

Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2017

Lab 5 - Structures

Attendance/Demo

To receive credit for this lab, you must demonstrate your solutions to the exercises. When you have finished all the exercises, call a TA, who will review your code, ask you to run the test harness provided on cuLearn, and assign a grade. For those who don't finish early, a TA will ask you to demonstrate the functions 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.**

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 structure that represents fractions:

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

The structure has two members, both of type `int`. Member `num` is the fraction's numerator and member `den` is the fraction's denominator. The `typedef` declares `fraction_t` as the name of the structure's "type".

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 `fraction_t` "type":

```
fraction_t fr;
```

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

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

General Requirements

In Exercises 1 through 6, you are going to define functions that operate on structures that represent fractions.

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` structure, 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.**

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 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). Pelles C makes it easy to do this - instructions were provided in Labs 1 and 2.

Finish each exercise (i.e., write the function and verify that it 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.

Getting Started

1. Launch Pelles C and create a new project named `fraction`.
 - If you're using the 64-bit edition of Pelles C, the project type should be **Win 64 Console program (EXE)**. (Although the 64-bit edition of Pelles C can build 32-bit programs, you may run into difficulties if you attempt to use the debugger to debug 32-bit programs.)
 - If you're using the 32-bit edition of Pelles C, the project type should be **Win32 Console program (EXE)**.

When you finish this step, Pelles C will create a folder named `fraction`.

2. Download file `main.c`, `fraction.c`, `fraction.h` and `sput.h` from cuLearn. Move these files into your `fraction` folder.
3. You must also add `main.c` and `fraction.c` to your project. To do this:
 - Select **Project > Add files to project...** from the menu bar.
 - 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 step 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`.

4. Build the project. It should build without any compilation or linking errors.
5. 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 the harness tests.
6. Open `fraction.c` in the editor. Do 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.

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

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. You have to compare the expected and actual output and determine if your function is correct.

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 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 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.)

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 test suite #3 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 should 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 test suite #4 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 $\frac{a}{b} + \frac{c}{d}$ 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. Look at the console output, and verify that your `add_fractions` function passes all the tests in test suite #5 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 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 back up 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 final exam, 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 initialize 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.