Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2018

Lab 5 - Structures and Pointers

After you finish all the exercises, call a TA, who 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.**

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` structure, as well as 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. 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 - instruction were provided in the handouts for Labs 1 and 2.

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.

Getting Started

Step 1: Launch Pelles C and create a new Pelles C project named fraction_pointer.

- 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_pointer.

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

Step 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 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.

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. Do Exercises 1 through 6. Don't make any changes to main.c, fraction.h or sput.h. All the code you'll write must be in fraction.c.

Exercise 1

File fraction.c contains the incomplete definition of a function named `print_fraction`. Notice that the function's argument is a pointer to a `fraction_t` structure. Read the documentation for this function and complete the definition.

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.

Review the console output, compare the expected and actual output and verify that your `print_fraction` function is correct before you start Exercise 2.

Exercise 2

File fraction.c contains the incomplete definition of a function named `gcd`. Read the documentation for this function and complete the definition, using Euclid's algorithm. You can reuse the function you wrote during Lab 4.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output and verify that your `gcd` function passes all the tests in test suite #2.

Exercise 3

File fraction.c contains the incomplete definition of a function named `reduce`. In Lab 4, the header for this function was:

```
fraction_t reduce(fraction_t f)
```

For this lab, the function header has been changed to:

```
void reduce(fraction_t *pf)
```

The function's argument is now a pointer to a `fraction_t` structure and the function's return type is now `void`. This means that this lab's implementation of `reduce` will not return a reduced fraction. Instead, it will reduce the fraction pointed to by parameter `pf`.

Read the documentation for `reduce`, 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 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. Review the console output and verify that your `reduce` function passes all the tests in test suite #3.

Exercise 4

File fraction.c contains the incomplete definition of a function named `make_fraction`. In Lab 4, the header for this function was:

```
fraction_t make_fraction(int a, int b)
```

For this lab, the function header has been changed to:

```
void make_fraction(int a, int b, fraction_t *new_fraction)
```

Read the documentation for `make_fraction`, 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. Review the console output and verify that your `make_fraction` function passes all the tests in test suite #4.

Exercise 5

File fraction.c contains the incomplete definition of a function named `add_fractions`. In Lab 4, the header for this function was:

```
fraction_t add_fractions(fraction_t f1, fraction_t f2)
```

For this lab, the function header has been changed to:

```
void add_fractions(const fraction_t *pf1, const fraction_t *pf2,
                   fraction_t *sum)
```

Read the documentation for `add_fractions`, carefully, and complete the definition. The fraction produced by this function must be in reduced form. (Hint: the fraction created by `make_fraction` is always in reduced form.)

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output and verify that your `add_fractions` function passes all the tests in test suite #5.

Exercise 6

File fraction.c contains the incomplete definition of a function named `multiply_fractions`. In Lab 4, the header for this function was:

```
fraction_t multiply_fractions(fraction_t f1, fraction_t f2)
```

For this lab, the function header has been changed to:

```
void multiply_fractions(const fraction_t *pf1,
                        const fraction_t *pf2,
                        fraction_t *product)
```

Read the documentation for `multiply_fractions`, carefully, and complete the definition. The fraction produced by this function must be in reduced form. (Hint: the fraction created by `make_fraction` is always in reduced form.)

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Review the console output, and verify that your `multiply_fractions` function passes all the tests in the 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 attendance/grading 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 pointers to `struct`s, 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 control returns from `make_fraction`, `reduce`, `gcd`, `add_fractions` and `multiply_fractions`. 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.