UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

Assignment 2 Specifications: Polynomial Calculator

Release date: 18-09-2023 at 06:00
Due date: 13-10-2023 at 23:59
Total Marks: 660

Contents

1 General Instructions **2**

2 Plagiarism **3**

3 Outcomes **3**

4 Introduction **3**

5 Tasks **4**
5.1 term . . . 5
5.2 polynomial . . . 10
5.3 univariate . . . 12
5.4 bivariate . . . 15
5.5 linear . . . 18
5.6 quadratic . . . 19
5.7 circle . . . 20
5.8 ellipse . . . 21

6 Memory Management **22**

7 Testing **22**

8 Upload checklist **23**

9 Allowed libraries **23**

10 Submission **23**

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of C++'s built-in data structures. Doing so will result in a mark of zero. You may only make use of 1-dimensional native arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a mark of zero. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with C++98**

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this assignment, you will have gained experience with the following:

- Operator overloading.
- Polymorphism.

4 Introduction

Polynomials are mathematical objects which are expressions of variables and coefficients added together. These polynomials can use operator overloading in a very intuitive manner since most of the operators that we can overload are mathematical operators, which can also be applied to polynomials.

In this assignment, we will create a polynomial class, as well as a few derived classes, which will inherit some of the basic functionality of polynomials but add their own restrictions.

5 Tasks

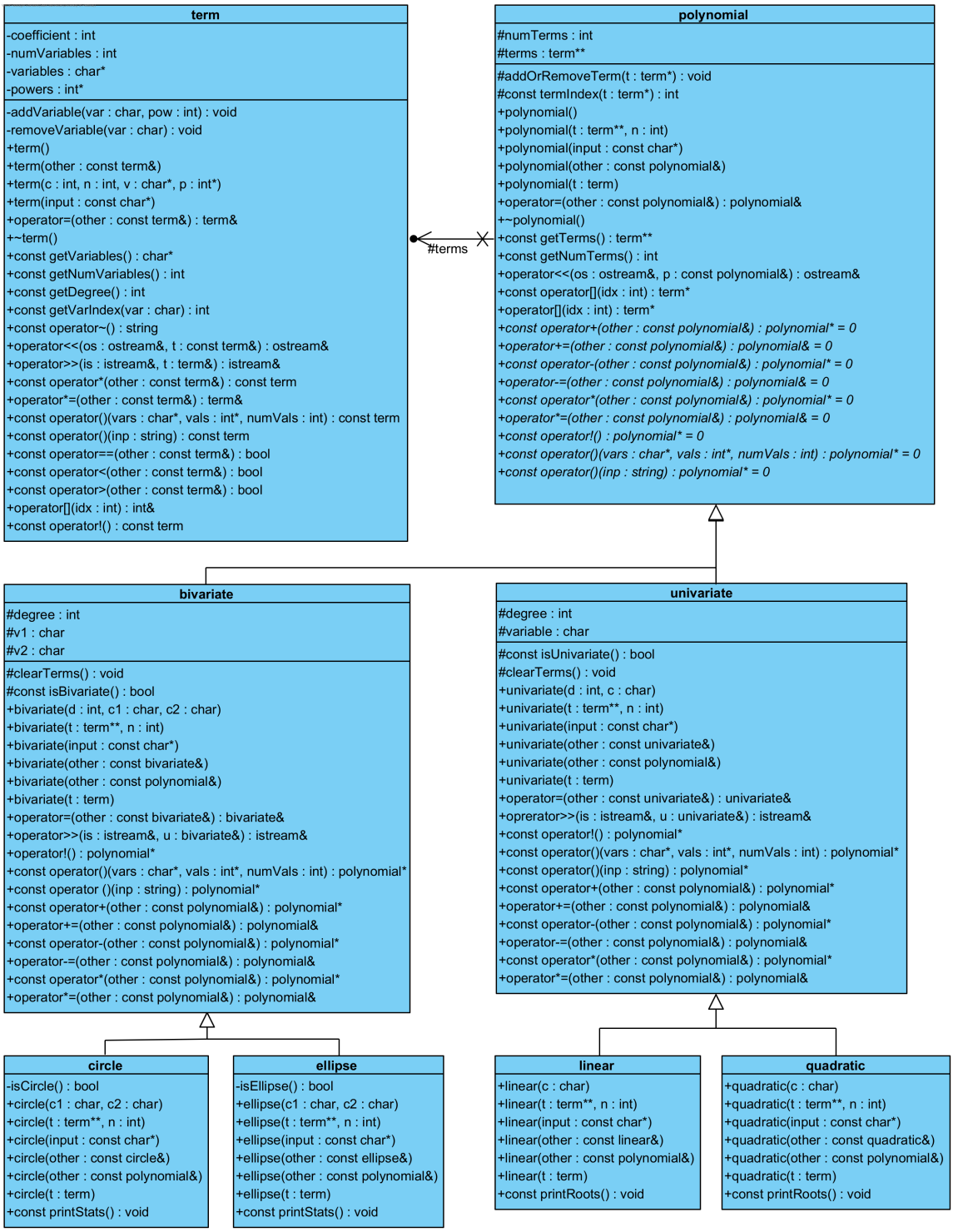

Figure 1: UML

Note 1: This assignment contains a lot of constant functions. This was done to try lead students in the right direction, by not allowing students to make changes to objects where they should not do that. In the UML and the spec, to indicate constant functions the word const will be added to the front of the function name.

Note 2: Most of this diagram had to be fixed by hand since the software used to generate this does not support all of the C++ features. You are given the header files for this assignment. Thus if you see inconsistencies on the diagram, you may assume that the provided header files are correct.

5.1 term

This class will be used as the terms inside polynomials. A term in a polynomial consists of a coefficient, which for simplicity reasons, will be an integer value. A term will also contain a number of variables. Every variable will have a positive power. Note that a variable to the power 0, won't be stored because this is the same as multiplying by 1. Negative powers are not allowed since polynomials can only contain positive powers.

- Members *For the explanations listed below, the example:* $10x^2y^3z^4$ *will be used.*
  - coefficient: int
    - This is the number that forms the first part of the term.
    - In the example, this will have the value 10.
    - If this is 0, that means the whole term is 0. Thus, numVariables should be 0, and the sizes of the variables array and powers array should also be 0. **If this is 0 at any time, then the arrays should be cleared to achieve this.**
    - The default value for this member variable is 1.
  - numVariables: int
    - This is the number of unique variables within the term.
    - In the example, this will have the value 3.
    - The default value for this is 0.
  - variables: char*
    - This is a character array which holds the unique variables within the term.
    - In the example, this will have the value [x,y,z].
    - This array will always be of length numVariables.
    - This array will always be sorted alphabetically.
    - The default value for this is an array of length 0.
  - powers: int*
    - This is an integer array which holds the powers of the term.
    - In the example, this will have the value [2,3,4].
    - This array will always be of length numVariables.
    - The indices of this array will match the variables array. Thus, the value at position 0, is the power of the variable that is stored in variables[0].
    - The default value for this is an array of length 0.
- Functions
  - addVariable(var: char, pow: int): void
    - Please note, in a mathematical sense this should actually be seen as multiplication, but programming-wise, you are adding the variable to the array.
    - This function will attempt to add a variable to the term.
    - If the term's coefficient is 0, then no variables should be added.
    - If the passed-in power is negative or 0, then don't add this variable.
    - If the passed-in variable is already in the term, then increase the power of the variable by the passed-in amount. For example: If the term is $2x^2$, and you call this function with parameters var = x, pow = 3, then the term should change to $2x^5$.
    - If the passed-in variable is not already in the term, then the passed-in parameters should be added to the corresponding arrays. **Make sure that the variables array stays sorted.**

- removeVariable(var: char): void
  - This function will attempt to remove a variable from the term.
  - If the passed-in variable is within the term, then remove that variable from the variables array. Remember to also remove the corresponding power. **Make sure that the variables array stays sorted and that there are no gaps.**
- term()
  - This is the default constructor.
  - All member variables should be set to their default values.
- term(other: const term&)
  - This is the copy constructor for the term class.
  - Deep copies (where applicable) should be made of all member variables.
- term(c: int, n: int, v: char*, p: int*)
  - This is a parameterized constructor.
  - c is the coefficient of the term.
  - n is the number of variables in the term.
  - v is the array of variables.
  - p is the array of powers.
  - Deep copies should be made where applicable.
  - Note that the passed-in variables might not be sorted, but the member variable must be sorted.
- term(input: const char*)
  - This is a parameterized constructor.
  - The passed-in parameter is a c-string. This should be converted to a string for ease of use.
  - The passed-in string will be used to initialize the term.
  - The coefficient may be positive or negative.
  - If the coefficient is ± 1, the 1 may be written or omitted. The + sign can also be written out or omitted.
  - After the coefficient, there will be a *. This represents multiplication. *Note that it is possible for the variables to start immediately, in this case, the coefficient is 1.*
  - There will be no spaces in the passed-in string.
  - Every variable will be separated using a *.
  - If the power of a variable is 1, then the exponent may be written out or omitted.
  - If a variable has a power, it will be indicated by ^. The number after the caret is the power.
  - The order of the variables can be in any order, but this should be sorted when storing the variables.
  - It is also possible for variables to be repeated. In this case, the powers should be added.
  - **Remember to use the functions you already coded.**
  - See Figure 2 for examples.
- operator=(other: const term&): term&
  - This is the assignment operator.
  - Make sure to make deep copies, where applicable.

- ∼ term()
  - This is the destructor for the class.
  - Deallocate all dynamic memory to prevent memory leaks.
- const getVariables(): char*
  - This is a getter for the variables member.
- const getNumVariables(): int
  - This is a getter for the numVariables member.
- const *getDegree(): int
  - The degree of a term is obtained by summing over all the values in the powers array.
- const getVarIndex(var: char): int
  - This function returns the position of the passed-in parameter in the variables array.
  - If the passed-in parameter is not inside the variables array then return -1.
- const operator∼(): string
  - This returns a string representation of the term.
  - If there are no variables, then only return the coefficient.
  - If the coefficient is ±1, then the 1 should be omitted.
  - If there is a power equal to 1, then "^1" should be omitted.
  - The variables should be separated using *.
  - There should be no spaces or newlines added.
  - See Figure 2 for examples.
- operator«(os: ostream&, t: const term&): ostream&
  - This will be used to print out the term.
  - Use the result from the ∼ operator and return this on its own line.
- operator»(is: istream&, t: term&): istream&
  - This will be used to create a term.
  - If this is called on an existing term, then the previous data should be overwritten.
  - The same rules as the parameterized constructor taking a string should be used.
- const operator*(other: const term&): const term
  - This operator should not change the current object.
  - A term should be returned, which is calculated by multiplying the two terms.
  - The coefficients of the terms should be multiplied together.
  - The same rules as addVariable() should be followed for the variables and powers.
- operator*=(other: const term&): term&
  - The same rules as operator * should be followed.
  - The only change is that this time, the result should be stored in the current object.

- const operator()(vars: char*, vals: int*, numVals: int): const term
  - This is the substitution operator.
  - *numVals* is the size of the *vars* and *vals* arrays.
  - *vars* is an array of characters which represent variables.
  - *vals* is an array of numbers which represent numbers that should be substituted in.
  - This function should **not** change the current object.
  - The result should be a term where all the corresponding variables are substituted.
  - Only substitute the variables which are in the term. Remember to take their power into account.
  - Example: If we have a term t = $2x^2y^3z^4$, and we call substitute with the following parameters : t(`[x,a]`,`[2,0]`,2), then the result should be the term $8y^3z^4$.
- const operator()(inp: string): const term
  - This is also a substitution operator.
  - The same rules as the previous version should be followed.
  - This time, the input is a string. The format is as follows: {Variable}={value}. Note the brackets are not part of the format. If there are multiple substitutions, these are separated by spaces.
  - The example in the previous version can thus be called using: t("x=2 a=0").
- const operator==(other: const term&): bool
  - Terms are considered equal if they can be added (this time, we mean it in a mathematical sense) together.
  - The coefficients of the terms are irrelevant.
  - The terms are equal if they have the same variables and powers.
- const operator<(other: const term&): bool
  - This will be used for sorting purposes in the polynomial class.
  - If the terms are equal, return false.
  - The relational checks should be done in the following order.
    1. If the LHS's numVariables is 0, then return false.
    2. If the RHS's numVariables is 0, then return true.
    3. Start a for loop that loops to the smallest numVariables.
       - If the variables at this index are equal, then compare the powers. If the powers are also equal, then go to the next index. If they are not equal, then return true if the LHS's power is larger than the RHS's power; otherwise, return false.
       - If the LHS's variable is greater than the RHS's variable, then return false; otherwise, return true.
    4. If the LHS's numVariables are less than the RHS's numVariables then return true; otherwise, return false;
- const operator>(other: const term&): bool
  - This will be used for sorting purposes in the polynomial class.
  - If the terms are equal, return false.
  - The checks should return the ***opposite*** of the operator <'s rules.

- operator[](idx: int): int&
  - This is the subscript operator.
  - If the passed-in parameter is inside the valid range of the powers array, then return the power at that index.
  - If the passed-in parameter is not a valid index, then return the ***coefficient***.
- const operator!(): const term
  - This is the negation operator.
  - The current object should not be altered.
  - The negation of a term should swap the sign of the coefficient.

input	coefficient	numVariables	variables	powers	degree	~
10*x^2*w^3*a^1	10	3	[a,w,x]	[1,3,2]	6	10*a*w^3*x^2
+8*x*x^1*x^0*x^-1	8	1	[x]	[2]	2	8*x^2
1*x^2	1	1	[x]	[2]	2	x^2
-x^2	-1	1	[x]	[2]	2	-x^2
0*x^2	0	0	[]	[]	0	0
+z^-3	1	0	[]	[]	0	1

Figure 2: Examples for term

5.2 polynomial

This is an abstract class that will be used to represent a polynomial. A polynomial is a collection of terms which are added together.

- Members
  - numTerms: int
    - This is the number of terms in the polynomial.
    - The default value for this is 0.
  - terms: term**
    - This is a 1D dynamic array of dynamic term objects of size numTerms.
    - This will store the terms in the polynomial.
    - This array will always be sorted ascending using the term relational operators.
    - The default value for this is an array of size 0.
- Functions
  - addOrRemoveTerm(t: term*): void
    - This function will be used to add or remove terms from the polynomial.
    - If there is a term in the polynomial which is equal (according to term equality) to the passed-in parameter, then add the coefficients together. If the resulting coefficient is zero, then remove this term from the array.
    - If there is no term which is equal to the passed-in parameter, then the passed-in parameter should be added to the terms array (making a deep copy). **Make sure the terms array stays sorted.**
  - const termIndex(t: term*): int
    - If there is a term inside the terms array which is equal, according to term equality, to the passed-in parameter, then return the index of this term.
    - If the term was not found, then return -1.
  - polynomial()
    - This is the default constructor.
    - Initialize all parameters to the default values.
  - polynomial(t: term**, n: int)
    - This is a parameterized constructor.
    - t is a 1D dynamic array of dynamic term objects of size n.
    - Use the addOrRemoveTerm function to add the terms to the polynomial.
    - Note that the passed-in array might not be sorted, but the member variable must be sorted.
  - polynomial(input: const char*)
    - This is a parameterized constructor.
    - The passed-in parameter is a c-string. This should be converted to a string for ease of use.
    - The passed-in string will be used to initialize the term.
    - Terms will be separated by a space, followed by a $\pm$, followed by a space. If its the first term, the first space will be left out.
    - See Figure 3 for examples.

- polynomial(other: const polynomial&)
  - This is the copy constructor.
  - Deep copies should be made of all member variables.
- polynomial(t: term)
  - This is a parameterized constructor.
  - Make a deep copy of the passed-in parameter and set that as the first term.
- operator=(other: const polynomial&): polynomial&
  - This is the assignment operator.
  - Make deep copies of the passed-in parameter's members.
- ∼polynomial()
  - This is the destructor for the class.
  - Make sure that no memory gets leaked.
- const getTerms(): term**
  - Return the terms variables.
- const getNumTerms(): int
  - Return the numTerms variable.
- operator«(os: ostream&, p: const polynomial&): ostream&
  - This will print the string representation of the term on its own line.
  - If numTerms is 0, then print 0 on its own line.
  - Use the operator∼ for term to get the string representation of each term.
  - Separate the terms by a space, followed by a +, followed by a space.
  - See Figure 3 for examples.
- const operator[](idx: int): term*
  - If the index is valid, return the term at that index.
  - If the index is invalid, return null.
- operator[](idx: int): term*
  - This is the non-constant version of the previous operator.
  - It should follow the exact same rules as the previous operator.
- ***The rest of the functions in this class are pure virtual. Since these won't be implemented in this class, they will explained where they will be implemented.***

input	numTerms	terms	operator«
x^2*y^2 + 2*x*y + 1 + x^-3	3	[x^2*y^2,2*x*y,2]	x^2*y^2 + 2*x*y + 2
- -a^2*y^2 - -2*a*y - -1	3	[a^2*y^2,2*a*y,1]	a^2*y^2 + 2*a*y + 1
-x^2*b^1 - 2*x*b^5 - 1 + b^-5	2	[-2*b^5*x,-b*x^2]	-2*b^5*x + -b*x^2
- +c*y^2 - +2*c^3*y - +1	3	[-2*c^3*y,-c*y^2,-1]	-2*c^3*y + -c*y^2 + -1
+ -c^2*y^2 + -2*c*y + -1	3	[-c^2*y^2,-2*c*y,-1]	-c^2*y^2 + -2*c*y + -1
e^-3 + f^-2	1	[2]	2

Figure 3: Examples for polynomial*(Note there is an endline at the end of «)*

5.3 univariate

Univariate inherits publically from polynomial. This is a specialised version of polynomial, with 2 restrictions. The first restriction is that the polynomial can only contain one type of variable. The second restriction is a degree restriction. The univariate class has a degree variable. All terms in the univariate must have a degree less than or equal to the degree variable.

- Members
  - degree: int
    - This is the max degree for this univariate.
  - variable: char
    - This is the only variable allowed in this univariate.
- Functions
  - const isUnivariate(): bool
    - If the 2 restrictions mentioned at the start of this subsection are true, then return true. If there are any invalid terms, return false.
  - clearTerms(): void
    - This should set the polynomial parameters to their default values.
    - Don't change the degree or variable.
  - univariate(d: int, c: char)
    - This is a parameterized constructor.
    - Call the default polynomial constructor, and then set the degree and variable to the passed-in parameter.
  - univariate(t: term**, n: int)
    - This is a parameterized constructor.
    - Call the corresponding polynomial constructor.
    - Set *variable* equal to x, and *degree* to 2.
    - If the univariate has terms, then set *degree* to the degree of the first term.
    - If the first term contains a variable, set this object's *variable* to the first variable of the first term.
    - Use the isUnivariate() function to check if the current object is a valid univariate. If it is not a valid univariate, then call clearTerms.
  - univariate(input: const char*)
    - Follow the same rules as the previous constructor.
  - univariate(other: const univariate&)
    - This is the copy constructor.
  - univariate(other: const polynomial&)
    - Follow the same rules as the string constructor.
  - univariate(t: term)
    - Follow the same rules as the string constructor.
  - operator=(other: const univariate&): univariate&
    - This is the assignment operator.

- operator»(is: istream&, u: univariate&): istream&
  - This will be used to set the member variables of the passed-in parameter.
  - Extract one line from the passed-in stream.
  - Use the string constructor to create a univariate.
  - If the created univariate is a valid univariate, the passed-in parameter should be changed to this.
  - If the created univariate is not a valid univariate, the passed-in parameter should stay unchanged.
  - **Note: if you called the constructor, the result will always be univariate; thus, use the numTerms to see if clearTerms() was called.**
- const operator!(): polynomial*
  - This is the negation operator. This won't change the current object.
  - The returned univariate, should be made up of all the negated terms.
- const operator()(vars: char*, vals: int*, numVals: int): polynomial*
  - This is the substitution operator.
  - The returned univariate, should be the result of calling the substitution operator on every term.
- const operator()(inp: string): polynomial*
  - This is the substitution operator.
  - This should follow the same rules as the previous operator.
  - You may assume that the passed-in parameter will be the same format as the term substitution operator.
- const operator+(other: const polynomial&): polynomial*
  - This should return a univariate that is the result of adding (this will also be the correct mathematical addition meaning) the univariate and polynomial together. Note that the result might be an invalid univariate. This is fine as the return type is polynomial.
  - polynomial::addOrRemove() should be used to create a univariate which is the result of adding all the terms together.
- operator+=(other: const polynomial&): polynomial&
  - This operator might change the current object. If the result of adding the current object and the passed-in parameter is a valid univariate, then the current object should change to the result.
  - If the result of adding them together is not univariate, then the current object should stay unchanged.
- const operator-(other: const polynomial&): polynomial*
  - This should return a univariate that is the result of subtracting the passed-in parameter from the current object. Note that the result might be an invalid univariate. This is fine as the return type is polynomial.
  - *Hint: Subtraction is just adding the negation.*
- operator-=(other: const polynomial&): polynomial&
  - This operator might change the current object. If the result of subtracting the passed-in parameter from the current object is a valid univariate, then the current object should change to the result.
  - If the result of subtracting them is not univariate, then the current object should stay unchanged.

- const operator*(other: const polynomial&): polynomial*
  - This should return a univariate which is the result of multiplying the current object by the passed-in parameter. Note that the result might be an invalid univariate. This is fine as the return type is polynomial.
  - The distributive property of polynomial multiplication should be used. This means that you must multiply every term in the first polynomial with every term in the second polynomial. The results of these multiplications should then be added together.
  - For another explanation, click here.
- operator*=(other: const polynomial&): polynomial&
  - This operator might change the current object. If the result of multiplying the passed-in parameter with the current object is a valid univariate, then the current object should change to the answer.
  - If the result of multiplying them together is not univariate, then the current object should stay unchanged.
  - Note that the same variable and degree should be used as the current object.

5.4 bivariate

Bivariate inherits publically from polynomial. This is a specialised version of polynomial, with 2 restrictions. The first restriction is that the polynomial can have at most 2 types of variables. The second restriction is a degree restriction. The bivariate class has a degree variable. All terms in the bivariate must have a degree less than or equal to the degree variable.

- Members
  - degree: int
    - This is the max degree for this bivariate.
  - v1: char
    - This is one of the variables for the bivariate.
    - Note that there is no restriction about which variable is v1 and v2, other than that v1 cannot be the same as v2.
  - v2: char
    - This is one of the variables for the bivariate.
    - Note that there is no restriction about which variable is v1 and v2, other than that v2 cannot be the same as v1.
- Functions
  - const isBivariate(): bool
    - If the 2 restrictions mentioned at the start of this subsection are true, then return true. If there are any invalid terms, return false.
  - clearTerms(): void
    - This should set the polynomial parameters to their default values.
    - Don't change the *degree* or *variables*.
  - bivariate(d: int, c1: char, c2: char)
    - This is a parameterized constructor.
    - Call the default polynomial constructor, and then set the degree and variables to the passed-in parameters.
  - bivariate(t: term**, n: int)
    - This is a parameterized constructor.
    - Call the corresponding polynomial constructor.
    - Set the *v1* to x, v2 to y, and the *degree* to 2.
    - If the bivariate has terms, then set *degree* to the degree of the first term.
    - If the first term has a variable, set this object's v1 to the first variable of the first term.
    - Loop through all the variables in the polynomial and set v2 to the first variable which is not v1.
    - Use the isBivariate() function to check if the current object is a valid bivariate. If it is not a valid bivariate, then call clearTerms.
  - bivariate(input: const char*)
    - Follow the same rules as the previous constructor.
  - bivariate(other: const bivariate&)
    - This is the copy constructor.

- bivariate(other: const polynomial&)
  - Follow the same rules as the string constructor.
- bivariate(t: term)
  - Follow the same rules as the string constructor.
- operator=(other: const bivariate&): bivariate&
  - This is the assignment operator.
- operator»(is: istream&, u: bivariate&): istream&
  - This will be used to set the member variables of the passed-in parameter.
  - Extract one line from the passed-in stream.
  - Use the string constructor to create a bivariate.
  - If the created bivariate is a valid bivariate, the passed-in parameter should be changed to this.
  - If the created bivariate is not a valid bivariate, the passed-in parameter should stay unchanged.
  - **Note: if you called the constructor, the result will always be bivariate; thus, use the numTerms to see if clearTerms() was called.**
- const operator!(): polynomial*
  - This is the negation operator.
  - The returned bivariate should be made up of all the negated terms.
- const operator()(vars: char*, vals: int*, numVals: int): polynomial*
  - This is the substitution operator.
  - The returned bivariate should be the result of calling the substitution operator on every term.
- const operator()(inp: string): polynomial*
  - This is the substitution operator.
  - This should follow the same rules as the previous operator.
  - You may assume that the passed-in parameter will be the same format as the term substitution operator.
- const operator+(other: const polynomial&): polynomial*
  - This should return a bivariate that is the result of adding (this will also be the correct mathematical addition meaning) the bivariate and polynomial together. Note that the result might be an invalid bivariate. This is fine as the return type is polynomial.
  - polynomial::addOrRemove() should be used to create a bivariate which is the result of adding all the terms together.
- operator+=(other: const polynomial&): polynomial&
  - This operator might change the current object. If the result of adding the current object and the passed-in parameter is a valid bivariate, then the current object should change to the result.
  - If the result of adding them together is not bivariate, then the current object should stay unchanged.

- const operator-(other: const polynomial&): polynomial*
  - This should return a bivariate that is the result of subtracting the passed-in parameter from the current object. Note that the result might be an invalid bivariate. This is fine as the return type is polynomial.
  - *Hint: Subtraction is just adding the negation.*
- operator-=(other: const polynomial&): polynomial&
  - This operator might change the current object. If the result of subtracting the passed-in parameter from the current object is a valid bivariate, then the current object should change to the result.
  - If the result of subtracting them is not bivariate, then the current object should stay unchanged.
- const operator*(other: const polynomial&): polynomial*
  - This should return a bivariate which is the result of multiplying the current object by the passed-in parameter. Note that the result might be an invalid bivariate. This is fine as the return type is polynomial.
  - The distributive property of polynomial multiplication should be used. This means that you must multiply every term in the first polynomial with every term in the second polynomial. The results of these multiplications should then be added together.
  - For another explanation, click here.
- operator*=(other: const polynomial&): polynomial&
  - This operator might change the current object. If the result of multiplying the passed-in parameter with the current object is a valid bivariate, then the current object should change to the result.
  - If the result of multiplying them together is not bivariate, then the current object should stay unchanged.

5.5 linear

Linear inherits publically from univariate. This is a specialised version of univariate. In this class, the degree variable is set to 1.

- Functions
  - linear(c: char)
    - Call the univariate constructor using the passed-in parameter and the restriction mentioned at the start of this subsection.
  - linear(t: term**, n: int)
    - Call the univariate constructor using the passed-in parameters.
    - Note that the univariate constructor might use a different degree. Thus after calling the parent constructor, set the degree to match the restriction, and then call use isUnivariate() to check that the object is still univariate with this degree. If it is not univariate, then call clearTerms.
  - linear(input: const char*)
    - Follow the same rules as the previous constructor.
  - linear(other: const linear&)
    - Follow the same rules as the string constructor.
  - linear(other: const polynomial&)
    - Follow the same rules as the string constructor.
  - linear(t: term)
    - Follow the same rules as the string constructor.
  - const printRoots(): void
    - This function should print out the roots of the object.
    - The explanation will assume the polynomial has the form $mx + c$.
    - Note that the x can be any variable.
    - Extract the values of m and c from the linear polynomial.
    - If m is 0, then print out the following on its own line.

      ```
      No roots
      ```

    - If m is not 0, then print out the following on its own line, with the root displayed to 2 decimal values. Note that values in curly braces should be evaluated, and _ indicates spaces.

      ```
      Roots_:_{variable}_=_{-c/m}
      ```

5.6 quadratic

Quadratic inherits publically from univariate. This is a specialised version of univariate. In this class, the degree variable is set to 2.

- Functions
  - quadratic(c: char)
    - Call the univariate constructor using the passed-in parameter and the restriction mentioned at the start of this subsection.
  - quadratic(t: term**, n: int)
    - Call the univariate constructor using the passed-in parameters.
    - Note that the univariate constructor might use a different degree. Thus after calling the parent constructor, set the degree to match the restriction, and then call use isUnivariate() to check that the object is still univariate with this degree. If it is not univariate, then call clearTerms.
  - quadratic(input: const char*)
    - Follow the same rules as the previous constructor.
  - quadratic(other: const quadratic&)
    - Follow the same rules as the string constructor.
  - quadratic(other: const polynomial&)
    - Follow the same rules as the string constructor.
  - quadratic(t: term)
    - Follow the same rules as the string constructor.
  - const printRoots(): void
    - This function should print out the roots of the object.
    - The explanation will assume the polynomial has the form $ax^2 + bx + c$.
    - Note that the x can be any variable.
    - Extract the values of a, b and c from the quadratic polynomial.
    - If a is 0, then this is a linear polynomial of the from $bx + c$. Thus use the previous subsection.
    - If a is not 0, then calculate the discriminant of the quadratic equation:

$$d = b^2 - 4ac$$

    - If the discriminant is negative, then print the following on its own line

```
No roots
```

    - If d is positive, then print out the following on its own line, with the roots displayed to 2 decimal values. Note that values in curly braces should be evaluated, and _ indicates spaces. Note that you should still use this format even if the roots are the same. ***variable* has been shortened to *var* to allow the line to fit.**

```
Roots_:_{var}_=_{(-b+sqrt(d))/(2a)}_,_{var}_=_{(-b-sqrt(d))/(2a)}
```

5.7 circle

Circle inherits publically from bivariate. This is a specialised version of bivariate. In this class, the degree variable is set to 2. Additionally, we will only be using circles centred around the origin. Thus only three terms are allowed. The first and second terms must have a degree of 2, and the last term must have a degree of 0. The first two terms must also have the same coefficient.

- Functions
  - const isCircle(): bool
    - This function returns true if the current object is a valid circle. It returns false otherwise.
  - circle(c1: char, c2: char)
    - Call the bivariate constructor using the passed-in parameters and the restriction mentioned at the start of this subsection.
  - circle(t: term**, n: int)
    - Call the bivariate constructor using the passed-in parameters.
    - Note that the bivariate constructor might use a different degree. Thus after calling the parent constructor, set the degree to match the restriction, and then call use isCircle() to check that the object is still a circle with this degree. If it is not a circle, then call clearTerms.
  - circle(input: const char*)
    - Follow the same rules as the previous constructor.
  - circle(other: const circle&)
    - Follow the same rules as the string constructor.
  - circle(other: const polynomial&)
    - Follow the same rules as the string constructor.
  - circle(t: term)
    - Follow the same rules as the string constructor.
  - const printStats(): void
    - This function will print statistics about the circle.
    - The explanation will assume our circle has the equation: $cx^2 + dy^2 + f$.
    - Note that the x, y can be any variables.
    - Start by extracting the values for c, d and f.
    - If f is not negative or the circle is invalid, then print out the following on its own line:

      ```
      Not a real circle
      ```

    - Calculate the radius of the circle using $r = \sqrt{\frac{-f}{c}}$.
    - Look at the two character member variables. Save the smallest of these two in *c1* and the larger in *c2*.
    - Print out the following on its own line. Note that everything should be on one line, even though the spec shows multiple lines. All values should be displayed to 2 decimal values. Note that values in curly braces should be evaluated, and _ indicates spaces.

      ```
      Area_=_{pi*r*r}_units^2.Perimeter_=_{2*pi*r}_units.
      Intercepts_:_{c1}_=_{r}_,_{c1}_=_{-r}_,_{c2}_=_{r}_,_{c2}_=_{-r}
      ```

      Example

      ```
      Area = 28.27 units^2.Perimeter = 18.85 units.Intercepts : a = 3 , a
          = -3 , b = 3 , b = -3
      ```

5.8 ellipse

Ellipse inherits publically from bivariate. This is a specialised version of bivariate. In this class, the degree variable is set to 2. Additionally, we will only be using ellipses centred around the origin. Thus only three terms are allowed. The first and second terms must have a degree of 2, and the last term must have a degree of 0.

- Functions
  - const isEllipse(): bool
    - This function returns true if the current object is a valid ellipse. It returns false otherwise.
  - ellipse(c1: char, c2: char)
    - Call the bivariate constructor using the passed-in parameters and the restriction mentioned at the start of this subsection.
  - ellipse(t: term**, n: int)
    - Call the bivariate constructor using the passed-in parameters.
    - Note that the bivariate constructor might use a different degree. Thus after calling the parent constructor, set the degree to match the restriction, and then call use isEllipse() to check that the object is still an ellipse with this degree. If it is not an ellipse, then call clearTerms.
  - ellipse(input: const char*)
    - Follow the same rules as the previous constructor.
  - ellipse(other: const ellipse&)
    - Follow the same rules as the string constructor.
  - ellipse(other: const polynomial&)
    - Follow the same rules as the string constructor.
  - ellipse(t: term)
    - Follow the same rules as the string constructor.
  - const printStats(): void
    - This function will print statistics about the ellipse.
    - The explanation will assume our ellipse has the equation: $cx^2 + dy^2 + f$.
    - Note that the x, y can be any variables.
    - Start by extracting the values for c, d and f.
    - If f is not negative or the ellipse is invalid, then print out the following on its own line

      ```
      Not a real ellipse
      ```

    - Calculate the following values : $a = \sqrt{\frac{-f}{c}}, b = \sqrt{\frac{-f}{d}}$.
    - Look at the two character member variables. Save the smallest of these two in *c1* and the larger in *c2*.
    - Print out the following on its own line. The same formatting rules as circle apply.

      ```
      Area_=_{pi*a*b}_units^2.Perimeter_=_{2*pi*sqrt([a*a+b*b]/2)}_units.
      Intercepts_:_{c1}_=_{a}_,_{c1}_=_{-a}_,_{c2}_=_{b}_,_{c2}_=_{-b}
      ```

      Example

      ```
      Area = 28.27 units^2.Perimeter = 18.85 units.Intercepts : a = 3 , a
         = -3 , b = 3 , b = -3
      ```

6 Memory Management

As memory management is a core part of COS110 and C++, each task on FitchFork will allocate approximately 10% of the marks to memory management. The following command is used:

```
valgrind --leak-check=full ./main
```

Please ensure, at all times, that your code *correctly* de-allocate *all* the memory that was allocated.

7 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the main.cpp file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j term polynomial univariate bivariate linear quadratic circle
    ellipse
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ration based on class size.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the Instructor provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

8 Upload checklist

The following files should be in the root of your archive

- main.cpp
- term.cpp
- polynomial.cpp
- univariate.cpp
- bivariate.cpp
- linear.cpp
- quadratic.cpp
- circle.cpp
- ellipse.cpp
- All text files your main uses for testing purposes.

9 Allowed libraries

- cmath
- iomanip
- iostream
- sstream
- string

10 Submission

You need to submit your source files, only the .cpp files, on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. There is no need to include any other files or .h files in your submission. Your code must be able to be compiled with the C++98 standard

For this practical, you will have 5 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**