Lab 9 - Dynamic Memory Allocation

Background - A Polynomial Data Structure

A *univariate polynomial*; that is, a polynomial one variable with constant coefficients, can be expressed as:

$$a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0$$

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 **integer** values a_n, a_{n-1}, … a_2, a_1, a_0[1].

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

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

Member `coeff` is the term's coefficient and member `exp` is its exponent.

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

```
#define MAX_TERMS 10

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

A `polynomial_t` structure has two members: `term` and `num_terms`. Notice the declaration of `terms`:

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

This means that `terms` is an array of `MAX_TERMS` elements, and these elements have type "pointer to `term_t`"; in other words, each element in `terms` stores a pointer to a `term_t` struct.

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

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

General Requirements

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

Your functions must not be recursive. Repeated actions must be implemented using C's `while`, `for` or `do-while` loop structures.

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 were given in the VS Code installation instructions.

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

Getting Started

Step 1: Depending on your operating system, download (**lab9_win.zip** or **lab9_mac.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Double click on lab9.code-workspace

Step 3: lab9_xxx folder contains the following 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.**

Step 4: Run the project. It should build without any compilation or linking errors.

Step 5: Execute the project. For the functions where the test harness is provided (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.

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

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.

To keep things simple, a term is always printed using the format `"ax^e"`., even if the coefficient is 1 or the exponent is 0 or 1. Here is a table that lists, for several polynomial terms, the coefficient and exponent that will be stored in the `term_t` structure, and how the polynomial should appear when it is printed:

Term	Coefficient	Exponent	Printed as
1	1	0	`1x^0`
x	1	1	`1x^1`
x^2	1	2	`1x^2`
3	3	0	`3x^0`
$3x$	3	1	`3x^1`
$3x^2$	3	2	`3x^2`

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.

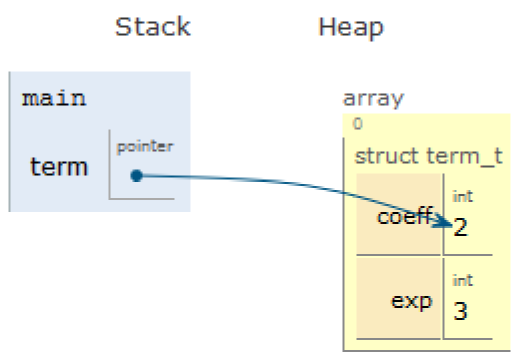
Exercise 2

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

Suppose `main` contains this statement:

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

Here's a C Tutor diagram that depicts memory after `term` is initialized with the pointer returned by `make_term`:



Complete the function definition. Your implementation must call `assert` so that the program terminates if:

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

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. (If `make_term` fails any tests, consider using C Tutor to help you debug your code. Copy the declaration of `term_t` and your `make_term` function into the C Tutor editor, and write a little `main` function that calls `make_term`. The image displayed by C Tutor should look like the screen capture shown above.)

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.

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.

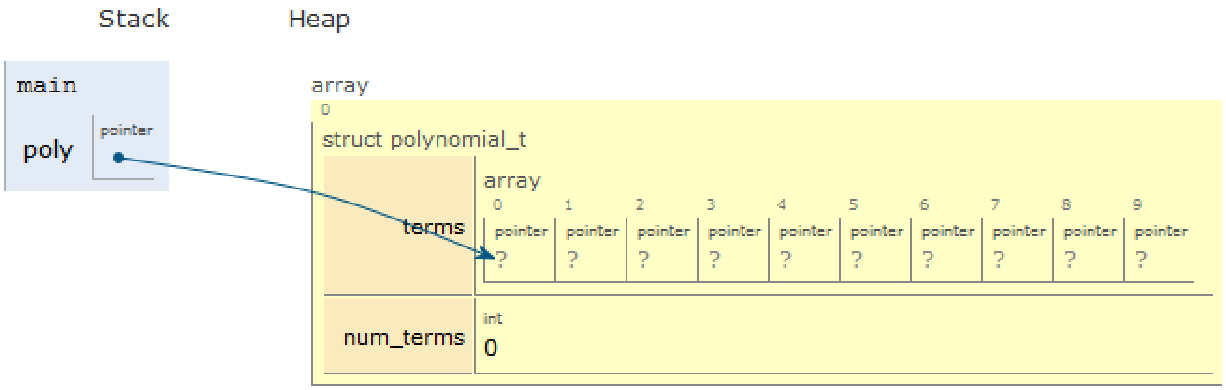
Exercise 4

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

Suppose `main` contains this statement:

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

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



Complete the function definition. Your implementation must call `assert` so that the program terminates if `malloc` was unable to allocate memory for a `polynomial_t` struct.

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. (If `make_polynomial` fails any tests, consider using C Tutor to help you debug your code. Copy the declarations of `term_t` and `polynomial_t`, and your `make_term` function, into the C Tutor editor, and write a little `main` function that calls `make_polynomial`. The image displayed by C Tutor should look like the screen capture shown above.)

Exercise 5

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

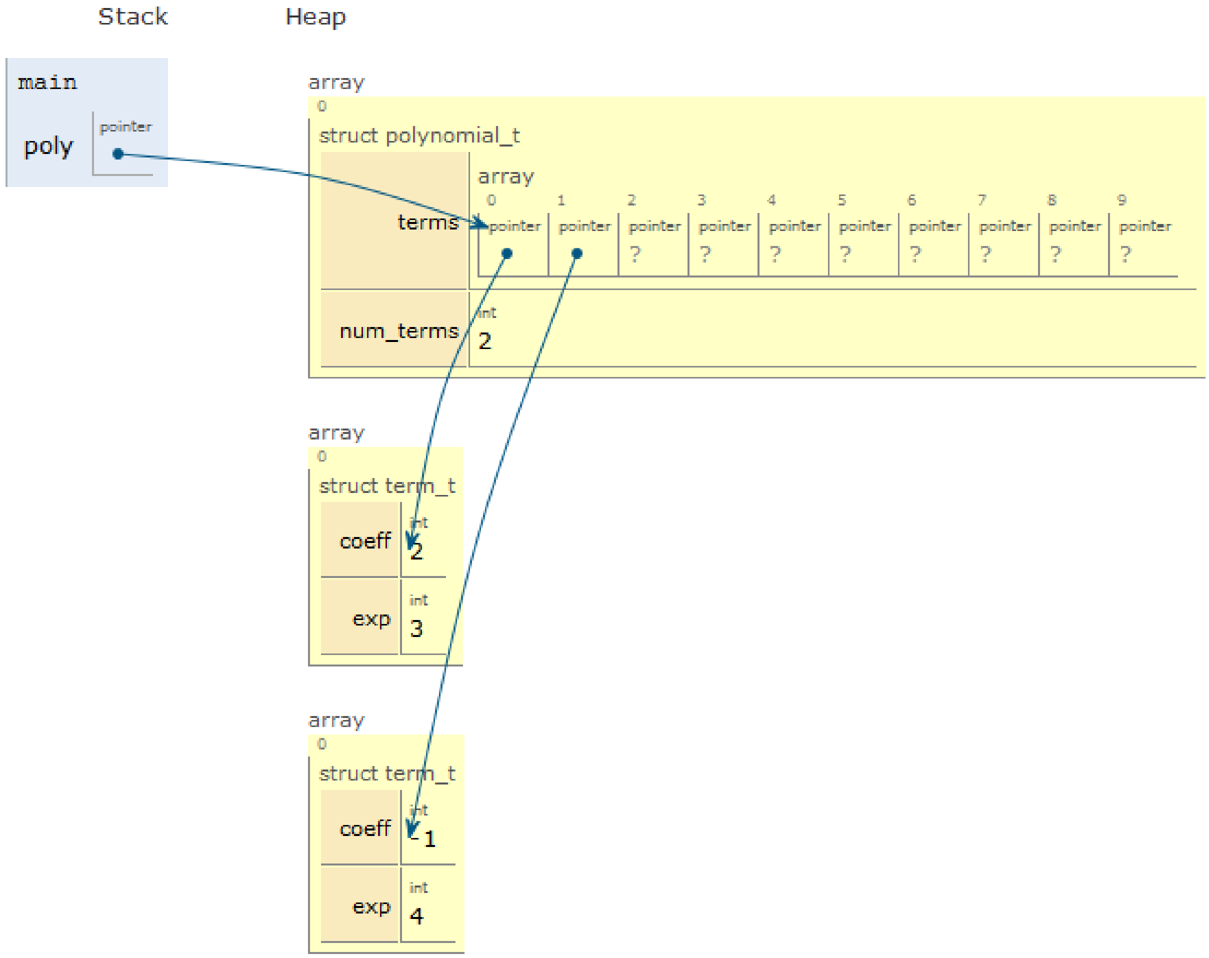
Suppose `main` contains these statements:

```
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 these 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 $-1x^4 + 2x^3$.

Read the following notes, then complete the function definition. Your implementation must call `assert` so that the program terminates if there's no room in the polynomial for additional terms.



Note 1: `add_term` can access element `i` in the `terms` 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 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.)

For example, we could 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: $-1x^4 + 2x^3$.

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.

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

Submit polynomial.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace* (lab3 file). Remember: submit early, submit often.
- Make sure that polynomial.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you are submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

Homework Exercise - Visualizing Program Execution

In the final exam, you will be expected to understand and draw diagrams that depict the execution of short C programs that use dynamically allocated memory, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills.

1. Click on this link to open C Tutor.
2. Copy/paste your solutions to Exercises 2 through 6 into the C Tutor 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. If C Tutor complains that your program is too long, delete the comments above the function definitions.
6. Do not submit your solutions for this exercise.