Calculator List

COP 4530 Programming Project 1

Instructions

For Programming Project 1, the student will implement a linked list based arithmetic calculator. The calculator will be able to perform addition, subtraction, multiplication, and division. The calculator will keep a running total of the operations completed, the number of operations completed, and what those operations were. The calculator will also have an "undo" function for removing the last operation. The calculator will also be able to output a string of the operations completed so far with fixed precision.

The calculator (which must be called “CalcList”) has to be implemented using a singly, doubly, or circularly linked list. Any projects that use the C++ Standard Library Lists or other sources to implement the linked list will receive a zero. The calculator has to implement at least four methods:

Abstract Class and Files

`double total() const`

This method returns the current total of the CalcList. Total should run as a constant time operation. The program should not have to iterate through the entire list each time the total is needed.

`void newOperation(const FUNCTIONS func, const double operand)`

Adds an operation to the CalcList and creates a new total. The operation alters total by using the function with the operand. Example: `newOperation(ADDITION, 10)` => adds 10 to the total.

`void removeLastOperation()`

Removes the last operation from the calc list and restores the previous total.

`std::string toString(unsigned short precision) const`

Returns a string of the list of operations completed so far formatted with a fixed point precision. The form of the string should strictly be: "(step): (totalAtStep)(Function)(operand) = (newTotal) \n".
Example: `toString(2) => "3: 30.00*1.00=30.00\n2: 10.00+20.00=30.00\n1: 0.00+10.00=10.00\n"`

This project includes an abstract class for the CalcList from which to inherit. This abstract class (CalcListInterface) contains the pure virtual version of all the required methods. This file also includes a typedef of an enum used for the four arithmetic functions called FUNCTIONS.

Error Message handling and Throw exceptions Handling
Your code will be tested for all functionality and in order to successfully pass the test cases, please ensure the following is included in your CaclList.cpp File

- Throw a std::invalid_argument with message " Cannot Divide By Zero"
- Throw std::runtime_error with message "No Operations to Remove"

Please submit complete projects as zipped folders. The zipped folder should contain: CalcListInterface.hpp, CalcListTests.cpp, CalcList.hpp (Your Code),CalcList.cpp (Your Code)

Examples

Below are some examples of how your code should run. The test file can also be used to get an idea of how the code should run.

```
CalcList calc;                          // Total == 0
calc.newOperation(ADDITION, 10);        // Total == 10

calc.newOperation(MULTIPLICATION, 5);   // Total == 50
calc.newOperation(SUBTRACTION, 15);     // Total == 35
calc.newOperation(DIVISION, 7);         // Total == 5
calc.removeLastOperation();             // Total == 35
calc.newOperation(SUBTRACTION, 30);     // Total == 5
calc.newOperation(ADDITION, 5);         // Total == 10
calc.removeLastOperation();             // Total == 5


// Should Return:
// 4: 35.00-30.00=5.00
// 3: 50.00-15.00=35.00
// 2: 10.00*5.00=50.00
// 1: 0.00+10.00=10.00
std::cout << calc.toString(2);

calc.removeLastOperation();             // Total == 35

// Should Return:
// 3: 50-15=35
// 2: 10*5=50
// 1: 0+10=10
std::cout << calc.toString(0);
```

Example #1:

```
newOperation Subtraction 10

removeLastOperation                                          //total =0.01
```

Example #2:

```
newOperation (Addition, 50)

newOperation (Multiplication, 0)

removeLastOperation()                                      //total = 50.01

newOperation (Division, 0)
```

Example #3:

```
newOperation (Addition, 10)

newOperation (Addition, 20)

newOperation (Subtraction, 20)

removeLastOperation()

newOperation (Multiplication, 1)

toString(1)       // output ="3: 30.0*1.0=30.0\n2: 10.0+20.0=30.0\n1: 0.0+10.0=10.0\n";
```

Example #5:

```
newOperation (Addition, 10)

newOperation (Addition, 20)

newOperation (Division, 5)      //total = 10
```

Submission instructions

1. Name your program
2. Develop and test your program on the student cluster
3. Download the program from student cluster and submit it on **Canvas->Gradescope**. Make sure you submit a file with the correct name!
4. You can submit your program as many times as needed before the due date. Gradescope will indicate the test cases with incorrect output, if any exists.
5. Projects will be manually graded after running the test cases on GradeScope.

Please note that GradeScope is used for testing some of the functionalities. It’s part of the grading process. The grade you received on GradeScope only reflects the results of your program against the test cases, it is not your final project grade.

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

- Your program should begin with a comment that briefly summarizes what it does. This comment should also include your name.
- In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only needed in order for a reader to understand what is happening.
- Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
- Use consistent indentation to emphasize block structure.
- Full line comments inside function bodies should conform to the indentation of the code where they appear.
- Macro definitions (**#define**) should be used for defining symbolic names for numeric constants.
- Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
- Use underscores to make compound names easier to read: **tot_vol** and **total_volumn** are clearer than **totalvolumn**.

Hints

When implementing the `toString` method, the headers `sstream` and `iomanip` will have functions that make controlling precision and creating returnable string easier. Also remember, zero does not have a multiplicative inverse.

Rubric

Any code that does not compile will receive a zero for this project.

Criteria	Points
Total should be initially zero	2.5
Operations should be removable	12.5

Zero multiplication operation should be removable	5
Operations cannot divide by zero and should throw	5
Removal of operations from an empty CalcList should throw	5
toString functions should return string list of operations at precision	12.5
Operations should change the total	25
Calculator uses a student implemented Linked List	17.5
Code uses object oriented design principles (Separate headers and sources, where applicable)	7.5
Code is well documented	7.5
Total Points	**100**