Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2014

Lab 6 - 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. To do this, you'll develop another version of module you developed for Lab 5.

Attendance/Demo

To receive credit for this lab, you must make the effort to complete a reasonable number of exercises and demonstrate the code you complete. **Also, you must submit your lab work to cuLearn by the end of the lab period**. (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 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.

Background - The Heap and Dynamic Memory Allocation

In lectures, you've had a brief introduction to the heap. but we've not yet explored this topic in detail. Here is a review of what you need to know for this lab.

The C standard library has a function named `malloc`, which *dynamically allocates* memory from a region of memory known as the *heap*. Here is how we can allocate a `fraction_t` structure:

```
fraction_t *pf1;
...
pf1 = malloc(sizeof(fraction_t));
assert(pf1 != NULL);
```

The argument passed to `malloc` is the amount of heap memory that should be allocated. Here, the expression `sizeof(fraction_t)` calculates the amount of memory required to store one `fraction_t` structure. After allocating the structure in the heap, `malloc` returns a pointer to the structure. This pointer is saved in `pf1`.

If `malloc` is unsuccessful (most likely because the heap doesn't have enough available memory), it returns `NULL` as the pointer value. Conversely, when `malloc` successfully allocates heap memory, the pointer it returns will never be `NULL`.

What should we do if `malloc` fails? Often, we treat this as an unrecoverable problem, and terminate the program. To handle this possibility, we pass the value of the expression `pf1 != NULL` to `assert`. If this expression is `false` (because `pf1` contains a `NULL` pointer, which means `malloc` failed), `assert` causes the program to terminate with an error message. If the expression is `true`, `assert` does nothing, and subsequent statements can assume that `malloc` succeeded and `pf1` contains a valid pointer.

There is no difference between using a pointer returned by `malloc` and a pointer obtained by using the address-of operator (`&`). As an example, the following code fragment initializes two pointers of type "pointer to `fraction_t`":

```
fraction_t *pf1;
...
pf1 = malloc(sizeof(fraction_t)); // pf1 points into the heap
assert(pf1 != NULL);

fraction_t frac;  // frac is a structure declared inside a
                  // function (a local variable)
...
fraction_t *pf2 = &frac;  // pf2 points to the local
                          // variable, frac
```

We can initialize both two fractions to 2/3:

```
pf1->num = 2;
pf1->den = 3;
pf2->num = 2;
pf2->den = 3;
```

Notice that, from these statements, there's no way to determine whether the structures pointed to by `pf1` and `pf2` are in the heap or in a function's activation frame.

There's one important difference between the two structures: their lifetimes. Local variable `frac` is created in the function's activation frame when the function is called, and it will "disappear" when the function returns and its activation frame is deallocated. As discussed in class, returning the pointer to this structure (i.e., `pf2`) is a dangerous bug.

In contrast, the structure that was allocated on the heap remains available until we explicitly deallocate it (by calling another standard library function, `free`). This means that a function can return a pointer to heap memory allocated by `malloc`, and that pointer can be passed to and used by other functions. Here is an example:

```
fraction_t *first, *second, *sum;
first = make_fraction(2, 3);
second = make_fraction(4, 5);
sum = add_fractions(first, second);
```

`make_fraction` calls `malloc` to allocate a `fraction_t` structure from the heap, initializes the fraction with the specified numerator and denominator, and returns a pointer to the fraction. So, in this example, `first` and `second` are assigned pointers to two structures are in the heap. Those pointers are then passed to `add_fractions`, which returns a pointer to a `fraction_t` structure (also allocated by `malloc`) that contains the sum of the fractions pointed to by `first` and `second`. This pointer is stored in `sum`. We can continue to use the three fractions (via the pointer variables) until we explicitly deallocate them.

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, instead of passing structures as function arguments 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 four 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.**
- 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.**

Instructions

1. Create a new folder named Lab 6.
2. Create a new Pelles C project named fraction_pointer inside your Lab 6 folder. The project type must be Win32 Console program (EXE). After creating the project, you should have a folder named fraction_pointer inside your Lab 6 folder (check this).
3. Download file main.c, fraction.c, fraction.h and sput.h from cuLearn. Move these files into your fraction_pointer folder.
4. You must also add main.c and fraction.c to your project: from the menu bar, select Project > Add files to project... 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.

5. Build the project. It should build without any compilation or linking errors.
6. 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 that the harness tests.
7. 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, 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.

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 copy 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 the test suite 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, 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. Inspect the console output, and verify that your `reduce` function passes all the tests in the test suite before you start Exercise 4.

Exercise 4

If you haven't done so already, read the *Background* section at the beginning of this lab handout.

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:

```
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 calls `malloc` to allocate a `fraction_t` structure from the heap, but it does not check the pointer returned by `malloc`, and does not initialize the fraction before returning the pointer to it.

Read the documentation for `make_fraction`, carefully, and complete the definition. Make sure that your function calls `assert` to check the pointer returned by `malloc`. Notice that there will be a second call to `assert`, to check the value of the denominator. Also, 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 the test suite 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 two 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:

```
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 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 the test suite 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 two 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:

```
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 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.

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) named fraction_pointer.

- 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 the same 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.).

- 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.

- Click the Submit Lab 6 link. After you click the Add submission button, drag fraction_pointer.zip to the File submissions box. Do not submit another type of file (e.g., a Pelles C .ppj file, RAR file, a .txt file, etc.)

- 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.

- Once you're sure that you don't want to make any changes, click the Submit assignment button. This will change the submission status to "Submitted for grading". **Note: after you've clicked the Submit assignment button, you cannot resubmit the file.**