Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2016

Lab 6 - Structures and Pointers

Attendance/Demo

To receive credit for this lab, you must demonstrate your solutions to the exercises. **Also, you must submit your work to cuLearn**. (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 the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code 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

In Exercises 1 through 6, you are going to define functions that operate on structures that represent fractions. This lab is similar to Lab 5, and you should be able to reuse much of the code you developed then. The biggest difference is that, in this week's lab, we won't be using structures as function arguments. Instead, many of the function arguments will be pointers to structures, and the functions will operate on the structures through these pointers.

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 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.**

Getting Started

1. Launch Pelles C and create a new Pelles C project named fraction_pointer (all letters are

lowercase, with underscores separating the words). If you're using one of our lab computers, the project type must 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 your own computer, the project type should be Win 64 Console program (EXE) or Win32 Console program (EXE), depending on whether you installed the 64-bit or 32-bit edition of Pelles C. After creating the project, you should have a folder named fraction_pointer. Check this. If you do not have a project folder named fraction_pointer, close this project and repeat Step 1.

2. Download file main.c, fraction.c, fraction.h and sput.h from cuLearn. Move these files into your fraction_pointer 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 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. 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`. 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.

File main.c contains a function that exercises `print_fraction`. The test function does not determine if the information printed by `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.

Inspect the console 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 implement it, using Euclid's algorithm. You can reuse the function you wrote during Lab 5.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Inspect the console output, and verify that your `gcd` function passes all the tests in test suite #1 before you start Exercise 3.

Exercise 3

File fraction.c contains the incomplete definition of a function named `reduce`. In Lab 5, 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, <u>carefully</u>, 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. Inspect the console output, and verify that your `reduce` function passes all the tests in test suite #2 before you start Exercise 4.

Exercise 4

File fraction.c contains the incomplete definition of a function named `make_fraction`. In Lab 5, 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)
```

In other words, this function does not return a structure. Instead, it initializes the `fraction_t` structure pointed to parameter `new_fraction`.

Read the documentation for `make_fraction`, <u>carefully</u>, and complete the definition. **Remember that `make_fraction` 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. Inspect the console output, and verify that your `make_fraction` function passes all the tests in test suite #3 before you start Exercise 5.

Exercise 5

File fraction.c contains the incomplete definition of a function named `add_fractions` that is passed pointers to three fractions. In Lab 5, 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)
```

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

Read the documentation for this function, carefully, and complete the definition. The fraction created 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. Inspect the console output, and verify that your `add_fractions` function passes all the tests in test suite #4 before you start Exercise 6.

Exercise 6

File fraction.c contains the incomplete definition of a function named `multiply_fractions` that is passed pointers to three fractions. In Lab 5, 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)
```

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

Read the documentation for this function, carefully, and complete the definition. The fraction created 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. Inspect the console output, and verify that your `multiply_fractions` function passes all the tests in the test suite #5.

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 demo/sign-out sheet.

2. The next thing you'll do is package the project in a ZIP file (compressed folder) named fraction_pointer.zip. To do this:

   2.1. From the menu bar, select Project > ZIP Files... A Save As dialog box will appear. If you named your Pelles C project fraction_pointer, the zip file will have this name by default; otherwise, you'll have to edit the File name: field and rename the file to fraction_pointer before you save it. **Do not use any other name for your zip file** (e.g., lab6.zip, my_project.zip, etc.).

   2.2. Click Save. Pelles C will create a compressed (zipped) folder, 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_pointer).

3. Log in to cuLearn and submit fraction_pointer.zip. To do this:

   3.1. Click the Submit Lab 6 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag fraction_pointer.zip to the File submissions box. **Do not submit another type of file (e.g., a Pelles C .ppj file, a RAR file, a .txt file, etc.)**

   3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you aren't ready to do this; for example, you want to do some more work on the project and resubmit it later, you can leave the file with "draft" submission status. When you're ready to submit the final version, you can replace or delete your "draft" file submission by following the instructions in Step 3.3, then finish the submission process by following the instructions in Step 3.4.

   3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

      3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

      3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".

Extra Practice Exercise #1 - Functions with Pointer Parameters

General Requirements

When writing the code for this part, do not use arrays or structs. They aren't necessary.

The function you write should not perform console input; for example, contain `scanf` statements. The function should not produce console output; for example, contain `printf` statements.

You have been provided with file cube_main.c. This file contains an incomplete implementation of the function you have to design and code. It also contains a *test harness* (a function that will test your code, and a `main` function that calls the test function). Do not modify `main` or the test function.

Getting Started

1. Create a new project named cube. If you're using one of our lab computers, the project type must 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 your own computer, the project type should be Win 64 Console program (EXE) or Win32 Console program (EXE), depending on whether you installed the 64-bit or 32-bit edition of Pelles C. After creating the project, you should have a project folder named cube. Check this. If you do not have a folder named cube, close this project and repeat Step 1.

2. Download files cube_main.c and sput.h from cuLearn. Move these files into your cube folder.

3. You must also add cube_main.c to your project. To do this, select Project > Add files to project... from the menu bar. In the dialogue box, select cube_main.c, then click Open. An icon labelled cube_main.c will appear in the Pelles C project window.

   You don't need to add sput.h to the project. Pelles C will do this after you've added cube_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 cube_main.c in the editor. Design and code the function described below.

Exercise

This exercise requires you to write a simple function that has two pointer parameters.

A cube is a geometric solid consisting of six square faces that meet each other at right angles. Write a function that calculates the surface area and the volume of a cube. The function prototype is:

```
void cube_area_volume(double len, double *area, double *volume);
```

Parameter `len` is the length of the cube's sides. Assume that `len` will always be positive (your function does not have to check this.) Parameters `area` and `volume` point to the variables where the function will store the area and volume that the function calculates.

Suppose we have the declarations for two variables:

```
double surface_area;
double volume;
```

To calculate the surface area and volume of a cube whose sides have length 2, the function is called this way:

```
cube_area_volume(2, &surface_area, &volume);
```

When the function returns, the calculated area and volume will be in variables `surface_area` and `volume`, respectively.

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 function passes all the tests in the test suite.

Extra Practice Exercise #2 - Memory Diagrams with Pointers

Consider this program, which is adapted from an example in *Essential C*:

```
void swap(int* a, int* b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;   /* Point A */
```

```
}


void increment_and_swap(int* x, int* y)
{
   *x = *x + 1;
   *y = *y + 1;
   swap(x, y); // don't need & here since x and y are
               // already pointers-to-ints.
}


int main()
{
   int alice = 5;
   int bob = 10;
   swap(&alice, &bob);

   increment_and_swap(&alice, &bob);

   return 0;
}
```

(a) Draw a memory diagram that depicts the program's activation frame(s) immediately after the statement at Point A is executed for the **first** time; that is, immediately after

```
*b = temp;
```

is executed, but before the function returns, when `swap` is called by `main`.

(b) Draw a memory diagram that depicts the program's activation frame(s) immediately after the statement at Point A is executed for the **second** time; that is, immediately after

```
*b = temp;
```

is executed, but before the function returns, when `swap` is called by `increment_and_swap`.