Полиморфизъм.

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

От какво ще се състои настоящата лекция?

Статично и динамично свързване. Виртуални функции. Ключови думи - override, final. Виртуални таблици. Полиморфизъм - runtime and compile time. Примери.

Какво прави ключовата дума virtual?

`virtual` означава: методът може да бъде презаписан (`overridden`) в наследниците на класа и извикването му при `runtime` да избере правилната имплементация.

Позволява динамично свързване (`dynamic dispatch`).

Използва се най-често за дефиниране на абстрактни интерфейси и полиморфизъм.

Синтаксис

```
struct A
{
      virtual void f() { /* статично свързване
*/
    }
};

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

Статично и динамично свързване

Статично свързване: компилаторът решава коя функция да се извика (например non-virtual функции, inline, templates). Бързо (no indirection).Известно още като (compile time binding).

Динамично свързване: изборът се прави при изпълнение чрез vtable; позволява поведение според действителния (runtime) тип. (runtime binding)

Кога се използва кой: при нужда от runtime binding се използва virtual.

Синтаксис

```
struct Base {

void f() { /* статично */ }

virtual void g() { /* динамично */ }

};

Base* p = new Derived;

p->f(); // винаги Base::f()

p->g(); // ако g е виртуална ->
Derived::g() при runtime
```

Виртуални функция - основни правила

Трябва да се декларира `virtual` в базовия клас.
В наследника да се имплементира функция със същия подпис (и по силно се препоръчва `override`).

Ако базовата функция е `virtual`, всички наследници автоматично имат виртуално поведение за този метод (без да се повтаря `virtual`).

Възможни са и чисти виртуални функции (`= 0`) → правят класа абстрактен.

Синтаксис

```
struct Base {
    virtual void speak() { std::cout << "Base\n"; }
    virtual ~Base() = default;
};
struct Derived : Base {
    void speak() override { std::cout << "Derived\n"; }
};
//Деструкторът трябва да е винаги виртуален!
```

Override и final ключови думи

Ключовата дума override указва, че функцията е предназначена да презапише виртуална функция от базов клас и кара компилатора да провери коректността на сигнатурата (име, параметри, const-квалификатори, ref-квалификатори и др.).

Компилаторът ще даде грешка при несъвпадение на подписа.

final може да маркира метод като непрезаписваем или клас като непренаследваем.

Използвайте override винаги – предотвратява тихи грешки.

Синтаксис

```
struct B {
    virtual void f(int) {}
};
struct D : B {
    void f(int) override {}       // OK
    // void f(double) override {} // ERROR - няма съвпадаща виртуална функция
};
struct NoMore : D final {};      // NoMore не може да бъде наследяван
```

Пример предметната област Student

```cpp
#include <iostream>
#include <memory>
#include <vector>

struct Person {
    virtual void describe() const = 0;
    virtual ~Person() = default;
};

struct Student : Person {
    void describe() const override { std::cout <<
"Student\n"; }
};

struct Teacher : Person {
    void describe() const override final { std::cout <<
"Teacher\n"; } // final: няма да позволим override в
наследници
};

struct AssistantTeacher : Teacher {
    // void describe() const override {} // ERROR -
Teacher::describe is final
};
int main() {
    std::vector<std::unique_ptr<Person>> group;
    group.push_back(std::make_unique<Student>());
    group.push_back(std::make_unique<Teacher>());
    for (const auto& p : group)
      p->describe(); // dynamic dispatch чрез vtable
}
```

Защо override е важен?

Обяснение

При неправилна функция (напр. различен тип аргумент), без override кодът може да компилира, но методът няма да бъде викнат полиморфно.

override прави грешката явна при компилация.

Пример

```
struct B {
  virtual void f(int) {}
};

struct D : B
{
        void f(double)
        { /* НЕ override */
        }
};

B* p = new D;
p->f(1); // извиква B::f (изненадващо)

)
```

Пример

```
struct Derived : Base {
   void print() override {
      std::cout << "Derived\n";
   }
};
```

Компилационна грешка:
function marked override but does not override any base class function

Чисти виртуални функции virtual = 0;

Чиста виртуална функция: virtual void foo() = 0;
Клас с поне една чиста виртуална функция е абстрактен – не може да се инстанцира.
Полезно за интерфейси.

```
struct Animal {
    virtual void speak() = 0; // интерфейс
    virtual ~Animal() = default;
};
struct Dog : Animal {
    void speak() override { std::cout << "Woof\n";
}
};
```

Виртуални деструктори

Ако имате полиморфни класове и триете чрез Base*, базовият деструктор трябва да е виртуален, иначе има undefined behavior (ресурси не се освобождават правилно).

```
struct Base { virtual ~Base() = default; };

struct Derived : Base { ~Derived() { /*
освобождаване */ }

};

Base* p = new Derived;

delete p; // безопасно, извиква
Derived::~Derived()
```

Virtual Table

За всеки клас с виртуални функции компилаторът генерира vtable (една таблица на клас).

Всеки обект съдържа указател (vptr) към vtable на своя реален клас.

При виртуално извикване компилаторът следва vptr → намира адрес на функция в vtable → косвено извикване.

Връзка: един vtable на клас (споделен), един vptr на обект.

Virtual Table пояснения

Клас Base има vtable с елементи: address of Base::f, Base::~Base, ...

Derived vtable съдържа адреси към Derived::f и/или към Base ако неизменени.

Обект на Derived има vptr → point to Derived::vtable.

Това е причината за runtime overhead: една индирекция при извикване и невинаги възможност за inlining.

Чести грешки

❌ Забравен virtual деструктор
❌ Липса на override
❌ Object slicing
❌ Virtual в performance-critical код
❌ Лоша ownership семантика

```
struct Base {
    ~Base() {
        std::cout << "~Base\n";
    }
};

struct Derived : Base {
    ~Derived() {
        std::cout << "~Derived\n";
    }
};

int main() {
    Base* p = new Derived;
    delete p;   // ❌ UB
}
```

Примери

```
struct Base {

    Base() { f();  } // ❌

    virtual void f() { std::cout << "Base\n"; }

};

struct Derived : Base {

    void f() override { std::cout <<
"Derived\n"; }

};

int main() {

    Derived d;

}
```

Open/Closed principle (O от SOLID)

Идея:Класовете са:отворени за разширяване и затворени за модификация

❌ if / switch за типове

Типичен проблем:

```
if (type == CIRCLE) ...

else if (type == RECTANGLE) ...
```

➡ Всеки нов тип = нов if

✅ Полиморфизъм

Пример

```cpp
#include <iostream>
#include <memory>

class Shape {
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;
};

class Circle : public Shape {
    double r;
public:
    Circle(double r) : r(r) {}
    double area() const override { return 3.14 * r * r; }
};

class Rectangle : public Shape {
    double a, b;
public:
    Rectangle(double a, double b) : a(a), b(b) {}
    double area() const override { return a * b; }
};

int main() {
    std::unique_ptr<Shape> s1 = std::make_unique<Circle>(2.0);
    std::unique_ptr<Shape> s2 = std::make_unique<Rectangle>(3.0, 4.0);

    std::cout << s1->area() << "\n";
    std::cout << s2->area() << "\n";
}
```

Compile-time (статичен) полиморфизъм

Compile-time (static): templates, function overloading, constexpr, CRTP – изборът се прави при компилация (no vtable).

Runtime и compile-time имат свои предимства:
runtime е гъвкав (plug-in архитектури);
compile-time е по-бърз (оптимизируем, inlinable).

Пример за compile-time полиморфизъм (templates):

```
template<class T>
void doSpeak(const T& x) { x.speak(); } //
compile-time resolution
```

CRTP пример

```
template<typename Derived>struct BaseCRTP {

    void interface() { static_cast<const
Derived*>(this)->impl(); }

};

struct Derived : BaseCRTP<Derived> {

    void impl() const { std::cout << "Derived
impl\n"; }

};
```

CRTP позволява "полиморфизъм" без виртуални функции (без runtime overhead).
Подходящо когато типовете са известни при компилация.

Статичен полиморфизъм с std::variant

Нека разгледаме класическия пример с класа Shape и негови наследници - в случая нямаме базов общ клас и типовете са известни по време на компилация:

```
struct Circle {
    double r;
};

struct Rectangle {
    double w, h;
};

using Shape = std::variant<Circle, Rectangle>;
```

Пример

```
double area(const Shape& shape) {
    return std::visit([](const auto& s) -> double {
        using T = std::decay_t<decltype(s)>;

        if constexpr (std::is_same_v<T, Circle>)
            return 3.14 * s.r * s.r;
        else if constexpr (std::is_same_v<T, Rectangle>)
            return s.w * s.h;
    }, shape);
}
```

Обяснение на примера

```
using T = std::decay_t<decltype(shape)>;
decltype взима типа на променливата shape, а
std::decay_t премахва const, референции
Тук резултатът е чистия тип: T == Circle   //
или Rectangle
Да кажем, че е Circle:
За Circle:
if constexpr (true) {
    return 3.14 * shape.r * shape.r;
}

За Rectangle:
if constexpr (false) {
    // този код се ИЗРЯЗВА
}
```

Пример

Какво реално генерира компилатора!

```
double area_circle(const Circle& c) {
   return 3.14 * c.r * c.r;
}

double area_rectangle(const Rectangle& r) {
   return r.w * r.h;
}
И runtime:

switch (s.index()) {
   case 0: return area_circle(get<Circle>(s));
   case 1: return area_rectangle(get<Rectangle>(s));
}
```

Пример предметната област Student

```
class School {
public:
    void enroll(const Student& s) {
        cout << "Enroll student\n";
    }

    void enroll(const Teacher& t) {
        cout << "Hire teacher\n";
    }
}; //Compile time polymorphism

//Runtime пример
void printInfo(const Person& p) {   // референция!
    cout << p.role()
         << " | salary = " << p.getSalary()
         << "\n";
}

int main() {
    Student s("Ivan", 5.50);
    Teacher t("Maria", 2500);

    printInfo(s);   // Student::role()
    printInfo(t);   // Teacher::role()
}
```

Пример на Java

```java
abstract class Person {
    protected String name;

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

    public abstract String role();

    public double getSalary() {
        return 0.0;
    }
}

final class Teacher extends Person {
    private double salary;

    public Teacher(String name, double salary) {
        super(name);
        this.salary = salary;
    }

    @Override
    public String role() {
        return "Teacher";
    }

    @Override
    public double getSalary() {
        return salary;
    }
}
```

Въпроси?

Благодаря за вниманието!