LW: Introduction to Software Testing

**A grade of "complete" on this lab work requires a score of at least 89.50%.**

**Collaboration: groups of 2 or 3**

# Problem Description

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*: , where *n* is the degree of the root.[[1]](#footnote-0) The nth root of x is written as .

Examples:

In this lab work, you will be writing unit tests for a function that computes nth roots.

**You will NOT be implementing the nth root function.**

But, even if you were writing this function, you should still write your tests first. This is part of a process known as **test-driven development (TDD)**. Writing your tests first helps you to design (and plan and think about) the 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[[2]](#footnote-1).

# 

# Walkthrough

Read [Appendix: Requirements Engineering → Design](#_frzpsh3olhmj)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:

* 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: , the *n*th root of *x*, i.e. *r* such that
  + The positive root if n is even.
* Exceptions:
  + throws std::domain\_error if
    - *n* = 0
    - *n* is even and *x* is negative
    - *n* is negative and *x* = 0

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

Read [Appendix: Starter Code Tour](#_fumcv03bmfnz) for information about what is in the starter code on Mimir.

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 -pedantic -Weffc++ test\_nth\_root.cpp nth\_root.cpp

1. Run it:

$ ./a.out

JLP

JLP

nth\_root(2, 1) = 0

JLP

[FAIL] (n=2, x=1)

expected nth\_root(2, 1) to be 1

got 0

1. 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.
   1. Don’t worry about the output which looks like the function is incorrect, e.g. the [FAIL] line. The function is only defined to measure test coverage, so of course, it’s not going to give correct results. Comment-out the lines which were originally 11 -- 28 in test\_nth\_root.cpp, if the output bothers you (you’ll want to come back to that code later).
   2. Read nth\_root.cpp to find out how to get each letter to appear. Every letter has a comment on the same line which describes the test case that will reach that line.
2. Submit test\_nth\_root.cpp to Mimir. The test cases are scripts that will check the code coverage of your tests against the dummy nth root function which was provided to you and a correct nth root function. Your goal is to earn a score of at least 89.50 points, which signifies a total of at least 89.50% coverage between the two functions.
   1. 100% coverage of the provided code may fail the correct code test (test 2 on Mimir). This is because the correct solution throws exceptions. To “fix” this, don’t cover the exceptional cases. If all non-exceptional cases are covered, the score will be 89.45 (which does not round to 89.50). The remaining points require the exceptional test cases to run without crashing, which requires that the exceptions be caught. You can learn about how to catch exceptions from your TA, the zyBook, the video modules for this week, and by reading to the end of this document.

Try Hard and Harder

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

1. Try printing the return value of every invocation to check the correctness manually.
2. Try automatically comparing the actual value to the expected value.
3. Submit to Mimir and click for details on test case 2 to see the output from a correct implementation of the nth root function.

# 

# 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! -- are crucial to designing and building high-quality software.

The function should compute . 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 .

The value of *x* can likewise 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 non-negative real number when *n* is even.

**Design Decision:** The function shall throw the std::domain\_error exception if *n* is even and .

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 and .

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.

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.

Sanity check: the output should be correct.

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

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.

The function should compute , 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

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 single 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 Mimir) contains:

* 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.
* nth\_root.h - header file containing the function declaration, necessary during compilation to link test\_nth\_root.cpp with nth\_root.cpp.
* test\_helpers.h - a bootleg version of Google’s unit testing framework for C++.
* 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.

Nobody:

Student: *“Let’s GO already (skip to the part where I get to mash the keyboard)!”*

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

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;

}

}

}

Next, read nth\_root.cpp. The #define directive creates a *macro* that replaces print(X) with the expression that follows (a C++ print statement), 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. 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 Mimir

// 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;

}

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).

Oh? You wanna know how to earn that last 0.05% (and how to earn 100%)?

I have *exception*ally relevant examples for you.

The method string.at(size\_t pos) throws an exception (specifically std::out\_of\_range) when the argument (pos) is not less than the length of the string (i.e. attempted access out of bounds). You can test for this behavior.

You can use 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." << endl;

}

Or…

The test framework has a neat macro you may be interested in using: 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);

}

1. <https://en.wikipedia.org/wiki/Nth_root> [↑](#footnote-ref-0)
2. 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. [↑](#footnote-ref-1)