

Лекция 2.2. Модульное тестирование

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- 1 Зачем нужно модульное тестирование?
- 2 Создание основного проекта и тестируемых функций.
- 3 Компиляция объектного файла.
- 4 Создание проекта модульного теста.
- 5 Настройка связей между проектами.
- 6 Написание тестов.
- 7 Правила оформления тестов.
- 8 Запуск тестов.

Зачем нужно модульное тестирование?

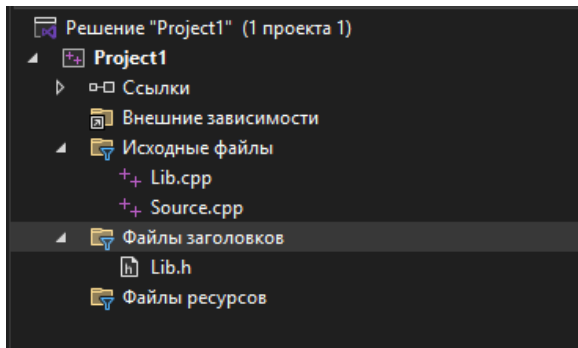
- **Автоматизация проверки** — тесты выполняются без участия человека.
- **Регрессионное тестирование** — после изменений в коде можно быстро проверить, что ничего не сломалось.
- **Документация** — тесты показывают, как должны работать функции.
- **Уверенность в коде** — можно смело рефакторить, зная, что тесты укажут на ошибки.

Важно

Модульные тесты проверяют отдельные компоненты программы (функции, классы) изолированно.

Шаг 1. Создание основного проекта

- Создаём проект с именем, например, Project1.
- В проекте создаем три файла:
 - 1 Source.cpp — точка входа (функция main).
 - 2 Lib.h — заголовочный файл с прототипами функций.
 - 3 Lib.cpp — реализация функций.



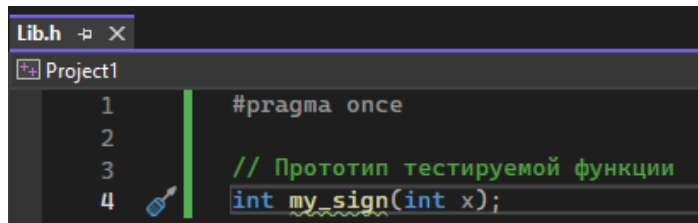
Заголовочный файл (объявления)

```
#pragma once

// Прототип тестируемой функции
int my_sign(int x);
```

- `#pragma once` — защита от повторного включения.
- Здесь объявляем функцию, которую будем тестировать.
- В нашем примере — функция `my_sign`, возвращающая:
 - 1 для положительных чисел,
 - 0 для нуля,
 - -1 для отрицательных.

Шаг 1. Lib.h



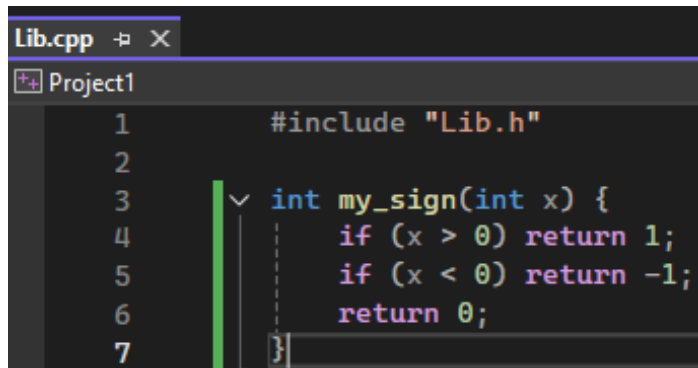
```
Lib.h  [icon] X
Project1
1      #pragma once
2
3      // Прототип тестируемой функции
4      int my_sign(int x);
```

Файл реализации

```
#include "Lib.h"

int my_sign(int x) {
    if (x > 0) return 1;
    if (x < 0) return -1;
    return 0;
}
```

- Подключаем свой заголовочный файл "Lib.h".
- Определяем функцию my_sign.
- Это простая функция, которую мы будем тестировать.



```
Lib.cpp [icon] X
Project1
1      #include "Lib.h"
2
3      int my_sign(int x) {
4          if (x > 0) return 1;
5          if (x < 0) return -1;
6          return 0;
7      }
```

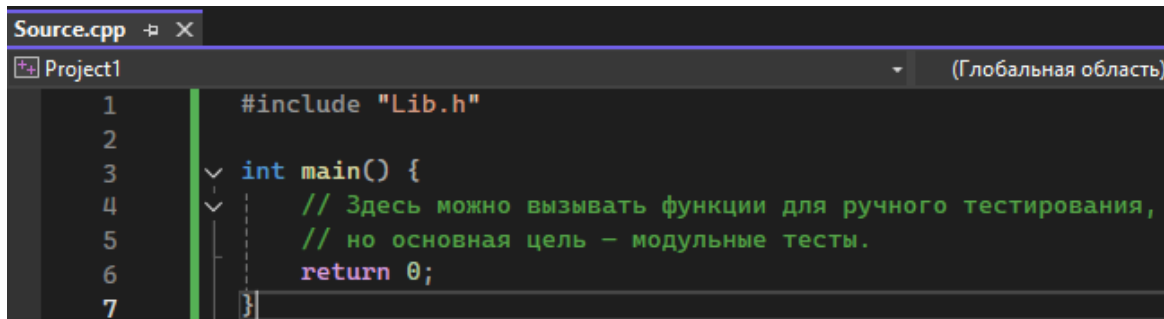
Точка входа (main)

```
#include "Lib.h"

int main() {
    // Здесь можно вызывать функции для ручного тестирования,
    // но основная цель - модульные тесты.
    return 0;
}
```

- Обычно в `main` пишут ручную проверку.
- Но мы будем использовать автоматические тесты.
- Поэтому `main` может быть пустым.

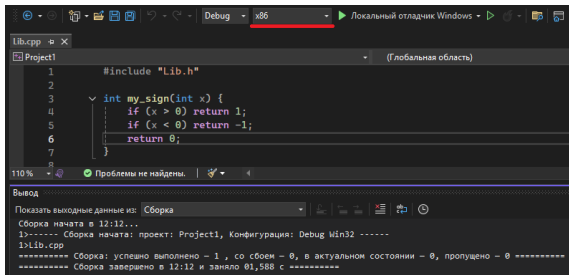
Шаг 1. Source.cpp



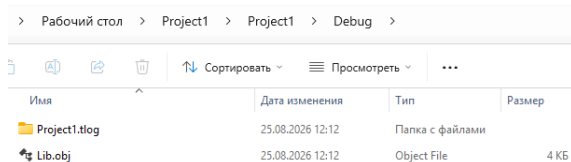
```
Source.cpp  + X
Project1  (Глобальная область)
1      #include "Lib.h"
2
3      ∨ int main() {
4      ∨      // Здесь можно вызывать функции для ручного тестирования,
5      |      // но основная цель – модульные тесты.
6      |      return 0;
7      |  }
```

Шаг 2. Компиляция Lib.cpp

- Установите вверху по центру платформу решения x86.
- Нажмите Ctrl+F7 при открытом файле Lib.cpp.
- При успешной компиляции в папке проекта появляется файл Lib.obj.
- **Запомните путь!** (обычно Project1\Debug или Project1\Project1\Debug).



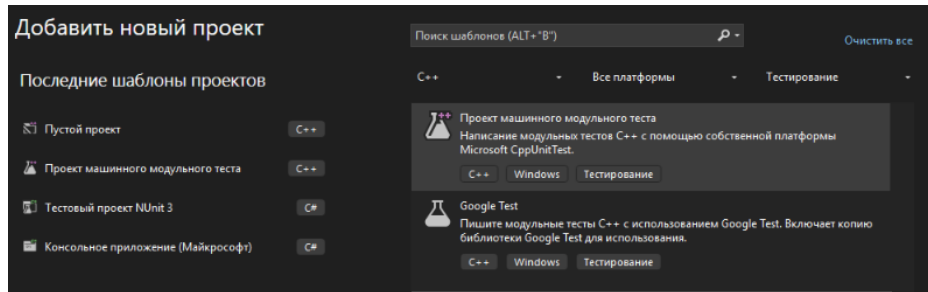
```
Lib.cpp
Project1
1  #include "Lib.h"
2
3  int my_sign(int x) {
4      if (x > 0) return 1;
5      if (x < 0) return -1;
6      return 0;
7  }
110 %
Проблемы не найдены.
Вывод
Показать выходные данные из: Сборка
Сборка начата в 12:12...
1>----- Сборка начата: проект: Project1, Конфигурация: Debug Win32 -----
1>Lib.cpp
----- Сборка: успешно выполнено - 1, со сбоем - 0, в актуальном состоянии - 0, пропущено - 0 -----
----- Сборка завершено в 12:12 и заняло 01,588 с -----
```



> Рабочий стол > Project1 > Project1 > Debug >			
Сортировать Просмотреть ...			
Имя	Дата изменения	Тип	Размер
Project1.tlog	25.08.2026 12:12	Папка с файлами	
Lib.obj	25.08.2026 12:12	Object File	4 КБ

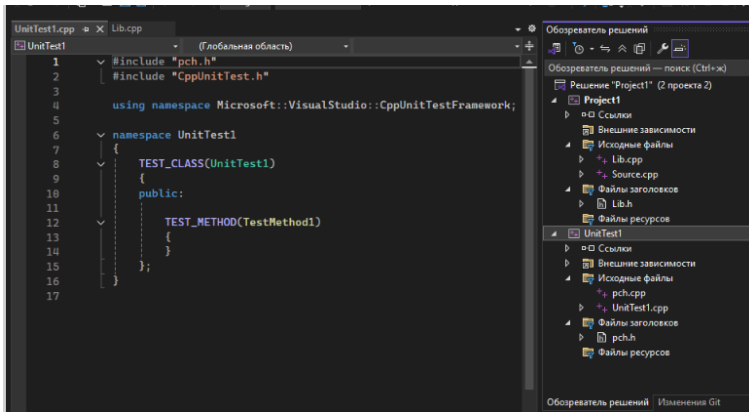
Шаг 3. Создание проекта модульного теста

- 1 ПКМ по решению в обозревателе → Добавить → Создать проект.
- 2 В правом верхнем углу выбрать "Тестирование".
- 3 Выбрать "Проект машинного модульного теста".
- 4 Ввести имя, например, `UnitTest1`, и нажать "Создать".



Шаг 4. Общий вид решения

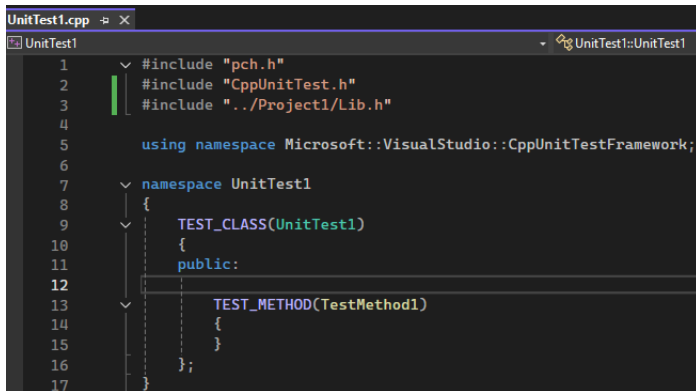
- В обозревателе решений появились два проекта:
 - 1 Project1 (основной)
 - 2 UnitTest1 (тестовый)
- Каждый проект содержит свои файлы.



Шаг 5. Подключение Lib.h в тестовом проекте

- В основном файле тестового проекта (обычно UnitTest1.cpp) нужно подключить заголовочный файл основного проекта.
- Используем относительный путь:

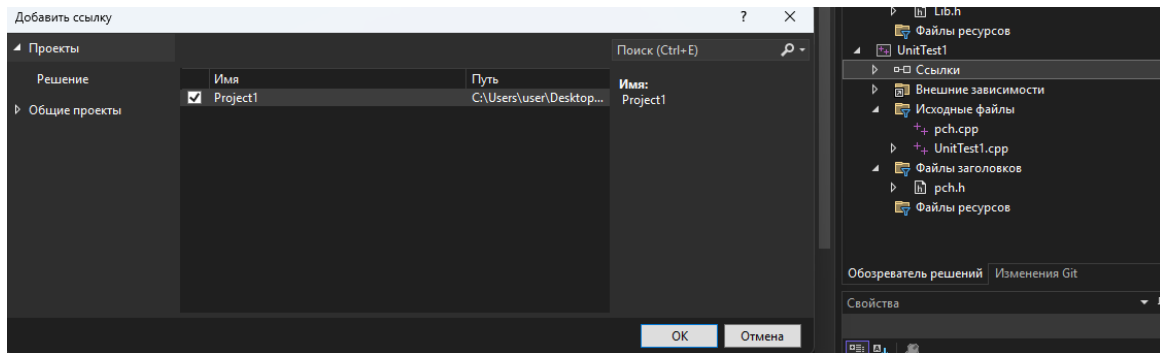
```
#include "../Project1/Lib.h"
```



```
UnitTest1.cpp  ➤ ✕
UnitTest1
1  #include "pch.h"
2  #include "CppUnitTest.h"
3  #include "../Project1/Lib.h"
4
5  using namespace Microsoft::VisualStudio::CppUnitTestFramework;
6
7  namespace UnitTest1
8  {
9      TEST_CLASS(UnitTest1)
10     {
11     public:
12
13         TEST_METHOD(TestMethod1)
14         {
15         }
16     };
17 }
```

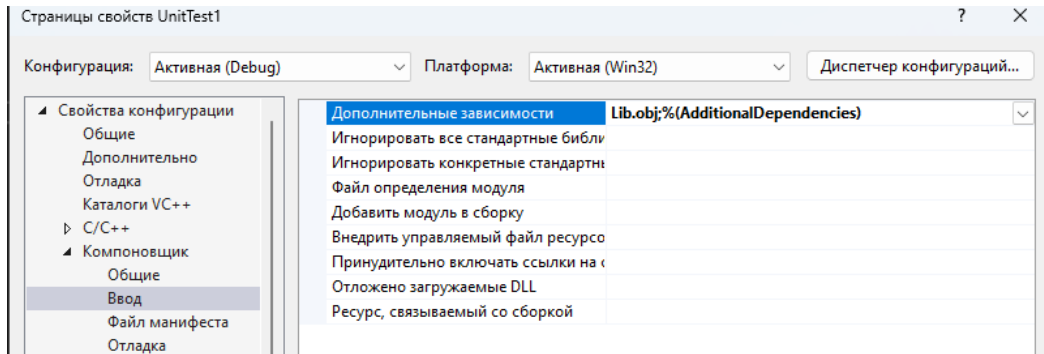
Шаг 6. Установка зависимости проектов

- 1 В обозревателе решений глазами найти проект тестов `UnitTest1`.
- 2 Щелкнуть ПКМ по пункту "Ссылки" → "Добавить ссылку".
- 3 Поставить галочку напротив `Project1`. Нажать "ОК".



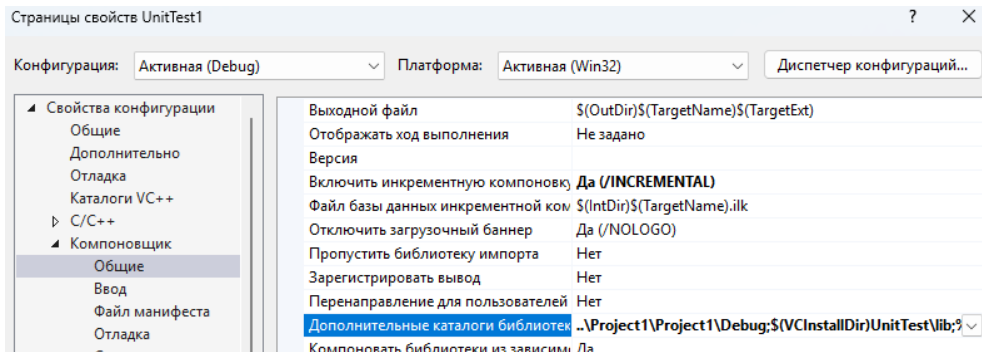
Шаг 7. Добавление Lib.obj

- 1 ПКМ по проекту тестов UnitTest1 в обозревателе решений → "Свойства".
- 2 Перейти в "Компоновщик" → "Ввод".
- 3 В поле "Дополнительные зависимости" добавить Lib.obj.



Шаг 8. Добавление каталога для библиотек

- 1 В том же окне свойств перейти в "Компоновщик" → "Общие".
- 2 В поле "Дополнительные каталоги библиотек" прописать путь к папке, где лежит Lib.obj.
- 3 Обычно: ..\Project1\Debug. Вы должны были его запомнить на **Шаге 2**.
- 4 В новых версиях VS может быть так: ..\Project1\Project1\Debug.



Шаг 9. Пишем тесты

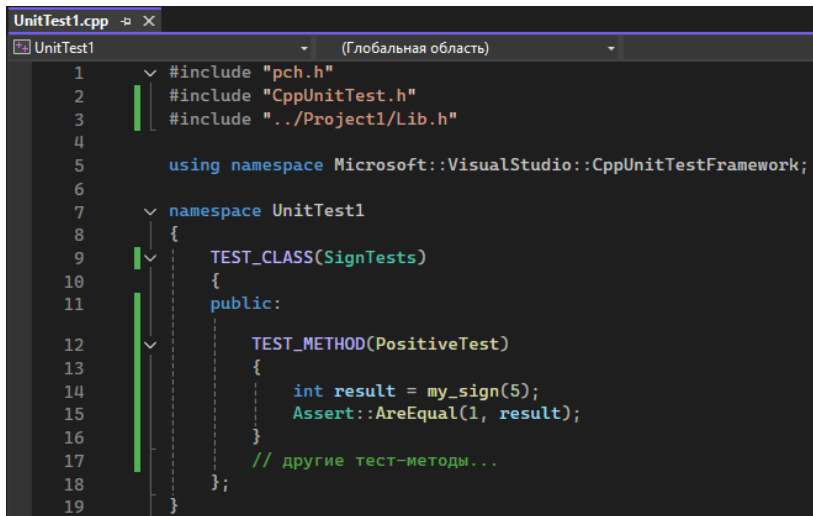
Пример тест-класса

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Project1/Lib.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
    TEST_CLASS(SignTests) {
    public:
        TEST_METHOD(PositiveTest)
        {
            int result = my_sign(5);
            Assert::AreEqual(1, result);
        }
        // другие тест-методы...
    };
}
```

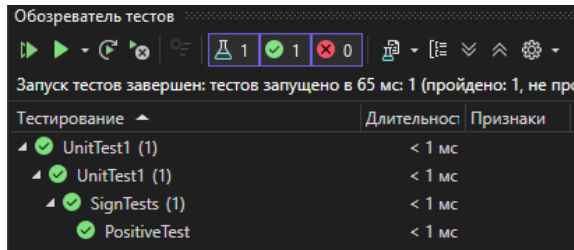
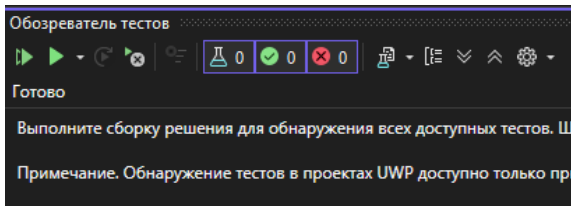
Шаг 9. Пишем тесты



```
UnitTest1.cpp  ▢ ×
+ UnitTest1  (Глобальная область)
1  #include "pch.h"
2  #include "CppUnitTest.h"
3  #include "../Project1/Lib.h"
4
5  using namespace Microsoft::VisualStudio::CppUnitTestFramework;
6
7  namespace UnitTest1
8  {
9      TEST_CLASS(SignTests)
10     {
11     public:
12
13         TEST_METHOD(PositiveTest)
14         {
15             int result = my_sign(5);
16             Assert::AreEqual(1, result);
17             // другие тест-методы...
18         };
19     }
```

Шаг 10–11. Запуск тестов

- 1 В верхнем меню выбрать "Тест" → "Обозреватель тестов".
- 2 Нажать зелёную кнопку "Запустить все".
- 3 Тесты выполняются, и вы увидите результаты (успешно/неуспешно).



Правила оформления тестов

- ❶ **1 функция** → **1 тест-класс** — группируем тесты по тестируемой функции.
- ❷ **1 гипотеза** → **1 тест-метод** — каждый тест проверяет один сценарий.
- ❸ **В каждом тест-методе ровно 1 Assert** (исключения — массивы, меняющие размер).

На примере для функции `my_sign`

- ❶ Тест-класс: `SignTests`
- ❷ Тест-методы: `PositiveTest`, `ZeroTest`, `NegativeTest`

Полный пример: функция my_sign

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Project1/Lib.h"
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
    TEST_CLASS(SignTests) {
    public:
        TEST_METHOD(PositiveTest) {
            Assert::AreEqual(1, my_sign(5));
        }
        TEST_METHOD(ZeroTest) {
            Assert::AreEqual(0, my_sign(0));
        }
        TEST_METHOD(NegativeTest) {
            Assert::AreEqual(-1, my_sign(-3));
        }
    };
}
```

Правила оформления тестов (продолжение)

Assert::AreEqual

```
Assert::AreEqual(expected, actual); // для целых чисел  
Assert::AreEqual(expected, actual, accuracy); // для вещественных
```

Булевы значения

```
Assert::IsTrue(condition);  
Assert::IsFalse(condition);
```

Тестирование целых и вещественных

Целочисленные функции

```
TEST_METHOD(PositiveTest) {  
    Assert::AreEqual(1, my_sign(5));  
}
```

Вещественные функции (с точностью)

```
TEST_METHOD(SqrtTest) {  
    double my_eps = 1e-6; // ваша точность  
    double result = my_sqrt(4.0, my_eps); // считаем корень из 4.0 с точностью my_eps  
    Assert::AreEqual(2.0, result, 1e-4); // OK, посчитали 6 знаков, а требуем 4  
    // Assert::AreEqual(2.0, result, 1e-9); // упадет, требуется точность выше, чем даёт функция  
}
```

- **Корректная точность:** указывайте разумный эпсилон в зависимости от погрешности вычислений.
- **Неправильная точность:** если указать эпсилон меньше погрешности, тест может упасть.

Тестирование статических массивов с `std::equal`

Сравнение статических массивов

```
#include <algorithm>
TEST_METHOD(ArrayTest) {
    int expected[] = {1, 2, 3};
    int actual[3];
    // вызываем функцию, которая заполняет actual
    Assert::IsTrue(std::equal(expected, expected + 3, actual, actual + 3)); // сравниваем значения
}
```

Для статических массивов С-строк типа `char` все аналогично.

Примеры сравнений

```
#include <algorithm>
TEST_METHOD(ArrayTest) {
    int expected[3] = { 1, 2, 3 };
    int actual[5] = {1, 2, 3, 4, 5};
    Assert::IsTrue(std::equal(expected, expected + 3, actual)); // True, перегрузка с 3 аргументами

    Assert::IsTrue(std::equal(actual, actual + 5, expected)); // False

    int expected2[3] = { 1, 2, 3 };
    int actual2[3] = { 1, 2, 3 };
    Assert::IsTrue(std::equal(expected2, expected2 + 3, actual2, actual2 + 3)); // True, перегрузка
    ↪ с 4 аргументами
}
```

Тестирование статических массивов без `std::equal`

Сравнение статических массивов

```
TEST_METHOD(ArrayTest) {  
    int expected[] = {1, 2, 3};  
    int actual[3];  
    // вызываем функцию, которая заполняет actual  
    for (int i = 0; i < 3; ++i) {  
        Assert::AreEqual(expected[i], actual[i]); // сравниваем значения элементов  
    }  
}
```

Тестирование динамических массивов с `std::equal`

Сравнение динамических массивов

```
#include <algorithm>
TEST_METHOD(DynamicArrayTest) {
    int expected[] = { 1, 2, 3, 4 };
    int expectedSize = 4;
    int* actual;
    int actualSize;
    // вызываем функцию, которая заполняет массив actual и его размер
    Assert::AreEqual(expectedSize, actualSize); // сначала сравниваем размеры
    Assert::IsTrue(std::equal(expected, expected + expectedSize, actual)); // True
    delete[] actual; // если выделяли память
}
```

Если размер массива не изменяется, проверка упрощается.
Для динамических массивов С-строк типа `char*` все аналогично.

Тестирование динамических массивов без `std::equal`

Сравнение динамических массивов

```
TEST_METHOD(DynamicArrayTest) {
    int expected[] = {1, 2, 3, 4};
    int expectedSize = 4;
    int* actual;
    int actualSize;
    // вызываем функцию, которая заполняет массив actual и его размер
    Assert::AreEqual(expectedSize, actualSize); // сначала сравниваем размеры
    for (int i = 0; i < actualSize; ++i) {
        Assert::AreEqual(expected[i], actual[i]); // затем значения элементов
    }
    delete[] actual; // если выделяли память
}
```

Способ 1: Ручная проверка через try/catch

```
TEST_METHOD(DivideByZeroThrowsManual) {  
    try {  
        divide(10, 0);  
        Assert::Fail(L"Expected exception not thrown"); // не было выброшено -> ошибка  
    }  
    catch (const std::runtime_error&) {  
        Assert::IsTrue(true); // тест пройден, поймал исключение типа std::runtime_error  
    }  
    catch (...) {  
        Assert::Fail(L"Unexpected exception type"); // не тот тип исключения  
    }  
}
```

Способ 2: Использование Assert::ExpectException

```
TEST_METHOD(DivideByZeroThrows) {  
    auto func = [] { divide(10, 0); };  
    Assert::ExpectException<std::runtime_error>(func);  
}
```

Способ 2 короче и читаемее. Способ 1 полезен, если нужно проверить сообщение об ошибке или ловить простой тип.

Тестирование векторов

Сравнение векторов

```
#include <vector>
TEST_METHOD(VectorTest) {
    std::vector<int> expected = {1, 2, 3, 4, 5};
    std::vector<int> actual = getVector(); // тестируемая функция

    Assert::IsTrue(std::equal(begin(expected), end(expected), begin(actual))); // True

    Assert::IsTrue(expected == actual); // True, другой способ
}
```

Проверка пустого вектора

```
TEST_METHOD(EmptyVectorTest) {
    std::vector<int> actual = getEmptyVector();
    Assert::IsTrue(actual.empty());
}
```

Тестирование строк `std::string`

Сравнение строк целиком

```
#include <string>
TEST_METHOD(StringTest) {
    std::string expected = "Hello, World!";
    std::string actual = getGreeting(); // тестируемая функция
    Assert::AreEqual(expected, actual); // оператор == перегружен, поэтому просто сравниваем
}
```

Сравнение строк целиком

```
TEST_METHOD(ConstCharTest) {  
    const char* expected = "Hello, World!";  
    const char* actual = getConstChar(); // тестируемая функция  
    Assert::IsTrue(!strcmp(expected, actual)); // strcmp возвращает 0, если строки равны  
}
```

- Научились создавать проект модульного теста.
- Подключать тестируемый код через файлы `'h'` и `'obj'`.
- Познакомились с основными правилами написания тестов.
- Теперь вы можете автоматизировать проверку своих функций.
- Это основа для разработки надёжного кода.

Что дальше?

- На практике — учимся писать тесты для своих функций.
- В следующей лекции — структуры, перечисления, кортежи.

Вопросы?