MECHTRON 2MD3
Data Structures and Algorithms for Mechatronics
Winter 2022

Assignment 02

Published: February 22, 2022

Due at 11:59pm on March 6th, 2022.

Please read the following carefully:

- Late submissions will be marked with a penalty of 20% per day.
- The answers to Questions 1 and 2 must be submitted as two separate source code files (cpp or zip file containing the source code). Do not submit the source code as PDF file, otherwise you will receive a mark of **zero**.
- Include your name and student ID number in the source code files. You can add a comment on top of the source code indicating your name and student ID.
- **Your work must be your own. Plagiarism and copying will not be tolerated. This is very important!**
- All files are to be submitted via Avenue to Learn.
- Please ensure you have read the questions carefully and watched the companion video.

This assignment consists of 2 questions, and is worth 50 marks.

Q 1 Inheritance and Polymorphism [20 marks]

The task is to design a set of C++ classes and their relations that perform simple statistical computations on given **non-negative integer** arrays.

You have to design three classes:

1) First, you have to design a *Stats* class that provides basic capabilities to other more specific statistical classes. The *Stats* class has a constructor that receives as parameters a pointer to an integer array and the length of that array. *Stats* class creates a deep copy of the array and stores it in a member array called *array1*. Other than the array, the length and the maximum value of the input array are also set in the constructor and stored as member variables.

Stats class has two other member variables *mean* and *stdev* to store the results of statistical computations, namely, array mean and standard deviation respectively. They should be available to outside users only with getter functions.

2) The second class is the *OneVarStats* class that is meant to compute more advanced statistical descriptors such as **histogram** for a single array. It **inherits** members from the *Stats* class and can use them for its own purposes. Its constructor, similar to *Stats*, receives an array and its length. However, despite *Stats*, it does not store a copy of the array; rather, it sends the input array to *Stats*. The only functionality of *OneVarStats* is to compute the histogram of values in the given array. Histogram represents the frequency distribution of values, and in our simple example, it will show the frequency of values in the array, along with the frequency of values that are not in the array. Definitely, the frequency of values that are not in the array is zero. Thus, for histogram, we need an array with indices from 0 to maximum value of the array (array size must be $1 + max$). Each element of the histogram will show the frequency of the value represented by its index.

For example, for the array $[2, 4, 2, 6, 4]$, the frequency of 2 is two, 4 is two and 6 is one. To represent this histogram, we need an array of length $1+6 = 7$. The histogram will be: $[0, 0, 2, 0, 2, 0, 1]$.

OneVarStats has two member variables. The first one is *histLength* which is the length of histogram, and the second is *hist* array that holds the result of histogram computation. Both of these members must be initialized appropriately in the constructor.

3) The last class is *TwoVarStats*. It **inherits** members from the *Stats* class. It will be used to compute statistical descriptors for two arrays. An example of such descriptor is **Correlation Coefficient**, which is a measure that determines how strong is the relationship of corresponding elements of two arrays.

The constructor of this class accepts two arrays of the same length, and their size. The first array is sent to *Stats* and a copy of the second array is stored in *TwoVarStats* as *array2*. The class must have a member variable that stores the result of statistical computation which is a single value. Let's call it *correlationCoef*. The class *TwoVarStats* must have a *computeCorrCoef* **utility function** that its implementation is given in the companion source code files. You don't need to worry about that. It works on *array1* (which comes from *Stats*) and *array2* (which is stored in *TwoVarStats*). The function must also have access to *count* that is stored in *Stats*. You have to use this function in an appropriate manner.

Polymorphic functions: Each of the three classes above must implement two polymorphic functions that can provide run-time polymorphism based on the dynamically-bound values of objects.

1) function *computeStats*: This function will compute and store:

- for *Stats*: the mean and standard deviation (see Appendix A) of the input array.
- for *OneVarStats*: the mean and standard deviation of the input array along with the histogram array of the input array.
- for *TwoVarStats*: the mean and standard deviation of the first input array and the Correlation Coefficient of the two input arrays. You have to use

the *computeCorrCoef* utility function for this.

2) function *printStats*: This function will output the stored statistics:

- for *Stats*: the mean and standard deviation of the input array.
- for *OneVarStats*: the mean and standard deviation of the input array along with the histogram array of the input array.
- for *TwoVarStats*: the mean and standard deviation of the first input array and the Correlation Coefficient of the two input arrays.

Important: All three classes must have appropriate destructors to avoid memory leak. You have to think about the order of destructors in inheritance and decide what to implement. A quick hint here is that you must define the destructor of *Stats* as virtual (`virtual ~Stats();`) to allow sub-classes define their own destructors.

The main function for test purposes is provided. If everything is implemented correctly, the *computeStats* and *printStats* functions must function according to their actual run-time values.

```
int main( ) {
    int x[] = {2, 4, 7, 11, 5};
    int y[] = {5, 9, 14, 20, 10};
    int z[] = {14, 7, 4, 9, 21};

    int stats_len = 4;
    Stats* pp[stats_len];
    pp[0] = new Stats(X, 5);
    pp[1] = new OneVarStats (x,  5);
    pp[2] = new TwoVarStats (x, y, 5);
    pp[3] = new TwoVarStats (y, z, 5);

    for (int i=0; i < stats_len; i++){
        pp[i]->computeStats();
        cout << "\n";
    }

    for (int i=0; i < stats_len; i++){
        pp[i]->printStats();
        cout << "\n";
    }
}
```

The desired output of the above code is:

```
mean= 5.8, std= 3.05941
```

```
mean= 5.8, std= 3.05941
0 0 1 0 1 1 0 1 0 0 0 1

mean= 5.8, std= 3.05941
corr coef= 0.997947

mean= 11.6, std= 5.08331
corr coef= -0.389055
```

You are **NOT** required to check if arrays have the same size or any other extreme cases. Just think of the simplest solution. What's important is to design classes, their relationships, constructors (storing a copy of arrays), destructors and polymorphism. If you have issues in understanding the problem, please ask in the Question and Answer channel in Teams. You should not ask questions regarding your answers. All the details are included in this text and the companion video.

Q 2 Linked Data Structures [30 marks]

The aim of this question is to implement a data structure to work with **Polynomial** expressions. A polynomial expression contains **Terms**. Each term has a *coefficient* and an *exponent*. For example, the polynomial expression $5x^2 - 4x + 3$ has three terms with the coefficient and exponent 5, 2 for term one, -4, 1 for term two and 3, 0 for term three, respectively.

The data structure will store the coefficient and exponents of terms. The underlying structure where these data of the polynomial expression will be stored is a **Doubly Linked List**. Any solution violating this requirement will get a mark of **zero** for this question. An implementation of the *Doubly Linked List* class and Term class is given in the companion code. As you can observe, the *DLinkedList* is implemented to store elements of type *Term*. You have to implement the Polynomial class with the following details:

- declare a private *DLinkedList* that stores terms, in which terms are always sorted by their exponent in decreasing order.
- implement function *insertTerm(c, e)* that inserts the term with coefficient c and exponent e into the proper location in the *DLinkedList*. The idea is to insert the terms in the **sorted order** such that terms are sorted by their exponents in decreasing order. This means, for every input term, the program must identify the proper location in the *DLinkedList* and insert it there.
  A few notes here:
  a) coefficient 0 must be handled appropriately.
  b) if the polynomial already had a term with the same exponent as e, the coefficient of that term must be updated to c.
- overload >> operator so that a user can use such an statement to input the values of coefficient and exponent for each term. Note that the operator should first ask for the total number of terms and then start reading the terms values:

  ```
  Polynomial p1;
  cin >> p1;
  ```

  The operator will use *insertTerm* function to insert the input term into the *DLinkedList*. The operator should also support chains of input stream statements.
- implement utility function *toString()* that puts a decent representation of the Polynomial in a string and returns it. This function must not be available to outside users. You have to think how to handle terms with exponent 1 and terms with exponent 0. See Appendix B for details about making a **string** by gradual appending to it.

- overload << operator so that a user can use such an statement to output a representation of the polynomial:

  ```
  Polynomial p1;
  // some code here
  cout << p1;
  ```

  The operator should use the *toString()* function. The operator should also support chains of output stream statements.

- overload + operator so that a user can use such an statement to add two polynomials:

  ```
  Polynomial p1, p2;
  // some code here
  p1 + p2;
  ```

  The operator performs the normal polynomial addition. The idea is simple. You have to create an empty result polynomial. You must have one pointer on the first polynomial's terms and another pointer on the second polynomial's terms. If the two terms pointed by the pointers had identical exponents, their coefficient must be summed up, the new term should be added to the result, and the two pointers must hop to the next terms. If either of exponents were larger, the larger one should be added to the result and its pointer must hop to the next. If any of the pointers reach to the end of its corresponding polynomial, the remaining terms of the other polynomial must be inserted to the result. Please watch the companion video for more details.

- implement function *eval(x)* that receives a value x as the parameter and evaluates the polynomial with that x. This means, x's in the expression must be replaced with the value given as parameter and the final value must be computed by evaluating the terms and summing them. For example, with $x = 2$ the expression $5x^2 - 4x + 3$ evaluates to 15.

Please note that:

- for simplicity, we assume that all coefficients and exponents are integers.
- Many of the functions above need to traverse through the polynomial's nodes. You have to exploit your knowledge of traversing linked lists to perform these operations.
- Regarding the previous note, you have to define appropriate friendship relations between classes to allow the Polynomial access private members of DLinkedList, DNode, and Term.

- you are not allowed to move the functions in the companion code between access specifiers. That is, if `add(DNode* v, const Elem&)` is defined in *protected*, it must be left there. Of course, you can add your functions wherever you want and according to the requirements of the question.
- It is very convenient to use *toString* utility function defined above. If you want to use another solution (implementing everything in the overloading operator) you will face a lot of problems. Of course, it is possible to implement like that, but my suggestion is to exploit *toString* in overloading <<. Your solution will be more readable.
- You have to use *add* function of *DLinkedList* to add a node in a given place. If you remember from the lectures, this was a utility function for *DLinkedList*, but of course we can use it if we traverse the *DLinkedList* and have pointers to specific nodes.

Appendix A

Given an input list of values $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$ is their variance and $\sqrt{\sigma^2}$ is the standard deviation, where N is the number of items in the list, x_i is each item in the list and $\bar{x}$ is the mean of values in the list.

The following code snippet computes standard deviation:

```
double var = 0.0;
for (int i=0; i<N; i++){
    var += pow(x[i] - mean, 2);
}
var =  var  / count;  // variance
stdev = sqrt(var);    // standard deviation
```

Appendix B

Sometimes we need to build a long string gradually, by appending values to it. In that case, we need to declare a string variable (available in *std::string*) and initialize it to the empty string. Then, for instance, we can append values to it in every iteration of a loop.

When appending values, we must ensure that the value that we are about to append is string itself. Otherwise, we have to use *to_string* function (available in *std*) to convert the value to string format. The following example creates a string, appends 4 to it, appends a comma and a space and at the end appends a 5 to it. The result string will be "4, 5".

```
string a = "";
a += to_string(4);
a += ", ";
a += to_string(5);
```