

Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Fall 2013

Lab 8 - 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 a reasonable number of the exercises and demonstrate the code you complete.

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 - Fractions and Structures

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. If we require a variable named `fr` that stores a fraction, we declare it this way, using 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 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.

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`.**
- `test_fraction.c` contains a *test harness* consisting of 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. Launch Pelles C and create a new Pelles C project named `fraction`. The project type must be Win32 Console program (EXE). You should now have a folder named `fraction`.
2. Download file `test_fraction.c`, `fraction.c` and `fraction.h` from cuLearn. Move these files into your `fraction` folder. **You must also add `test_fraction.c` and `fraction.c` to your project:** from the menu bar, select `Project > Add files to project...`. In the dialogue box, select `test_fraction.c`, then click `Open`. An icon labelled `test_fraction.c` will appear in the Pelles C project window. Repeat this for `fraction.c`. Pelles C will automatically add `fraction.h` to the

project.

3. Build the project. It should build without any compilation or linking errors.
4. Execute the project. The test harness will run. Read the console output carefully. The harness will report several errors while it tests the functions in `fraction.c`, which is what we'd expect, because you haven't started working on these functions.

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 `test_fraction.c` contains a function that exercises `print_fraction`. Build your project, execute the program, and verify that the output produced by `print_fraction` is correct.

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 your project, execute the program, and verify that the results returned by `gcd` are correct.

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. (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 your project, execute the program, and verify that the results returned by

reduce are correct.

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?) Our code 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 `reduce`. Build your project, execute the program, and verify that the results returned by `make_fraction` are correct.

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 your project, execute the program, and verify that the results returned by `add_fractions` are correct.

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 your project, execute the program, and verify that the results returned by `multiply_fractions` are correct.

Posted: Monday, November 4, 2013