2a)

The language I used to write this program is C++.

The program I wrote is a text-based single-variable polynomial calculator. (It will only take derivatives of polynomial expressions, in which there is only one variable: x) Its purpose is to take derivatives of those kinds of expressions.

The video first shows syntax rules for input. The rest of the video demonstrates how to run the program for it to be able to fulfill its purpose of taking the derivative of the expressions in the input, as well as where to find the result.

2b)

To begin programming my calculator, I started by implementing simple versions of an input reader, a power rule function, and a function to output an expression. Each was iteratively tested for functionality before the product rule was added.

One difficulty I had was coming up with a way to allow expressions to be inside other expressions, with the expressions inside being dynamically allocated. I had originally used a pointer to an expression, but that would not work for multiple expressions inside one another. This was collaborative discussion. We considered using pointers or linked lists. I settled on putting vectors (linked lists) of expressions inside expressions because vectors are both dynamically allocated and easy to use.

I encountered another difficulty when I needed two functions to be able to call each other. This was not possible in the current program because one of the functions was declared before the other. To resolve this, I created function prototypes and put the implementations after all of the prototypes.

After all of the functions proved they worked at least for a few tests, I incremented my calculator to be able to simplify its input and output so it was easier to read.

2c)

The selected code segment implements derivative_inside(). This algorithm takes the derivative of the sum or product of expressions inside a given expression and returns the result as an expression.

```
struct expression derivative_inside(struct expression in)
{
        struct expression ans;
        if(in.inside.size() == 0)
        {
                ans.power = 0;
                ans.coefficient = 0;
                ans.type = false;
                return ans;
        }
        if(in.type == false) // added together
        {
                ans.type = false; // sum of the individual derivatives
                ans.coefficient = 1;
                ans.power = 1;
                for(int i = 0; i < in.inside.size(); i++)
                {
                        ans.inside.push_back(derivative(in.inside[i]));
                }
                return ans;
        }
        if(in.type == true) // multiplied together
        {
                // product rule...
                return product_rule(in);
        }
}
```

derivative_inside() accomplishes this by first determining if the input expression is a sum or product of expressions. The derivative of the sum of expressions is the sum of the derivatives of the summands, evaluated using derivative(). For products, it simply calls on product_rule(), which is designed to handle this case.

```
struct expression derivative(struct expression in)
{
        if(in.inside.size() == 0)
        {
                return power_rule(in);
        }
        else
        {
                return chain_rule(in);
        }
}
```

derivative() takes the derivative of an expression (including coefficient and exponent). It calls power_rule() when there is only one expression inside, and chain_rule() otherwise.

```
struct expression product_rule(struct expression in)
{
        struct expression ans;

        if(in.type != true)
        {
                cout << "product rule was given a sum" << endl;
                exit(2);
        }

        ans.type = false; // sum of the products, it's a holder for the result
        ans.coefficient = ans.power = 1; // just defaults

        struct expression terms; // this is the stuff multiplied together
        terms.inside = in.inside;
        terms.coefficient = terms.power = 1;
        terms.type = true;

        for(int i = 0; i < terms.inside.size(); i++) // applying the generalized product rule
        {
                struct expression temp;
                temp = terms.inside[i];

                terms.inside[i] = derivative(temp);

                ans.inside.push_back(terms);

                terms.inside[i] = temp;
        }

        return ans;
}
```

product_rule() functions independently and takes the derivative of a series of expressions multiplied together. It does this by maintaining the product of all of the expressions, but one at a time taking the derivative of each of the expressions and adding all of those products together for a final answer.

```
struct expression power_rule(struct expression in)
{
        struct expression ans;
        if(in.power == 0) // it is a constant so just set everything to 0
        {
                ans.power = 0;
                ans.coefficient = 0;
                return ans;
        }

        ans.coefficient = in.coefficient * in.power; // apply the power rule
        ans.power = in.power - 1;
        ans.type = in.type;
        ans.inside = in.inside;

        return ans;
}
```

```
struct expression chain_rule(struct expression in)
{
        struct expression ans;
        ans.type = true; // product of f'(g(x)) and g'(x), so we need a new expression to contain them

        ans.inside.push_back(derivative_inside(in));

        in = power_rule(in); // only affects outer layer, f'(g(x))

        ans.inside.push_back(in);

        ans.coefficient = 1;
        ans.power = 1;

        return ans;
}
```

power_rule() and chain_rule() apply the power rule and chain rule respectively.

The selected algorithm, derivative_inside(), is required for taking derivatives, and it does this by organizing the rules into a process by which derivatives can be taken, which is the overall purpose of the program.

2d)

```cpp
struct expression
{
	bool type; // true for product, false for sum

	double power;

	double coefficient;

	vector <struct expression> inside;

	// returns if expression evaluates to 0
	bool is_zero() const
	{
		if(coefficient == 0)
		{
			return true;
		}
		return false;
	}

	// returns if function is a constant
	bool is_constant() const
	{
		if(coefficient != 0 and power == 0)
		{
			return true;
		}
		return false;
	}

	// used by the sort functions, it just needs to put identical things next to each other
	bool operator< (const struct expression other) const
	{
		if(inside.size() == other.inside.size()) // on this case, we want to separate expressions that are not equal from each other,
		{
			return power < other.power;
		}

		return inside.size() < other.inside.size();
	}
};
```

One abstraction that I used in the implementation of my program is my struct: expression. This struct is used to represent a polynomial expression. The struct definition uses mathematical and logical concepts in that it has both integers, a boolean, a vector, and helper functions which use logical concepts. This struct helped me manage the complexity of my program by allowing integers, a boolean and a vector of expressions to be easily bundled into a struct with guaranteed properties that I can easily call upon anywhere in the program without having to think about each of the individual pieces of data in the struct. The use of the struct is similar to that of the use of an expression, so it helps the programmer conceptually to think of the struct as an expression instead of a collection of assorted pieces of data. This helps manage the complexity of the program, as a collection of data of various types is conceptually harder to work with as opposed to a struct, even though they contain the same data. In other words, the struct manages the complexity of the program by making the use of the data more intuitive.