Абстрактни класове и интерфейси.

Абстрактни класове и интерфейси. Pure virtual functions, type casts. LID от SOLID. Type Erasure (std::function C++ 11). Сравнение с Java/C#.

д-р Тодор Цонков
todort@uni-sofia.bg

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

При полиморфни класове и триене 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() { /**/ } };

Base* p = new Derived;
delete p; // безопасно, извиква
Derived::~Derived()
```

Какво е интерфейс?

В C++ няма ключова дума interface.

Интерфейсът е дизайнерска концепция, която се реализира чрез клас, който съдържа само pure virtual функции без състояние (data members) с виртуален деструктор. Трябва всички методи да са публични.

Целта е да дефинира какво може обектът, а не как е реализиран.

Следва Dependency Inversion Principle (D от SOLID).

```
#include <print>
struct Shape{
    virtual double area() const = 0;
    virtual ~Shape() = default;
};

struct Rectangle : Shape {
double width, height;
Rectangle(double w, double h)
: width(w), height(h) {}
double area() const override
{
    return width * height;
}
};
```

Какво е абстрактен клас?

Абстрактен клас:

1)Не може да се инстанцира
2)Служи като интерфейс + базова логика
3)Има поне една pure virtual функция

Гарантира, че всички наследници: Имплементират нужните функции и имат един и същ публичен достъп.

Може да има и член данни за разлика от интерфейса!

```
class Person {
protected:
    std::string name_;
public:
    Person(std::string name) :
name_(std::move(name)) {}
    virtual void print() const {
        std::cout << "Name: " << name_ << "\n";
    }
    virtual std::string role() const = 0;
};
```

Добри практики

Интерфейс се прави когато има реална нужда от runtime полиморфизъм и типовете се сменят по време на изпълнение. Множествено наследяване почти винаги трябва да е с интерфейси, а не класове със стейт!

Не трябва да се започва с интерфейс, а с конкретен клас и след това да се въведат интерфейси при нужда. В много случаи се препоръчва std::function която ще бъде разгледана в настоящата лекция.

```
struct Vehicle {
    virtual void start() = 0;
    virtual void stop() = 0;
    virtual void fly() = 0;
    virtual void sail() = 0;
};
```
- Лош пример.

Правилно:

```
struct Drivable {
    virtual void drive() = 0;
};
struct Flyable {
    virtual void fly() = 0;
};
```

type casts в C++

Type cast = изрично преобразуване на тип.В C++ има 4 различни оператора, всеки със строго определено предназначение. Различните cast-ове имат различна степен на безопасност.

Student* s = (Student*)p; //

1) може да означава static_cast, reinterpret_cast или const_cast, но не е ясно от кода, че се транслира до него

2)няма яснота какво реално прави

3)трудно се чете

static_cast е оператор за явно преобразуване на типове, което се проверява на compile time.
Показва, че програмистът е сигурен какво прави.
Синтаксис:

static_cast<NewType>(expression)

double d = 3.14;

int i = static_cast<int>(d);

✔ проверява се по време на компилация

✔ без runtime overhead

❌ не проверява реалния тип на обекта

dynamic_cast

Проблем:

```
Person* p = new Student{};
Student*s=static_cast<Student*>(p);
```

Опасно, ако p не сочи към Student

```
Person* p = new Student{};
Student* s =
dynamic_cast<Student*>(p);
if (s) {
   // безопасно
}
```

проверява реалния тип
връща nullptr при грешка, но има runtime overhead.

Класът трябва да е polymorphic (да има поне една virtual функция), в противен случай няма да работи:

```
class Person {
public:
   virtual ~Person() = default;
};
```

При често ползване на dynamic_cast, това обикновено означава: Дизайнът не използва добре виртуални функции.
Нарушава се LSP (L от SOLID).
Липсва правилен полиморфизъм

const_cast, reinterpret_cast

const_cast е единственият cast в C++, който:
може да премахва (const) и (volatile) квалификатори
2)не променя типа, само cv-квалификациите
const int x = 10;
int& y = const_cast<int&>(x);
3) Това не означава, че е безопасно да се модифицира x.

Какво МОЖЕ да прави const_cast
✔ маха const / volatile
✔ добавя const (рядко полезно)
✔ работи с указатели и референции

reinterpret_cast е най-ниско-нивовият и най-опасният cast в C++.Казва на компилатора:
„Интерпретирай тези битове по друг начин.“
int* p = reinterpret_cast<int*>(0x12345678);
Какво може да прави:
✔ преобразува между несвързани типове указатели
✔ pointer ↔ integer
✔ function pointer ↔ function pointer
Ако мислите, че се нуждаете от reinterpret_cast,
99% от времето дизайнът е грешен.

Liskov Substitution Principle (L in SOLID)

Определение:
Обект от наследен клас трябва напълно да замества базовия.
Пример - Проблемът със Square / Rectangle:
Rectangle обещава: ширина и височина са независими
Square нарушава това обещание
❗ LSP е за поведение, не за синтаксис

Практически сигнал за проблем:
1)if (dynamic_cast…)
2)проверки за тип
3)изключения в override

```
class Shape {
public:
    virtual int area() const = 0;
    virtual ~Shape() = default;
};
class Rectangle : public Shape
class Square : public Shape,
```

а НЕ class Square : public Rectangle

👉 Ако Derived наследява Base, то навсякъде, където използваме Base, трябва да можем да сложим Derived без да чупим логиката.LSP е семантичен принцип, не езиков.

Interface Segregation Principle (I in SOLID)

Какво казва принципът?

“Клиентите не трябва да бъдат принуждавани да използват интерфейси, които не им трябват.”

С други думи:

По-добре няколко малки, специфични интерфейса, отколкото един голям, общ интерфейс.

Всеки клас трябва да имплементира само методите, които използва, а не да наследява

- Защо е важно?

Намалява нежеланата зависимост
Ако клас трябва да имплементира метод, който не ползва → това е тясна връзка (tight coupling).

Улеснява разширяемостта
Можеш да добавяш нови интерфейси, без да засягаш всички класове.

Подобрява четимост и поддръжка
Малки интерфейси са по-лесни за разбиране и тестване.

Interface Segregation Principle пример

```
struct Worker
{
   virtual void work() = 0;
   virtual void eat() = 0; // не всеки работник яде в офиса
};

struct Robot : Worker
{
   void work() override { /* работа */ }
   void eat() override { /* ??? */ } // проблем
};
```

Решение:

```
struct Workable
{
  virtual void work() = 0;
};
struct Eatable
{
  virtual void eat() = 0;
};
struct Human : Workable, Eatable
{
  void work() override { /* работа */ }
  void eat() override { /* ядене */ }
};

struct Robot : Workable
{
  void work() override { /* работа */ }
};
```

Interface Segregation Principle добри практики

❌ Проблеми:
1)интерфейси с много методи
2)override-и с празно тяло
3)трябва да пишем:
throw NotImplemented
4)класът „знае твърде много“
✅ Добра практика:
интерфейси с ясна роля, често имена завършващи на -able

По-добре много малки интерфейси, които всеки клиент разбира и използва, отколкото един огромен интерфейс, който всички ползват

Да се ползва за да избегнем "нежелани зависимости" и ненужен код

Ако даден клас зависи от интерфейс, но реално не използва всички методи, при промяна на тези методи може да се наложи ненужно да модифицирате класа.

Да се ползва за по-лесно тестване и mock-обекти
Малките интерфейси са по-лесни за unit тестове, защото не трябва да се имплементират излишни методи.

Dependency Inversion Principle (D in SOLID)

📌 Какво казва принципът?

“Високо ниво модули не трябва да зависят от ниско ниво модули. И двата трябва да зависят от абстракции (интерфейси). Абстракциите не трябва да зависят от детайли. Детайлите трябва да зависят от абстракции.”

С други думи - вместо класовете да се зависими директно един от друг (конкретни реализации), зависят от

Това прави кода по-гъвкав и лесен за разширяване.

```
struct FileLogger
{
   void log(const std::string& msg) { /* пише в файл */ }
};

struct UserService
{
   FileLogger logger; // зависимост от конкретен клас

   void addUser(const std::string& name)
   {
      // ...
      logger.log("User added");
   }
};
```

Dependency Inversion Principle (D in SOLID)

Решение:

```
struct ILogger{

  virtual void log(const std::string& msg) = 0;
  virtual ~ILogger() = default;
};

struct FileLogger : ILogger{

   void log(const std::string& msg) override { /*
пише в файл */ }

};

struct ConsoleLogger : ILogger{

  void log(const std::string& msg) override { /*
пише на конзолата */ }

};
```

Високите нива (business logic) не трябва да знаят за детайли (low-level modules).

И двата слоя зависят от абстракции.

Конкретните детайли имплементират интерфейса, не обратното.

Принцип:

“Нека модулите за бизнес логика да говорят само с интерфейси, а не с конкретни реализации. Детайлите трябва да се адаптират към интерфейсите, а не интерфейсите към детайлите.”

Type Erasure

Type Erasure е техника, при която:

Скриваме конкретния тип на обект

Работим с него чрез унифициран интерфейс

👉 Идея: „Не ме интересува какъв е типът, важно е какво може да прави“

Цели:

1)Полиморфизъм без наследяване

2)Runtime гъвкавост

```
template<typename F>
void run(F f) {
    f();
}
```

❌ Всеки различен F → нова инстанция

❌ Няма общ тип за съхранение

❌ Трудно предаване между модули

👉 Не можем да кажем:

```
std::vector<??> tasks;
```

std::function пример за Type Erasure

std::function:

Приема всякакъв callable обект:

функция, lambda, functor, std::bind

obj(args...); => държи се като функция

Скрива реалния тип → type erasure

Осигурява един общ тип

Какво е std::bind -> стандартна функция (в <functional>) в C++, която създава нов callable обект, като фиксира някои от аргументите на друга функция или пренарежда тяхната последователност.

Позволява частично прилагане и пренасочване на аргументи към функция.

В C++ функтор (functor) е обект, който може да се използва като функция, защото дефинира оператор operator().

С други думи:

“Функторът е обект, който може да се извика като функция.”

lambda функции в езика C++

Lambda (анонимна функция) е функционален обект, който може да се дефинира "на място", без да се създава отделен именован функция или клас.

В C++ се появяват от C++11.

Синтаксисът е:

[capture](parameters) -> return_type { body }

parameters – аргументи на lambda, като при нормална функция

return_type – (по избор) тип на връщаната стойност. body – тялото на lambda, което се изпълнява

```
std::vector<int> v{1, 2, 3, 4, 5};

// Lambda, която принтира числата

std::for_each(v.begin(), v.end(), [](int x) {
std::print("{} ", x); });

int a = 10;
int b = 5;
auto sum = [a, &b](int x) { return x + a + b; };
b = 7;
std::print("{}\n", sum(3)); // 3 + 10 + 7 = 20
```

std::function - примери

```
std::function<int(int,int)> add = [](int a,
int b) { return a + b; };

std::print("{}\n", add(3, 4)); // 7

int multiply(int a, int b) { return a * b; }

std::function<int(int,int)> op = multiply;

struct Divider {

    double operator()(double a, double
b) { return a / b; }

};

std::function<double(double,double)>;
```

```
std::vector<int> v{1,2,3,4,5};

std::function<void(int)> printSquare =
[](int x) { std::print("{} ", x*x); };

std::for_each(v.begin(), v.end(),
printSquare); // 1 4 9 16 25
```

Всички тези функции може да се пълнят във вектор стига да са от един тип:

```
// Създаваме vector от std::function,
които приемат int и връщат int

std::vector<std::function<int(int)>>
funcs;
```

Сравнение с Java/C#

В C++, абстрактният клас е клас, който съдържа поне една чисто виртуална функция (virtual func() = 0;) и може да има конкретни методи и полета. Интерфейс в C++ обикновено се реализира чрез клас с всички методи чисто виртуални и без данни, т.е. чисто абстрактен клас.

В Java и C#, абстрактният клас може да съдържа както реализирани, така и абстрактни методи, докато интерфейсът първоначално дефинира само

Съвременните версии позволяват default реализации.

Основните разлики са: абстрактният клас позволява наследяване на имплементация и полета, а интерфейсът дефинира само контракт и поддържа множествено „наследяване“. В C++ интерфейсите са просто чисто виртуални класове, тъй като езикът няма отделна концепция за интерфейс за разлика от C# и Java, които имат.

Пример с класа Student

```cpp
struct StudentWrapper {
    std::string studentName;
    std::function<void()> studyFunc; // type-erased функция

    StudentWrapper(const std::string& name, std::function<void()> func)
        : studentName(name), studyFunc(func) {}

    void study() { studyFunc(); }
};

int main() {
    Student alice("Alice");
    Student bob("Bob");

    // Type erasure: съхраняваме различни студенти чрез std::function
    StudentWrapper s1(alice.name(), [&alice](){ alice.study(); });
    StudentWrapper s2(bob.name(), [&bob](){ bob.study(); });

    s1.study(); // Alice is studying hard!
    s2.study(); // Bob is studying hard!
}
```

Въпроси?

Благодаря за вниманието!