

# Основи на модерния дизайн на клас.

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

практически  
примери

теория

# От какво ще се състои настоящата лекция?

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

## Пример

### Какво са инварианти на клас?

Инварианти на клас са условия (правила), които винаги трябва да са верни за обекта, след като той е успешно конструиран и след всяко извикване на публичен метод. Тоес това са нещата, които класът обещава, че никога няма да се счупят.

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.

```
/*Invariants: 0 <= size_
size_ <= capacity_
data_.size() == capacity_ */

class Buffer {
public:
    explicit Buffer(size_t capacity)
        : capacity_(capacity), size_(0) {}

    void push(int x) {
        if (size_ == capacity_)
            throw std::runtime_error("Full");
        data_[size_++] = x;
    }

private:
    size_t size_;
    size_t capacity_;
    std::vector<int> data_;
};
```

## Какво са SOLID принципите?

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 компиляция

## Какво е разделна компиляция?

.h → декларации  
.cpp → имплементация

Предимства:  
по-бърза компиляция  
по-чист дизайн  
независими модули

## Student.h

```
#pragma once
#include <string>

class Student {
    std::string name;
public:
    Student(const std::string& n);
    std::string getName() const;
};
```

// В C++ 20 се въвеждат  
модулите  
// за да решат проблеми - като  
multiple includes in headers

## Student.cpp

```
#include "Student.h"
Student::Student(const std::string& n) :
name(n) {}

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

## Грешки при използване на header-и

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

// Student.cpp
void Student::print() { // ✗ липсва
const
}
```

## Добри практики

- ✓ .h = интерфейс
- ✓ .cpp = имплементация
- ✓ Минимални include-и
- ✓ Forward declarations
- ✓ Никакви .cpp includes
- ✓ Никакъв using namespace в header

-> Circular dependencies -> forward declaration

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

```
// B.h
#include "A.h"
```

```
class B {
    A a;
};
```

## 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
```

## Добри практики

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

```
#define PI 3.14          //лоша практика  
constexpr double PI = 3.14; // добра  
практика
```

# Обяснение на разделната 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.

Изпълнимият файл на програмата (с разширение .exe) 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 С библиотека, както и във всички допълнително указанi 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++?

1) проектът има много .cpp файлове

2) различни платформи (Linux / Windows / macOS)

3) различни компилатори

4) “Правилният” начин да билднете проект

## Основна идея

- Описваме проекта веднъж (CMakeLists.txt)
- CMake генерира подходящ build
- компилаторът върши останалото

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

## Пример

### Кога да ползваме `=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.

Класът притежава уникален ресурс: **файл**, **сокет**, **mutex**, **GPU ресурс**, **OS handle**

Копиране би довело до double free / undefined behavior.

```
● ● ●

class File {
    FILE* f_;
public:
    File(const char* name) : f_(fopen(name, "r"))
{} ~File() { if (f_) fclose(f_); }

    File(const File&) = delete;
    File& operator=(const File&) = delete;

    File(File&&) noexcept = default;
    File& operator=(File&&) noexcept = default;
};
```

# Още примери за =delete

## Примери

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

## Примери

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

когато искаме класът  
няма смисъл без  
дефолтни данни.

## Примери за функции

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

# Пример

## Кога да ползваме =default?

```
struct A {  
    A() = default;  
}; //По този начин се генерира стандартна  
имплементация на функцията.  
=default могат да бъдат:  
Default constructor, Destructor, Copy  
constructor, Copy assignment operator  
Move constructor, Move assignment operator,  
operator<=> (C++20)
```

Пише се `default`, когато няма специална логика.  
Също така се осигурява контрол върху достъпа по  
експлицитен начин.

```
struct NonCopyable {  
    NonCopyable() = default;  
  
    NonCopyable(const NonCopyable&) = delete;  
    NonCopyable& operator=(const NonCopyable&) = delete;  
};
```

```
struct B {  
    int x;  
    B() = default;  
};  
//Ако не е имплементиран  
//Може и да не е default
```

## Какво са 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`).



```
class MyClass {  
public:  
    static int s_count; // декларация  
    MyClass();  
};  
  
// source MyClass.cpp  
int MyClass::s_count = 0; // дефиниция + инициализация  
  
struct Widget {  
    inline static int s_count = 0;  
    // C++17+: дефиниция в header  
    static constexpr int MAX = 256;  
    // константа известна на компилатора  
};  
  
//constexpr - смята се по време на компилация
```

## Проблеми при статичните 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;
```

## Решение

```
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.

## constexpr ( тема за напреднали)

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 разпределение на памет (new) – до C++20 хвърляне на изключения, виртуални функции.

```
#include <iostream>

class Point {
    int x;
    int y;
public:
    // constexpr конструктор
    constexpr Point(int x_val, int y_val) : x(x_val), y(y_val) {}

    // constexpr функции
    constexpr int getX() const { return x; }
    constexpr int getY() const { return y; }

    // обикновена функция (runtime)
    void print() const { std::cout << "(" << x << ", " << y << ")\n"; }
};

int main() {
    constexpr Point p1(3, 4); // обект изчислен по време на компилация

    constexpr int x = p1.getX(); // compile-time
    constexpr int y = p1.getY(); // compile-time

    std::cout << x << ", " << y << "\n"; // runtime print
}
```

# Пример

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

namespace School {

enum class GradeLevel { Freshman, Sophomore, Junior, Senior };

class Student {
private:
    std::string name;
    int age;
    GradeLevel grade;

    static int studentCount; // брояч на студентите

public:
    // Конструктори
    Student() = default; // default конструктор
    Student(const std::string& n, int a, GradeLevel g);

    // Забраняваме конструктор с 1 параметър (пример за delete)
    Student(int a) = delete;

    ~Student();

    void print() const;

    static int getStudentCount() { return studentCount; }
};

} //
```



# Въпроси?

Благодаря за вниманието!