استاد: رامتین خسروی
طراح: هادی صفری
مهلت تحویل: یکشنبه ۱۹ خرداد ۱۳۹۸، ساعت ۲۳:۵۵

۱ مقدمه‌ای بر آزمون نرم‌افزار

با آزمون نرم‌افزار و *آزمون واحد*[۱] و مزایا و لزوم آن‌ها در کلاس درس آشنا شده‌اید. آزمون واحد روشی برای آزمودن واحدهای کد منبع و اطمینان از مناسب بودن آن‌ها است. واحد کوچک‌ترین بخش آزمودنی کد منبع است که در زبان‌ها و الگوهای برنامه‌نویسی‌ای که با آن‌ها آشنایی دارید معمولاً معادل یک تابع است. واحد تحت تست[۲] معمولاً یا تست‌ها را می‌گذراند[۳] یا در آن‌ها شکست می‌خورد[۴]. گاهی ممکن است آزمونی در حلقهٔ بینهایت بیفتد یا زمان آن تمام شود[۵]. برای اطلاعات بیشتر دربارهٔ آزمون واحد می‌توانید به منبع [۴] مراجعه کنید.

۱.۱ طراحی آزمون‌ها

آزمون واحد معمولاً بر اساس ادعاها[۶] بنا می‌شود. انتظار می‌رود ادعاها برقرار باشند و در غیر این صورت آزمون شکست می‌خورد. یک راه معمول طراحی آزمون واحد آن است که ابتدا شرایط مناسب برای آزمون فراهم می‌گردد؛ مثلاً یک شیء از کلاس تحت آزمون با شرایط مورد نظر ایجاد شود. سپس عملیات خاصی صورت می‌گیرد که هدف آزمون بررسی صحت آن عملیات خاص است؛ مثلاً یکی از توابع کلاس تحت آزمون صدا می‌شود تا محاسبه‌ای را انجام دهد و نتیجهٔ آن ذخیره می‌گردد. در انتها با بازبینی اتفاقات رخ‌داده از صحت عملیات انجام‌شده اطمینان حاصل می‌شود؛ مثلاً نتیجهٔ اجرای تابع محاسباتی مذکور را با مقدار مورد انتظار مقایسه می‌کنیم. این الگو به الگوی AAA[۷] مشهور است [۱، ۳]. تعدادی آزمون مختلف بسته به نیاز طراحی و دسته‌بندی می‌شود و نهایتاً یک برنامه یا ابزار آن‌ها را اجرا می‌کند.

در طراحی آزمون واحد خوب است به برخی نکات توجه داشته باشید [۱]:

- آزمون واحد باید تا حد امکان ساده باشد و منطق پیچیده‌ای نداشته باشد.
- هر آزمون واحد باید دقیقاً یک شرط را بیازماید. در صورت شکست یک آزمون واحد باید بتوان به راحتی متوجه شد کدام عملکرد دچار مشکل است.
- آزمون‌ها نباید اثرات جانبی داشته باشند. اگر آزمون‌ها اثر جانبی نداشته باشند تکرارپذیر می‌شوند و ترتیب اجرایشان نیز بی‌اهمیت می‌گردد.
- آزمون واحد باید رفتار قابل مشاهده را بیازماید، نه ساختار داخلی کد را. اجزای خصوصی[۸] کلاس‌ها نیز معمولاً به عنوان جزئیات پیاده‌سازی در نظر گرفته می‌شوند. هرچند آزمون واحد باید کد را به خوبی پوشش دهد، آزمون‌ها نباید بیش از حد به جزئیات پیاده‌سازی گره خورده باشند تا بیش از حد شکننده نشوند. به جای فکر کردن دربارهٔ این که «آیا اگر مقادیر x و y وارد شوند، تابع ابتدا تابع a و سپس تابع b را صدا خواهد کرد و سپس مجموع نتایج را به عنوان نتیجهٔ نهایی باز می‌گرداند؟» به این فکر کنید که «آیا اگر مقادیر x و y وارد شوند، نتیجه برابر z خواهد بود؟» [۳].

طراحی موارد آزمون[۹] با پیاده‌سازی آزمون‌ها متفاوت است. در بسیاری از روش‌های انتخاب موارد آزمون مناسب، هرچند نه در همهٔ آن‌ها، به ساختار داخلی کد نیز توجه می‌شود تا موارد آزمون بحرانی و شرایط مرزی در نظر گرفته شوند و کد موجود به خوبی پوشش داده شود؛ اما با تغییر پیاده‌سازی داخلی بدون تغییر رفتار متد یا کلاس، آزمون‌های قبلی نباید شکست بخورند.

[1] unit testing
[2] unit under test
[3] pass
[4] fail
[5] time out
[6] assert
[7] Arrange, Act, Assert
[8] private
[9] test case

- کد اصلی باید فاقد منطق مربوط به تست باشد. استفاده از جملاتی مانند `if (TEST_MODE)` پیشنهاد نمی‌شود.
در صورت نیاز به دسترسی یا تغییر فیلدهای درونی کلاس‌ها می‌توانید از روش‌هایی مانند *بدل‌های آزمونی*[10] یا تعریف کلاس‌های مشتق کمک بگیرید.
- کد آزمون نیز نوعی کد است و تمام مسائل مربوط به نوشتن کد تمیز دربارهٔ آن صادق است.

۲.۱ آزمون در C++

هرچند می‌توان با استفاده از ماکروی[11] `assert` که در فایل سرآیند[12] `cassert` (و `assert.h` در C) قرار دارد آزمون طراحی کرد، معمولاً برای نوشتن آزمون از چهارچوب‌های خاصی استفاده می‌شود تا نوشتن آزمون‌های پیچیده ساده‌تر شود. در C++ نیز چنین چهارچوب‌هایی وجود دارد؛ در حال حاضر دو چارچوب Google Test و Boost.Test رایج‌ترند.

چهارچوب Catch و نسخهٔ دوم آن (Catch2) امکانات زیادی فراهم می‌کند و استفاده از آن نیز نسبتاً آسان‌تر است. به دلایل مذکور، این کتابخانه برای استفاده انتخاب شد. برای آشنایی با این کتابخانه می‌توانید به مخزن گیت‌هاب آن که در آدرس https://github.com/catchorg/Catch2 در دسترس است مراجعه کنید. مطالعهٔ مستندات پروژه از جمله صفحات آموزش مقدماتی و ادعاها برای آشنایی با کتابخانه و نحوهٔ استفاده از آن پیشنهاد می‌گردد.

۱.۲.۱ آزمودن کلاس Person

در ادامه نحوهٔ نوشتن آزمون برای یک کلاس ساده را بررسی خواهیم کرد.

فرض کنید کلاسی به نام `Person` وجود دارد که نام افراد را در خود نگه می‌دارد:

```
class Person {
private:
  std::string firstname;
  std::string lastname;

public:
  Person(std::string firstname, std::string lastname);
  std::string get_firstname() const;
  std::string get_lastname() const;
  std::string get_fullname() const;
};
```

سازندهٔ این کلاس در صورت دریافت `firstname` خالی استثنای `std::invalid_argument`[13] پرتاب می‌کند.

همچنین، این کلاس تابعی به نام `get_fullname` دارد که قرار است با به هم چسباندن نام و نام خانوادگی افراد نام کامل آن‌ها را برگرداند:

```
std::string Person::get_fullname() const { return firstname + lastname; }
```

حال کلاس آزمونی به نام `person_test` برای آزمودن کلاس `Person` می‌نویسیم:

[10] test doubles
[11] macro
[12] header file
[13] exception

```cpp
#include "Person.hpp"
#include <stdexcept>
#include <string>

#define CATCH_CONFIG_MAIN
#include "../catch.hpp"

TEST_CASE("`get_fullname` test") {
  Person person("Edsger", "Dijkstra");
  REQUIRE(person.get_fullname() == "Edsger Dijkstra");
}

TEST_CASE("constructor empty firstname test") {
  REQUIRE_THROWS_AS(Person("", "Dijkstra"), std::invalid_argument);
}
```

این کلاس شامل تعدادی مورد آزمون[۱۴] است که هر یک چیزی را می‌آزمایند. چهارچوب آزمون ــ به دلیل تعریف شدن `CATCH_CONFIG_MAIN` ــ یک تابع `main` می‌سازد که همهٔ موارد آزمون را اجرا می‌کند.

به نحوهٔ آزمون خروجی متدها و همچنین روش استفاده‌شده برای آزمودن استثناها توجه کنید.

بعد از ترجمه و اجرای همهٔ کدها مشاهده می‌شود که یکی از آزمون‌ها که مربوط به عملکرد صحیح متد `Person::get_fullname` است شکست می‌خورد:

```
$ ./person_test.out --success

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
person_test.out is a Catch v2.8.0 host application.
Run with -? for options

-------------------------------------------------------------------------------
`get_fullname` test
-------------------------------------------------------------------------------
person/person_test.cpp:8
...............................................................................

person/person_test.cpp:10: FAILED:
  REQUIRE( person.get_fullname() == "Edsger Dijkstra" )
with expansion:
  "EdsgerDijkstra" == "Edsger Dijkstra"

-------------------------------------------------------------------------------
constructor empty firstname test
-------------------------------------------------------------------------------
person/person_test.cpp:13
...............................................................................

person/person_test.cpp:14: PASSED:
  REQUIRE_THROWS_AS( Person("", "Dijkstra"), std::invalid_argument )

===============================================================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

متد مذکور را اصلاح می‌کنیم:

[14] testcase

```
std::string Person::get_fullname() const { return firstname + " " + lastname; }
```

بعد از اصلاح کلاس تحت آزمون، همهٔ آزمون‌ها با موفقیت گذارنده می‌شوند:

```
$ ./person_test.out --success

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
person_test.out is a Catch v2.8.0 host application.
Run with -? for options

-------------------------------------------------------------------------------
`get_fullname` test
-------------------------------------------------------------------------------
person/person_test.cpp:8
...............................................................................

person/person_test.cpp:10: PASSED:
  REQUIRE( person.get_fullname() == "Edsger Dijkstra" )
with expansion:
  "Edsger Dijkstra" == "Edsger Dijkstra"

-------------------------------------------------------------------------------
constructor empty firstname test
-------------------------------------------------------------------------------
person/person_test.cpp:13
...............................................................................

person/person_test.cpp:14: PASSED:
  REQUIRE_THROWS_AS( Person("", "Dijkstra"), std::invalid_argument )

===============================================================================
All tests passed (2 assertions in 2 test cases)
```

در هنگام نوشتن ادعاها برای انواع مختلف متغیرها به معنی‌دار بودن و درستی عملگرهای مقایسه‌ای که استفاده می‌کنید دقت کنید. مثلاً دقت کنید در مقایسهٔ انواع عددی ممیز شناور (`float` و `double`) عملگر `==` قابل اتکا نیست و باید از مقایسهٔ قدر مطلق اختلاف دو عدد با یک δ کوچک استفاده کرد. همچنین هنگام مقایسهٔ اشیاء کلاس‌هایی که خودتان نوشته‌اید باید عملگر مورد استفاده را سربارگذاری کرده باشید.

۲ تمرین

در این تمرین از شما انتظار می‌رود برای چند کلاس یا تابع با محوریت آزمون واحد و با استفاده از چهارچوب Catch2 آزمون بنویسید.

۱.۲ نحوهٔ طراحی و پیاده‌سازی

۱.۱.۲ کلاس `Triangle`

کلاس `Triangle` یک مثلث و برخی ویژگی‌های آن را مدل می‌کند. در پیاده‌سازی این کلاس که در اختیار شما قرار گرفته است چهار نوع اشتباه متفاوت وجود دارد. برای عملکردهای متفاوت این کلاس آزمون طراحی کنید. انتظار می‌رود در صورت وجود

هر یک از این چهار اشکال (یا هر ترکیبی از آن‌ها) حداقل یکی از آزمون‌هایی که نوشته‌اید با شکست مواجه شود و در صورت برطرف شدن همهٔ اشتباهات پیاده‌سازی آزمون‌ها با موفقیت گذرانده شوند.

اگر نیاز دارید به متغیرهای داخلی کلاس `Triangle` دسترسی پیدا کنید می‌توانید کلاسی مانند `TriangleUnderTest` از آن مشتق کنید و به این کلاس گیرنده‌ها[۱۵] یا متدهای مورد نیاز خودتان را اضافه کنید.

آزمون‌های خود و همهٔ نیازمندی‌های آن‌ها را *فقط* در یک فایل به نام `triangle_test.cpp` بنویسید. این فایل در کنار فایل‌های `catch.hpp` و `Triangle.hpp` و به همراه نسخه‌های مختلفی از فایل `Triangle.cpp` ترجمه خواهد شد و آزمون‌های شما ارزیابی خواهد گردید.

۲.۱.۲ تابع `get_avg_of_vector`

تابع `get_avg_of_vector` با دریافت یک بردار[۱۶] از اعداد صحیح، میانگین اعشاری آن‌ها را محاسبه می‌کند. علاوه بر پیاده‌سازی اصلی، چند نسخهٔ مختلف از پیاده‌سازی این تابع در اختیار شما قرار گرفته است. آزمون‌هایی برای این تابع طراحی کنید که پیاده‌سازی‌های اشتباه حداقل در یکی از آن‌ها شکست بخورند و پیاده‌سازی‌های درست همه را با موفقیت بگذرانند.

آزمون‌های خود و همهٔ نیازمندی‌های آن‌ها را *فقط* در یک فایل به نام `get_avg_of_vector_test.cpp` بنویسید. این فایل در کنار فایل‌های `catch.hpp` و `get_avg_of_vector.hpp` و به همراه نسخه‌های مختلف فایل `get_avg_of_vector.cpp` ترجمه خواهد شد و آزمون‌های شما ارزیابی خواهد گردید.

۳.۱.۲ تابع `satisfies_hailstone` اختیاری

تابع `satisfies_hailstone` عددی می‌گیرد و بررسی می‌کند آیا یک *عدد تگرگی*[۱۷] [۲] است یا نه.

یک عدد تگرگی عددی است که با شروع از آن و نوشتن دنباله‌ای با این رابطه، دنباله در نقطه‌ای به عدد ۱ برسد:

$$a_n = \begin{cases} \frac{a_{n-1}}{2} & \text{اگر } a_{n-1} \text{ زوج باشد} \\ 3a_{n-1}+1 & \text{اگر } a_{n-1} \text{ فرد باشد} \end{cases}$$

حدس کولاتز[۱۸] بیان می‌کند که همهٔ اعداد مثبت تگرگی‌اند.

سعی کنید جز حالت بدیهی ۰، مورد آزمون دیگری بیابید که تابعی که در اختیارتان قرار گرفته است در آن شکست بخورد. اگر این سؤال در زبان Python مطرح شده بود می‌توانستید برای آن پاسخی بیابید؟ آیا می‌توانید کد آزمون زبان C++ را طوری بنویسید که مستقل از معماری ماشین در حال استفاده نتیجهٔ مورد نظر را تولید کند؟

۲.۲ نحوهٔ تحویل

- فایل‌های آزمون خود را با نام‌های `AT-SID-triangle_test.cpp` و `AT-SID-get_avg_of_vector_test.cpp` در صفحهٔ CECM درس بارگذاری کنید که SID شمارهٔ دانشجویی شماست؛ برای مثال اگر شمارهٔ دانشجویی شما ۸۱۰۱۹۷۹۹۹ باشد، نام فایل‌های شما باید `AT-810197999-triangle_test.cpp` و `AT-810197999-get_avg_of_vector_test.cpp` باشد.

  نیازی به تحویل آزمون تابع `satisfies_hailstone` نیست.

[15] getter
[16] vector
[17] Hailstone number
[18] Collatz conjecture

- آزمون‌های شما باید الزاماً با استفاده از چهارچوب آزمون Catch2 نوشته شده باشند. حتماً با استفاده از `CATCH_CONFIG_MAIN` اجازه بدهید چهارچوب آزمون خودش تابع `main` را تولید کند. به نمونهٔ بارگذاری‌شده از آزمون کلاس `Person` دقت کنید.
- به فرمت و نام فایل‌های خود دقت کنید. از بارگذاری فایل فشرده خودداری کنید.
- توجه داشته باشید که با توجه به تست خودکار کدهای شما، عدم رعایت نکات مذکور ممکن است منجر به از دست دادن همهٔ نمرهٔ شما بشود.
- برنامه‌ی شما باید در سیستم‌عامل لینوکس و با مترجم g++ با استاندارد C++11 ترجمه و در زمان معقول اجرا شود.
- هدف این تمرین یادگیری شماست. لطفاً تمرین را خودتان انجام دهید. در صورت کشف تقلب مطابق قوانین درس با آن برخورد خواهد شد.

مراجع

[1] Erik Dietrich. 2014. Introduction to Unit Testing (Don't Worry, Your Secret is Safe with Me). Retrived from https://daedtech.com/introduction-to-unit-testing-dont-worry-your-secret-is-safe-with-me/.

[2] Francis E. Su, *et al.* "Hailstone Numbers." In *Math Fun Facts*. Retrived from https://www.math.hmc.edu/funfacts/ffiles/10008.5.shtml.

[3] Ham Vocke. 2018. The Practical Test Pyramid. Retrived from https://martinfowler.com/articles/practical-test-pyramid.html.

[4] Wikibooks. 2018. "Unit Tests." In *Wikibooks, The Free Textbook Project*. Retrieved from https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing/Unit_Tests.