Assignment 5: Recipe Scaler

This assignment and handout are simplified versions of those written by Julie Zelenski.

One of the classic unsolved problems in cooking is scaling recipes so that you make a reasonable amount of food. Cookbook authors seem to enjoy providing recipes that serve improbable numbers of people. The problem is compounded because no one can remember how to convert among all those wacky English measurements (just how many teaspoons are in a ounce anyway?) For this assignment, you are to write a program that will do the necessarily recipe re-sizing automagically. In addition to making your life in the kitchen easier (assuming your kitchen has a C++ compiler in it), this assignment is intended to illustrate the implementation and use of C++ operator overloading.

Due Tuesday, November 28 at 11:59 p.m.

The recipe input

A set of recipes is provided in the data file `recipes.data`. Each recipe looks like this:

```
Banana Split Sarcasm Surprise
1 servings
8 ingredients
4 gal chocolate ice cream
3 gal butterscotch syrup
12 pint sliced bananas
3+1/3 cup nuts
5 peck tofu bricks
1/10 bushel whipped cream
1+1/5 qt chocolate sprinkles
1 cup maraschino cherries

Place tofu in trough. Pile ice cream on tofu. Smother in syrup.
Garnish with other ingredients. Top with cherries and sprinkles.
```

In general, the recipe format is as follows:

- The name of the recipe
- A line indicating how many servings the recipe makes
- A line indicating how many ingredients are in the recipe
- A line for each ingredient
- A blank line
- One or more lines of recipe instructions
- A blank line at the end to mark the end of the instructions

Each ingredient amount is a fraction, followed by one of these English measurement units:

```
dram, tsp, tbsp, cup, pint, qt, gal, peck, bushel, barrel, acre_ft
```

The measurement may not be in the most appropriate units to start with, but as part of your task, you will convert it (later discussed in the `Measure` class). You may assume the recipe files are otherwise correctly formatted.

The `Fraction` class

You will write a class to handle objects of `Fraction` type. In its simplest form, a fraction is just two integer values: a numerator and a denominator. `Fraction`s can be negative, may be improper (larger numerator than denominator), and can be whole numbers (denominator of 1). In order to seamlessly integrate your user-defined type with all the behavior of the usual built-in numeric types, your `Fraction` object should support a full suite of operations including:

- Construction of a fraction from two, one, or zero integer arguments. If two arguments, they are assumed to be the numerator and denominator, just one is assumed to be a whole number, and zero arguments creates a zero fraction.
- Printing a fraction to a stream with an overloaded << operator. The fraction should be printed in reduced form (not `3/6` but `1/2`). Whole numbers should print without a denominator (i.e. not `3/1` but just `3`). Improper fractions should be printed as a mixed number with a + sign between the two parts `(2+1/2)`. Negative fractions should be printed with a leading minus sign `(-3+1/7)`.
- Reading a fraction from a stream using an overloaded >> operator. You should be able to read any of the formats described above (mixed number, negative, whole numbers). Jerry's written most of this code for you, since it's annoying.
- All six of the relational operators (<,<=,>,>=,==,!=) should be supported. They should be able to compare `Fraction`s to other `Fraction`s as well as `Fraction`s to integers. Either `Fraction`s or integers can appear on either side of the binary comparison operator.
- The four basic arithmetic operations (+, -, *, /) should be supported. Again, they should allow `Fraction`s to be combined with other `Fraction`s, as well as with integers. Either `Fraction`s or integers can appear on either side of the binary operator.
- The shorthand arithmetic assignment operators (+=, -=, *=, /=) should also be implemented. `Fraction`s can appear on the left-hand side, and `Fraction`s or integers on the right-hand side.
- The increment and decrement (++, --) operators should be supported in both prefix and postfix form for `Fraction`s.

The `fracttest` program

One of the appealing benefits of object-oriented programming is the ability to test objects independently in isolation. As you develop the `Fraction` object, you want to put it through its paces by itself and fix any problems before you throw it into the fray in the recipe program. To get you started, we supply you with a fraction test program that you can use to exercise the basic functionality. It tests creating, printing, and reading `Fraction`s as well as all the relational and arithmetic operators. It has pretty good coverage of the things you need to support, although you may want to do some additional testing of your own. Don't assume that just because your `Fraction` class passes the supplied test that it's safe to proceed. As your programming your `Fraction` class, make it a point to think of edge cases that should be handled by your implementation, and when testing, add that case to the test code I've written to really strecth the Fraction class to the limits. Remember that you want your Fraction class to be unbreakable—pretend you're folding it into the C++ standard.

Additional classes

Once you have your Fraction class working and tested, you're ready to start on your recipe conversion task. Our suggestion for an object decomposition strategy is to create these three additional classes:

- The `Measure` class. The `Measure` class consists of a fractional amount and an associated English unit ("`3/4` tsp"). A `Measure` object can be scaled by multiplying by another fraction. The `Measure` class is responsible for converting the amount to the most "reasonable" unit, which is the largest unit where the amount is greater than or equal to `1` if such a unit exists. Otherwise, the unit should be the smallest known unit. For example, `3` pint should be printed as `1+1/2` qt, `1/2` gal should be printed as `2` qt (not `4` pint), and `1/3` dram remains `1/3` dram.
- The `Ingredient` class. An `Ingredient` is simply a `Measure` and ingredient name (stored as a `string`).
- The `Recipe` class which consists of a name, an array of `Ingredients`, and the instructions. Both the name and the instructions are stored as `string` objects. Rather than use an array of `string`s for the collection of instructions, use the concatenate feature of `string`s to combine the instruction lines into one longer `string` object as you read them from the file. Concatenate newlines in between the lines to retain the original formatting.

Whereas the `Fraction` class is designed to be very broad and general-purpose, these classes are a little more narrow. For example, your `Measure` class needs only to support multiplication and even though in its full generality, you would want addition and subtraction, you won't need it for this assignment and thus need not include that functionality. Each of these objects should support << and >> so they can be easily read and written using streams. Other than the functionality to support the scaling conversions, not much else is going on in these classes.

The `recipe` program

Your program should process the recipes in the file one by one. Read in the first recipe, announce the recipe name to the user, ask how many servings they would like, and then convert the recipe appropriately and print it out. If the user request 0 servings, skip printing this recipe. Repeat this process for each recipe in the file until you get to the end.

```
How many servings of Banana Split Sarcasm Surprise? 2

    Banana Split Sarcasm Surprise
    2 servings

    1 bushel chocolate ice cream
    3 peck butterscotch syrup
    1+1/2 peck sliced bananas
    1+2/3 qt nuts
    2+1/2 bushel tofu bricks
    1+3/5 gal whipped cream
    2+2/5 qt sprinkles
    1 pint maraschino cherries

    Place tofu in trough. Pile ice cream on tofu. Smother in syrup.
    Garnish with other ingredients. Top with cherries and sprinkles.

How many servings of Moon Quake Shake? 0

...
```

Table of measures

Rather than writing a bunch of special conditionals, it is a good idea to store the conversion factors in some kind of table so you can convert up and down as needed in the `Measure` object. Since this table is only used by the `Measure` class and can be shared among all `Measure` objects, this is a good opportunity for use of `static` data. By the way, the conversions are:

```
4/3 dram    = 1 tsp
3 tsp       = 1 tbsp
2 tbsp      = 1 oz
8 oz        = 1 cup
2 cup       = 1 pint
2 pint      = 1 qt
4 qt        = 1 gal
2 gal       = 1 peck
4 peck      = 1 bushel
55/7 bushel = 1 barrel
6000 barrel = 1 acre_ft
```

Some design guidelines

In addition to the ones given on your first assignment (no public data members, etc.), here is some more advice and information about what we look for:

- There will be a lot of repetition among the similar methods in the `Fraction` class (< and <= and ==, for example). However, most of these methods are quite short (1-2 lines) and it is probably not worthwhile to factor common code into helper methods when it only saves minimal work and comes with some run-time inefficiency. You might consider developing small `private inline` helper methods if it assists in readability or maintainability. Where there is more substantial repeated code (more than 2-3 lines), you definitely should factor out the common code rather than repeat it.
- There are some design tradeoffs within the `Fraction` arithmetic operators. You can implement some operations in terms of others, such as doing division by multiplying the first fraction by the reciprocal of the second, or handling += by adding into a new fraction and then assigning into self. Doing this usually will save you some coding and debugging time, but has a run-time performance cost due to the extra steps. Since there is tedium in optimizing each of the operations independently, you are free to take the easier way out if you're not interested in learning how to make tight operations.
- You will be allowed to use `friend` functions on this assignment as necessary. However, do not make things `friend` indiscriminately. For example, the binary and relational operators in the `Fraction` class should be global `friend` functions instead of class methods so that they can be used in expressions mixed with integers without regards for what type is on the left-hand side. However the shorthand arithmetic-assignment operators are only be supported for `Fraction` types on the left-hand side, and thus can be methods in the `Fraction` class instead of `friend` functions.

Getting started

In our leland class directory `/usr/class/cs193d/assignments` you'll find the project directory `hw5` which contains the testing code we give you, skeleton files for the rest, and a `Makefile` to build the project. You want to make a copy of the directory to get started. Those developing their program on the `lelands` will want to navigate to the location where they'd like a hw5 directory to be created for them. Typing:

```
cp -r /usr/class/cs193d/assignments/hw5/ .   (note the dot at the end)
```

will make a copy of the `hw5` directory (and all of its contents) from `/usr/class/cs193d/assignments` to your current location. PC and Mac students can visit the assignments page and download the files, but you'll have to build your own projects. You are free to work under an development environment you want to, but if you run into problems, you'll be on your own.

Note that the `makefile` has provisions for making two different executables (the fraction test program, and the recipe converter). Refer to the `Makefile` for a little more information about how to use the various targets.

No matter where you do your development work, when done, you must be sure your project will compile and run on the `leland` workstations. All assignments will be electronically submitted there and that is where we will compile and test your project.