Задания 04

Модульное тестирование (часть 2)

Параметризованные тесты

Часто бывает необходимо протестировать корректность работы функции на большом количестве комбинаций аргументов. Организовать это в виде цикла не удобно, поскольку макросы `EXPECT*` при возникновении ошибки выдают номер строки, но не показывают номер итерации. Для такого случая в библиотеке Gtest предусмотрен запуск тестов на наборе параметров. Удобнее всего в качестве параметра использовать структуру с входным данными и ожидаемыми результатами работы тестируемой функции.

Для того чтобы создать тест с параметрами, необходимо создать класс с называнием этого теста (для тестов без параметров библиотека Gtest делает это автоматически). Рассмотрим тест для функции, вычисляющей хэш строки по алгоритму SHA-1.

```
struct sha1_param {
    std::string input;
    std::string expected_output;
};

class sha1_test: public ::testing::TestWithParam<sha1_param> {};
```

Здесь `sha1_test` является названием, а `sha1_param` — типом параметра теста. Далее следует реализовать сам тест.

```
TEST_P(sha1_test, _) {
    const sha1_param& param = GetParam();
    mysha1 sha;
    sha.input(param.input);
    sha.compute();
    EXPECT_EQ(param.expected_output, sha.result());
}
```

Здесь вместо привычного `TEST` используется `TEST_P`, что означает тест с параметрами. Конкретное значение параметра получается путем вызова функции `GetParam()`.

Вместо знака подчеркивания можно написать название теста, которое обычно является излишним. Наконец, надо определить список параметров, на которых будет тестироваться функция.

```
INSTANTIATE_TEST_CASE_P(
    _,
    sha1_test,
    ::testing::Values(
        sha1_param{
            "The quick brown fox jumps over the lazy dog",
            "2fd4e1c6 7a2d28fc ed849ee1 bb76e739 1b93eb12"
        },
        sha1_param{
            "", "da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709"
        }
    )
);
```

Здесь вместо знака подчеркивания можно написать название экземпляра теста, которое обычно является излишним. Внутри `Values` через запятую записываются параметры, на которых тестируется функция. Если список параметров сгенерировать с помощью средств языка C++ и сохранить в контейнере (например, в векторе), то такой список указывается с помощью `ValuesIn(myparams)`.

Типизированные тесты

Типизированный тест — это тест, параметром которого выступает тип. Рассмотрим тест, в котором проверяется работа операторов ввода/вывода для произвольных классов.

```
template <class T>
struct input_output_test: public ::testing::Test {};

typedef my_ipv4_address type1;
typedef my_ipv6_address type2;

TYPED_TEST_CASE(input_output_test, ::testing::Types<type1,type2>);

TYPED_TEST(input_output_test, symmetry) {
    TypeParam addr1 = random_address<TypeParam>();
    TypeParam addr2;
    std::stringstream s;
    s << addr1;
    s >> addr2;
    EXPECT_EQ(addr1, addr2);
}
```

Здесь `input_output_test` — название теста, `TypeParam` — тип параметра теста, `type1`, `type2` — типы, для которых запускается тест. Отдельные переопределения типов рекомендуется делать, поскольку макрос `TYPED_TEST_CASE` может не сработать, если в него в качестве типа передать шаблон со списком аргументов.

Тесты с локальным состоянием

Если в вашем тесте используется объект, который нужно инициализировать в начале и/или очистить в конце каждого теста, то для этого подойдет тест с локальным состоянием. Рассмотрим тест расчета качки судна.

```
struct ship_motion_test: public ::testing::Test {

    void
    SetUp() {
        ship.set_mass(1e8);
        // ...
    }

    myship ship;
};

// равноускоренное движение
TEST_F(ship_motion_test, uniformly_accelerated) {
    float t0 = 0, t = 1;
    auto x = ship.state_vector();
    // ...
}

TEST_F(ship_motion_test, angular) {
    // ...
}
```

Здесь тест объявляется с помощью макроса `TEST_F`, и его название совпадает с название класса, в котором хранится локальное состояние. Для всех тестов, имеющих такое название, создается один экземпляр класса `ship_motion_test`. В начале каждого теста вызывается метод `SetUp`, а в конце `TearDown` (здесь он отсутствует). Внутри теста поля класса доступны.

Тесты с локальным состоянием позволяют избежать дублирование кода инициализации. Часто такие тесты используются для генерации нужных таблиц в базе данных перед проверкой корректности выполнения запросов к ней.

Тесты с глобальным состоянием

Иногда необходимо провести глобальную инициализацию, чтобы выполнить некоторые тесты. Примером может служить тесты для OpenCL. Инициализировать OpenCL в начале каждого теста неэффективно, а в начале функции `main` нежелательно, поскольку может привести к нечитаемым ошибкам в выводе программы. Для глобальной инициализации в Gtest предусмотрены окружения. Рассмотрим инициализацию OpenCL с помощью окружения.

```
class opencl_environment: public ::testing::Environment {
public:
    void
    SetUp() {
        my_init_opencl("opencl.conf");
    }
};

// тесты ...

int main(int argc, char* argv[]) {
    ::testing::InitGoogleTest(&argc, argv);
    ::testing::AddGlobalTestEnvironment(new opencl_environment);
    return RUN_ALL_TESTS();
}
```

Здесь перед запуском тестов вызывается метод `SetUp` у всех окружений, зарегистрированных в Gtest. Окружения регистрируются в функции `main`. После завершения всех тестов вызывается метод `TearDown` (здесь он отсутствует).

Тесты с глобальным состоянием удобны для создания временных рабочих директорий, в которых тест будет создавать, читать и писать файлы, а также для тестов, которые требуют изменения состояния всего процесса (например, текущей директории, маски сигналов и т.п.).

Задания

1. Создайте сторожевой объект для сохранения состояния потока вывода `std::ostream`. В состояние входят флаги форматирования (метод `flags`) и символ заполнения (метод `fill`). Проверьте его работоспособность на любом из стандартных потоков. Пример использования:

   ```
   void print_hex(int i) {
       MyGuard g(std::cout);
       std::cout.setf(std::ios_base::hex, std::ios_base::basefield);
       // или std::cout << std::hex
       std::cout << i;
   ```

```
    // в деструкторе MyGuard флагу возвращается исходное
значение
}
```

2. Создайте класс для двухмерного массива. Определите для него поэлементные арифметические и логические операции. Каждая арифметическая операция должна возвращать массив с тем же размером и типом элемента, каждая логическая — массив с тем же размером и булевым типом элемента. Реализуйте также оператор `()` для доступа к элементу по двум индексам, а также для создания среза массива по булевой маске.

   ```
   MyMaskedArray2D operator()(const MyArray2D<bool>& mask);
   ```

   Результатом работы оператора является псевдо-объект, который содержит ссылку на исходный массив и на маску и для которого определен оператор присваивания. Оператор присваивания принимает в качестве аргумента скаляр и приравнивает его всем элементам массива, для которых в соответствующем элементе маски стоит значение ИСТИНА.

3. Проверьте работу с помощью параметризованного теста.