**08. Destructors, Constructors and Copy-Assignment**

Write C++ code for solving the tasks on the following pages.

Submit your solutions here: <https://judge.softuni.bg/Contests/1280/08-CPlusPlus-OOP-Inheritance-and-Polymorphism> (select “Compete” when prompted)

Any code files that are part of the task are provided under the folder **Skeleton**.

Please follow the exact instructions on uploading the solutions for each task.

**Task 1 – Shapes**

You are given code which reads information about one of 3 possible shapes

* a **Circle** (defined by **radius** and **center**)
* a **Rectangle** (defined by **width**, **height** and **center**)
* a **CoordinateSystemCenter** (not really a shape, always has **(0, 0)** as its center and an **area** of **0**)

The provided code does not have the definition for the base Shape class – your task is to create it and any members necessary for the code to compile and accomplish the task described.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| c 1 3 2 | Circle at (3.00, 2.00), area: 3.14 |
| x | Center at (0.00, 0.00), area: 0.00 |

**Task 2 – Aggregators**

You are given code that reads a series of integer numbers and does an “aggregation” on them, which results in a single integer. Aggregation types are one of:

* sum – sums the items in the series into a single integer
* average – calculates the average of the series, as calculated by the integer division of the sum and the number of items in the series
* min – finds the minimum integer in the series

The provided code handles input and output but lacks the functionality for the aggregations – you need to implement them so that the code compiles successfully and accomplishes the task described.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| average  4 8 -2 | 3 |
| min  4 8 2 | 2 |
| sum  4 8 -2 | 10 |

**Task 3 – Typed Stream**

You are given code that reads a series of objects, spread on multiple lines, then prints them back on the console in a single line. The code uses typed streams, which know how to parse the objects from a string. However, the base class for those streams is missing. Your task is to study the provided code and implement that base class, so that the code compiles and accomplishes the task described.

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3  4 5  end  int | 1 2 3 4 5 |
| this  is  the  string stream  end  string | this is the string stream |
| 0.5 1.1  42 4.2 1.3 13  end  vector | (0.5, 1.1) (42, 4.2) (1.3, 13) |

**Task 4 – Word**

You are given the skeleton of a word-processing program (like MS Word, OpenOffice Writer, etc.). The program reads a line of text from the console, then starts reading commands for editing (text-transform) and executing them on the text. Each command changes the text, the following command works on the changed text. When the command **exit** is entered, the program prints out the modified text and exits. All commands are of the form:  
**commandName startIndex endIndex**  
Where **commandName** is a string describing which command should be used, **startIndex** is an integer which describes from which index in the text the command should be applied, **endIndex** is an integer which describes to which index (exclusive) the command should be applied (i.e. the command is applied on indices starting from **startIndex** and ending in **endIndex - 1** inclusively)

The skeleton you are provided with contains the following files:

* **main.cpp** – contains the **main()** function, reads input and prints output on the console
* **TextTransform.h** – contains a base class for any text-transform added to the program
* **CommandInterface.h** – defines a base class which handles commands represented as strings (coming from the console, read from **main()**)

The code uses an **Initialization.h** file, which is missing, but should define a way to generate a **CommandInterface**.

The files you are given support all logic necessary to implement the following command:

* **uppercase** – transforms any alphabetical character in the text in the range **[startIndex, endIndex)** to its uppercase variant.  
  E.g. if the current text is **som3. text**   
  and we are given the command **uppercase 1 7**  
  the current text will change to **sOM3. Text**  
  Note: if **startIndex == endIndex**, the command has no effect

Your task is to add the following commands:

* **cut** – cuts (removes) characters in the text in the range **[startIndex, endIndex)**, and remembers the last thing that was removed (Hint: **std::string::erase**)  
  E.g. if the current text is **som3. text**   
  and we execute the command **cut 1 7**  
  the current text will change to **sext** (… *I honestly didn’t plan in advance for this to be the result*)  
  Note: if **startIndex == endIndex**, the command has no effect on the text, but “clears” the last remembered cut
* **paste** – replaces the characters in the text in the range **[startIndex, endIndex)** with the characters which were removed by the last cut (Hint: **std::string::replace**)  
  E.g. if we have the text **som3. Text** and the commands  
  **cut 1 7** (text changed to **sext**)  
  **paste 3 4**  
  the current text will change to **sexom3. t**  
  (we paste the last cut – **"om3. t"** – over the **'t'** at the end of the text)  
  Note: if **startIndex == endIndex**, **paste** will insert the text at position **startIndex**, meaning that any text at **startIndex** will be pushed to the right by the inserted text. E.g. if the last command was **paste 0 0** (not **paste 3 4**), the text would be **om3. Tsext**

**Input**

The program defined in **WordMain.cpp** reads the following input:

A line of text, followed by a sequence of lines containing commands of the format   
**commandName startIndex endIndex**,   
ending with the command **exit**.

**Output**

The program defined in **WordMain.cpp** writes the following output:

The modified line of text.

**Restrictions**

The input text will be no more than **30** characters long and there will be no more than **10** commands in the input (this task is not about algorithm optimization).

For **currentTextLength** equal to the current number of characters in the text, for any command:  
**0 <= startIndex <= endIndex < currentTextLength**(i.e. the input will always be valid)

There will always be at least 1 **cut** command before any **paste** command. Consecutive **paste** commands (without **cut** between them) will paste the same text (just like in any text editor – you can cut something and paste it several times).

The total running time of your program should be no more than **0.1s**

The total memory allowed for use by your program is **16MB**

**Example I/O**

|  |  |
| --- | --- |
| Example Input | Expected Output |
| som3. text  cut 1 7  paste 3 4  exit | sexom3. t |
| abc d e  cut 0 4  uppercase 1 3  paste 1 2  exit | dabc E |

**Task 5 – Calculator**

You are given the skeleton of a calculator program (like the Calculator app in Windows, or the calculator on your smartphone, etc.). The program reads numbers and operations from the console and executes those operations on the numbers. The numbers are positive integers, while the operations can be single symbols (e.g. the star symbol '**\***' means multiplication), or strings of characters (e.g. the operation "**end**" stops the program and prints out the result).

Operations are executed immediately after they receive all their needed operands. For example, the expression **3 \* 4 / 2** will first store **3**, then see the multiplication and wait for a number to multiply – when it receives **4** it will calculate **3 \* 4 = 12**, then see the division and wait for a number to divide by – when it receives **2**, it will divide **12** by **2**.

Any number input overwrites the current result of the calculator, just like in normal calculators. For example, if the expression **3 1 \* 4 16 / 2** is input, we’d first have **3**, overwrite it with **1**, multiply by **4** and get **4**, but then we overwrite with **16** and divide that by **2** – the result will be **8**.

The skeleton you are provided with contains the following files:

* **CalculatorMain.cpp** – contains the **main()** function, reads input and prints output on the console
* **Operation.h** – contains a base class for any operation done by the calculator
* **MultiplicationOperation.h** – defines a class which inherits the base **Operation** class and implements the multiplication operation (**\***)
* **CalculationEngine**.**h** – defines the calculator’s central logic of handling number and operations input
* **InputInterpreter.h** –defines a class which can interpret a string into either a number or an operation and invoke the engine accordingly

The files you are given support all logic necessary to implement the **multiplication** operation, as well as console input and output (note that input items don’t need to be on the same line – you can write 1 operation or number per line and the code will still work) but are missing the logic to instantiate an **InputInterpreter**, which should be defined in the missing **Extensions.h** file.

Your task is to study the provided code and add the following operations:

* **/** – division, divides the current result of the calculator by the next number the calculator receives, and pushes the result to the calculator (i.e. same as multiplication, but divides)
* **ms** – saves the current result of the calculator to “memory”. The result of this operation is the current result of the calculator. For example, the expression **3 \* 4 ms \* 5** and the expression **3 \* 4 \* 5** are equivalent in their result
* **mr** – memory recall, removes the last item from memory, and sends it to the calculator. Note that this operation can be used in combination with other operations, for example the expression **3 ms \* 4 ms \* 5 \* mr \* mr** will save **3** to memory, calculate to **12**, save to memory, calculate **60**, multiply that by **12** from memory, resulting in **720**, then multiply that by **3** from memory, resulting in **2160**. It can also be used without operations – **3 ms 4 mr** is the same as **3 4 3**

**Input**

The program defined in **CalculatorMain.cpp** reads the following input:

Strings, representing numbers or operations, separated by spaces (or new lines, or any “blank” space), ending with the string **end**.

**Output**

The program defined in **CalculatorMain.cpp** writes the following output:

The calculated result of all the numbers and operations from the input.

**Restrictions**

The numbers in the input will always be positive integers and no operation will result in a number larger than 1 billion.

There will always be at least 1 **ms** operation before any **mr** operation. There will be no more **mr** operations than the preceding **ms** operations. There will be no **ms** operation following an operation expecting a value (e.g. **3 \* ms 4** is not a valid input, but **3 ms \* 4** is). There will never be an invalid series of operations (e.g. **3 / / 4**, or **3 \* \* 4**, etc.)

The first **40%** of the tests will NOT contain **ms** or **mr** operations.

The total running time of your program should be no more than **0.1s**

The total memory allowed for use by your program is **16MB**

**Example I/O**

|  |  |
| --- | --- |
| Example Input | Expected Output |
| 1 \* 2 \* 3 ms \* 4 \* mr / 2 end | 72 |
| 12 / 3 ms / 2 ms \* 5 mr \* mr end | 8 |