01-Rule of Three / Five / Zero - 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.
Code should compile under the C++03 or the C++11 standard.
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. MinBy

You are given code for a program, which finds the minimum element of a list of strings entered on the console (lowercase English letters, separated by spaces). The program has 3 modes of operation, each using a different characteristic for finding the minimum:

- Lexicographical minimum i.e. find the string that is earliest lexicographically
- Size minimum i.e. finds the shortest string
- Reverse size minimum i.e. finds the longest string

The code is missing the **`MinBy.h`** file. Your task is to study the provided code and create a **`MinBy.h`** file such that the program compiles successfully and performs the described task.

Your **`MinBy.h`** file should resemble the following:

MinBy.h

```
#ifndef MIN_BY_H
#define MIN_BY_H

// Place your code here

#endif // !MIN_BY_H
```

You should submit a single **`.zip`** file for this task, containing ONLY the **`MinBy.h`** file. The Judge system has a copy of the other files and will compile them, along with your file, in the same directory.

Restrictions

There will always be at least **1** string in the list of strings entered from the console. There will be no empty strings.

Examples

Input	Output
hear the rime of the ancient mariner see his eyes as he stops one of 3	ancient
e abc df e 1	Abc

2. Remove Invalid

You are given code for a program, which reads **Company** objects from the console, allocates dynamic memory for them, adds them to a list, and calls a **removeInvalid** function multiple times during the read and once after all companies have been read, to **remove companies with invalid ids**. The program then **prints all companies with a valid id**. Valid ids are **non-negative**.

Your task is to implement the **removeInvalid** function a **RemoveInvalid.h** file. The function should remove all companies with negative ids (use the **getId()** getter in **Company**) from the list.

Your **RemoveInvalid.h** file should resemble the following:

RemoveInvalid.h

```
#ifndef REMOVE_INVALID_H
#define REMOVE_INVALID_H

#include "Company.h"

// Place your code here

#endif // !REMOVE_INVALID_H
```

You should submit a single **.zip** file for this task, containing ONLY the **RemoveInvalid.h** file. The Judge system has a copy of the other files and will compile them, along with your file, in the same directory.

Additional Requirements

The order of the valid companies in the input must match the order in the output.

Make sure there are no memory leaks.

Examples

Input	Output
123 valid -123 invalid 321 ancientmarinerinc end	123 valid 321 ancientmarinerinc
42 noinvalidhere end	42 noinvalidhere

3. Make Company

You are given a program in a **MakeCompanyMain.cpp**, as well as a **Company.h** file, that reads information about **companies** and writes it to the console.

Each company has:

An **id** (a 4-byte integer)

A **name** (a **string** containing a sequence of lowercase English letters **a-z**)

Employees by their initials (a **vector** of **pair**s of characters, containing at most **255** employee initials)

The **MakeCompanyMain.cpp** file reads the information from the console as a single line for each company, containing:

- company **id** string
- space
- company **name**
- 2 characters representing the initials of the first employee
- Space
- Again 2 characters, representing the second employee, etc

NOTE: there could be companies without employees, in which case the line ends with the company name.

For example, if we have the companies:

- **id = 42, name = "uni", employees = { {'I', 'K'}, {'S', 'N'} }** and
  id = 13, name = "joro", employees = { {'G', 'G' } }

Their representation as **string**s read by **MakeCompanyMain.cpp** will be:

```
42 uni IK SN
13 joro GG
```

The program reads each line from the console, then calls a function named **makeCompany** to **dynamically allocate memory** for a **Company** object and set its values, then prints its representation a back to the console (i.e. the program should print the exact line it read), until a line containing the single string **"end"** is reached.

Your task is to create a file called **MakeCompany.h** containing the function **makeCompany** such that the program compiles successfully and performs the described task.

Your file should resemble the following:

MakeCompany.h

```
#ifndef MAKE_COMPANY_H
#define MAKE_COMPANY_H

#include "Company.h"

// Place your code here

#endif // !MAKE_COMPANY_H
```

You should submit a single **.zip** file for this task, containing ONLY the **MakeCompany.h** file. The Judge system has a copy of the other files and will compile them, along with your file, in the same directory.

Additional Requirements

Make sure you correctly create the Company objects and ensure that there are **no memory leaks**.

Examples

Input	Output
42 uni IK SN 13 joro GG end	42 uni IK SN 13 joro GG
188 noemployees 58 oneemployee SA	188 noemployees 58 oneemployee SA

end	

4. Parse Unique Companies

You are given code for a program, which reads lines from the console containing the information about **Company** objects, sends those lines as a single **`string`** to a function named **`parseUniqueCompanies`**, and expects the function to return a pointer to a dynamically allocated array, which the program iterates and prints information about each parsed **Company**.

In the input, each **Company** is described by a single line, containing the **Company** id (an integer) followed by the **Company** name (a string), separated by a single space.

In addition to the string, the program passes two other parameters:

- an integer, which it expects to be set (by the **`parseUniqueCompanies`** function) with the **number of companies** in the input
- a function pointer/reference to a function that generates a unique identifier for a company. The **`parseUniqueCompanies`** function is expected to return no more than one company (the first in the input) for each unique identifier

The program decides which unique identifier function to use based on a number in the last line of the input:

- a unique identifier is the **Company**'s id
- a unique identifier is the **Company**'s name
- a unique identifier is the **Company**'s name concatenated by the **Company**'s id

Your task is to implement the **`parseUniqueCompanies`** function in a **`ParseCompanies.h`** file.

Your **`ParseCompanies.h`** file should resemble the following:

ParseCompanies.h

```
#ifndef PARSE_COMPANIES_H
#define PARSE_COMPANIES_H

#include "Company.h"

// Place your code here

#endif // !PARSE_COMPANIES_H
```

You should submit a single **`.zip`** file for this task, containing ONLY the **`ParseCompanies.h`** file. 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
123 valid 123 copy 321 ancientmarinerinc end 1	123 valid 321 ancientmarinerinc

42 theanswertolifetheuniverseandeverything 420 theanswertolifetheuniverseandeverything end 3	42 theanswertolifetheuniverseandeverything 420 theanswertolifetheuniverseandeverything

5. Register of Three

NOTE: this task is the same as the task **Register** from previous exercises , however the **`main()`** function in the skeleton is different and requires you to implement the Rule of Three for the **`Register`** class.

You are given code that reads information about **Company** objects from the console, parses it multiple times (the number of repetitions is entered on the first line on the console), and prints the information about one of the **Company** objects, specified by its **id**.

The provided code handles input, output, and the repeated executions – your task is to **implement** the **`Register`** (which is declared in the **`Register.h`** file, you need to create the **`Register.cpp`** file) class it uses for storing and looking up the **Company** objects.

You should submit a single **`.zip`** file for this task, containing ONLY the file(s) YOU created. The Judge system has a copy of the other files and will compile them, along with your file, in the same directory.

Restrictions

There will always be a **Company** with the specified **id**.

Make sure there are no memory leaks.

Examples

Input	Output
1 2 42 theanswer 69 thehub 42 end	42 theanswer
1000 2 42 theanswer 69 thehub 42 end	42 theanswer

6. Array of Pointers

NOTE: this task is the same as the tasks **Register**, and **Task 5 – Register of Three,** however it does NOT use a `Register` class, but instead allocates **Company** objects in dynamic memory and uses an **`ArrayOfPointers`** class to store these pointers (NOTE: the provided code does NOT deallocate the memory it allocates).

You are given code that reads information about **Company** objects from the console, parses it multiple times (the number of repetitions is entered on the first line on the console), and prints the information about one of the **Company** objects, specified by its **id**.

The provided code handles input, output, and repeated executions – your task is to **declare** and **implement** the **ArrayOfPointers** class in any way you think will accomplish the task, **without leaking memory**. The provided code expects the declaration to be in an **ArrayOfPointers.h** file, but you are free to choose whether to implement the class in the same file, in a **.cpp** file, or multiple files.

You should submit a single **.zip** file for this task, containing ONLY the file(s) YOU created. The Judge system has a copy of the other files and will compile them, along with your file, in the same directory.

Restrictions

There will always be a **Company** with the specified **id**. Make sure there are no memory leaks.

Examples

Input	Output
1 2 42 theanswer 69 thehub 42 end	42 theanswer
1000 2 42 theanswer 69 thehub 42 end	42 theanswer

7. List

You are given a **List.h** file containing the declarations for a **List** class representing a linked list, and a **ListMain.cpp** file, which defines a **main()** function and uses the **List** class to merge several sorted lists from the standard input into a single sorted list printed on the standard output.

- Create a **List.cpp** file which contains the implementation of the **List** class.
- The files should be successfully compiled together.
- The resulting program should correctly merge sorted lists read from the console into a sorted list, which should be printed on the console (main.cpp does this if you implement List.cpp).
- Submit a **.zip** file containing the **List.cpp** file and nothing else.

How you choose to implement the linked list is up to you, but you should make sure all the public methods of the **List** class work correctly, as they are used by main.cpp. The declarations in **List.h** should be mostly self-explanatory, but if you are unsure what a method should do – just see how **ListMain.cpp** uses it and make sure you implement it so that the program works correctly.

You are NOT allowed to modify **ListMain.cpp** or **List.h**.

The task this program solves is merging multiply sorted (ascending) lists of integer numbers into a single sorted (ascending) list of integer numbers. For example, the lists **1 17**, and **-3 6 25 42** should be merged into the following list: **-3 1 6 17 25 42**.

Input

One or more lines, each of which contains from **1** to **100** integers, separated by single spaces. The final line will not contain numbers and will only contain the string **"end"**.

Output

A line containing the items of the merged, sorted list, in ascending order, separated by single spaces.

Restrictions

The total number of elements entered in the input will NOT exceed **10000.**

The number of elements per input list (line) will NOT exceed **100.**

Numbers in the input will be from **-9999** to **+9999** (both inclusive).

The total running time of your program should be no more than **0.5s.**

The total memory allowed for use by your program is **5MB.**

Examples

Input	Output
1 17 -3 6 25 42 end	-3 1 6 17 25 42
4 5 6 1 2 3 end	1 2 3 4 5 6
1 3 2 end	1 2 3

8. Overloading Madness

You are given 2 files: main.cpp and Matrix.h.

Your task is to study the provided Skeleton and implement the missing functionalities for Matrix.cpp.

As the name states the Matrix class is a representation of a simple 2D array of integers.

You need to implement the overloading of 4 math operations 'add', 'subtract', 'multiply', 'divide', and an additional overload for operator<<, which will print to the standard output (the console).

For the example let's assume we have 2x2 Matrix A == Matrix B == |3 3|
|3 3|

- A += B would result in |6 6|
|6 6|
A -= B would result in |0 0|
|0 0|

- A *= B would result in |9 9| A /= B would result in |1 1|
  |9 9| |1 1|

  Important note: remember that in math you can not divide by 0! Otherwise, the universe would explode. If there seems to be any case of division by 0 -> simply treat the result as 0.
  Example 5 / 0 = 0.

Keep in mind that the matrix sizes will **NOT** always be the same. In this case - simply perform the operation on their **common intersection** (the smaller matrix). You are **assured** that in this case, the bigger matrix in size will be from the **LEFT** side of the mathematical operand (will simply be from the left).

For the example let's assume we have 3x3 Matrix A |1 2 3| and Matrix B |4 3|
|4 5 6| |1 0|
|7 8 9|

- A += B would result in |5 5 3|
  |5 5 6|
  |7 8 9|

For the operator<< - print **whitespace** separated all elements of the matrix and at the end of each row print a **newline**.

Note: there is **whitespace** even after the last element on each row, before the **newline**.

You are given the **`main()`** function, which read from the standard input and populates MATRICES_COUNT number of Matrices (each Matrix may have a different size and will always be a **square**(number of rows == number of cols)).

The matrices are then sorted (by their sizes) in Descending order (or at least their indexes).

The next 5 actions will be:

- the 'first' Matrix is added with the 'second' Matrix
- the 'first' Matrix is subtracted from the 'third' Matrix
- the 'first' Matrix is multiplied by the 'fourth' Matrix
- the 'first' Matrix is divided by the 'fifth' Matrix
- the 'first' Matrix is printed to the standard output (the console)

Your task is to study the code and implement the function so that the code accomplishes the task described.

You should submit a single **`.zip`** file for this task, containing **ONLY** the files you created.

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

Restrictions

Mathematical division by 0 (zero) is not allowed. Handle this case as explained above.

Examples

Input	Output
2 5 5 5 5 2 4 4 4 4	2 6 6 6

2 3 3 3 3 1 2 1 5	
2 5 5 5 5 2 4 4 4 4 2 3 3 3 3 2 2 2 2 2 2 5 20 20 0	2 0 0 0
1 10 2 1 2 3 4 3 9 8 7 6 5 4 3 2 1 4 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 5 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9	0 6 4 11 9 15 24 7 11 9 8 9 10 11 9 11 11 11 11 9 9 9 9 9 9