Department of Computer Science

Faculty of Engineering, Built Environment & IT

University of Pretoria

COS110 - Program Design: Introduction

Practical 5 Specifications

Release Date: 02-09-2024 at 06:00

Due Date: 06-09-2024 at 23:59

Total Marks: 150

Contents

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.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of c++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use c++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.
- Note, UML notation for variable and function signatures are used in this assignment.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.

2 Overview

Operator overloading is a C++ feature that allows programmers to define functions for some of the built-in C++ operators. This can be implemented to allow more elegant use of the functions for a specific class if implemented correctly, but can also cause confusion since there are very few limitations on how the operators can be implemented.

3 Your Task:

You are required to implement the following class diagram illustrated in Figure 1. Pay close attention to the function signatures as the `h` files will be overwritten, thus failure to comply with the UML, will result in a mark of 0.

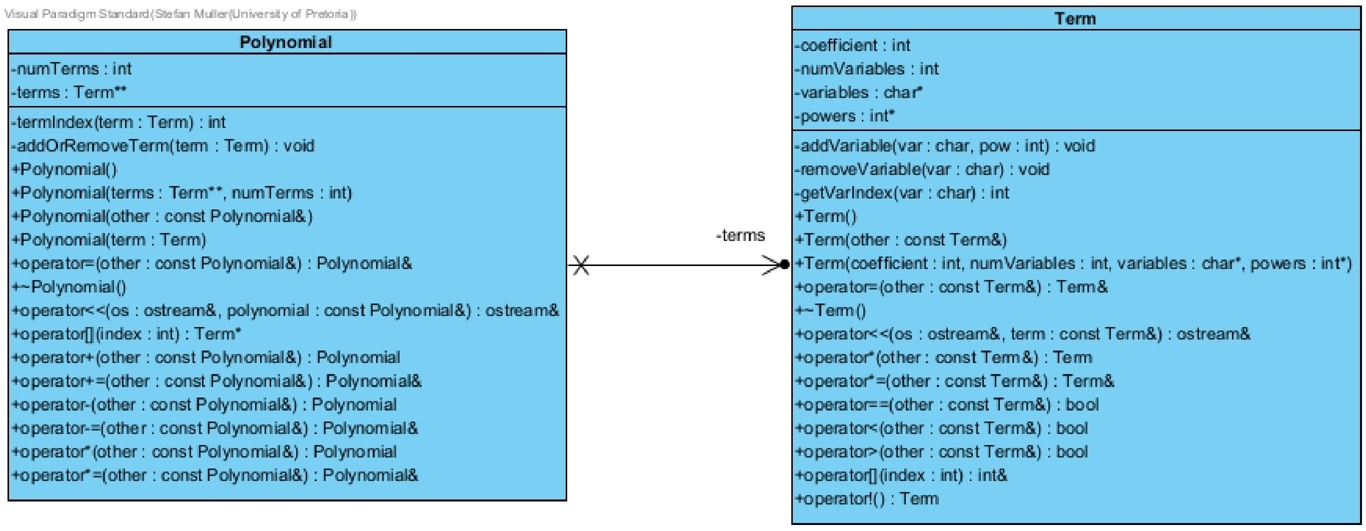
Visual Paradigm Standard(Stefan Muller(University of Pretoria))
Polynomial
-numTerms : int
-terms : Term**
-termIndex(term : Term) : int
-addOrRemoveTerm(term : Term) : void
+Polynomial()
+Polynomial(terms : Term**, numTerms : int)
+Polynomial(other : const Polynomial&)
+Polynomial(term : Term)
+operator=(other : const Polynomial&) : Polynomial&
+~Polynomial()
+operator<<(os : ostream&, polynomial : const Polynomial&) : ostream&
+operator[](index : int) : Term*
+operator+(other : const Polynomial&) : Polynomial
+operator+=(other : const Polynomial&) : Polynomial&
+operator-(other : const Polynomial&) : Polynomial
+operator-=(other : const Polynomial&) : Polynomial&
+operator*(other : const Polynomial&) : Polynomial
+operator*=(other : const Polynomial&) : Polynomial&
-terms
Term
-coefficient : int
-numVariables : int
-variables : char*
-powers : int*
-addVariable(var : char, pow : int) : void
-removeVariable(var : char) : void
-getVarIndex(var : char) : int
+Term()
+Term(other : const Term&)
+Term(coefficient : int, numVariables : int, variables : char*, powers : int*)
+operator=(other : const Term&) : Term&
+~Term()
+operator<<(os : ostream&, term : const Term&) : ostream&
+operator*(other : const Term&) : Term
+operator*=(other : const Term&) : Term&
+operator==(other : const Term&) : bool
+operator<(other : const Term&) : bool
+operator>(other : const Term&) : bool
+operator[](index : int) : int&
+operator!() : Term

Figure 1: Class diagrams

Note, the importance of the arrows between the classes are not important for COS 110 and will be expanded on in COS 214.

3.1 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 corresponding power. Note that a variable to the power 0, won't be stored because this is the same as multiplying by 1.

3.1.1 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.

3.1.2 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 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.**
- -getVarIndex(var: char) const: int
  - This is a constant function.
  - 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.
- +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(coefficient: int, numVariables: int, variables: char*, powers: int*)
  - This is a parameterized constructor.
  - Deep copies should be made where applicable.
  - Note that the passed-in variables might not be sorted, but the member variable must be sorted.
- +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.

- +operator«(os: ostream&, term: const Term&): ostream&
  - This will be used to print out the term.
  - The formatting follows the following rules:
    - There should be no spaces or newlines in the result.
    - The coefficient with its sign should be printed first. Note that if the coefficient is positive, the + sign should also be printed.
    - If there are no variables in the term, then only print the coefficient.
    - If there are variables, then print the variables and powers by indicating the power using the caret symbol (ˆ).
    - There should be a star between each variable, and also one between the coefficient and the first variable.
  - Example: If the term is $10x^2y^3z^4$ then this function will print

```
+10*x^2*y^3*z^4
```

- +operator*(other: const Term&) const: Term
  - This is a constant function.
  - 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.
- +operator==(other: const Term&) const: bool
  - This is a constant function.
  - 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.

- +operator<(other: const Term&) const: bool
  - This is a constant function.
  - 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;
- +operator>(other: const Term&) const: bool
  - This is a constant function.
  - 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***.
- +operator!() const: Term
  - This is a constant function.
  - This is the negation operator.
  - The current object should not be altered.
  - The negation of a term should swap the sign of the coefficient.

3.2 Polynomial

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

3.2.1 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.

3.2.2 Functions

- -addOrRemoveTerm(term: Term): void
  - If the coefficient of the passed-in term is zero then don't add it.
  - 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.**
- -termIndex(term: Term) const: int
  - This is a constant function.
  - 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(terms: Term**, numTerms: int)
  - This is a parameterized constructor.
  - 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(other: const Polynomial&)
  - This is the copy constructor.
  - Deep copies should be made of all member variables.
- +Polynomial(term: 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.
- +operator«(os: ostream&, polynomial: const Polynomial&): ostream&
  - This will print the string representation of the terms. There should be no spaces or newlines in the result.
  - If numTerms is 0, then print +0.
  - Use the stream insertion operator for each term to get the string representation of that term. Add these together directly, without adding any formatting.
- +operator`[]`(index: int) const: Term*
  - This is a constant function.
  - If the index is valid, return the term at that index.
  - If the index is invalid, return NULL.

- +operator+(other: const Polynomial&) const: Polynomial
  - This is a constant function.
  - This should return a polynomial which is the result of adding the two polynomials together, following the normal rules from Mathematics.
  - If you want to see an example of this, click here.
- +operator+=(other: const Polynomial&): Polynomial&
  - This should change the current Polynomial to the result of adding the two polynomials together, following the normal rules from Mathematics.
  - If you want to see an example of this, click here.
- +operator-(other: const Polynomial&) const: Polynomial
  - This is a constant function.
  - This should return a Polynomial which is the result of subtracting the passed-in Polynomial from this Polynomial, following the normal rules from Mathematics.
  - If you want to see an example of this, click here.
- +operator-=(other: const Polynomial&): Polynomial&
  - This should change the current Polynomial to the result of subtracting the passed-in Polynomial from this Polynomial, following the normal rules from Mathematics.
  - If you want to see an example of this, click here.
- +operator*(other: const Polynomial&) const: Polynomial
  - This is a constant function.
  - This should return a polynomial which is the result of multiplying the two polynomials together, following the normal rules from Mathematics.
  - If you want to see an example of this, click here.
- +operator*=(other: const Polynomial&): Polynomial&
  - This should change the current Polynomial to the result of multiplying the two polynomials together, following the normal rules from Mathematics.
  - If you want to see an example of this, click here.

4 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-allocates *all* the memory that was allocated.

5 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 ${files}
```

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

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

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 functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can

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

only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

6 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **c++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- You may only use the following libraries:
  - iostream
  - sstream
  - string

7 Upload Checklist

The following c++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- Term.cpp
- Polynomial.cpp
- `main.cpp`
- Any textfiles used by your `main.cpp`
- `testingFramework.h` and `testingFramework.cpp` if you used these files.

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

8 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 10 submissions and your best mark will be your final mark. Upload your archive to the Practical 5 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**

9 Accessibility

Figure 1 shows the UML of the following classes:

- Term
- Polynomial

The member functions and variables for each class is discussed in Section 3. The following relationships are shown in Figure 1:

- Polynomial has a relation with Term due to the *terms* variable.