Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2014

Lab 5 - Structures

Objective

To develop a module that supports math with fractions, using `structs` as the underlying data structure.

Attendance/Demo

To receive credit for this lab, you must make the effort to complete the exercises and demonstrate the code you complete. Also, you must submit your lab work to cuLearn by the end of the lab period. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review your code. For those who don't finish early, a TA will ask you to demonstrate whatever exercises you've completed, starting about 30 minutes before the end of the lab period. Finish any exercises that you haven't completed by the end of the lab on your own time.

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 of a C structure that represents fractions:

```
struct fraction {
    int num;
    int den;
};
```

The `struct fraction` clause declares that the name of this structure's *tag* is `fraction`. The structure has two members, both of type `int`. Member `num` is the fraction's numerator and member `den` is the fraction's denominator.

It is important to remember that a structure declaration does not declare a variable or reserve any space in memory. To declare a variable named `fr` that can store a fraction, we use the `struct` reserved word followed by the structure tag (`fraction`) to specify the variable's type:

```
struct fraction fr;
```

To initialize this fraction to 1/3, we must initialize both the `num` and `den` members:

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

C programmers often declare structure "types" using a `typedef` statement. This statement:

```
typedef struct fraction fraction_t;
```

simply declares that the identifier `fraction_t` is a shorthand for `struct fraction`. We can then use `fraction_t` Instead of `struct fraction` in variable and parameter declarations; for example,

```
fraction_t fr;
```

We can combine the structure declaration and the `typedef` statement:

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

General Requirements

In Exercises 1 through 6, you are going to define functions that operate on structures that represent fractions. Finish each exercise (i.e., write the function and verify that is passes all of its tests) before you move on to the next one. Don't leave testing until after you've written all your functions.

When writing the 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 have been provided with three files:

- fraction.c contains incomplete definitions of several functions you have to design and code.;
- fraction.h contains declaration (function prototypes) for the functions you'll implement. **Do not modify fraction.h.**
- main.c contains 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.**

Instructions

1. Create a new folder named Lab 5.
2. Launch Pelles C and create a new Pelles C project named fraction inside your Lab 5 folder. The project type must be Win32 Console program (EXE). After creating the project, you should have a folder named fraction inside your Lab 5 folder (check this).
3. Download file main.c, fraction.c, fraction.h and sput.h from cuLearn. Move these files into your fraction folder.

4. You must also add main.c and fraction.c to your project: from the menu bar, select Project > Add files to project... In the dialogue box, select main.c, then click Open. An icon labelled main.c will appear in the Pelles C project window. Repeat this for fraction.c.

   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.

5. Build the project. It should build without any compilation or linking errors.

6. Execute the project. The test harness will report several errors as it runs, which is what we'd expect, because you haven't started working on the functions that the harness tests.

7. Open main.c in the editor. Design and code the functions described in Exercises 1 through 6.

Exercise 1

File fraction.c contains the incomplete definition of a function named `print_fraction`. Read the documentation for this function and implement it. File main.c contains a function that exercises `print_fraction`.

Build your project, correcting any compilation errors, then execute the project.

The test function for Exercise 1 does not determine if `print_fraction` 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. You have to compare the expected and actual output to determine if your function is correct.

Look at the console output and verify that your `print_fraction` function is correct 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 the incomplete definition of a function named `gcd`. Read the documentation for this function and implement it, using Euclid's algorithm.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your `gcd` function passes all the tests in the test suite 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 the incomplete definition of a function named `reduce`. Read the documentation for this function, carefully, and implement it. **Your reduce function must call the `gcd` function you wrote in Exercise 2.** (Hint: the C standard library has functions for calculating absolute values, which are declared in stdlib.h. Use the Pelles C online help to learn about these functions.) Your reduce function must call the `gcd` function you wrote for Exercise 3.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Look at the console output, and verify that your `reduce` function passes all the tests in the test suite before you start Exercise 4.

Exercise 4

Initializing fractions this way:

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

is prone to error (what if we forget to initialize one of the members?) Programs that use the `fraction_t` type would be more robust if we could pass the values for a numerator and a denominator to a function that returns an initialized fraction; for example,

```
fraction_t fr;
fr = make_fraction(1, 4);
```

File fraction.c contains the incomplete definition of a function named `make_fraction`. Read the documentation for this function, carefully, and implement it. **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. Look at the console output, and verify that your `make_fraction` function passes all the tests in the test suite before you start Exercise 5.

Exercise 5

File fraction.c contains the 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 implement it. The fraction returned by this function must be in reduced form. (Hint: the fraction returned by `make_fraction` is always in reduced form.)

Note that (despite what some students think!) $\frac{a}{b} + \frac{c}{d}$ is not calculated as $\frac{a+c}{b+d}$.

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. Look at the console output, and verify that your `add_fractions` function passes all the tests in the test suite before you start Exercise 6.

Exercise 6

File fraction.c contains the 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 implement it. 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. Look at the console output, and verify that your `multiply_fractions` function passes all the tests in the test suite.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises. Make sure you initial the signout sheet after the TA has recorded your results..

2. The next thing you'll do is package the project in a ZIP file (compressed folder). From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. Click Save. Pelles C will create a compressed (zipped) folder named fraction.zip, which will contain copies of the the source code and several other files associated with the project. (The original files will not be removed). The compressed folder will be stored in your project folder (i.e., folder fraction).

3. Log in to cuLearn, click the Submit Lab 5 link and submit fraction.zip. After you click the Add submission button, drag the file to the File submissions box. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your file is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button. Once you're sure that you don't want to make any changes, click the Submit assignment button. This will change the submission status to "Submitted for grading". **Note: after you've clicked the Submit assignment button, you cannot resubmit the file.**