Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming - Winter 2019

Lab 6 - C Structs, Pointers and the Heap

Demo/Grading

After you finish the exercises, a TA 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.**

Prerequisite Reading

Lecture slides for:

- Topic 8: *C: Structures*
- Topic 9c: *C: Pointers to Structures*
- Topic 10: *C: The Heap and Dynamically Allocated Memory*. Make sure you understand the implementations of `makepoint` and `addpoints` that return a pointer to a dynamically allocated `point_t` struct. (For this lab, you can ignore the slides that show how to use `free` to deallocate heap memory and how to use `malloc` to dynamically allocate an array.)

Background - Using structs to Represent Polynomials

A *polynomial in one variable with constant coefficients* is the sum of terms given by the expression:

$$a_n x^n + ... + a_2 x^2 + a_1 x + a_0$$

Each term in the polynomial consists of a variable x raised to an exponent and multiplied by a coefficient. Here, the coefficients of the polynomial are the constant values $a_n, a_{n-1}, \ldots a_2, a_1, a_0$.

Here is the declaration for a C struct that represents terms in a polynomial:

```
typedef struct {
      int coeff;
      int exp;
} term_t;
```

This struct has two members, both of type `int`[1]. Member `coeff` is the term's coefficient and member `exp` is its exponent.

[1] Coefficients in polynomial terms can, in general, be real numbers; however, in order to simplify some of the exercise test cases, we'll only consider polynomials in which the coefficients are integers.

Here is the declaration for a struct that represents polynomials that have at most `MAX_TERMS` terms[2] :

```
#define MAX_TERMS 10

typedef struct {
     term_t *terms[MAX_TERMS];
     int num_terms;
} polynomial_t;
```

Member `terms` is an array that keeps track of the polynomial's terms. Instead of storing `term_t` structs in the array, each element stores a pointer to a `term_t` struct. That's why the array declaration is:

```
term_t *terms[MAX_TERMS];
```

(`terms` is an array with `MAX_TERMS` elements of type "pointer to `term_t`").

Member `num_terms` keeps track of the number of terms in the polynomial; that is, the number of pointers stored in the array.

General Requirements

You have been provided with four files:

- polynomial.c contains incomplete definitions of several functions you have to design and code.
- polynomial.h contains the declarations of the `term_t` and `polynomial_t` structs, as well as the declarations (function prototypes) for the functions you'll implement. **Do not modify polynomial.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.**

None of the functions you write should call `calloc`, `realloc` or `free`. Only `make_term` (Exercise 2) and `make_polynomial` (Exercise 4) are permitted to call `malloc`.

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). Instructions for selecting the formatting style and formatting blocks of code are in the Lab 1 handout.

Finish each exercise (i.e., write the function and verify that is passes all its tests) before you move on to the next one. Don't leave testing until after you've written all the functions.

[2] Later in the course, we'll cover two data structures (the dynamic array and the linked list) that can be used to implement a polynomial that isn't limited to having a fixed, maximum number of terms.

Getting Started

Step 1: Launch Pelles C and create a new project named polynomial. (Instructions for creating projects are in the handout for Lab 1.) If you're using the 64-bit edition of Pelles C, select Win 64 Console program (EXE) as the project type. If you're using the 32-bit edition of Pelles C, select Win32 Console program (EXE). **Don't click the icons for Console application wizard, Win32 Program (EXE) or Win64 Program (EXE) - these are not correct types for this project.**

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

Step 3: Add main.c and polynomial.c to your project. (Instructions for doing this are in the handout for Lab 1.) You don't need to add polynomial.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 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 polynomial.c in the editor.

Exercise 1

File polynomial.c contains an incomplete definition of a function named `print_term`. Read the documentation for this function and complete the definition. Notice that the function parameter is a pointer to a `term_t` struct; in other words, the function argument is the *address* of a struct that represents a polynomial term.

Build the project, correcting any compilation errors, then execute the project. The test harness will run.

Test suite #1 exercises `print_term`, but it cannot verify that the information printed by the function is correct. Instead, it displays what a correct implementation of `print_term` should print (the expected output), followed by the actual output from your implementation of the function.

Use the console output to help you identify and correct any flaws. Verify that `print_term` passes all the tests in test suite #1 before you start Exercise 2.

Exercise 2

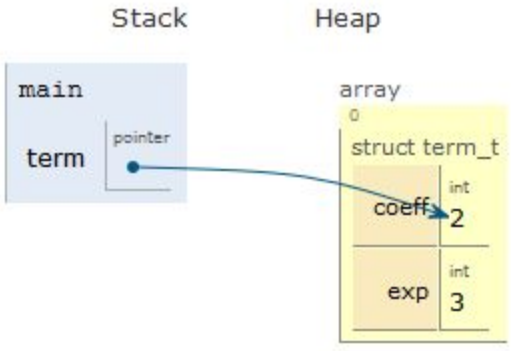
File polynomial.c contains an incomplete definition of a function named `make_term`. Read the documentation for this function and complete the definition. Your implementation must call `assert` so that the program terminates if:

- the term's coefficient is 0;
- the term's exponent is negative;
- `malloc` was unable to allocate memory for a `term_t` struct.

Suppose `main` contains this statement:

```
term_t *term = make_term(2, 3);
```

Here's a C Tutor diagram that depicts memory after the statement is executed:



`make_term` called `malloc` to allocate a `term_t` struct on the heap, initialized the struct's members, then returned the pointer to the struct to `main`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `make_term` passes all the tests in test suite #2 before you start Exercise 3.

Exercise 3

File polynomial.c contains an incomplete definition of a function named `eval_term`. Read the documentation for this function and complete the definition.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `eval_term` passes all the tests in test suite #3 before you start Exercise 4.

Exercise 4 is on the next page.

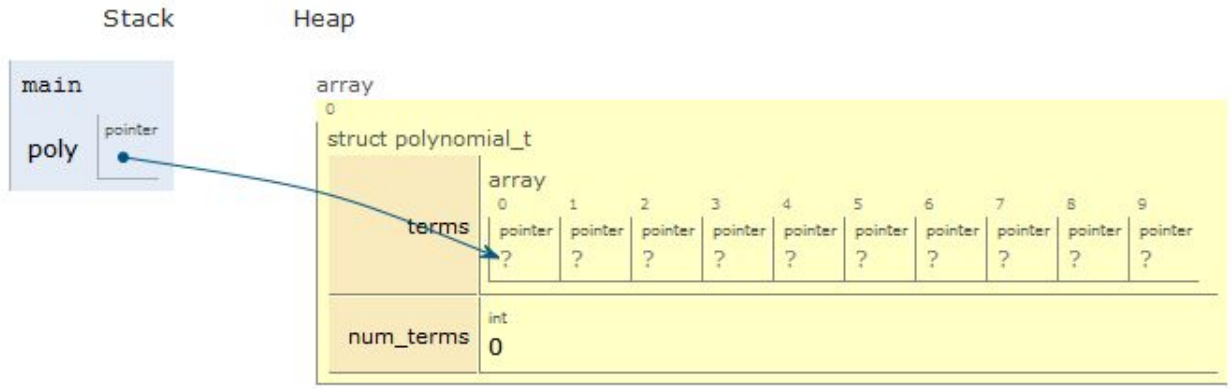
Exercise 4

File polynomial.c contains an incomplete definition of a function named `make_polynomial` that, when completed, will return a pointer to a new polynomial containing 0 terms. Read the documentation for this function and complete the definition. Your implementation must call `assert` so that the program terminates if `malloc` was unable to allocate memory for a `polynomial_t` struct.

Suppose `main` contains this statement:

```
polynomial_t *poly = make_polynomial();
```

Here is a C Tutor diagram that depicts memory after the statement is executed:



`make_polynomial` called `malloc` to allocate a `polynomial_t` struct on the heap, initialized the struct's `num_terms` member, then returned the pointer to the struct to `main`.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `make_polynomial` passes all the tests in test suite #4 before you start Exercise 5.

Exercise 5 is on the next page.

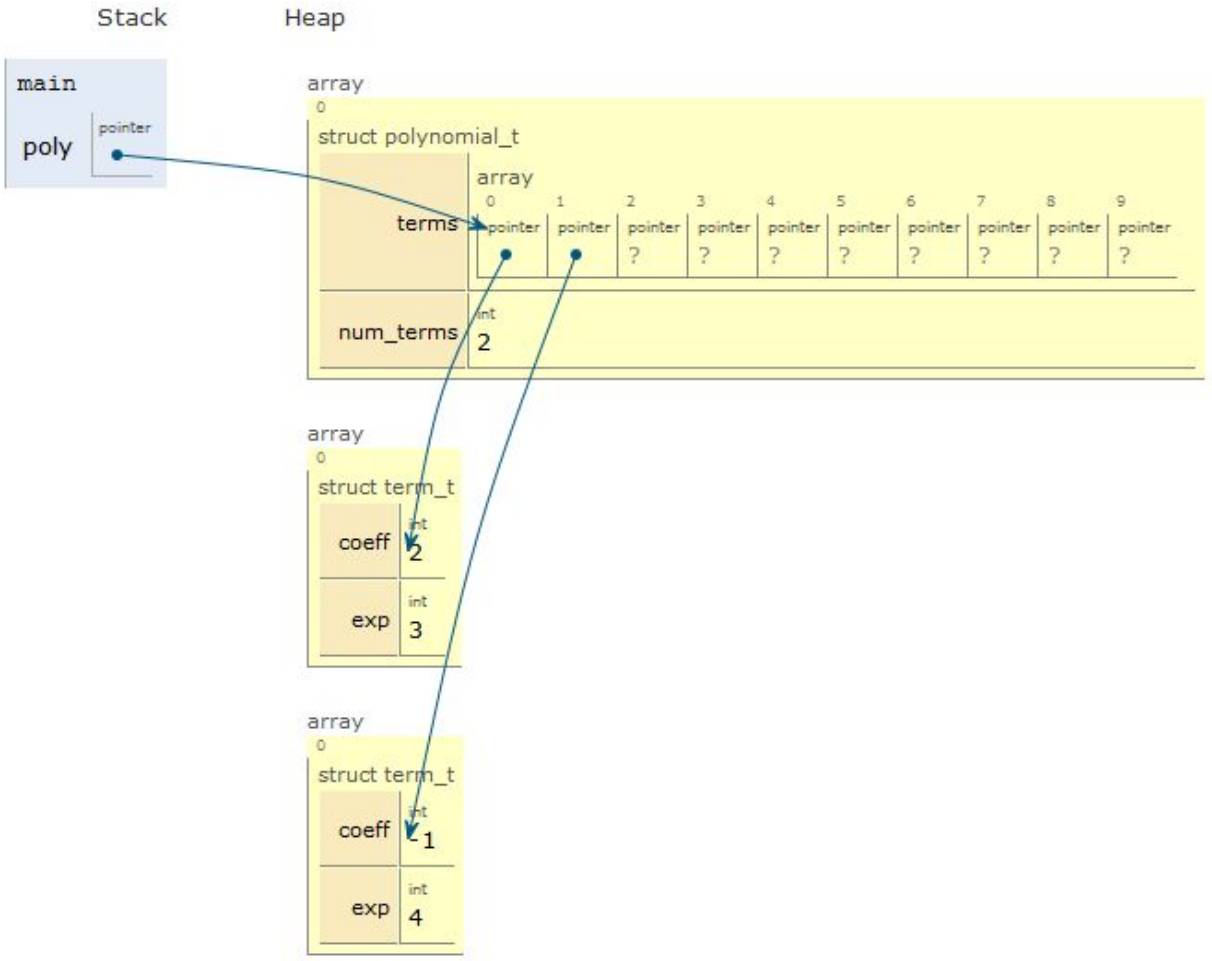
Exercise 5

File polynomial.c contains an incomplete definition of a function named `add_term`. Read the documentation for this function and complete the definition. Your implementation must call `assert` so that the program terminates if there's no room in the polynomial for additional terms.

Suppose `main` contains these statements:

```
polynomial_t *poly = make_polynomial();
add_term(poly, make_term(2, 3));
add_term(poly, make_term(-1, 4));
```

Here is a C Tutor diagram that depicts memory after the statements are executed:



The first time `add_term` was called, it stored the pointer to the `term_t` struct created by `make_term(2, 3)` in element 0 of array `terms`. The second time `add_term` was called, it stored the pointer to the `term_t` struct created by `make_term(-1, 4)` in element 1 of array `terms`. Variable `poly` now represents the polynomial $-1x^4 + 2x^3$.

Note 1: `add_term` can access element `i` in the struct's array using this expression:

```
poly->terms[i]
```

This expression might appear complicated, so let's break it into pieces:

- Parameter `poly` is a pointer to a `polynomial_t` struct; for example, a pointer returned

by `make_polynomial`.

- Expression `poly->terms` is equivalent to `(*poly).terms`, so `poly->terms` selects the array named `terms` in the `struct` pointed to by `poly`.
- Because `terms` is an array, individual elements are accessed using the `[]` operator. So, `poly->terms[i]` is the element at position `i` in the array. This element stores a pointer to a `term_t` struct.

Note 2: Your function shouldn't make a copy of the struct pointed to by parameter `term`; instead, it should just store the the pointer in array `terms`;

Note 3: Don't worry about arranging the terms in increasing or decreasing order of exponents. Each time `add_term` is called, just store the `term_t *` pointer in the next unused element in array `terms`. This means that, if `poly` represents a polynomial of degree *n*, the first array element (`poly->terms[0]`) doesn't necessarily point to the term with the highest power (*n*) (See the memory diagram on the previous page.)

We could, of course, reverse the order of the two calls to `add_term`:

```
polynomial_t *poly = make_polynomial();
add_term(poly, make_term(-1, 4));
add_term(poly, make_term(2, 3));
```

In this case, the memory diagram would be different from the one shown earlier (`poly->terms[0]` would point to the highest-order term), but `poly` would represent the same polynomial: $-1x^4 + 2x^3$.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `make_polynomial` passes all the tests in test suite #5 before you start Exercise 6.

Exercise 6

File polynomial.c contains an incomplete definition of a function named `eval_polynomial`. Read the documentation for this function and complete the definition. Your implementation must call `assert` so that the program terminates if the polynomial has 0 terms.

Build the project, correcting any compilation errors, then execute the project. The test harness will run. Use the console output to help you identify and correct any flaws. Verify that `eval_polynomial` passes all the tests in 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 grading/sign-out sheet.
2. Remember to backup 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.

A homework exercise is on the next page.

Homework Exercise - Visualizing Program Execution

In the midterm and final exams, you will be expected to be able to draw diagrams that depict the execution of short C programs that use pointers to dynamically allocated `struct`s, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills when working with programs that allocate allocate memory from the heap.

1. The *Labs* section on cuLearn has a link, Open C Tutor in a new window. Click on this link.

2. Copy/paste the declarations from polynomial.h into the C Tutor editor. Copy/paste your solutions to Exercises 2 through 6 into the editor. You can comment out the `assert` calls.

3. Write a short `main` function that exercises your functions.

4. *Without using C Tutor,* trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before each of your functions returns. Use the same notation as C Tutor.

5. Use C Tutor to trace your program one statement at a time, stopping just before each function returns. Compare your diagrams to the visualizations displayed by C Tutor.