cse3150_week_5_hw

Dr. Justin Furuness

Starter Files

You are given two files:

- `main.cpp` — Code that will print out several ways of adding our fractions together. This file is already complete.
- `test.py` — Pytest tests that validate your implementation.

You will build your solution by building out the required header and implementation (cpp) files so that the main function works and the tests pass. Ideally place the cpp files in a folder called src, and the header files in a folder called include.

Assumptions

- Assume ever new will succeed, no need to guard against memory allocation failures
- Assume no integer overflow

The Assignment

In this lab, we will be building out a fraction class that behaves similar to how you'd expect a fraction to behave. Follow the steps below:

1. Start with `main.cpp`.
    - This file takes in some input to build three fractions, and then performs several operations on them.
    - Specifically, it will:
        1.1. Create a fraction with the default constructor.
        1.2. Create a fraction with the custom constructor.
        1.3. Create a fraction with the copy constructor.
        1.4. Demonstrate the move constructor.
        1.5. Demonstrate the assignment operator with an lvalue.
        1.6. Demonstrate the assignment operator with an rvalue.
        1.7. Demonstrate the addition and multiplication operators
        1.8. Demonstrate the << operator
        1.9. Demonstrate the nature of abstract classes with the ability for fractions to call the parent abstract classes convertToDecimal.

1.10. When the function ends, demonstrate the destructor.

2. Build `abstract_decimal.h`.
   - Define an abstract class for decimals.
   - Add a pure virtual function `convertToDecimal()` that outputs the object in decimal form.

3. Build `fraction.h`.
   - Make `Fraction` inherit from `Decimal`.
   - Provide the implementation for `convertToDecimal()`.
   - Add attributes `numer_` and `denom_`.
   - Store these attributes on the heap (use raw pointers). If you're thinking, wait, why on the heap? wouldn't we want these on the stack? And even if they are on the heap, why not use shared pointers? Yes on all counts, however, we want to be able to practice properly using these constructors and operators, and if we leave them on the stack then the compiler will write all of them for us, and memory management will be non existent. So we will leave them on the heap, manually managed, for this assignment.

4. In `fraction.h`, write stubs for the following:
   - Default constructor
   - Custom constructor
   - Copy constructor
   - Move constructor
     - Note: Don't let the old fraction continue to have pointers that point to the numerator and denominator that were moved somewhere else! Set them to be null pointers
     - Don't forget to manage the memory of the existing numerator and denominator before you assign new values to them
   - Assignment operator (lvalue)
   - Assignment operator (rvalue)
     - Remember don't let the old fraction continue to have pointers that point to the numerator and denominator that were moved somewhere else! Set them to be null pointers
     - Don't forget to manage the memory of the existing numerator and denominator before you assign new values to them
   - Destructor
   - Plus operator
     - If it's been a while, look up this formula!
   - Multiplication operator
   - << operator
     - Output should be similar to 4/5
   - convertToDecimal

- Don't forget, an integer divided by an integer... is an integer. What's the C++ way to convert this to something different?

5. For **every function**, add a print statement at the top:

   ```
   I am in the <function name>
   ```

   This ensures the tests can detect the function calls.

6. Implement all of the functions from `fraction.h` in `fraction.cpp`.

   - The plus, multiplication, and `<<` operators should be implemented as **friend functions**.

7. Make sure to include `#include <stdexcept>`. For any of the constructors, make sure to throw std::runtime_error("Can't divide by zero!") if the denominator is zero. Wrap main in a try catch loop and catch this error, printing "Caught a runtime error!"

8. While not in our main.cpp, there is the possibility of undefined behavior here if we move a fraction, and later try to add it to something else (in which case we will dereference a null pionter that was moved). To gaurd against this, for every function that could dereference a null pointer, at the top of that function check for null pointers and throw an std::runtime_error that says "can't dereference null pointers"

9. Compile and run your code.

10. Run the provided tests (`test.py`) to validate your implementation.

11. Ensure that you are properly freeing all dynamically allocated memory.

12. Push your code to cse_3150_week_5_hw (yes, week 5 and not week 6, we are one week behind on homeworks and this is testing week 5)