LW: Introduction to Software Testing

Starter Code

- Starter Code

Submission

- A score of **at least 90 on Gradescope** for completion.
    - You will only submit `test_nth_root.cpp` to Gradescope.

Overview

Test-driven development (TDD) is a software development process in which you write tests for your code before actually implementing the code. Writing tests first helps you to design (and plan and think about) a program ***before*** you write it. Students who use TDD are more productive (i.e. spend less time designing, implementing, and debugging their code) than students that do not use TDD[1].

From a practical perspective, think about the errors you frequently encounter when writing new programs. You may need to support a variety of "typical" cases, but there are also edge cases - or even exceptional cases - that your code needs to handle. Edge cases can be missed until you run into one and a failure occurs. In TDD, you would have already created tests for many of these instances, and you would find such errors more quickly.

In this lab, we will practice TDD concepts by writing a variety of test cases for a provided function, and we will practice a couple simple ways to output test results.

Problem Description

You will be writing unit tests for the ***nth root*** function. In mathematics, an ***nth root*** of a number *x*, where *n* is usually assumed to be a positive integer, is a number *r* which, when raised to the power *n* yields *x*: $r^n = x$, where *n* is the degree of the root.[2] The nth root of x is written as $\sqrt[n]{x}$.

[1] Erdogmus, Hakan & Morisio, Maurizio & Torchiano, Marco. (2005). On the effectiveness of the test-first approach to programming. Software Engineering, IEEE Transactions on. 31. 226- 237. 10.1109/TSE.2005.37.

[2] https://en.wikipedia.org/wiki/Nth_root

Examples:

- $\sqrt[2]{4} = 2$
- $\sqrt[3]{-1} = -1$
- $\sqrt[\pi]{e} = 1.3748...$
- $\sqrt[7]{2025} = 2.9672...$

You will NOT be implementing the *n*th root function, only writing code to test it!

Walkthrough

Read Appendix: Requirements Engineering → Design for information about Requirements Engineering for this function. After going through the Requirements Engineering process, we know more about what this function is supposed to be and do. In summary:

- **Declaration:** `double nth_root(int n, double x);`
- **Input:** `n` := the degree of the root, x := the number from which to extract the nth root
- **Output:** $\sqrt[n]{x}$, the *n*th root of *x*, i.e. *r* such that $r^n = x$
  - The positive root if n is even.
- **Exceptions:**
  - throws `std::domain_error` if
    - *n* = 0
      - (x)^(1/0) -> undefined
    - *n* is even and *x* is negative
      - (-5)^(1/-3) -> imaginary
    - *n* is negative and *x* = 0
      - (0)^(1/-4) -> undefined

REMEMBER: DO NOT IMPLEMENT THE FUNCTION. ONLY WRITE TESTS.

Read Appendix: Starter Code Tour for information about what is in the starter code on Gradescope.

Submission

You will edit and submit exactly one file: `test_nth_root.cpp`. Open it now.

Quickstart

1. Compile it (note: there are two files to compile):

```
$ g++ -std=c++17 -Wall -Wextra -Weffc++ -pedantic -o test test_nth_root.cpp
nth_root.cpp
```

2. Run it:

```
$ ./test
JLP
JLP
nth_root(2, 1) = 0
JLP
[FAIL] (n=2, x=1)
  expected nth_root(2, 1) to be 1
  got 0
```

3. Edit the MINIMUM REQUIREMENT section of `test_nth_root.cpp` to add invocations of the function with different arguments, recompile, run, rinse, and repeat until you see every letter A - P at least once. For example, in the execution above, the program prints "JLP", which indicates that cases J, L, and P were covered by the tests invoking the `nth_root()` function.
    a. Read `nth_root.cpp` to find out how to get each letter to appear. There's a comment for each which describes the test case input that would reach it.
    b. Remember, under TDD, you want tests to cover all cases, so you need a variety of calls to `nth_root`: normal, edge, and error/exception cases.
    c. Don't worry about the output that looks like the function is incorrect, e.g. the `[FAIL]` line. The provided function is only defined to measure test coverage and will not give the correct result. You may comment out the "Try Harder" section if the output bothers you, but you may want to do that section later!

4. Submit `test_nth_root.cpp` to Gradescope. Gradescope will use a tool called "`gcov`" to check code coverage of your tests against the provided dummy nth_root function and a correct nth_root function. Your goal is to earn a score of at least 90 points.
    a. You can obtain up to 80 points just by getting all the letters to print at least once for the provided dummy nth_root.cpp file (100% coverage for that file).
    b. To reach the required 90 points, you must get your test cases to run against the correct nth_root implementation on Gradescope.
        i. Before the test will even attempt to run, you must have at least 95% coverage on the provided nth_root.cpp.
        ii. Once this Gradescope case runs, you may be surprised to see a failure occur, depending on how you wrote your unit tests.
        iii. This is because the correct solution throws exceptions, as described in the requirements appendix. The remaining points require the exceptional test cases to run without crashing, which requires that the exceptions be caught. If you are not yet familiar with exceptions, you can learn about how to catch exceptions from your TA, the zyBook, and by reading to the end of this document: Appendix: The last few points.
    c. Optionally, you can obtain 5pts from the Try Hard and Harder section described below. These cannot get you to 90pts if you don't get your tests running against the correct nth_root, but they can get you to 100% if all else is done.

Try Hard and Harder

Just invoking the function will exercise the code, but it's not actually checking correctness.

1. For the "Try Hard" points, print the return value of several invocations to check the correctness manually.
    a. Of course, for the provided dummy nth_root.cpp, it will not be correct, but if you check the output for Gradescope's test case against the actual implementation, you should see correct values.
2. For the "Try Harder" points, compare the actual value to the expected value.
    a. Again, this will fail on the dummy file but should yield [PASS] values on the correct implementation, assuming you are using the correct expected values.

Appendix: Requirements Engineering → Design

This part typically takes a while and involves some thought and discussion. Under some software development lifecycles, requirements are not set in stone and can be revisited and revised later as the project progresses. However, "good" requirements -- learn what that means in CSCE 431 (and study abroad in Singapore)! -- are crucial to designing and building high-quality software.

Below, you will see a collection of requirements and corresponding design decisions for the ***n*th root** function. An understanding of the function requirements helps you to write appropriate test cases for sufficient coverage.

`Requirement:` The function should compute $\sqrt[n]{x}$. Mathematically, the function is defined for all numbers *n* except 0. For now, though, let's limit ourselves to dealing with integer values for *n*.

`Design Decision:` The input value of *n* can be any non-zero integer.

`Design Decision:` The function shall throw the `std::domain_error` exception if $n = 0$.

`Requirement:` The value of *x* can be almost any number, but let's limit ourselves to just the real numbers. The square root of -1 is the imaginary number *i*, but we don't want to deal with complex numbers.

`Design Decision:` The input value of *x* can be any real number when *n* is odd.

`Design Decision:` The input value of *x* can be any non-negative real number when *n* is even.

`Design Decision:` The function shall throw the `std::domain_error` exception if *n* is even and $x < 0$.

`Requirement:` Since *n* can be negative, which corresponds to the inverse nth root, the function is undefined when *n* is negative and *x* is 0.

`Design Decision:` The input value of *x* can be any non-zero real number when *n* is negative.

`Design Decision:` The function shall throw the `std::domain_error` exception if $n < 0$ and $x = 0$.

`Requirement:` There are two square roots of 4: -2 and +2. We don't want to deal with multiple roots.

`Design Decision:` The output value shall be the positive root when *n* is even.

`Requirement:` Most numbers are not perfect powers, so their nth roots will have fractional parts.

`Design Decision:` The output value shall be a floating-point number.

`Requirement:` Sanity check: the output should be correct.

`Design Decision:` The output value shall be an nth root of the input *x*.

Requirement: The name of the function, the list of input parameters, and the return data type make up the *function prototype*. The function name should be descriptive.

Design Decision: The function shall be named `nth_root`.

Requirement: The function should compute $\sqrt[n]{x}$, therefore it should take two arguments: the value of *n* and the value of *x*. We already decided that *n* should be an integer and *x* should be a floating-point number.

Design Decision: The function shall have 2 parameters: `int n, double x`

Requirement: We can only have at most 1 return value. We've already decided it should be a floating-point number.

Design Decision: The function's return value shall be of type `double`.

Summary:

- The value of *n* is an integer.
- The value of *x* is a floating-point number.
- Not all combinations of inputs are valid
    - throw the `std::domain_error` exception on invalid input.
- The value of the output is a floating-point number
    - an nth root of input *x*
    - the positive root of *x* when *n* is even.
- The function's prototype is `double nth_root(int n, double x)`.

Appendix: Starter Code Tour

The provided starter code (download from Canvas) contains:

- `nth_root.h` - header file containing the function declaration, necessary during compilation to link `test_nth_root.cpp` with `nth_root.cpp`.
- `test_nth_root.cpp` - skeleton code for test cases, contains the `main()` function, this is the one and only file you will edit and submit.
- `nth_root.cpp` - contains the for-testing-purposes-only definition of the `nth_root()` function which helps to measure test coverage for this lab work.
- `test_helpers.h` - a bootleg version of Google's unit testing framework for C++.

nth_root.h

Start by reading `nth_root.h`. The lines beginning with # are *preprocessor directives*. This particular configuration is called a *header guard* and it prevents the compiler from re-declaring the function when the header file is included more than once in the program.

```
#ifndef NTH_ROOT_H
#define NTH_ROOT_H

double nth_root(int n , double x);

#endif  // NTH_ROOT_H
```

test_nth_root.cpp

Next, read `test_nth_root.cpp`. The preprocessor directives at the top tell the compiler to read a few other files to find the declarations and/or definitions of several objects and functions that are used in this program, e.g. `std::cout` is defined in `iostream` and `std::fabs` is defined in `cmath`. The `main()` function contains code blocks of code labeled MINIMUM REQUIREMENT, TRY HARD, and TRY HARDER, corresponding to different levels of effort for completing this lab work, each containing a short description of how to write tests at that level and an example of a test case.

```
#include <iostream>
#include <cmath>
#include "nth_root.h"

int main() {
    {   // MINIMUM REQUIREMENT (for this lab)
        // just call the function with various values of n and x
        nth_root(2, 1);
    }

    {   // TRY HARD
        // report the value
        double actual = nth_root(2, 1);
        std::cout << "nth_root(2, 1) = " << actual << std::endl;
    }

    {   // TRY HARDER
       // compare the actual value to the expected value
        double actual = nth_root(2, 1);
        double expected = 1;
        if (std::fabs(actual - expected) > 0.00005) {
            std::cout << "[FAIL] (n=2, x=1)" << std::endl;
            std::cout << "  expected nth_root(2, 1) to be " << expected <<
std::endl;
            std::cout << "  got " << actual << std::endl;
        } else {
            std::cout << "[PASS] (n=2, x=1)" << std::endl;
        }
    }
}
```

nth_root.cpp

Next, read `nth_root.cpp`. The `#define` directive creates a *macro* that replaces `print(X)` with the expression that follows (a C++ statement that prints X to standard output), where X is

whatever is actually between the parenthesis in the code, i.e. `print(covered)` expands to `std::cout << covered << std::endl`. The comment at the top of the function clearly indicates that this code is only for the purposes of measuring test coverage for this lab work, not an actual implementation. The body of the function contains conditional statements that check whether a test case covers a particular set of values. For every case covered, a letter is appended to a string. At the end, the string is printed. Once you get all the letters to print (in the aggregate, collecting subsets over the course of several test cases), you'll have 100% test coverage.

```
#include <iostream>
#include <string>

#define print(X) std::cout << X << std::endl

double nth_root(int n, double x) {
    // this code is here to estimate test coverage when running locally
    // it is not an implementation of the function
    // you should NOT implement this function for this labwork
    // you should NOT submit this file to Gradescope
    // you are to ONLY WRITE TEST CASES (in test_nth_root.cpp)

    std::string covered;

    if (n == 0) {
        covered += "A";  // n = 0
    } else if (n%2 == 0 && x < 0) {
        covered += "B";  // even root of a negative number
    } else if (n < 0 && x == 0) {
        covered += "C";  // negative root of 0
    } else {
        if (n == 1) {
            covered += "D";  // n = 1
        }
        if (n == -1) {
            covered += "E";  // n = -1
        }
        .
        .
        .
        covered += "P";  // valid input
    }
```

```
    print(covered);

    return 0;
}
```

`test_helpers.h` (OPTIONAL)

Peek at the contents of `test_helpers.h`. It contains several macros and functions that handle generating unit test code. The multi-line comment at the top contains skeleton code for writing unit tests within the framework. Test functions are named `test_NAME` and are invoked by calling `TEST(NAME)`. Within the test functions, the unit test commands are used, e.g. `EXPECT_EQ(X, Y)` will check that `X == Y` and update the value of the boolean variable `pass` and explain why the test failed. The `EXPECT_*` versions do not immediately return on failure, the `ASSERT_*` versions do immediately return on failure. For testing floating-point equality, use `EXPECT_NEAR` and `ASSERT_NEAR` functions, e.g. `EXPECT_NEAR(1.0/3, 0.3333)`.

NOTE: The use of `test_helpers.h` is OPTIONAL. You can complete the assignment without it. If you choose to use it, you should read Appendix: The last few points carefully.

Appendix: The last few points

Oh? You wanna know how to earn those last few points to reach the threshold? I have *exceptional*ly relevant examples for you.

The method `string.at(size_t pos)` throws an exception (`std::out_of_range`) when the argument (`pos`) is not less than the length of the string (i.e. out of bounds). You can test for this behavior using a `try/catch` block:

```
string str = "pineapple";  // length = 9
try {
    char bad_char = str.at(10);
    cout << "[FAIL] expected an exception, none thrown." << endl;
} catch (std::out_of_range) {
    cout << "[PASS] caught an exception." << variable << endl;
}
```

Or… The test framework has a macro you may be interested in: `EXPECT_THROW(X, Y)`

`X :=` some expression to evaluate, e.g. an invocation of a function which is expected to throw an exception.

`Y :=` the name of the exception that is expected to be thrown.

So our test from above can be written simply as:

```
string str = "pineapple";  // length = 9
EXPECT_THROW(str.at(10), std::out_of_range);
```

You just need to put `bool pass = true;` inside your `main` method and put `#include "test_helpers.h"` at the top of of your `test_nth_root.cpp` file:

```
#include <string>
#include "test_helpers.h"

int main() {
    bool pass = true;
    std::string str = "pineapple";  // length = 9
    EXPECT_THROW(str.at(10), std::out_of_range);
}
```

If you review the design requirements in our earlier appendix, you might find some *exceptionally* interesting cases there as well, and perhaps, that might be worth considering when your test cases run against the actual implementation… maybe you could use `EXPECT_THROW` to your benefit?