مدرس درس: **محمدامین صادقی**
طراح: **هادی صفری**
مهلت تحویل: یکشنبه ۲ دی ۱۳۹۷، 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 [۵] مراجعه کنید.

۱.۱ طراحی آزمون‌ها

آزمون واحد معمولاً بر اساس `assert` بنا می‌شود. انتظار می‌رود `assert`ها 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[۶] مشهور است [۲، ۴]. 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 [۲]:

- 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.

[1] unit testing
[2] unit under test
[3] pass
[4] fail
[5] time out
[6] Arrange, Act, Assert

- 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[۷] 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؟» [۴].
طراحی موارد آزمون[۸] 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.
- 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*[۹] 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++

معمولاً برای نوشتن آزمون واحد از چهارچوب‌های خاصی استفاده می‌شود تا نوشتن آزمون‌های پیچیده ساده‌تر شود. در C++ نیز چنین چارچوب‌هایی وجود دارد؛ در حال حاضر دو چارچوب Google Test و Boost.Test رایج‌ترند.

اما می‌توان با استفاده از ابزارهای خود C++ نیز آزمون واحد طراحی کرد. برای این کار از ماکروی[۱۰] `assert` استفاده می‌شود. این ماکرو که در فایل سرآیند[۱۱] *cassert* (و *assert.h* در C) قرار دارد، یک ماکروی پردازشی است که مقدار یک عبارت شرطی را در زمان اجرا ارزیابی می‌کند. اگر عبارت شرطی برقرار بود، `assert` کاری نمی‌کند؛ در غیر این صورت اجرای برنامه متوقف می‌شود[۱۲] و پیام خطایی شامل عبارت شرطی متناظر، نام فایل حاوی `assert` و شمارهٔ خط `assert` در آن فایل نمایش می‌یابد[۱۳].

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

[7] private
[8] test case
[9] test doubles
[10] macro
[11] header file

[۱۲] با کد خطای ۱۳۴، معادل سیگنال *SIGABRT*
[۱۳] در جریان خطای متعارف (*stderr*)

```
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` 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 `PersonTest` برای آزمودن کلاس `Person` می‌نویسیم:

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

class PersonTest {
private:
  void get_fullname_test();
  void constructor_empty_firstname_test();

public:
  void run();
};

void PersonTest::get_fullname_test() {
  Person person("Edsger", "Dijkstra");
  assert(person.get_fullname() == "Edsger Dijkstra");
}

void PersonTest::constructor_empty_firstname_test() {
  try {
    Person person("", "Dijkstra");
    assert(false);
```

```
  } catch (std::invalid_argument) {
  }
}

void PersonTest::run() {
  get_fullname_test();
  constructor_empty_firstname_test();
}

int main(int argc, char const *argv[]) {
  PersonTest person_test;
  person_test.run();

  return 0;
}
```

این کلاس شامل تعدادی متد است که هر یک چیزی را می‌آزمایند. همهٔ آزمون‌ها در متد `Person::run` صدا شده‌اند که در `main` فراخوانی شده است. به این ترتیب همهٔ آزمون‌ها اجرا می‌شوند.

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

بعد از ترجمه و اجرای همهٔ کدها، برنامه متوقف می‌شود و خروجی زیر نمایش می‌یابد:

```
Assertion failed: (person.get_fullname()== "Edsger Dijkstra"), function get_fullname_test,
file PersonTest.cpp, line 17.
Abort trap: 6
```

با اصلاح متد `Person::get_fullname` به شکل زیر، آزمون با موفقیت گذرانده می‌شود:

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

در هنگام استفاده از `assert` 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 (`double` و `float`) 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.

می‌توان از روش‌های خلاقانه برای معنی‌دار کردن پیام خطا استفاده کرد. مثلاً استفاده از AND رایج است [۱]:

```
assert(found && "Car could not be found in database");
```

۲ تمرین

در این جلسهٔ تمرینی از شما انتظار می‌رود برای چند کلاس یا تابع با محوریت آزمون واحد 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[۱۴] یا متدهای مورد نیاز خودتان را اضافه کنید.

آزمون‌های خود و همهٔ نیازمندی‌های آن‌ها را *فقط* در یک فایل به نام *TriangleTest.cpp* بنویسید. این فایل در کنار فایل *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[۱۵] 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 *GetAvgOfVectorTest.cpp* بنویسید. این فایل در کنار فایل *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.

[14] getter
[15] vector

۳.۱.۲ تابع satisfies_hailstone (اختیاری)

تابع satisfies_hailstone عددی می‌گیرد و بررسی می‌کند آیا یک *عدد تگرگی*[۱۶] [۳] 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[۱۷] 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؟

۲.۲ نحوهٔ تحویل

- فایل‌های برنامهٔ خود را با نام‌های *TriangleTest.cpp* و *GetAvgOfVectorTest.cpp* در صفحهٔ CECM درس در بخش مربوط بارگذاری کنید.

  نیازی به تحویل آزمون تابع satisfies_hailstone نیست.

- 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 این نکات ممکن است منجر به از دست دادن همه یا بخش زیادی از 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.

[16] Hailstone number
[17] Collatz conjecture

مراجع

[1] Alex. 2017. Assert and `static_assert`. Retrieved from https://www.learncpp.com/cpp-tutorial/7-12a-assert-and-static_assert/.

[2] 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/.

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

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

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