Полиморфизъм.

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

д-р Тодор Цонков
todort@uni-sofia.bg

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.

```
struct A
{
  virtual void f() {
     std::print("A::f()\n"); // динамично свързване, virtual
  }
};
struct B : A
{
  void f() {
     std::print("B::f()\n"); // override на виртуалната функция
  }
};
```

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)

При нужда от runtime binding се използва virtual.

```
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

Основни правила при virtual функции

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.

```
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!
```

Ключови думи override и final

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 или клас като такъв, който не може да се наследи.

Използвайте override винаги — предотвратява тихи грешки.

```
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
```

Защо ключовата дума override е важна?

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.

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

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

```
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)

)
```

Какво е полиморфизъм?

Полиморфизъм означава „много форми“ – един и същ интерфейс, различно поведение.

В C++ той позволява да работим с обекти чрез общ тип, но реалното поведение да се определя от конкретния обект.

1)Compile-time полиморфизъм (статичен)

Решава се по време на компилация.

✔ Function overloading

✔ Operator overloading

✔ std::variant

✔ Templates (най-мощният статичен полиморфизъм) - специфично за C++

2)Runtime(динамичен) полиморфизъм

Решава се по време на изпълнение чрез наследяване + virtual функции.

Кое кога да ползваме?

Различни типове, известни на compile-time - **templates**

Малък затворен набор типове - **std::variant**

Разширяема йерархия, plugin архитектура - **virtual**

Performance critical код - **templates**

Пример с класа Student

```cpp
#pragma once
#include <string>
#include <vector>
#include <iostream>

class Student {
public:
    static constexpr double maxGPA = 4.0;

    // Конструктор с параметри
    Student(const std::string& name, int age)
        : name_(name), age_(age)
    {
        ++studentCount_;
    }

    // Забраняваме copy и move
    Student() = delete;
    ~Student() = default;

    // Getter-и
    const std::string& getName() const { return name_; }
    int getAge() const { return age_; }

    // Setter-и
    void setAge(int age) { age_ = age; }

    static int getStudentCount() { return studentCount_; }

private:
    std::string name_;
    int age_;
    static int studentCount_;
};
```

Пример с класа Student

```cpp
class BachelorStudent : public Student {
    double examScore;

public:
    BachelorStudent(std::string name, double score)
        : Student(std::move(name)), examScore(score) {}

    // override гарантира, че наистина предефинираме virtual функция
    double calculateGrade() const override {
        return examScore * 0.06;
    }

    void printInfo() const override {
        std::print("Bachelor: {}, grade: {}\n", name, calculateGrade());
    }
};

class MasterStudent final : public Student {
    double thesisScore;

public:
    MasterStudent(std::string name, double score)
        : Student(std::move(name)), thesisScore(score) {}

    double calculateGrade() const override {
        return thesisScore * 0.1;
    }

    void printInfo() const override {
        std::print("Master: {}, grade: {}\n", name, calculateGrade());
    }
};

int main() {
    std::vector<std::unique_ptr<Student>> students;

    students.push_back(std::make_unique<BachelorStudent>("Ivan", 85));
    students.push_back(std::make_unique<MasterStudent>("Maria", 92));

    for (const auto& s : students)
        s->printInfo();  // runtime dispatch
}
```

Какво е виртуалната таблица?

Виртуална таблица (vtable, virtual table) е механизъм, който компилаторът използва в C++, за да реализира динамичен (runtime) полиморфизъм чрез virtual функции.
Когато един клас има поне една virtual функция, компилаторът:

Създава виртуална таблица за този клас

Всеки обект от този клас получава скрит указател (vptr) към тази таблица

При извикване на virtual функция през указател/референция към базов клас, се използва vtable, за да се избере правилната имплементация

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

Derived vtable съдържа адреси към Derived::f и/или към Base ако са неизменени.
Обект на Derived има vptr → point to Derived::vtable.

Виртуалната таблица - продължение

Base има virtual функция → получава vtable

Derived също има vtable (с override)
Обектът d има скрит vptr

При ref.speak():
не се гледа статичният тип (Base)
използва се vptr

от vtable се взима адресът на Derived::speak()

извиква се правилната функция

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";
    }
};
```

Виртуалната таблица - разяснения

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

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

Обект в паметта изглежда приблизително така:

[d.vptr] ---> [vtable_Derived]
[data members...]

А vtable изглежда така:

vtable_Derived:
  &Derived::speak

Кога се използва vtable?

Само когато имаме virtual функции и извикването става през указател или референция към базов клас

Vtable е имплементационен детайл (не е част от стандарта), но всички компилатори я използват.

Статичен полиморфизъм

Compile-time (static): templates, function overloading, constexpr, CRTP — изборът се прави при компилация

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"; }
```

Статичен полиморфизъм

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

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>;
```

Използваме std::visit, който извиква правилната функция според текущия тип вътре в variant.

Важно:

Няма virtual, няма vtable

Няма dynamic dispatch логика.

Компилаторът генерира всички възможни комбинации

Статичен полиморфизъм

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

double area(const Shape& shape)
{
    return std::visit([](const auto& s) -> double {
        if constexpr (std::is_same_v<std::decay_t<decltype(s)>, Circle>)
            return 3.14159 * s.radius * s.radius;
        else if constexpr (std::is_same_v<std::decay_t<decltype(s)>, Rectangle>)
            return s.width * s.height;
    }, shape);
}
```

Обяснения:

using T = std::decay_t<decltype(shape)>;

decltype взима типа на променливата shape, а std::decay_t премахва const, референции

Статичен полиморфизъм

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
}
```

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 на C#

```csharp
using System;

class Shape
{
    public virtual double Area()
    {
        return 0; // базова имплементация
    }
}

class Circle : Shape
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area()
    {
        return Math.PI * Radius * Radius;
    }
}

class Rectangle : Shape
{
    public double Width { get; }
    public double Height { get; }

    public Rectangle(double width, double height)
    {
        Width = width;
        Height = height;
    }

    public override double Area()
    {
        return Width * Height;
    }
}

class Program
{
    static void PrintArea(Shape shape)
    {
        Console.WriteLine($"Area: {shape.Area()}");
    }

    static void Main()
    {
        Shape s1 = new Circle(5);
        Shape s2 = new Rectangle(4, 3);

        PrintArea(s1);
        PrintArea(s2);
    }
}
```

Въпроси?

Благодаря за вниманието!