ECE 150: Fundamentals of Programming

(Sections 001 and 002)

Project 2

Deadline: Tuesday November 20, 2018

Problem 0: Polynomial arithmetic

In this problem, we will use dynamic allocation to perform polynomial math. We will use the following `poly_t` struct.

```
struct poly_t {
    double *a_coeffs;      // array of coefficients
    unsigned int degree;   // the degree of the polynomial
};
```

The member variable `a_coeffs` (*coefficients*) is a pointer that will store the address of a dynamically allocated array of doubles. Index zero of the coefficients corresponds to the zeroeth term (constant term). For example, the polynomial $7x^3 - 3x + 6$ has the following coefficients array:

0	1	2	3
6	-3	0	7

The `degree` member variable indicates the degree of the polynomial. For example, the polynomial $7x^3 - 3x + 6$ has a degree equal to 3.

Notice that the degree will always be one less than the capacity of the array.

The zero polynomial will be defined to have a degree equal to 0, and it should be represented by an array of capacity `1` storing the value `0`.

When you declare a local variable to be an instance of a `poly_t` data structure, you should initialize its member variables with nullptr and 0:

```
poly_t my_poly{nullptr, 0};
```

Header file

You are provided with a header file (Polynomial.h) that includes all the function declarations as well as the struct definition. For testing, copy this file into the same folder as your code. Add the following statement to the top of your code file:

```
#include "Polynomial.h"
```

You do not need to submit this header file to Marmoset, as Marmoset will provide its own copy of this file.

Polynomial initialization

You are asked to implement the following functions:

```
void init_poly( poly_t &p, double const init_coeffs[],
                           unsigned int const init_degree );
```

This function initializes a polynomial *p* by dynamically allocating memory for an array of sufficient dimension to store a polynomial of the specified degree. If the member function `a_coeffs` is not assigned `nullptr`, you should assume the value assigned is the address of a previously dynamically allocated array and you should therefore delete it. The coefficient array passed will contain the degree + 1 coefficients, and the entries of the array should be copied over to the newly created coefficient array.

Polynomial destructor

```
void destroy_poly( poly_t &p );
```

This function deallocates the dynamically allocated memory allocated for the passed `poly_t` data structure and assigns that member variable the value `nullptr`. This should be called immediately prior to any local variable declared to be of this type goes out of scope.

Polynomial degree

```
unsigned int poly_degree( poly_t const &p );
```

The `poly_degree(…)` function returns the degree of the polynomial.
If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Polynomial coefficient

```
double poly_coeff( poly_t const &p, unsigned int n );
```

The `poly_coeff(…)` function returns the coefficient of x^n where the coefficient of any term greater than the degree is equal to 0.
If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Polynomial evaluation

```
double poly_val( poly_t const &p, double const x );
```

The `poly_val(…)` function (*poly*nomial *eval*uate) returns the value of the polynomial $p(x)$ evaluated at the given argument x. For example, if $p(x) = 7x^3 - 3x + 6$, then this evaluated at $x = 1.1$ should evaluate to the value 12.017. You should use Horner's rule for polynomial evaluation, as this is the most efficient implementation.

If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Polynomial addition

```
void poly_add( poly_t &p, poly_t const &q );
```

The `poly_add(…)` adds the polynomial $q(x)$ onto the polynomial $p(x)$. You will usually have to allocate a new coefficient array and subsequently delete the old coefficient array, as the degree of $p(x)$ may change. Ensure that the degree is correct, for the sum of two polynomials of the same degree may have a degree less than either.
If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Polynomial subtraction

```
void poly_subtract( poly_t &p, poly_t const &q );
```

The `poly_subtract(…)` subtracts the polynomial $q(x)$ from the polynomial $p(x)$. You will usually have to allocate a new coefficient array and subsequently delete the old coefficient array, as the degree of $p(x)$ may change. Ensure that the degree is correct, for the difference of two polynomials of the same degree may have a degree less than either.
If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Polynomial multiplication

```
void poly_multiply( poly_t &p, poly_t const &q );
```

The `poly_multiply(…)` multiplies the polynomial $p(x)$ by the polynomial $q(x)$. You will usually have to allocate a new coefficient array and subsequently delete the old coefficient array, as the degree of $p(x)$ may change. Ensure that the degree is correct, for unless either argument is the zero polynomial, the degree of the product of two polynomials is the sum of the degrees.
If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Polynomial division

```
double poly_divide( poly_t &p, double r );
```

The `poly_divide(…)` function performs a polynomial division on $p(x)$ by dividing out the term $(x - r)$. It returns the remainder, which in this case must be a constant. If the argument r is a root of the polynomial, the remainder should be zero. If the polynomial is already of degree 0, the quotient will be 0 and the remainder will be the constant coefficient.

For example, starting with the polynomial $p(x) = x^4 - 15x^2 + 10x + 24$:

1. If we divide through by $(x - 3)$ where 3 is a root, the result is the polynomial $x^3 + 3x^2 - 6x - 8$ with a remainder of 0; after all, note that
$$x^4 - 4x^2 + 10x + 24 = (x - 3)(x^3 + 3x^2 - 6x - 8) + 0.$$
2. If we divide the result by $(x - 5)$ where 5 is not a root, the result is the polynomial $x^2 + 8x + 34$ with a remainder of 162; after all, note that
$$x^3 + 3x^2 - 6x - 8 = (x - 5)(x^2 + 8x + 34) + 162.$$
3. If we divide the result by $(x + 1)$ where 5 is not a root, the result is the polynomial $x + 7$ with a remainder of 27; after all, note that
$$x^2 + 8x + 34 = (x + 1)(x + 7) + 27.$$

If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Polynomial differentiation

```
void poly_diff( poly_t &p );
```

The `poly_diff(…)` function differentiates the given polynomial $p(x)$ and updates the polynomial to be the result of the differentiation. Unless the polynomial is already a constant polynomial, in which case the derivative is the zero polynomial, the derivative of a polynomial will have a degree one less than that of the original polynomial. You will have to create a new coefficient array of the appropriate size, properly initialize those entries, delete the old coefficient array, and then assign the new coefficient array to the appropriate member variable.
If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

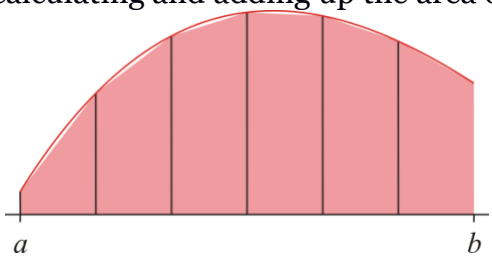
Polynomial integral approximation

```
double poly_approx_int( poly_t const &p,
                        double a, double b, unsigned int n );
```

The `poly_approx_int(…)` function approximates the integral

$$\int_a^b p(x)\,dx$$

by using the *composite trapezoidal rule*. The interval $[a, b]$ is divided into n sub-intervals, and the area of the trapezoids defined by the polynomial on each interval are added up. For example, in approximating the below polynomial with $n = 6$, we are approximating the area under the red polynomial by calculating and adding up the area of the six trapezoids in pink:



If $h = \frac{b-a}{n}$, then $x_k = a + kh$ for $k = 0, \ldots, n$ where $x_0 = a$ and $x_n = b$. Then the sum of the area of the trapezoids is equal to

$$\int_a^b p(x)\,dx \approx \frac{h}{2}\left(p(x_0) + 2p(x_1) + 2p(x_2) + 2p(x_3) + \cdots + 2p(x_{n-1}) + p(x_n)\right)$$

where the right-hand side is the composite trapezoidal rule.
If the member variable `a_coeffs` equals the `nullptr`, throw the integer `0`.

Marmoset Instructions

The Marmoset submission filename is Polynomial.cpp.

If you create your own int main() function for testing purposes, enclose it within the following preprocessor directives:

```
#ifndef MARMOSET_TESTING
```

```
int main();
#endif
```

Suggested modifications

1. Add instructions for Marmoset.
2. Typo in poly_multiplication() description. Change ‘poly_add’ to ‘poly_multiplication’.
3. Type in poly_divide() description. In step 3, change ‘5’ to ‘-1’.
4. Provide header file.
5. Make parameters for poly_degree(), poly_coeff(), and polyval() const.
6. destroy_poly() – add instruction to set p.degree to 0.