CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #5

Spring 2021

Assignment Due: Wednesday, May 12th, 11:59 PM (EST)

Change Log:

04/30: Fixed a typo in starter test files. Please re-download.
05/03: Added a test case to parts 7 and 8.
Clarified by way of an example when a failure could occur for part 7 and 8.
Added an explicit assumption to part 3.

05/04: Clarified N < array_size case in part 4.

Learning Outcomes

After completion of this programming project you should be able to:

- Dynamically allocate memory
- Manage allocated memory using linked lists

Getting Started

Visit the course website and download MarsSpring2021.jar. From Blackboard, download the starter code (**hw5.zip**) accompanying this document. The archive is also available on the course website.

Inside **hw5.zip** you will find **hw5.asm** and a bunch of test files. The file **hw5.asm** contains several function stubs, which you will need to implement. These stubs contain only `jr $ra` instructions. Your job in this assignment is to implement all the functions as specified below. Do not change the function names since the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, and add them to **hw5.asm**.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic into MIPS assembly code. It may seem like this approach takes more time. But once you have the implementation details pinned down in a higher-level language, then translating it to assembly is less error prone. This will help save time and errors.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

IMPORTANT: Do not define a `.data` section in your hw5.asm file. A submission that contains a `.data` section will most likely not integrate with our grading script. This may lead to you getting no credit.

Important Information about CSE 220 Programming Projects

- Read this document carefully. If you cannot understand something, ask for clarification . The course staff will try and answer those questions proactively. However, questions whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- You must use the Stony Brook version of MARS posted on the course website. Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you might need to complete the homework assignments.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. This will make your code readable, which will help us grade your code and also assist you better if you get stuck.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- Submit to Blackboard by the due date and time. Late work will be penalized as described in the course syllabus. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

Your programming assignments will be graded almost entirely in an automated way. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this assignment, your program will be generating output and your functions will be returning values that will be checked for exact matches by the grading scripts. It is your responsibility to output/return the expected values.

Some other items you should be aware of:

- Each test case must execute in 100,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed. To find the instruction count of your code in MARS, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

  If you are using the command line to assemble and run your code, you can use the following command:

  ```
  $ java -jar /path/to/MarsSpring2021.jar /path/to.asm --argv <args> --noGui -i -q
  ```

  The path separators will be '\' instead of '/' if you are on a Windows machine.

- Any excess output from your program (debugging notes, etc.) will impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

You must follow the register conventions discussed in lecture and recitation. You can also review the register conventions from this illustrative MIPS register conventions guide by Prof. McDonnell. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any $s registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save $ra before calling the callee. In addition, if the caller wants a particular $a, $t or $v register's value to be preserved across the secondary function call, then place a copy of that register in an $s register before making the function call.
- A function which allocates stack space by adjusting $sp must restore $sp to its original value before returning.
- There is no reason to modify the registers $fp and $gp in this assignment. So, try not to change them. However, if a function modifies one or both, the function must restore them before returning to the caller.

Note that you will lose points, if you do not follow register conventions. Grading scripts will check to see your code follows the conventions.

Unit-Testing Functions

To test your implemented functions, you may use the files ending with `_test.asm` in the starter code (**hw5.zip**) provided to you. You should add your own test code to these files before assembling and running them. The .data section in the test files contain some initial test cases. They are by no means comprehensive. Add more test data to the .data section in the test files to gain comprehensive test coverage.

Each test file contains initial code to help you hook/call into the functions that you are supposed to implement in *hw5.asm*. You will need to add code to call the appropriate functions with appropriate arguments. The astute reader may also write a script to automatically run all the test files with the necessary test cases. This is of course not a requirement and should not be attempted unless you are comfortable with scripting. Your focus should be on first implementing the expected functionality. Your submission will be graded using the examples provided in this document and additional test cases. Do not submit your test files. They will be deleted. We will use different test files for grading.

Dynamic Memory Allocation

To store data in the system heap, MARS provides system call #9, which is called sbrk. For example, to allocate *N* bytes of memory, where *N* is a positive integer literal, we would write this code:

```
li $a0, N
li $v0, 9
syscall
```

When the system call completes, the address of the newly-allocated memory buffer will be available in $v0. The address will be on a word-aligned boundary, regardless of the value passed through $a0. Unfortunately, there is no way in MARS to de-allocate memory to avoid creating memory leaks. The run-time systems of Java, Python and some other languages take care of freeing unneeded memory blocks with garbage collection, but assembly languages, C/C++, and other languages put the burden on the programmer to manage memory. You will learn more about this in CSE 320.

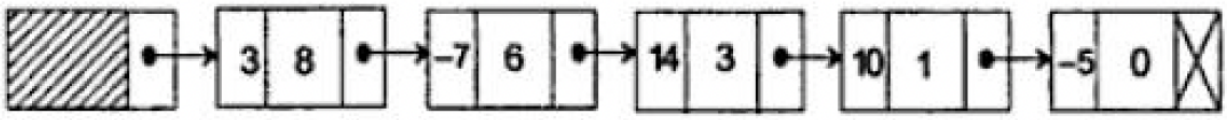
Single Variable Polynomial Arithmetic

A single variable polynomial is an algebraic expression of the form:
$P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + ... + a_1 x^{e_1}$, where $a_i x^{e_i}$ is a term in the polynomial so that a_i is

a non-zero coefficient and e_i is a positive coefficient. We will assume an ordering of the terms in the polynomial such that $e_n > e_{n-1} ... > e_2 > e_1 >= 0$.

An application of linked lists is to represent polynomials and their manipulations. The primary advantage of this representation is that it can accommodate polynomials of growing sizes. One way to represent a polynomial as linked lists is to think of the terms in the polynomial as nodes in a linked list. Hence, a term can be represented as:

A term should have three fields, a coefficient (COEFF), an exponent (EXP), and a pointer to the next term (LINK). An additional node called the head term is used to represent the polynomial itself. Hence, a polynomial $P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$ is visually represented as the following linked list:



Our job in this assignment is to implement functions that will implement polynomials as linked lists and perform basic operations on them such as addition and multiplication. But before that, we define the data structures that we will use for this purpose.

Data Structures

The terms in the polynomial have the following structure:

```
struct Term {
   int coeff          // coefficient of a term (4 bytes)
   int exp            // exponent of a term (4 bytes)
   Term* next_term    // Pointer to the next term in the polynomial
                      //(4 bytes)
}
```

If a `Term` has `next_term` set to the NULL character, then it is the last term in the polynomial.

The polynomial itself is a simple structure that holds the head term of the polynomial.

```
struct Polynomial {
   Term* head_term    // Pointer to the head term of the polynomial
```

```
                        // (4 bytes)
}
```

Implement the functions defined in the following parts to build a basic polynomial arithmetic calculator in MIPS.

Part 1: Create A Term

```
Term* create_term(int coeff, int exp)
```

The `create_term` function initializes a single-variable polynomial term with the inputs coefficient (`coeff`) and exponent (`exp`), both of which are signed integers. It allocates 12 bytes of memory in the heap to hold the term structure. The structure is initialized as follows:

1. The first 4 bytes holds the coefficient,
2. The next 4 bytes holds the exponent, and
3. The final 4 bytes holds the address of another term. Set this to 0 at this point.

You must use *syscall sbrk* to allocate 12 bytes for the term.

The function returns the address of the term in $v0. If the coefficient is 0 or the exponent is negative, then return -1 in $v0.

Returns in $v0:

- The address of a single-variable polynomial term or -1.

Additional requirements:

- The function must not make any changes to main memory except as needed to allocate memory for creating a term.

Test Cases:

1. `create_term(2,3)` => Address of *Term(2,3,NULL)* in $v0
2. `create_term(0,3)` => -1 in $v0
3. `create_term(2,-3)` => -1 in $v0

Term(c,e,n) represents an instance of the *struct Term* with fields *c,e,n* denoting coefficient, exponent, and address of the next term respectively.

Part 2: Initialize Polynomial

```
int init_polynomial(Polynomial* p, int[2] pair)
```

The `init_polynomial` function takes a pointer, `p`, and a 2-element array, `pair`, as inputs. The argument `pair` contains the coefficient and the exponent used to form a term in a single-variable polynomial. For example, the term $3x^2$ is formed by the coefficient 3 in

`pair[0]` and exponent 2 in `pair[1]`. The function should create a term from the argument `pair` and initialize `p->head` with the term's address. If the pair contains a 0 coefficient or a negative exponent, then the function returns -1 in $v0; otherwise returns 1 in $v0.

Returns in $v0:

- -1 or 1

Additional requirements:

- The function must not make any changes to main memory except as necessary.

Test Cases:

1. pair: 2 3
   `init_polynomial(p,pair)` => 1 in $v0
   SideEffect: p->head_term = Address of *Term(2,3,NULL)*

2. pair: 0 3
   `init_polynomial(p,pair)` => -1 in $v0

3. pair: 2 -3
   `init_polynomial(p,pair)` => -1 in $v0

Part3: Extend Polynomial with N Terms

```
int add_N_terms_to_polynomial(Polynomial* p, int[] terms, N)
```

The function `add_N_terms_to_polynomial` takes as input:

1. a pointer, `p`, that represents a polynomial.
2. an array of pairs, of the form (coefficient, exponent) represented by `terms`. Assume that the array will always be terminated by the pair (0,-1).
3. the no. of element in the `terms` array to be added to the polynomial, represented by integer `N`.

For each pair in `terms`, the function creates a term in the heap, and places the term in the linked list that represents the polynomial. *The term must be placed in a way that keeps the polynomial terms sorted by their exponents*. If `terms` has duplicate exponents, then only the first exponent that appears in `terms` is considered for adding to the list, the rest are discarded. If N is more than the no. of terms in `terms` array, then add all terms in the array.

The function returns the no. of terms added in $v0.

You can assume that p is non-empty. Do not make this assumption for any other part.

Returns in $v0:

- No. of terms added

Additional requirements:

- The function must not make any changes to main memory except as necessary.

Test Cases:

1. terms: 2 2 4 3 5 0 0 -1
   `add_N_terms_to_polynomial(p,terms,3)` => 3 in $v0
   SideEffect:
   p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*
2. terms: 2 3 4 3 5 0 0 -1
   `add_N_terms_to_polynomial(p,terms,3)` => 2 in $v0
   SideEffect:
   p->head_term = Address of *Term(2,3,next_term0)*
   next_term0 = Address of *Term(5,0,NULL)*

3. terms: 2 2 4 3 5 0 0 -1
   `add_N_terms_to_polynomial(p,terms,0)` => 0 in $v0
   SideEffect: None

4. terms: 2 2 4 3 5 0 0 -1
   `add_N_terms_to_polynomial(p,terms,-1)` => 0 in $v0
   SideEffect: None

Part4: Update N Terms In Polynomial

```
int update_N_terms_in_polynomial(Polynomial* p, int[] terms, N)
```

The function `update_N_terms_in_polynomial` takes as input:

1. a pointer, `p`, that represents a polynomial.
2. an array of pairs, of the form (coefficient, exponent) represented by `terms`. Assume that the array will always be terminated by the pair (0,-1).
3. the no. of elements in `terms` to be considered for updating, represented by an integer `N`.

For each pair in `terms`, the function searches for a term in the polynomial with the exponent in the pair, and updates the term with the coefficient in the pair. Updates maintain the sorted property of the polynomial.

The function returns the no. of terms updated in $v0. If a term with the same exponent is updated multiple times, then it is considered as 1 term being updated. If N is greater than or

equal to the size of `terms`, then consider all elements in the array for updating. If N is less than the size of `terms`, then consider upto N elements in the array.

Returns in $v0:

- No. of terms updated

Additional requirements:

- The function must not make any changes to main memory except as necessary.

Test Cases:

1. terms: 1 2 3 3 1 0 0 -1
   p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `update_N_terms_in_polynomial(p,terms,3)` => 3 in $v0
   SideEffect:
   p->head_term = Address of *Term(3,3,next_term0)*
   next_term0 = Address of *Term(1,2,next_term1)*
   next_term1 = Address of *Term(1,0,NULL)*

2. terms: 1 3 3 3 1 0 0 -1
   p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `update_N_terms_in_polynomial(p,terms,3)` => 2 in $v0
   SideEffect:
   p->head_term = Address of *Term(3,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(1,0,NULL)*

3. `add_N_terms_to_polynomial(p,terms,0)` => 0 in $v0
   SideEffect: None

4. `add_N_terms_to_polynomial(p,terms,-1)` => 0 in $v0
   SideEffect: None

Part 5: Get Nth highest term

```
(int,int) get_Nth_term(Polynomial* p, N)
```

The function `get_Nth_term` takes as inputs a pointer, `p`, to a linked list representing a polynomial and an integer, `N`. It finds the term with Nth highest exponent in the polynomial and returns the exponent in $v0 and the coefficient in $v1. Returns (-1,0) if no term with Nth-highest exponent exists in the polynomial.

Returns in $v0:

- Exponent of term with Nth highest exponent.
- -1, if exponent not found

Returns in $v1:

- Coefficient of term with Nth highest exponent.
- 0, if exponent not found

Additional requirements:

- The function must not make any changes.

Test Cases:

1. p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `get_Nth_term(p,terms,1)` => 3 in $v0 and 4 in $v1

2. p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `get_Nth_term(p,terms,2)` => 2 in $v0 and 2 in $v1

3. p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `get_Nth_term(p,terms,3)` => 0 in $v0 and 5 in $v1

4. p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `get_Nth_term(p,terms,6)` => -1 in $v0 and 0 in $v1
   SideEffect: None

Part 6: Remove A Term

`(int,int) remove_Nth_term(Polynomial* p, N)`

The function `remove_Nth_term` takes as inputs a pointer, `p`, to a linked list representing a polynomial and a positive integer, `N`. It **removes** the term with the Nth highest exponent in the polynomial and returns the exponent in $v0 and the coefficient in $v1. Return (-1,0) if no term with Nth highest exponent exists in the polynomial.

Returns in $v0:

- Exponent of term with Nth highest exponent.
- -1, if exponent not found

Returns in $v1:

- Coefficient of term with Nth highest exponent.
- 0, if exponent not found

Additional requirements:

- The function must not make any changes to main memory except as necessary.

Test Cases:

1. p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `remove_Nth_term(p,terms,1)` => 3 in $v0 and 4 in $v1
   SideEffect:
   p->head_term = Address of *Term(2,2,next_term0)*
   next_term0 = Address of *Term(5,0,NULL)*

2. p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `remove_Nth_term(p,terms,3)` => 0 in $v0 and 5 in $v1
   SideEffect:
   p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,NULL)*

3. p->head_term = Address of *Term(4,3,next_term0)*
   next_term0 = Address of *Term(2,2,next_term1)*
   next_term1 = Address of *Term(5,0,NULL)*

   `remove_Nth_term(p,terms,6)` => -1 in $v0 and 0 in $v1
   SideEffect: None

Part 7: Polynomial Addition

```
int add_poly(Polynomial* p, Polynomial* q, Polynomial* r)
```

The function `add_poly` takes as input pointers to two polynomials, `p` and `q` and a third pointer `r`. Both `p` and `q` contain pointers to the head term in the linked list representing the corresponding polynomials. The function adds `p` and `q` and stores the result in a new linked list with the head term pointer in `r`.

Addition of two polynomials, `p` and `q`, works by comparing the terms in `p` and `q`, starting at their head terms and moving towards the end one by one. Three cases arise while performing this comparison:

1. *The exponent in the current term in p is equal to the exponent in the current term in q*. In this case, the coefficients in the two nodes are added and a new term is created with the values:
    a. r -> coefficient = (p -> coefficient + q -> coefficient) and
    b. r -> exponent = p -> exponent
2. *The exponent of the current term in p is greater than the exponent in the current term in q*. In this case, a term with the coefficient and exponent of the current term in `p` is created and inserted to the result linked list.
3. *The exponent of the current term in p is less than the exponent in the current term in q*. In this case, a term with the coefficient and exponent of the current term in `q` is created and inserted to the result linked list.

If polynomial p has more terms than polynomial q or vice-versa after this comparison, then the remaining terms in either p or q are inserted at the end of the result linked list.

Tip: You can collect the terms that need to be inserted in a dynamically allocated array and use `add_terms_to_polynomial` to add all terms.

The function returns 1 in $v0 if addition is successful. If any failure is encountered, then the function returns 0 in $v0. It is your responsibility to think of what (if any) failures may occur. *An example of a failure is when both p and q are empty.*

Returns in $v0:

- 0 or 1.

Additional requirements:

- The function must not make any changes to main memory except as necessary.

Test Cases:

1. p->head_term = Address of Term(5,2,next_term0)
   next_term0 = Address of Term(7,1,NULL)
   q->head_term = Address of Term(3,2,next_term0)
   next_term0 = Address of Term(1,1,NULL)
   r->head_term = NULL

   `add_poly(p,q,r) => 1`
   SideEffect:
   r->head_term = Address of Term(8,2,next_term0)
   next_term0 = Address of Term(8,1,NULL)

2. p->head_term = Address of Term(5,2,next_term0)
   next_term0 = Address of Term(7,1,NULL)
   q->head_term = Address of Term(3,3,next_term0)
   next_term0 = Address of Term(1,2,NULL)
   r->head_term = NULL

   `add_poly(p,q,r) => 1`
   SideEffect:
   r->head_term = Address of Term(3,3,next_term0)
   next_term0 = Address of Term(6,2,next_term1)
   next_term1 = Address of Term(7,1,NULL)

3. p->head_term = Address of Term(5,2,NULL)
   q->head_term = Address of Term(-5,2,NULL)
   r->head_term = NULL

   `add_poly(p,q,r) => 0`
   SideEffect:
   r->head_term = NULL, because a polynomial cannot have 0 coefficient. In this case, the terms with coefficient 5 and -5 will cancel each other out.

4. p->head_term = Address of Term(5,2,NULL)
   q->head_term = NULL
   r->head_term = NULL

   `add_poly(p,q,r) => 1`
   SideEffect:
   r->head_term = Address of Term(5,2,NULL).

Part 8: Polynomial Multiplication

```
int mult_poly(Polynomial* p, Polynomial* q, Polynomial* r)
```

The function `mult_poly` takes as input pointers to two polynomials, `p` and `q`. Both `p` and `q` contain pointers to the head term in the linked list representing the corresponding polynomials. The function stores the product of `p` and `q` in a new linked list with the head term pointer in `r`.

Polynomial multiplication works by multiplying each term in `p` with each term in `q`. Specifically, for each term in `p` we visit all the terms in `q`; the exponent values in two terms are added as follows to obtain a new exponent:

```
new_exponent = (p -> exponent + q -> exponent).
```

The coefficient values, on the other hand, are multiplied as follows to obtain a new coefficient:

```
new_coefficient = (p -> coefficient × q -> coefficient).
```

Assume that the coefficient product will always be 32-bits.

Two cases arise when constructing `r`:

1. *A term with* `new_exponent` *already exists in* `r`*.* In such a case, update `r -> coefficient` as:
   `r -> coefficient = r -> coefficient + (p -> coefficient × q -> coefficient).`
2. *No term with* `new_exponent` *exists in* `r`. In such a case, create a new term and insert it in `r`.

The function returns 1 in $v0 if multiplication is successful. If any failure is encountered, then the function returns 0 in $v0. It is your responsibility to think of what (if any) failures may occur. *An example of a failure is when both p and q are empty.*

Returns in $v0:

- 0 or 1.

Additional requirements:

- The function must not make any changes to main memory except as necessary.

Test Cases:

1. p->head_term = Address of Term(5,2,next_term0)
   next_term0 = Address of Term(7,1,NULL)
   q->head_term = Address of Term(3,2,next_term0)
   next_term0 = Address of Term(1,1,NULL)
   r->head_term = NULL

   `mult_poly(p,q,r) => 1`
   SideEffect:
   r->head_term = Address of Term(15,4,next_term0)
   next_term0 = Address of Term(26,3,next_term1)

next_term1 = Address of Term(7,2,NULL)

2. p->head_term = Address of Term(5,2,next_term0)
   next_term0 = Address of Term(7,1,NULL)
   q->head_term = Address of Term(3,3,next_term0)
   next_term0 = Address of Term(1,2,NULL)
   r->head_term = NULL

   `mult_poly(p,q,r) => 1`
   SideEffect:
   r->head_term = Address of Term(15,5,next_term0)
   next_term0 = Address of Term(26,4,next_term1)
   next_term1 = Address of Term(7,3,NULL)

3. p->head_term = Address of Term(5,2,next_term0)
   next_term0 = Address of Term(7,1,NULL)
   q->head_term = Address of Term(-5,2,next_term0)
   next_term0 = Address of Term(1,1,NULL)
   r->head_term = NULL

   `mult_poly(p,q,r) => 1`
   SideEffect:
   r->head_term = Address of Term(-25,4,next_term0)
   next_term0 = Address of Term(-30,3,next_term1)
   next_term1 = Address of Term(7,2,NULL)
4. p->head_term = NULL
   q->head_term = Address of Term(-5,2,next_term0)
   next_term0 = Address of Term(1,1,NULL)
   r->head_term = NULL

   `mult_poly(p,q,r) => 1`
   SideEffect:
   r->head_term = Address of Term(-5,2,next_term0)
   next_term0 = Address of Term(1,1,NULL)

How to Submit Your Work for Grading

Go to **Assignments->Assignment 5** on **Blackboard** and submit **hw5.asm**.

You can submit multiple times. We will grade your most recent submission before the due date.