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

Lab 9 - Structures and Pointers

Objective

- To learn how to write functions that work with pointers to structures, including structures that are dynamically allocated from the heap.

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 the 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.

General Requirements

In Exercises 1 through 6, you are going to define functions that operate on structures that represent fractions. This lab is similar to last week's lab, and you should be able to reuse much of the code you developed then. The biggest difference is that, in this week's lab, instead of passing structures into functions and returning structures from functions, you'll be dealing with pointers to structures.

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. 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. Create a new Pelles C project named fraction. The project type must be Win32 Console program (EXE).

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`. Notice that the function's argument is a pointer to a `fraction_t` structure, and that the first statement calls `assert` to verify that this pointer is not `NULL`. Read the documentation for this function and complete the definition.

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

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. You can copy the function you wrote last week. Build your project, execute the program, and verify that the results returned by `gcd` are correct.

Exercise 3

File fraction.c contains the incomplete definition of a function named `reduce`. In last week's lab, 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)
```

In other words, 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 `reduce` will no longer return a reduced fraction. Instead, the function will reduce the fraction pointed to by parameter `pf`.

Read the documentation for this function, carefully, and implement it. Build your project, execute the program, and verify that the results produced by `reduce` are correct.

Exercise 4

File fraction.c contains the incomplete definition of a function named `make_fraction`. In last week's lab, the header for this function was:

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

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

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

In other words, the function's return type is now "pointer to a `fraction_t` structure".

As currently defined, the function allocates a `fraction_t` structure from the heap, but does not initialize the fraction before returning the pointer to it.

Read the documentation for this function, carefully, and complete the definition. Make sure that your function calls `assert` to check the pointer returned by `malloc`. Also, remember that, as part of the initialization, `make_fraction` must call `reduce` to convert the fraction to reduced form.

Build your project, execute the program, and verify that the fractions created by `make_fraction` are correct.

Exercise 5

File fraction.c contains the incomplete definition of a function named `add_fractions` that is passed pointers to two fractions. In last week's lab, 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:

```
fraction_t *add_fractions(fraction_t *pf1, fraction_t *pf2)
```

In other words, the function's arguments are now pointers to `fraction_t` structures, and the function's return type is now "pointer to a `fraction_t` structure".

As currently defined, the function always allocates the fraction 0/1 from the heap and returns the pointer to that fraction.

Read the documentation for this function, carefully, and complete the definition. Make sure that your function calls `assert` to check parameters `pf1` and `pf2` before adding the fractions. 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 fractions created by `add_fractions` are

correct.

Exercise 6

File fraction.c contains the incomplete definition of a function named `multiply_fractions` that is passed pointers to two fractions. In last week's lab, 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:

```
fraction_t *multiply_fractions(fraction_t *pf1, fraction_t *pf2)
```

In other words, the function's arguments are now pointers to `fraction_t` structures, and the function's return type is now "pointer to a `fraction_t` structure".

As currently defined, the function always allocates the fraction 0/1 from the heap and returns the pointer to that fraction.

Read the documentation for this function, carefully, and complete the definition. Make sure that your function calls `assert` to check parameters `pf1` and `pf2` before multiplying the fractions. 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 fractions created by `multiply_fractions` are correct.

Wrap-up

1. Remember to have a TA review and grade your solutions to the exercises before you leave the lab.

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 9 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. After you've finished uploading your file, remember to click the Submit assignment button. This will change the submission status to "Submitted for grading".