Юнит-тестирование в C++11 с использованием gtest и doctest

Лекторы:
Аспирант МФТИ, Шер Артём Владимирович
Аспирант МФТИ, Зингеренко Михаил Владимирович

17 сентября 2024

Но сначала ещё немного про линковку

Pybind11

- Для C++
- Нужна небольшая обвязка исходников под питон
- Собирает модуль из исходников

CFFI

- Для чистого C и обёртками под него C++
- Неудобная сборка, нужны пути до всех хедеров и библиотек
- Дополнительная динамическая библиотека
- Собирает модуль из исходников

ctypes

- Для чистого C и обёртками под него C++
- В питоне нужно описывать, ret и arg для C функций
- Загрузка через собственные модули

Пример для pybind11

```cpp
#include <pybind11/pybind11.h> //my_cpp_code.cpp

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

PYBIND11_MODULE(my_cpp_module, m) {
    m.def("add", &add, "A function that adds two numbers");
}
```

```python
from setuptools import setup, Extension  # setup.py
from pybind11.setup_helpers import Pybind11Extension, build_ext

ext_modules = [
    Pybind11Extension(
        "my_cpp_module",        # Name of the resulting Python module
        ["my_cpp_code.cpp"],    # Source file
    ),
]

setup(
    name="my_cpp_module",
    ext_modules=ext_modules,
    cmdclass={"build_ext": build_ext},
)
```

```python
import my_cpp_module # test.py

result = my_cpp_module.add(3, 4)
print(f"The result of addition is: {result}")
```

Пакетные менеджеры

- FetchContent (CMake >= 3.11)

  `https://cmake.org/cmake/help/latest/module/FetchContent.html`
- Vcpkg `https://github.com/Microsoft/vcpkg`
- Hunter `https://github.com/cpp-pm/hunter`

Что такое юнит-тестирование?

Юнит-тестирование — это метод тестирования программного обеспечения, при котором отдельные компоненты (например, функции или классы) тестируются на корректность их работы в изоляции.

Основные цели юнит-тестирования

- Обеспечение качества кода: гарантирует, что каждая часть программы работает правильно.
- Защита от регрессий: позволяет убедиться, что изменения в коде не нарушают уже существующую функциональность.
- Документация кода: тесты могут выступать в роли документации, демонстрируя, как использовать функции и классы.
- Безопасный рефакторинг: тесты позволяют изменять код без риска сломать его логику.

Преимущества юнит-тестирования

- Повышение надежности программного обеспечения.
- Ускорение разработки за счет автоматической проверки корректности кода.
- Улучшение поддержки и сопровождения кода в будущем.
- Улучшение проектирования: возможность продумывать сценарии использования кода с самого начала.

gtest (Google Test)

Простота использования: Разработан Google и имеет мощный функционал для написания юнит-тестов. Параметризованные тесты: gtest позволяет запускать один и тот же тест с различными наборами данных.
Ассерты (утверждения):
ASSERT_EQ, ASSERT_TRUE, ASSERT_STRNE — проверка условий, приводящая к немедленному завершению теста при неудаче.
EXPECT_EQ, EXPECT_TRUE, EXPECT_STRNE — проверка условий с продолжением теста при неудаче.

Пример использования

```
1  #include <gtest/gtest.h>
2
3  int add(int a, int b) {
4      return a + b;
5  }
6
7  TEST(AdditionTest, PositiveNumbers) {
8      ASSERT_EQ(add(1, 2), 3);
9  }
10
11 TEST(AdditionTest, NegativeNumbers) {
12     ASSERT_EQ(add(-1, -2), -3);
13 }
14
15 int main(int argc, char **argv) {
16     ::testing::InitGoogleTest(&argc, argv);
17     return RUN_ALL_TESTS();
18 }
```

Параметризованные тесты

gtest поддерживает возможность запускать один и тот же тест с разными наборами данных. Пример параметризованного теста:

```
1 class ParamTest : public ::testing::TestWithParam<int> { };
2
3 TEST_P(ParamTest, PositiveTest) {
4     ASSERT_GT(GetParam(), 0);
5 }
6
7 INSTANTIATE_TEST_SUITE_P(MyParams, ParamTest, ::testing::Values(1, 2, 3));
```

doctest

- Легковесный: doctest — это минималистичный и быстрый фреймворк, встраиваемый в код, как в Python.
- Простота написания тестов: тесты могут быть включены прямо в код программы, как часть комментариев.
- Малый оверхед: doctest загружается быстро и не добавляет много лишнего к проекту.
- Удобные макросы для ассертов:
  - CHECK(), REQUIRE(): макросы для проверки условий, где REQUIRE завершает тест при неудаче.

Пример использования

```
1  #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
2  #include "doctest.h"
3
4  int factorial(int number) { return number <= 1 ? 1 : number * factorial(number -
       1); }
5
6  TEST_CASE("testing the factorial function") {
7      CHECK(factorial(1) == 1);
8      CHECK(factorial(2) == 2);
9      CHECK(factorial(3) == 6);
10     CHECK(factorial(10) == 3628800);
11 }
12
13 TEST_CASE("multiple subcases") {
14     SUBCASE("subcase 1") {
15         CHECK(1 == 1);
16     }
17     SUBCASE("subcase 2") {
18         CHECK(2 == 2);
19     }
20 }
```

До следующей лекции!