

## ЗАДАЧИ ЗА ООП ПРАКТИКУМ, 22 МАРТ

---

ЧЛЕН-ФУНКЦИИ. ЖИЗНЕН ЦИКЪЛ НА ОБЕКТ. КОНСТРУКТОРИ И ДЕКТРУКТОР.  
КАПСУЛАЦИЯ. МОДИФИКАТОРИ ЗА ДОСТЪП. ЧЛЕН-ФУНКЦИИ GET И SET.

### Задача първа:

Напишете клас *teacher*, който описва преподавател. Класът трябва да съдържа:

1. Име
2. Възраст
3. Години стаж като преподавател

Създайте подходящи конструктори. Създайте *get* и *set* член функции. Какво още трябва да добавите?

### Задача втора:

Явор, подготвяйки задачите за контролното по ООП, реши, че е добра идея да внимава точно как пише задачите. Всеки можеше да отвори лаптопа му и да открадне тази безценна информация.

Помогнете на Явор да опази задачите от изпита. Създайте клас *secret*. Класът трябва да съдържа:

1. Условие на задачата
2. Парола
3. Брой неуспешни влизания

Направете подходящи конструктори. Да се напише функция *getTask(const char\* pwd)*; която връща условието тогава и само тогава когато сме въвели правилната парола. Да се напише функция *setPassword(const char\* newPassword, const char\* oldPassword)*; която променя паролата, но само ако сме въвели правилно старата парола. Да се напише функция *setTask(const char\* newTask, const char\* passwd)*; която променя условието на задачата ако сме въвели правилна парола. Всеки път когато сме въвели неправилна парола броят на неуспешните влизания се увеличава. Направете функция *getLoginFails()*, която връща броя на неуспешните влизания. Трябва ли да има функция *setLoginFails()*? Какво още трябва да добавим?

### ДОПЪЛНИТЕЛНА ИНФОРМАЦИЯ

UNIT TESTING. ИЗПОЛЗВАНЕ НА DOCTEST. ENUM.

Unit testing е тестване на някаква малка част от кода, която не може смислено да се разбие на по-малки парчета функционалност. Unit тестовете (написани коректно) гарантират, че кодът работи както очакваме. Също така подсигуряват, че кодът ще

продължи да работи както трябва и след добавяне на функционалност, модифициране или разширяване на функционалността.

За тестване ще използваме *doctest* - <https://github.com/doctest/doctest>.

Това което ще ни трябва е - <https://github.com/doctest/doctest/blob/master/doctest/doctest.h>

Цялата логика на *doctest* е събрана в един *.h* файл, което го прави лесен за използване. След като се сдобие с *doctest.h* файла, поставете го в папката, в която се намира проекта Ви. След това нещата са почти готови. За да използвате *doctest* трябва да му кажете с коя точно конфигурация ще работите. Аз предпочитам *#define DOCTEST\_CONFIG\_IMPLEMENT*. Пример за използване на *doctest*:

```
#include <iostream>
#define DOCTEST_CONFIG_IMPLEMENT //define преди include!!!!
#include "doctest.h"

int id(int x) {
    return x;
}

size_t fastPowWrapper(size_t num, size_t power) {
    if (power == 0)
        return 1;
    if (power % 2 == 1)
        return num * fastPowWrapper(num, power - 1);
    return fastPowWrapper(num * num, power / 2);
}

size_t fastPow(size_t num, size_t power) {
    if (!num && !power)
        throw std::exception("undefined");
    return fastPowWrapper(num, power);
}

TEST_CASE("ID-FUNC-TEST") {
    CHECK_EQ(id(5), 5);
}

TEST_CASE("ID-FUNC-RANDOMVAL-TEST") {
    int checkRandomVal = rand();
    CHECK(id(checkRandomVal) == checkRandomVal);
}

TEST_CASE("FAST-POW-TEST") {
    CHECK(fastPow(2, 8) == 256);

    size_t randomNum = rand() % 100 + 1;
    size_t randomPow = rand() % 100 + 1;
```

```

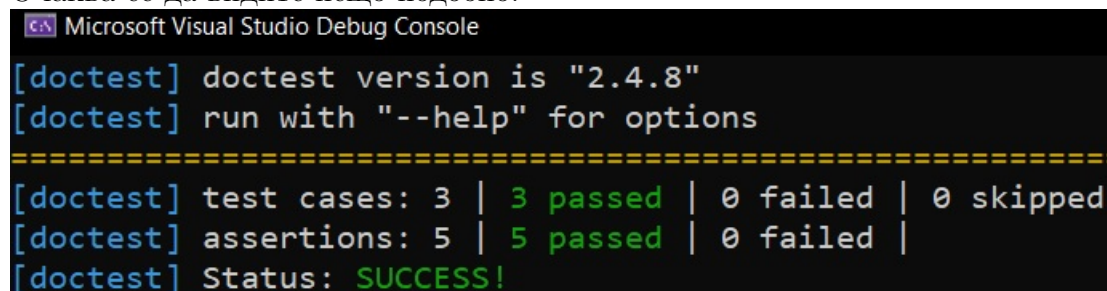
    int result = 1;
    for (int i = 0; i < randomPow; i++)
        result *= randomNum;

    CHECK(fastPow(randomNum, randomPow) == result);
    CHECK_THROWS(fastPow(0, 0));
}

int main() {
    doctest::Context c;
    c.run();
}

```

Очаква се да видите нещо подобно:



```

Microsoft Visual Studio Debug Console

[doctest] doctest version is "2.4.8"
[doctest] run with "--help" for options
=====
[doctest] test cases: 3 | 3 passed | 0 failed | 0 skipped
[doctest] assertions: 5 | 5 passed | 0 failed |
[doctest] Status: SUCCESS!

```

## ENUM

Преди няколко упражнения имахме да правим структура, описваща студент. В нея трябваше да се запише кой курс е студента. За целта използвахме число от тип `int`. Което работи, но носи и значителни проблеми. Какво ще стане ако някой ни подаде 55? Какво ще стане ако след една година отворим кода, и забравим какво точно значи 1 и 3, подадени като аргументи на функцията?

Когато имаме краен брой опции, нещата много лесно се описват чрез `enum` типа. Нека разгледаме следния пример:

```

enum class dayOfWeek{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}; // има точка и запетайка!

```

Този `enum` описва (крайното множество от) дните от седмицата. Та много по - удобно, красиво и разбираемо ще е да напишете

```
func(dayOfWeek : Monday);
```

пред алтернативата:

```
func(0);
```

Какво е Monday обаче? Реално enum class използва логиката с числата. Monday е 0. Следвано от Tuesday което е 1 и тн. Тези стойности може да се променят ако желаете, но са така по подразбиране. Та какво е предимството? Първо, както видяхме от примера по - горе, кодът става значително по - разбираем. Второто предимство е, че момента в който опитам да направя *func(13)*;, компилатора ще ми се скара. Ето един добър пример за проста имплементация на enum class:

```
#include <iostream>
```

```
enum class course {  
    First,  
    Second,  
    Third,  
    Fourth  
};
```

```
class Student {  
private:  
    int age;  
    course currentCourse;  
public:  
    Student() : age(20), currentCourse(course::First) {}  
    Student(int argAge, course argCourse) : age(argAge), currentCourse(argCourse)  
  
    void setCourse(course toSet) {  
        currentCourse = toSet;  
    }  
  
    bool gotAllExams() const {  
        return true;  
    }  
};
```

```
int main() {  
    Student myStudent(22, course::Second);  
  
    if (myStudent.gotAllExams())  
        myStudent.setCourse(course::Third);  
}
```

# 1 References

1. Unit testing:
  - <https://www.youtube.com/watch?v=7gD7lNVQUjU>