Множествено наследяване

титуляр на курса: д-р Тодор Цонков (ttsonkov@gmail.com)

От какво ще се състои настоящата лекция?

Множествено наследяване. Диамантен проблем. Колекции от обекти в полиморфна йерархия. Копиране и триене. Пример с класа Student.

Какво е множествено наследяване?

Множествено наследяване: клас може да наследи функционалност от повече от един базов клас.

Ползи:

- Повече повторно използване на код;
- Може да реализира „интерфейс + поведение“ (mixins).

Рискове:

- Конфликти на имена (ambiguity);
- Диамантен проблем (повторно присъствие на една и съща базова част);

Синтаксис

```
struct A { void f() {} };

struct B { void g() {} };

struct C : public A, public B { };
```

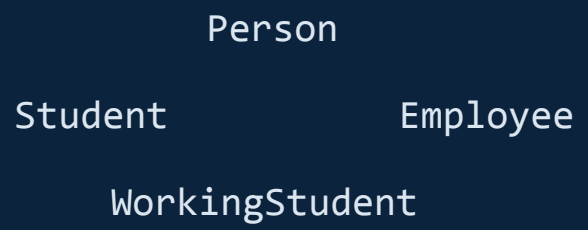
C има и A::f() и B::g().

Това дава възможности – комбиниране на поведения – но въвежда и усложнения (конфликти на имена, дублирани подобекти).

Какво е диамантен проблем?

Диамантът: базов Person, два класа Student и Employee наследяват Person, после

WorkingStudent наследява и двата → ромбоидна (диамант) структура.



Проблем: без virtual наследяване, WorkingStudent ще има две отделни Person подобекта (дублиране).

Синтаксис

```
struct Person { std::string name; };

struct Student : public Person { int grade; };

struct Employee : public Person { int id; };

struct WorkingStudent : public Student, public Employee { };
```

Проблеми:

Неясно коя Person::name използваме (working.name е ambiguous).

Увеличен размер на обекта.

Конструкторите на Person ще се извикват два пъти (една за всяка пътека).

Решение

```
//Използваме virtual наследяване:
struct Person {
    Person(std::string n): name(n) {}
    std::string name;
};
struct Student : virtual public Person {
  Student(std::string n,int g):
  Person(n), grade(g) {}
  int grade;
};
```

```
struct Employee : virtual public Person {
     Employee(std::string n,int i): Person(n), id(i)
     {

     }
     int id;
};
struct WorkingStudent : public Student, public Employee {
  WorkingStudent(std::string n,int g,int i)
    : Person(n), Student(n,g), Employee(n,i)
     {}
};
```

Конструиране при виртуален базов клас

Проблем

При диамантено наследяване без virtual, базовият клас се създава два пъти.
С virtual наследяване C++ гарантира един-единствен shared базов под-обект.

Но възниква въпросът: Кой конструира този общ базов клас?

Решение

```cpp
struct Person {
    Person(std::string n) : name(n) {
        std::cout << "Person\n";
    }
    std::string name;
};

struct Student : virtual public Person {
    Student(std::string n) : Person(n) {
        std::cout << "Student\n";
    }
};

struct Employee : virtual public Person {
    Employee(std::string n) : Person(n) {
        std::cout << "Employee\n";
    }
};

struct WorkingStudent : public Student, public Employee {
    WorkingStudent(std::string n)
        : Person(n), Student(n), Employee(n) {
        std::cout << "WorkingStudent\n";
    }
};
```

Кой конструира класа?

Редът е:

Person("Ivan") ← конструира се САМО ТУК

Student, Employee, WorkingStudent

👉 Конструкторите на Student и Employee НЕ конструират Person, дори да го имат в initializer list.

```
struct Student : virtual Person {

    Student() : Person("X") {} // игнорира се!

};
```

⚠️ Това няма ефект, ако Student не е най-отдаленият клас.

Какво се случва?

При virtual наследяване компилаторът:

не може да знае на compile-time точния offset до базовия клас и добавя допълнителни указатели/offset таблици, което прави достъпа индиректен

Илюстрация (концептуално)

Без virtual:

[Student], [Person]

С virtual:

[Student],[ptr → Person],[Person]

Object slicing

При копиране на производен обект по стойност в базов тип, производната част се „отрязва“.

```
struct Person {
    std::string name;
};
struct Student : public Person {
    int grade;
};
Student s{"Ivan", 5};
Person p = s;   // ❌ slicing
```

Какво се случва?

`p.name  == "Ivan"`

`p.grade == ❌ не съществува`

Решение:

1️⃣ Използване на указатели

```
Person* p = new Student{};
```

2️⃣ Използване на референции

```
void f(const Person& p);
```

3️⃣ Контейнери с умни указатели

```
std::vector<std::unique_ptr<Person>>
```

Клониране на полиморфни обекти

Имаме полиморфна йерархия и искаме:

1)да копираме обекти през базов тип;

2)да запазим реалния (динамичния) тип;

3)да избегнем object slicing.

❌ Невъзможно с обикновен copy constructor:

```
Person p2 = p1; // slicing
```

Защо не работи?

```
struct Person {
    std::string name;
};
struct Student : Person {
    int grade;
};
Person* p = new Student{"Ivan", 5};
Person* q = new Person(*p);  // ❌ slicing
```

➡️ q е Person, не Student.

Производната част е загубена завинаги.

Решение

Идеята на Clone pattern

„Всеки обект знае как да направи копие от себе си.“.Базовият клас декларира виртуална функция clone().Всяка производна класа връща копие от собствения си тип

```
struct Person {
    std::string name;
    Person(std::string n) : name(n) {}
    virtual ~Person() {}  // задължително!
    virtual Person* clone() const = 0;
};
```

Какво се случва?

```
struct Student : public Person {
    int grade;
    Student(std::string n, int g)
        : Person(n), grade(g) {}
    Student* clone() const override {
        return new Student(*this); // копира целия обект
    }
};
```

Реален пример

Решение

```
struct School {
    std::vector<Person*> members;
    School(const School& other) {
        for (Person* p : other.members) {
            members.push_back(p->clone());
        }
    }
    ~School() {
        for (Person* p : members)  delete p;
    }
};
```

Какви грешки се допускат?

❌ Грешка 1 – липсва virtual деструктор

```
struct Person { ~Person() {} };  // ❌
```

❌ Грешка 2 – clone не е virtual

```
Person* clone(); // ❌
```

❌ Грешка 3 – връщане по стойност

```
Person clone(); // ❌ slicing
```

Оправено в C++ 26: std::polymorphic_value

Колекция от полиморфни обекти.

Колекция от полиморфни обекти е контейнер, който съхранява обекти от различни конкретни типове, но ги третира чрез общ базов интерфейс.

Печелим: Runtime полиморфизъм, Отвореност за разширение (Open/Closed), Единен интерфейс за различни поведения.

```
struct Shape {

    virtual ~Shape() = default;

    virtual double area() const = 0;

};
```

Пример

```
struct Circle : Shape {

    double r;

    double area() const override { return 3.14 *
r * r; }

};

struct Rectangle : Shape {

    double w, h;

    double area() const override { return w * h;
}

};

std::vector<std::unique_ptr<Shape>> shapes;
```

Добри практики

1)Предпочитайте композиция пред множествено наследяване, когато е възможно.

2)Ако използвате множествено наследяване за интерфейси (без данни) – безопасно е.

3)За споделена базова под-обектност (diamond) – използвайте virtual наследяване.

4)Избягвайте slicing – съхранявайте полиморфни обекти чрез unique_ptr/shared_ptr.

5)За копиране използвайте виртуален clone() (или pimpl/immutable обекти).

6)Винаги правете базов деструктор virtual.

7)Документирайте ownership semantics (кой е собственик, кой копира, кой изтрива).

Пример на Java

Пример

```java
public interface Learner {
    int getGrade();
}

public interface Worker {
    int getEmployeeId();
}

//public class Person
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class Student extends Person implements Learner {
    private int grade;

    public Student(String name, int grade) {
        super(name);
        this.grade = grade;
    }

    @Override
    public int getGrade() {
        return grade;
    }
}
```

Пример на Java

Пример

```
public class WorkingStudent extends Student implements Worker {
    private int employeeId;

    public WorkingStudent(String name, int grade, int employeeId) {
        super(name, grade);
        this.employeeId = employeeId;
    }

    @Override
    public int getEmployeeId() {
        return employeeId;
    }
}

public class Main {
    public static void main(String[] args) {
        WorkingStudent ws = new WorkingStudent("Alex", 6, 12345);

        System.out.println(ws.getName());
        System.out.println(ws.getGrade());
        System.out.println(ws.getEmployeeId());
    }
}
```

Въпроси?

Благодаря за вниманието!