Project Description:

In this project, you will be making a header-only library that defines a class that can hold recipes and a class that represents a (extremely) simple recipe app.

Here are descriptions of the only three files that should be submitted:

- recipe.h: This header file needs to contain the declaration and definition of a class named Recipe. It can also contain other functions / classes if you find them useful.
- app.h: This header file needs to contain the declaration and definition of a class named RecipeApp. It can also contain other functions / classes if you find them useful.
- utility.h: This file is never directly included by the unit tests, but you can put code needed to be included by the other files in this file if you would like.

Only the public member functions of Recipe and RecipeApp you are allowed to have are described here. Your solutions will likely have private attributes, as well as, additional functions and classes to support the described classes.

> ℹ The test cases make use of raw string literals. The R"(...)" is how to make a raw string literal. This is useful for using multiline strings in code. See: https://stackoverflow.com/questions/1135841/c-multiline-string-literal

class Recipe:

- This class should have a constructor that takes a string (representing the name of a recipe) and a integer representing the number of servings this recipe will serve. In this project, all number of servings will be positive integers. Example: `Recipe r("Microwave Popcorn", 3);`
- There should be a member function named `AddIngredient` that accepts a string with three parts: a (possibly fractional) value, a unit, and a name each separated by white space. For example: `r.AddIngredient("1/2 cup unpopped popcorn")`

  > ★ The values can consist of whole numbers (i.e. 4), fractions (i.e. 3/4), mixed fractions (i.e. 3-4/5), or even improper fractions (i.e. 9/4). All values when written to an ostream must be in reduced-mixed form (3-9/2 should become 7-1/2).
- The class should also have a member function named `SetInstructions` that accepts a (possibly multiline) string denoting the instructions to follow for the recipe. This string may have blank lines, leading whitespace, and/or trailing whitespace on each line. This whitespace should be removed when a Recipe is written to an ostream.
- The class should support the `operator<<` to write to an ostream. Please see the test cases or below for examples of the specific formatting required.
- The class should also support a function, named `IngredientInOneServing`, which accepts the name of a possible ingredient, and results a string denoting the value, unit, and name (each delimited by a space character) of the ingredient that a single serving of the recipe would require. Example, if Microwave Popcorn serves 3, and has as "1/2 cup unpopped popcorn" an ingredient, then a call like `r.IngredientInOneServing("unpopped popcorn")` would return "1/6 cup unpopped popcorn". If the recipe doesn't use that ingredient, throw a std::invalid_argument exception.
- Lastly, the class should support a member function named `ChangeServings` that accepts a positive integer denoting the number of servings the values of the ingredients should be scaled to. This member function should mutate the Recipe to reflect the change.

Here is an example of using the Recipe class:

```
   Recipe r("Microwave Popcorn", 3);
   r.AddIngredient("1/2 cup unpopped popcorn");
   r.AddIngredient("1 teaspoon vegetable oil");
   r.AddIngredient("1/2 teaspoon salt");
   r.SetInstructions(
       R"***(In a cup or small bowl, mix together the unpopped popcorn and oil.
   Pour the coated corn into a brown paper lunch sack, and sprinkle in the salt.
     Fold the top of the bag over twice to seal in the ingredients.

   Cook in the microwave at full power for 2 1/2 to 3 minutes,
    or until you hear pauses of about 2 seconds between pops.

   Carefully open the bag to avoid steam, and pour into a serving bowl.
   From: https://www.allrecipes.com/recipe/87305/microwave-popcorn/
   )***");

   std::cout << r << std::endl;

   std::ostringstream oss;
   oss << r;
   std::string expected = R"***(Recipe for: Microwave Popcorn
 Serves 3
 Ingredients:
 1/2 cup unpopped popcorn
 1 teaspoon vegetable oil
 1/2 teaspoon salt

 Instructions:
 In a cup or small bowl, mix together the unpopped popcorn and oil.
 Pour the coated corn into a brown paper lunch sack, and sprinkle in the salt.
 Fold the top of the bag over twice to seal in the ingredients.
 Cook in the microwave at full power for 2 1/2 to 3 minutes,
 or until you hear pauses of about 2 seconds between pops.
 Carefully open the bag to avoid steam, and pour into a serving bowl.
 From: https://www.allrecipes.com/recipe/87305/microwave-popcorn/

 )***";

   std::cout << expected << std::endl;

   assert(oss.str() == expected);

   std::cout << r.IngredientInOneServing("unpopped popcorn") << std::endl;

   std::cout << "Changing servings to 6" << std::endl;
   r.ChangeServings(6);

   std::cout << r << std::endl;
```

class RecipeApp:

This class represents a simple recipe manager that can save recipes and items in a pantry. And can also scale recipes to consume nearly all of a particular ingredient a customer have have.

- This class should have a default constructor.
- This class should have a member function named `AddRecipe` that accepts a Recipe and saves it in the class.
- This class should have a member function named `AddIngredientToPantry` that accepts a string denoting the value, unit, and name of an ingredient and saves it in the class.
- This class should support the `operator<<` and write itself to an ostream with both the recipes and pantry contents ordered by name.
- Lastly, this class should implement a member function named `UseUpIngredient` that accepts a string denoting the value, unit, and name of an ingredient. It should search the recipes (in order by name) and if the recipe uses that ingredient, it should return a copy of that recipe with the most amount of number of servings that the recipe can support with the indicated ingredient. If no recipe uses that ingredient, throw a `std::invalid_argument` exception.

Here is an example of using the RecipeApp class:

Copy

```
RecipeApp ra;
Recipe simple_pop("Simple Popcorn", 1);
simple_pop.AddIngredient("1/4 cup unpopped popcorn");
simple_pop.AddIngredient("1/4 teaspoon vegetable oil");
simple_pop.AddIngredient("1/4 teaspoon salt");
simple_pop.SetInstructions(R"***(Pop it!)***");

Recipe apples("An Apple", 3);
apples.AddIngredient("1-5/6 unit apple");
apples.SetInstructions(R"***(Grab it!)***");

ra.AddRecipe(simple_pop);
ra.AddRecipe(apples);
ra.AddIngredientToPantry("2 cup unpopped popcorn");
ra.AddIngredientToPantry("4-7/8 unit apple");
std::cout << ra << std::endl;

Recipe r = ra.UseUpIngredient("10 unit apple");
std::cout << "Using up apples" << std::endl;
std::cout << r;

std::cout << ra;
```

Recipe.h

```
#ifndef RECIPE_H
#define RECIPE_H

#include <iostream>
#include <iomanip>
#include <string>
using std::string;
#include <algorithm>
using std::for_each;
#include <vector>
using std::vector;
#include <sstream>
using std::istringstream;
#include <istream>
using std::istream;
using std::ostream;
#include <algorithm>
using std::for_each;
using std::endl;
#include <string.h>
using std::ostringstream;
```

```cpp
#include <ios>
using std::ios;
#include <stdio.h>      /* printf, NULL */
#include <stdlib.h>     /* srand, rand */
#include <time.h>
#include <random>
#include <stdexcept>
using std::out_of_range;
#include <iterator>
using namespace std;
#include "fractions.h"

class Recipe{
    public:
    string recipe = "";
    int servings = 0;
    string SetInstruction = "";
    map<string,Fractions> ingredients;
    map<string,vector<string>> ingredientss;
    map<string,string> units;
    map<string,Fractions> Pantry;
    map<string,string> Pantryunits;
    vector<string> ordered;
    vector<string> setin;
    stringstream interstream;

    Recipe()= default;
    Recipe(string recipe,int servings);

    void AddIngredient(string s1);
    void SetInstructions(string s1);
    string IngredientInOneServing(string s1);
    void ChangeServings(int ss);
    friend std::ostream & operator<<(std::ostream &, Recipe const &);
};

const std::string WHITESPACE = " \n\r\t\f\v";

std::string ltrim(const std::string &s)
{
    size_t start = s.find_first_not_of(WHITESPACE);
    return (start == std::string::npos) ? "" : s.substr(start);
}

std::string rtrim(const std::string &s)
{
    size_t end = s.find_last_not_of(WHITESPACE);
    return (end == std::string::npos) ? "" : s.substr(0, end + 1);
}

std::string trim(const std::string &s) {
    return rtrim(ltrim(s));
}

Recipe::Recipe(string recipes,int servingss){
```

```
    recipe = recipes;
    servings = servingss;
}
void Recipe::AddIngredient(string s1){
    stringstream ss(s1);
    int count = 1;
    int counts =1;
    string fraction;
    string unit;
    string foodname;
    string foodword;
    string foodwords;
    stringstream sss;
    while (count!=4)
    {
        string thirdval;
        if (count ==1)
        {
            ss>>fraction;
        }
        if (count ==2)
        {
            ss>>unit;
        }
        if (count >2)
        {
        // ss>>foodword;
        std::getline(ss,foodword);
        foodwords = trim(foodword);
        if (foodword.empty())
        {
            break;
        }
        if (counts == 50)
        {
            foodname = foodword;
        }
        if (counts>50)
        {
            foodname = foodname + " ";
        }
        foodword = "";
        }
        count ++;
    }
    Fractions f1(fraction);
    ingredients.insert({foodwords,f1});
    units.insert({foodwords,unit});
    ingredientss.insert({foodwords,{fraction,unit}});
    ordered.push_back(foodwords);
}
void Recipe::SetInstructions(string s1){
    stringstream ss(s1);
    string newwords;
    while (std::getline(ss,newwords, '\n'))
```

```
    {
        if (newwords.empty())
        {
            continue;
        }
        string finalwords = trim(newwords);
        setin.push_back(finalwords);
    }
}
string Recipe::IngredientInOneServing(string s1){
    auto it = ingredientss.find(s1);
    if (it != ingredientss.end()){
    string fraction = ingredientss.at(s1).at(0);
    string unit = ingredientss.at(s1).at(1);
    Fractions frac(fraction);
    frac.divint(servings);
    string final = frac.fractostr();
    string finalss = final +" "+ unit+" " + s1;
    return finalss;
    }
    else
    {
        throw std::invalid_argument("");
    }


    // string fraction = ingredientss.at(s1).at(0);
    // string unit = ingredientss.at(s1).at(1);
    // Fractions frac(fraction);
    // frac.divint(servings);
    // string final = frac.fractostr();
    // string finalss = final +" "+ unit+" " + s1;
    // return finalss;
}
void Recipe::ChangeServings(int ss){
    for(auto var : ordered)
    {
        auto fraction = ingredientss.at(var).at(0);
        Fractions simpl(fraction);
        simpl.divint(servings);
        simpl.multint(ss);
        string reduced = simpl.fractostr();
        ingredientss.at(var).at(0) = reduced;
    }
    servings = ss;
}


std::ostream & operator<<(std::ostream & out, Recipe const & Re){
    stringstream ssss;
    ssss<<"Recipe for: "<<Re.recipe<<"\n";
    ssss<<"Serves "<<Re.servings<<"\n";
    ssss<<"Ingredients:"<<"\n";
    int count = 0;
    for(auto var : Re.ordered)
```

```
    {
        auto fraction = Re.ingredientss.at(var).at(0);
        Fractions simpl(fraction);
        string reduced = simpl.fractostr();
        auto unit = Re.ingredientss.at(var).at(1);
        ssss<<reduced<<" "<<unit<< " "<<var<<"\n";
    }
    // for(auto var : Re.ingredientss)
    // {
    //     string name = var.first;
    //     vector<string> vects;
    //     vects = var.second;
    //     string fraction = var.second.at(0);
    //     string unit= var.second.at(1);
    //     ssss<<fraction<<" "<<unit<< " "<<name<<"\n";
    // }
    ssss<<"\n"<<"Instructions:"<<"\n";
    for(auto var : Re.setin)
    {
        ssss<<var<<"\n";
    }

    ssss<<"\n";
    string final= ssss.str();
    out<<final;
    return out;
}

#endif
```

fractions.h

```
#ifndef FRACTIONS_H
#define FRACTIONS_H


#include <iostream>
#include <iomanip>
#include <string>
using std::string;
#include <algorithm>
using std::for_each;
#include <vector>
using std::vector;
#include <sstream>
using std::istringstream;
#include <istream>
using std::istream;
using std::ostream;
#include <algorithm>
using std::for_each;
using std::endl;
#include <string.h>
using std::ostringstream;
#include <ios>
using std::ios;
```

```
#include <stdio.h>      /* printf, NULL */
#include <stdlib.h>     /* srand, rand */
#include <time.h>
#include <random>
#include <stdexcept>
using std::out_of_range;
#include <iterator>
using namespace std;

struct Fractions{
public:
    Fractions()= default;
    Fractions(string fracvalue);
    void reduce();
    void add(Fractions);
    void sub(Fractions);
    void div(Fractions);
    void mult(Fractions);
    void divint(int);
    void multint(int);
    string fractostr();
    Fractions &operator=(Fractions const &);
    int num = 0;
    int den = 1;
    int whole = 0;
    string aa = "";

    friend int gcd(int,int);
    friend ostream &operator<<(ostream &, Fractions const &);
};
Fractions &Fractions::operator=(Fractions const&f){
    num = f.num;
    den = f.den;
    whole = f.whole;
    return *this;
}
string Fractions::fractostr(){
    stringstream out;
    Fractions firstval("0/1");
    firstval.num = num;
    firstval.den = den;
    firstval.whole = firstval.num / firstval.den;
    firstval.num = firstval.num % firstval.den;
    firstval.reduce();
    if (firstval.whole == 0)
    {
        out<<firstval.num<<"/"<<firstval.den;
    }
    else if ((firstval.num == 0) &&(firstval.den == 1))
    {
        out<<firstval.whole;
    }
    else {
        out<<firstval.whole<<"-"<<firstval.num<<"/"<<firstval.den;
    }
```

```
    string final = out.str();
    return final;
}
ostream& operator<<(ostream &out, Fractions const&f){
    Fractions firstval("0/1");
    firstval.num = f.num;
    firstval.den = f.den;
    firstval.whole = firstval.num / firstval.den;
    firstval.num = firstval.num % firstval.den;
    if (firstval.whole == 0)
    {
        out<<firstval.num<<"/"<<firstval.den;
    }
    else if ((firstval.num == 0) &&(firstval.den == 1))
    {
        out<<firstval.whole;
    }
    else {
        out<<firstval.whole<<"-"<<firstval.num<<"/"<<firstval.den;
    }
    return out;
}
void Fractions::add(Fractions f2)
{
    num = (num * f2.den) + (f2.num *den);
    den= den * f2.den;
    reduce();  //don't forget to reduce!
}
void Fractions::sub(Fractions f2){
    num = (num * f2.den) - (f2.num *den);
    den= den * f2.den;
    reduce();  //don't forget to reduce!
}
void Fractions::mult(Fractions f2){
    num = num *f2.num;
    den= den * f2.den;
    reduce();  //don't forget to reduce!
}
void Fractions::div(Fractions f2){
    num = num *f2.den;
    den= den * f2.num;
    reduce();  //don't forget to reduce!
}
void Fractions::divint(int s1){
    den = den * s1;
    reduce();
}
void Fractions::multint(int s2){
    num = num * s2;
    reduce();
}
void Fractions::reduce()
{
    int divisor = gcd(num, den);
    num /= divisor;
```

```
    den /= divisor;
}
int gcd(int a, int b)
{
    if(b == 0)
        return a;
    else
        return gcd(b, a%b);
}
Fractions::Fractions(string fracvalue){
    string b = fracvalue;
    string wholenum = "";
    string nums = "";
    string dems= "";
    string placement;
    string::iterator dash = find(fracvalue.begin(),fracvalue.end(),'-');
    string::iterator slash = find(fracvalue.begin(), fracvalue.end(),'/');
    if ((slash != fracvalue.end()) && (dash != fracvalue.end()))
    {
        std::for_each(fracvalue.begin(),dash,[&wholenum](auto fracvalue){
            char b = fracvalue;
            wholenum.push_back(b);
        });
        std::for_each(dash+1,slash,[&nums](auto fracvalue){
            char b = fracvalue;
            nums.push_back(b);
        });
        std::for_each(slash+1,fracvalue.end(),[&dems](auto fracvalue){
            char b = fracvalue;
            dems.push_back(b);
        });
        whole = stoi(wholenum);
        num = stoi(nums);
        den = stoi(dems);
    }
    else if (slash != fracvalue.end())
    {
        std::for_each(fracvalue.begin(),slash,[&nums](auto fracvalue){
            char b = fracvalue;
            nums.push_back(b);
        });
        std::for_each(slash+1,fracvalue.end(),[&dems](auto fracvalue){
            char b = fracvalue;
            dems.push_back(b);
        });
        num = stoi(nums);
        den = stoi(dems);
    }
    else
    {
        std::for_each(fracvalue.begin(),fracvalue.end(),[&wholenum](auto fracvalue){
            char b = fracvalue;
            wholenum.push_back(b);
        });
        whole = stoi(wholenum);
```

```
    }
    int newnum = whole *den;
    num = newnum + num;
    whole = 0;
}

#endif
```