84         cout << "=====\\n";
85         cout << "ID: " << _ID << endl;
86         cout << "FirstName: " << _FirstName << endl;
87         cout << "LastName: " << _LastName << endl;
88         cout << "FullName: " << GetFullName() << endl;
89         cout << "Email: " << _Email << endl;
90         cout << "Phone: " << _Phone << endl;
91         cout << "=====\\n";
92     }
93
94     // Method to send an email message
95     void SendEmail(string Subject, string Body)
96     {
97         cout << "The Following Message Sent Successfully To Email: " << _Email << endl;
98         cout << "Subject: " << Subject << "\\n";
99         cout << "Body: " << Body << endl;
100    }
101
102    // Method to send an SMS message
103    void SendSMS(string TextMessage)
104    {
105        cout << "The Following SMS Sent Successfully To Phone: " << _Phone << endl;
106        cout << TextMessage << "\\n";
107    }
108 };
109
110 class clsEmployee : public clsPerson
111 {
112 private:
113     double _Salary;
114     string _Department;
115     string _Title;
116
117 public:
118     // Constructor for clsEmployee class
119     // The constructor takes all the parameters for the clsPerson class (id, fName, lName, email, phone),
120     // and additional parameters for the clsEmployee class (title, department, salary).
121     // The constructor uses the member initializer list to initialize the base class (clsPerson) and derived class (clsEmployee)
122     // members.
123     clsEmployee(int id, string fName, string lName, string email, string phone, string title, string department, double salary)
124     : clsPerson(id, fName, lName, email, phone), // Call the base class constructor (clsPerson) with provided parameters.
125       _Salary(std::move(salary)), // Move the value of 'salary' to '_Salary' member.
126       _Department(std::move(department)), // Move the value of 'department' to '_Department' member.
127       _Title(std::move(title)) // Move the value of 'title' to '_Title' member.
128     {
129         // The body of the constructor is empty, as all member variables are already initialized in the member initializer list.
130     }
131
132     // Setter and Getter methods for modifying and accessing the salary
133     void SetSalary(double salary)
134     {
135         _Salary = salary;
136     }
137     double GetSalary()
138     {
139         return _Salary;
140     }
141
142     // Setter and Getter methods for modifying and accessing the Department
143     void SetDepartment(string department)
144     {
145         _Department = department;
146     }
147     string GetDepartment()
148     {
149         return _Department;
150     }
151
152     // Setter and Getter methods for modifying and accessing the title
153     void SetTitle(string title)
154     {
155         _Title = title;
156     }
157     string GetTitle()
158     {
159         return _Title;
160     }
161
162     // Method to print the Employee's information
163     // Override function
164     void printInfo()
165     {
166         cout << "Info:\\n";
167         cout << "=====\\n";
168         cout << "ID: " << GetID() << endl;
169         cout << "FirstName: " << GetFirstName() << endl;
170         cout << "LastName: " << GetLastName() << endl;
171         cout << "FullName: " << GetFullName() << endl;
172         cout << "Email: " << GetEmail() << endl;
173         cout << "Phone: " << GetPhone() << endl;
174         cout << "Title: " << GetTitle() << endl;
175         cout << "Department: " << GetDepartment() << endl;
176         cout << "Salary: " << GetSalary() << endl;
177         cout << "=====\\n";
178     }
179
180 int main(void)
181 {
182     clsEmployee Employee1(10, "Mahmoud", "Mattar", "mattar@gmail.com", "0101906523", "Software Engineer", "Development", 15000);
183     Employee1.printInfo();
184 }

```

**Function Overriding :** The function in derived class overrides the function in base class.

correct

True



False

If you override a function in base class will you be able to access this function from the object of derived class

incorrect

True



False

If you override a function in base class will you be able to access this function inside derived class?

correct

True



False

If you override a function in base how to access it from within the derived class?

BaseClassName.FunctionName()

correct

BaseClassName::FunctionName()



# Last Homework - Developer



Super Class/Base Class

## Person:

ID  
First name  
Last name  
FullName()  
Email  
Phone  
SendEmail(..)  
SendSMS(...)  
Print()

Sub Class/Derived Class

Super Class/Base Class

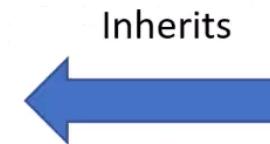
## Employee:

ID  
First name  
Last name  
FullName()  
**Title**  
Email  
Phone  
**Salary**  
**Department**  
SendEmail(..)  
SendSMS(...)  
Print()

Sub Class/Derived Class

## Developer:

ID  
First name  
Last name  
FullName()  
**Title**  
Email  
Phone  
**Salary**  
**Department**  
SendEmail(..)  
SendSMS(...)  
**MainProgrammingLanguage**  
Print()



Multi-Level Inheritance is a concept in object-oriented programming where a class inherits from another class, which in turn inherits from another class, forming a chain of inheritance relationships. In other words, a class serves as the base class for another class, and that derived class, in turn, serves as the base class for another class, and so on.

Here's an example to illustrate Multi-Level Inheritance:

cpp

Copy code

```
// Base class
class Animal {
public:
    void eat() {
        cout << "Animal is eating." << endl;
    }
};

// Derived class 1 (inherits from Animal)
class Mammal : public Animal {
public:
    void run() {
        cout << "Mammal is running." << endl;
    }
};

// Derived class 2 (inherits from Mammal)
class Dog : public Mammal {
public:
    void bark() {
        cout << "Dog is barking." << endl;
    }
};
```

Regenerate response

In this example, we have a base class `Animal`, which has a method `eat`. Then, we have a derived class `Mammal`, which inherits from `Animal` and adds a method `run`. Finally, we have another derived class `Dog`, which inherits from `Mammal` and adds a method `bark`.

With multi-level inheritance, the `Dog` class indirectly inherits the properties and methods of both `Mammal` and `Animal`. It forms a chain of inheritance relationships: `Dog` is a `Mammal`, and `Mammal` is an `Animal`.

Using multi-level inheritance, derived classes can build upon the functionality of their parent classes, creating a hierarchical structure that represents different levels of abstraction and specialization. However, it's essential to use inheritance judiciously and avoid deep inheritance hierarchies to maintain code clarity and avoid potential issues like the diamond problem.

```

1 // Multi level Inheritance means -> بخلاف الوراثة المقتضبة
2
3 #include <iostream>
4 using namespace std;
5 class clsPerson
6 {
7 private:
8     int _ID;           // Private member variable to store the person's ID
9     string _firstName; // Private member variable to store the person's first name
10    string _lastName; // Private member variable to store the person's last name
11    string _Email;    // Private member variable to store the person's email
12    string _Phone;    // Private member variable to store the person's phone number
13
14 public:
15     // default constructor
16     // we put the default constructor here temporarily because inheritance will use it,
17     // in the coming lectures, we will solve the parameterized constructor with inheritance.
18     clsPerson() {}
19
20     // Constructor to initialize the Person object with provided information
21     clsPerson(int id, string firstName, string lastName, string email, string phone)
22     {
23         _ID = id;
24         _FirstName = firstName;
25         _LastName = lastName;
26         _Email = email;
27         _Phone = phone;
28     }
29
30     // Setter and Getter methods for modifying and accessing the first name
31     void SetFirstName(string fName)
32     {
33         _FirstName = fName;
34     }
35     string GetFirstName()
36     {
37         return _FirstName;
38     }
39
40     // Setter and Getter methods for modifying and accessing the last name
41     void SetLastName(string lName)
42     {
43         _LastName = lName;
44     }
45     string GetLastName()
46     {
47         return _LastName;
48     }
49
50     // Setter and Getter methods for modifying and accessing the email
51     void SetEmail(string email)
52     {
53         _Email = email;
54     }
55     string GetEmail()
56     {
57         return _Email;
58     }
59
60     // Method to get the full name of the person by concatenating first and last name
61     string GetFullName()
62     {
63         return _FirstName + " " + _LastName;
64     }
65
66     // Getter method to access the person's ID (Read Only Property)
67     int GetID()
68     {
69         return _ID;
70     }
71
72     // Setter and Getter methods for modifying and accessing the phone number
73     void SetPhone(string phone)
74     {
75         _Phone = phone;
76     }
77     string GetPhone()
78     {
79         return _Phone;
80     }
81
82     // Method to print the person's information
83     void printInfo()
84     {
85         cout << "Info:\n";
86         cout << "*****\n";
87         cout << "ID: " << _ID << endl;
88         cout << "FirstName: " << _FirstName << endl;
89         cout << "LastName: " << _LastName << endl;
90         cout << "FullName: " << GetFullName() << endl;
91         cout << "Email: " << _Email << endl;
92         cout << "Phone: " << _Phone << endl;
93         cout << "*****\n";
94     }
95
96     // Method to send an email message
97     void SendEmail(string Subject, string Body)
98     {
99         cout << "The Following Message Sent Successfully To Email: " << _Email << endl;
100        cout << "Subject: " << Subject << "\n";
101        cout << "Body: " << Body << endl;
102    }
103
104     // Method to send an SMS message
105     void SendSMS(string TextMessage)
106     {
107         cout << "The Following SMS Sent Successfully To Phone: " << _Phone << endl;
108         cout << TextMessage << "\n";
109     }
110 };
111
112 class clsEmployee : public clsPerson
113 {
114 private:
115     double _Salary;
116     string _Department;
117     string _Title;
118
119 public:
120     // Constructor for clsEmployee class
121     // The constructor takes all the parameters for the clsPerson class (id, fName, lName, email, phone),
122     // and additional parameters for the clsEmployee class (title, department, salary).
123     // The constructor uses the member initializer list to initialize the base class (clsPerson) and derived class (clsEmployee)
124     // members.
125     clsEmployee(int id, string fName, string lName, string email, string phone, string title, string department, double salary)
126     : clsPerson(id, fName, lName, email, phone), // Call the base class constructor (clsPerson) with provided parameters.
127       _Salary(std::move(salary)), // Move the value of 'salary' to '_Salary' member.
128       _Department(std::move(department)), // Move the value of 'department' to '_Department' member.
129       _Title(std::move(title)) // Move the value of 'title' to '_Title' member.
130     {
131         // The body of the constructor is empty, as all member variables are already initialized in the member initializer list.
132     }
133
134     // Setter and Getter methods for modifying and accessing the salary
135     void SetSalary(double salary)
136     {
137         _Salary = salary;
138     }
139     double GetSalary()
140     {
141         return _Salary;
142     }
143
144     // Setter and Getter methods for modifying and accessing the Department
145     void SetDepartment(string department)
146     {
147         _Department = department;
148     }
149     string GetDepartment()
150     {
151         return _Department;
152     }
153
154     // Setter and Getter methods for modifying and accessing the title
155     void SetTitle(string title)
156     {
157         _Title = title;
158     }
159     string GetTitle()
160     {
161         return _Title;
162     }
163
164     // Method to print the Employee's information
165     // Override Function
166     void printInfo()
167     {
168         cout << "Info:\n";
169         cout << "*****\n";
170         cout << "ID : " << GetID() << endl;
171         cout << "FirstName : " << GetFirstName() << endl;
172         cout << "LastName : " << GetLastName() << endl;
173         cout << "FullName : " << GetFullName() << endl;
174         cout << "Title : " << GetTitle() << endl;
175         cout << "Email : " << GetEmail() << endl;
176         cout << "Phone : " << GetPhone() << endl;
177         cout << "Department : " << GetDepartment() << endl;
178         cout << "Salary : " << GetSalary() << endl;
179         cout << "*****\n";
180     }
181
182 class clsDeveloper : public clsEmployee{
183 private:
184     string _MainProgrammingLanguage;
185 public:
186     clsDeveloper(int id, string fName, string lName, string email, string phone, string title, string mainProgLang, double salary)
187     : clsEmployee(id, fName, lName, email, phone, title, "Development", salary),
188       _MainProgrammingLanguage = mainProgLang,
189     {
190
191         // Setter and Getter methods for modifying and accessing the MainProgramminglanguage
192         void SetMainProgrammingLanguage(string mainProgLang)
193         {
194             _MainProgrammingLanguage = mainProgLang;
195         }
196         string GetMainProgrammingLanguage()
197         {
198             return _MainProgrammingLanguage;
199         }
200
201         // Method to print the Developer's information
202         // Override function
203         void printInfo()
204         {
205             cout << "Info:\n";
206             cout << "*****\n";
207             cout << "ID : " << GetID() << endl;
208             cout << "FirstName : " << GetFirstName() << endl;
209             cout << "LastName : " << GetLastName() << endl;
210             cout << "FullName : " << GetFullName() << endl;
211             cout << "Department : " << GetDepartment() << endl;
212             cout << "Title : " << GetTitle() << endl;
213             cout << "Main Programming Language: " << _MainProgrammingLanguage << endl;
214             cout << "Salary : " << GetSalary() << endl;
215             cout << "Email : " << GetEmail() << endl;
216             cout << "Phone : " << GetPhone() << endl;
217             cout << "*****\n";
218         }
219     }
220
221     int main(void)
222     {
223         clsDeveloper dev1(10, "Mahmoud", "Mattar", "mattar@gmail.com", "0101986523", "Software Engineer", "c++", 15000);
224         dev1.printInfo();
225         dev1.SendEmail("Hi", "How are You");
226         dev1.SendSMS("Hi");
227
228         cout << dev1.GetMainProgrammingLanguage();
229     }
230 }
```



```
1 // Access_Specifiers Review
2
3 #include <iostream>
4 using namespace std;
5 class clsA
6 {
7 private:
8     //only accessible inside this class, neither derived classes nor outside class.
9     int _Var1;
10    void _Fun1()
11    {
12        cout << "Function 1";
13    }
14
15 protected:
16     //only accessible inside this class and all derived classes, but not outside class
17     int Var2;
18     void Fun2()
19     {
20         cout << "Function 1";
21     }
22
23 public:
24     // Accessible inside this class, all derived classes, and outside class
25     int Var3;
26     void Fun3()
27     {
28         cout << "Function 1";
29     }
30
31 };
32
33 class clsB : public clsA
34 {
35
36 public:
37
38     void Func1()
39     {
40         cout << clsA::Var2;
41     }
42 };
43
44 int main()
45
46 {
47     system("pause > nul");
48     return 0;
49 }
```

Access modifiers (or access specifiers) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members.

correct

True



False

# Which of the following is Access Specifiers/Modifiers:

correct

Public



correct

Private



correct

Protected



Constant

< Back

Continue >

Public Members can be accessed from inside and outside the class.

correct

True



False

Private Members can be accessed from outside the class through object.

True

correct

Flase



Private Members can be accessed by any class inherits the current class.

Ture

correct

False



Private Members can be accessed only from inside the class, it cannot be accessed from outside the class nor from the classes inherits the current class..

correct

True



Flase

Protected Members can be accessed from outside class through objects.

True

correct

False



Protected Members can be accessed from inside class and from all classes inherits the current class.

correct

Tue



False

OOP is more secured because you can hide members from developers.

correct

True



False

Inside the class I can access everything including Public, Private , and Protected Members.

correct

True



False

If you want to have a member that is private to outside class and public to classes inherits the current class, which access specifier/modifier you use?

Public

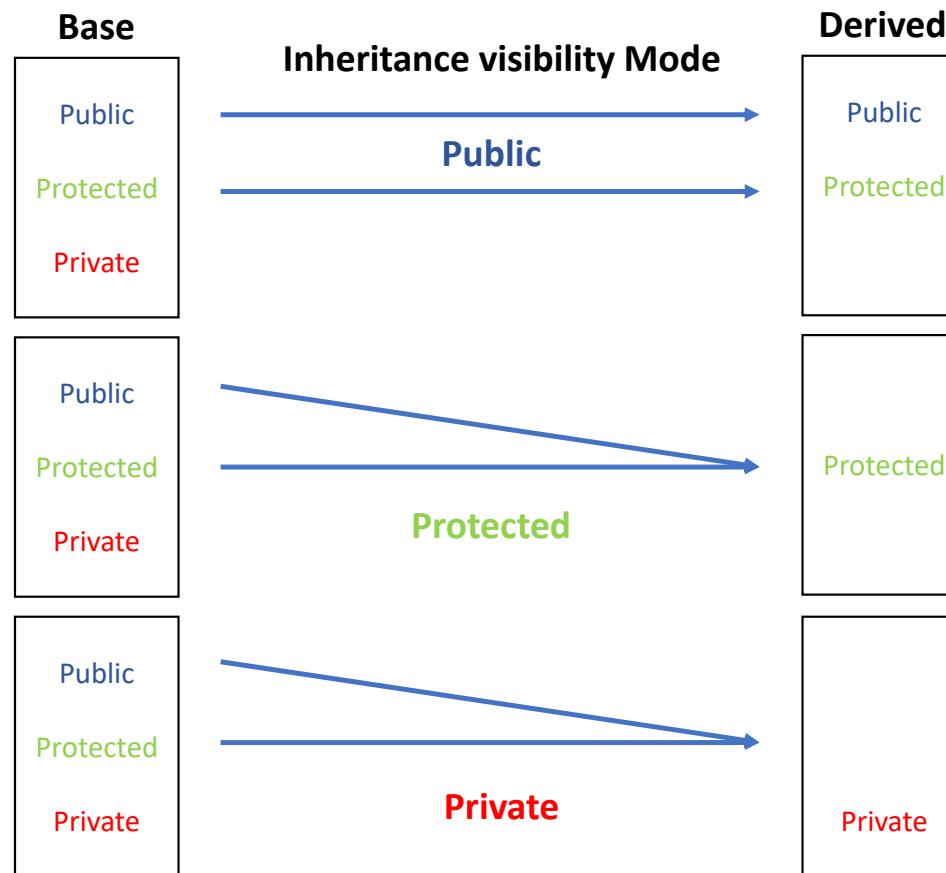
Private

correct

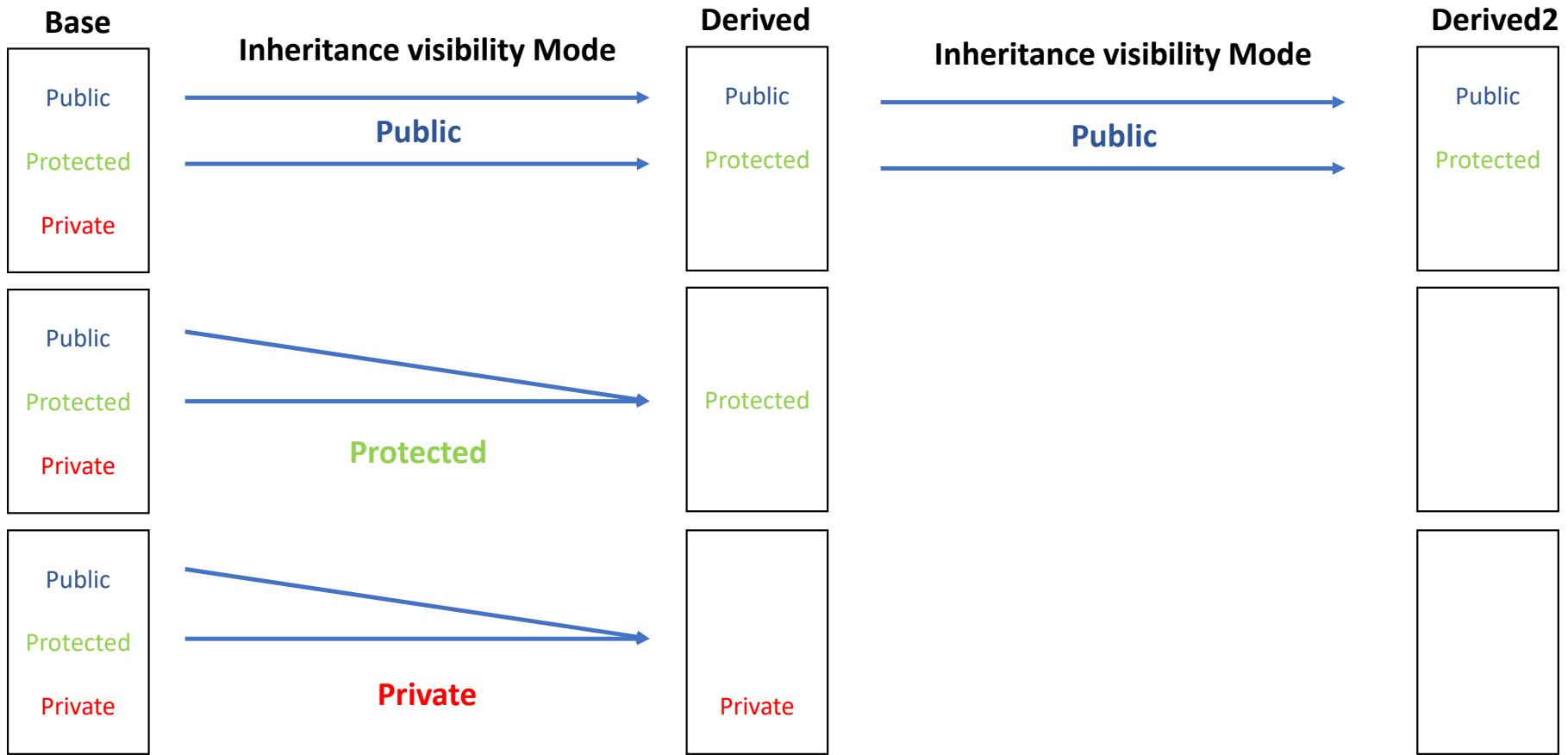
Protected



# Visibility Modes



# Visibility Modes





# Syntax:

```
class DerivedClassName : <Visibility Mode> BaseClassName  
{  
};
```

	Private Members	Protected Members	Public Members
Public Inheritance	Inaccessible	Protected	Public
Private Inheritance	Inaccessible	Private	Private
Protected Inheritance	Inaccessible	Protected	Protected



```
1 // inaccessible الوصول إليها ->
2 // accessible يمكن الوصول ->
3
4 #include <iostream>
5 using namespace std;
6
7 class clsA
8 {
9
10 private:
11     int v1;
12     int Fun1()
13     {
14         return 1;
15     }
16
17 protected:
18     int v2;
19     int Fun2()
20     {
21         return 2;
22     }
23
24 public:
25     int v3;
26     int Fun3()
27     {
28         return 3;
29     }
30 };
31
32 // try to change visibility mode public/private/protected
33 // and see in the main what will happen inside objects.
34 class clsB : private clsA
35 {
36
37 public:
38     int Fun4()
39     {
40
41         return 4;
42     }
43 };
44
45 class clsC : public clsB
46 {
47
48 public:
49     int Fun5()
50     {
51
52         return 5;
53     }
54 };
55
56 int main()
57 {
58
59     clsB B1;
60     // Try B1 after you change visibility mode in clsB. and see what you can see.
61
62     clsC C1;
63     // Try C1 after you change visibility mode in clsB. and see what you can see.
64 }
65
```

Visibility Mode "Private" will make everything private in the derived class, so it can make use of it and no one will make use of it neither derived classes nor objects.

correct

True



False

Visibility Mode "Private" will make everything private in the derived class, so it can make use of it and no one will make use of it neither derived classes nor objects.

correct

True



False

Visibility Mode "Protected" will make everything protected in the derived class, so it can make use of it and all other derived classes will make use of it, but no objects will make use of it.

correct

True



False

Visibility Mode "Public" will inherit the class publicly so every public members and protected members are useful for others.

correct

True



False

Private Members in the Base Class are not accessible from outside the class nor the derived classes.

correct

True



False

## 2- لما نورث كلاس ب Mode لـ Private

أ- الكلاس الموروثة من المستوى الأول :

- من البرا لا تستطيع الوصول لأي Member يخص الكلاس الأم .
- من الداخل تستطيع الوصول لل Members الخاصة بالكلاس الأم الموجودة في ال public و ال protected .

ب - الكلاسات الموروثة من المستوى الثاني أو أكثر :

- سواء من البرا أو من الداخل لا يستطيعون الوصول لأي Members يخص الكلاس الأم .

### 3- لما نورث كلاس ب ل Protected Mode :

أ- الكلاس الموروثة من المستوى الأول :

- من البرا لا تستطيع الوصول لأي members يخص الكلاس الأم .
- من الداخل تستطيع استخدام ال Member الخاصة بالكلاس الأم الموجة في ال public و ال protected .

ب - الكلاس الموروثة من المستوى الثاني أو أكثر :

- لها نفس صلاحيات الوصول الخاصة بالكلاس الموروثة من المستوى الأول.
- ( أي أن ال protected تسمح بالإستخدام الداخلي فقط بغض النظر عن ال Level )



Multi-Level Inheritance is a concept in object-oriented programming where a class inherits from another class, which in turn inherits from another class, forming a chain of inheritance relationships. In other words, a class serves as the base class for another class, and that derived class, in turn, serves as the base class for another class, and so on.



Here's an example to illustrate Multi-Level Inheritance:

```
cpp Copy code  
  
// Base class  
class Animal {  
public:  
    void eat() {  
        cout << "Animal is eating." << endl;  
    }  
};  
  
// Derived class 1 (inherits from Animal)  
class Mammal : public Animal {  
public:  
    void run() {  
        cout << "Mammal is running." << endl;  
    }  
};  
  
// Derived class 2 (inherits from Mammal)  
class Dog : public Mammal {  
public:  
    void bark() {  
        cout << "Dog is barking." << endl;  
    }  
};
```

Regenerate response

In this example, we have a base class `Animal`, which has a method `eat`. Then, we have a derived class `Mammal`, which inherits from `Animal` and adds a method `run`. Finally, we have another derived class `Dog`, which inherits from `Mammal` and adds a method `bark`.

With multi-level inheritance, the `Dog` class indirectly inherits the properties and methods of both `Mammal` and `Animal`. It forms a chain of inheritance relationships: `Dog` is a `Mammal`, and `Mammal` is an `Animal`.

Using multi-level inheritance, derived classes can build upon the functionality of their parent classes, creating a hierarchical structure that represents different levels of abstraction and specialization. However, it's essential to use inheritance judiciously and avoid deep inheritance hierarchies to maintain code clarity and avoid potential issues like the diamond problem.



Inheritance visibility modes, also known as access specifiers, are used to control the visibility of base class members in the derived class. In C++, there are three inheritance visibility modes:

#### 1. Public Inheritance:

In public inheritance, the public members of the base class remain public in the derived class, the protected members remain protected, and the private members are not accessible directly. The derived class can access the public and protected members of the base class.

Example of public inheritance:

cpp

Copy code

```
class Shape {
public:
    void display() {
        cout << "Shape" << endl;
    }
};

class Circle : public Shape {
public:
    void show() {
        display(); // Accessing the public member of the base class
        cout << "Circle" << endl;
    }
};
```

## 1. Protected Inheritance:

In protected inheritance, both the public and protected members of the base class become protected in the derived class, and the private members are not accessible directly. The derived class can access the protected members of the base class.

Example of protected inheritance:

cpp

Copy code

```
class Person {
protected:
    void greet() {
        cout << "Hello" << endl;
    }
};

class Employee : protected Person {
public:
    void welcome() {
        greet(); // Accessing the protected member of the base class
        cout << "Welcome" << endl;
    }
};
```

## 1. Private Inheritance:

In private inheritance, both the public and protected members of the base class become private in the derived class, and the private members are not accessible directly. The derived class can access the base class members only through its member functions.

Example of private inheritance:

```
cpp Copy code

class Animal {
public:
    void makeSound() {
        cout << "Animal sound" << endl;
    }
};

class Dog : private Animal {
public:
    void bark() {
        makeSound(); // Accessing the private member of the base class through
                     // a member function of the derived class
        cout << "Woof" << endl;
    }
};
```

Note: The choice of inheritance visibility mode depends on the specific requirements and design of your classes. It's essential to consider encapsulation and maintainability when choosing the appropriate visibility mode. Public inheritance is the most common and widely used mode in object-oriented programming.

- جملة الـ inheritance visibility modes  
تتعلق بـ **نوع دوئية الوراثة**

## Syntax:

CLASS Declared Class Name. <visibility mode> Base Class Name

Section 201(19) (b) (i) (A) (ii)  
Title II Body of the Class

visibility mode	private members	protected members	public members
public	inaccessible	protected	public
private	inaccessible	private	private
protected	inaccessible	protected	protected

- دلوقتي إنت عارفين إيه مطلوب كلasse صيغة عن date member اللي فما الفاريل معنها members methods اللي فما الفانكشن اللي  
الmembers دلوقتي جوة الملاس مبيكتوش موجود في الشاعرة بيكونو تحت مثلك (سمها  
Protected و Private و Public) «محددات الوصول» اللي فما Access Specifiers اسمها  
الـ private اللي لهم بيكونو من وتنفس جوة الملاس بس  
الـ protected اللي لهم من جمة الملاس ودى كلاس بيورده الملاس ده يقدر توصل  
الـ public اللي لهم بيكونو متاح من جمة الملاس ومناخ للكلasse اللي بيورس  
ومتاحة في ذي صفة في البرامج لما تدخل أو بيكدا من الملاس ده

SUBJECT:

DATE:

• مثب دلوقت اتنا الما يخلي Class يورث من Class  
و يستخدمت ال Public خلينها visibility mode

EXAMPLE:

Class Employee ::> Public Person

inaccessible لما فيه Person الى جوة الكلاس private members  
من ذي حبة برة ال Person كلاس.  
إذا accessible Person فيكونوا protected members  
جوا الكلاس Employee بس مش accessible لازى ذوبجكت يتعمل منه فيها  
accessible aجوة الكلاس وبيس. زما ال public members  
يتعمل منه Employee.

9.0.179

9.0.179

9.0.179

9.0.179

• مثب لما اخلي Class يورث من

private خلينها visibility mode

EXAMPLE:

Class Employee ::> private Person

inaccessible لما فيه Person الى جوة الكلاس private members  
من ذي حبة برة ال Person كلاس.  
إذا accessible Person فيكونوا protected members  
جوا الكلاس Employee بس ذي حبة برة ال Person كلاس حيث ورث عن الكلاس  
من ذي حبة برة ال Person كلاس حيث ورث عن الكلاس  
و كذلك ال public members

جوا الكلاس Employee ويزمتوه ذي كلاس حيث ورث عن Employee

ويعرف بورث عن عارف

SUBJECT: \_\_\_\_\_

DATE: \_\_\_\_\_

**protected** (أعلى **Visibility mode**) (ذمة لا يستخدمها).

Example:

Class Employee : protected Person

إذاً `Person` لا يمكن الوصول إليه من خارج المثلثة `Person`، فهو **غير متاح** (Inaccessible).

کما اے Public ممبرس و مکسیبل جوا الکاس  
و مکسیبل لئی کلاس یورٹہ یس مہیں ذی اوبیکٹ یقزرو ملائم

### • in summary:

: visto.

- هو يستخدم في الميinars الـ Public و Protected mode visibility الميinars الـ Public و Protected

-لو يستخدمت الـ **protected** والـ **public** العيّرس الـ **protected** الى في الـ class **base** موجود الـ **protected** members الى في الـ class **sub** كـ **protected** members يعني ذي الـ **objekt** فيعمل من الـ **sub class** من غير دفع المعيّرس الموروثة ذي والوصول لـ **private** متاح من داخل الـ **klass** نفسه بـ **public** وـ **protected** ذي الـ **klass** الثاني فيورت من الـ **sub class** وهو عازم في المود ذي **protected** **public**

## Base Class - King

Can't Inherit

Can't Inherit

Private

Protected

Public

class A : Private King

Private

Protected

Public

class B : Protected King

Private

Protected

Public

class C : Public King

Private

Protected

Public

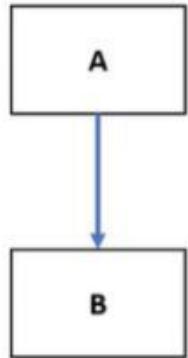


Base Class Visibility	Derived Class Visibility		
	Private	Protected	Public
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

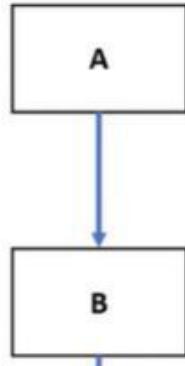


# Inheritance Types

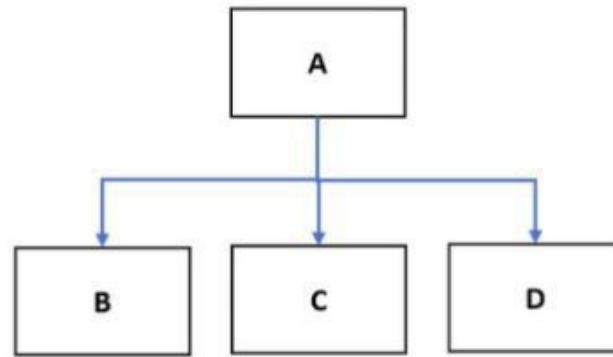
أنواع الوراثة



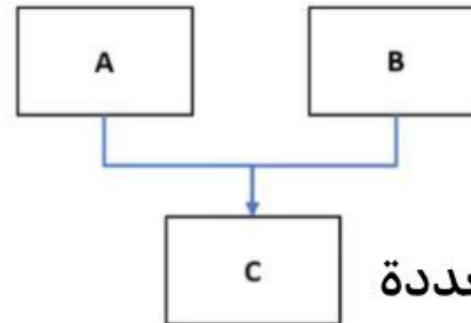
Single Inheritance  
وراثة فردية



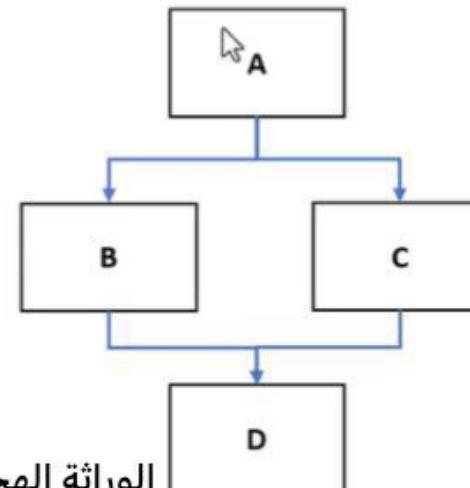
Multi Level Inheritance  
وراثة متعددة المستويات



Hierachal Inheritance  
وراثة هرمية



Multiple Inheritance



الوراثة الهجينية  
Hybrid Inheritance



In object-oriented programming, inheritance allows a class (called the derived class or subclass) to inherit properties and behaviors from another class (called the base class or superclass). There are different types of inheritance, each serving different purposes in code organization and reuse. The main types of inheritance are:

#### 1. Single Inheritance:

Single inheritance involves one class inheriting from a single base class. It is the simplest and most common form of inheritance.

Example of single inheritance:

```
cpp Copy code

class Shape {
public:
    void display() {
        cout << "Shape" << endl;
    }
};

class Circle : public Shape {
public:
    void show() {
        display(); // Accessing the base class member using single inheritance
        cout << "Circle" << endl;
    }
};
```

## 1. Multiple Inheritance:

Multiple inheritance involves one class inheriting from multiple base classes. The derived class acquires the properties and behaviors of all the base classes.

Example of multiple inheritance:

```
cpp Copy code

class A {
public:
    void methodA() {
        cout << "Method A" << endl;
    }
};

class B {
public:
    void methodB() {
        cout << "Method B" << endl;
    }
};

class C : public A, public B {
public:
    void methodC() {
        methodA(); // Accessing base class members from multiple inheritance
        methodB();
        cout << "Method C" << endl;
    }
};
```

## 1. Multilevel Inheritance:

Multilevel inheritance involves a chain of inheritance, where a derived class becomes the base class for another derived class.

Example of multilevel inheritance:

```
cpp Copy code

class Animal {
public:
    void makeSound() {
        cout << "Animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        makeSound(); // Accessing the base class member using multilevel inh
        cout << "Woof" << endl;
    }
};

class Labrador : public Dog {
public:
    void fetch() {
        cout << "Labrador fetches the ball" << endl;
    }
};
```

## 1. Hierarchical Inheritance:

Hierarchical inheritance involves one base class being inherited by multiple derived classes. It allows multiple classes to share common functionality from a single base class.

Example of hierarchical inheritance:

```
cpp Copy code

class Shape {
public:
    void display() {
        cout << "Shape" << endl;
    }
};

class Circle : public Shape {
public:
    void showCircle() {
        display(); // Accessing the base class member in hierarchical inherit
        cout << "Circle" << endl;
    }
};

class Square : public Shape {
public:
    void showSquare() {
        display(); // Accessing the base class member in hierarchical inherit
        cout << "Square" << endl;
    }
};
```

Each type of inheritance offers a different way to organize and structure classes in an object-oriented program, and the choice depends on the specific requirements and design considerations of the application.

# What are types of Inheritance?

Single inheritance

Multi-level inheritance

Multiple inheritance

Hybrid inheritance

Hierarchical inheritance

correct

All of the above



Multiple inheritance are not supported by modern languages such as  
JAVA and C#

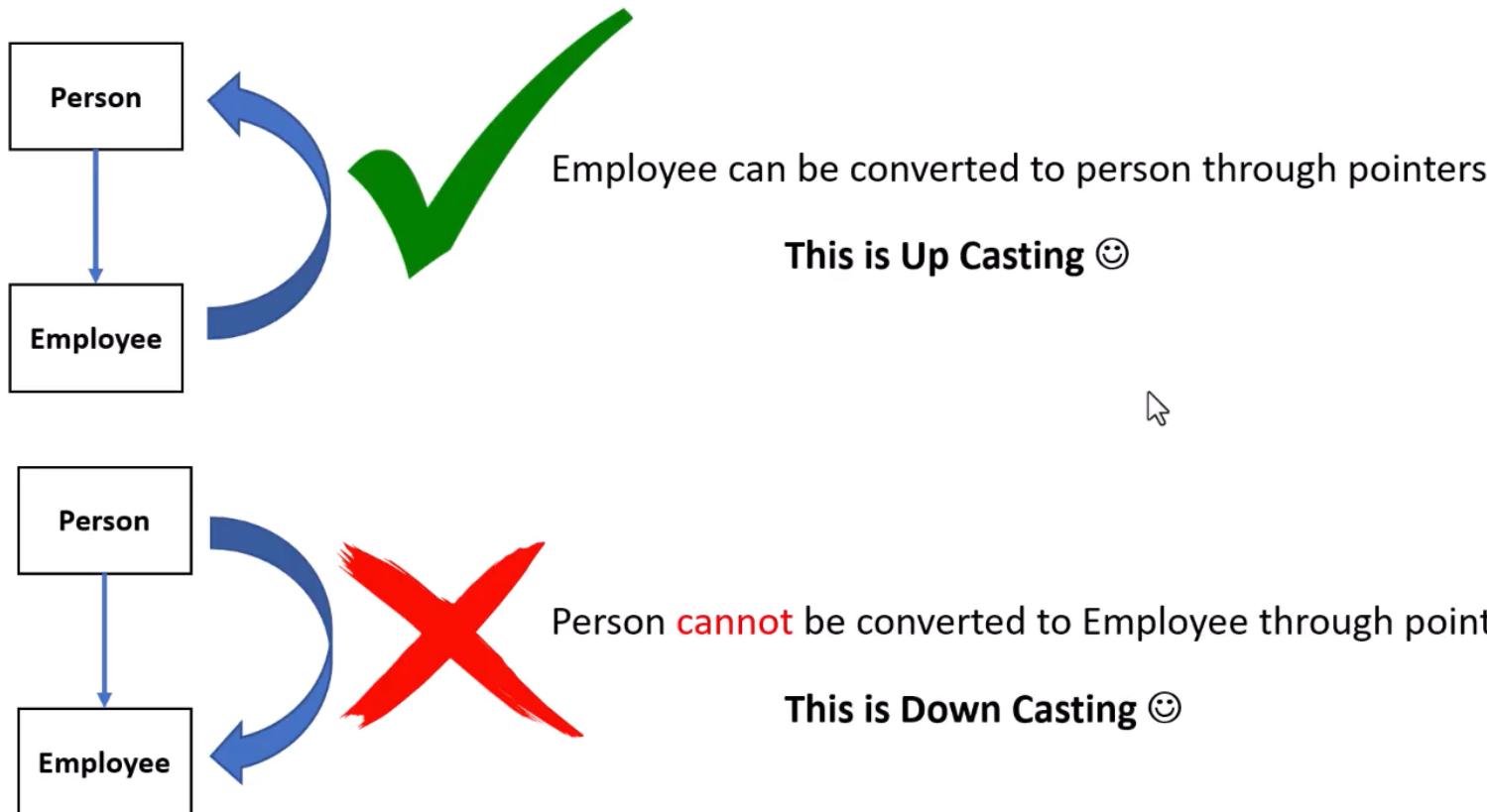
correct

True



False

# Up Casting vs Down Casting





```
1 #include <iostream>
2
3 using namespace std;
4
5 class clsPerson
6 {
7
8 public:
9     string FullName = "Mahmoud Mattar";
10 };
11
12 class clsEmployee : public clsPerson
13 {
14
15 public:
16     string Title = "CEO";
17 };
18
19 int main()
20 {
21
22     clsEmployee Employee1;
23
24     cout << Employee1.FullName << endl;
25
26     // upCasting
27     // this will convert employee to person.
28     clsPerson *Person1 = &Employee1;
29     cout << Person1->FullName << endl;
30
31     clsPerson Person2;
32     cout << Person2.FullName << endl;
33
34     // downcasting : you cannot convert person to employee
35     clsEmployee* Employee2 = &Person2;
36
37     // error (a value of type "clsPerson *" cannot be used to initialize an entity of type "clsEmployee *")
38     system("pause > nul");
39     return 0;
40 }
```

Up Casting is converting derived object to it's base object

correct

True

False

Down Casting is Converting Base object to Derived object

correct

True



False

A pointer of type parent can point to an object of child class.

correct

True, because all the members in which the pointer can access are exist in memory when the object of child class.



False

A pointer of child class cannot point to an object of parent class.

correct

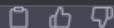
True, because the child class members the pointer can access do not exist in memory when  
the object is of parent class.



False



Upcasting and downcasting are concepts related to inheritance and object-oriented programming. They are used when dealing with classes and their inheritance hierarchies.



## 1. Upcasting:

Upcasting refers to the process of converting a derived class object to a pointer or reference of its base class. In other words, you are "casting up" the inheritance hierarchy. Upcasting is safe and implicit because a derived class is always a base class as well, so it can be treated as its base class without any issues.

Example:

```
cpp Copy code

class Animal {
public:
    void makeSound() {
        cout << "Animal makes a sound." << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() {
        cout << "Dog barks." << endl;
    }
};

int main() {
    Dog dog;
    dog.makeSound(); // Output: Dog barks.

    // Upcasting: Converting Dog to Animal (safe and implicit)
    Animal* animalPtr = &dog;
    animalPtr->makeSound(); // Output: Dog barks. (Polymorphism: dynamic binding)
}
```

Regenerate response

## 1. Downcasting:

Downcasting refers to the process of converting a base class pointer or reference back to a derived class pointer or reference. In other words, you are "casting down" the inheritance hierarchy. Downcasting is potentially unsafe and requires explicit type casting because a base class pointer may not point to an object of the derived class. It should only be done when you are sure that the object being pointed to is of the derived class.

Example:

```
cpp Copy code

class Animal {
public:
    void makeSound() {
        cout << "Animal makes a sound." << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() {
        cout << "Dog barks." << endl;
    }

    void fetch() {
        cout << "Dog fetches the ball." << endl;
    }
};

int main() {
    Animal* animalPtr = new Dog();
    animalPtr->makeSound(); // Output: Dog barks. (Polymorphism: dynamic bi

    // Downcasting: Converting Animal to Dog (explicit and potentially unsaf
    Dog* dogPtr = dynamic_cast<Dog*>(animalPtr);
    if (dogPtr != nullptr) {
        dogPtr->fetch(); // Output: Dog fetches the ball.
    } else {
        cout << "Cannot downcast to Dog." << endl;
    }

    // Remember to delete the allocated memory (not shown in this example)
    delete animalPtr;
}
```

```
class Animal {
public:
    void makeSound() {
        cout << "Animal makes a sound." << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() {
        cout << "Dog barks." << endl;
    }

    void fetch() {
        cout << "Dog fetches the ball." << endl;
    }
};

int main() {
    Animal* animalPtr = new Dog();
    animalPtr->makeSound(); // Output: Dog barks. (Polymorphism: dynamic bi

    // Downcasting: Converting Animal to Dog (explicit and potentially unsafe)
    Dog* dogPtr = dynamic_cast<Dog*>(animalPtr);
    if (dogPtr != nullptr) {
        dogPtr->fetch(); // Output: Dog fetches the ball.
    } else {
        cout << "Cannot downcast to Dog." << endl;
    }

    // Remember to delete the allocated memory (not shown in this example)
    delete animalPtr;
}
```

In the example above, `dynamic\_cast` is used for downcasting, and it returns a null pointer if the downcast is not possible. It is a safe way to perform downcasting because it checks the actual type of the object at runtime. However, downcasting should be used with caution, and you should always make sure that the object being downcasted is of the expected derived class type to avoid potential runtime errors.

## Up Casting & Down Casting

باختصار انت ديمماً تقدر تحمل up Casting ومبقرش تحمل down Casting

بالمحض المفهود ايماً ال base Class يقدر تعرف منها ابوينت بيوشر على ذي لاته يعني من حالة المفهود بتاعة مفهودة فنان لكن ال derived Class تحمل منها ابوينت يوش على Base Class مبقرش

Example:

Class CLSPerson {

public:  
string Name = "mahmoud";

};

Class CLSEmployee : Public CLSPerson {

public:

string title = "CEO";

};

int main() {

CLSEmployee EMP1;

//UpCasting

CLSPerson \*PER1 = &EMP1;

① cout << PER1->Name; //Mahmoud

② cout << PER1->title; //Error (CLSPerson) has No member type  
call 'title'