

# Множествено наследяване.

Множествено наследяване. Диамантен проблем. Колекции от обекти в полиморфна йерархия. Копиране и триене. Примери.

д-р Тодор Цонков  
todort@uni-sofia.bg

# 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);

Пример:

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

```
struct B { void g() {} };
```

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

С има и A::f() и B::g().

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

# Какво е mixin?

Клас, предназначен да бъде наследяван, който добавя конкретно поведение (имплементация) към друг клас, без самият той да представлява самостоятелен концептуален тип.

По-просто казано:

✗ Не моделира „is-a“ връзка (не е истински базов тип в домейна)

✓ Добавя функционалност

✓ Обикновено не се използва самостоятелно

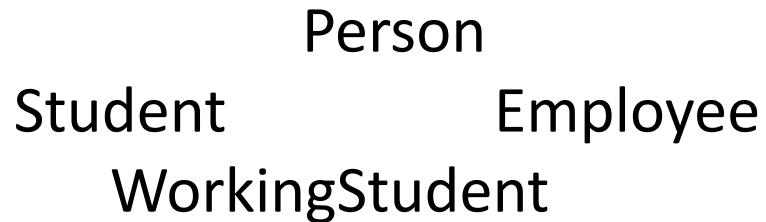
✓ Често съдържа частично реализиран интерфейс

✓ Използва множествено наследяване

```
struct Identifiable {  
    virtual int id() const = 0;  
    virtual ~Identifiable() = default;  
};  
// Mixin  
class PrintableIdMixin : public Identifiable  
{public:  
    void printId() const {  
        std::print("ID: {}\n", id());  
    }  
};
```

# 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).

```
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).

# Виртуално наследяване

Виртуалното наследяване е механизъм в C++, при който базов клас се наследява със спецификатора `virtual`, така че при множествено наследяване да съществува само една споделена инстанция на този базов клас в обекта от най-долния тип.

То решава проблема с диамантеното наследяване (diamond problem).

```
class A {};  
class B : virtual public A {};  
class C : virtual public A {};  
class D : public B, public C {};
```

По този начин D съдържа само едно копие на A.

При виртуално наследяване: Обектът съдържа една обща базова под-обектна инстанция.

Компилаторът добавя вътрешен механизъм (скрит указател), чрез който се намира споделеният базов клас.

Достъпът до виртуалната база става индиректен.

# Конструиране при виртуален базов клас

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?

👉 Най-долният клас (most derived class) е отговорен за конструирането на виртуалната база.

```
struct A {  
    A(int) {}  
};  
  
struct B : virtual A {  
    B() : A(1) {} // игнорира се  
};  
  
struct C : virtual A {  
    C() : A(2) {} // игнорира се  
};  
  
struct D : B, C {  
    D() : A(42), B(), C() {} // ТУК реално се  
    конструира A  
};
```

# Виртуално наследяване - обобщение

Виртуално наследяване се използва когато:

- 1)Имаме общ абстрактен базов клас
  - 2)Очакваме множествено наследяване
  - 3)Искаме да избегнем дублиране на базовия обект
  - 4)Проектираме йерархии тип „интерфейс + имплементации“
- Класически пример: `std::istream` и `std::ostream`, които виртуално наследяват `std::ios`.

Виртуалното наследяване:

- Решава проблема с множественото копиране на общ базов клас
- Осигурява единствена споделена инстанция
- Променя правилата за конструиране
- Въвежда допълнителна индирекция в обектното представяне

# 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
```

p.name == "Ivan"

p.grade == ❌ не съществува

Решение:

① Използване на указатели

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

② Използване на референции

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


③ 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
```

```
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.

# Clone pattern

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;  
};
```

```
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  
    }  
};
```

# Множествено наследяване - грешки

✗ 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
```

Често се използва множествено наследяване за повторна употреба на код, вместо за моделиране на типова йерархия.

Пример за лош дизайн:

```
class DatabaseConnection {};
```

```
class Logger {};
```

```
class UserService : public
```

```
DatabaseConnection, public Logger {};
```

Тук UserService не е нито DatabaseConnection, нито Logger.

Това е грешно „is-a“ отношение → трябва композиция.

# 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;  
  
};
```

```
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;
```

# Множествено наследяване - добри практики

- 1) Да се предпочита композиция пред множествено наследяване, когато е възможно.
- 2) Ако се използва множествено наследяване за интерфейси (без данни) — безопасно е.
- 3) За споделена базова под-обектност (diamond) — да се използва virtual наследяване.
- 4) Да се избягва slicing — да се съхраняват полиморфни обекти чрез unique\_ptr/shared\_ptr.

# Множествено наследяване - добри практики

- 5) За копиране да се ползва виртуален clone() (или pimpl/immutable обекти).
- 6) Винаги да се прави базов деструктор virtual.
- 7) Да се документира, ако не е достатъчно ясно от кода - ownership semantics (кой е собственик, кой копира, кой изтрива).

# Реален пример за множествоно наследяване

GUI framework (реален  
архитектурен сценарий)

Например UI система:

```
class Widget { ... };    // базов UI
```

елемент

```
class Focusable : virtual public Widget  
{ ... };
```

```
class Clickable : virtual public Widget {  
... };
```

```
class Button : public Focusable, public  
Clickable { ... };
```

```
class Entity {
```

```
public:
```

```
    int id;
```

```
    Transform transform;
```

```
};
```

```
class PhysicsObject : virtual public  
Entity {};
```

```
class Renderable : virtual public Entity  
{};
```

```
class Player : public PhysicsObject,  
public Renderable {};
```

# Пример с езика С#

```
using System;

public interface IReadable
{
    void Read();
}

public interface IWritable
{
    void Write();
}

public class File : IReadable, IWritable
{
    public void Read()
    {
        Console.WriteLine("Reading...");
    }

    public void Write()
    {
        Console.WriteLine("Writing...");
    }
}

class Program
{
    static void Main()
    {
        File file = new File();

        file.Read();
        file.Write();

        IReadable r = file; // полиморфизъм
        IWritable w = file; // полиморфизъм
    }
}
```

**Въпроси?**

**Благодаря за вниманието!**