OOP

Programming Paradigm

- A programming paradigm is a fundamental style or approach to programming that provides a way of structuring and organizing code.
- It defines the principles and patterns that shape how programs are designed, developed, and executed.
- Different programming paradigms offer different ways of thinking about and solving problems, influencing how developers write code and how programs are structured.

Procedural Programming:

- **Description:**
    - Involves writing procedures or functions that perform operations on data.
    - It follows a linear, step-by-step approach where the program executes a sequence of instructions.
- **Examples:** C, Pascal, Fortran.

Object-Oriented Programming (OOP):

- **Description:**
    - Organizes code into objects that encapsulate data and the methods that operate on that data.
    - It emphasizes concepts like inheritance, polymorphism, and encapsulation.
- **Examples:** Java, C++, Python.

Functional Programming:

- **Description:**
    - Focuses on writing pure functions that avoid mutable state and side effects.
    - It treats computation as the evaluation of mathematical functions.
- **Examples:** Haskell, Scala, Erlang.

Declarative Programming:

- **Description:**
    - Expresses the logic of computation without describing its control flow.
    - Instead of specifying how to perform tasks, you specify what you want to be done.
- **Examples:** SQL, HTML, CSS.

Event-Driven Programming:

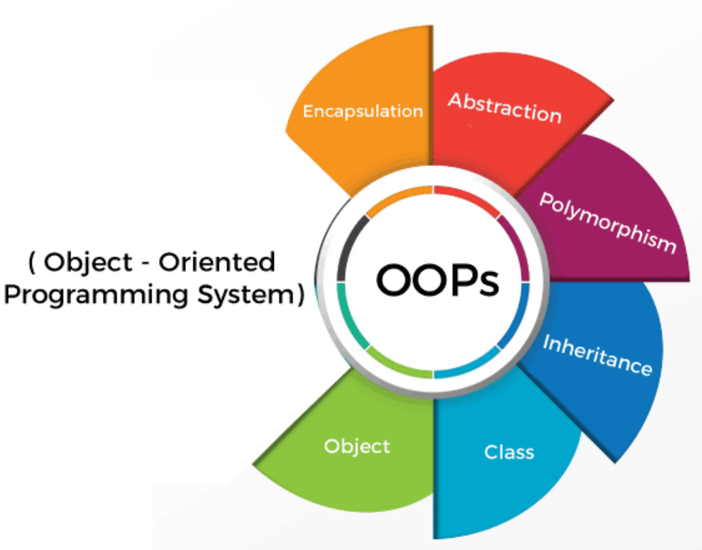
- **Description:**
    - The flow of the program is determined by events such as user actions, sensor outputs, or message passing.
    - This paradigm is commonly used in graphical user interfaces (GUIs) and real-time systems.
- **Examples:** JavaScript, Visual Basic, C# (when using events).

Concurrent Programming:

- **Description:**
    - Focuses on the execution of multiple computations or processes simultaneously, often to improve performance or responsiveness.
- **Examples:** Go, Erlang, Java (with multithreading).

Object-Oriented Programming

- Object-Oriented Programming (OOP) in C++ is a programming paradigm that uses objects and classes to organize and structure code.
- OOP focuses on representing real-world entities using **objects**, which are instances of classes, to design software.
- OOP helps to keep the C++ code **DRY** "Don't Repeat Yourself", which is a principle for reducing the repetition of code. You should extract out the codes that are common for the application and place them at a single place and reuse them instead of repeating it.
- **Key Concepts of OOP in C++:**
    - Classes and Objects
    - Encapsulation
    - Inheritance
    - Polymorphism
    - Abstraction



Important Features of OOP

1. Bottom-up Approach in Program Design

- **Explanation:**
    - In OOP, software development often starts with creating small, reusable components (**objects**) that are combined to form more complex systems.
    - This is in contrast to the top-down approach used in procedural programming, where the system is designed as a whole and then broken down into smaller parts.
- **Benefit:**
    - This approach allows for more flexibility and easier debugging since each component is self-contained and can be tested individually.

2. Programs Organized Around Objects, Grouped in Classes

- **Explanation:**
    - In OOP, the primary building blocks are objects, which are instances of classes.
    - Classes define the structure and behavior of objects.
    - Programs are designed by creating classes that represent real-world entities and their interactions.
- **Benefit:**
    - This organization around objects mirrors real-world entities, making the design more intuitive and modular.
    - It also promotes code reusability since classes can be reused across different programs.

3. Focus on Data with Methods to Operate Upon Object's Data

- **Explanation:**
    - OOP emphasizes the encapsulation of data (attributes) and the methods (functions) that operate on that data within objects.
    - This ensures that an object's data is accessed and modified only through its methods, which can enforce rules and constraints.
- **Benefit:**
    - Encapsulation protects the integrity of the data and reduces the likelihood of unintended side effects from code changes.
    - It also makes the code more modular and easier to maintain.

4. Interaction Between Objects Through Functions

- **Explanation:**
    - In OOP, objects interact with each other by calling methods (functions) on other objects.
    - This interaction mimics how real-world entities interact, with each object performing actions and requesting services from other objects.
- **Benefit:**
    - This method of interaction supports the creation of complex systems where objects can collaborate to perform tasks, leading to more organized and maintainable code.

5. Reusability of Design Through Creation of New Classes by Adding Features to Existing Classes

- **Explanation:**
    - OOP supports the concept of inheritance, where new classes can be created by extending existing classes.
    - This allows developers to reuse and enhance existing code without modifying it, promoting the reusability of design and code.
- **Benefit:**
    - Inheritance enables the creation of hierarchies and more complex relationships between objects, making it easier to build and manage large software systems.
    - It also reduces code duplication and enhances maintainability.

Why is C++ a partial OOP?

- The C++ programming language is categorized as a "partial" object-oriented programming language despite the fact that it supports OOP concepts.
- C++ is often referred to as a "partial" or "hybrid" object-oriented programming (OOP) language.

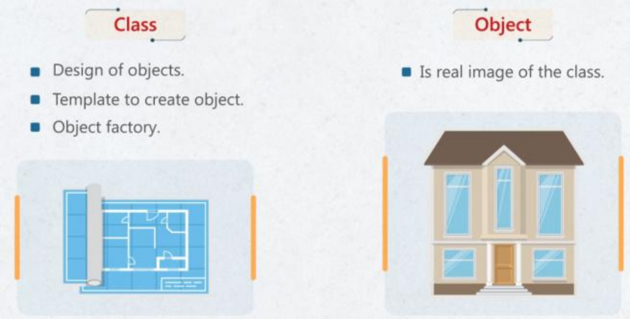
1) main() function

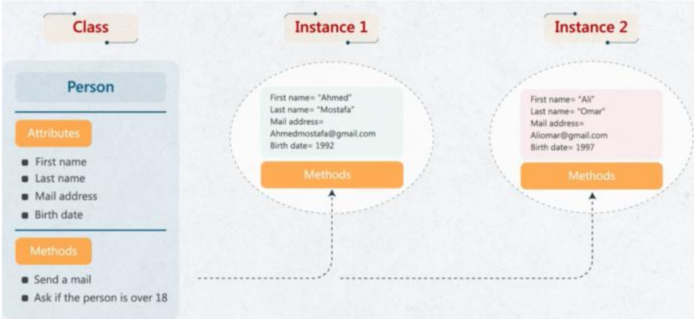
- In C++, the `main()` function is the entry point of the program and must be defined outside of any class.
- This means that a C++ program can exist without any classes or objects and still function, as the program's execution starts with the `main()` function, not an object or method within a class.
- This flexibility allows for procedural programming within C++, meaning you can write and execute code without using OOP principles.
- This is a key reason why C++ is not considered a "pure" OOP language, as pure OOP languages like Java require all code to be written within classes.

2) Global Variables

- C++ allows the use of global variables, which are variables defined outside of any class or function and can be accessed from any part of the program.
- This breaks the principle of **encapsulation**, which is a core concept of OOP that dictates that data (attributes) should be hidden within classes and only accessible through controlled interfaces (methods).

Classes and Objects





Class

- A class is a <u>user-defined data type</u> that we can use in our program, and it works as an object constructor, or a <u>"**blueprint**"</u> for creating objects.
- It defines <u>properties (attributes)</u> and <u>behaviors (methods)</u> that the objects of the class will have.
- When you define a class in C++, the class itself <u>does not consume memory</u> like an object does.
- The memory is <u>not allocated for the class itself</u>. However, when you create an instance (object) of that class, memory is allocated to store the object's data members (attributes).
- اول فايدة هتطلع من فكرة الClass هو انى عن طريقه هوحد كل الattributes و الmethods اللى كل الobjects بتوعه هيكونوا محتاجينها ، يعنى هبقى ضامن ان كله عنده نفس الحاجة مش واحد يكون عنده حاجة مختلفة عن التانى.
- تانى حاجة برضو لو بعد ما عملت الObjects مثلا وجيت لاقيت انى عاوز اعدل حاجة في الattributes او الmethods هبقى مضطر ساعتها انى امشى على كل واحد اعمل فيه التعديل دة ، لكن بوجود الClass كدة انا هعمل التعديل دة في الClass بس.
- **<u>Syntax:</u>**

```
class ClassName {
private:
    // Private attributes (member variables)
    Type1 attribute1;
    Type2 attribute2;

    // Private methods (member functions)
    ReturnType method1(Parameters) {  // Method implementation   }

protected:
    // Protected attributes and methods
    Type3 attribute3;
    ReturnType method2(Parameters) {   // Method implementation  }

public:
    // Public attributes (member variables)
    Type4 attribute4;

    // Constructor(s)
    ClassName(Parameters) {
        // Constructor implementation
    }

    // Public methods (member functions)
    ReturnType method3(Parameters) {   // Method implementation  }

    // Destructor
    ~ClassName() {
        // Destructor implementation
    }
};
```

Object

- An instance of a class. It represents a specific entity with defined attributes and behaviors.
- When you create an object, that object is allocated a block of memory to hold its data members (attributes).
- The object name itself is not a pointer, but it can be thought of as a reference to the memory location where the object's data members are stored.
- Methods are not stored within each object. Instead, they are typically stored in a single location in memory (part of the program’s code segment) and shared among all instances of the class.

Attributes

- Attributes, also known as **fields** or **properties**, are variables that belong to a class.
- They represent the state or characteristics of an object.
- Each object of a class has its own set of attribute values.
- **Example:** In a `Car` class, attributes might include `brand`, `model`, and `year`. These attributes hold specific values for each car object created from the class.

```
class Car {
public:
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
};

int main() {
    Car myCar;
    myCar.brand = "Toyota";
    myCar.model = "Corolla";
    myCar.year = 2020;
}
```

Methods

- Methods, also known as **functions** or **procedures** in other programming paradigms, are functions that belong to a class.
- They define the behavior of an object or the actions that an object can perform.
- **Example:** In the `Car` class, a method might be `start()`, which defines the action of starting the car.

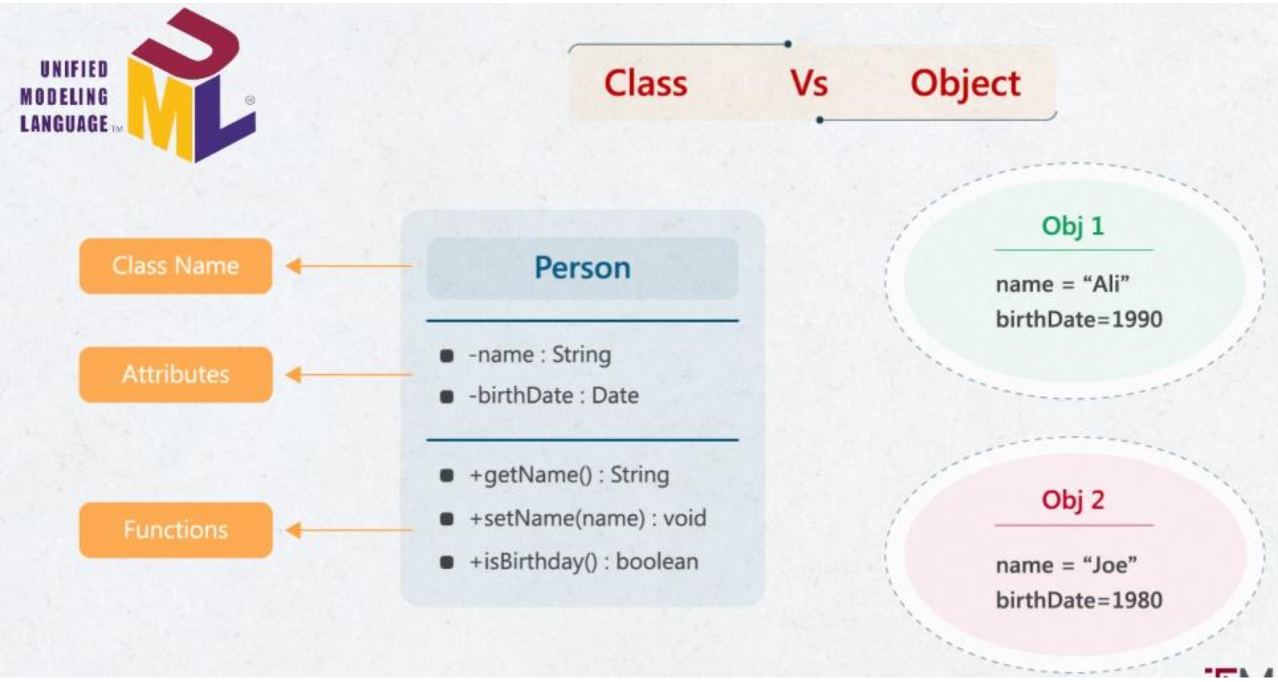
```
class Car {
public:
    string brand;
    string model;
    int year;

    void start() {         // Method
        cout << "Car started!" << endl;
    }
};

int main() {
    Car myCar;
    myCar.brand = "Toyota";
    myCar.model = "Corolla";
    myCar.year = 2020;

    myCar.start();  // Calling the method, Output: Car started!
}
```

- <u>*There are two ways to define functions that belongs to a class:*</u>
  - Inside class definition
  - Outside class definition
    - To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class.
    - This is done by specifying the name of the class, followed the scope resolution `::` operator, followed by the name of the function:

```
class MyClass {         // The class
  public:               // Access specifier
    void myMethod();    // Method/function declaration
};

// Method/function definition outside the class
void MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

UML

- UML, or **Unified Modeling Language**, is a standardized visual language used to model and design software systems.
- It provides a set of <u>diagrams</u> and <u>symbols</u> to represent the <u>structure</u>, <u>behavior</u>, and <u>interactions</u> within a system.
- UML is widely used in software engineering to document and communicate the architecture, design, and functionality of a system, helping developers, architects, and stakeholders to understand and collaborate on the system's development.



Constructor

- A constructor is a <u>special method</u> that is **automatically** called when an object of a class is <u>created</u>.
- It is usually used to **initialize** the attributes (member variables) of the object.
- The constructor has the <u>same name</u> as the class and <u>does not have a return type</u> (not even `void`).
- Constructors can be <u>overloaded</u>, meaning you can have <u>multiple constructors</u> with <u>different parameter lists</u>.
- You <u>**cannot**</u> call a constructor directly like you would call a regular function.
- If no constructor is explicitly defined, <u>C++ provides a default constructor</u> (a constructor with no parameters) with an <u>empty body</u>.

- طيب لو انا عملت ع الأقل constructor واحد ساعتها الـ++C مش هتعمل default constructor وهتعتمد بس على الـconstructors اللى انا عاملها ، بالتالى لو انا مكنتش عامل default constructor مش هقدر اعمل اى object بطريقة انى انادى على default constructor اللى هي دى `ClassName Obj();`.

- نقطة مهمة جدا اخلى بالى منها هي ان الـconstructor مش هو اللى بيعمل creation للـobject ، لكن في نفس الوقت مفيش object creation بيحصل غير اما يحصل بعده علطول constructor calling حتى لو انا مش عامل ولا constructor لأن ساعتها الـ++C بتعمل هي default constructor ، والدليل على كدة انى جوة اى constructor بعمل access للـattributes بتاعة الـclass عشان اخزن فيها values ودة معناه ان الـattributes دة أصلا خلاص اتحجز ليها مكان في الميمورى بالتالى الـobject كان معموله creation خلاص.

- If a constructor is defined as a <u>private member</u>, then <u>no object</u> can be created by it.

<u>Single Constructor:</u>

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object of MyClass (this will call the constructor)
  return 0;
}
```

<u>Constructor Overloading:</u>

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

class Car {         // The class
public:             // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute

    // Constructor declarations
    Car(string x, string y, int z);          // Constructor with all attributes
    Car(string x, string y);                 // Constructor with brand and model only
    Car(string x);                           // Constructor with brand only
    Car();                                   // Default constructor
};

// Constructor definitions outside the class
Car::Car(string x, string y, int z) {
    brand = x;
    model = y;
    year = z;
}
```

```cpp
Car::Car(string x, string y) {
    brand = x;
    model = y;
    year = 0; // Default value for year
}

Car::Car(string x) {
    brand = x;
    model = "Unknown"; // Default value for model
    year = 0;          // Default value for year
}

Car::Car() {
    brand = "Unknown"; // Default value for brand
    model = "Unknown"; // Default value for model
    year = 0;          // Default value for year
}

int main() {
    // Create Car objects and call the different constructors
    Car carObj1("BMW", "X5", 1999);        // Calls constructor with 3 parameters
    Car carObj2("Ford", "Mustang");        // Calls constructor with 2 parameters
    Car carObj3("Toyota");                 // Calls constructor with 1 parameter
    Car carObj4;                           // Calls the default constructor

    // Print values
    cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
    cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
    cout << carObj3.brand << " " << carObj3.model << " " << carObj3.year << "\n";
    cout << carObj4.brand << " " << carObj4.model << " " << carObj4.year << "\n";

    return 0;
}
```

Initialization List

- In C++, an initialization list is a feature used in constructors to initialize member variables of a class before the constructor's body executes.
- This is particularly useful (and sometimes necessary) for initializing const members, reference members, and members of classes that do not have a default constructor.
- An initialization list is generally faster and more efficient than initializing member variables inside the constructor body.
    - When you use an initialization list, the member variables are initialized directly with the given values. This means the values are set in memory as the object is created.
    - If you initialize variables inside the constructor body, the variables are first default-initialized (if they have a default constructor) and then reassigned to the new values you provide.
- Think of it like ordering a custom-built car:
    - *Initialization List:* You tell the factory exactly how you want your car built, and they assemble it exactly to your specifications from the start.
    - *Constructor Body:* The factory first builds a basic model (default initialization) and then modifies it to fit your specifications (assignment), which takes more time and resources.
- **Syntax:**

```cpp
ClassName(Type1 arg1, Type2 arg2) : member1(arg1), member2(arg2) {
    // Constructor body
}
```

Cases for initialization list:

1. `const` **Members:** If your class has const members, they must be initialized when the object is created, and this can only be done through an initialization list.

```
class Example {
private:
    const int value;
public:
    Example(int v) : value(v) {}  // Must use an initialization list
};
```

2. **Reference Members:** Since references must be initialized to refer to something at the point of their creation, they must be initialized in the initialization list.

```
class Example {
private:
    int& ref;
public:
    Example(int& r) : ref(r) {}  // Must use an initialization list
};
```

3. **Base Class Initialization:** If your class is derived from another class (i.e., it has a base class), the base class constructor must be called in the initialization list of the derived class constructor.

```
class Base {
public:
    Base(int i) {}
};

class Derived : public Base {
public:
    Derived(int i) : Base(i) {}  // Must initialize Base with the initialization list
};
```

4. **Members of Classes Without Default Constructors:** If a member variable is an object of a class that does not have a default constructor, it must be initialized through the initialization list.

```
class NoDefaultConstructor {
public:
    NoDefaultConstructor(int) {}
};

class Example {
private:
    NoDefaultConstructor member;
public:
    Example(int v) : member(v) {}  // Must use an initialization list
};
```

Copy Constructor

- A copy constructor is a special type of constructor used to create a new object as a copy of an existing object.
- It is called when:
    - An object is initialized from another object of the same class.
    - An object is passed by value to a function.
    - An object is returned by value from a function.
- **General Syntax:**

```
ClassName(const ClassName &obj);
```

- It takes a single parameter, which is a **reference** to an object of the same class.
- The parameter is usually passed as a **constant reference** (`const`) to prevent modification of the original object.

- **You should define a custom copy constructor if:**
  - Your class has pointers or dynamically allocated memory.
  - You need a deep copy to ensure the new object has its own separate copy of the data, rather than sharing memory with the original object.

Default Copy Constructor

- A default copy constructor in C++ is a constructor that is automatically provided by the compiler if you don't explicitly define **copy constructor** for your class.
- The default copy constructor performs a **shallow copy (Bitwise)** of the object's data members.
- يبقى الـdefault copy constructor بيتعمل عن طريق الـcompiler لو انا معملتش copy constructor بنفسى زى ، لكن مشكلته بقى انه بيعمل shallow copy مش deep copy وبالتالي ممكن مشكلة تظهر لو عندى dynamic memory allocation

Example of Default Copy Constructor (Shallow Copy):

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

class MyClass {
public:
    int a;
    int* b;

    // Constructor
    MyClass(int x, int y) {
        a = x;
        b = new int(y); // Dynamically allocate memory for b
    }

    // Destructor
    ~MyClass() {
        delete b; // Clean up dynamically allocated memory
    }
};

int main() {
    MyClass obj1(10, 20);  // Original object
    MyClass obj2 = obj1;   // Default copy constructor is used here

    // Display values
    cout << "obj1.a: " << obj1.a << ", *obj1.b: " << *obj1.b << endl;
    cout << "obj2.a: " << obj2.a << ", *obj2.b: " << *obj2.b << endl;

    // Modify obj2's data
    *obj2.b = 30;

    // Check obj1's data after modification
    cout << "After modification:" << endl;
    cout << "obj1.a: " << obj1.a << ", *obj1.b: " << *obj1.b << endl;
    cout << "obj2.a: " << obj2.a << ", *obj2.b: " << *obj2.b << endl;

    return 0;
}
```

- When `MyClass obj2 = obj1;` is executed, the compiler-generated default copy constructor is called.
  - It copies `obj1.a` to `obj2.a`.
  - It copies the pointer `obj1.b` to `obj2.b`, so both `obj1.b` and `obj2.b` point to the same memory location.
- This can lead to problems:
  - **Unintentional Data Sharing:** Modifying `*obj2.b` affects `*obj1.b` because they point to the same data.
  - **Double Free Error:** When both `obj1` and `obj2` are destroyed, they will both try to free the same memory, leading to a runtime error.

Example of Custom Copy Constructor (Deep Copy):

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

class Deep {
public:
    int* data;

    // Constructor
    Deep(int value) {
        data = new int(value);
        cout << "Constructor called" << endl;    }

    // Custom Copy Constructor (Deep Copy)
    Deep(const Deep& obj) {
        data = new int(*obj.data); // Create a new copy of the data
        cout << "Copy Constructor called" << endl;
    }

    ~Deep() {
        delete data;
        cout << "Destructor called" << endl;
    }
};

int main() {
    Deep obj1(42);
    Deep obj2 = obj1; // Custom copy constructor is called

    cout << "obj1 data: " << *obj1.data << endl;
    cout << "obj2 data: " << *obj2.data << endl;

    return 0;
}
```

Key Points:

- **Shallow Copy:**
  - Performed by the default copy constructor.
  - Copies the values of member variables bit by bit.
  - Dangerous when the class involves dynamic memory allocation (e.g., pointers).
- **Deep Copy:**
  - Performed by a custom copy constructor.
  - Creates a new memory allocation for the copied object, ensuring independence between the original and copied objects.

- **Use Cases:**
  - If your class manages resources (e.g., memory, file handles), use a deep copy.
  - If the default behavior suffices (no dynamic memory), a shallow copy is fine.
- **Rule of Three**: If you define a custom:
  - Destructor
  - Copy Constructor
  - Copy Assignment Operator
- Ensure you define all three, as they often work together to manage the lifecycle of an object.

Destructor

- A destructor is a special method that is automatically called when an object goes out of scope or is explicitly deleted.
- It is used to perform cleanup tasks, such as releasing resources or memory that the object may have acquired during its lifetime.
- The destructor has the same name as the class, but with a tilde (~) prefix.
- It does not have a return type (not even `void`), and it does not take any parameters.
- The destructor is automatically invoked when an object is destroyed.
- There can be only **one** destructor in a class, and it cannot be overloaded.
- يبقى نقطة مهمة اوى في مفهوم الdestructor بشكل عام انه مش بيعمل delete للobject ، هو بس بيدينى إمكانية انى انفذ حاجة أخيرة قبل ما الobject يبقى out of scope او يتشال من الميمورى ، والدليل على كدة انى لو معملتش destructor الobject برضو هيتشال من الميمورى عادى.

```
class Car {
public:
    string brand;
    string model;
    int year;

    // Constructor
    Car(string b, string m, int y) {
        brand = b;
        model = m;
        year = y;
        cout << "Car created: " << brand << " " << model << " " << year << endl;
    }

    // Destructor
    ~Car() {
        cout << "Car destroyed: " << brand << " " << model << endl;
    }
};

int main() {
    {
        Car myCar("Toyota", "Corolla", 2020);
        // Output: Car created: Toyota Corolla 2020
    } // Destructor is called here when myCar goes out of scope
    // Output: Car destroyed: Toyota Corolla
}
```

Default Arguments

- In C++, default arguments allow you to assign a default value to a function parameter, making the parameter optional when the function is called.
- This feature is particularly useful in object-oriented programming (OOP) for constructors and other methods.
- Default arguments are specified in the function declaration:
    - If a caller does not provide a value for that argument, the default value is used.
    - If a value is provided, it overrides the default.
- **Order of Default Arguments:**
    - When multiple parameters have default values, the default arguments must be provided from right to left in the parameter list.
    - You **cannot skip** an argument on the left while providing a value for an argument on the right.
    - ***Some Wrong Cases:***
        - `int sum(int x = 0, y);`

مينفعش اعمل الحالة دى لأن طالما اديت default argument لـparameter يبقى لازم كل اللى على يمينه ياخدوا برضو default arguments ، لأن لو هنا مثلا بعت قيمة واحدة هو مش هيكون عارف يحطها في x ولا y.

- `int sum(int x , y = 0);`
  `int sum(int x);`

الحالة دى برضو مينفعش تكون عندى لأن كدة الـfunctions 2 عندهم نفس الـsignature ، لأن الـdefault arguments مش بيتحسبوا لما باجي اعد الـparameters بتوعى ، بالتالى لو جيت مثلا ناديت على sum واديتها مثلا 5 كدة الـcompiler مش هيعرف هو هينادى على انهى واحدة.

- **Example:**

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

class Rectangle {
private:
    int length;
    int width;

public:
    // Constructor with default arguments
    Rectangle(int l = 5, int w = 3) {
        length = l;
        width = w;
    }

    int area() {
        return length * width;
    }
};

int main() {
    Rectangle rect1;        // Uses default values: length = 5, width = 3
    Rectangle rect2(10);    // Uses provided length = 10, and default width = 3
    Rectangle rect3(10, 8); // Uses provided values: length = 10, width = 8

    cout << "Area of rect1: " << rect1.area() << endl;
    cout << "Area of rect2: " << rect2.area() << endl;
    cout << "Area of rect3: " << rect3.area() << endl;

    return 0;
}
```

Output:

```
Area of rect1: 15
Area of rect2: 30
Area of rect3: 80
```

Static Members

- Static members in C++ are members of a class that are shared by all instances (objects) of that class.
- There are two types of static members: **static member variables** (also known as class variables) and **static member functions** (also known as class methods).

Static Member Variables

- **Shared Across All Objects:** A static member variable is shared among all instances of a class, meaning there is only one copy of the static variable, regardless of how many objects of the class are created.
- **Class-Level Scope:** Static member variables exist at the class level, not at the object level. They are associated with the class itself rather than any specific object.
- **Lifetime:** The lifetime of a static member variable is the duration of the program. It is created when the program starts and destroyed when the program ends.
- **Accessing:** A static member variable can be accessed directly through the class name or through one of the class's objects.
- يعتبر الStatic Variable هو زى الGlobal Variable بالظبط ، ولكن الفايدة هنا انى ربطه بالClass بتاعه عشان يكون مخصوص ليه ومتحدد له.
- ممكن يكون private او public.

Example:

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

class MyClass {
public:
    static int count;  // Declaration of a static member variable

    MyClass() {
        count++;  // Increment count whenever an object is created
    }
};

// Definition of the static member variable
int MyClass::count = 0;

int main() {
    MyClass obj1;
    MyClass obj2;

    cout << "Number of objects created: " << MyClass::count << endl;
    cout << "Number of objects created: " << obj2.count << endl;
    return 0;
}
```

Static Member Functions

- **<u>Associated with the Class, Not Objects:</u>** Static member functions belong to the class rather than to any specific object. They can be called without creating an instance of the class.
- **<u>Access Only Static Members:</u>** Static member functions can <u>only access other static members</u> (both variables and functions) of the class. They **cannot access non-static members** because those require an instance of the class and the static function itself doesn't have `this` pointer.
- **<u>Utility Methods That Don't Require Object State:</u>** Static member functions are perfect for operations that don't depend on instance-specific data.
  - For example, mathematical functions like conversion functions that don't need to access the state of an object can be made static.

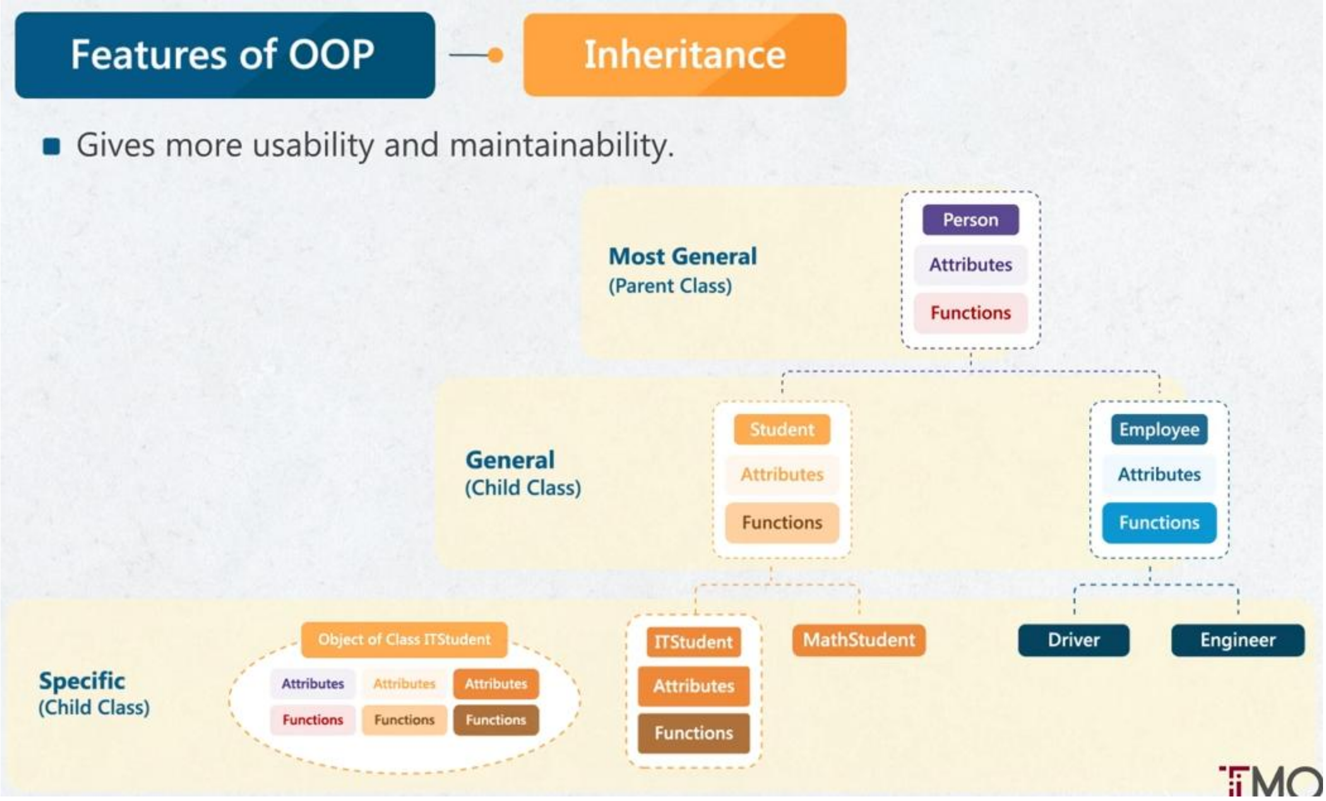
```
class MathUtils {
public:
    static int add(int a, int b) {
        return a + b;
    }
};
```

  - Here, you can call `MathUtils::add(5, 3)` without needing to create an instance of MathUtils.
  - يبقى اخر نقطة دى بتوضحلى ايه اللى ممكن استفاده من الstatic methods ، وهو انى مش هبقى مضطر انى اعمل object من الClass دة عشان استعملها ، انا ممكن استعملها باسم الClass علطول.
  - طب انا ليه اعمل كدة معملش standalone function علطول ، لأن وقتها انا ممكن اجمع بقى الfunctions اللى مرتبطة ببعض في class واحد ، كمان عشان ميحصلش conflicts في الnamespace يعنى ممكن مثلا اعمل 2 add functions ويخصل بينهم conflict لكن لو انا رابط كل واحدة بclass معين مش هيحصل كدة.

<u>Example:</u>

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

class MyClass {
public:
    static int count;  // Static variable

    MyClass() {
        count++;
    }

    static void displayCount() {  // Static function
        cout << "Number of objects created: " << count << endl;
    }
};

int MyClass::count = 0;

int main() {
    MyClass obj1;
    MyClass obj2;
    MyClass obj3;

    MyClass::displayCount();
    obj1.displayCount();
    return 0;
}
```

Inheritance

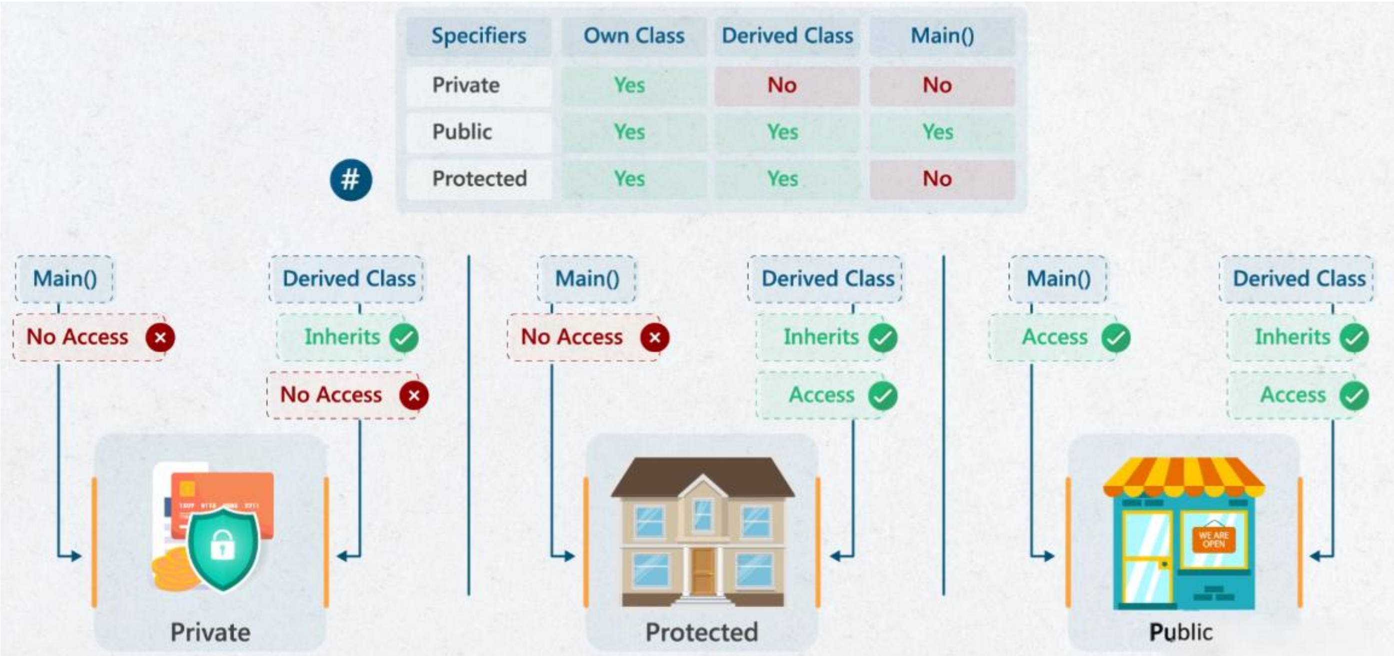


- Inheritance allows a new class (derived class) to inherit attributes and methods from an existing class (base class).
- This promotes code reusability and establishes a <u>relationship (**Is-A**)</u> between classes.
- The access specifiers determine how the members (attributes and methods) of the base class can be accessed in the derived class. These include `public`, `protected`, and `private`.
- ***Types of Inheritance***
  - **<u>Single Inheritance:</u>** A derived class inherits from a single base class.
  - **<u>Multiple Inheritance:</u>** A derived class inherits from more than one base class.
  - **<u>Multilevel Inheritance:</u>** A derived class is created from another derived class, forming a chain.
  - **<u>Hierarchical Inheritance:</u>** Multiple derived classes inherit from a single base class.
  - **<u>Hybrid Inheritance:</u>** A combination of two or more types of inheritance.
- **<u>Syntax:</u>**

```
class BaseClass {
    // Base class members (attributes and methods)
};

class DerivedClass : AccessSpecifier BaseClass {
    // Derived class members (attributes and methods)
};
```

Access specifiers

- Access specifiers in C++ are keywords that set the <u>access level</u> for members (attributes and methods) of a class.
- They determine **how** and **where** the members of the class can be accessed from other parts of the program.
- The **<u>default</u>** access specifier is <u>private</u> by default for classes.



Public

- Members declared as public are accessible from anywhere in the program.
- This means they can be accessed both inside and outside the class.
- There are no restrictions on the access level of these members, making them the most accessible type of members.

```
class MyClass {
public:
    int publicVar;

    void publicMethod() {
        // Accessible everywhere
    }
};

int main() {
    MyClass obj;
    obj.publicVar = 10;       // Accessible
    obj.publicMethod();       // Accessible
    return 0;
}
```

Protected

- Members declared as protected are accessible within the class itself, and in derived classes but not outside the class.
- The protected member will remain accessible to **any class** that is part of the inheritance chain, no matter how many levels deep the hierarchy goes.

```
class BaseClass {
protected:
    int protectedVar;

    void protectedMethod() {
        // Accessible within the class and derived classes
    }
};
```

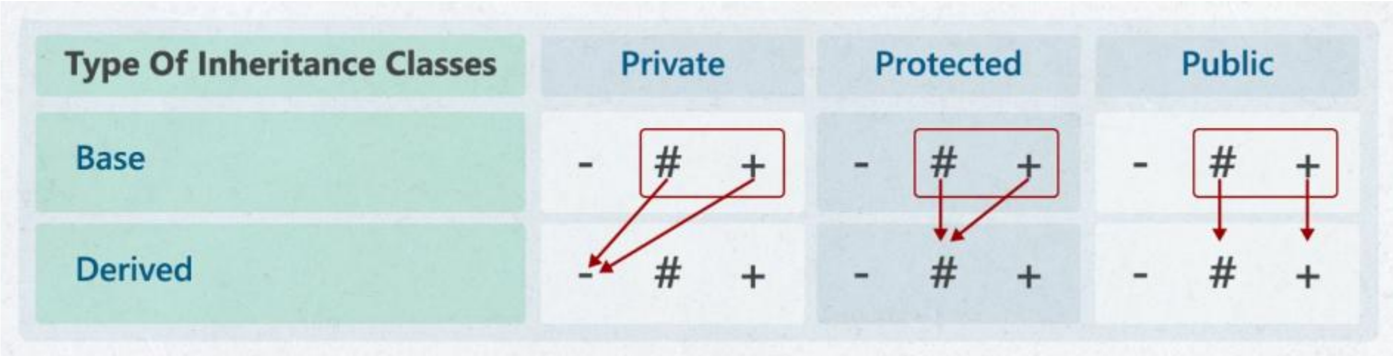
```
class DerivedClass : public BaseClass {
public:
    void accessProtected() {
        protectedVar = 20;     // Accessible here
        protectedMethod();     // Accessible here
    }
};

int main() {
    DerivedClass obj;
    // obj.protectedVar = 10;  // Not accessible (uncommenting will cause an error)
    // obj.protectedMethod();  // Not accessible (uncommenting will cause an error)
    return 0;
}
```

Private

- Members declared as private are accessible only within the class itself.
- They are not accessible from outside the class, **nor** in derived classes.
- Private is the most restrictive access level, making these members entirely hidden from other parts of the program except for the methods of the class they belong to.
- This supports the principle of data hiding and encapsulation.

```
class MyClass {
private:
    int privateVar;

    void privateMethod() {
        // Accessible only within the class
    }
public:
    void setPrivateVar(int val) {
        privateVar = val;      // Modify privateVar within the class
    }

    void callPrivateMethod() {
        privateMethod();       // Call privateMethod within the class
    }
};

int main() {
    MyClass obj;
    // obj.privateVar = 10;    // Not accessible (uncommenting will cause an error)
    // obj.privateMethod();    // Not accessible (uncommenting will cause an error)
    obj.setPrivateVar(10);     // Accessible: modifies privateVar
    obj.callPrivateMethod();   // Accessible: calls privateMethod
    return 0;
}
```

Inheritance Modes



- اللغات الاحدث من الـC++ مش موجود فيها غير الـpublic inheritance بس ، لأن المفروض ان ما يسمح به الإباء يسمح به الأبناء ، فمش منطقى انى اجى اقلل الـAccessibility لحاجة كانت مسموح بيها في الـParent ، بمعنى انه طالما الـparent كان مخلى member معين public لأى حد مش منطقى انى اجى بعدها اخليه Protected.

Public Inheritance:

- **Public** members of the base class remain **public** in the derived class.
- **Protected** members of the base class remain **protected** in the derived class.
- **Private** members of the base class are not directly accessible in the derived class.

```cpp
class  Base
{
public:
   int a;
protected:
   int b;
private:
   int c;
};
```

```cpp
class  Derived : public Base
{
void dosomething()
{
  a = 10; //Allowed
  b = 20; //Allowed
  c = 30; //Not Allowed, Compiler Error
}
};
```

```cpp
int main();
{
 Derived obj;
 obj.a=10; //Allowed
 obj.b=20; //Not Allowed, Compiler Error
 obj.c=30; //Not Allowed, Compiler Error
};
```

Protected Inheritance:

- **Public** members of the base class become **protected** in the derived class.
- **Protected** members of the base class remain **protected** in the derived class.
- **Private** members of the base class are not directly accessible in the derived class.

```cpp
class  Base
{
public:
   int a;
protected:
   int b;
private:
   int c;
};
```

```cpp
class  Derived : protected Base {
void dosomething()
{ a = 10; //Allowed
  b = 20; //Allowed
} c = 20; //Not Allowed, Compiler Error
};
```

```cpp
class Derived2: public Derived {
void dosomethingMore ()
{ a = 10; //Allowed, a is protected member inside Derived
  b = 20; //Allowed, b is protected member inside Derived
} c = 30; //Not Allowed, Compiler Error
};
```

```cpp
int main();
{
 Derived obj;
 obj.a=10; //Not Allowed, Compiler Error
 obj.b=20; //Not Allowed, Compiler Error
 obj.c=30; //Not Allowed, Compiler Error
};
```

Private Inheritance:

- **Public** members of the base class become **private** in the derived class.
- **Protected** members of the base class become **private** in the derived class.
- **Private** members of the base class are not directly accessible in the derived class.

```cpp
class  Base
{
public:
   int a;
protected:
   int b;
private:
   int c;
};
```

```cpp
class  Derived : private Base //Not mentioning private is OK because for
{
void dosomething()
{
  a = 10; //Allowed
  b = 20; //Allowed
  c = 30; //Not Allowed, Compiler Error
}
};
```

```cpp
int main();
{
Derived obj;
obj.a=10; //Not Allowed, Compiler Error
obj.b=20; //Not Allowed, Compiler Error
};
```

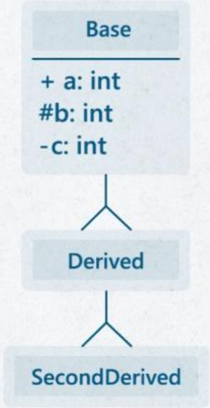
```cpp
class Derived2: public Derived {
void dosomethingMore ()
{ a = 10; //Not Allowed, Compiler Error
  b = 20; //Not Allowed, Compiler Error
} c = 30; //Not Allowed, Compiler Error
};
```

- اخلى بالى من نقطة مهمة في المثال اللى فات دة ان a و b كانوا Accessible عادى جوة Derived لأنهم بقوا private جواه يعنى باق الmembers بتوعه بس هما اللى هيستعملوه لكن مثلا بتوع الDerived2 مش هيقدروا.

<u>يبقى ملخص الAccessibility عندى:</u>

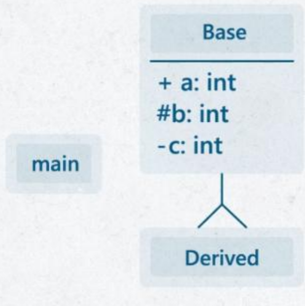
■ Accessibility in SecondDerived Class.

Inheritance Base & Derived / Class Base Members	Private	Protected	Public
+a	NO	YES	YES
#b	NO	YES	YES
-C	NO	NO	NO

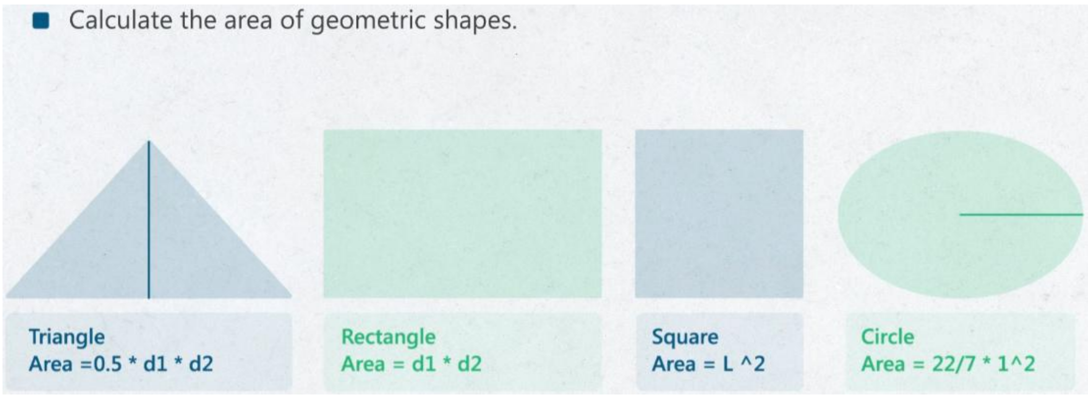


■ Accessibility in Main for object from Derived Class.   Derived obj;

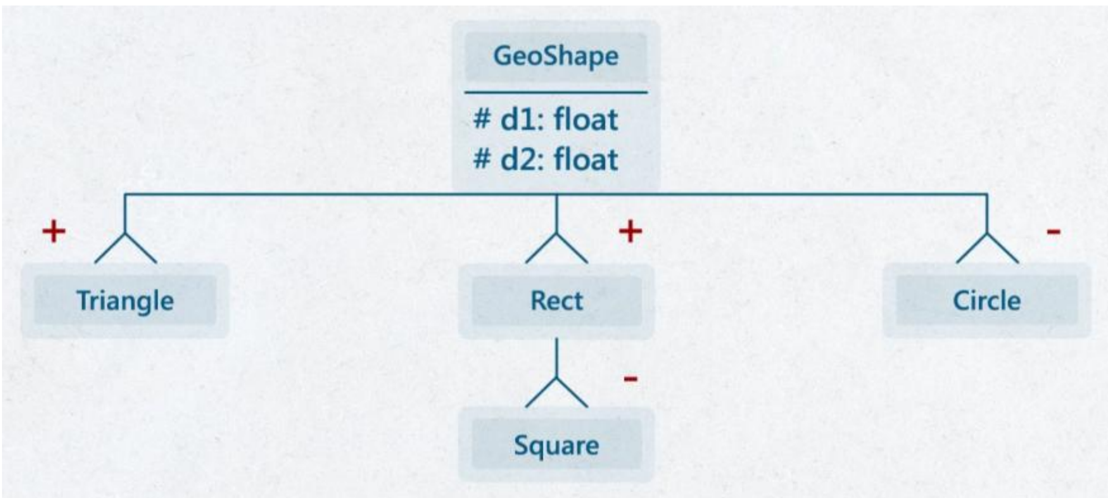
Inheritance Base & Derived / Class Base Members	Private	Protected	Public
+a	NO	NO	YES
#b	NO	NO	NO
-C	NO	NO	NO



Inheritance Modes Example



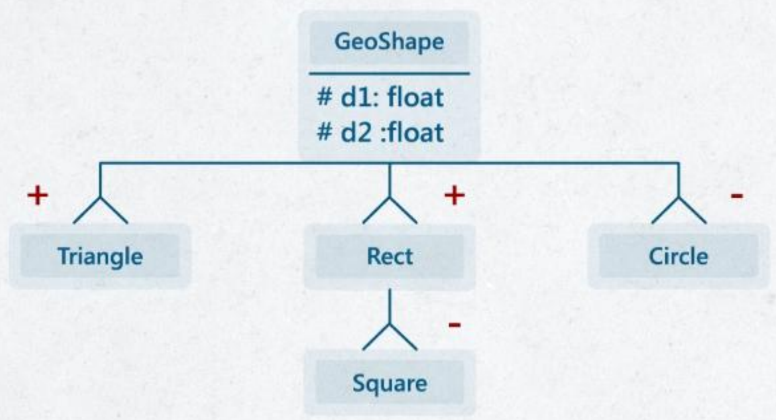
- في المثال دة انا عاوز اعمل Class لكل نوع يحسب مساحة الشكل ، ممكن ابص للموضوع انى دايما عندى 2 Dimensions للشكل عشان احسب منه مساحته ، حتى في المربع والدايرة هعتبر ان البعدين ليهم نفس الطول.



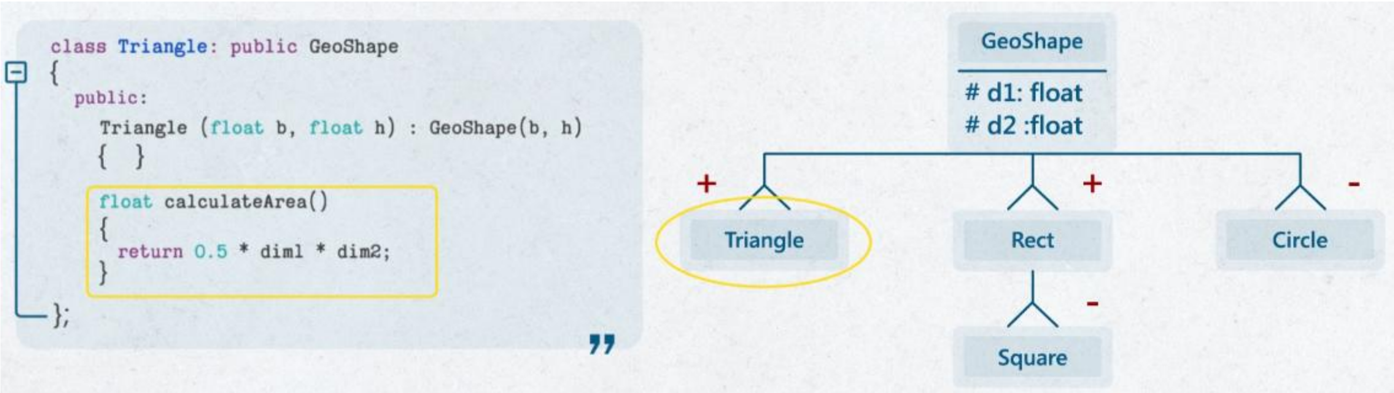
- هنا انا هخلى الTriangle والRect يورثوا Public من الGeoShape لأنهم هيكونوا ليهم نفس المواصفات.
- لكن هخلى الCircle تورث Private عشان اخبى الDimensions 2 واخليهم واحد بس.
- نفس الكلام مع الSquare لكن هخليه يورث من الRect علطول لأنه اقرب ليه.

  - GeoShape:

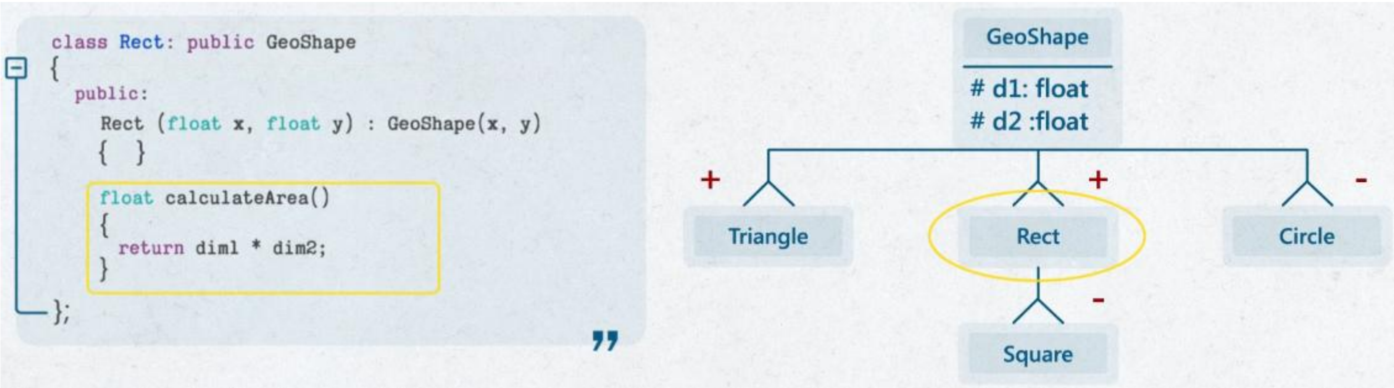
```
class  GeoShape
{
protected:
   float dim1;
   float dim2;
public:
   GeoShape ()
   {dim1 = dim2 = 0;}
   GeoShape (float x)
   {dim1 = dim2 = x;}
   GeoShape (float x, float y)
   {
     dim1 = x;
     dim2 = y;
   }
   void setDimi(float x)
   {dim1 = x;}
   void setDim2 (float x)
   {dim2 = x;}
   float getDim1()
   {return dim1;}
   float getDim2()
   {return dim2;}
   float calculateArea()
   {
     return 0;
   }
};
```



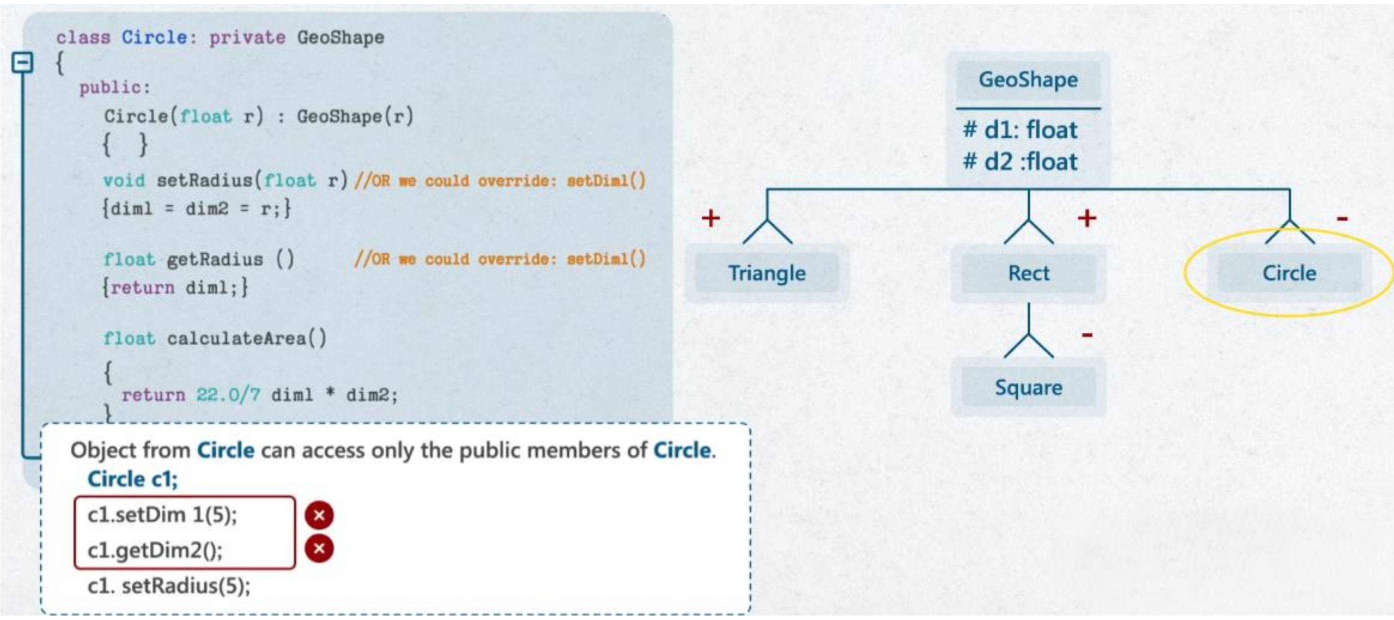
- `Triangle`:



- `Rect`:

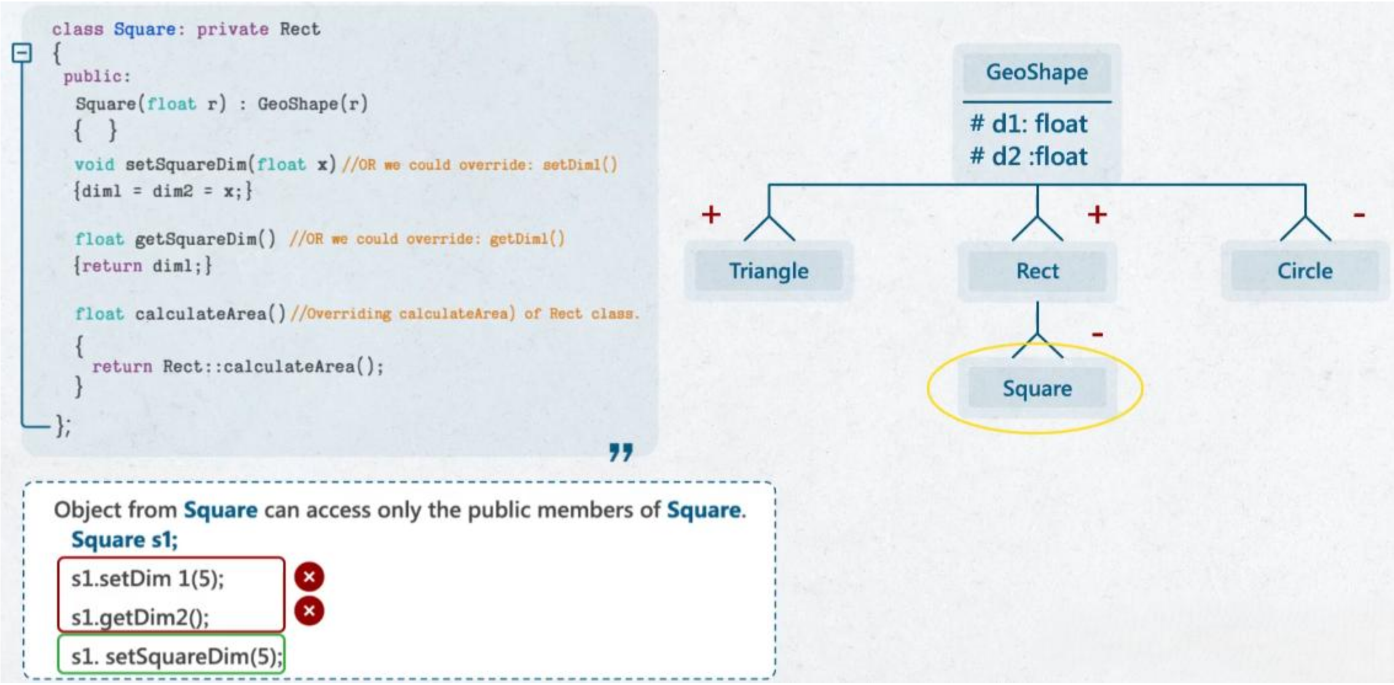


- `Circle`:



- هنا بدل ما عمل member زيادة يبقى اسمه مثلا radius وانا أصلا عندى مكانين بتوع `dim1` و `dim2` مش مستفيد منهم ، انا هعمل methods يحاكوا وكأنى عندى member اسمه radius لكن هما في الحقيقة بيشتغلوا على `dim1` و `dim2`.
- في نفس الوقت خليت الوراثة private عشان اى public methods قديمة في الـ`GeoShape` كانت بتشتغل على الـ`dim1` و `dim2` يبقوا مش متشافين من حد برة بالتالى هخلى الـmethods الجديدة اللى عملتها هي بس اللى متشافة.

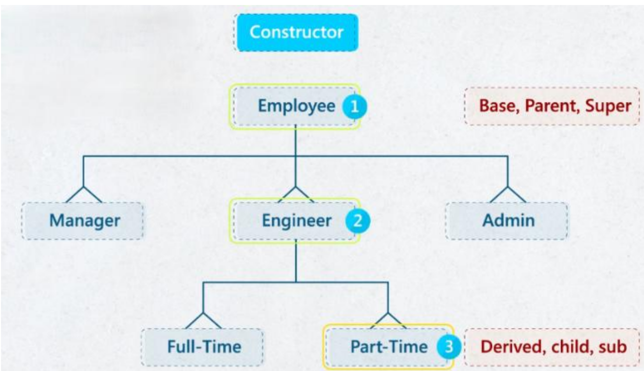
- Square:

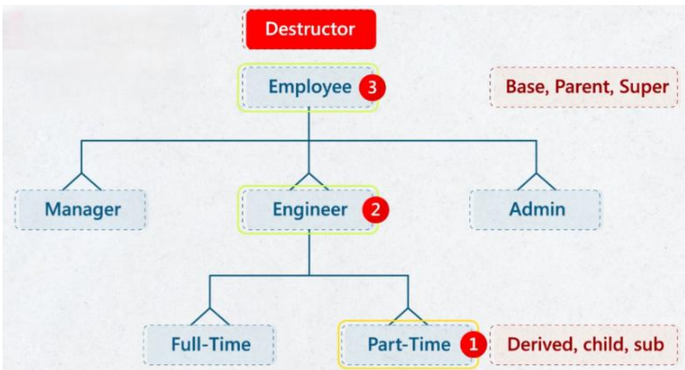


- هنا انا عملت نفس الكلام اللى عملته مع الCircle لكن سواء انى عملت methods جديدة تحاكى انى عندى member جديد اسمه dim ، او انى خليت الوراثة private عشان محدش من برة يستخدم الmethods القديمة.
- بالنسبة للcalculateArea() فهى المفروض هتعمل نفس اللى كانت بتعمله في Rect لكن انا اضطريت انى اعيد تعريفها هنا لأنها متورثة private ، فبالتالي مش هقدر انادى عليها من برة ، وجوة الbody نفسه حددت Rect::calculateArea() عشان في نسختين واحدة من الRect و واحدة من الGeoShape.

Constructor Inheritance

- In C++, constructors are special member functions that are automatically called when an object of a class is created.
- When it comes to inheritance, constructors have specific behaviors that differ from other member functions.
- **Constructors are not inherited:**
  - Derived classes do not automatically inherit constructors from their base classes.
  - Each class must define its own constructors, even if those constructors just call the base class constructors.
  - المقصود هنا ان الConstructors بتاعة الBase Class حتى لو كانت متشافة في الDerived Class فهى مش هيتنادى عليها اوتوماتيك لازم انادى عليها جوة الConstructors بتاعة الDerived Class ، الكلام دة مش بينطبق فقط على الDefault Constructor.
- **Calling Base Class Constructors:**
  - When a derived class object is created, the constructor of the base class is called first, followed by the constructor of the derived class.





    - To explicitly call a base class constructor from a derived class, you use the initialization list in the derived class constructor.
    - **Syntax:**

```
DerivedClass(parameters) : BaseClass(arguments) {
    // Derived class constructor body
}
```

- **Default Constructor Call:**
    - If the base class has a default constructor (a constructor with no parameters), it is **automatically** called when a derived class object is created.
    - If the base class does not have a default constructor, the derived class constructor **must** explicitly call one of the base class's constructors using the initialization list.
- **Multiple Constructors:**
    - If the base class has multiple constructors, the derived class can choose which one to call using the initialization list.

Examples of Inheritance of Constructors

Example 1: Default Constructor in Base Class

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

class Base {
public:
    Base() {
        cout << "Base class default constructor called" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived class constructor called" << endl;
    }
};

int main() {
    Derived obj;  // Base class constructor is called first, then Derived class constructor
    return 0;
}
```

Output:

```
Base class default constructor called
Derived class constructor called
```

Example 2: Parameterized Constructor in Base Class

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

class Base {
public:
    Base(int x) {
        cout << "Base class parameterized constructor called with value: " << x << endl;
    }
};

class Derived : public Base {
public:
    Derived(int y) : Base(y) {  // Calling Base class constructor with a parameter
        cout << "Derived class constructor called with value: " << y << endl;
    }
};

int main() {
    Derived obj(10);  // Base class constructor is called first, then Derived class constructor
    return 0;
}
```

Output:

```
Base class parameterized constructor called with value: 10
Derived class constructor called with value: 10
```

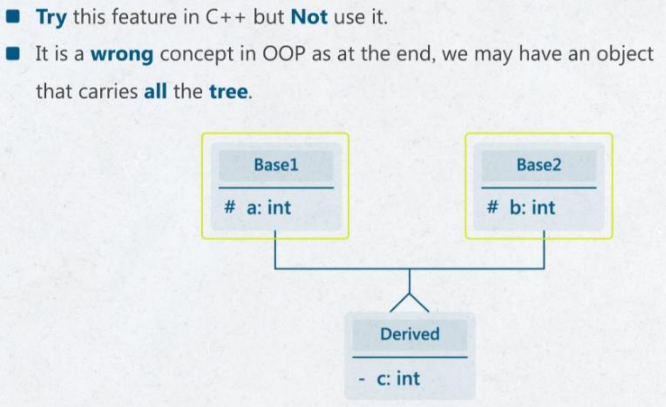
Example 3: No Default Constructor in Base Class

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

class Base {
public:
    Base(int x) {
        cout << "Base class parameterized constructor called with value: " << x << endl;
    }
};

class Derived : public Base {
public:
    Derived(int y) : Base(y) {  // Must call Base class constructor explicitly
        cout << "Derived class constructor called with value: " << y << endl;
    }
};

int main() {
    Derived obj(20);  // Must pass a value since Base class does not have a default constructor
    return 0;
}
```

Output:

```
Base class parameterized constructor called with value: 20
Derived class constructor called with value: 20
```

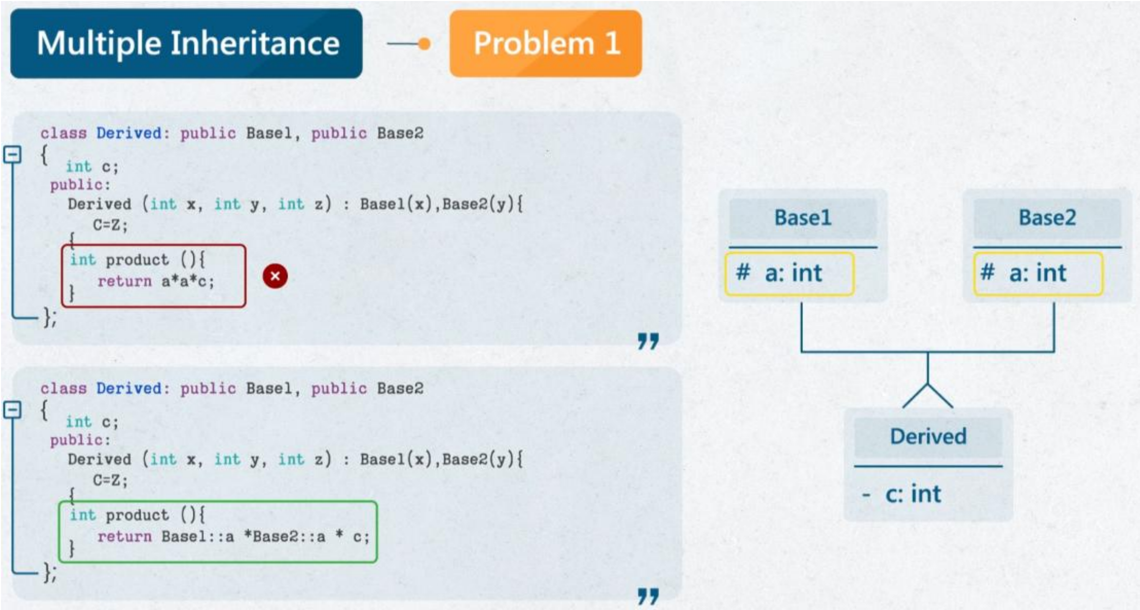
Multiple Inheritance



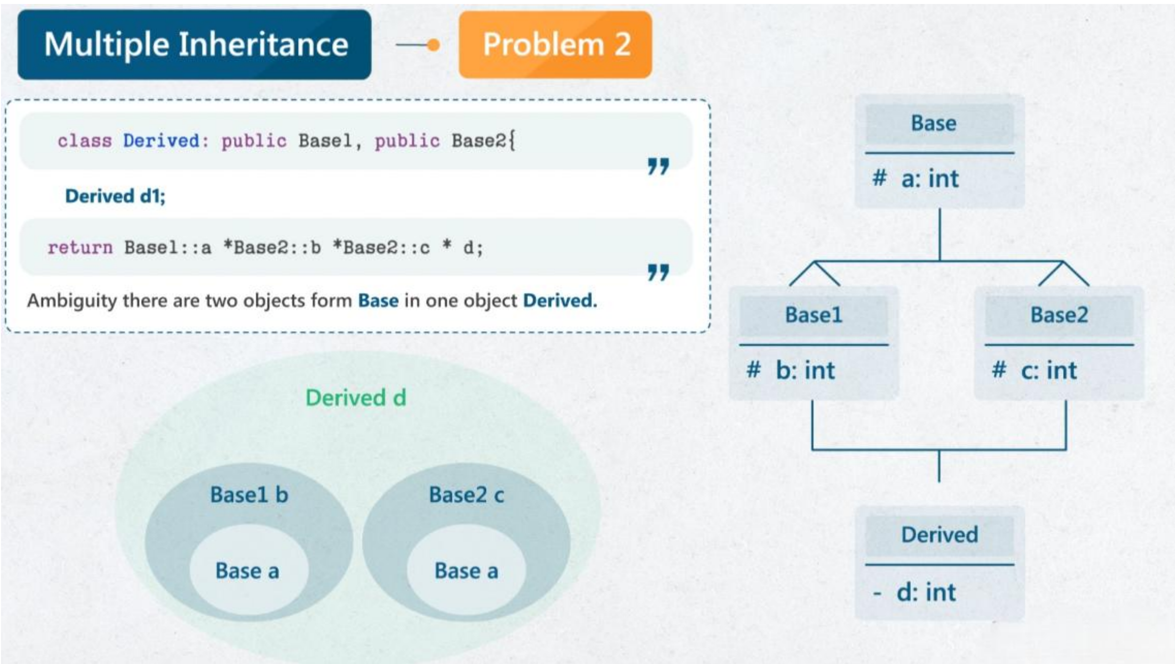
- المشكلة اللى عندى هنا ان هيبقى في object واحد شايل الtree كلها ، يبقى كدة انا مستفدتش حاجة من العلاقات وكان الاحسن انى اعمل class واحد بقى فيه كل حاجة وخلاص.

- واصلا فكرة انى اورث من مكانين دى مفيش مقابل ليها في الحياة الواقعية ، حتى أصلا علاقة انى اورث من ابويا وامى هي فعليا علاقة Association.

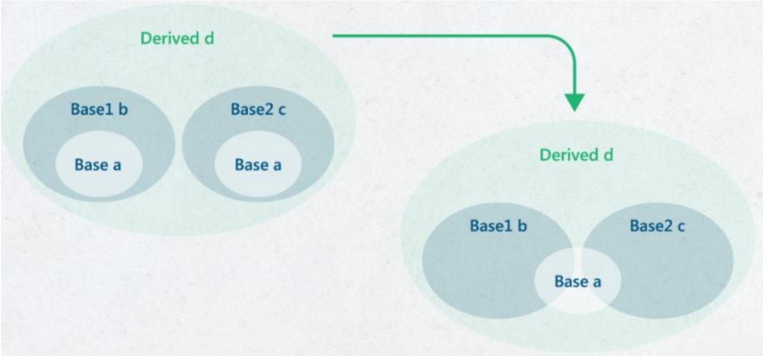
- ممكن تظهرلى مشكلة زى كدة ، وهى انى عندى 2 attributes ليهم نفس الاسم عند كذا class ، بس ممكن احلها زى نص الصورة التانى:



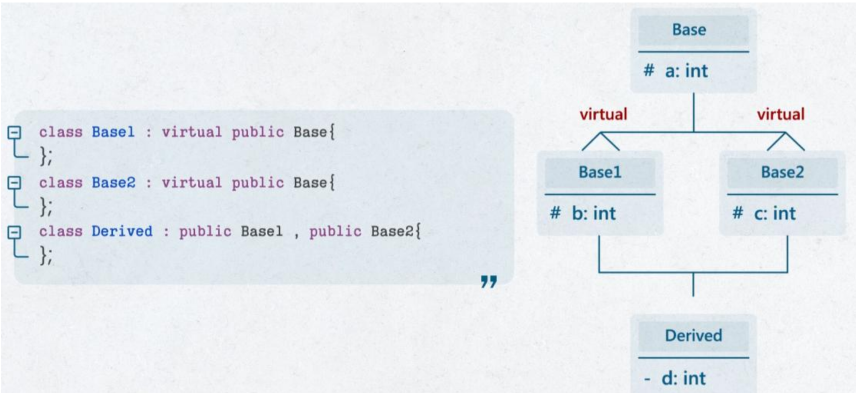
- مشكلة تانية كمان زى كدة **"Diamond Problem"** ، وهى انى هيبقى عندى 2 instances من الـ`Base Class` في النهاية جوة الـ`Derived` ، لأن `Base1` و `Base2` كل واحد فيهم عمل `Base` جواه:



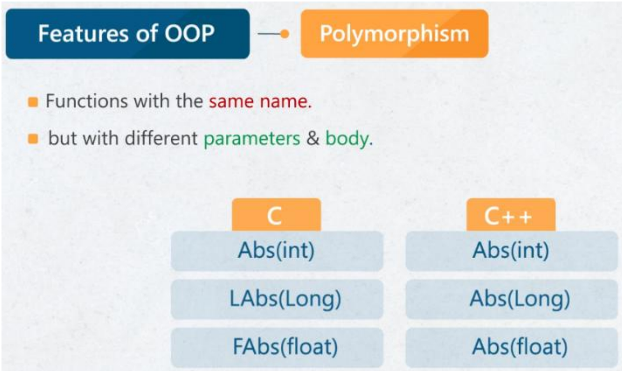
- حل المشكلة دى انى اخلى الtwo parents يبقوا مشتركين في نفس الBase بدل ما كل واحد يعمل لنفسه واحد.



- اللى هعمله بقى انى هخلى الInheritance بتاع كل واحدة منهم virtual (**Virtual Inheritance**)، ودة تأثيره هيظهر بس لما اجى اعمل Derived منهم هما الاتنين ، ساعتها لما اجى اقفل الtree واعمل واحد مشترك لما اجى اعمل constructing بقى هعمل Base مرة جوة Base1 لما واجى اعمل constructing لBase2 هلاقى ان Base موجودة أصلا فمش هعمل ليها constructing تانى.
- وحطيت كلمة virtual عند الاتنين بدل Base2 بس عشان لو عكست الترتيب جوة Derived.



Polymorphism



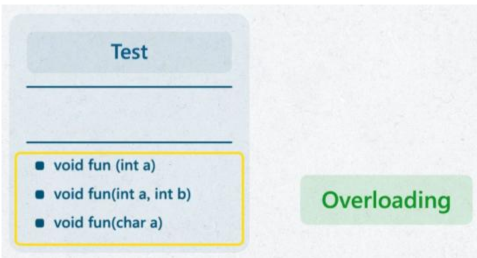
- Polymorphism allows objects of different classes to be treated as objects of a common base class.
- It enables the same function or method to behave differently based on the object that is invoking it.
- It is mainly achieved through function overriding and function overloading.
- In C++, polymorphism is mainly achieved in two ways:
    - **Compile-time Polymorphism** (also known as Static Polymorphism)
    - **Run-time Polymorphism** (also known as Dynamic Polymorphism)

Compile-time Polymorphism

- Compile-time polymorphism, also known as static polymorphism, is resolved during the compilation of the program.
- The decision about which function or method to invoke is made at compile time.
- *Key Features:*
    - **Function Overloading:** Multiple methods in the same class with the same name but different parameters.
    - **Operator Overloading:** Operators can be redefined to perform different operations based on the context.
    - **Early Binding:** The function to be called is determined at compile time, hence the term early binding.

Function Overloading

- Function Overloading occurs when multiple functions in the same scope have the same name but different parameter lists.
- The functions must differ in the signature.
- **Compile-Time Polymorphism:** The function to be executed is determined during compilation.



Function Signature

- The function signature is determined by:
    - **Function Name:** The name of the function or method.
    - **Parameter List:** The number, type, and order of parameters the function or method takes.
- The return type of a function is not part of the method signature.
- The method signature is used by the compiler to differentiate between different functions with the same name (i.e., function overloading).

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

class Example {
public:
    void display(int i) {
        cout << "Integer: " << i << endl;
    }

    void display(double d) {
        cout << "Double: " << d << endl;
    }

    void display(int i, double d) {
        cout << "Integer: " << i << ", Double: " << d << endl;
    }
};

int main() {
    Example obj;
    obj.display(10);        // Calls display(int)
    obj.display(3.14);      // Calls display(double)
    obj.display(10, 3.14);  // Calls display(int, double)
    return 0;
}
```

Operator Overloading

- Operator overloading in C++ is a feature that allows you to redefine the way operators work for user-defined types (like classes).
- Essentially, it lets you create more intuitive and readable code by enabling operators (such as `+`, `-`, `*`, `==`, etc.) to work with objects in a way that makes sense for those objects.
    - For example, if you have a `ComplexNumber` class, you can overload the `+` operator to add two complex numbers using the familiar `+` syntax.
- You can define exactly what happens when an operator is used with your class objects, giving you control over how your objects interact with standard operators.

Unary Operators

- Unary operators operate on a single operand.
- Common unary operators include `+` (sign), `-` (sign), `++`, `--`, and `!`.

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

class Vector {
private:
    int x, y;

public:
    // Constructor
    Vector(int a, int b) : x(a), y(b) {}

    // Overload the unary - operator
    Vector operator - () const {
        return Vector(-x, -y);
    }
```

```cpp
    void display() const {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Vector v1(3, 4);
    Vector v2 = -v1;  // Using the overloaded unary - operator

    v2.display();  // Output: (-3, -4)

    return 0;
}
```

Binary Operators

- Binary operators take two operands.
- Examples include `+`, `-`, `*`, `/`, and `==`.

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

class Vector {
private:
    int x, y;

public:
    // Constructor
    Vector(int a, int b) : x(a), y(b) {}

    // Overload the binary + operator
    Vector operator + (const Vector& other) const {
        return Vector(x + other.x, y + other.y);
    }

    void display() const {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

int main() {
    Vector v1(3, 4);
    Vector v2(1, 2);

    Vector v3 = v1 + v2;  // Using the overloaded binary + operator

    v3.display();  // Output: (4, 6)

    return 0;
}
```

Run-time Polymorphism

- Run-time polymorphism, also known as dynamic polymorphism, is resolved during the execution of the program.
- The function that gets invoked is determined at run time.
- *Key Features:*
    - **Method Overriding:** A derived class provides a specific implementation of a method that is already defined in its base class.
    - **Virtual Functions:** In C++, methods in the base class are marked as virtual to indicate that they can be overridden in derived classes.
    - **Late Binding:** The function to be called is determined at run time, hence the term late binding.

Virtual Functions

- A virtual function is a function in a base class that is overridden by a derived class.
- When you refer to a **derived** object using a pointer or reference to the **base** class, you can call a virtual function, and the derived class's version of the function will be executed.
- لأن الطبيعي هنا ان الpointer اللى من نوع Base هيكون شايف بس الجزء الBase من الDerived object ، بالتالى المفروض ينادى على الmethod بتاعة الBase ، لكن عشان انا محدد نوعها virtual ساعتها هيروح يدور على نسخة ليها جوة الDerived وينفذها.

```
class Base {
public:
    virtual void show() { // Virtual function
        cout << "Base class show function called" << endl;
    }

    void print() { // Non-virtual function
        cout << "Base class print function called" << endl;
    }
};

class Derived : public Base {
public:
    void show() { // Overriding the base class function
        cout << "Derived class show function called" << endl;
    }

    void print() { // Hiding the base class function
        cout << "Derived class print function called" << endl;
    }
};

int main() {
    Base* basePtr;
    Derived derivedObj;
    basePtr = &derivedObj;

    // Virtual function, binded at runtime
    basePtr->show(); // Calls Derived class's show function

    // Non-virtual function, binded at compile time
    basePtr->print(); // Calls Base class's print function

    // Direct call to derived object (Overriding)
    derivedObj.show();  // Calls Derived class's show function
    derivedObj.print(); // Calls Derived class's print function

    return 0; }
```

Output:

```
Derived class show function called
Base class print function called
Derived class show function called
Derived class print function called
```

- في المثال اللى فات دة ، الـbasePtr نادى على show() بتاعة الـderived class بالرغم من ان الـpointer نفسه من نوع Base ، لكن دة حصل عشان هو بيشاور على object من نوع Derived وعلشان كمان الـshow() هي virtual function.
- اما في حالة print() فهو نادى عليها من Base لأن هنا رجع لنوع الـpointer فقط لأنها مش virtual function.
- اخر سطرين بقى انا مش شغال polymorphism عن طريق الـpointer والـvirtual function ، انا بدخل للـmethods بتاعة الـDerived مباشر عن طريق الـobject بتاعه ، لذلك بيحصل الـoverriding function العادى وبيتنادى على الـmethods بتاعة الـDerived.
- اخلى بالى برضو ان الـpointer كان من نوع الـBase وبعدين بقى بيشاور على الـinherited class ، يعنى من الـparent وبيشاور على الـchild ، لكن مكنش ينفع يحصل **العكس**.
- وانا بكتب الـimplementation بتاع الـvirtual function عند الـDerived ممكن استعمل الـkeyword بتاعة override وممكن لا.

```
void show() override { // Overriding the base class function
        cout << "Derived class show function called" << endl;
}
```

Function Overriding

- Function Overriding occurs when a derived class provides a specific implementation of a function that is already defined in its base class.
- **Same Function Signature:** The overridden function must have the same name, return type, and parameters as the base class function.
- **Runtime Polymorphism:** The function to be executed is determined during runtime, not at compile time.
- في الـFunction Overriding انا اقدر اكبر الـAccessibility مقدرش اقللها ، بمعنى لو كانت الـfunction موجودة Public في الـParent يبقى تفضل Public في الـChild ، لو كانت Protected ممكن تبقى Protected او Public ، اما لو كانت Private بقى فانا أصلا مش هقدر اعملها Overriding في الـChild لانى ساعتها مش هبقى قادر اشوفها أصلا.

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

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

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Meow" << endl;
    }
};

int main() {
    Animal* myAnimal = new Dog();
    myAnimal->makeSound(); // Output: Bark

    myAnimal = new Cat();
    myAnimal->makeSound(); // Output: Meow
}
```

- ملحوظة مهمة بالنسبة للFunction overriding ، لو انا كنت عامل Standalone function بتاخد object من نوع Base يبقى ساعتها ممكن ابعت ليها Object من نوع Base او Derived ، لأن الDerived فيه كل الحاجات بتاعة الBase ، ولو انا جوة الfunction ندهت على method معمول ليها overridden فهى هتنادى برضو على الversion بتاعة الBase.
- لكن لو كانت الfunction بتاخد object من نوع Derived فهى مش هينفع تاخد object من نوع Base.

- If function takes a **Base** Type the **Base** or **Derived** object can be sent to it.

```cpp
void someFunction( Base t){
    t.basePublicMemeber();
}
int main(){
    Base b0(5,4);
    Derived obj;
    someFunction(b0);
    someFunction(obj);
}
```

```cpp
void someFunction( Derived t){
    t.derivedPublicMemeber();
}
int main(){
    Base b0(5,4);
    Derived obj;
    someFunction(b0);
    someFunction(obj);
}
```

- If function takes a **Derived** Type the **only Derived** object can be sent to it.

- ملحوظة تانية مرتبطة بالpointers ، لو انا عملت pointer من نوع Base وخليته يشاور على object من نوع Derived ، فهو هيكون شايف الجزء بتاع الBase اللى جوة Derived بس.

```cpp
int main(){
  Derived obj (10,20,30) ;
  Base *pt = &obj;
  cout<<obj.product()<<endl;
  cout<<obj.Base::product()<<endl ;
  cout<<pt->product()<<endl ;
}
```

Derived Version

Base Version

Feature	Function Overloading	Function Overriding
Definition	Same function name, different parameter list	Same function signature in both base and derived classes
Scope	Functions within the same class or scope	Functions across base and derived classes
Binding	Compile-time (Static Binding)	Runtime (Dynamic Binding)
Inheritance	Not required	Required
Return Type	Can be different (but not used for overloading)	Must be the same as the base class
Purpose	To provide multiple functions for similar tasks	To modify or extend the base class function

Static Binding vs Dynamic Binding

Static Binding (Early Binding)

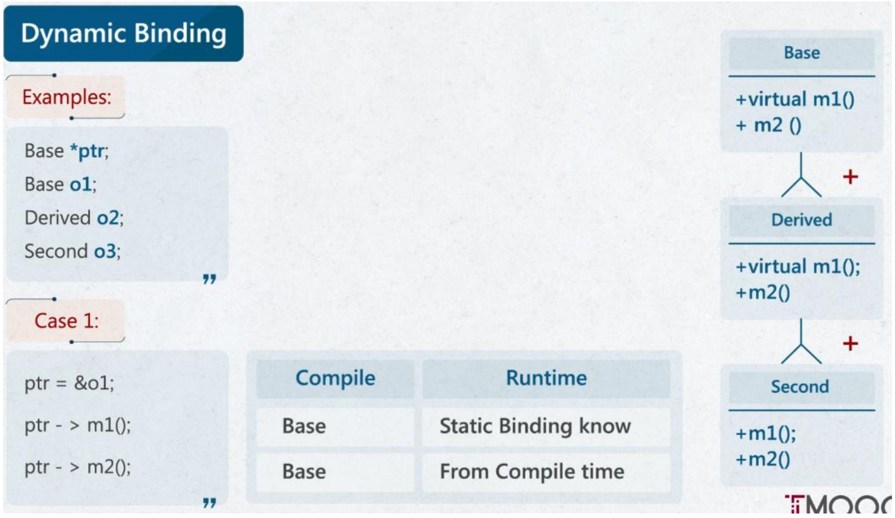
- Static binding, also known as early binding, is when the method or function call is resolved at **compile time**.
- The compiler determines which function to call based on the type of the object or reference.
- Since the binding occurs at compile time, static binding is faster as it doesn't require any overhead of determining the method to invoke during runtime.

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

class Base {
public:
    void show() {
        cout << "Base class show function called" << endl;
    }
};

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

int main() {
    Base obj1;
    Derived obj2;

    obj1.show();  // Calls Base::show
    obj2.show();  // Calls Derived::show
    return 0;
}
```

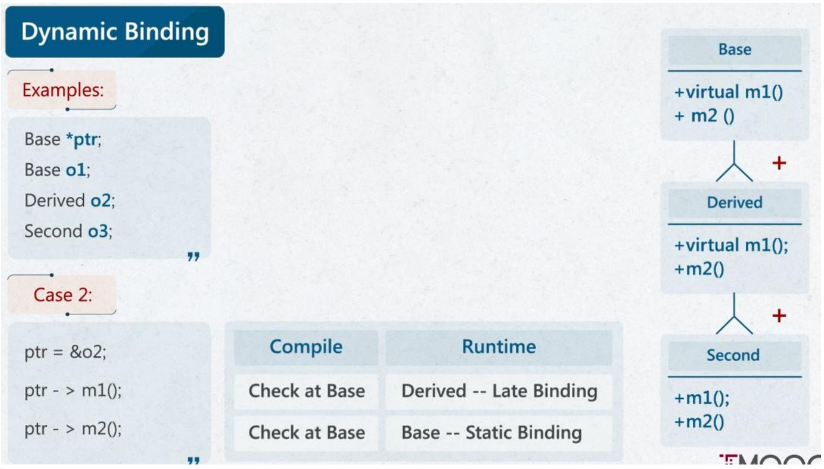
Here, the compiler determines which show function to call based on the type of the object (Base or Derived). The decision is made at compile time.

Dynamic Binding (Late Binding)

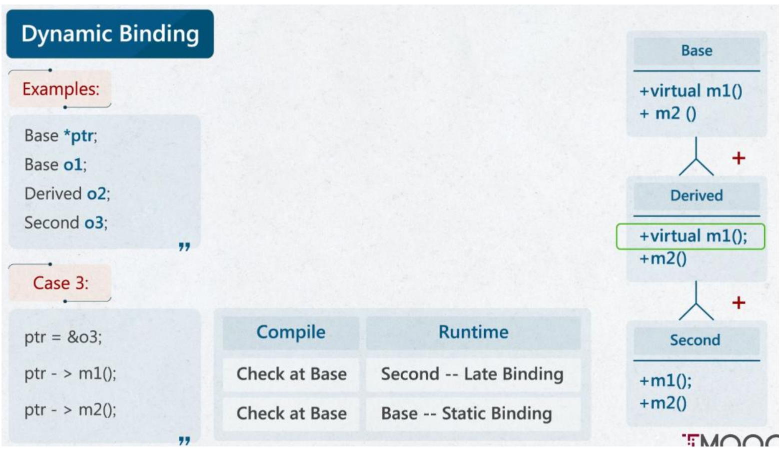
- Dynamic binding, also known as late binding, occurs when the method or function call is resolved at **runtime**.
- This is typically associated with polymorphism where the exact method to be called is determined based on the object type at runtime, rather than at compile time.
- Dynamic binding allows for more flexible and extensible code, as it can decide at runtime which method to invoke.
- It is slightly slower than static binding because the decision is made at runtime.
- **Example:**



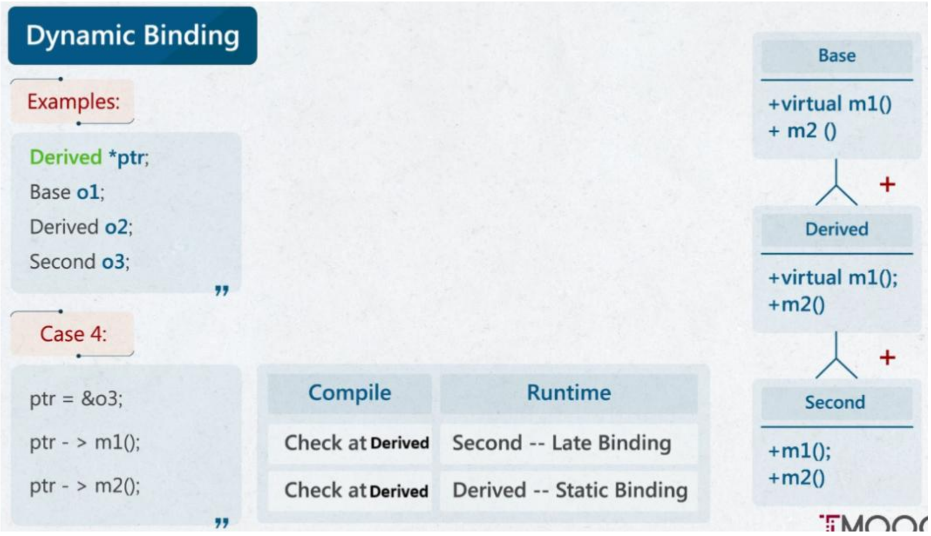
- في الCase الأولى هنا الCompiler بيعمل check على حسب نوع الcaller فهيبص جوة الBase وهيحصل Static Binding ، لأن الpointer والobject من نفس النوع.



- في الCase التانية برضو هيعمل check من جوة الBase لكن هيلاقى `m1()` من نوع virtual فهيعمل ليها Dynamic Binding يعنى هيستنى يشوف هل الDerived فيه تعريف تانى للـ`m1()` ولا لا ، لكن بالنسبة لـ`m2()` فهى مش virtual function بالتالى هيعمل ليها Static Binding.
- طب لو كان `m1()` مش معمول ليها implementation جوة Derived ساعتها هيروح يشغل بتاعة Base عادى.



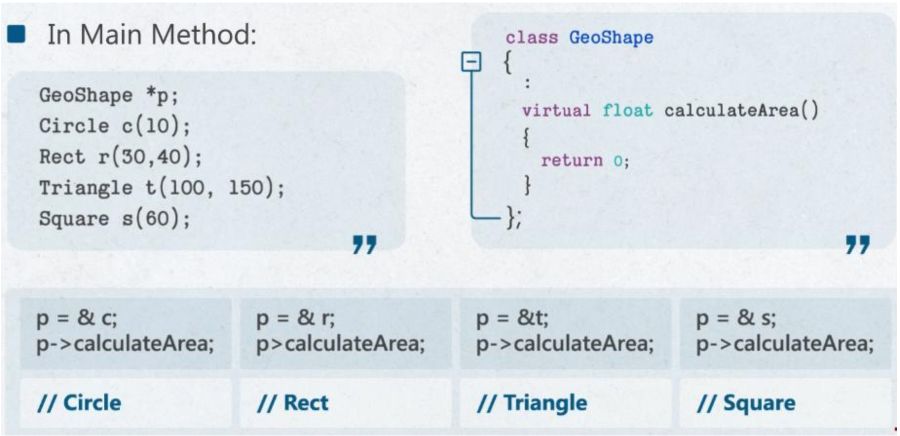
- في الCase التالتة هيعمل زى التانية بالظبط ، طيب لو كان `m1()` مش معمول ليها implementation جوة Second ساعتها هيروح ينادى على الأقرب ليها اللى هي بتاعة Derived.



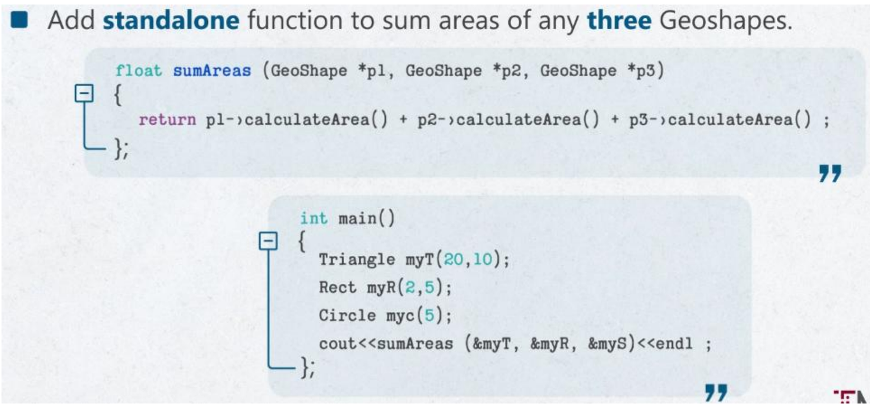
- في الCase الرابعة بقى البوينتر عندى بقى من نوع Derived يبقى كدة الCompiler هيروح يعمل check جوة Derived.

Edited Inheritance Modes Example

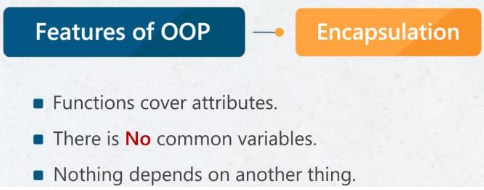
- ممكن بقى اعمل تعديلات على المثال بتاع Inheritance Modes Example زى انى اخلى ال`calculateArea()` عبارة عن virtual function.



- ممكن كمان اعمل standalone function تجمع مساحات اشكال:



<u>Encapsulation</u>



- In C++, encapsulation refers to the <u>bundling</u> of data (variables) and methods (functions) that operate on that data into a single unit called a class.
- Encapsulation also involves <u>restricting access</u> to certain components of an object, which helps protect the data and maintain control over how it is accessed and modified.
- Encapsulation is more about <u>protecting the internal state of an object</u> and providing a public interface to interact with it.
- To interact with an object's data, a class provides a set of <u>public methods</u>, known as the public interface. These methods allow controlled access to the object's internal state.
- Users of the class interact with the object through this interface, which ensures that data is accessed and modified in a controlled and expected manner.

Abstraction

- Abstraction is one of the fundamental principles of Object-Oriented Programming (OOP) that focuses on exposing only the necessary features of an object while hiding the implementation details.
- In other words, it simplifies complex systems by providing a clear and concise interface, without revealing the intricate inner workings of the system.
    - Abstraction is hiding the complex code. For example, we directly use `cout` object in but we don’t know how is it actually implemented.
    - Encapsulation is data binding, as in, we try to combine a similar type of data and functions together.
- The internal workings of a class (like data members and helper methods) are hidden from the outside world.
- Only the necessary functions (methods) are exposed, making it easier to interact with the object without needing to understand its inner complexities.
- In C++, abstraction is achieved through:
    - **Abstract Classes**
    - **Interfaces** (via Pure Virtual Functions)

Abstract Classes

- An abstract class is a class that cannot be instantiated on its own and is designed to be a base class for other classes.
- It can contain both concrete (**fully implemented**) methods and abstract (**unimplemented, or pure virtual**) methods.
- It is used as a base class and typically contains at least **one pure virtual function**.
- A pure virtual function means that any derived class **must** provide an implementation for this function.

```
// Abstract class
class Shape {
public:
    // Pure virtual function (abstract method)
    virtual void draw() = 0;

    // Concrete method
    void setDimensions(int w, int h) {
        width = w;
        height = h;
    }

protected:
    int width, height;
};

class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing Rectangle with width = " << width << " and height = " << height <<
endl;
    }
};

int main() {
    // Shape shape;        // Error: Cannot instantiate abstract class

    Rectangle rect;
    rect.setDimensions(5, 10);
    rect.draw();  // Output: Drawing Rectangle with width = 5 and height = 10
    return 0;
}
```

Interfaces

- In C++, interfaces are usually created using abstract classes that **only contain pure virtual functions**.
- This ensures that derived classes implement all the methods defined in the interface.

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

// Interface (pure abstract class)
class IShape {
public:
    virtual void draw() = 0;   // Pure virtual function
    virtual double area() = 0; // Another pure virtual function
};

class Rectangle : public IShape {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    void draw() override {
        cout << "Drawing a rectangle" << endl;
    }

    double area() override {
        return width * height;
    }
};

int main() {
    Rectangle rect(5.0, 3.0);
    rect.draw();
    cout << "Area: " << rect.area() << endl;

    return 0;
}
```

Final Classifier

- In C++, the `final` specifier is a keyword used to prevent further inheritance of a class or to stop overriding a virtual method in derived classes.
- It ensures that a particular class or virtual function remains unchanged in its current form, adding a layer of control to the class design.

Final Class

- When you mark a class with the final keyword, it cannot be used as a base class.
- This means no other class can inherit from it.

```
class FinalClass final {
    // class members
};
```

Final Virtual Function

- You can also use final to prevent a virtual function from being overridden in derived classes.
- When a function is marked as final, it can’t be overridden by any further derived classes.

```
class Base {
public:
    virtual void show() final {  // 'final' keyword prevents further overriding
        cout << "Base class show function" << endl;
    }
};

class Derived : public Base {
    // Attempting to override the 'final' function will cause a compilation error
    // void show() override { // Error: cannot override 'final' function 'Base::show'
    //     cout << "Derived class show function" << endl;
    // }
};

int main() {
    Derived obj;
    obj.show();  // Calls Base class's show function since overriding is prevented

    return 0;
}
```

Friend Function & Friend Class

- In C++, the concepts of friend function and friend class are mechanisms that allow one function or class to access the private and protected members of another class.
- These are used when you need to allow a specific function or class to interact with a class's internal data in a controlled manner, without exposing those members to the entire world.
- Whether a function or class is a friend, this relationship is **not inherited** by derived classes.

Friend Function

- A friend function is a function that is **not a member** of a class but has access to its private and protected members.
- To declare a friend function, you use the `friend` keyword inside the class whose private or protected members the function needs to access.
- Friendship is not transitive, and each friendship must be explicitly declared.
    - If a function is a friend of a base class, it does not automatically become a friend of its derived classes. Similarly, if a function is a friend of a derived class, it is not a friend of the base class.
    - If `FunctionA` is a friend of `ClassA`, and `ClassA` is a friend of `ClassB`, `FunctionA` is not automatically a friend of `ClassB`.
- The function to be friend can be a stand-alone function of a member function of another class.
- One function can be friend to many classes.
- The friend function violates the Encapsulation concept.
- **Syntax:**

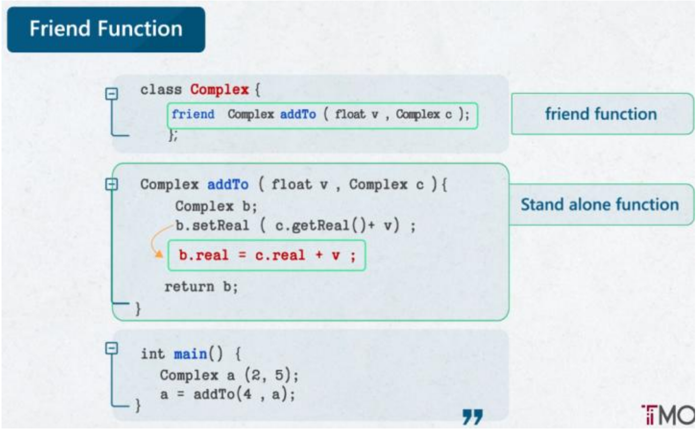
```
class ClassName {
    // Declare the friend function
    friend ReturnType FunctionName(Parameters);
};
```

Example:

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

class Box {
private:
    double width;

public:
    // Constructor to initialize width
    Box(double w) : width(w) {}

    // Friend function declaration
    friend void printWidth(Box box);
};

// Friend function definition
void printWidth(Box box) {
    // Can access the private member width
    cout << "Width of the box: " << box.width << endl;
}

int main() {
    Box box(10.5);
    printWidth(box);  // Calls the friend function to print the width

    return 0;
}
```

- ملحوظة مهمة هنا انى لازم ابعت للfriend function جوة الparameter list بتاعتها object من الclass اللى هي friend ليه ، يعنى مينفعش الfriend function تعمل access للprivate and protected members عن طريق اسم الclass مباشر.

Example:



- في المثال دة انا عندى stand alone function اسمها `addTo` وظيفتها انها بتضيف رقم float على الجزء الreal بتاع complex number معمول من `Complex` Class.
- في العادى عشان أوصل للجزء الreal واعدل فيه لازم استعمل الset والget بتوع الreal attribute ، لكن لو عوزت اكتب سطر ابسط يخليني أوصل للattribute بشكل مباشر يبقى لازم اخلى الstand alone function دى عبارة عن friend function للClass.

Friend Class

- A friend class is a class that has access to the private and protected members of another class.
- When you declare a class as a friend, all member functions of the friend class can access the private and protected members of the other class.
- Friendship is not reciprocal (mutual). Friendship **must** be explicitly declared in **each direction** if needed.
  - If `ClassA` is a friend of `ClassB`, `ClassB` does not automatically become a friend of `ClassA`.
- **Syntax:**

```
class ClassName1 {
    // Declare the friend class
    friend class ClassName2;
};
```

Example:

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

class Engine {
private:
    int horsepower;

public:
    Engine(int hp) : horsepower(hp) {}

    // Friend class declaration
    friend class Car;
};
```

```cpp
class Car {
public:
    void displayHorsepower(Engine engine) {
        // Can access the private member horsepower of Engine
        cout << "Car's engine horsepower: " << engine.horsepower << endl;
    }
};

int main() {
    Engine engine(500);
    Car car;
    car.displayHorsepower(engine);  // Calls the Car method to display engine's horsepower

    return 0;
}
```

Size of Object

The total size of an object in memory in C++ is determined by several factors, including:

1- Size of Member Variables:

- The size of the object's member variables, including both data members (like integers, floats, or custom data types) and inherited members if the class is derived from another class.

2- Padding and Alignment:

- Compilers often add padding between members to ensure proper alignment, which can increase the size of the object.
- Alignment requirements depend on the architecture and the types of data members.

3- Virtual Table Pointer (vptr):

- If the class has any virtual functions, the compiler typically adds a pointer to a virtual table (vtable) to the object.
- This pointer is used to resolve function calls at runtime, which adds to the object's size.

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

class Base {
public:
    int baseVar;
    virtual void func() {}
};

class Derived : public Base {
public:
    int derivedVar;
};

int main() {
    Derived obj;
    cout << "Size of Derived object: " << sizeof(obj) << " bytes" << endl;
    return 0;
}
```

this Pointer

- The `this` pointer is automatically passed to non-static member functions and is not required to be explicitly passed by the programmer.
- The `this` pointer points to the address of the current object on which the member function is invoked.
- It allows member functions to access or modify the calling object's data members and other member functions.
- Only non-static member functions have access to the `this` pointer.
- Static member functions do not have a `this` pointer because they do not operate on a specific instance of the class.
- Example:

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

class MyClass {
private:
    int value;

public:
    MyClass(int value) {
        this->value = value;  // Using 'this' to differentiate between parameter and member
variable
    }

    void display() {
        cout << "Value: " << this->value << endl; // Using 'this' to access the member variable
    }
};

int main() {
    MyClass obj(42);
    obj.display();  // Calls display(), which uses 'this' internally

    return 0;
}
```

Key Use Cases:

1- Avoiding Name Clashes:

- When parameter names are the same as member variable names, the this pointer helps distinguish between them.

2- Returning the Object Itself:

- `this` pointer is used to return the calling object itself, useful in operator overloading and method chaining.

```
MyClass& setValue(int value) {
    this->value = value;
    return *this;  // Returning the current object
}
```

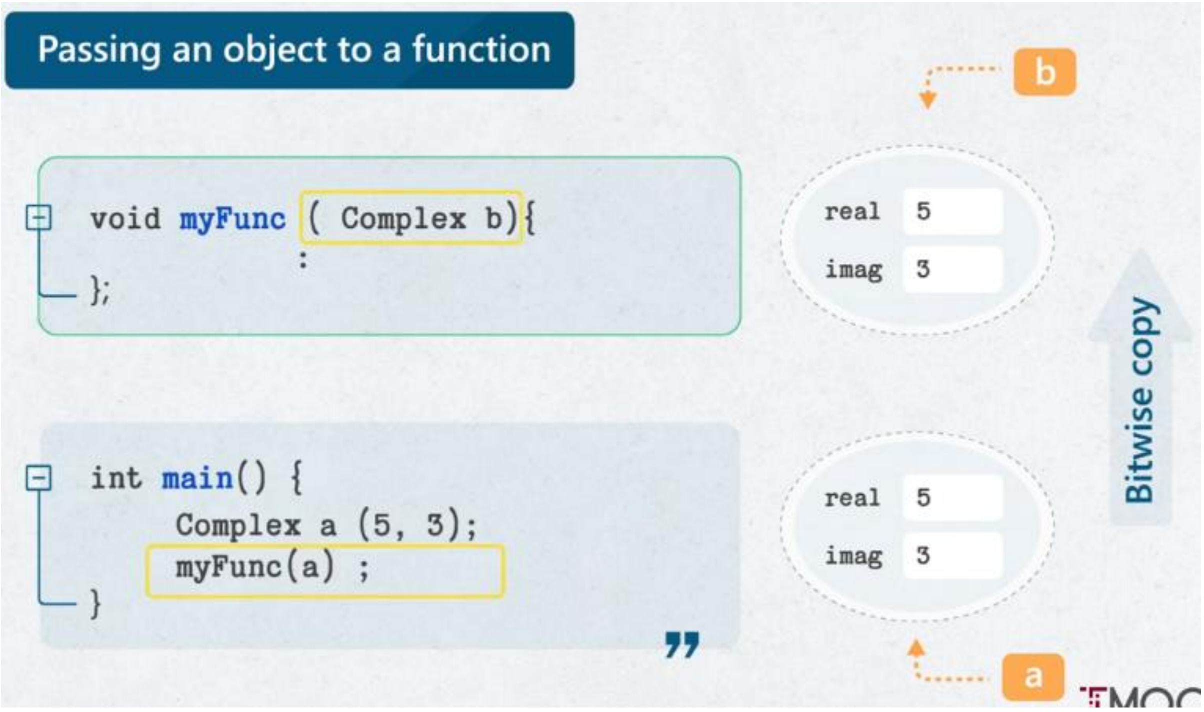
3- Pointer Dereferencing:

- `*this` can be used to dereference the pointer, yielding the object itself.

```
MyClass& func() {
    return *this;  // Returning the object itself
}
```

Passing Object By Value

- When you pass an object by value to a function in C++, a copy of the object is made.
- The function works with this copy, so any modifications made to the object inside the function do not affect the original object.
- This process of copying is known as a bitwise copy (**Shallow Copy**).
    - This means that the binary representation of the object's memory is duplicated directly to create the new object.

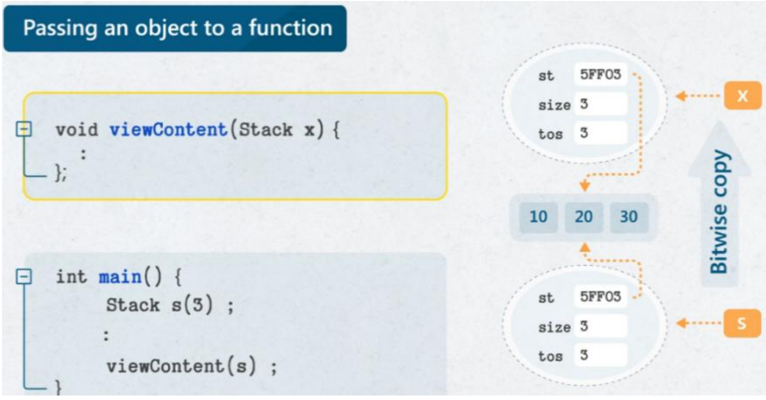


- في المثال اللى فات دة الـ object اللى اتعمل جوة الـmain نادى على الـconstructor بتاع الـClass Complex وعمل initialize للـattributes عن طريقه.
- لكن object b اللى هو local variable اتعمل جوة الـfunction مرحش ينادى على الـconstructor ، هو عمل Bitwise Copy يعنى راح ياخد نسخة من قيم الـattributes بتاعة الـobject a (لأن a هو اللى اتبعت ليه في الـcalling بتاع الـfunction) وعمل initialization عن طريق الـcopy دى.
- يبقى ملخص اللى حصل عندى زى الصورة دى:

1. b is a local variable in myFun and myFun uses call by value.
2. The constructor for complex class will run only once for a.
3. And a will be bitwise copied into b when myFunc is called.
4. At the end of myFun the destructor for complex class will run for b.
5. At the end of main the destructor for complex class will run for a.

Constructor	Destructor
a //main	b //myfun
	a //main

Dynamic area problem



- When an object of a class with dynamic memory allocation is passed by value to a function (including a friend function), the function receives a copy of the object.
- This copying process typically performs a bitwise (shallow) copy.
- A shallow copy simply duplicates the memory address of the dynamically allocated memory (like an array or a pointer).
- As a result, both the original object and the copied object now point to the same memory location.
- If the original object and the copied object share the same dynamically allocated memory, any changes made to this memory through the copy will affect the original object as well.
- When the copied object goes out of scope, its destructor is called, which typically frees the dynamically allocated memory.

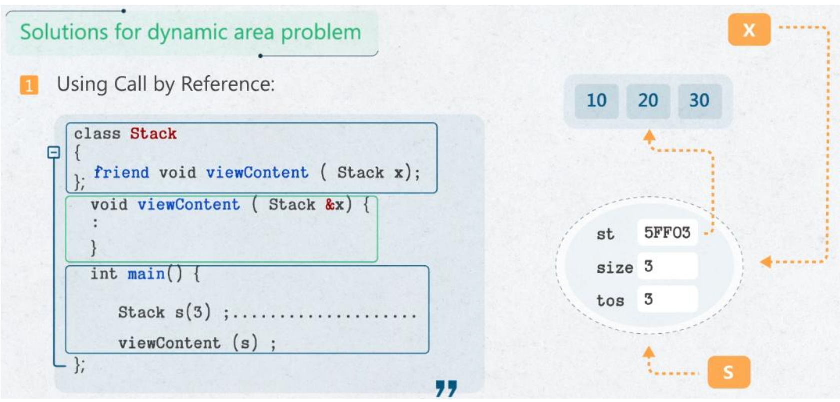
- يبقى المشكلة اللى ظهرت هنا اني لما جيت عملت calling by value لـobject بيستخدم dynamic memory allocation جوة friend function ، الـpassing by value عمل bitwise copy فبالتالي الـobject والـcopy بقى بيشاوروا على نفس الميمورى ، ودة عملى مشكلتين :
  - الأولى ان كدة اى تعديل غير مرغوب فيه هيحصل جوة الـfriend function هيأثر على الـobject الاصلى.
  - التانية ان لما تيجى الـfriend function تخلص والـcopy يبقى out of scope ساعتها هيروح ينادى على الـdestructor بتاعه واللى بالتالى هيمسح المساحة الاصلية بتاعة الـobject.

Solutions for dynamic area problem

Pass by Reference

- Instead of passing the object by value, pass it by reference.
- This avoids copying the object and thus avoids the issues with shallow copying.

```
void viewContent(Stack& x) {
    // Work with the original stack without copying
}
```

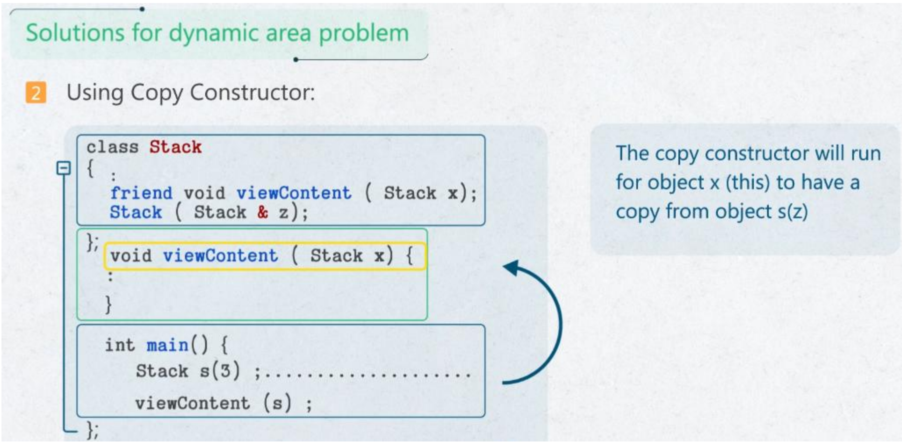


- يبقى هنا انا حليت المشكلة عن طريق اني بدل ما اعمل bitwise copy عملت بس Reference بيشاور على الـobject الاصلى عشان لما يجى الـdestructor بتاع الـreference يشتغل يمسحه هو بس مش هيمسح الميمورى كمان ، بس انا لسه هنا عندى مشكلة انهم بيشتغلوا ع نفس المساحة وبالتالي الحل اللى جاى هيكون recommended اكتر.

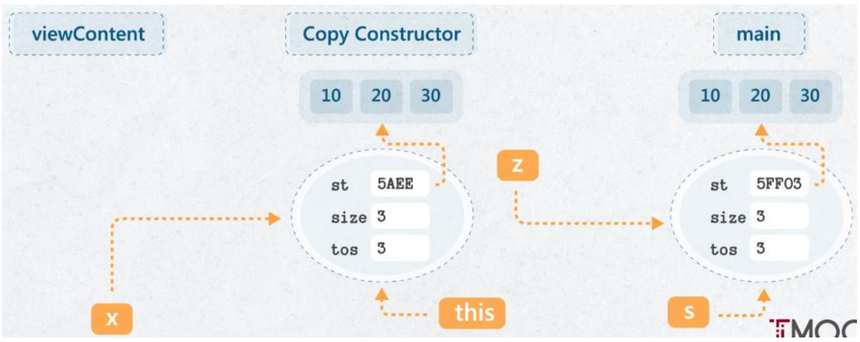
Implement a Deep Copy Constructor

- Implement a copy constructor that performs a deep copy, allocating new memory for the copy instead of just copying the pointer.

```
Stack::Stack(Stack& z) {
    size = z.size;
    tos = z.tos;
    st = new int[size];    // Allocate new memory
    for (int i = 0; i < size; ++i) {
        st[i] = z.st[i];   // Copy the data
    }
    Counter++;             // Because new object is created
}
```



- يبقى الفكرة هنا انى بستخدم Copy Constructor عشان اعمل object جديد خالص وانقل كل محتويات الobject الاصلى جوة الـ copy object في مساحة جديدة من الميمورى مخصوصة للcopy object.



Collections of Objects

In C++, a collection of objects can be created using arrays, and this can be done either through static allocation or dynamic allocation.

Static Allocation

- Static allocation involves creating an array of objects on the stack, where the size of the array is known at compile time.
- The memory for the array is automatically allocated and deallocated by the system when the program enters and exits the scope in which the array is defined.
- The size of the array must be known at compile time and cannot be changed during runtime.

- An array of objects exactly `Complex carr[10];`
- The above line is declare an array of 10 Complex objects and each object call the default constructor.
- May call a different constructor for each object as we need:

```
Complex carr[5] = {Complex(2, 4), Complex),Complex(8)};
```

- If we write constructors to some objects and not to all, the remaining will call the default constructor.
- Like any array in C, we may not write the size in array declaration if we make initialization with constructors.

```
int main()
{
    Complex arr[3] = {Complex(2), Complex(), Complex(5,7)};
    for(int i = 0, i<3; i++)
            arr[i].printComplex();
    getch();
    return 0;
}
```

Dynamic Allocation

- Dynamic allocation involves creating an array of objects on the heap, where the size of the array can be determined at runtime.
- This method allows for more flexibility but requires manual memory management (i.e., you must allocate and deallocate the memory yourself).
- Dynamic allocation on the heap is generally slower than static allocation on the stack due to the overhead of managing heap memory.

```
MyClass* arr = new MyClass[n];  // Dynamic allocation of an array of n objects
```

- Use a pointer to object to make dynamic memory allocation with number of objects allocated in the heap memory and deal with them like array.

Object

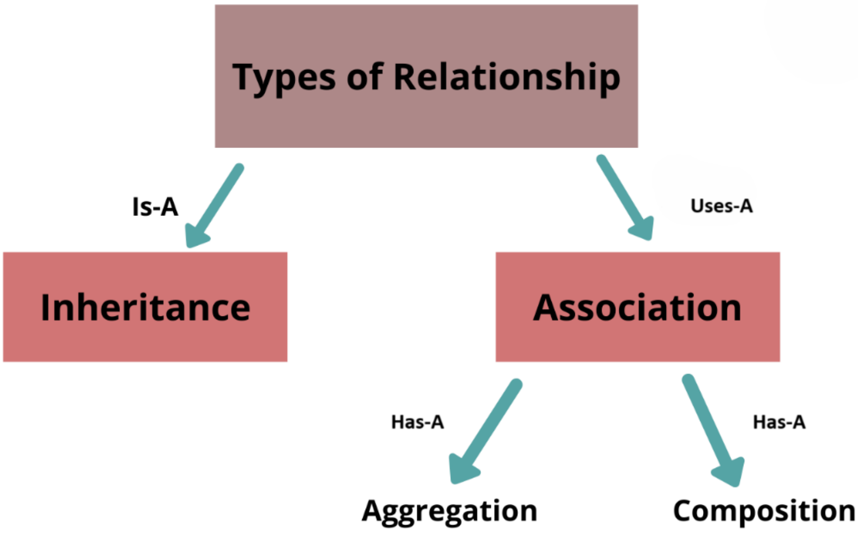
```
Complex * cptr;
```

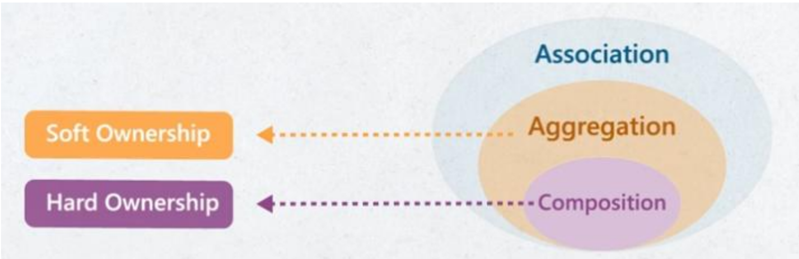
Array

```
cptr = new Complex(2.1, 7.3); // just allocate one Complex
Cptr = new Complex|12]; //just allocate one Complex
                        //but with default constructor only
```

Aspect	Static Allocation	Dynamic Allocation
Memory Location	Stack	Heap
Size Flexibility	Fixed size at compile time	Size can be determined at runtime
Memory Management	Automatic (handled by the system)	Manual (handled by the programmer)
Lifetime	Limited to the scope in which it is defined	Exists until explicitly deleted
Efficiency	Generally, more efficient	Less efficient due to heap management
Risk of Memory Leaks	None	Possible if delete[] is not used

Class Relations





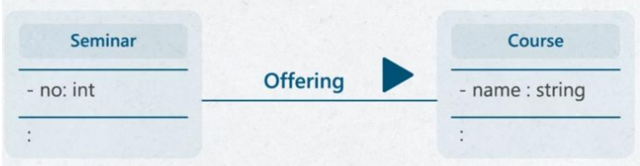
Association

- Association is a relationship between two classes where one class uses or interacts with another class.
- The association is usually depicted as a line connecting the two classes in UML diagrams.
- The objects of the two classes have their own lifecycle and there is no owner.
- Both can be created and deleted independently.

Examples:

- Students are "on waiting list " for a seminar.
- Professors "instruct" a seminar.
- Seminar is an "offering" of courses

- Seminar is an "offering" of courses




```
class Course {
   :
   Course();
}
```

```
class Seminar{
    Course *x;
public:
    Seminar();
    void offer ( Course *C);
}
```

```
int main() {
   Course c1;
   Seminar s1;
   s1. offer (&c1);
}
```

ClassA may be linked to ClassB in order to show that one of its methods include a reference to the another one as a parameter.

Aggregation

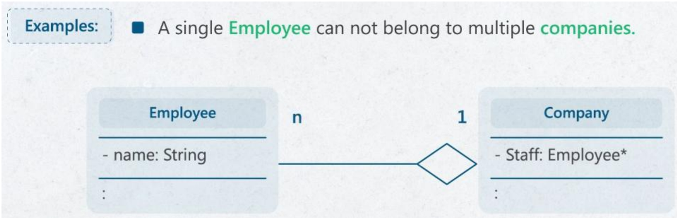
- Aggregation is a specialized form of association where one class contains a **reference** to another class, but the contained class can exist independently of the container.
- It's often referred to as a "has-a" relationship and is typically represented in UML with a hollow diamond.
- The contained object can exist independently of the container.
- The contained object can be shared across multiple container objects.
- If the container is destroyed, the contained object can continue to exist.
- All aggregations are associations but not all associations are aggregations.

Aggregation

- Is a special type of association [strong Association].
- Object of one class "has" an object of the another one.
- Two objects have their own lifecycle and there is one owner at a time.
- Both can created and deleted independently.

Examples:

- Room contains many Tables.



- الAggregation هي Association متعرفة جوة الConstructor مش أي method وخلاص.

```
class Employee {
    :
    Employee();
}

class Company {
    Employee* staff;
public:
    Company (Employee *x){
    staff= x; }
}

int main() {
    Employee emp;
    Company c1 (&emp);
}
```

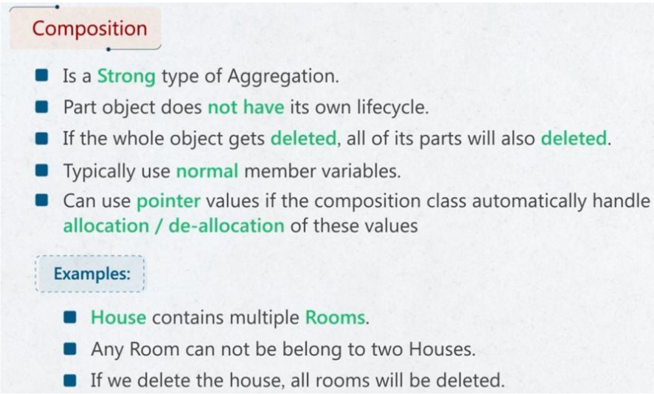
ClassA may be linked to ClassB in order to show that one of its constructors include a reference to the another one as a parameter.

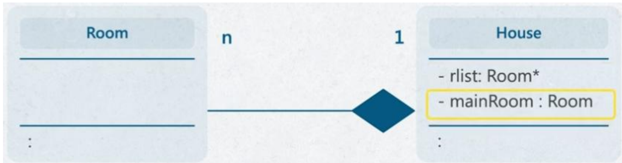
- **Example:**

```
class Department {
private:
    string name;
public:
    Department(string deptName) : name(deptName) {}
    string getName() { return name; }
};

class Employee {
private:
    string name;
    Department* department; // Aggregation: Employee has a reference to Department
public:
    Employee(string empName, Department* dept) : name(empName), department(dept) {}
    void showEmployee() {
        cout << "Employee Name: " << name << ", Department: " << department->getName() << endl;
    }
};

int main() {
    Department* dept = new Department("Sales");
    Employee emp("John Doe", dept);
    emp.showEmployee();

    delete dept; // Even if the department is deleted, the employee object still exists
}
```

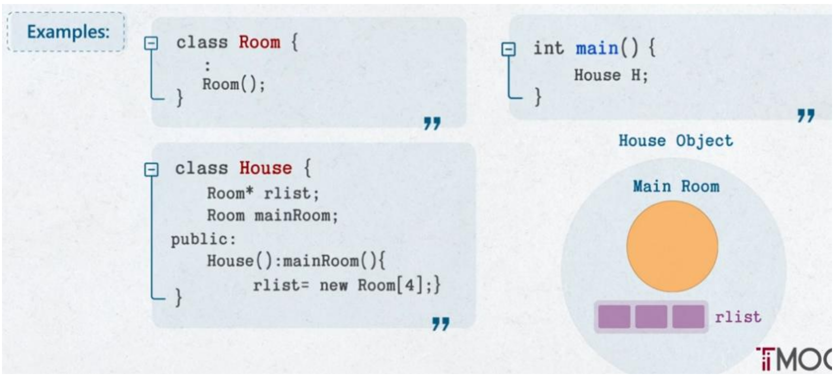
Composition

- Composition is a **stronger** form of association than aggregation.
- In composition, the contained class is entirely dependent on the container class for its existence.
- The child (contained class) **cannot exist independently** of the parent (container class).
- The lifecycle of the contained object is tied to the lifecycle of the container object.
- If the container is destroyed, the contained class will also be destroyed.
- This is often represented with a filled diamond in UML.
- All compositions are aggregations but not all aggregations are compositions.
- علاقة الـComposition بشكل عام هي علاقة نادرة ، يعنى لازم اتأكد كويس اوى قبل ما اقرر ان العلاقة Composition ، والأفضل دايما انى اميل لانى اخلى العلاقة Aggregation.





- في المثال اللى فات دة انا عندى ع الأقل object واحد من الcontained class اللى هو mainRoom موجود كnormal member جوة الContainer class ، يبقى وجودة الـmainRoom معتمد كلى على وجود object من House ولما الobject بتاع House يتمسح هيتمسح هو كمان معاه.



- **Example:**

```
class Engine {
private:
    int horsepower;
public:
    Engine(int hp) : horsepower(hp) {}
    int getHorsepower() { return horsepower; }
};

class Car {
private:
    Engine engine; // Composition: Car contains an Engine object
public:
    Car(int hp) : engine(hp) {}
    void showCarDetails() {
        cout << "Car with " << engine.getHorsepower() << " HP" << endl;
    }
};
```

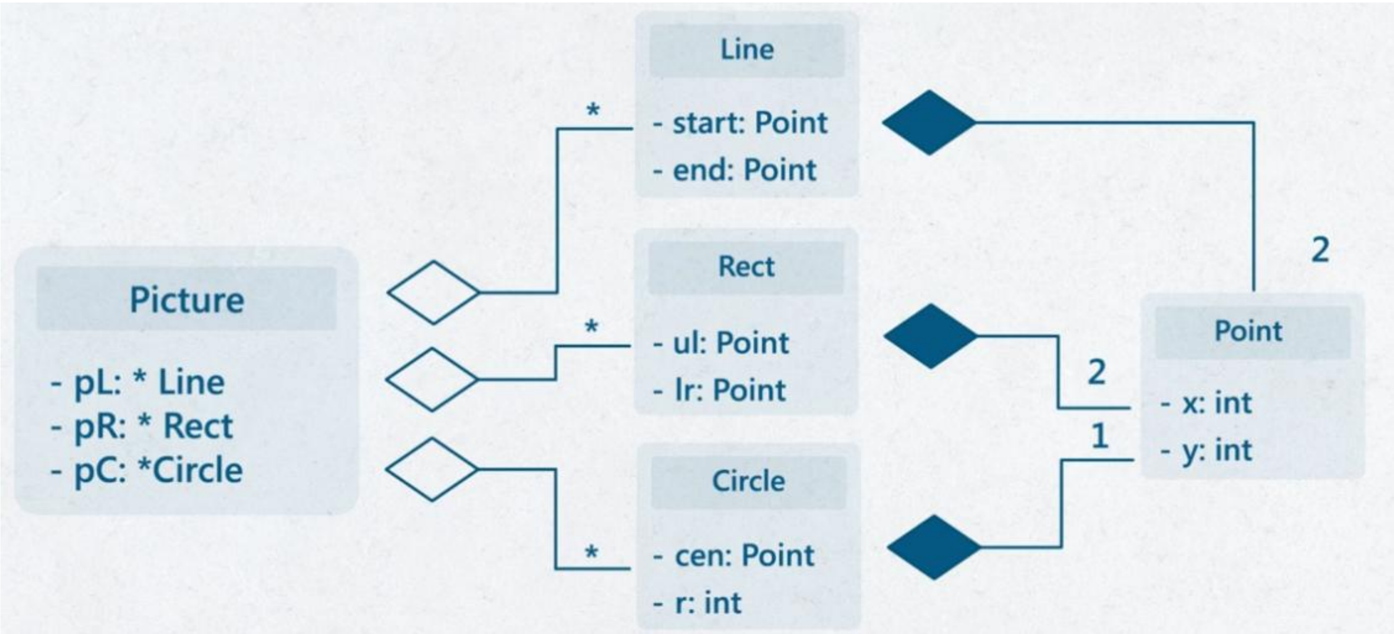
```
int main() {
    Car myCar(250);
    myCar.showCarDetails();
    // The engine object is destroyed when myCar is destroyed
}
```

Aggregations vs Compositions

Aspect	**Aggregation**	**Composition**
Ownership	The contained object (child) can be shared across multiple container objects (parents), and each container only holds a reference to the object.	The contained object is exclusively owned by the container. The contained object is created and destroyed along with the container.
Lifecycle Dependency	The contained object's lifecycle is independent of the container. If the container is destroyed, the contained object continues to exist.	The contained object's lifecycle is dependent on the container. If the container is destroyed, the contained object is also destroyed.
Relationship Strength	Represents a weak "has-a" relationship. The container and the contained object can have different lifecycles.	Represents a strong "has-a" relationship where the contained object's lifecycle is tightly bound to the container.
Representation in UML	Represented with a hollow diamond at the container end of the relationship line.	Represented with a filled diamond at the container end of the relationship line.

Drawing Example

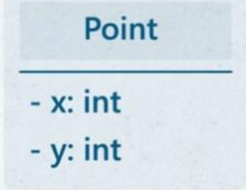
- Create an application to draw a Picture of Lines, Rectangles and Circles.
- Point is the main component of all shapes.



- هنا انا قولت انى محتاج 2 Points عشان اعرف ارسم Line بالتالى العلاقة كانت Composition.
- محتاج 2 Points عشان اعرف ارسم Rectangle بالتالى العلاقة كانت Composition. (النقطتين هما Upper left و Lower right)
- محتاج 1 Point عشان اعرف ارسم Circle بالتالى العلاقة كانت Composition. (كمان محتاج الradius بس دة int member عادى)
- الPicture بقى نفسها هي ممكن يبقى فيها 0 او 1 او n من الLines بالتالى العلاقة كانت Aggregation وعملت reference للlines اللى ممكن الصورة تكون واخداهم ، نفس الكلام مع Rectangle و Circle.

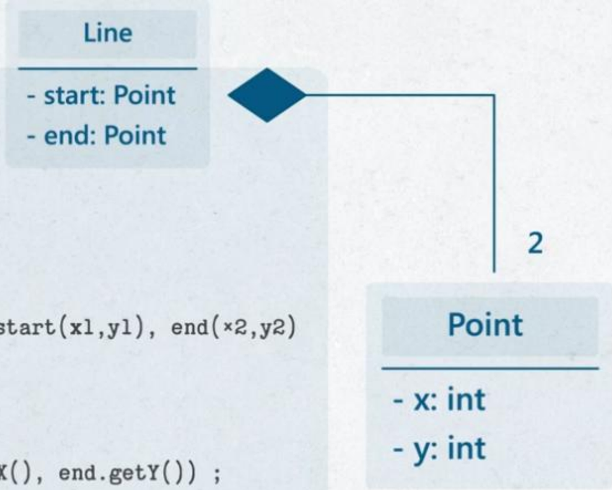
- `Point`:

```cpp
class  Point
{
     int x;
     int y;
   public:
     Point();
     Point (int m, int n);
     void setX(int m);
     void setY(int n);
     int getX();
     int getY();
};
```



- `Line`:

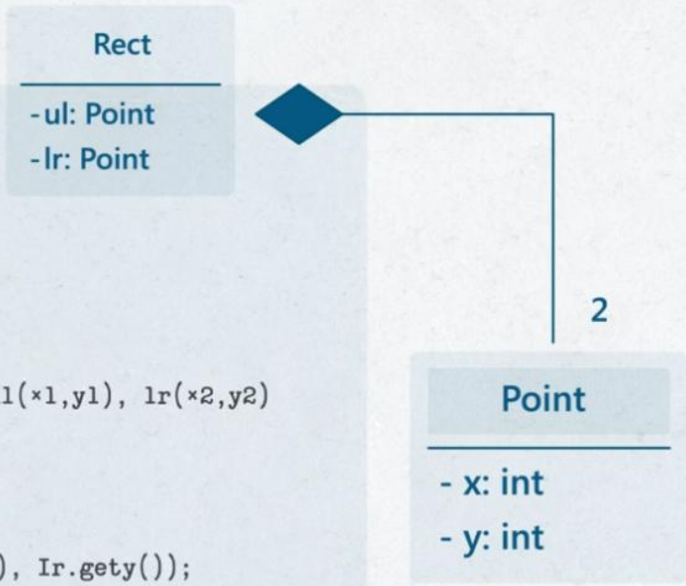
```cpp
class  Line
{  Point start;
   Point end;
 public:
   Line() : start(), end()
   {cout<<"At line Const."; }
   Line(int x1, int y1, int x2, int y2) : start(x1,y1), end(x2,y2)
   {cout<<"At line Const."; }
   void draw()
   {
     line(start.getX(), start.getY(), end.getX(), end.getY()) ;
   }
};
```



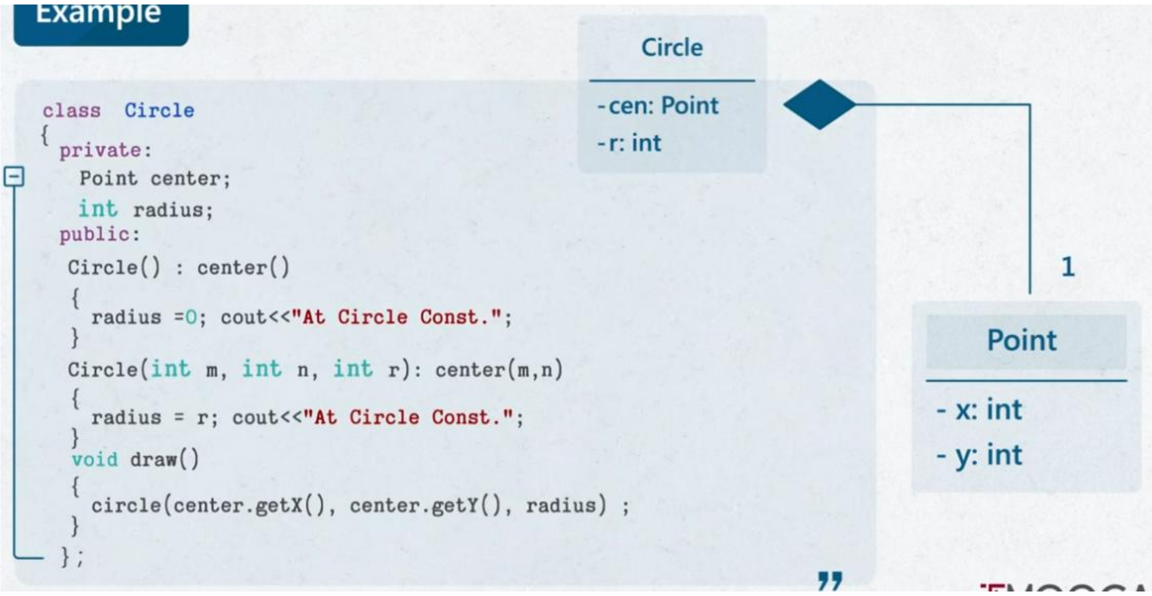
- `Rectangle`:

Example

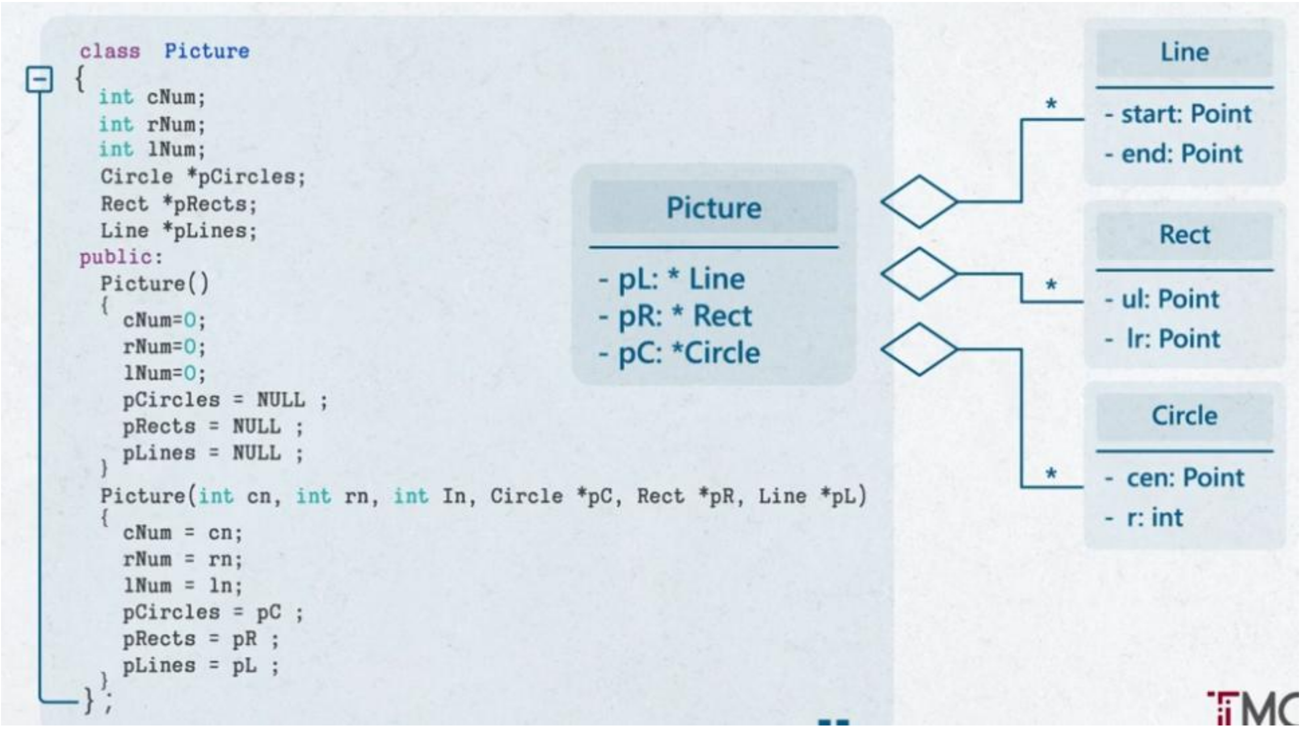
```cpp
class  Rect
{
 private:
    Point ul;
    Point lr;
 public:
   Rect() : ul(), lr()
   { cout<<"At Rect Const."; }
   Rect(int x1, int y1, int x2, int y2) : ul(x1,y1), lr(x2,y2)
   { cout<<"At Rect Const."; }
   void draw()
   {
     rectangle(ul.getX(), ul.getY(), Ir.getx(), Ir.gety());
   }
};
```

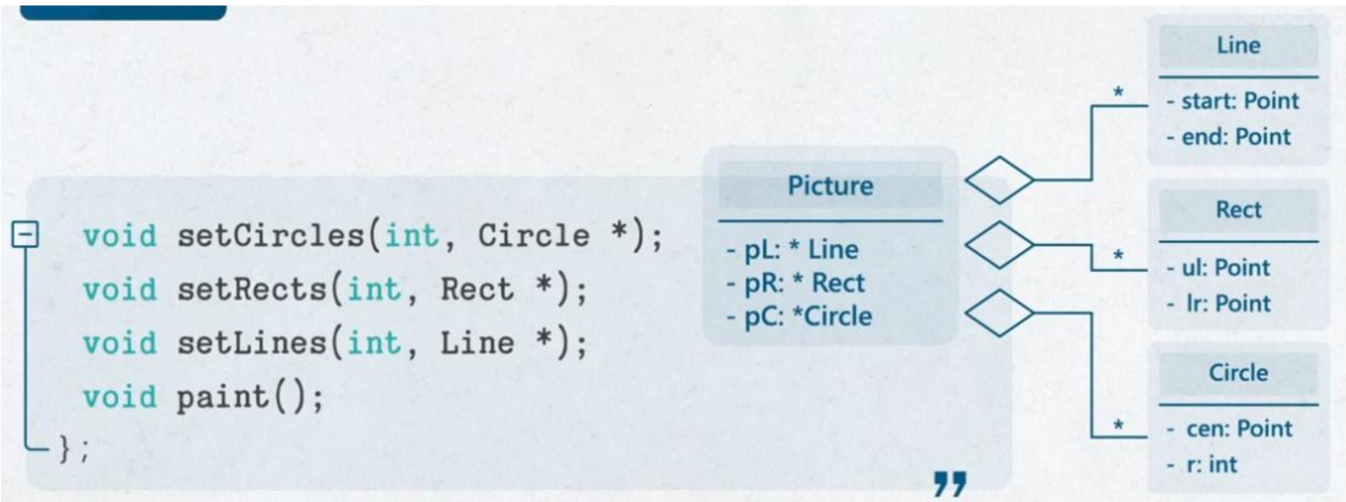


- `Circle`:

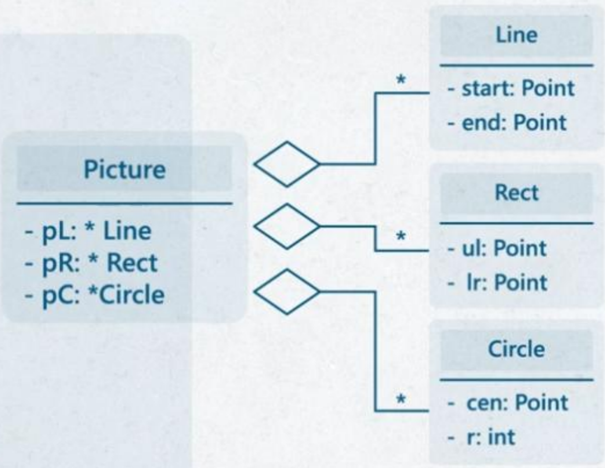


- `Picture`:





```cpp
void Picture:: setLines(int In, Line * lptr)
{
  lNum = ln ;
  pLines = lptr ;
}
void Picture::  paint ()
{
  int i;
  for(i=0; i<cNum ; i++)
  { pCircles[i].draw();
  }
  for(i=0; i<rNum ; i++)
  { pRects[i].draw();
  }
  for(i=0; i<lNum ; i++)
  { pLines[i].draw();
  }
}
```

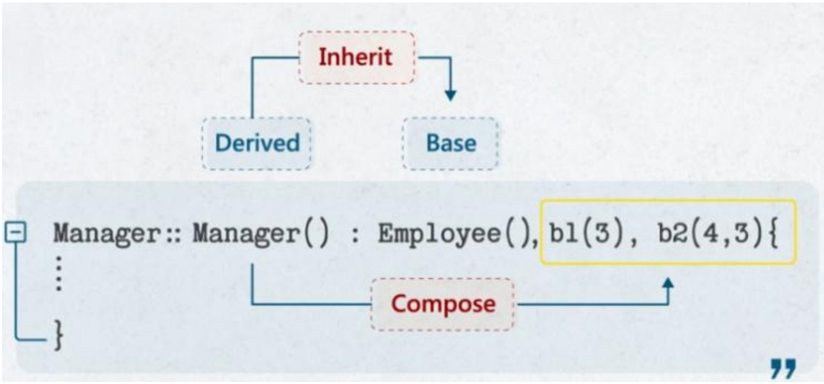


- هنا الconstructor بتاع الPicture بيمثل علاقة Aggregation ، والmethods زى `setCircles()`, `setLines()` , `setRects()` دى بتمثل علاقة Association عشان لو الobjects اتعملوا بعيد عن الPicture object اقدر اربطهم ببعض.
  - `main()`:

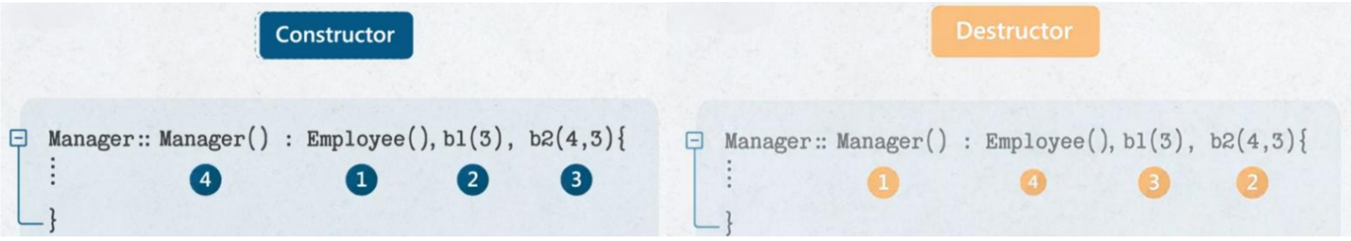
```cpp
//simple main()
int main()
{
  // Graphic Mode
    Picture myPic;

    Circle cArr[3]={Circle(50,50,50), Circle(200,100,100), Circle(420,50,30)};

    Rect rArr[2]={Rect(30,40,170,100), Rect (420,50,500,300)} ;
    Line lArr[2]={Line(420,50,300,300), Line (40,500,500,400)} ;

    myPic. setCircles (3,cArr) ;

    myPic. setRects (2, rArr) ;

    myPic.setLines (2,lArr) ;

    myPic.paint() ;

return 0;
}
```

```cpp
int main()
{     Picture myPic;
  :
 //example on dynamic allocation, using temporary objects (on the fly)
      Line * lArr ;
      lArr = new Line[2];
      lArr[0] = Line(Point(420,50), Point(300,300)) ;
      lArr[1] = Line(40,500,500,400) ;
      myPic.setCircles (3,cArr) ;
      myPic.setRects(2,rArr) ;
      myPic.setLines(2,lArr) ;
      myPic.print() ;
      delete[] lArr ;
}
```

Inheritance & Composition Example



- في المثال دة انا عندى class Manager بيعمل inheritance لـclass Employee وفى نفس الوقت عنده علاقة composition مع class تانى معرفوش لكن class Manager عنده جواه 2 objects من الclass اللى معرفوش دة.
- طيب عرفت منين ان العلاقة كانت composition ، لأن انا بنادى على الconstructor بتاع b1 و b2 وقت ما بنادى على الconstructor بتاع Manager.
- ترتيب الconstructors والdestructors هيكون كالتالى بقى:



- تطبيق كود على المثال دة ممكن يكون زى كدة:

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

// Base class
class Employee {
public:
    Employee() {
        cout << "Employee constructor called" << endl;
    }
};

// Class for composing objects
class Budget {
    int amount;
    int duration;

public:
    // Constructor for Budget
    Budget(int a, int d) : amount(a), duration(d) {
        cout << "Budget constructor called with amount: " << amount << " and duration: " <<
duration << endl;
    }
};

// Derived class inheriting from Employee and composing Budget objects
class Manager : public Employee {
    Budget b1;
    Budget b2;
```

```
public:
    // Constructor for Manager
    Manager() : Employee(), b1(3, 3), b2(4, 3) {
        cout << "Manager constructor called" << endl;
    }
};

int main() {
    // Creating an instance of Manager
    Manager m;
    return 0;
}
```

Template Class

- A template class in C++ is a way of writing generic and reusable code that can work with any data type.
- By using template classes, you can define a blueprint for a class without specifying the data types it will work with until the class is instantiated.
- This allows you to create classes that are more flexible and can handle different data types without duplicating code.
- Templates are expanded at compiler time. This is like macros. The difference is that the compiler does type-checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.
- هنا انا مش بوفر عدد الClasses خالص ، بالعكس دة انا عندى نفس عدد الClasses وزيادة عليهم الTemplate كمان ، احنا بس قللنا عدد سطور الكود وخليناه more generic.
- **Syntax:**

```
template <typename T>
class MyClass {
private:
    T data;  // T can be any data type

public:
    MyClass(T value) : data(value) {}

    void display() {
        cout << "Data: " << data << endl;
    }

    T getData() {
        return data;
    }
};
```

- **Explanation:**
  - `template <typename T>`: This declares a template with a type parameter `T`. The `typename` keyword indicates that `T` is a placeholder for a data type. You can also use `class` instead of `typename`, and it has the same meaning in this context.
  - `T data`: The class now has a member variable data of type `T`, which can be any type specified when the class is instantiated.
  - `MyClass(T value)`: The constructor takes an argument of type `T` and initializes the data member.
- **Example Usage:**

```
int main() {
    // Create an object of MyClass with int type
    MyClass<int> intObj(100);
    intObj.display();  // Output: Data: 100

    // Create an object of MyClass with double type
    MyClass<double> doubleObj(99.99);
    doubleObj.display();  // Output: Data: 99.99

    // Create an object of MyClass with string type
    MyClass<std::string> stringObj("Hello, Templates!");
    stringObj.display();  // Output: Data: Hello, Templates!

    return 0;
}
```

- **Defining a Class Member Outside the Class Template:**

```
template <class T>
class ClassName {
    ... .. ...
    // Function prototype
    returnType functionName();
};

// Function definition
template <class T>
returnType ClassName<T>::functionName() {    // code      }
```

- **Another Example:**

```
template <class T>
class Stack
{
private:
   int top;
   int size;
   T *ptr;
   static int counter ;
public:
   Stack();
   Stack(int n);
   ~Stack();
   static int getCounter();
   Stack(Stack &) ;
   void push(T);
   T pop();
   Stack& operator= (Stack&);
   friend void viewContent (Stack);
};
//static variable initialization
template <class T>
int Stack<T>::counter = 0 ;

template <class T>
Stack<T>::Stack()
{
    top = 0;
    size = 10;
    ptr = new T[size];
    counter++;
}
```

```
int main()
{
    Stack<int> s1(5);

    cout << "\nNumber of Integer Stacks is: " « Stack<int>::getCounter();
    s1.push(10);
    s1.push(3);
    s1.push(2);

    cout << "\n1st integer: " << s1.pop();
    cout << "\n2nd integer: " << s1.pop();

    Stack<char> s2;

    cout << "\nNumber of Character Stacks is:" << Stack<char>::getCounter();
    s2.push('q');
    s2.push('r');
    s2.push('s');
    viewContent (s2);
    cout << "\n1st character: " << s2.pop();
    cout << "\n2nd character: " << s2.pop();

return 0;
}
```

- **Class Templates With Multiple Parameters:**

```
template <typename T1, typename T2>
class MyPair {
private:
    T1 first;
    T2 second;

public:
    MyPair(T1 a, T2 b) : first(a), second(b) {}

    void display() { std::cout << "First: " << first << ", Second: " << second << std::endl;  }

    T1 getFirst() { return first; }

    T2 getSecond() { return second; }
};

int main() {
    // Create an object of MyPair with int and double types
    MyPair<int, double> intDoublePair(42, 3.14);
    intDoublePair.display();  // Output: First: 42, Second: 3.14

    // Create an object of MyPair with string and int types
    MyPair<std::string, int> stringIntPair("Age", 30);
    stringIntPair.display();  // Output: First: Age, Second: 30
```

```
    // Create an object of MyPair with char and bool types
    MyPair<char, bool> charBoolPair('X', true);
    charBoolPair.display();  // Output: First: X, Second: 1

    return 0;
}
```

About the Author

Connect with me on Linkedin: mohaned-ahmad

Explore more at: GitHub Repository