Polymorphism - Exercise

Please submit your solutions (source code) to all below-described problems in Judge.

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

1. Duplicates

You are given a code that reads information about companies – each having a **name** and a **list of employees by their initials** – and removes duplicate companies (companies with the same name are considered duplicates). The given code accomplishes this by using a function named **`removeDuplicates`**, which accepts a single parameter – a **`list`** of **`Company*`**.

Note that the list of pointers may contain multiple pointers pointing to the same **`Company`** object, as well as pointers pointing to different objects, which have the same name. The function should remove any duplicates but leave their first occurrence in the list (i.e. the first time a **`Company`** is found in the list, it is considered the "original" and all subsequent Companies with the same name are considered duplicates).

Also note that you should ensure that removed duplicates are cleared from memory, using the **`delete`** keyword.

You should submit a single **`.zip`** file for this task, containing ONLY the **`RemoveDuplicates.h`** file, containing an **`int main()`** function that solves the task described.

RemoveDuplicates.h

```
#ifndef REMOVE_DUPLICATES_H
#define REMOVE_DUPLICATES_H


#include "Company.h"
// Place your code here


#endif // !REMOVE_DUPLICATES_H
```

The Judge system has a copy of the other files and will compile them, along with your file, in the same directory.

Examples

Input	Output	Explanation

`uni (I.K.,S.N.)`
`*begin`
`*end`
`*begin`
`joro (G.G.)`
`*begin`
`*begin`
`end`
	`joro (G.G.)`
	`uni (I.K.,S.N.)`
		The *begin line means insert a pointer to the last company at the start of the list. The *end line means insert such a pointer at the end of the list. Hence, after reading the input, the program will have the following list (* denotes an object inserted as a copied pointer):
		*joro (G.G.)
		*joro (G.G.)
		*uni (I.K.,S.N.)
		*uni (I.K.,S.N.)
		uni (I.K.,S.N.)
		*uni (I.K.,S.N.)
		joro (G.G.)

		In this list, the joro company is followed by the uni company, even though it is created later – that's due to the inserting of the *begin pointers after creating the joro company. Hence joro will appear first in the output, and uni second, because the first pointer to joro in the list is before the first pointer to uni in the list. The idea behind this unusual input is that your code for the removeDuplicates function should not assume anything about the input – just traverse the list and remove the duplicates, don't attempt to figure out a way to cheat the input

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`

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`	`this is the string stream`

`is`
`the`
`string stream`
`end`
`string`	

`0.5 1.1`
`42 4.2 1.3 13`
`end`
`vector`
	`(0.5, 1.1) (42, 4.2) (1.3, 13)`

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 that describes from which index in the text the command should be applied, **`endIndex`** is an integer that 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 that 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`**
  <u>Note:</u> 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`**)
  For 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`**, the **`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

Input	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`

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 that 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 that 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

Input	Output
`1 * 2 * 3 ms * 4 * mr / 2 end`	`72`
`12 / 3 ms / 2 ms * 5 mr * mr end`	`8`

6. Pistol Round

You will be playing a very famous console version of a computer shooter game where Terrorists fight versus Counter-Terrorists.
Usually, in that game the very first round of the game is called a "Pistol Round" and it is very important. In that round, both playing teams have money only to purchase guns. The winner of this round has enough money to buy automatic weapons in the next round, while the loser team sticks to their pistols.

Your task is to write a program that battles two players from opposing teams (Player 0 and Player 1).
Most of the game is already coded.
In the game you should find a header prototypes for the 2 different supported Pistol types:

```
enum class PistolType
{
      GLOCK        = 0,
      DESERT_EAGLE = 1
};
```

Your task is to provide a **concrete implementation** for the **DesertEagle** and **Glock** classes.
Also, there is a missing functionality in the **Player** class. You should provide that implementation as well.

The game consists of both players taking turns shooting each other with their provided pistols for the round.
First goes Player 0, then Player 1 …, and so on until one of them dies (has no health left).

Each player has 2 vital attributes "health" and "armor"

```
struct PlayerVitalData
{
      int health;
      int armor;
};
```

The armor is a way for the player to negate some of the incoming damage that he is about to take.
A player will **not** always have armor.

Each player also acquires a Pistol at the start of the round (determined by the input).
This Pistol is either a DesertEagle or a Glock.
Both Pistols have the same attributes, but they do have different effects.

```
class Pistol
{
      protected:
            int _damagePerRound;
            int _clipSize;
            int _currClipBullets;
            int _remainingAmmo;
};
```

The Pistol attributes explained:

- `damagePerRound` – indicates how much damage does each bullet deal
- `clipSize` – indicates how much bullet capacity the concrete pistol clip has
- `currClipBullets` – indicates how many bullets are left in the current clip
- `remainingAmmo` – indicates how many spare bullets are left for the player to reload with. This count **does not include** in itself the bullets that are loaded in the current pistol clip

The input to the program consists of (in that order):

- player 0 health, player 0 armor
- player 1 health, player 1 armor
- player 0 PistolType, player 0 pistol damage per round, player 0 clipSize, player 0 remainingAmmo
- player 1 PistolType, player 1 pistol damage per round, player 1 clipSize, player 1 remainingAmmo

Game rules:

- At the start of the game – each player has a full clip of bullets ready and loaded.
- Players take turns shooting at each other (starting always from Player with ID: 0)
- If a player's pistol is required to fire and current bullets in the **clip reach 0** – the player **must reload its pistol**. **The player loses his remaining turn and does not deal any more damage for this turn.**
- After each shout fired (inside the ::fire() method) you should **print** to the console the opponent player's vital details following this exact format:
  `"Enemy left with: N health and M armor"` followed by a **newline**.
  Where **N** indicates the remaining opponent's **health** (after the shot) and
  M indicates the remaining opponent's **armor** (after the shot)
- If a player has no bullets left on each of his next turns print "`No ammo left`" followed by a **newline.**

Common **DesertEagle** and **Glock** rules:

- If an opponent has a **positive armor value** and the shot fired against this opponent has **a bigger value than the armor –** the armor is set to 0 and the remaining damage is **subtracted** from the target's health indicator.
- Both **DesertEagle** and **Glock** ::fire() methods should return a bool variable, which indicates whether or not the target (enemy player) has been killed by this round of bullets that were fired.
  An opponent is considered killed when his health indicator drops below 0 (or is equal to 0).

DesertEagle Pistol specifics (see concrete examples at the 'Examples' section):

- If the opponent has no armor:
  - Deals 100% of the pistol's original damage to the opponent's health indicator
- If the opponent has armor:
  - Deals 75% of the pistol original damage to the opponent's health indicator
  - Deals 25% of the pistol original damage to the opponent's armor indicator
- Pistol damage will **always** be dividable by 4 (Example: 32, 16, 4, 100)
- When reloading – no shots will be made in the same turn.

Glock Pistol specifics (see concrete examples at the 'Examples' section):

- **Fires 3 bullets one after another for a single turn**. A print to the console should be made **for each** shot fired.
  If there are less than 3 bullets remaining for this turn – fire all the remaining bullets and initiate a **reload**.
  No bullets should be fired after the reload in the same turn.

- If the opponent has no armor:
    - Deals 100% of the pistol original damage to the opponent's health indicator
- If the opponent has armor:
    - Deals 50% of the pistol original damage to the opponent's health indicator
    - Deals 50% of the pistol original damage to the opponent's armor indicator
- Pistol damage will **always** be dividable by 4 (Example: 36, 8, 112, 44)

Restrictions

You should only submit **.h** and **.cpp** files compressed in a **.zip** archive.

There should be no folders in your **.zip** archive.

Examples

Input	Output

```
100 0
120 0
0 8 9 71
1 24 7 35
```
	```
	PlayerID 0 turn:
	Enemy left with: 112 health and 0 armor
	Enemy left with: 104 health and 0 armor
	Enemy left with: 96 health and 0 armor

	PlayerID 1 turn:
	Enemy left with: 76 health and 0 armor

	PlayerID 0 turn:
	Enemy left with: 88 health and 0 armor
	Enemy left with: 80 health and 0 armor
	Enemy left with: 72 health and 0 armor

	PlayerID 1 turn:
	Enemy left with: 52 health and 0 armor

	PlayerID 0 turn:
	Enemy left with: 64 health and 0 armor
	Enemy left with: 56 health and 0 armor
	Enemy left with: 48 health and 0 armor

	PlayerID 1 turn:
	Enemy left with: 28 health and 0 armor

	PlayerID 0 turn:
	Reloading...
	currClipBullets: 9, remainingAmmo: 62

	PlayerID 1 turn:
	Enemy left with: 4 health and 0 armor
	```

	PlayerID 0 turn: Enemy left with: 40 health and 0 armor Enemy left with: 32 health and 0 armor Enemy left with: 24 health and 0 armor PlayerID 1 turn: Enemy left with: -20 health and 0 armor Player with ID: 1 wins!

100 50
120 50
0 12 8 50
1 24 4 24
	PlayerID 0 turn:
	Enemy left with: 114 health and 44 armor
	Enemy left with: 108 health and 38 armor
	Enemy left with: 102 health and 32 armor
	
	PlayerID 1 turn:
	Enemy left with: 82 health and 44 armor
	
	PlayerID 0 turn:
	Enemy left with: 96 health and 26 armor
	Enemy left with: 90 health and 20 armor
	Enemy left with: 84 health and 14 armor
	
	PlayerID 1 turn:
	Enemy left with: 64 health and 38 armor
	
	PlayerID 0 turn:
	Enemy left with: 78 health and 8 armor
	Enemy left with: 72 health and 2 armor
	Reloading...
	currClipBullets: 8, remainingAmmo: 42
	
	PlayerID 1 turn:
	Enemy left with: 46 health and 32 armor
	
	PlayerID 0 turn:
	Enemy left with: 62 health and 0 armor
	Enemy left with: 50 health and 0 armor
	Enemy left with: 38 health and 0 armor
	
	PlayerID 1 turn:
	Enemy left with: 28 health and 26 armor
	
	PlayerID 0 turn:
	Enemy left with: 26 health and 0 armor
	Enemy left with: 14 health and 0 armor
	Enemy left with: 2 health and 0 armor
	
	PlayerID 1 turn:
	Reloading...
	currClipBullets: 4, remainingAmmo: 20
	
	PlayerID 0 turn:
	Enemy left with: -10 health and 0 armor

	`Player with ID: 0 wins!`

```
200 100
120 20
1 16 4 3
1 24 7 35
```
	```
	PlayerID 0 turn:
	Enemy left with: 108 health and 16 armor

	PlayerID 1 turn:
	Enemy left with: 182 health and 94 armor

	PlayerID 0 turn:
	Enemy left with: 96 health and 12 armor

	PlayerID 1 turn:
	Enemy left with: 164 health and 88 armor

	PlayerID 0 turn:
	Enemy left with: 84 health and 8 armor

	PlayerID 1 turn:
	Enemy left with: 146 health and 82 armor

	PlayerID 0 turn:
	Enemy left with: 72 health and 4 armor

	PlayerID 1 turn:
	Enemy left with: 128 health and 76 armor

	PlayerID 0 turn:
	Reloading...
	currClipBullets: 3, remainingAmmo: 0

	PlayerID 1 turn:
	Enemy left with: 110 health and 70 armor

	PlayerID 0 turn:
	Enemy left with: 60 health and 0 armor

	PlayerID 1 turn:
	Enemy left with: 92 health and 64 armor

	PlayerID 0 turn:
	Enemy left with: 44 health and 0 armor
	```

```
PlayerID 1 turn:
Enemy left with: 74 health and 58 armor

PlayerID 0 turn:
Enemy left with: 28 health and 0 armor

PlayerID 1 turn:
Reloading...
currClipBullets: 7, remainingAmmo: 28

PlayerID 0 turn:
No ammo left

PlayerID 1 turn:
Enemy left with: 56 health and 52 armor

PlayerID 0 turn:
No ammo left

PlayerID 1 turn:
Enemy left with: 38 health and 46 armor

PlayerID 0 turn:
No ammo left

PlayerID 1 turn:
Enemy left with: 20 health and 40 armor

PlayerID 0 turn:
No ammo left

PlayerID 1 turn:
Enemy left with: 2 health and 34 armor

PlayerID 0 turn:
No ammo left

PlayerID 1 turn:
Enemy left with: -16 health and 28 armor

Player with ID: 1 wins!
```