Lab 6 - C Structs

Background - Using structs to Represent Fractions

A fraction is a rational number expressed in the form *a*/*b*, where *a* (the numerator) and *b* (the denominator) are integers.

Here is the declaration for a C struct that represents fractions:

```
typedef struct {
    int num;
    int den;
} fraction_t;
```

The struct has two members, both of type `int`. Member `num` is the fraction's numerator, and member `den` is the fraction's denominator.

To declare a variable named `fr1` that can store a fraction, we use `fraction_t` as the variable's type:

```
fraction_t fr1;
```

A struct's members can be initialized individually; for example, these statements initialize the `num` and `den` members of `fr1` so that it represents the fraction 1/3:

```
fr1.num = 1;
fr1.den = 3;
```

Modern versions of C (C99, C11 and C17) let us use *compound literals* to initialize structs. The two assignment statements can be replaced by a single statement, which assigns 1 to `fr1.num` and 3 to `fr1.den`:

```
fr1 = (fraction_t) {1, 3};
```

The variable declaration and initialization can be combined into a single statement:

```
fraction_t fr2 = {1, 3};
```

When we do this, there's no need to cast the initializer list to "type" `fraction_t`.

General Requirements

Your functions must not be recursive. Repeated actions must be implemented using C's `while`, `for` or `do-while` loop structures.

None of the functions you write should perform console input; for example, contain `scanf` statements. None of your functions should produce console output; for example, contain `printf` statements.

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). Instructions were given in the VS Code installation instructions.

Finish each exercise (i.e., write the function and verify that it passes all its tests) before you move on to the next one. Do not leave testing until after you have written all the functions.

Getting Started

Step 1: Depending on your operating system, download (**lab6_win.zip** or **lab5_mac.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Double click on lab6.code-workspace

Step 3: lab6_xxx folder contains the following files:

- fraction.c contains incomplete definitions of several functions you have to design and code.
- fraction.h contains the declaration of the `fraction_t` struct, struct and the function prototypes for the functions you'll implement. **Do not modify fraction.h.**
- main.c and sput.h implement a *test harness* (functions that will test your code, and a `main` function that calls these test functions). **Do not modify `main` or any of the test functions.**

Step 4: Run the project. It should build without any compilation or linking errors.

Step 5: Execute the project. For the functions where the test harness is provided (the functions in main.c) will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions.

Step 6: Open fraction.c in the editor. Do Exercises 1 through 8. Remember, you must not make any changes to main.c, fraction.h or sput.h. All the code you write must be in fraction.c.

Exercise 1

File fraction.c contains an incomplete definition of a function named `print_fraction`. Read the documentation for this function and complete the definition.

Test suite #1 exercises `print_fraction`, but it cannot verify that the information printed by the function is correct. Instead, it displays what a correct implementation of `print_fraction` should print (the *expected output*), followed by the *actual output* from your implementation of the function.

Use the console output printed by the test harness to help you identify and correct any flaws. Verify that `print_fraction` passes all the tests in test suite #1 before you start Exercise 2.

Exercise 2

The *greatest common divisor* of two integers *a* and *b* is the largest positive integer that evenly divides both values. Here is Euclid's algorithm for calculating greatest common divisors, which uses iteration and calculation of remainders:

1. Store the absolute value of *a* in *q* and the absolute value of *b* in *p*.
2. Store the remainder of *q* divided by *p* in *r*.
3. while *r* is not 0:
    i. Copy *p* into *q* and *r* into *p*.
    ii. Store the remainder of *q* divided by *p* in *r*.
4. *p* is the greatest common divisor.

File fraction.c contains an incomplete definition of a function named `gcd`. Read the documentation for this function and complete the definition, using Euclid's algorithm. The C standard library has functions for calculating absolute values, which are declared in stdlib.h. You can find the functions available in stdlib.h on the following link: https://cplusplus.com/reference/cstdlib/

Use the console output to help you identify and correct any flaws. Verify that `gcd` passes all the tests in test suite #2 before you start Exercise 3.

Exercise 3

A *reduced fraction* is a fraction *a*/*b* written in lowest terms, which is obtained by dividing the numerator and denominator by their greatest common divisor. For example, 2/3 is the reduced form of 8/12. For our purposes, we will also include the following in our definition of a reduced fraction:

- If the numerator is equal to 0, the denominator is always 1. For example, 0/1 is the reduced form of 0/7.
- If the numerator is not equal to 0, the denominator is always a positive integer, and the numerator is a positive or negative integer. For example, -1/4 is the reduced form of 2/-8, and 3/5 is the reduced form of -9/-15.

File fraction.c contains an incomplete definition of a function named `reduce`. Read the documentation for this function, carefully, and complete the definition. **`reduce` must call the `gcd` function you wrote in Exercise 2.**

Use the console output to help you identify and correct any flaws. Verify that `reduce` passes all the tests in test suite #3 before you start Exercise 4.

Exercise 4

If we initialize fractions this way:

```
fraction_t fr;
fr.num = 2;
fr.den = -8;
```

or this way:

```
fraction_t fr = {0, 7};
```

the fraction is not in reduced form.

Programs that use the `fraction_t` type will be more robust if we implement a function named `make_fraction` that takes a numerator and a denominator as arguments and returns an initialized, reduced fraction; for example,

```
fraction_t fr;
fr = make_fraction(2, -8);

/* fr.num will be -1 and fr.den will be 4;
 * that is, fr represents the reduced fraction -1/4.
 */
```

File fraction.c contains an incomplete definition of a function named `make_fraction`. Read the documentation for this function, carefully, and complete the definition. **`make_fraction` must call the `reduce` function you wrote in Exercise 3.**

Use the console output to help you identify and correct any flaws. Verify that `make_fraction` passes all the tests in test suite #4 before you start Exercise 5.

Exercise 5

File fraction.c contains an incomplete definition of a function named `add_fractions` that is passed two fractions and returns their sum. Read the documentation for this function, carefully, and complete the definition. The fraction returned by this function must be in reduced form. (Hint: the fraction returned by `make_fraction` is always in reduced form.)

The sum of fractions $\frac{a}{b}$ and $\frac{c}{d}$ is not calculated as $\frac{a+c}{b+d}$ (despite what some people think!) If you don't remember the formula for adding fractions, look at this page:

http://mathworld.wolfram.com/Fraction.html

Use the console output to help you identify and correct any flaws. Verify that `add_fractions` passes all the tests in test suite #5 before you start Exercise 6.

Exercise 6

File fraction.c contains an incomplete definition of a function named `multiply_fractions` that is passed two fractions and returns their product. Read the documentation for this function, carefully, and complete the definition. The fraction returned by this function must be in reduced form. (Hint: the fraction returned by `make_fraction` is always in reduced form.)

Use the console output to help you identify and correct any flaws. Verify that `multiply_fractions` passes all the tests in test suite #6.

Wrap-up

Submit fraction.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace* (lab3 file). Remember: submit early, submit often.
- Make sure that fraction.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you are submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

Homework Exercise - Visualizing Program Execution

You are expected to understand and draw diagrams that depict the execution of short C programs that use structs, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills.

1. Click on this link to open C Tutor.
2. Copy/paste your solutions to Exercises 2 through 6 into the C Tutor editor.
3. Write a short `main` function that uses all the functions you have defined in this lab.
4. *Without using C Tutor,* trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before the `return` statements in each of the functions. Use the same notation as C Tutor.
5. Use C Tutor to trace your program one statement at a time, stopping just before each `return` statement is executed. Compare your diagrams to the visualization displayed by C Tutor. If C Tutor complains that your program is too long, delete the comments above the function definitions.
6. Do not submit your solutions for this exercise.