Assignment 3 (7%)

Due Date: 27 April 2024, 23:59 PM

1. This assignment contains **2** questions(Q2 has 3 parts). You are required to finish a complete C++ program (.cpp only) for each question on E-quiz before the deadline.

2. You can check as many times as you want before the deadline. We will grade your **highest version**.

3. Only a small set of the test cases are visible for the testing, and a more comprehensive set of test cases will be used for grading. In other word, passing all the visible test cases does not mean that you can get the full mark for this question. Try your best to thoroughly check your program.

4. Hidden test cases will not be released.

5. The marking of each question is based on the percentage of the total test cases that your solution can pass. If your submitted solution leads to a compilation error on E-quiz, zero mark will be given to that question and no manual checking to your solution is provided in such a case.

6. No late submission will be accepted.

7. Plagiarism check will be performed.

8. You only need to use the material from Lectures 1 to 11. It is **NOT** necessary to include any other library, except <iostream> <fstream><string><cstring>.

9. You need to check your solution and submit it on E-quiz.

To ensure timely feedback to the students' questions, each of the following TAs will be responsible for one question. If you have any questions, you can contact the corresponding TA directly.

Question	TA responsible	Email Address
Q1	SHATOKHIN Anton	ashatokhi2-c@my.cityu.edu.hk
Q2	YAN Yachao(odd)	yachaoyan2-c@my.cityu.edu.hk
	SANIYAZOV Dias(even)	dsaniyazo2-c@my.cityu.edu.hk

For Q2 question, please contact YAN Yachao if your student number has an odd tail, or SANIYAZOV Dias if it has an even tail.

1. Program Development [40%]

Write a program to implement a **calculator** to compute equations of **complex numbers**. An equation consists of two **operands** and an **operation**. For example, for equation (2+3i) + (1 +2i), complex number 2+3i and 1+2i are two operands and the operation is addition ("+"). A complex number is represented as $\boldsymbol{a} + \boldsymbol{b}i$, where **a** is the real part, **b** is the imaginary part and **i** is the imaginary unit. Assume the first operand of an equation is $\boldsymbol{a} + \boldsymbol{b}i$, the second operand is $\boldsymbol{c} + \boldsymbol{d}i$. In this question, we consider three operations: addition, subtraction, and multiplication with the following definitions:

- **Addition**: return the result as $(\boldsymbol{a} + \boldsymbol{c}) + (\boldsymbol{b} + \boldsymbol{d})i$
- **Subtraction**: return the result as $(\boldsymbol{a} - \boldsymbol{c}) + (\boldsymbol{b} - \boldsymbol{d})i$
- **Multiplication**: return the result as $(\boldsymbol{ac} - \boldsymbol{bd}) + (\boldsymbol{ad} + \boldsymbol{bc})i$

To implement the calculator for complex number, first define a struct **MyComplex** consisting of two ints **Real** and **Imaginary**, which stores the real part and the imaginary part of a complex number. Then implement an **abstract** class **Calculator** with two **protected** members:

- `num1`: a **MyComplex** variable, storing the value of the first operand for a equation.
- `num2`: a **MyComplex** variable, storing the value of the second operand for a equation.

The class **Calculator** further have two public member functions:

- `set(MyComplex n1, MyComplex n2)`:set the values for the two operands `num1` and `num2`.
- `getResult()`: This function is used to return the result of the current equation. In class **Calculator**, `getResult()` is a pure virtual function and will be reloaded by other derived classes.

Then implement three operarions above with the following classes:

- class **AdditionCal**. It (public) inherits class **Calculator** to perform addition of complex numbers by overloading virtual function `getResult()` in **Calculator**.
- class **SubtractionCal**. It (public) inherits class **Calculator** to perform subtraction of complex numbers by overloading virtual function `getResult()` in **Calculator**.
- class **MultiplyCal**. It (public) inherits class **Calculator** to perform multiplication of complex numbers by overloading virtual function `getResult()` in **Calculator**.

In this question, we will provide inputs for two complex numbers as operands to calculate the result of three operations (addition, subtraction, and multiplication). After reading inputs, you should define a **pointer Cal** with the data type of class **Calculator**. Then use **Cal** to create objects for each derived classes (AdditionCal, SubtractionCal, and MultiplyCal) to perform the corresponding operations, i.e., **polymorphism**.

Note 1: you can add proper constructor for the struct **MyComplex**.
Note 2: Check the examples below carefully for the output of complex numbers in different situations.

Example-1 (Input is underlined):

```
Input the first operand:
4 2
Input the second operand:
1 3
The result of (4+2i)+(1+3i) is 5+5i
The result of (4+2i)-(1+3i) is 3-1i
The result of (4+2i)*(1+3i) is -2+14i
```

Example-2 (Input is underlined):

```
Input the first operand:
2 5
Input the second operand:
2 5
The result of (2+5i)+(2+5i) is 4+10i
The result of (2+5i)-(2+5i) is 0
The result of (2+5i)*(2+5i) is -21+20i
```

Example-3 (Inputs are underlined):

```
Input the first operand:
3 -4
Input the second operand:
-2 -4
The result of (3-4i)+(-2-4i) is 1-8i
The result of (3-4i)-(-2-4i) is 5
The result of (3-4i)*(-2-4i) is -22-4i
```

Example-4 (Inputs are underlined):

```
Input the first operand:
0 2
Input the second operand:
0 -3
The result of (2i)+(-3i) is -1i
The result of (2i)-(-3i) is 5i
The result of (2i)*(-3i) is 6
```

Example-5 (Inputs are underlined):

```
Input the first operand:
0 0
Input the second operand:
2 3
The result of (0)+(2+3i) is 2+3i
The result of (0)-(2+3i) is -2-3i
The result of (0)*(2+3i) is 0
```

2. Program Comprehensive [60%]

Write a program to implement a **template<class T>** class **MyArray** that simulates functions of the standard **array** and can store variables with **any** datatype. The class **MyArray** uses **dynamic allocation** to access memory for data storage. In particular, it contains the following three private members:

- `pAddress`: a **T** pointer, which points to the **dynamic array** storing the data.
- `mSize`: an **int** variable, storing the amount of variables currently storing in the array.
- `mCapacity`: an **int** variable, recording that how many variables the current object can store.

The basic workflow of using the class **MyArray** is as following. First, the object of the class **MyArray** is initialized with an **int** number assigning to `mCapacity`. Then a dynamic array with size of `mCapacity` and datatype **T** is created to store data for the current object and returned to `pAddress` for access. The member `mSize` records how many variables are currently stored and hence is used as a virtual boundary. Initially, there are **NO** variables storing in the object. Therefore, the value of `mSize` is 0. To add a new variable to the object, assign the value to the last position of the current object (i.e., `pAddress[mSize]`) and then increase the value of `mSize` by 1. Implement the above functionalities with the following **public** members:

- `MyArray(int capacity)`:the parameterized constructor to initialize `pAddress`, `mSize`, `mCapacity` in the way demonstrated in the workflow above.
- `MyArray(const MyArray & a)`:the copy constructor that uses another object `a` to initialize the current object. Use deep copy to implement the copy constructor.
- `~MyArray()`:the destructor to release the memory pointed by `pAddress`.
- `pushBack(const T& val)`:add a new **T** variable **val** at the position of `mSize`. If the current object cannot store more variables (`mSize == mCapacity`), cancel the insertion and print proper alert messages as shown in examples below.
- `getMax()`:return the maximum value among all the variables storing in the object.
- `operator=`: overload the `operator=` that copies the value of another object to the current object. Use deep copy.
- `operator[]`:overload the `operator[]` to directly access the variables through **index**. For example, for an object `MyArray arr`, the first variable storing in `arr` should be able to access through direct access as `arr[0]`.

- For the deep copy in `operator=`, you should first **release** the memory occupied by the dynamic array of the current object.

After implementing the class **MyArray**, test the program with different datatypes as following.

Part-A). **[35%]** Create two objects of **MyArray** by assigning the **template T** as **int** and **char**, separately. The capacities of two objects are both set as 5. Then print the max variables storing in the object.

Note1: in Example-2, **7** integers are inputted to store in the object while the capacity is set as **5**. Hence, two last two integers won't be stored, and the alter messages are outputted twice.
Note2: in this question, you can assume the amount of variables in input is always **larger** than 0.

Example-1 (Input is underlined):

```
Input the amount of variables to store in the int array:
3
Input the integers:
9 12 3
The max value of the array is: 12
Input the amount of variables to store in the char array:
3
Input the chars:
qwe
The max value of the array is: w
```

Example-2 (Input is underlined):

```
Input the amount of variables to store in the int array:
7
Input the integers:
3 4 2 3 9 1 8
Inserting new value failed: the array is full.
Inserting new value failed: the array is full.
The max value of the array is: 9
Input the amount of variables to store in the char array:
5
Input the chars:
hello
The max value of the array is: o
```

Part-B). **[15%]** Further implement a class **MyClassInt** as the **datatype** of variables to store in an **MyArray** object. The class **MyClassInt** stores an int number and only contains one **private** value:

- `val`: an int variable storing a number.

MyClassInt further contains the following **public** member functions:

- `MyClassInt()`:the default constructor to initialize `val=0`.
- `MyClassInt(int v)`:the parameterized constructor to initialize `val=v`.
- `operator=`:overload the `operator=` that copies the value of another object to the current object. Use deep copy.
- `operator<<`: overload the `operator<<` to print the value of `val`. The overloaded `operator<<` should support **chain** function calls.
- `operator>`: overload the `operator>` to compare the object with another object through the value of their member variables `val`. Return **true** if the current object is **larger**; otherwise return **false**.

Create an object of **MyArray** by assigning the **template T** as **MyClassInt** with capacity of 5. Then print the max variables storing in the object as shown in the examples.

Note: You need to overload the **operators** of **MyClassInt** correctly in order to call the function `getMax()` defined in **MyArray**.

Example-1 (Input is underlined):

```
Input the amount of variables to store in the MyClassInt array:
3
Input the integers:
9 2 4
The max value of the array is: 9
```

Example-2 (Input is underlined):

```
Input the amount of variables storing in the MyClassInt array:
8
Input the integers:
9 10 6 4 9 2 2 4
Inserting new value failed: the array is full.
Inserting new value failed: the array is full.
Inserting new value failed: the array is full.
The max value of the array is: 10
```

Part-C). **[10%]** Further implement a class **MyClassArr** as the **datatype** of variables to store in an **MyArray** object. The class **MyClassArr** stores integers through **MyArray<int>**. It contains two **private** values:

- `MyArray<int>* mArray`: a pointer of **MyArray<int>**. It points to the object of **MyArray<int>** that stores the integers.
- `Len`: stores the length of `mArray`.

- For **MyClassArr**, we assume that its member `mArray` is fully occupied (i.e., `mArray.mSize == mArray.mCapacity`).

MyClassArr further contains the following **public** member functions:

- `MyClassArr()`:the default constructor to initialize `mArray=NULL, Len=0`.
- `MyClassArr(int arr[], int l)`:the parameterized constructor to initialize `mArray` with the values of `arr`, and `Len=l`.
- `MyClassArr(const MyClassArr& a)`:the copy constructor that uses another object `a` to initialize the current object. Because `mArray` contains dynamic array (defined in class **MyArray**), use deep copy by creating a new object of **MyArray** and calling `pushBack()` function.
- `~MyClassArr()`:the destructor to release the memory occupied by `mArray`.
- `operator=`: overload the `operator=` that copies the value of another object to the current object. Use deep copy with the same principle as the copy constructor.
- `operator<<`: overload the `operator<<` to print the integers stored in `mArray`. There is a space between the adjacent integers as shown in the example. The overloaded `operator<<` should support **chain** function calls.
- `operator>`: overload the `operator>` to compare two objects through the value of their member variables `mArray`. Return **true** if the current object is **larger**; otherwise return **false**.

- We define the rule to compare two **MyArray<int>** objects a1 and a2 as follows:
    - Compare arr1 and arr2 integer by integer from the beginning of both dynamic arrays (i.e., `a1.pAddress` and `a2.pAddrerss`) used to store the integers.
    - If the i-1 integers of the two objects are identical, then:
        a. If ith integer of a1 is larger than a2, a1 is larger than a2 regardless the remaining part;
        b. If ith integer of a1 is smaller than a2, s1 is smaller than s2 regardless the remaining part;
        c. If ith integer of s1 is equal to s2, continue the operation on i+1th bit until the dynamic array `pAddress` of one object ends;
    - When the dynamic array of one object ends, and there is still no result for the comparison, the object with longer length (`mSize`) is larger. If the lengths of the two objects are identical, the two object are equal.

Create an object of **MyArray** by assigning the **template T** as **MyClassArr** with capacity of 5. We will first provide the amount of **MyClassArr** objects in the input. Then the size and the content of **int** arrays to store in each **MyClassArr** object will be provided. After reading the inputs properly, print the max variables storing in the object as shown in the examples.

Note: You need to overload the **operators** of **MyClassArr** correctly in order to call the function `getMax()` defined in **MyArray**.

Example-1 (Input is underlined):

```
Input the amount of variables to store in the MyClassArr array:
3
Input the size of MyArray 1:
3
Input array 1:
3 2 3
Input the size of MyArray 2:
2
Input array 2:
1 10
Input the size of MyArray 3:
2
Input array 3:
3 2
The max value of the array is: 3 2 3
```

Example-2 (Input is underlined):

```
Input the amount of variables to store in the MyClassArr array:
7
Input the size of MyArray 1:
6
Input array 1:
3 4 5 1 2 3
Input the size of MyArray 2:
3
Input array 2:
3 4 6
Input the size of MyArray 3:
1
Input array 3:
2
Input the size of MyArray 4:
4
Input array 4:
1 3 4 2
Input the size of MyArray 5:
2
Input array 5:
3 4
Input the size of MyArray 6:
6
Input array 6:
3 5 5 1 2 3
Inserting new value failed: the array is full.
Input the size of MyArray 7:
3
Input array 7:
3 3 1
Inserting new value failed: the array is full.
The max value of the array is: 3 4 6
```

– End –