Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2019

Lab 5 - C Structs

Demo/Grading

After you finish the exercises, a TA will review your solutions, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will grade the work you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before your next lab.**

Prerequisite Reading

How to Think Like a Computer Scientist, C Version 1.09, Chapter 9 (*Structures*). The URL for this book is in the course outline.

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;
```

This 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 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 (e.g., C99 and C11) let us use *compound literals* to initialize structs, so the previous two assignment statements can be replaced by:

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

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

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

Notice that there's no need to "cast" the brace-enclosed initializer list.

General Requirements

You have been provided with four 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, as well as the declarations (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.**

Do not use arrays or pointers. They aren't necessary for this lab.

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your functions should produce console output; i.e., 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 for selecting the formatting style and formatting blocks of code are in the Lab 1 handout.

Finish each exercise (i.e., write the function and verify that is passes all its tests) before you move on to the next one. Don't leave testing until after you've written all the functions.

Getting Started

Step 1: Launch Pelles C and create a new project named fraction. (Instructions for creating projects are in the handout for Lab 1.) If you're using the 64-bit edition of Pelles C, select Win 64 Console program (EXE) as the project type. If you're using the 32-bit edition of Pelles C, select Win32 Console program (EXE). **Don't click the icons for Console application wizard, Win32 Program (EXE) or Win64 Program (EXE) - these are not correct types for this project.**

Step 2: Download main.c, fraction.c, fraction.h and sput.h from cuLearn. Move these files into your fraction folder.

Step 3: Add main.c and fraction.c to your project. (Instructions for doing this are in the handout for Lab 1.) You don't need to add fraction.h and sput.h to the project. Pelles C will do this after you've added main.c.

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

Step 5: Execute the project. The test harness (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 the harness tests.

Step 6: Open fraction.c in the editor.

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.

Build the project, correcting any compilation errors, then execute the project. The test harness will run.

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 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.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. 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, by dividing the numerator and denominator by their greatest common divisor. For example, 2/3 is the reduced fraction of 8/12.

For our purposes, we'll also include the following in our definition of a reduced fraction:

- if the numerator is equal to 0, the denominator is always 1;
- if the numerator is not equal to 0, the denominator is always positive and the numerator can be positive or negative.

File fraction.c contains an incomplete definition of a function named `reduce`. Read the documentation for this function, carefully, and complete the definition. **Your `reduce` function must call the `gcd` function you wrote in Exercise 2.** (Hint: the C standard library has functions for calculating the absolute value of an integer; these functions are declared in stdlib.h. Use the Pelles C Help > Contents command to learn about these functions.)

Build the project, correcting any compilation errors, then execute the project. The test harness will run. 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

Initializing fractions this way:

```
fraction_t fr;
fr.num = 1;
fr.den = 4;
```

or this way:

```
fraction_t fr = {1, 4};
```

is prone to error (what if we forget to put the fraction in reduced form?)

Programs that use the `fraction_t` type will be more robust if we implement a function that is 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 has numerator -1, denominator 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. **This function must call the `reduce` function you wrote in Exercise 3.**

Build the project, correcting any compilation errors, then execute the project. The test harness will run. 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

Build the project, correcting any compilation errors, then execute the project. The test harness will run. 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 starts on the following page.

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.)

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Verify that `multiply_fractions` passes all the tests in test suite #6.

Wrap-up

1. Remember to have a TA review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the grading/sign-out sheet.

2. Remember to backup your project folder before you leave the lab; for example, copy it to a flash drive and/or a cloud-based file storage service. All files you've created on the hard disk will be deleted when you log out.

Homework Exercise - Visualizing Program Execution

In the midterm and final exams, you will be expected to be able to 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. The *Labs* section on cuLearn has a link, Open C Tutor in a new window. Click on this link.

2. Copy/paste your solutions to Exercises 2 through 6 into the C Tutor editor.

3. Write a short `main` function that calls `make_fraction` to create two fractions, then calls `add_fractions` and `multiply_fractions` to add and multiply the fractions.

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 `make_fraction`, `reduce`, `gcd`, `add_fractions` and `multiply_fractions` are executed. 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.