**03. Exercise-Pointers-References-Memory**

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

Submit your solutions here: <https://judge.softuni.bg/Contests/1153/03-Exercises-on-Pointers-References-and-Dynamic-Memory>

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 – Sort

You are given information about companies – lines with names and ids, ending with the line "end" – as well as a Company class that represents a company. Your task is to sort the companies either by their name or by their id. The last line of the input will contain either "name" or "id", indicating what you should sort the companies by

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

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** |
| acme 424242420  softuni\_foundation 20140414  itjoro 878968302  end  id | softuni\_foundation 20140414  acme 424242420  itjoro 878968302 |

# Task 2 – Sort Pointers

You are given information about companies – lines with names and ids, ending with the line "end" – as well as a Company class that represents a company. Your task is to sort the companies either by their name or by their id. The last line of the input will contain either "name" or "id", indicating what you should sort the companies by

You are also given code that does the reading and writing to the console, and uses a function named sortBy, defined in a file SortBy.h, to do the actual sorting. Note that the code you are given works on pointers, instead of normal Company objects. The sortBy function accepts 3 parameters:

* A Company\* to the first company in an array of companies
* A Company\* pointing AFTER the last company in the array of companies
* A function pointer/reference, which compares 2 Company objects (not pointers) and returns true if its first parameter is “less than” its second parameter. Study the SortPointersMain.cpp file for more information on what functions can be passed in to your function

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

|  |
| --- |
| **SortBy.h** |
| #ifndef SORT\_BY\_H  #define SORT\_BY\_H  #include "Company.h"  // Place your code here  #endif // !SORT\_BY\_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** |
| acme 424242420  softuni\_foundation 20140414  itjoro 878968302  end  id | softuni\_foundation 20140414  acme 424242420  itjoro 878968302 |

# Task 3 – Duplicates

You are given 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 |

# Task 4 – Ranges

A range is a **pair of integer numbers** – let’s say that from and to form the range[from, to].

If an integer number x is such that from <= x <= to, then we say that x is **inside** the range [from, to]**,** or that the range[from, to]containsx.

You are given a set of ranges, in which no two ranges intersect. That means that no range contains the from or to of another range.

You are also given a sequence of integer numbers – let’s call them **check numbers**.  
For each of the **check numbers**, print "in" if the number is inside any range, and "out" otherwise (i.e. if no range contains the number).  
**NOTE:** there will be a large number of ranges and an even larger number of integer numbers.

### Input

The input will be separated into two parts.

The first part will contain the ranges, each described as two integer numbers on a separate line of the standard input (the from and to of the range), until a line containing only the symbol '.' (dot) is reached.

After that, each line of the standard input will contain exactly one check number, until a line containing only the symbol '.' (dot) is reached.

### Output

For each **check number** in the input, print "in" if that number is contained in any range, or "out" if no range contains that number.

### Restrictions

There will be between 1 and 10000 ranges (inclusive).  
There will be between 1 and 100000 check numbers (inclusive).  
For every range, from <= to.  
In 30% of the tests, there will be no more than 10 ranges and 10 numbers.  
The total running time of your program should be no more than 0.4s  
The total memory allowed for use by your program is 8MB

### Example I/O

|  |  |
| --- | --- |
| Example Input | Expected Output |
| 1 3  5 10  20 20  .  0  2  3  4  5  7  19  20  . | out  in  in  out  in  in  out  in |
| -5 0  1 3  .  -10000  -1000  0  10  . | out  out  in  out |