Основи на модерния дизайн на клас.

Принципи на дизайна. S от SOLID принципите. Разделна компилация. default/delete на специални функции (C++ 11). Статични член данни.

д-р Тодор Цонков
todort@uni-sofia.bg

Инварианти на клас в детайли

Какво НЕ е инвариант?
❌ Локални променливи
❌ Временни състояния по време на функция
❌ Условия, които важат само за конкретен метод
Инвариантите важат за целия живот на обекта и се проверяват след конструкторите – всеки обект трябва да започне в валидно състояние.
и преди и след публични методи – за да се гарантира, че обектът остава валиден.

Пример за нарушени инварианти:

```
class Buffer {
public:
    char* data;
    std::size_t size;
    Buffer(std::size_t s) {
        size = s;
        data = new char[s];
    }
    ~Buffer() {
        delete[] data;
    }
};
```

Никъде не се проверява, че size>0, дали data е валиден и дали паметта е валидна.

Какво е интерфейс на клас?

Интерфейсът на един клас е:
Всичко, което е публично достъпно отвън. Тоест: public методи, типове, константи, конструктори.

Принципи:

1)Да се излага възможно най-малко -
Колкото повече public методи има:
= толкова повече начини за неправилна употреба, зависимости, по-труден refactoring

2)Методите трябва да казват какво, не как.
void update(double x); Какво?
По-добре void deposit(Money x);

3) Малки и ясни методи:
void process();
или void validate();
void calculateInterest();
void generateStatement();

Интерфейсът трябва да показва:

Кой притежава обекта?
Кой го унищожава?
Може ли да е null?

SOLID принципи - SRP

SOLID е набор от 5 принципа за обектно-ориентиран дизайн, целящи: по-лесна поддръжка, разширяемост, по-малко грешки, по-добра архитектура.

Принципи:

S – Single Responsibility

O – Open / Closed

L – Liskov Substitution

I – Interface Segregation

D – Dependency Inversion

Single Responsibility principle

Един клас → една отговорност → една причина за промяна

❌ Лош пример:

```
class Student {
public:
    void calculateGrade();
    void saveToFile();
};
```

✅ Добър:

```
class Student { void calculateGrade();
};
class StudentRepository { void
save(const Student&);
```

Разделна компилация

Какво е разделна компилация?

.h → декларации

.cpp → имплементация

Предимства:

по-бърза компилация

по-чист дизайн

независими модули

//В C++ 20 се въвеждат модулите

// за да решат проблеми - като

multiple includes in headers и други

```
//Student. h
#pragma once
class Student {
    std::string name;
public:
    Student(const std::string& n);
    std::string getName() const;
};
//Student.cpp
#include "Student.h"
Student::Student(const std::string&
n) : name(n) {}

std::string Student::getName() const {
    return name;
}
```

Разделна компилация добри практики

```
// Student.h
class Student {
public:
    void print() const;
};
// Student.cpp
   void Student::print() {   // ❌ липсва
const
}
```

✔ .h = интерфейс

✔ .cpp = имплементация

✔ Минимални include-и

✔ Forward declarations

✔ Никакви .cpp includes

✔ Никакъв using namespace в header

```
// A.h
#include "B.h"
class B; // forward declaration
class A {
    B b;
};
// B.h
#include "A.h"
class B {
    A a;
};
```

Какво са модулите в езика C++?

Модулите са нов механизъм за организация на кода, въведен в C++20. Целта им е да заместят класическите header файлове и #include директивите.

Основни предимства на модулите:

1)По-бързо компилиране (намаляват повторното обработване на хедъри).

2)Ясна изолация на интерфейси и имплементации.

3)Намаляване на проблеми с multiple inclusion (#pragma once или

```
export module MathLib;  //име
export class Calculator {
public:
  Calculator() = default;
  // Методи
  int add(int a, int b) const;
  int multiply(int a, int b) const;
};
module MathLib; //
имплементационна единица на
модула
```

Модули - продължение

```
module MathLib; //
имплементационна единица на
модула

int Calculator::add(int a, int b) const {
  return a + b;
}

int Calculator::multiply(int a, int b)
const {
  return a * b;
}
```

```
import MathLib;  // import
замества //#include

#include <iostream>
int main() {
  Calculator calc;
  std::cout << "3 + 4 = " << calc.add(3,
4) << "\n";
  std::cout << "3 * 4 = " <<
calc.multiply(3, 4) << "\n";
}
```

Какво е препроцесор?

Работи преди компилация

Основни директиви:

#include

#define

#ifdef / #ifndef

Условна компилация:

```
#ifdef DEBUG
std::cout << "Debug mode\n";
#endif
// полезна за дебъгване
```

```
#ifndef STUDENT_H
#define STUDENT_H
// код
#endif
};
```

```
#define PI 3.14       //лоша практика
constexpr double PI = 3.14; // добра практика - ще го разгледаме по-късно в лекцията
```

Макросите се считат за лоша практика, защото нямат типова проверка, енкапсулация, не могат да се дебъгват

Разделна компилация

Една C++ програма може да бъде разделена на множество изходни файлове (.cpp), които се компилират независимо един от друг — това се нарича разделна компилация.

Преди същинската компилация всеки изходен файл се обработва от препроцесора, който изпълнява всички директиви, започващи със символа #.

Например, при всяка среща на директивата #include препроцесорът я заменя със съдържанието на съответния хедър файл, който обикновено съдържа декларации.
В резултат на компилацията на всеки .cpp файл се получава отделен обектен файл (с разширение .obj), съдържащ машинен код.

Разделна компилация

Изпълнимият файл на програмата (с разширение .exe под Windows) се създава на следващ етап — свързване (linking) — при което линкерът обединява всички обектни файлове.

Линкерът свързва всички референции към имена (променливи, функции, класове и др.) от даден обектен файл със съответните им дефиниции, които могат да се намират в други обектни файлове.

Възможно е дадена дефиниция да липсва във всички обектни файлове. В такъв случай линкерът я търси в стандартната C++ библиотека, стандартната C библиотека, както и във всички допълнително указани от програмиста библиотеки. Ако дефиницията не бъде открита никъде, линкерът генерира грешка.

Какво е CMake?

CMake е инструмент, който генерира build файлове.
(Makefile, Visual Studio, Ninja и др.)
от билд конфигурация.

❓ Защо не компилираме директно с g++ или clang?

1)проектът има много .cpp файлове

2)различни платформи (Linux / Windows / macOS)

3)различни компилатори

CMake е “Правилният” начин да билднете проект.

➡️ Описваме проекта веднъж (CMakeLists.txt)

➡️ CMake генерира подходящ build

➡️ компилаторът върши останалото

Примерна структура:

```
project/
├── CMakeLists.txt
├── main.cpp
├── Student.h
└── Student.cpp
```

Сравнение между CMake/Maven/MSBuild

CMake – генератор на build системи за C/C++/Objective-C, ръчно управление на зависимости чрез find_package или FetchContent, конфигурация чрез скриптове CMakeLists.txt, отлична cross-platform поддръжка, интеграция с IDE като Visual Studio, Xcode, Make/Ninja.

Maven – build и project management за Java/JVM, автоматично управление на зависимости чрез Maven Central, декларативна конфигурация чрез pom.xml, кросплатформен (JVM), интеграция с IntelliJ, Eclipse, NetBeans.

MSBuild – native build system за C#/C++/.NET, управление на зависимости чрез NuGet, декларативни XML проекти (.csproj), основно Windows (с .NET Core – cross-platform), интеграция с Visual Studio и VS Code.

Какво е ключова дума =delete?

=delete е добра идея винаги, когато трябва изрично да забраним определена операция, защото тя няма смисъл, е опасна или нарушава дизайна на класа.

Пример:

Класът не трябва да се копира (owning ресурс) - най-честият и правилен случай или притежава уникален ресурс: файл, сокет, mutex, GPU ресурс и т.н.

Копиране би довело до double free / undefined behavior.

```
#include <cstdio>

class File {
    std::FILE* handle;

public:
    explicit File(const char* name)
        : handle(std::fopen(name, "r")) {}

    ~File() {
        if (handle)
            std::fclose(handle);
    }

    // ❌ Prevent copying
    File(const File&) = delete;
    File& operator=(const File&) = delete;
};
```

Още примери за =delete?

```
class Student {
public:
    Student(int id);
    Student(double) = delete;   //
забраняваме неясни конверсии
};
```

Пример за функции:

```
void log(int) = delete;
void log(double) = delete;


void log(const std::string& msg) {
    std::cout << msg;
}
```

Когато искаме класът да няма смисъл без дефолтни данни:

```
class Student {
    std::string name_;
public:
    Student() = delete;
    explicit Student(std::string name) :
name_(name) {}
};

Student s;          // ❌ compile error
Student s("Ivan");  // ✅ valid
```

По-този начин налагаме задължително условието (инвариантна) на класа, че Студент трябва да има име.

Ключова дума =default

```
struct A {
    A() = default;
}; //По този начин се генерира
```

стандартна имплементация на функцията.

default могат да бъдат всички специални функции на клас, които ще разгледаме по-нататък.

Пише се default, когато няма специална логика.

Също така се осигурява контрол върху достъпа по експлицитен начин.

```
class Student {
    std::string name_;
public:
    Student() = default;
    Student(const Student&) = default;          //
копиране
};
```

Когато създадем конструктор с параметри НЕ се генерира дефолтния конструктор.

Ползва се за
Яснота в дизайна, модерна C++ практика, възстановяване на автоматично генерирана функция или да се възстанови автоматично генерирана функция.

Статични член данни

Член-променлива, споделена между всички инстанции на класа. Декларира се в класа; дефинира се (и, при нужда, инициализира) извън него — с изключение на inline static (C++17+).

Статичните член-данни имат живот, различен от обектите: инициализират се преди main() или при първия използван translation unit (implementation-defined ordering).

```
#include <iostream>
class Math {
public:
    static int add(int a, int b) {
        return a + b;
    }
  static int count;
};

int Math::count = 0; // дефиниция извън класа

int main() {
    int result = Math::add(3, 4);  // извикване без
обект
    std::cout << result << std::endl;
}
```

Проблеми при статичните член данни

Редът на инициализация на статични обекти в различни translation units е неопределен. Това води до бъгове, когато един статичен обект използва друг статичен обект от друг .cpp файл.

```
extern int valueB;
int valueA = valueB + 1; // може да е неинициализирана
// B.cpp
int valueB = 42;
```

```
int& safeValue() {
    static int value = 42; // lazy + thread-safe
    return value;
}
//Scott Meyers - пример за Singleton
```

Практически съвети

Избягвайте глобални и статични обекти със сложна логика.

Предпочитайте constexpr, inline static или function-local static.

Ако редът има значение — контролирайте го чрез функции.

Ключова дума constexpr в езика C++

constexpr е ключова дума в C++, която казва, че нещо може да се изчисли по време на компилация, а не чак при изпълнение.

Когато я приложим към клас, това означава, че можем да създаваме обекти на класа, които са константи по време на компилация.

За да може класът да бъде constexpr, конструкторите му трябва да са constexpr, а всички член-функции, които искаме да използваме по време на компилация, също трябва да са constexpr.

Конструкторът и функциите не трябва да правят runtime операции, като: динамично разпределяне на памет (new) като в C++ 20 това вече е позволено.

Пример за constexpr в езика C++

```
class Point {
double x_;
double y_;
public:
// constexpr конструктор
constexpr Point(double x, double y)
: x_(x), y_(y) {}
// constexpr getter-и
constexpr double x() const { return x_; }
constexpr double y() const { return y_;
}
```

```
// constexpr метод за транслация
constexpr Point translate(double dx,
double dy) const {
    return Point(x_ + dx, y_ + dy);
}
constexpr Point p1(2.0, 3.0);
constexpr Point p2 = p1.translate(1.0,
-1.0);
```

consteval (C++20) функция е функция, която задължително се изчислява по време на компилация, а по време на runtime, компилаторът ще даде грешка.

```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_;
    }

    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_;
};
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 21 + chapter F