INHERITANCE

Lecturer： 陈笑沙

REVIEW

Which of the following descriptions is incorrect?

- A. Non-static methods can call static methods and attributes
- B. Static methods can call other static methods and attributes
- C. Static methods can call non-static methods and attributes
- D. Non-static methods can call other non-static methods and attributes

TABLE OF CONTENTS

8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- Classes describe the commonality of groups, creating different objects from classes realizes code reuse. However, this reuse is insufficient.

8.1 CONCEPT OF INHERITANCE

- Classes describe the commonality of groups, creating different objects from classes realizes code reuse. However, this reuse is insufficient.
- For example, the Car class has functions such as starting, braking, shifting gears, and changing direction. When the Car class needs to be extended to the Transformer class, adding the "transform" function, how to do it?

8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- Method one:

8.1 CONCEPT OF INHERITANCE

- Method one:
  - Create a new class Transformer, paste the Car class code inside it.

8.1 CONCEPT OF INHERITANCE

- Method one:
  - Create a new class Transformer, paste the Car class code inside it.
  - Add a new method void transform(){...}, and if the Car class adds a new feature: automatic parking void stop(), then we need to modify both the Car class and the Transformer class.

8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- Method two:

8.1 CONCEPT OF INHERITANCE

- Method two:
  - The Transofomer class "uses" the features of the Car class, so it doesn't need to copy and paste, and the Transformer class automatically has all the features of the Car class.

8.1 CONCEPT OF INHERITANCE

- Method two:
  - The Transofomer class "uses" the features of the Car class, so it doesn't need to copy and paste, and the Transformer class automatically has all the features of the Car class.
  - The better part is that whenever the Car class is modified, the Transformer class can apply this modification. This is called **inheritance and derivation**.

8.1 CONCEPT OF INHERITANCE

INHERITANCE IN REALITY

8.1 CONCEPT OF INHERITANCE

INHERITANCE IN REALITY

- Cats and dogs are both mammals, possessing characteristics such as viviparity, lactation, and homeothermy, but also have their own unique traits.

8.1 CONCEPT OF INHERITANCE

INHERITANCE IN REALITY

- Cats and dogs are both mammals, possessing characteristics such as viviparity, lactation, and homeothermy, but also have their own unique traits.
- This is an important property of "inheritance" relationships.

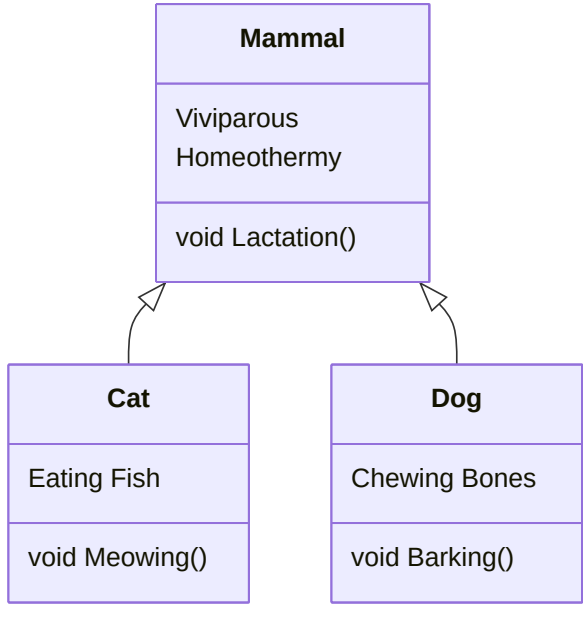
8.1 CONCEPT OF INHERITANCE

INHERITANCE IN REALITY

- Cats and dogs are both mammals, possessing characteristics such as viviparity, lactation, and homeothermy, but also have their own unique traits.
- This is an important property of "inheritance" relationships.
- The two classes forming an inheritance relationship have an IS_A relationship.

8.1 CONCEPT OF INHERITANCE

INHERITANCE IN REALITY



8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- Things are related to each other through classification and stratification

8.1 CONCEPT OF INHERITANCE

- Things are related to each other through classification and stratification
- Upper-level things have the commonality of lower-level things

8.1 CONCEPT OF INHERITANCE

- Things are related to each other through classification and stratification
- Upper-level things have the commonality of lower-level things
- For example

8.1 CONCEPT OF INHERITANCE

- Things are related to each other through classification and stratification
- Upper-level things have the commonality of lower-level things
- For example
  - Sichuanese and Zhejiangese are classified as Chinese people

8.1 CONCEPT OF INHERITANCE

- Things are related to each other through classification and stratification
- Upper-level things have the commonality of lower-level things
- For example
  - Sichuanese and Zhejiangese are classified as Chinese people
  - They have the common characteristics of yellow skin, black hair, and speaking Chinese

8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- Programming needs to handle layered things (data)

8.1 CONCEPT OF INHERITANCE

- Programming needs to handle layered things (data)
- With upper-level data or objects, use inheritance to describe lower-level data and objects

8.1 CONCEPT OF INHERITANCE

- Programming needs to handle layered things (data)
- With upper-level data or objects, use inheritance to describe lower-level data and objects
- Both shared the upper-level data and code to get the benefits of code reuse, while maintaining the encapsulation of the class structure

8.1 CONCEPT OF INHERITANCE

- Programming needs to handle layered things (data)
- With upper-level data or objects, use inheritance to describe lower-level data and objects
- Both shared the upper-level data and code to get the benefits of code reuse, while maintaining the encapsulation of the class structure
- So that the inherited entity can also be inherited as upper-level data and code

8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- **Inheritance**

8.1 CONCEPT OF INHERITANCE

- **Inheritance**
  - Once the essential characteristics of a parent are specified, its children will automatically possess those qualities.

8.1 CONCEPT OF INHERITANCE

- **Inheritance**
  - Once the essential characteristics of a parent are specified, its children will automatically possess those qualities.
  - This is a naive and reusable concept.

8.1 CONCEPT OF INHERITANCE

- **Inheritance**
  - Once the essential characteristics of a parent are specified, its children will automatically possess those qualities.
  - This is a naive and reusable concept.
  - Inheritance involves creating a new class based on an existing one.

8.1 CONCEPT OF INHERITANCE

- **Inheritance**
  - Once the essential characteristics of a parent are specified, its children will automatically possess those qualities.
  - This is a naive and reusable concept.
  - Inheritance involves creating a new class based on an existing one.
- **Derivation**

8.1 CONCEPT OF INHERITANCE

- **Inheritance**
  - Once the essential characteristics of a parent are specified, its children will automatically possess those qualities.
  - This is a naive and reusable concept.
  - Inheritance involves creating a new class based on an existing one.
- **Derivation**
  - Children can have characteristics that their parents don't have, which is an extensible concept.

8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- The main features of the derived class are demonstrated through the following means

8.1 CONCEPT OF INHERITANCE

- The main features of the derived class are demonstrated through the following means
  - Absorbing base class members

8.1 CONCEPT OF INHERITANCE

- The main features of the derived class are demonstrated through the following means
  - Absorbing base class members
  - Adding new members

8.1 CONCEPT OF INHERITANCE

- The main features of the derived class are demonstrated through the following means
  - Absorbing base class members
  - Adding new members
  - Modifying base class members

8.1 CONCEPT OF INHERITANCE

8.1 CONCEPT OF INHERITANCE

- From a coding perspective, derived classes gain significant flexibility from base classes at a relatively low cost

8.1 CONCEPT OF INHERITANCE

- From a coding perspective, derived classes gain significant flexibility from base classes at a relatively low cost
  - Derived classes can extend, restrict, or modify inherited properties

8.1 CONCEPT OF INHERITANCE

- From a coding perspective, derived classes gain significant flexibility from base classes at a relatively low cost
  - Derived classes can extend, restrict, or modify inherited properties
  - Once a reliable base class is established, only the modifications made in the derived class need to be debugged

8.2 HOW INHERITANCE WORKS

8.2 HOW INHERITANCE WORKS

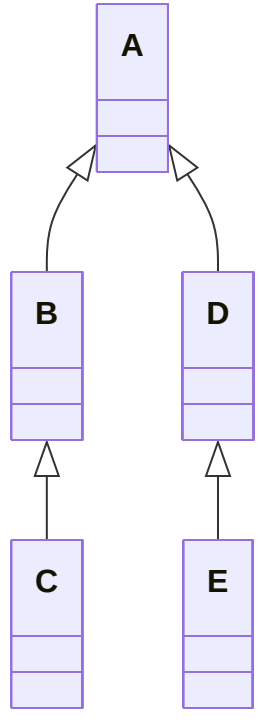
- Single inheritance: Derived class has only one direct base class

8.2 HOW INHERITANCE WORKS

- Single inheritance: Derived class has only one direct base class
- Multiple inheritance: Derived class has multiple direct base classes

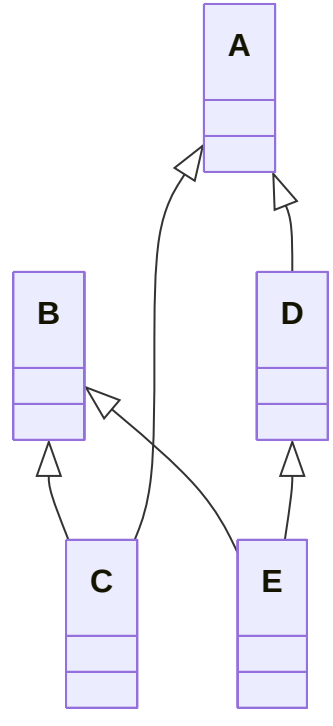
8.2 HOW INHERITANCE WORKS

SINGLE INHERITANCE



8.2 HOW INHERITANCE WORKS

MULTIPLE INHERITANCE



8.2 HOW INHERITANCE WORKS

Define the syntax format for single inheritance derived class

```
1  class DerivedClassName : <InheritanceMode> BaseClassName{
2      ...//Derived class modifies base class members
3  };
```

8.2 HOW INHERITANCE WORKS

- Although all members of the base class are inherited, the derived class does not necessarily have access to all members of the base class. The inheritance mode affects the derived class's access to various members of the base class.

8.2 HOW INHERITANCE WORKS

Inheritance Mode	Public Member	Protected Member	Private Member
Public Inheritance	Public Member	Protected Member	Inaccessible Member
Private Inheritance	Private Member	Private Member	Inaccessible Member
Protected Inheritance	Protected Member	Protected Member	Inaccessible Member

8.2 HOW INHERITANCE WORKS

```
1  class Base {
2  public:
3      int publicVar = 1;
4      void publicFunc() {
5          cout << "Base::publicFunc()" << endl;
6      }
7
8  protected:
9      int protectedVar = 2;
10     void protectedFunc() {
11         cout << "Base::protectedFunc()" << endl;
12     }
13
14 private:
15     int privateVar = 3;
```

8.2 HOW INHERITANCE WORKS

```
1  class Base {
2  public:
3      int publicVar = 1;
4      void publicFunc() {
5          cout << "Base::publicFunc()" << endl;
6      }
7
8  protected:
9      int protectedVar = 2;
10     void protectedFunc() {
11         cout << "Base::protectedFunc()" << endl;
12     }
13
14 private:
15     int privateVar = 3;
```

8.2 HOW INHERITANCE WORKS

```
3       int publicVar = 1;
4       void publicFunc() {
5           cout << "Base::publicFunc()" << endl;
6       }
7
8   protected:
9       int protectedVar = 2;
10      void protectedFunc() {
11          cout << "Base::protectedFunc()" << endl;
12      }
13
14  private:
15      int privateVar = 3;
16      void privateFunc() {
17          cout << "Base::privateFunc()" << endl;
```

8.2 HOW INHERITANCE WORKS

```
5         cout << "Base::publicFunc()" << endl;
6     }
7
8 protected:
9     int protectedVar = 2;
10    void protectedFunc() {
11        cout << "Base::protectedFunc()" << endl;
12    }
13
14 private:
15    int privateVar = 3;
16    void privateFunc() {
17        cout << "Base::privateFunc()" << endl;
18    }
19 };
```

8.2 HOW INHERITANCE WORKS

```
1  class Base {
2  public:
3      int publicVar = 1;
4      void publicFunc() {
5          cout << "Base::publicFunc()" << endl;
6      }
7
8  protected:
9      int protectedVar = 2;
10     void protectedFunc() {
11         cout << "Base::protectedFunc()" << endl;
12     }
13
14 private:
15     int privateVar = 3;
```

8.2 HOW INHERITANCE WORKS

```
class PublicDerived : public Base {
public:
    void accessBaseMembers() {
        cout << "PublicDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→public)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PublicDerived : public Base {
public:
    void accessBaseMembers() {
        cout << "PublicDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→public)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PublicDerived : public Base {
public:
    void accessBaseMembers() {
        cout << "PublicDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→public)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
1  class PublicDerived : public Base {
2  public:
3      void accessBaseMembers() {
4          cout << "PublicDerived access base class members:" << endl;
5          publicFunc();          // ✔️ Accessible (public→public)
6          protectedFunc();       // ✔️ Accessible (protected→protected)
7          // privateFunc();  // ❌ Compilation error: base class private member inaccessible
8      }
9  };
```

8.2 HOW INHERITANCE WORKS

```
1  class PublicDerived : public Base {
2  public:
3      void accessBaseMembers() {
4          cout << "PublicDerived access base class members:" << endl;
5          publicFunc();          // ✔️ Accessible (public→public)
6          protectedFunc();       // ✔️ Accessible (protected→protected)
7          // privateFunc();  // ❌ Compilation error: base class private member inaccessible
8      }
9  };
```

8.2 HOW INHERITANCE WORKS

```
class PublicDerived : public Base {
public:
    void accessBaseMembers() {
        cout << "PublicDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→public)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PublicDerived : public Base {
public:
    void accessBaseMembers() {
        cout << "PublicDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→public)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class ProtectedDerived : protected Base {
public:
    void accessBaseMembers() {
        cout << "ProtectedDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→protected)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();     // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
1  class ProtectedDerived : protected Base {
2  public:
3      void accessBaseMembers() {
4          cout << "ProtectedDerived access base class members:" << endl;
5          publicFunc();          // ✔️ Accessible (public→protected)
6          protectedFunc();       // ✔️ Accessible (protected→protected)
7          // privateFunc();      // ❌ Compilation error: base class private member inaccessible
8      }
9  };
```

8.2 HOW INHERITANCE WORKS

```
class ProtectedDerived : protected Base {
public:
    void accessBaseMembers() {
        cout << "ProtectedDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→protected)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();     // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class ProtectedDerived : protected Base {
public:
    void accessBaseMembers() {
        cout << "ProtectedDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→protected)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();     // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
1  class ProtectedDerived : protected Base {
2  public:
3      void accessBaseMembers() {
4          cout << "ProtectedDerived access base class members:" << endl;
5          publicFunc();          // ✔️ Accessible (public→protected)
6          protectedFunc();       // ✔️ Accessible (protected→protected)
7          // privateFunc();      // ❌ Compilation error: base class private member inaccessible
8      }
9  };
```

8.2 HOW INHERITANCE WORKS

```
class ProtectedDerived : protected Base {
public:
    void accessBaseMembers() {
        cout << "ProtectedDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→protected)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();     // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class ProtectedDerived : protected Base {
public:
    void accessBaseMembers() {
        cout << "ProtectedDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→protected)
        protectedFunc();      // ✔️ Accessible (protected→protected)
        // privateFunc();     // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PrivateDerived : private Base {
public:
    void accessBaseMembers() {
        cout << "PrivateDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→private)
        protectedFunc();      // ✔️ Accessible (protected→private)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PrivateDerived : private Base {
public:
    void accessBaseMembers() {
        cout << "PrivateDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→private)
        protectedFunc();      // ✔️ Accessible (protected→private)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PrivateDerived : private Base {
public:
    void accessBaseMembers() {
        cout << "PrivateDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→private)
        protectedFunc();      // ✔️ Accessible (protected→private)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PrivateDerived : private Base {
public:
    void accessBaseMembers() {
        cout << "PrivateDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→private)
        protectedFunc();      // ✔️ Accessible (protected→private)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PrivateDerived : private Base {
public:
    void accessBaseMembers() {
        cout << "PrivateDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→private)
        protectedFunc();      // ✔️ Accessible (protected→private)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PrivateDerived : private Base {
public:
    void accessBaseMembers() {
        cout << "PrivateDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→private)
        protectedFunc();      // ✔️ Accessible (protected→private)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
class PrivateDerived : private Base {
public:
    void accessBaseMembers() {
        cout << "PrivateDerived access base class members:" << endl;
        publicFunc();         // ✔️ Accessible (public→private)
        protectedFunc();      // ✔️ Accessible (protected→private)
        // privateFunc();  // ❌ Compilation error: base class private member inaccessible
    }
};
```

8.2 HOW INHERITANCE WORKS

```
1  int main() {
2      // Public inheritance object
3      PublicDerived pubObj;
4      // ✔️ External can access base class public members
5      pubObj.publicFunc();
6      // ❌ External cannot access base class protected members
7      // pubObj.protectedFunc();
8
9      // Protected inheritance object
10     // ❌ External cannot access
11     // base class public members become protected in derived class
12     ProtectedDerived proObj;
13     // proObj.publicFunc();
14
15     // Private inheritance object
```

8.2 HOW INHERITANCE WORKS

```
int main() {
    // Public inheritance object
    PublicDerived pubObj;
    // ✔️ External can access base class public members
    pubObj.publicFunc();
    // ❌ External cannot access base class protected members
    // pubObj.protectedFunc();

    // Protected inheritance object
    // ❌ External cannot access
    // base class public members become protected in derived class
    ProtectedDerived proObj;
    // proObj.publicFunc();

    // Private inheritance object
```

8.2 HOW INHERITANCE WORKS

```
4     // ✔️ External can access base class public members
5     pubObj.publicFunc();
6     // ❌ External cannot access base class protected members
7     // pubObj.protectedFunc();
8
9     // Protected inheritance object
10    // ❌ External cannot access
11    // base class public members become protected in derived class
12    ProtectedDerived proObj;
13    // proObj.publicFunc();
14
15    // Private inheritance object
16    PrivateDerived priObj;
17    // ❌ External cannot access
18    // base class public members become private in derived class
```

8.2 HOW INHERITANCE WORKS

```
 8
 9      // Protected inheritance object
10      // ❌ External cannot access
11      // base class public members become protected in derived class
12      ProtectedDerived proObj;
13      // proObj.publicFunc();
14
15      // Private inheritance object
16      PrivateDerived priObj;
17      // ❌ External cannot access
18      // base class public members become private in derived class
19      // priObj.publicFunc();
20
21      return 0;
22  }
```

8.3 INHERITANCE VS COMPOSITION

```
1  // Inheritance example (is-a relationship)
2  class Vehicle {  // Base class
3  public:
4      virtual void start() { cout << "Vehicle Starts" << endl; }
5  };
6
7  class Car : public Vehicle {  // Public inheritance
8  public:
9      void start() override {  // Override base class method
10         cout << "Car starting: Press the clutch pedal" << endl;
11     }
12     void drive() { cout << "Car in motion" << endl; }
13 };
14
15 // Composition example (has-a relationship)
```

8.3 INHERITANCE VS COMPOSITION

```cpp
// Inheritance example (is-a relationship)
class Vehicle {  // Base class
public:
    virtual void start() { cout << "Vehicle Starts" << endl; }
};

class Car : public Vehicle {  // Public inheritance
public:
    void start() override {  // Override base class method
        cout << "Car starting: Press the clutch pedal" << endl;
    }
    void drive() { cout << "Car in motion" << endl; }
};

// Composition example (has-a relationship)
```

8.3 INHERITANCE VS COMPOSITION

```
3  public:
4      virtual void start() { cout << "Vehicle Starts" << endl; }
5  };
6
7  class Car : public Vehicle {  // Public inheritance
8  public:
9      void start() override {  // Override base class method
10         cout << "Car starting: Press the clutch pedal" << endl;
11     }
12     void drive() { cout << "Car in motion" << endl; }
13 };
14
15 // Composition example (has-a relationship)
16 class Engine {  // Independent component class
17 public:
```

8.3 INHERITANCE VS COMPOSITION

```
10          cout << "Car starting: Press the clutch pedal" << endl;
11      }
12      void drive() { cout << "Car in motion" << endl; }
13  };
14
15  // Composition example (has-a relationship)
16  class Engine {  // Independent component class
17  public:
18      void startEngine() { cout << "Engine Starts" << endl; }
19  };
20
21  class ElectricCar {  // Composition class
22  private:
23      Engine engine;     // Embedded Engine object
24      Vehicle vehicle;  // Compose other class objects
```

8.3 INHERITANCE VS COMPOSITION

```
17  public:
18      void startEngine() { cout << "Engine Starts" << endl; }
19  };
20
21  class ElectricCar {  // Composition class
22  private:
23      Engine engine;    // Embedded Engine object
24      Vehicle vehicle;  // Compose other class objects
25  public:
26      void start() {
27          engine.startEngine();  // Call component functionality
28          vehicle.start();       // Reuse base class logic
29          cout << "ElectricCar Starts Finished" << endl;
30      }
31  };
19  // Composition example (has-a relationship)
```

8.3 INHERITANCE VS COMPOSITION

INHERITANCE VS COMPOSITION

8.3 INHERITANCE VS COMPOSITION

INHERITANCE VS COMPOSITION

- Both methods can reuse existing class functionality:

8.3 INHERITANCE VS COMPOSITION

INHERITANCE VS COMPOSITION

- Both methods can reuse existing class functionality:
  - Inheritance calls base class methods directly through the derived class (e.g., `Car` calls `Vehicle::start()`)

8.3 INHERITANCE VS COMPOSITION

INHERITANCE VS COMPOSITION

- Both methods can reuse existing class functionality:
  - Inheritance calls base class methods directly through the derived class (e.g., `Car` calls `Vehicle::start()`)
  - Composition calls its interface by holding an object (e.g., `ElectricCar` calls `Engine::startEngine()`)

8.3 INHERITANCE VS COMPOSITION

INHERITANCE VS COMPOSITION

Feature	Inheritance	Composition
Relationship Type	is-a	has-a
Coupling	High coupling (subclasses depend on base class implementation details, base class modifications may affect subclasses)	Low coupling (only interact through interfaces, internal modifications to component classes do not affect composite classes)
Reuse Method	White-box reuse (can override base class methods)	Black-box reuse (only use public interfaces of component classes)
Dynamic	Determined at compile time, cannot switch base class at runtime	Can dynamically switch components at runtime (such as replacing different types of engines)
Design Principle	Violates the single responsibility principle (subclasses may inherit unrelated features)	Follows the single responsibility principle (each component is independently encapsulated)

8.3 INHERITANCE VS COMPOSITION

Good coding style

8.3 INHERITANCE VS COMPOSITION

Good coding style

Composition is better than inheritance!

8.3 INHERITANCE VS COMPOSITION

8.3 INHERITANCE VS COMPOSITION

- In the following scenarios, inheritance is prioritized:

8.3 INHERITANCE VS COMPOSITION

- In the following scenarios, inheritance is prioritized:
  - Need to extend base class functionality (e.g., `Car` overrides the `start()` method)

8.3 INHERITANCE VS COMPOSITION

- In the following scenarios, inheritance is prioritized:
  - Need to extend base class functionality (e.g., `Car` overrides the `start()` method)
  - Need polymorphic features (e.g., operate different derived class objects through base class pointers)

8.3 INHERITANCE VS COMPOSITION

8.3 INHERITANCE VS COMPOSITION

- In the following scenarios, inheritance is prioritized:

8.3 INHERITANCE VS COMPOSITION

- In the following scenarios, inheritance is prioritized:
  - Need to dynamically replace components (such as freely switching between fuel engines and electric engines)

8.3 INHERITANCE VS COMPOSITION

- In the following scenarios, inheritance is prioritized:
  - Need to dynamically replace components (such as freely switching between fuel engines and electric engines)
  - Avoid excessive hierarchical nesting (composition can replace complex multiple inheritance scenarios)

8.4 HOW MULTIPLE INHERITANCE WORKS

- Multiple different parent classes derive the same child class
- Or a child class inherits multiple different parent classes
- According to the relationship between composition and inheritance, since classes can have multiple member objects, classes can also inherit multiple parent classes to have the characteristics of multiple parent classes

8.4 HOW MULTIPLE INHERITANCE WORKS

Syntax:

```
class Derived : public Base1, protected Base2, private Base3, ... {
 ...
};
```

8.4 HOW MULTIPLE INHERITANCE WORKS

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  // Base class 1: Flying ability
 5  class Flyable {
 6  public:
 7      void fly() {
 8          cout << "Using jet engine to fly" << endl;
 9      }
10  };
11
12  // Base class 2: Swimming ability
13  class Swimmable {
14  public:
15      void swim() {
```

8.4 HOW MULTIPLE INHERITANCE WORKS

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

// Base class 1: Flying ability
class Flyable {
public:
    void fly() {
        cout << "Using jet engine to fly" << endl;
    }
};

// Base class 2: Swimming ability
class Swimmable {
public:
    void swim() {
```

8.4 HOW MULTIPLE INHERITANCE WORKS

```
        cout << "Using jet engine to fly" << endl;
    }
};

// Base class 2: Swimming ability
class Swimmable {
public:
    void swim() {
        cout << "Using propeller to navigate" << endl;
    }
};

// Derived class: Multiple inheritance of two base classes
class AmphibiousAircraft : public Flyable, public Swimmable {
public:
    void swim() {
```

8.4 HOW MULTIPLE INHERITANCE WORKS

```
17      }
18  };
19
20  // Derived class: Multiple inheritance of two base classes
21  class AmphibiousAircraft : public Flyable, public Swimmable {
22  public:
23      void showMode() {
24          cout << "=== Switch Mode ===" << endl;
25          fly();  // Call the first base class method
26          swim(); // Call the second base class method
27      }
28  };
29
30  int main() {
31      AmphibiousAircraft aa;
```

8.4 HOW MULTIPLE INHERITANCE WORKS

```
24          cout << "=== Switch Mode ===" << endl;
25          fly();  // Call the first base class method
26          swim(); // Call the second base class method
27      }
28  };
29
30  int main() {
31      AmphibiousAircraft aa;
32
33      // Directly access member functions of different base classes
34      aa.fly();    // Output: Using jet engine to fly
35      aa.swim();   // Output: Using propeller to navigate
36
37      // Call the derived class's own method
```

8.4 HOW MULTIPLE INHERITANCE WORKS

```
27      }
28  };
29
30  int main() {
31      AmphibiousAircraft aa;
32
33      // Directly access member functions of different base classes
34      aa.fly();    // Output: Using jet engine to fly
35      aa.swim();   // Output: Using propeller to navigate
36
37      // Call the derived class's own method
38      aa.showMode();
39
40      return 0;
41  }
    void swim() {
```

8.4 HOW MULTIPLE INHERITANCE WORKS

```
    }
};

int main() {
    AmphibiousAircraft aa;

    // Directly access member functions of different base classes
    aa.fly();    // Output: Using jet engine to fly
    aa.swim();   // Output: Using propeller to navigate

    // Call the derived class's own method
    aa.showMode();

    return 0;
}
    void swim() {
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

- Multiple inheritance must face the problem of parent class name conflicts

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

- Multiple inheritance must face the problem of parent class name conflicts
- The member functions of the subclass from parent class A and parent class B may be completely identical in form

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

- Multiple inheritance must face the problem of parent class name conflicts
- The member functions of the subclass from parent class A and parent class B may be completely identical in form
- Application programming must understand the details of multiple inheritance, which is contrary to the purpose of object-oriented programming

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
1  #include <iostream>
2  using namespace std;
3
4  // Base class 1: Electronic device
5  class ElectronicDevice {
6  public:
7      void powerOn() { // Same name member function
8          cout << "Electronic device power on" << endl;
9      }
10 };
11
12 // Base class 2: Mechanical device
13 class MechanicalDevice {
14 public:
15     void powerOn() { // Same name member function
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
1  #include <iostream>
2  using namespace std;
3
4  // Base class 1: Electronic device
5  class ElectronicDevice {
6  public:
7      void powerOn() { // Same name member function
8          cout << "Electronic device power on" << endl;
9      }
10 };
11
12 // Base class 2: Mechanical device
13 class MechanicalDevice {
14 public:
15     void powerOn() { // Same name member function
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
        cout << "Electronic device power on" << endl;
    }
};

// Base class 2: Mechanical device
class MechanicalDevice {
public:
    void powerOn() { // Same name member function
        cout << "Mechanical device power on" << endl;
    }
};

// Derived class: Smart robot (multiple inheritance)
class Robot : public ElectronicDevice, public MechanicalDevice {
public:
    void powerOn() { // Same name member function
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
20  // Derived class: Smart robot (multiple inheritance)
21  class Robot : public ElectronicDevice, public MechanicalDevice {
22  public:
23      void startup() {
24          // Direct call will cause ambiguity
25          // powerOn();  // ❌ Compilation error: 'powerOn' is ambiguous
26
27          // Solution 1: Use scope qualifier
28          ElectronicDevice::powerOn();  // Explicitly call the specified base class version
29          MechanicalDevice::powerOn();
30
31          // Solution 2: Redefine unified interface in derived class
32          combinedPowerOn();
33      }
34      void powerOn() { // Same name member function
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
31          // Solution 2: Redefine unified interface in derived class
32          combinedPowerOn();
33      }
34
35      // Solution 2: Create unified interface
36      void combinedPowerOn() {
37          cout << "==== System startup ====" << endl;
38          ElectronicDevice::powerOn();
39          MechanicalDevice::powerOn();
40      }
41  };
42
43  int main() {
44      // Basic multiple inheritance conflict resolution
45      void powerOn() { // Same name member function
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

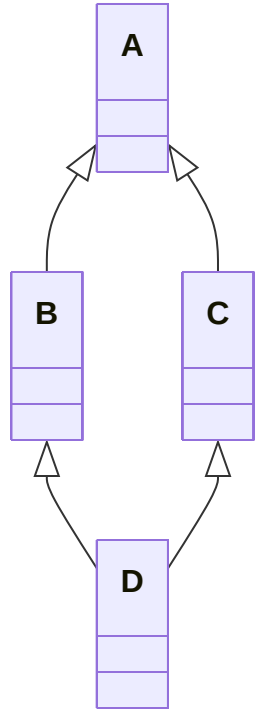
```
38          ElectronicDevice::powerOn();
39          MechanicalDevice::powerOn();
40      }
41  };
42
43  int main() {
44      // Basic multiple inheritance conflict resolution
45      Robot bot;
46      bot.startup();
47
48      // Need to explicitly specify scope when calling externally
49      bot.ElectronicDevice::powerOn();
50      bot.MechanicalDevice::powerOn();
51
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
39          MechanicalDevice::powerOn();
40      }
41  };
42
43  int main() {
44      // Basic multiple inheritance conflict resolution
45      Robot bot;
46      bot.startup();
47
48      // Need to explicitly specify scope when calling externally
49      bot.ElectronicDevice::powerOn();
50      bot.MechanicalDevice::powerOn();
51
52      return 0;
53  }
54      void powerOn() { // Same name member function
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

DIAMOND INHERITANCE PROBLEM



8.5 AMBIGUITY IN MULTIPLE INHERITANCE

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

// Public base class
class Animal {
public:
    int age;
    void eat() { cout << "Animal eating" << endl; }
};

// First Inheritance Path
class Mammal : public Animal {
public:
    void breathe() { cout << "Mammal breathing" << endl; }
};
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

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

// Public base class
class Animal {
public:
    int age;
    void eat() { cout << "Animal eating" << endl; }
};

// First Inheritance Path
class Mammal : public Animal {
public:
    void breathe() { cout << "Mammal breathing" << endl; }
};
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
public:
    int age;
    void eat() { cout << "Animal eating" << endl; }
};

// First Inheritance Path
class Mammal : public Animal {
public:
    void breathe() { cout << "Mammal breathing" << endl; }
};

// Second Inheritance Path
class Bird : public Animal {
public:
    void fly() { cout << "Bird flying" << endl; }
};
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
12  class Mammal : public Animal {
13  public:
14      void breathe() { cout << "Mammal breathing" << endl; }
15  };
16
17  // Second Inheritance Path
18  class Bird : public Animal {
19  public:
20      void fly() { cout << "Bird flying" << endl; }
21  };
22
23  // Derived class of diamond inheritance
24  class Platypus : public Mammal, public Bird {
25  public:
26      void layEggs() { cout << "Platypus laying eggs" << endl; }
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
18  class Bird : public Animal {
19  public:
20      void fly() { cout << "Bird flying" << endl; }
21  };
22
23  // Derived class of diamond inheritance
24  class Platypus : public Mammal, public Bird {
25  public:
26      void layEggs() { cout << "Platypus laying eggs" << endl; }
27  };
28
29  int main() {
30      Platypus perry;
31
32      // Ambiguous access
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
// Derived class of diamond inheritance
class Platypus : public Mammal, public Bird {
public:
    void layEggs() { cout << "Platypus laying eggs" << endl; }
};

int main() {
    Platypus perry;

    // Ambiguous access
    // perry.age = 2; // ❌ Compilation error: ambiguous access of 'age'
    // perry.eat(); // ❌ Compilation error: ambiguous access of 'eat'

    // Explicit path specification can solve the problem
    perry.Mammal::age = 2;  // ✅ Explicitly use the age of the Mammal path
}
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
26      void layEggs() { cout << "Platypus laying eggs" << endl; }
27  };
28
29  int main() {
30      Platypus perry;
31
32      // Ambiguous access
33      // perry.age = 2; // ❌ Compilation error: ambiguous access of 'age'
34      // perry.eat(); // ❌ Compilation error: ambiguous access of 'eat'
35
36      // Explicit path specification can solve the problem
37      perry.Mammal::age = 2;  // ✅ Explicitly use the age of the Mammal path
38      perry.Bird::eat();      // ✅ Explicitly use the eat of the Bird path
39
40      // Verify the existence of two base class copies
    };
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
30      Platypus perry;
31
32      // Ambiguous access
33      // perry.age = 2; // ❌ Compilation error: ambiguous access of 'age'
34      // perry.eat(); // ❌ Compilation error: ambiguous access of 'eat'
35
36      // Explicit path specification can solve the problem
37      perry.Mammal::age = 2;  // ✅ Explicitly use the age of the Mammal path
38      perry.Bird::eat();      // ✅ Explicitly use the eat of the Bird path
39
40      // Verify the existence of two base class copies
41      cout << "Mammal's age: " << perry.Mammal::age << endl;
42      perry.Mammal::age = 3;
43      // Output different values
44      cout << "Bird's age: " << perry.Bird::age << endl;
```

8.5 AMBIGUITY IN MULTIPLE INHERITANCE

```
    // perry.age = 2; // ❌ Compilation error: ambiguous access of 'age'
    // perry.eat(); // ❌ Compilation error: ambiguous access of 'eat'

    // Explicit path specification can solve the problem
    perry.Mammal::age = 2;  // ✅ Explicitly use the age of the Mammal path
    perry.Bird::eat();      // ✅ Explicitly use the eat of the Bird path

    // Verify the existence of two base class copies
    cout << "Mammal's age: " << perry.Mammal::age << endl;
    perry.Mammal::age = 3;
    // Output different values
    cout << "Bird's age: " << perry.Bird::age << endl;

    return 0;
}
};
```

8.6 VIRTUAL INHERITANCE

Core problem of diamond inheritance：

- The derived class contains two copies of the Animal member
- The compiler cannot determine which copy to use when directly accessing common base class members

8.6 VIRTUAL INHERITANCE

```
1  // Change inheritance to virtual inheritance
2  class Mammal : virtual public Animal {};  // Virtual inheritance
3  class Bird : virtual public Animal {};     // Virtual inheritance
4
5  class Platypus : public Mammal, public Bird {
6  public:
7      void layEggs() {
8          age = 2;         // ✅ Can directly access (unique copy)
9          eat();           // ✅ No ambiguity
10     }
11 };
12
13 int main() {
14     Platypus perry;
15     perry.age = 5;       // ✅ Normal access
```

8.6 VIRTUAL INHERITANCE

```
1  // Change inheritance to virtual inheritance
2  class Mammal : virtual public Animal {};  // Virtual inheritance
3  class Bird : virtual public Animal {};    // Virtual inheritance
4
5  class Platypus : public Mammal, public Bird {
6  public:
7      void layEggs() {
8          age = 2;        // ✅ Can directly access (unique copy)
9          eat();          // ✅ No ambiguity
10     }
11 };
12
13 int main() {
14     Platypus perry;
15     perry.age = 5;      // ✅ Normal access
```

8.6 VIRTUAL INHERITANCE

```cpp
// Change inheritance to virtual inheritance
class Mammal : virtual public Animal {};  // Virtual inheritance
class Bird : virtual public Animal {};    // Virtual inheritance

class Platypus : public Mammal, public Bird {
public:
    void layEggs() {
        age = 2;        // ✅ Can directly access (unique copy)
        eat();          // ✅ No ambiguity
    }
};

int main() {
    Platypus perry;
    perry.age = 5;      // ✅ Normal access
```

8.6 VIRTUAL INHERITANCE

```
class Platypus : public Mammal, public Bird {
public:
    void layEggs() {
        age = 2;        // ✅ Can directly access (unique copy)
        eat();          // ✅ No ambiguity
    }
};

int main() {
    Platypus perry;
    perry.age = 5;      // ✅ Normal access
    cout << perry.age;  // Output 5 (unique value)
    return 0;
}
```

8.6 VIRTUAL INHERITANCE

```
 1  class A {};
 2  class B1 : public A {};
 3  class B2 : public A {};
 4
 5  class C1 : public B1, public B2 {
 6    // B1 is real base, B2 is fake base.
 7  }
 8
 9  class C2 : public B2, public B1 {
10    // B2 is real base, B1 is fake base.
11  }
```

8.6 VIRTUAL INHERITANCE

```
1  class A {};
2  class B1 : public A {};
3  class B2 : public A {};
4
5  class C1 : public B1, public B2 {
6    // B1 is real base, B2 is fake base.
7  }
8
9  class C2 : public B2, public B1 {
10   // B2 is real base, B1 is fake base.
11 }
```

8.6 VIRTUAL INHERITANCE

```
1  class A {};
2  class B1 : public A {};
3  class B2 : public A {};
4
5  class C1 : public B1, public B2 {
6    // B1 is real base, B2 is fake base.
7  }
8
9  class C2 : public B2, public B1 {
10   // B2 is real base, B1 is fake base.
11 }
```

8.6 VIRTUAL INHERITANCE

```
1  class A {};
2  class B1 : public A {};
3  class B2 : public A {};
4
5  class C1 : public B1, public B2 {
6    // B1 is real base, B2 is fake base.
7  }
8
9  class C2 : public B2, public B1 {
10   // B2 is real base, B1 is fake base.
11 }
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

- First complete the construction of virtual inheritance

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

- First complete the construction of virtual inheritance
- Otherwise follow inheritance rules

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

- First complete the construction of virtual inheritance
- Otherwise follow inheritance rules
  - Construct non-virtual base classes first

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

- First complete the construction of virtual inheritance
- Otherwise follow inheritance rules
  - Construct non-virtual base classes first
  - Then construct member objects

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

- First complete the construction of virtual inheritance
- Otherwise follow inheritance rules
  - Construct non-virtual base classes first
  - Then construct member objects
  - Finally construct the derived class's own data

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

- First complete the construction of virtual inheritance
- Otherwise follow inheritance rules
  - Construct non-virtual base classes first
  - Then construct member objects
  - Finally construct the derived class's own data
- If there are multiple virtual base classes in the same layer, the construction of virtual base classes is called in the order they are declared

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

- First complete the construction of virtual inheritance
- Otherwise follow inheritance rules
  - Construct non-virtual base classes first
  - Then construct member objects
  - Finally construct the derived class's own data
- If there are multiple virtual base classes in the same layer, the construction of virtual base classes is called in the order they are declared
- The order of destruction is the opposite

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
1  #include <iostream>
2  using namespace std;
3
4  // Base class
5  class Base {
6  public:
7      Base() { cout << "Base constructor" << endl; }
8      ~Base() { cout << "Base destructor" << endl; }
9  };
10
11 // Ordinary inheritance of intermediate class 1
12 class Derived1 : public Base {
13 public:
14     Derived1() { cout << "Derived1 constructor" << endl; }
15     ~Derived1() { cout << "Derived1 destructor" << endl; }
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

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

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

// Ordinary inheritance of intermediate class 1
class Derived1 : public Base {
public:
    Derived1() { cout << "Derived1 constructor" << endl; }
    ~Derived1() { cout << "Derived1 destructor" << endl; }
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
 7      Base() { cout << "Base constructor" << endl; }
 8      ~Base() { cout << "Base destructor" << endl; }
 9  };
10
11  // Ordinary inheritance of intermediate class 1
12  class Derived1 : public Base {
13  public:
14      Derived1() { cout << "Derived1 constructor" << endl; }
15      ~Derived1() { cout << "Derived1 destructor" << endl; }
16  };
17
18  // Ordinary inheritance of intermediate class 2
19  class Derived2 : public Base {
20  public:
21      Derived2() { cout << "Derived2 destructor" << endl; }
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
14      Derived1() { cout << "Derived1 constructor" << endl; }
15      ~Derived1() { cout << "Derived1 destructor" << endl; }
16  };
17
18  // Ordinary inheritance of intermediate class 2
19  class Derived2 : public Base {
20  public:
21      Derived2() { cout << "Derived2 constructor" << endl; }
22      ~Derived2() { cout << "Derived2 destructor" << endl; }
23  };
24
25  // Virtual inheritance of intermediate class 1
26  class VirtualDerived1 : virtual public Base {
27  public:
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
    ~Derived2() { cout << "Derived2 destructor" << endl; }
};

// Virtual inheritance of intermediate class 1
class VirtualDerived1 : virtual public Base {
public:
    VirtualDerived1() { cout << "VirtualDerived1 constructor" << endl; }
    ~VirtualDerived1() {
        cout << "VirtualDerived1 destructor" << endl;
    }
};

// Virtual inheritance of intermediate class 2
class VirtualDerived2 : virtual public Base {
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
31      }
32  };
33
34  // Virtual inheritance of intermediate class 2
35  class VirtualDerived2 : virtual public Base {
36  public:
37      VirtualDerived2() { cout << "VirtualDerived2 constructor" << endl; }
38      ~VirtualDerived2() {
39          cout << "VirtualDerived2 destructor" << endl;
40      }
41  };
42
43  // Ordinary diamond inheritance of final derived class
44  class Diamond : public Derived1, public Derived2 {
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
39          cout << "VirtualDerived2 destructor" << endl;
40      }
41  };
42
43  // Ordinary diamond inheritance of final derived class
44  class Diamond : public Derived1, public Derived2 {
45  public:
46      Diamond() { cout << "Diamond constructor" << endl; }
47      ~Diamond() { cout << "Diamond destructor" << endl; }
48  };
49
50  // Final derived class using virtual inheritance
51  class VirtualDiamond : public VirtualDerived1, public VirtualDerived2 {
52  public:
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
48 };
49
50 // Final derived class using virtual inheritance
51 class VirtualDiamond : public VirtualDerived1, public VirtualDerived2 {
52 public:
53     VirtualDiamond() : Base() {  // Explicitly call virtual base class constructor
54         cout << "VirtualDiamond constructor" << endl;
55     }
56     ~VirtualDiamond() {
57         cout << "VirtualDiamond destructor" << endl;
58     }
59 };
60
61 int main() {
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

```
57          cout << "VirtualDiamond destructor" << endl;
58      }
59  };
60
61  int main() {
62      cout << "========== Ordinary diamond inheritance ==========\n";
63      Diamond d;
64
65      cout << "\n========== Virtual inheritance diamond inheritance ==========\n";
66      VirtualDiamond vd;
67
68      cout << "\n========== Destructor call ==========\n";
69
70      return 0;
71  }
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

Output content

```
========== Ordinary diamond inheritance ==========
Base constructor
Derived1 constructor
Base constructor
Derived2 constructor
Diamond constructor

========== Virtual inheritance diamond inheritance ==========
Base constructor
VirtualDerived1 constructor
VirtualDerived2 constructor
VirtualDiamond constructor

========== Destructor call ==========
VirtualDiamond destructor
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

Output content

```
========== Ordinary diamond inheritance ==========
Base constructor
Derived1 constructor
Base constructor
Derived2 constructor
Diamond constructor

========== Virtual inheritance diamond inheritance ==========
Base constructor
VirtualDerived1 constructor
VirtualDerived2 constructor
VirtualDiamond constructor

========== Destructor call ==========
VirtualDiamond destructor
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

Output content

```
3  Derived1 constructor
4  Base constructor
5  Derived2 constructor
6  Diamond constructor
7
8  ========== Virtual inheritance diamond inheritance ==========
9  Base constructor
10 VirtualDerived1 constructor
11 VirtualDerived2 constructor
12 VirtualDiamond constructor
13
14 ========== Destructor call ==========
15 VirtualDiamond destructor
16 VirtualDerived2 destructor
17 VirtualDerived1 destructor
```

8.7 CONSTRUCTION ORDERIN MULTIPLE INHERITANCE

Output content

```
9  Base constructor
10 VirtualDerived1 constructor
11 VirtualDerived2 constructor
12 VirtualDiamond constructor
13
14 ========== Destructor call ==========
15 VirtualDiamond destructor
16 VirtualDerived2 destructor
17 VirtualDerived1 destructor
18 Base destructor
19 Diamond destructor
20 Derived2 destructor
21 Base destructor
22 Derived1 destructor
23 Base destructor
```

ADDITIONAL READING

Modern GUI programming frameworks have largely abandoned inheritance, opting instead for composition, functional approaches, and more.

Reference materials:

- React (JavaScript frontend framework)
- ftxui (C++ modern TUI framework)