

Тестирование

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center



Проверка связи





Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



Поставьте в чат:

-  если меня видно и слышно
-  если нет

Амиран Мстоян

О спикере:

AI System Engineer (C++) в Huawei Moscow Research Center

Математик-алгоритмист лаборатории научного центра при МФТИ



Вспоминаем прошрое занятие

Вопрос: какие 2 модели памяти есть в C++?



Вспоминаем прошрое занятие

Вопрос: какие 2 модели памяти есть в C++?

Ответ: память на стеке и память в куче



Вспоминаем прошрое занятие

Вопрос: из чего состоит жизненный цикл
объекта в C++?



Вспоминаем прошрое занятие

Вопрос: из чего состоит жизненный цикл объекта в C++?

Ответ: создание, действия с объектом, удаление



Вспоминаем прошрое занятие

Вопрос: в чем хорошее свойство
автоматических переменных (переменных на
стеке)?



Вспоминаем прошрое занятие

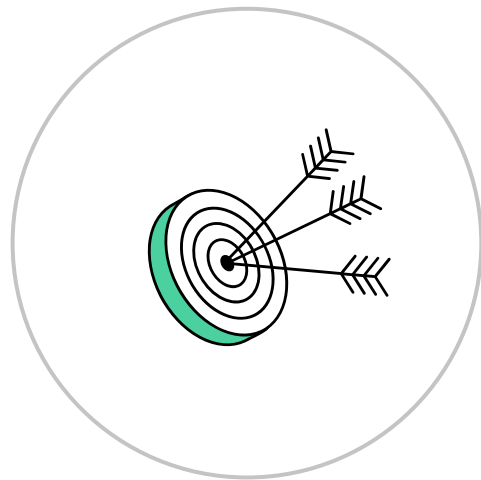
Вопрос: в чем хорошее свойство автоматических переменных (переменных на стеке)?

Ответ: автоматическое удаление после выхода из блока кода, где они объявлены



Цели занятия

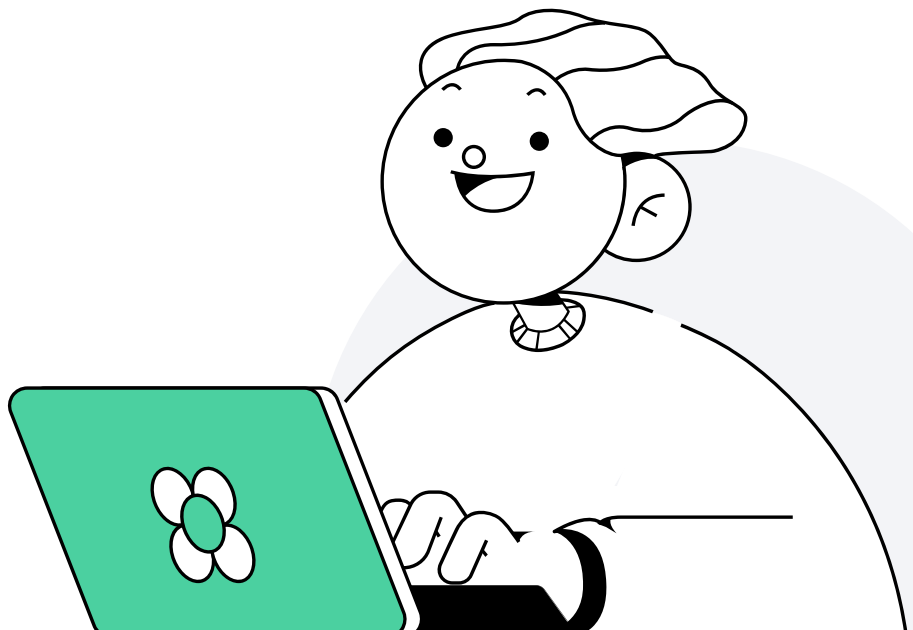
- Узнаем зачем нужно писать тесты
- Изучим виды тестирования ПО
- Посмотрим как писать тесты на C++



План занятия

- 1 Виды тестирования ПО
- 2 Обзор фреймворка Catch2
- 3 Итоги

*Нажми на нужный раздел для перехода



Зачем тестировать?

Например, вы могли опечататься и перепутать знаки. Или в выводе случайно написать “меньше” вместо “больше”.

```
void compare(int a, int b)
{
    if (a > b)
        std::cout << a << " больше " << b << '\n';
    else if (a < b)
        std::cout << a << " меньше " << b << '\n';
    else
        std::cout << a << " равно " << b << '\n';
}
```

Зачем тестировать?

Например, вы могли опечататься и перепутать знаки. Или в выводе случайно написать “меньше” вместо “больше”.

```
void compare(int a, int b)
{
    if (a > b)
        std::cout << a << " больше " << b << '\n';
    else if (a < b)
        std::cout << a << " меньше " << b << '\n';
    else
        std::cout << a << " равно " << b << '\n';
}
```

В случае, если эта функция используется где-то еще, то ошибку будет найти сложнее: нужно будет проверять все вызовы функций. Тесты помогут уменьшить количество таких ситуаций.

А как тестировать?

```
void compare(int a, int b)
{
    if (a > b)
        std::cout << a << " больше " << b << '\n';
    else if (a < b)
        std::cout << a << " меньше " << b << '\n';
    else
        std::cout << a << " равно " << b << '\n';
}
```

Вопрос: Как можно протестировать данную функцию?



А как тестировать?

```
void compare(int a, int b)
{
    if (a > b)
        std::cout << a << " больше " << b << '\n';
    else if (a < b)
        std::cout << a << " меньше " << b << '\n';
    else
        std::cout << a << " равно " << b << '\n';
}
```

Вопрос: Как можно протестировать данную функцию?

Ответ: Сравнивать абсолютно все пары чисел типа **int** бессмысленно. В данном случае можно протестировать всего 3 пары чисел.



Виды тестирования ПО



1

Виды тестирования

1

Модульное
Unit

2

Сквозное
End to end

3

Интеграционное
Integration

4

Тестирование производительности
Performance

5

Функциональное
Functional

6

Дымовое
Smoke

Модульное тестирование

Как правило сложные системы состоят из различных частей - **модулей**

Для корректной работы системы необходимо, чтобы каждый **модуль работал правильно**: это и проверяют модульные тесты

Модульное тестирование

Модульные тесты работают на низком уровне - близко к исходному коду приложения. Они заключаются в тестировании отдельных методов и функций классов, компонентов или модулей, используемых в ПО



Выполнение модульных тестов не занимает много времени

Интеграционное тестирование

В ходе интеграционного тестирования проверяется, **хорошо ли работают** вместе различные **модули и сервисы**, используемые приложением

Например, можно протестировать взаимодействие с базой данных или убедиться, что микросервисы работают вместе так, как задумано



Этот вид тестирования является более затратным, поскольку для проведения тестов требуется запуск различных компонентов приложения

Функциональное тестирование

Почти то же самое, что и интеграционное тестирование. Только теперь проверяется результат работы системы на **предмет удовлетворения бизнес-требований**

Например, интеграционный тест нужен чтобы убедиться, что вы можете отправлять запросы к серверу, тогда как функциональный тест будет ожидать получения от сервера определенного ответа в соответствии с требованиями продукта



Этот вид тестирования также может быть затратным по времени

Функциональное тестирование

Может быть двух видов:

- белый ящик - тестирование со знанием внутренней структуры системы
- черный ящик - тестирование без знания внутренней реализации

Сквозное тестирование

Моделирует поведение пользователя. Обеспечивает контроль того, что **пользовательские сценарии** работают корректно.

Сценарии могут быть:

- очень простыми (авторизация)
- гораздо более сложными (проведение долгих расчётов)



Этот вид тестирования очень затратен по времени

Тестирование производительности

Оценивается **работа системы** при определенной рабочей нагрузке.

С помощью таких тестов можно оценить:

- надежность,
- скорость,
- масштабируемость
- отзывчивость приложения



Этот вид тестирования также может быть затратным по времени

Тестирование производительности

- ① Нагрузочное тестирование
 - позволяет определить максимально возможный **пик, при котором система работает правильно**
- ② Стресс тестирование
 - предназначено для проверки **системы в нестандартных нагрузках** и для выявления результата, когда система приходит в нерабочее состояние

Дымовое тестирование

Тесты, которые позволяют определить корректно ли работают **базовые возможности системы**.

Обычно используют сразу после развертывания системы или после создания новой сборки (для определения необходимости запуска более ресурсоемких тестов)



Этот вид тестирования может быть затратен по времени

Обзор фреймворка Catch2



2

Расчет факториала

Разберемся с unit-тестированием на примере функции расчета факториала.

Факториал числа n - это произведение всех чисел от 1 до n включительно.
Факториал 0 равен 1.

Расчет факториала

Определим следующую функцию перед функцией main.

```
unsigned int Factorial(unsigned int number) {  
    if (number == 0 || number == 1)  
        return 1;  
    else  
        return Factorial(number - 1) * number;  
}
```

Будем проверять нашу функцию на следующих значениях: 0, 1, 5, 10.

$0! = 1$, $1! = 1$, $5! = 120$, $10! = 3\,628\,800$

Функции проверки истинности в Catch

В данном фреймворке есть 2 макроса проверки истинности выражения: REQUIRE и CHECK. Разница между ними лишь в том, что если выражение в вызове REQUIRE ложно, то весь код после вызова этого макроса не выполняется.

Синтаксис:

```
REQUIRE( левая часть выражения == правая часть выражения );  
CHECK( левая часть выражения == правая часть выражения );
```

Тесты будут выглядеть следующим образом:

```
CHECK(Factorial(0) == 1);  
CHECK(Factorial(1) == 1);  
CHECK(Factorial(5) == 120);  
CHECK(Factorial(10) == 3628800);
```

TEST_CASE

В Catch вызовы функций CHECK и REQUIRE должны находиться в блоке TEST_CASE. Синтаксис:

```
TEST_CASE( "название блока", "[название для фильтра]");
```

Название для фильтра можно использовать, чтобы отфильтровывать запуск нужных тестов через аргументы командной строки.

Наш блок тестов будет выглядеть следующим образом:

```
TEST_CASE("test factorial", "[factorial]") {  
    CHECK(Factorial(0) == 1);  
    CHECK(Factorial(1) == 1);  
    CHECK(Factorial(5) == 120);  
    CHECK(Factorial(10) == 3628800);  
}
```

SECTIONS

Для удобства можно блок TEST_CASE поделить на секции, логически сгруппировав тесты. Синтаксис:

```
SECTION( "название секции");
```

В нашем примере можем выделить вызовы с положительными числами в отдельную секцию. Блок тестов изменится следующим образом:

```
TEST_CASE("test factorial", "[factorial]") {  
    CHECK(Factorial(0) == 1);  
    SECTION("positive numbers") {  
        CHECK(Factorial(1) == 1);  
        CHECK(Factorial(5) == 120);  
        CHECK(Factorial(10) == 3628800);  
    }  
}
```


Запуск тестов

Запуск тестов можно осуществить следующим образом:

```
Catch::Session().run(argc, argv);
```

Для этого нужно подключить заголовочный файл

#include<catch2/catch_session.hpp>

Функция **run** возвращает кол-во непрошедших тестов. В итоге получим такой исходный код.

Исходный код с тестами

```
#pragma once

#include <catch2/catch_test_macros.hpp>
#include <catch2/catch_session.hpp>

unsigned int Factorial(unsigned int number) {
    if (number == 0 || number == 1)
        return 1;
    else
        return Factorial(number - 1) * number;
}

TEST_CASE("test factorial", "[factorial]") {
    CHECK(Factorial(0) == 1);
    SECTION("positive numbers") {
        CHECK(Factorial(1) == 1);
        CHECK(Factorial(5) == 120);
        CHECK(Factorial(10) == 3628800);
    }
}

int main(int argc, char* argv[]) {
    return Catch::Session().run(argc, argv);
}
```

Результат работы

После запуска программы получим сообщение, что все тесты пройдены

```
=====
All tests passed (4 assertions in 1 test case)
```

Давайте, добавим ошибки в некоторых тестах. Умышленно будем сравнивать факториал 0 и 5 с неправильными числами. Вывод изменится следующим образом:

```
C:\Users\sinit\Desktop\test\tests\tests_main.cpp(15): FAILED:
```

```
  CHECK( Factorial(0) == 10 )
```

```
with expansion:
```

```
  1 == 10
```

```
-----
test factorial
```

```
  positive numbers
```

```
-----
C:\Users\sinit\Desktop\test\tests\tests_main.cpp(16)
```

```
C:\Users\sinit\Desktop\test\tests\tests_main.cpp(18): FAILED:
```

```
  CHECK( Factorial(5) == 1 )
```

```
with expansion:
```

```
 120 == 1
```

```
-----
test cases: 1 | 1 failed
```

```
assertions: 4 | 2 passed | 2 failed
```

Макрос INFO

Бывает полезно выводить какую-то дополнительную информацию, когда тест упал. Для этого существует макрос INFO.

Когда тест пройден, вывод содержимого игнорируется. Добавим следующий код перед проверкой факториала 5.

```
INFO("Factorial(5) calculated wrong");
```

```
C:\Users\sinit\Desktop\test\tests\tests_main.cpp(19): FAILED:
```

```
  CHECK( Factorial(5) == 1 )
```

```
with expansion:
```

```
  120 == 1
```

```
with message:
```

```
Factorial(5) calculated wrong
```

```
=====
```

```
test cases: 1 | 1 failed
```

```
assertions: 4 | 2 passed | 2 failed
```

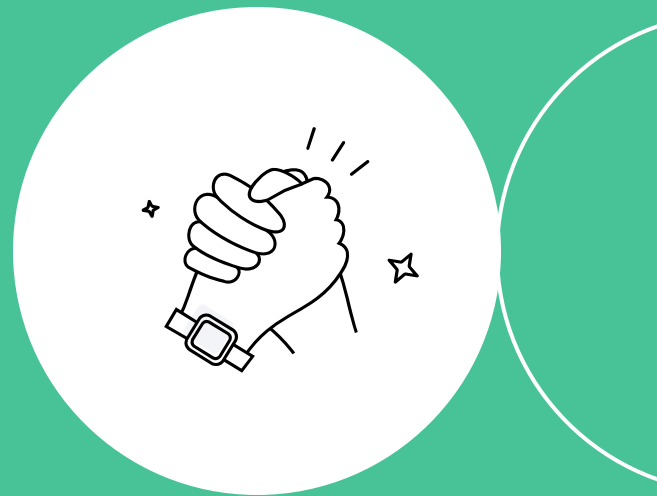
Сравнение чисел с плавающей точкой

Числа с плавающей точкой удобно сравнивать с помощью класса `Approx`. У него есть методы для сравнения с допущением относительной ошибки (`epsilon`) и абсолютной (`margin`)

```
Approx target = Approx(100).epsilon(0.01);  
100.0 == target; // Очевидно выполнено  
200.0 == target; // Очевидно не выполнено  
100.5 == target; // Выполнено, так как допускаем отклонение в 1%
```

```
Approx target = Approx(100).margin(5);  
100.0 == target; // Очевидно выполнено  
200.0 == target; // Очевидно не выполнено  
104.0 == target; // Выполнено, так как допускаем абсолютную ошибку 5
```

Итоги



Итоги занятия

- 1 Узнали какие бывают виды тестирования ПО
- 2 Познакомились с фреймворком Catch для создания unit-тестов



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- [Статья](#) про виды тестирования ПО
- [Ссылка](#) на документацию Catch2



Задавайте вопросы и пишите отзыв о лекции

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center

