CHAPTER-5

Inheritance

What is inheritance?

Definition:
Inheritance is the process by which an object of one class acquires the properties and behaviors (methods) of another class. It allows one class to reuse the fields and methods of another, promoting code reusability and logical hierarchy.

Key Idea:
It enables sharing of common characteristics from a parent (base) class to a child (derived) class, without modifying the existing class. This helps in extending functionality and maintaining clean code.

Example:

Consider the following hierarchy:

- "Robin" is a type of "Flying Bird"
- "Flying Bird" is a subclass of "Bird"
- Thus, the class Robin inherits properties from Flying Bird, which in turn inherits from Bird.

Some terminology

- Base Class / Parent Class: The class whose properties are inherited.
- Derived Class / Child Class: The class that inherits properties from the base class.
- **Advantages of Inheritance**

1. **Code Reusability:**
   Common features and logic can be written once in a base class and reused by derived classes, reducing code duplication.
2. **Extensibility:**
   New features can be added to existing code without changing the original code, by simply extending the base class.
3. **Improved Maintainability:**
   Changes in the base class automatically reflect in all derived classes, making code easier to update and maintain.

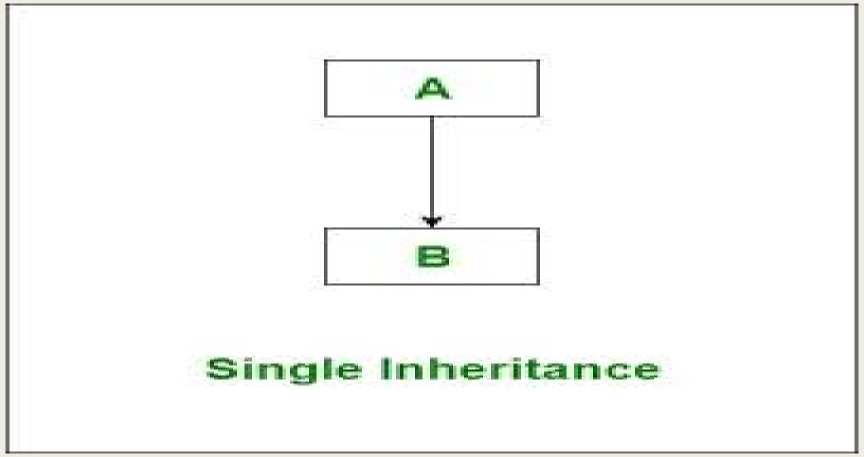
Types of inheritance

Here are the **types of inheritance in C++:**

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

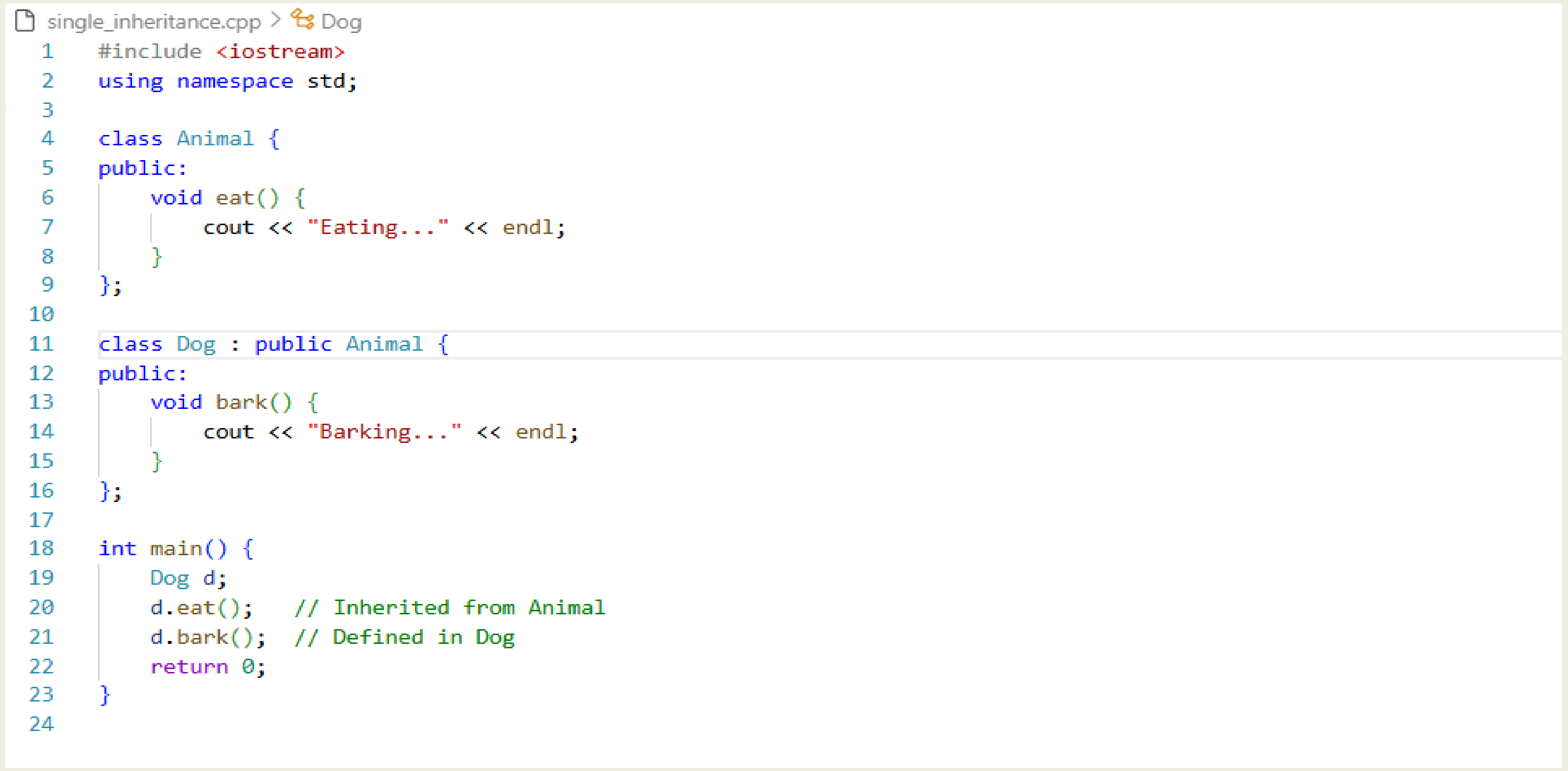
Single Inheritance

Single Inheritance is a type of inheritance in which a **derived class inherits from only one base class.** It allows the child class to reuse the methods and properties of the parent class.



Prepared by Ayush Lamsal | |
layush788@gmail.com

Single Inheritance



single_inheritance.cpp > Dog

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

class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};

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

int main() {
    Dog d;
    d.eat();    // Inherited from Animal
    d.bark();  // Defined in Dog
    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

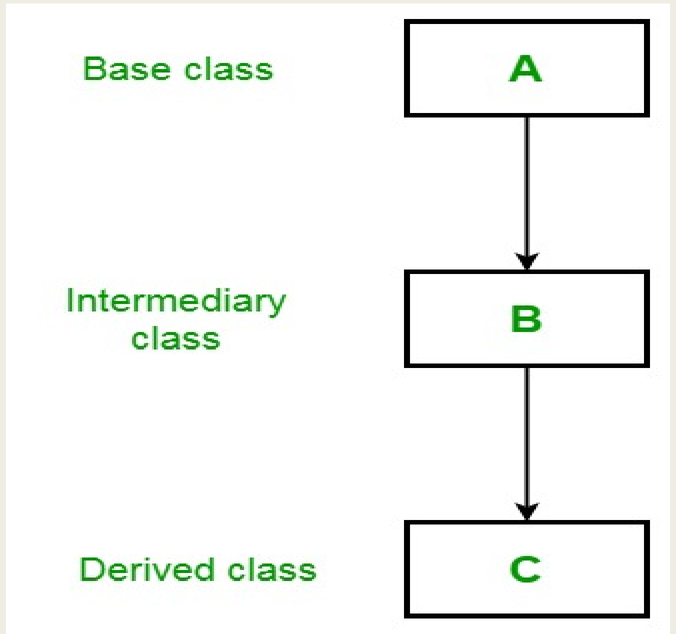
Multilevel Inheritance

Multilevel Inheritance is a type of inheritance where a **class is derived from a derived class**, forming a **chain of inheritance.** Each new class inherits features from the one before it.

In this inheritance:

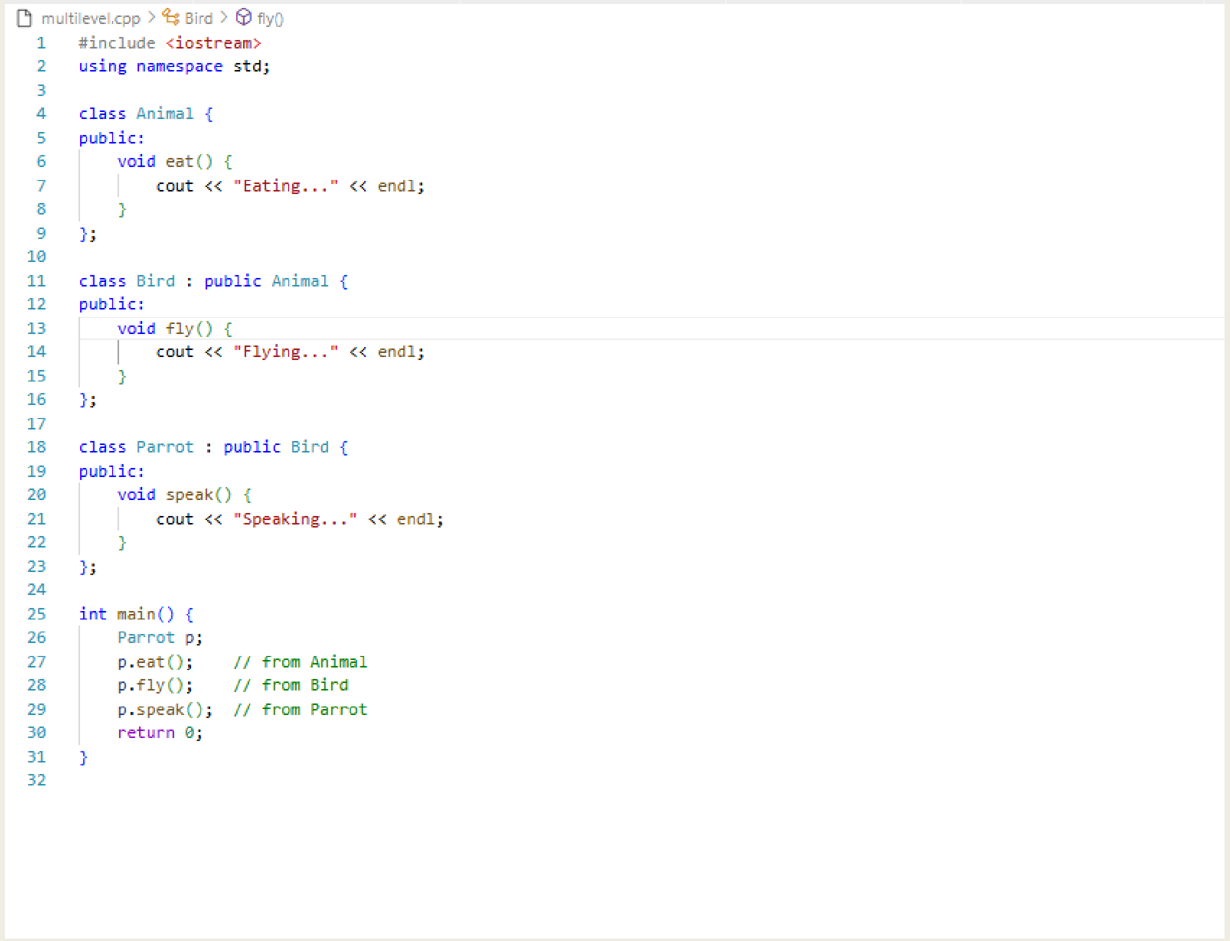
- Class B inherits from Class A.
- Then Class C inherits from Class B. So, Class C gets the properties of both A and B.

This helps in building a layered class structure, where each level adds new features.



Prepared by Ayush Lamsal | |
layush788@gmail.com

Multilevel Inheritance



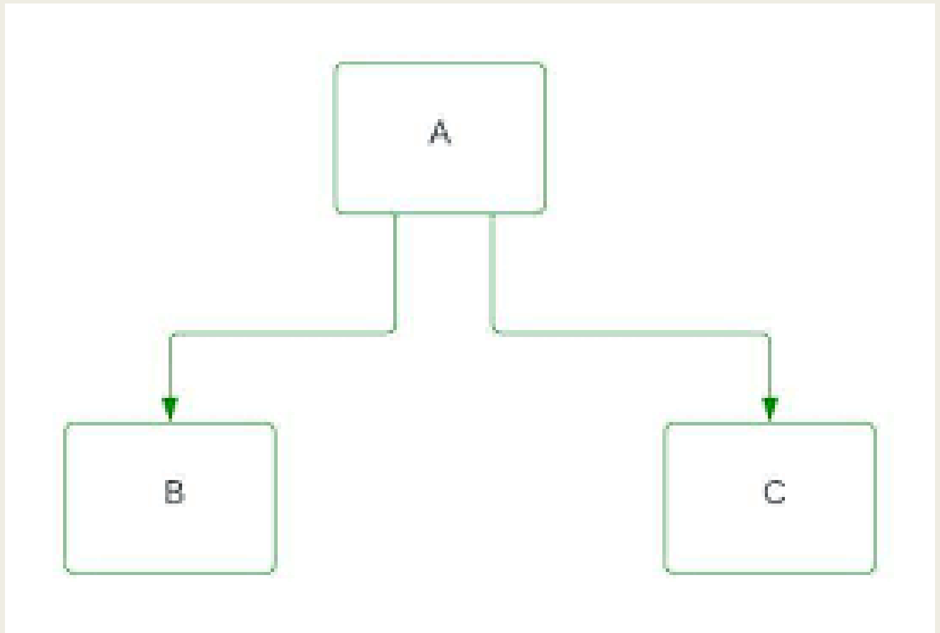
multilevel.cpp > Bird > fly()

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

class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};

class Bird : public Animal {
public:
    void fly() {
        cout << "Flying..." << endl;
    }
};

class Parrot : public Bird {
public:
    void speak() {
        cout << "Speaking..." << endl;
    }
};

int main() {
    Parrot p;
    p.eat();    // from Animal
    p.fly();    // from Bird
    p.speak();  // from Parrot
    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

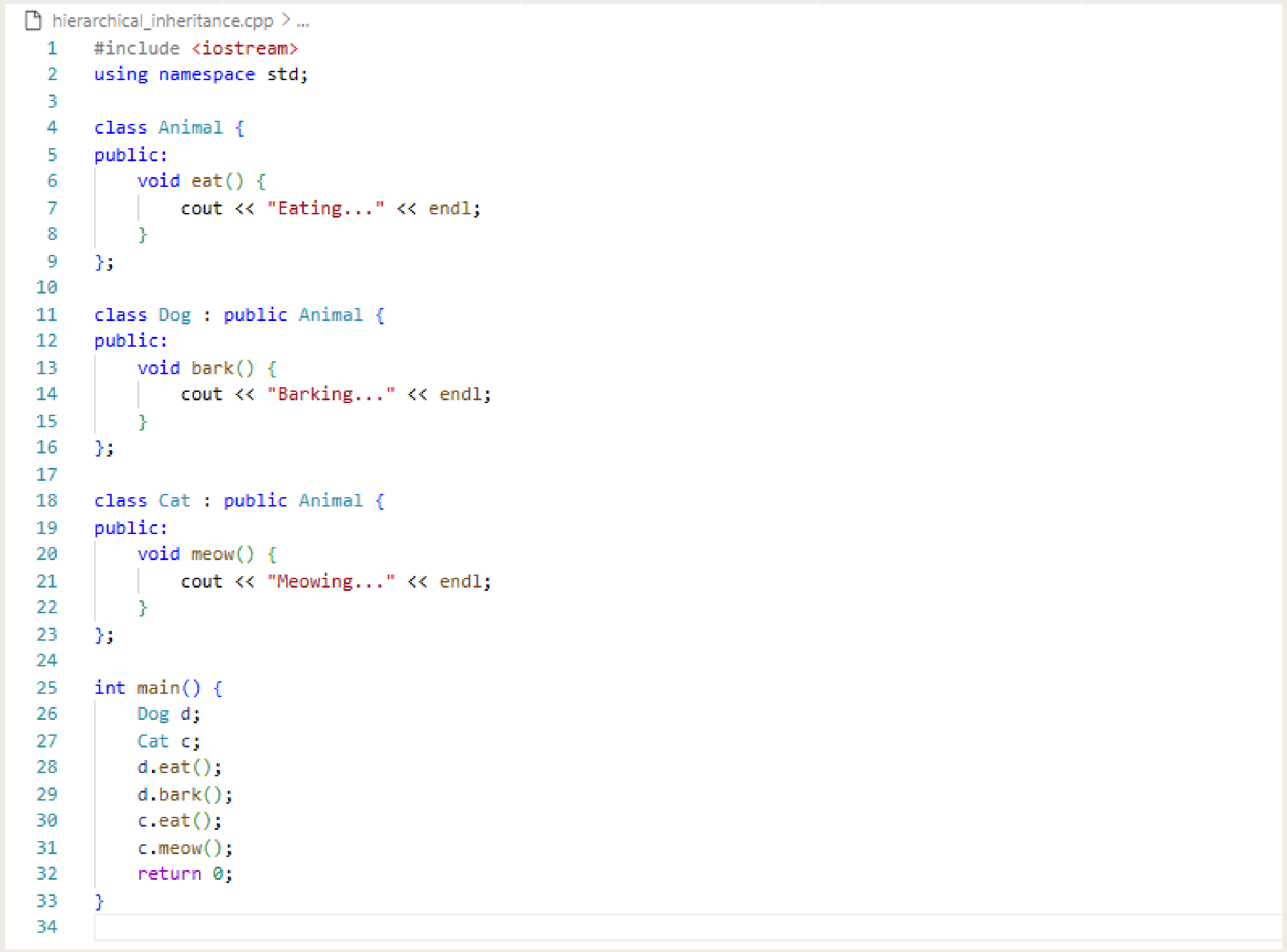
Hierarchical Inheritance

- **Definition:**
  Hierarchical Inheritance is a type of inheritance where **multiple derived classes inherit from a single base class.**
- In this type:
  - One base class has **common features.**
  - Multiple child classes **inherit** these features and can have their **own additional features.**
- It is useful when several classes share common behavior but also need their own special behavior.



Prepared by Ayush Lamsal | |
layush788@gmail.com

Hierarchical Inheritance



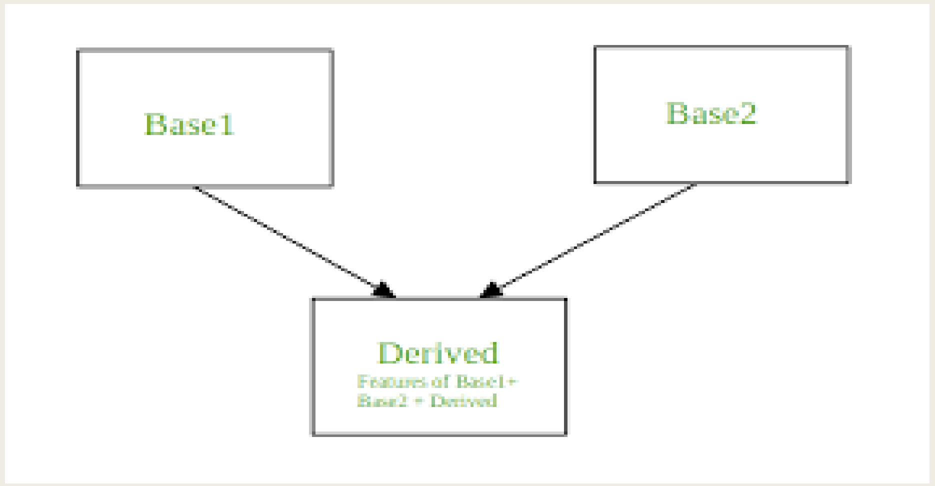
hierarchical_inheritance.cpp > ...

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

class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};

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

class Cat : public Animal {
public:
    void meow() {
        cout << "Meowing..." << endl;
    }
};

int main() {
    Dog d;
    Cat c;
    d.eat();
    d.bark();
    c.eat();
    c.meow();
    return 0;
}
```

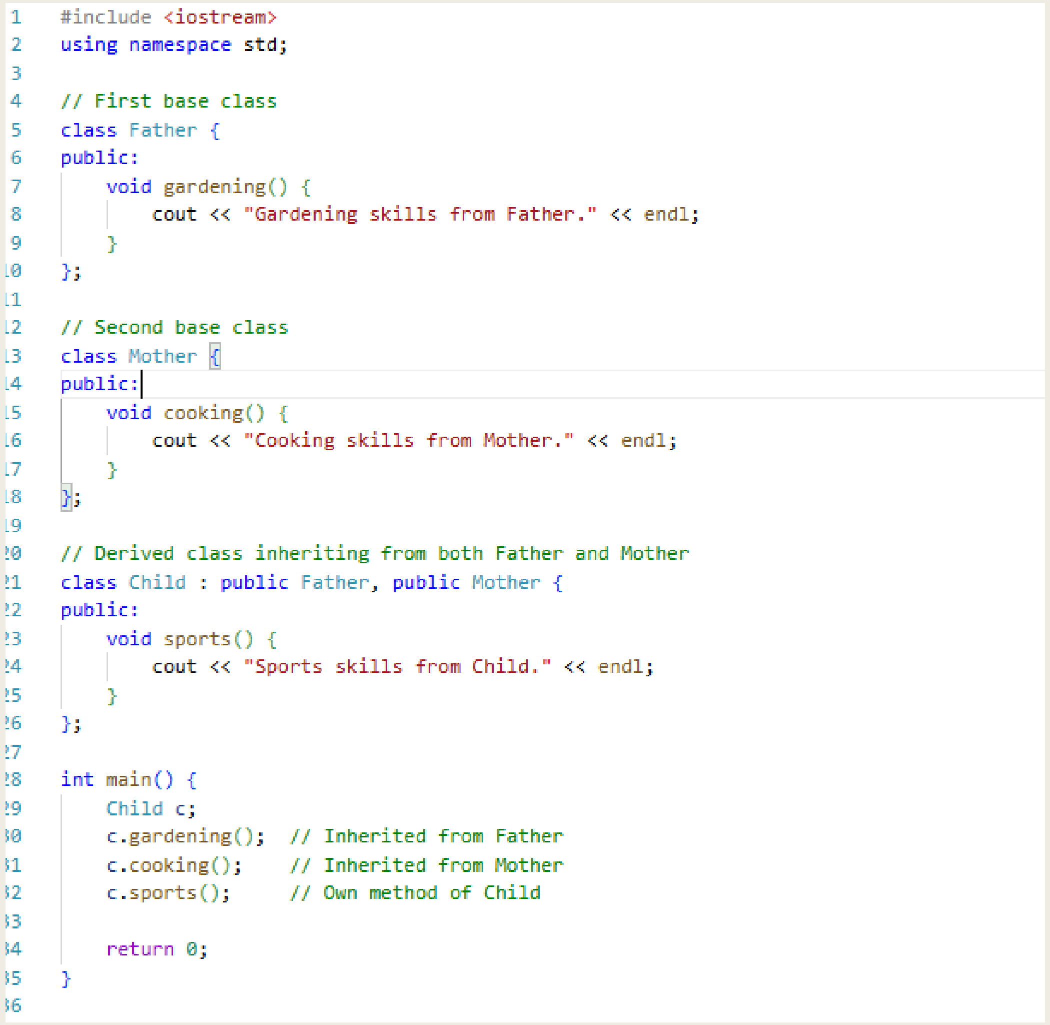
Prepared by Ayush Lamsal | |
layush788@gmail.com

Multiple Inheritance

- **Multiple Inheritance** is a feature of object-oriented programming where a **derived class inherits from more than one base class.**
- This means the derived class can use the properties and methods of **all its base classes.**



Example


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

// First base class
class Father {
public:
    void gardening() {
        cout << "Gardening skills from Father." << endl;
    }
};

// Second base class
class Mother {
public:
    void cooking() {
        cout << "Cooking skills from Mother." << endl;
    }
};

// Derived class inheriting from both Father and Mother
class Child : public Father, public Mother {
public:
    void sports() {
        cout << "Sports skills from Child." << endl;
    }
};

int main() {
    Child c;
    c.gardening();  // Inherited from Father
    c.cooking();    // Inherited from Mother
    c.sports();     // Own method of Child

    return 0;
}
```

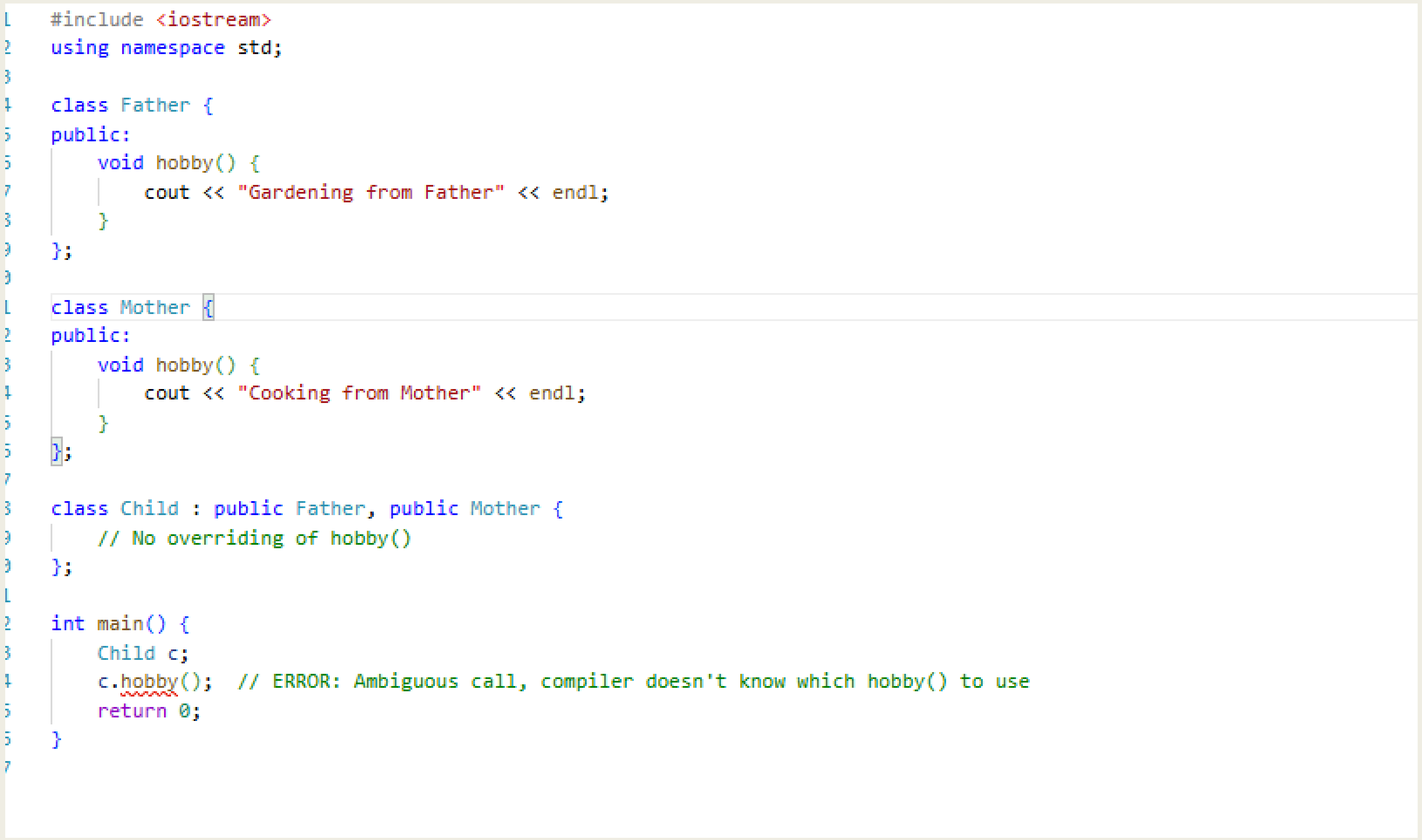
Ambiguity in multiple inheritance

- **Ambiguity** occurs in multiple inheritance when **two or more base classes have methods or members with the same name**, and the derived class tries to access that member without specifying which base class it belongs to.

What happens?

- The compiler **gets confused** because it doesn't know which base class's method or member to use, and this results in an **error.**

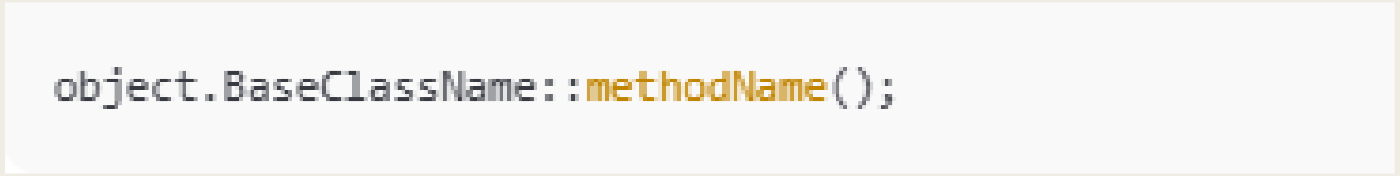
Ambiguity in multiple inheritance Example



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

class Father {
public:
    void hobby() {
        cout << "Gardening from Father" << endl;
    }
};

class Mother {
public:
    void hobby() {
        cout << "Cooking from Mother" << endl;
    }
};

class Child : public Father, public Mother {
    // No overriding of hobby()
};

int main() {
    Child c;
    c.hobby();  // ERROR: Ambiguous call, compiler doesn't know which hobby() to use
    return 0;
}
```

How ambiguity can be resolved?

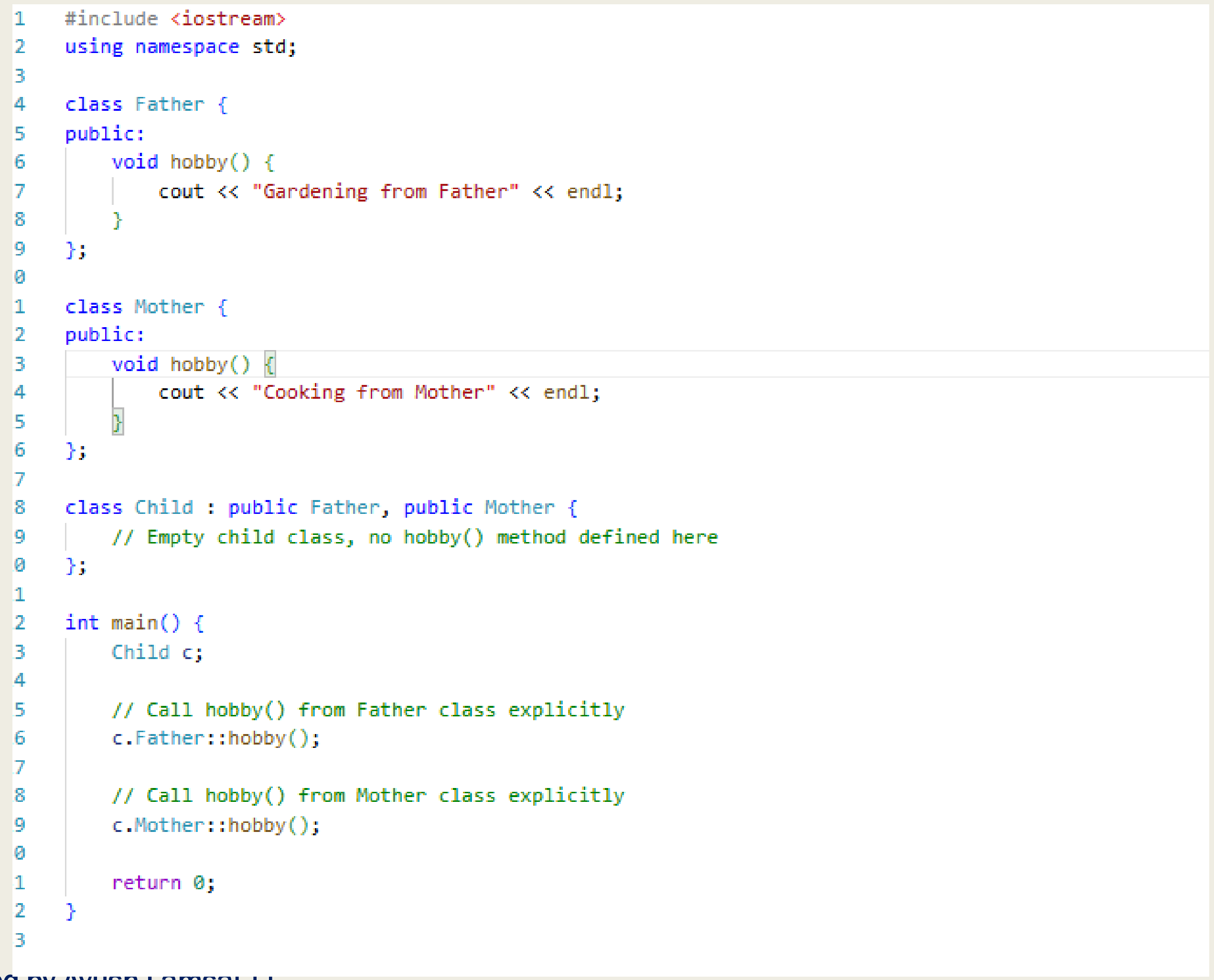
- Ambiguity in multiple inheritance can be resolved by **explicitly specifying which base class's method or member you want to use** with the **scope resolution operator (::)**.
- Syntax:



```
object.BaseClassName::methodName();
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Fixing ambiguity example of multiple inheritance



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

class Father {
public:
    void hobby() {
        cout << "Gardening from Father" << endl;
    }
};

class Mother {
public:
    void hobby() {
        cout << "Cooking from Mother" << endl;
    }
};

class Child : public Father, public Mother {
    // Empty child class, no hobby() method defined here
};

int main() {
    Child c;

    // Call hobby() from Father class explicitly
    c.Father::hobby();

    // Call hobby() from Mother class explicitly
    c.Mother::hobby();

    return 0;
}
```

Hybrid Inheritance

- **Hybrid Inheritance in C++** is a combination of **two or more types of inheritance** (like single, multiple, multilevel, or hierarchical) in one program. It is used when a class is derived from multiple classes in a way that combines different inheritance types.

Example Scenario:

- Suppose we have:
  - *A base class A*
  - *Two classes B and C derived from A (hierarchical inheritance)*
  - *A class D derived from both B and C (multiple inheritance)*
- This is Hybrid Inheritance because it combines hierarchical and multiple inheritance.

Hybrid Inheritance

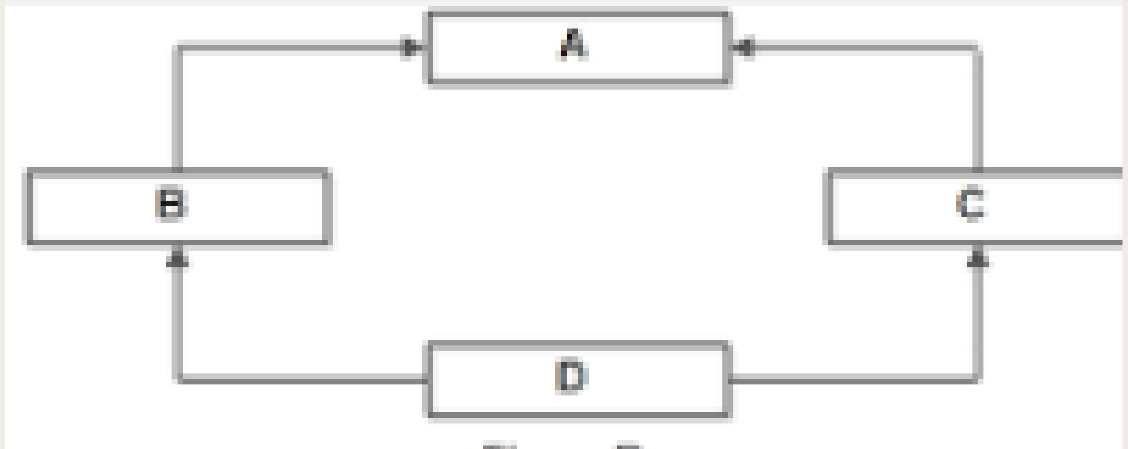
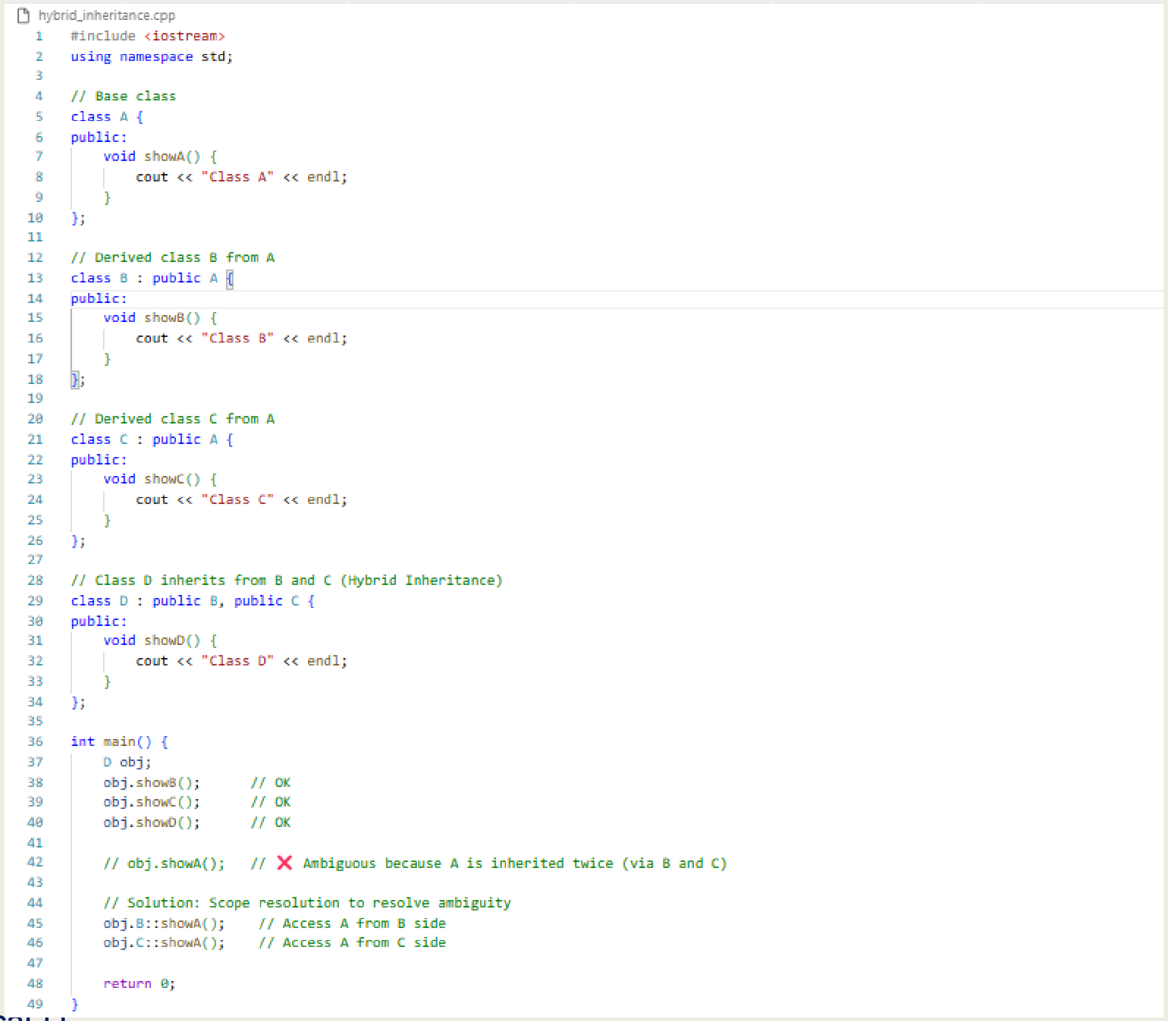


Figure B

Prepared by Ayush Lamsal | |
layush788@gmail.com

Hybrid Inheritance

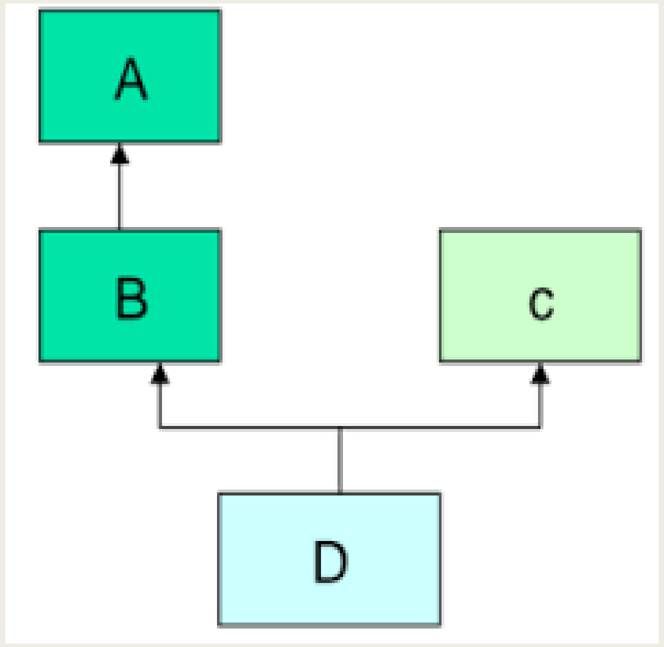

hybrid_inheritance.cpp

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

// Base class
class A {
public:
    void showA() {
        cout << "Class A" << endl;
    }
};

// Derived class B from A
class B : public A {
public:
    void showB() {
        cout << "Class B" << endl;
    }
};

// Derived class C from A
class C : public A {
public:
    void showC() {
        cout << "Class C" << endl;
    }
};

// Class D inherits from B and C (Hybrid Inheritance)
class D : public B, public C {
public:
    void showD() {
        cout << "Class D" << endl;
    }
};

int main() {
    D obj;
    obj.showB();      // OK
    obj.showC();      // OK
    obj.showD();      // OK

    // obj.showA();   // ❌ Ambiguous because A is inherited twice (via B and C)

    // Solution: Scope resolution to resolve ambiguity
    obj.B::showA();    // Access A from B side
    obj.C::showA();    // Access A from C side

    return 0;
}
```

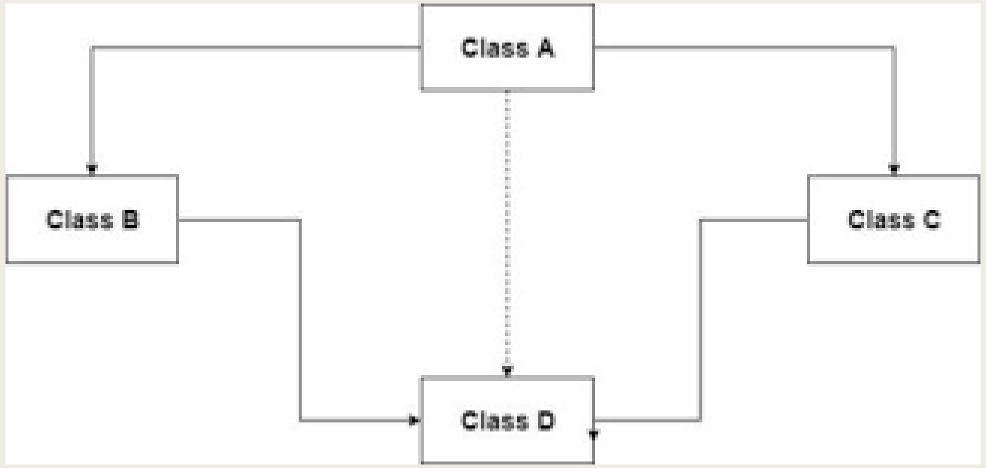
Prepared by Ayush Lamsal ||
layush788@gmail.com

Hybrid Inheritance

- Try this example



Try this

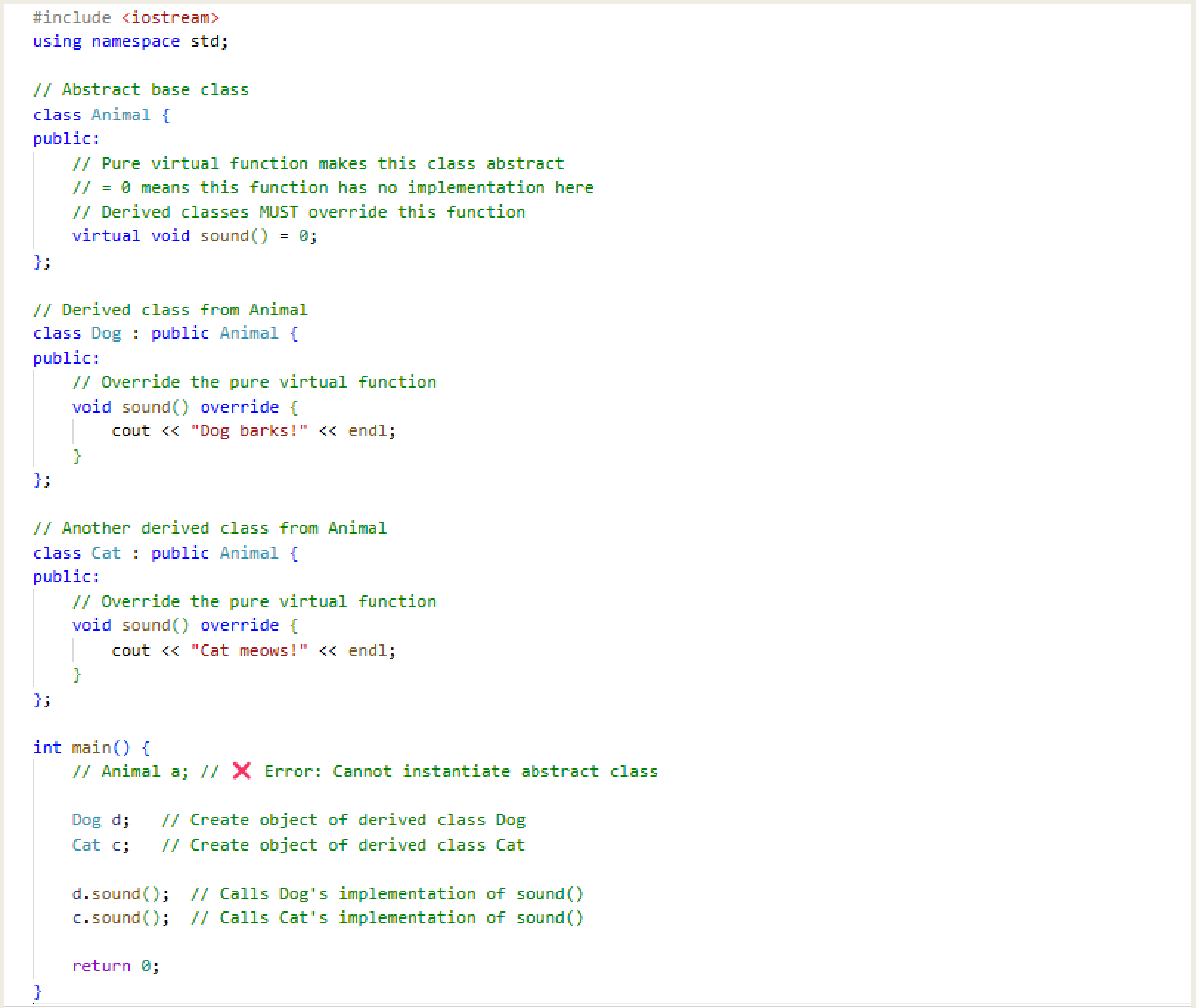


Prepared by Ayush Lamsal | |
layush788@gmail.com

Abstract class

- An **abstract class** is one that **cannot be used to create objects directly**. It is designed **only to act as a base class** to be inherited by other classes.
- An abstract class is a **design concept** in program development and provides a **common foundation** upon which other classes may be built. It often contains **pure virtual functions** that **must be implemented** by its derived classes.

Abstract class example



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

// Abstract base class
class Animal {
public:
    // Pure virtual function makes this class abstract
    // = 0 means this function has no implementation here
    // Derived classes MUST override this function
    virtual void sound() = 0;
};

// Derived class from Animal
class Dog : public Animal {
public:
    // Override the pure virtual function
    void sound() override {
        cout << "Dog barks!" << endl;
    }
};

// Another derived class from Animal
class Cat : public Animal {
public:
    // Override the pure virtual function
    void sound() override {
        cout << "Cat meows!" << endl;
    }
};

int main() {
    // Animal a; // ❌ Error: Cannot instantiate abstract class

    Dog d;   // Create object of derived class Dog
    Cat c;   // Create object of derived class Cat

    d.sound();  // Calls Dog's implementation of sound()
    c.sound();  // Calls Cat's implementation of sound()

    return 0;
}
```

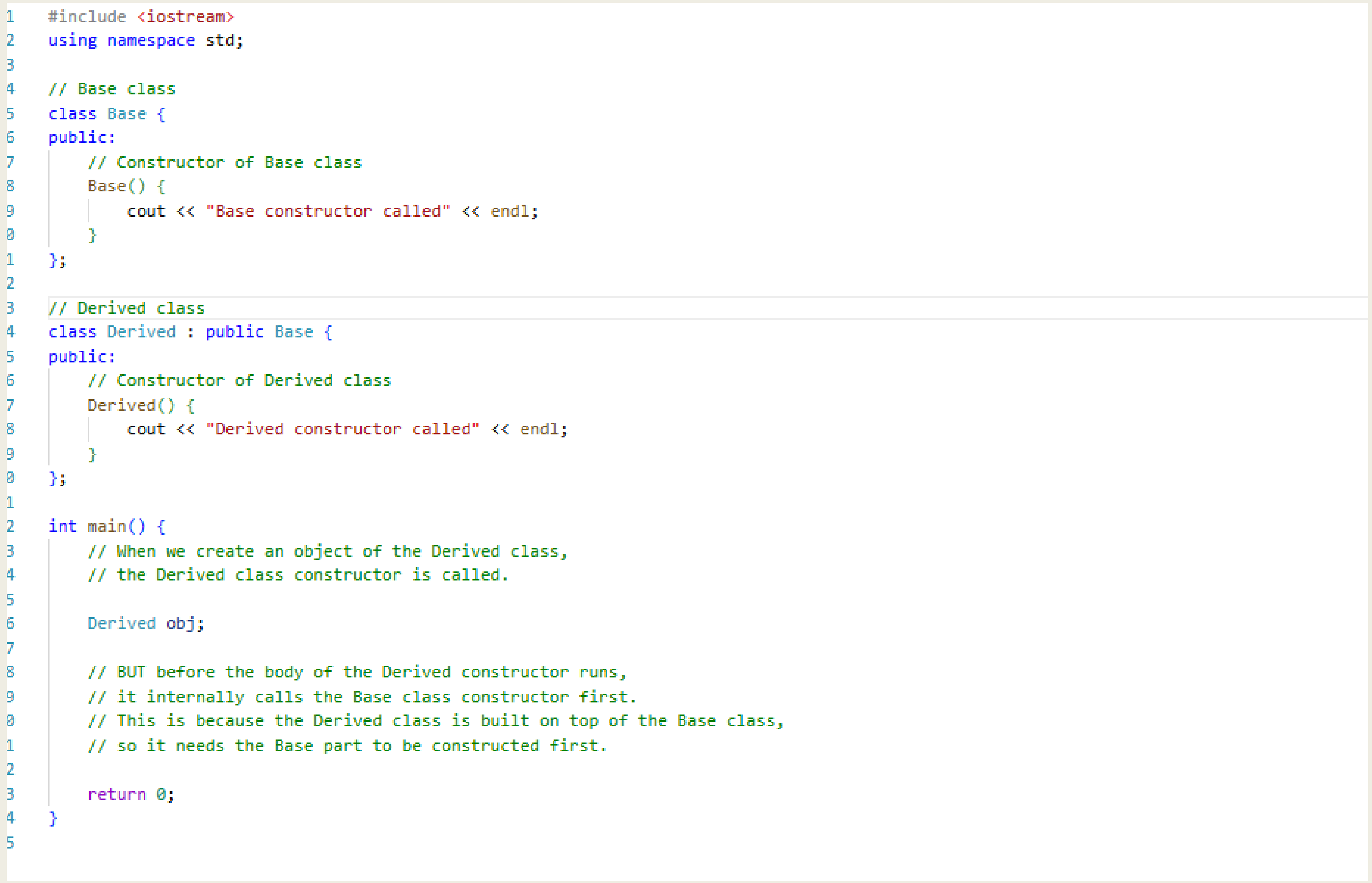
Prepared by Ayush Lamsal | |
layush788@gmail.com

Constructor in Derived class

- A **constructor** in a **derived class** is a special member function that initializes objects of the derived class. It is responsible for initializing both the members inherited from the base class and those declared in the derived class itself.
- When an object of a derived class is created:
  - *The **base class constructor is called first** to initialize the base part of the object.*
  - *Then, the **derived class constructor is called** to initialize the members specific to the derived class.*
- If the base class has a **parameterized constructor**, the derived class constructor must explicitly call it using an **initializer list**; otherwise, the base class's **default constructor** is called automatically.
- This ensures proper and complete initialization of the entire object, starting from the base class and moving down to the derived class.

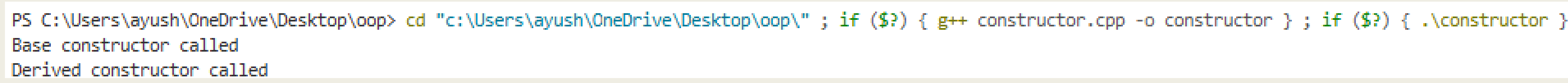
Note: If asked for more mark you can also explain what exactly constructor is

Example of constructor inheritance



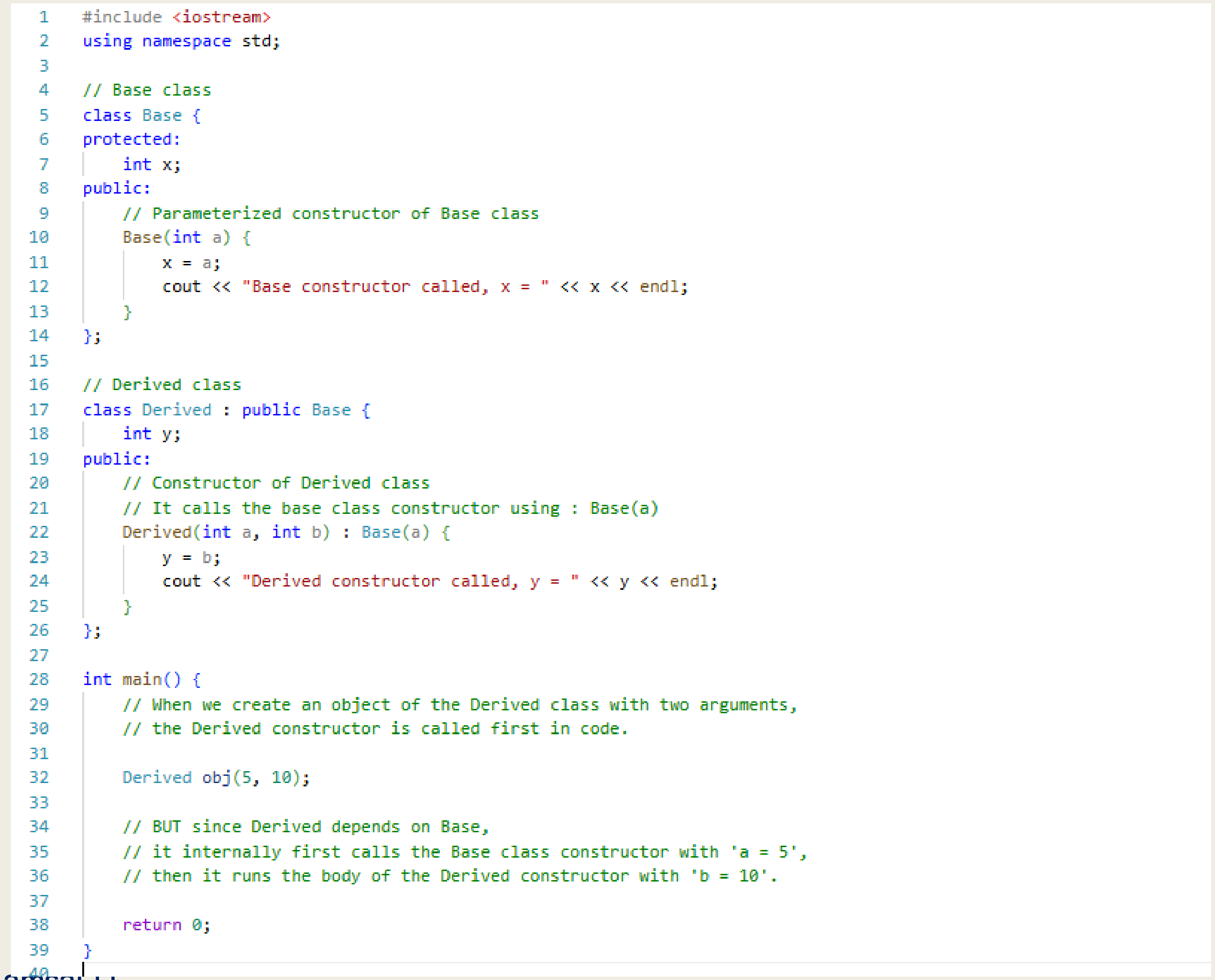
```cpp
#include <iostream>
using namespace std;

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

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

int main() {
    // When we create an object of the Derived class,
    // the Derived class constructor is called.

    Derived obj;

    // BUT before the body of the Derived constructor runs,
    // it internally calls the Base class constructor first.
    // This is because the Derived class is built on top of the Base class,
    // so it needs the Base part to be constructed first.

    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com



```
PS C:\Users\ayush\OneDrive\Desktop\oop> cd "c:\Users\ayush\OneDrive\Desktop\oop\" ; if ($?) { g++ constructor.cpp -o constructor } ; if ($?) { .\constructor }
Base constructor called
Derived constructor called
```

Parameterized Constructor in Derived class



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

// Base class
class Base {
protected:
    int x;
public:
    // Parameterized constructor of Base class
    Base(int a) {
        x = a;
        cout << "Base constructor called, x = " << x << endl;
    }
};

// Derived class
class Derived : public Base {
    int y;
public:
    // Constructor of Derived class
    // It calls the base class constructor using : Base(a)
    Derived(int a, int b) : Base(a) {
        y = b;
        cout << "Derived constructor called, y = " << y << endl;
    }
};

int main() {
    // When we create an object of the Derived class with two arguments,
    // the Derived constructor is called first in code.

    Derived obj(5, 10);

    // BUT since Derived depends on Base,
    // it internally first calls the Base class constructor with 'a = 5',
    // then it runs the body of the Derived constructor with 'b = 10'.

    return 0;
}
```

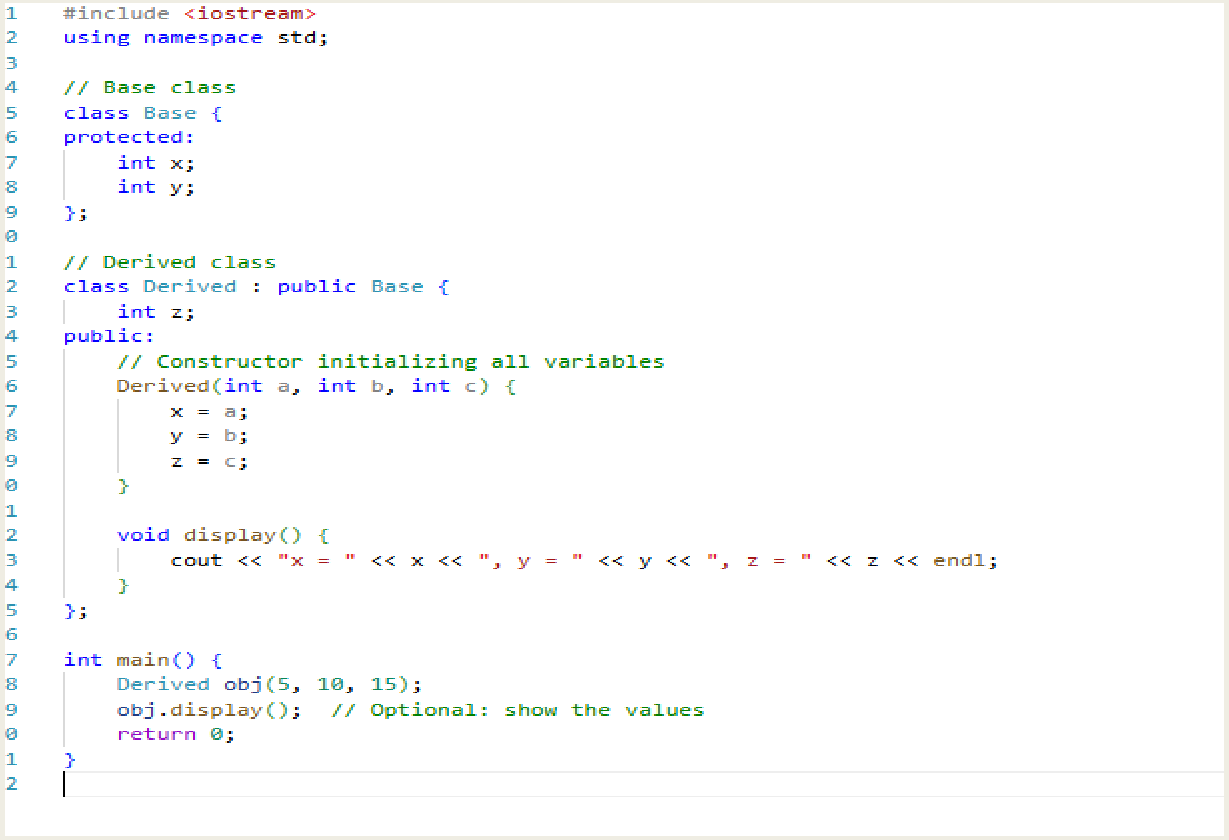
Prepared by Ayush Lamsal | |
layush788@gmail.com

Try yourself

- In derived class use both value that is passed as a argument and also pass one value to the base class

Important question

Write a simple program to initialize member variable of both base and derived class using the constructor of derived class



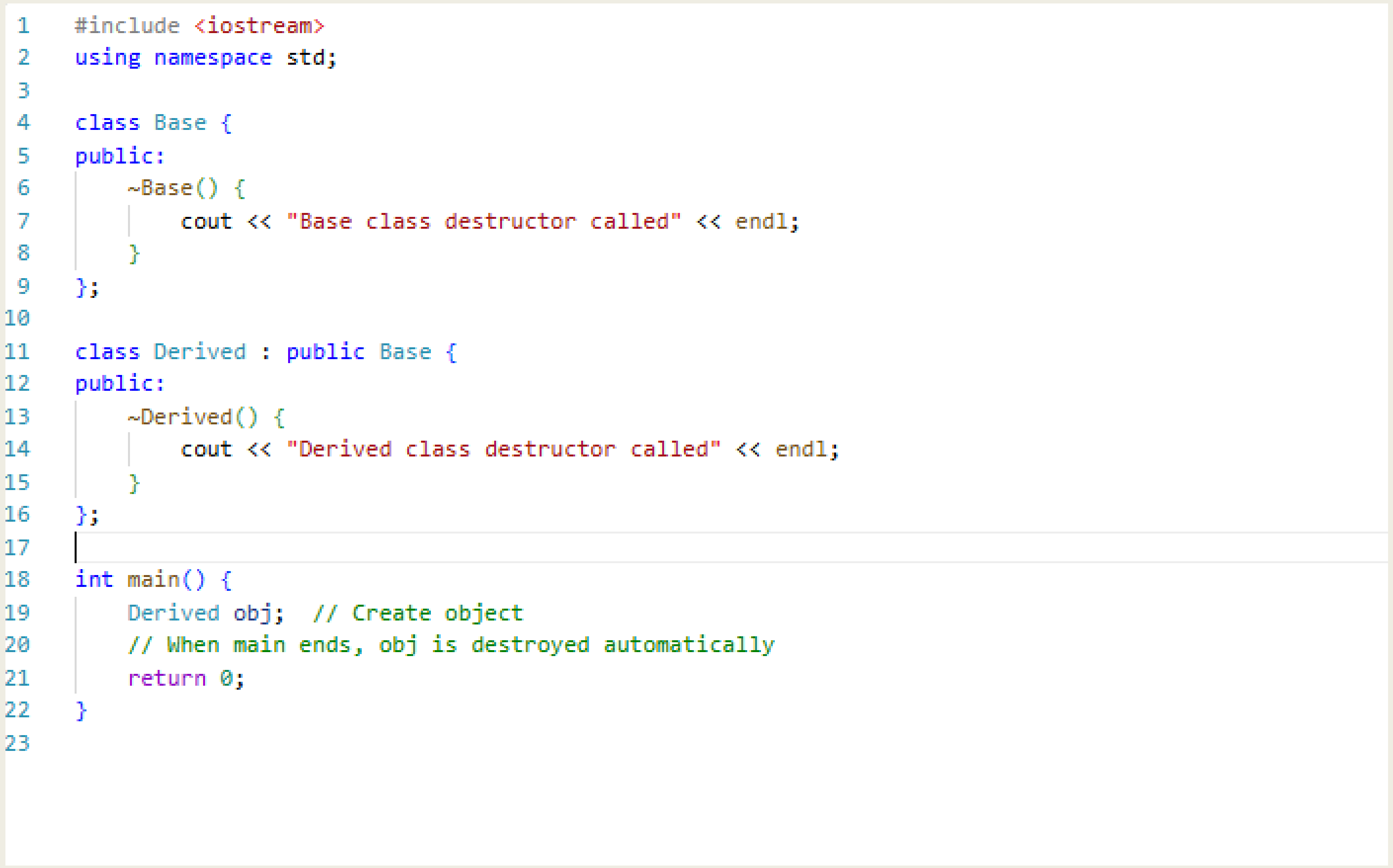
```cpp
#include <iostream>
using namespace std;

// Base class
class Base {
protected:
    int x;
    int y;
};

// Derived class
class Derived : public Base {
    int z;
public:
    // Constructor initializing all variables
    Derived(int a, int b, int c) {
        x = a;
        y = b;
        z = c;
    }

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

int main() {
    Derived obj(5, 10, 15);
    obj.display();  // Optional: show the values
    return 0;
}

```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Destructors in derived classes

- The execution of destructors occurs in the **opposite order** compared to constructors. While constructors are executed from the **base class to the derived class**, destructors are executed from the **derived class back to the base class.**
- This means that the object created last is destroyed first, following the **Last In, First Out (LIFO)** principle.

Destructor in derived classes



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

class Base {
public:
    ~Base() {
        cout << "Base class destructor called" << endl;
    }
};

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

int main() {
    Derived obj;  // Create object
    // When main ends, obj is destroyed automatically
    return 0;
}
```

Prepared by Ayush Lamsal | |
layush788@gmail.com

Explanation

Explanation:

- When obj goes out of scope (at the end of main()), destructors are called.
  - ***First the derived class destructor runs**, cleaning up derived class stuff.*
  - ***Then the base class destructor runs,** cleaning up base class stuff.*
- This ensures proper destruction from the most specific (derived) to the most general (base).

Important question

Write a program to create a derived class form base classes use both concept of constructor and destructors together in a single program

Prepared by Ayush Lamsal | |
layush788@gmail.com

Aggregation in c++

Aggregation is a concept in object-oriented programming where one class **uses** another class, but the used class can **exist independently.**

Aggregation means **"has-a" relationship**, where one class **has a pointer or reference** to another class.

In aggregation, one class uses another class, but the **contained object is not owned**, and **both can work independently.**

Example:

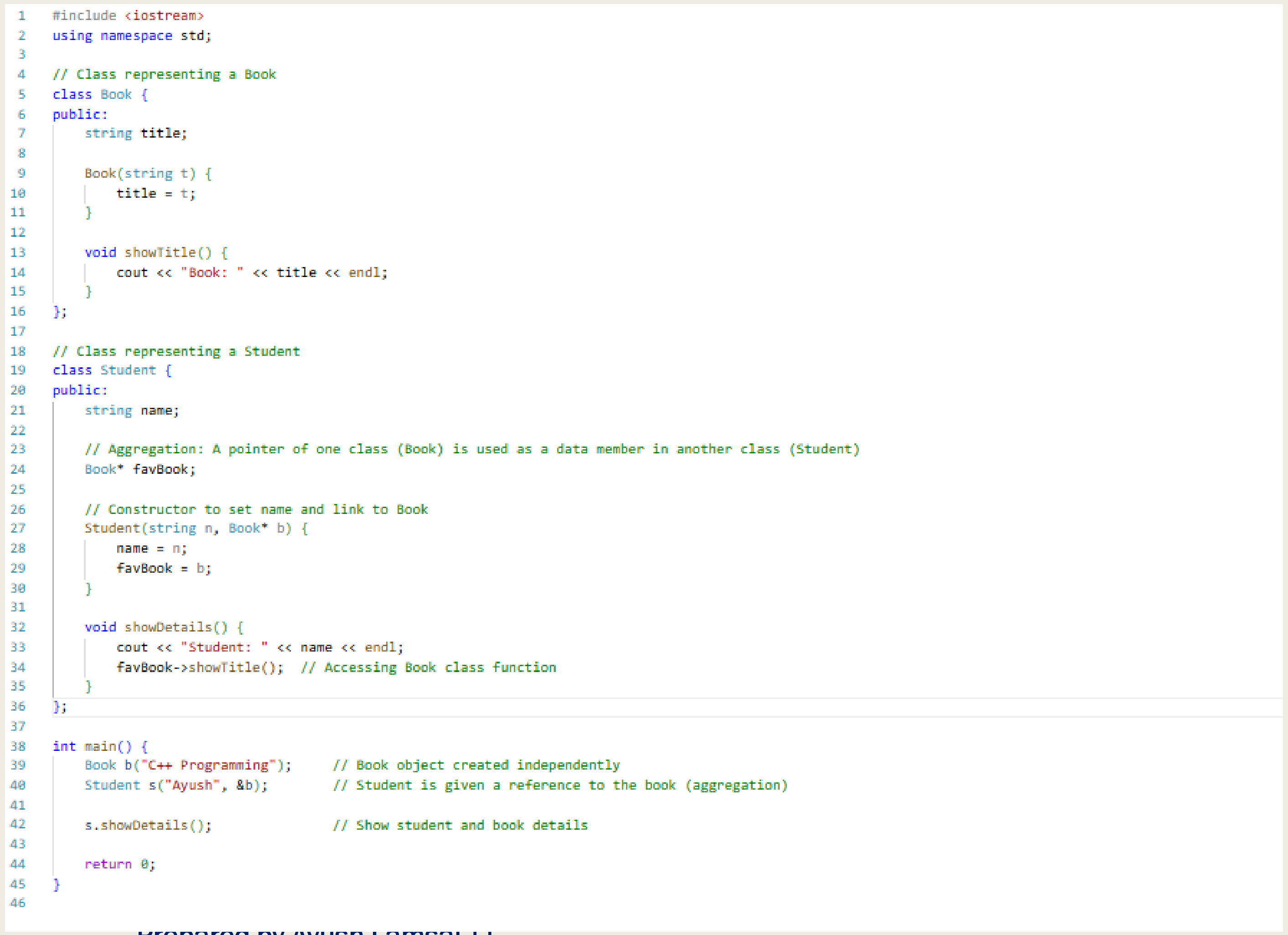
- *A Student has a Book, but the Book can exist without the Student.*

- In aggregation, one class contains a pointer or reference to an object of another class.
- This lets the first class use the object of the other class inside it.
- So, you can call the methods of that contained object whenever you want.

Example Summary:

- If ClassA has a pointer to an object of ClassB, then inside ClassA you can use that pointer to call ClassB's methods.

Example of aggregation in c++


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

// Class representing a Book
class Book {
public:
    string title;

    Book(string t) {
        title = t;
    }

    void showTitle() {
        cout << "Book: " << title << endl;
    }
};

// Class representing a Student
class Student {
public:
    string name;

    // Aggregation: A pointer of one class (Book) is used as a data member in another class (Student)
    Book* favBook;

    // Constructor to set name and link to Book
    Student(string n, Book* b) {
        name = n;
        favBook = b;
    }

    void showDetails() {
        cout << "Student: " << name << endl;
        favBook->showTitle();  // Accessing Book class function
    }
};

int main() {
    Book b("C++ Programming");     // Book object created independently
    Student s("Ayush", &b);        // Student is given a reference to the book (aggregation)

    s.showDetails();               // Show student and book details

    return 0;
}
```

ASSIGNEMENT

- Public and private inheritance