Goals of this assignment

- Practice test driven development
- Design a project consisting of several components

Part 1: Test-driven development

Introduction

Test-driven development (TDD) is a software development technique where you must first write the tests that the product needs to pass and then write the code necessary to pass the tests.

Development follows an iterative process:

- Based on the project specifications, decide what functionality must be implemented first.
- Write a test for it.
- Verify that it fails. This is important, as we want the test to fail when it should.
- Write code to make the test pass. If it still fails, go back to the code and try to fix it. Only proceed when the test passes.
- If necessary, write additional tests for this functionality. For example, if you are currently testing a list insert operation, you’ll need to have tests for insertion at the beginning, middle and end of the list. It is very important to always check “border” cases (i.e. what happens at the endpoints) in addition to the most common ones.
- Once all tests for the current feature have passed, repeat the process for another feature.

You will use the testing package CppUnit to develop your project, in the same way as demonstrated in class:

- Write a test method in the FractionTests class
- Add the test to the suite as defined in the FractionTests class
• Compile and run.

For more information about assertions, look at http://cppunit.sourceforge.net/doc/lastest/group__assertions.html

The project

You will develop a simple Fraction class. A Fraction object has a numerator and a denominator which may only be specified through the constructor (no mutators). The default constructor creates the fraction \( \frac{0}{1} \). Additional operations that must be supported are:

• overloaded operator\+ that can add a fraction to a fraction, or a fraction to an integer, or an integer to a fraction or an integer to an integer. Use member functions where appropriate, global functions otherwise. Avoid friends; use accessors instead. In all cases, the result should be a fraction in reduced form. For example, \( \frac{3}{4} + \frac{7}{4} \) should result in \( \frac{5}{2} \). To reduce a fraction divide its numerator and denominator by their greatest common divisor (gcd). Euclid’s algorithm for finding the gcd of two integers is described at http://en.wikipedia.org/wiki/Euclidean_algorithm.

Division by zero should print a warning and return \( \frac{1}{0} \)

• overloaded operators for subtraction, multiplication, division. Same rules as with addition.

• equality operator. Two fractions are equal if they have the same reduced form.

Every time you run a test, save the results in a log file (log1, log2, etc.). What we are primarily interested in is the process of development, so we need to be able to examine all the steps you followed. Take care to not overwrite any log files accidentally. Make certain to write comments for each test, explaining exactly what it tests.

The sample code at /home/b11/PA6 is the same code we used at the demonstration in class. Build your project on that.

What to submit

For this part, you’ll need to submit all your log files and the final project (fraction.cpp, fraction.h, fractiontests.cpp)

Part 2: Project development

Introduction

Go to http://scv.bu.edu/cgi-bin/wcl. Read the instructions and play the game Hunt the Wumpus.

For this part of the assignment you will design (but not implement) the class hierarchy for a possible implementation of this game.

Game components

The game universe consists of
• Caves

• Tunnels. A tunnel connects two caves. Not all caves have the same number of tunnels leading out, but all caves can be reached through a tunnel.

• Players. Players can be found in caves. It is possible to have more than one player in the same cave. Players can think on their own and move from cave to cave.

• The Wumpus. The Wumpus is also in a cave and can choose to stay there or go to another cave. For more fun, let’s assume there may be more than one Wumpus in the cave system.

• Items such magic arrows which can be found in caves after a player has met with unfortunate circumstances.

• Bats. Bats can be found in caves. They always stay in the same cave, but they have a way (magic!) to transfer a player to another cave.

• Pits. Pits can be found in caves, and are of course stationary.

A player starts at a random cave and then moves around the cave network looking for the Wumpus. Things he can do include shooting arrows as described on the website, and picking up arrows from the ground. He can die if he gets hit by an arrow, gets eaten by Wumpus or falls into a pit. Keep in mind that you may want to expand the game in the future, for example by allowing the player to find food in the caves so that he doesn’t starve if his quest to kill the Wumpus takes too long. Design your project with this in mind. Be creative and document your decisions.

You are to write the .h files for this project. Explain your reasoning for the inheritance hierarchy that you have come up with, document the attributes of each type of object as well as the methods and how they interact. Try to be as detailed as possible. Good documentation is important for this project.

What to submit

For this part, you’ll need to submit your header files.

1 How to submit your work

As usual, put your files for each part in a directory called parti, make a tarball of them and submit it to b11@cs.northwestern.edu

Do not neglect to cc yourself and verify that the file was attached properly and was not corrupted in any way.