Inheritance – Exercise

This document defines the lab for the "C++ OOP" course @ Software University. 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. 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

2. 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]`** contains **`x`**.

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 to **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

Input	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	out

1 3 . -10000 -1000 0 10 .	out in out

3. Max Sum Array

You are given code that reads arrays from the console and prints the array with the maximum sum of elements. The code uses an Array class that you have to implement – make sure you handle memory management correctly.

You should submit only the file(s) you created. The Judge system has the other files and will compile them, along with your file(s), in the same directory.

Restrictions

There will be between **1** and **1000** (inclusive) arrays in the input. Each array will have no more than **1000** elements, and each of its elements will be a value between **-100** and **+100** (inclusive).

Examples

Input	Output
3 4 1 -2 3 4 1 505 2 13 42	505

4. Shapes

You are given code that 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

5. Serialize

NOTE: this task is the reverse of Task 5 – Memory from lecture 01. Pointers and References.

You are given a program that reads information about **companies** and writes it to the console.

Each company has:

- An **id** (an integer between **0** and **255**)
- 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 program reads the information in its string representation and calls a function named **serializeToMemory**. The function should parse the companies from the input and then write them to memory as a sequence of bytes in a dynamically allocated array (the format is detailed below). The function will be called with the following two parameters:

- A **string** containing lines, where each line is the **string** representation of a **Company**
- An integer which the program expects to be set with the number of bytes serialized to memory which contains the representation of the companies from the first parameter

The program expects the function **serializeToMemory** to return a pointer to the memory where the companies have been written (serialized).

The memory format of each company is the following:

- the first byte contains the **id** of the company (**0-255**)
- the **name** of the company starts from the second byte and ends with a null terminator (the value **0**, or **'\0'**), i.e. the name of the company is placed in memory the same way a null-terminated C-String would be
- the next byte contains the number of employees the company has (**0-255**). Let's call it **numEmployees**
- the following **numEmployees * 2** bytes contain pairs of initials of the employees, i.e. if the **numEmployees** byte is at address **x**, then the **first employee's first initial** is at address **x + 1**, their **second initial** is at address **x + 2**, the **second employee's first initial** is at address **x + 3** and their **second** is at address **x + 4** and so on.

Additionally, since there can be more than one Company:

- the **first byte** in the memory describing the companies contains an **integer representing the number of companies** serialized

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 a **string** read by the program and passed to **serializeToMemory** will be:

```
"42 uni (I.K.,S.N.)\n13 joro (G.G.)"
```

Their representation in memory, assuming the memory starts at byte address **M,** will be:

Offset from start	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14
Value	2	42	'u'	'n'	'i'	'\0'	2	'I'	'K'	'S'	'N'	13	'j'	'o'	'r'

Offset from start	+14	+15	+16	+17	+18
Value	'o'	'\0'	1	'G'	'G'

And their representation in the output for the task will be:

2 42 117 110 105 0 2 73 75 83 78 13 106 111 114 111 0 1 71 71

Your task is to create a file called **Serialize.h** containing the function **serializeToMemory**, implemented in such a way that the program compiles and works as described.

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

Hint: the Company class supports reading from a stream, so you don't need to implement the parsing of the string yourself. The following code reads companies from the **string**, until there are no more companies to read:

```
std::istringstream companiesIn(companiesString);
Company company;
while (companiesIn >> company) { }
```

Examples

Input	Output (*NOTE: single line*)
42 uni (I.K.,S.N.) 13 joro (G.G.) end	2 42 117 110 105 0 2 73 75 83 78 13 106 111 114 111 0 1 71 71
188 icyha (B.Q.,H.P.,F.S.) 58 uadel (S.A.,C.H.,L.T.) end	2 188 105 99 121 104 97 0 3 66 81 72 80 70 83 58 117 97 100 101 108 0 3 83 65 67 72 76 84
13 joro (G.G.) end	1 13 106 111 114 111 0 1 71 71

6. Indexed Set

You are tasked with implementing methods for an **IndexedSet** class. An indexed set works just like a normal **std::set**, but also keeps an array of the sorted elements and can access the **i**th element immediately. For example, to get the **5**th element of a normal **std::set** you need to run a **for** loop **5** iterations from the **begin()** of the set – the **IndexedSet** supports the **operator[]** and you can directly ask for the **5**th element, retrieving it in constant time i.e. **O(1)**, just like with an array/vector.

To work fast, the **IndexedSet** doesn't constantly update the sorted array – it does a so-called "lazy-initialization" of the sorted array, creating it only when it is needed and clearing it when the **IndexedSet** is modified. More formally:

- The first time **operator[]** is called, the sorted array is built
- If **operator[]** is called and the sorted array exists, it is used without being rebuilt. If it doesn't exist (or is empty – implementations can vary), it is rebuilt and then used.
- If the **IndexedSet** is modified, the sorted array is cleared, so that the next **operator[]** call will rebuild it to match the current contents of the **IndexedSet**
- You are given a skeleton containing the declaration of an **IndexedSet** class and its members. You need to implement the **IndexedSet** class in a new **IndexedSet.cpp** file:
- Fields **std::set<Value> valueSet** and **Value * valuesArray** represent the actual set and the sorted array of the set's elements, correspondingly
- **IndexedSet()** – constructs an empty **IndexedSet**
- **IndexedSet(const IndexedSet& other)** – copy-constructs **IndexedSet**

- **`void add(const Value& v)`** – adds an element to the **`IndexedSet`** (and clears the sorted array, so that the next call to **`operator[]`** will rebuild it)
- **`size_t size() const`** – returns the size of the set (the number of elements)
- **`const Value& operator[](size_t index)`** – returns a **`const`** reference to the element at the given index (and constructs the sorted array, if necessary, as described above). *Hint: don't worry about* **`const`***, just return the value at that position in the array*
- **`IndexedSet& operator=(const IndexedSet& other)`** – copy-assigns **`IndexedSet`**
- **`~IndexedSet()`** – destructs **`IndexedSet`** (only necessary to clear the sorted array)
- **`buildIndex()`** and **`clearIndex()`** are intended as helper methods you can define to create and clear the sorted array – they aren't used in external code, so you can choose whether you want to implement them

The skeleton also contains the **`main.cpp`** file, which defines the **`main()`** function. The program in **`main.cpp`** reads a series of arrays from the console, then reads a line of indices from the console. It then finds the array, for which the sum of the indices (read last from the console) of its elements – when the elements are sorted and the duplicates removed – is the maximum of the given arrays, and prints it in that form (sorted with duplicate values removed). To do that, it uses the **`IndexedSet`**.

For example, given the following arrays:

```
1 1 7 2 7 3 1
4 12 2 8 8
5 6 5 1 1 1
```

and the indices **`0`** and **`2`**, the program will sort and remove duplicates from the arrays:

```
1 2 3 7
2 4 8 12
1 5 6
```

and will print the result **`2 4 8 12`**, because that array has the maximum sum of the indices **`0`** and **`2`** (the sum is **`2 + 8 = 10`**, which is bigger than both **`1 + 3`** and **`1 + 6`** for the other two arrays). The program uses **`IndexedSet`** to get a representation of the sorted arrays with removed duplicates.

Input

The program is defined in **`main.cpp`** reads the following input:

One or more lines from the standard input, containing the input arrays, ending with a line containing the string **`end`**. Then one more line containing the indices, which will be used for the sums for determining the output array.

Output

The program is defined in **`main.cpp`** writes the following output:

The sorted array with duplicates removed, which has the highest sum of the indices defined in the input (the sum is calculated after the array is sorted and its duplicates are removed).

Restrictions

There will be between **`1`** and **`50`** (inclusive) arrays in the input. Each array will have no more than **`2000`** elements, and each of its elements will be a value between **`0`** and **`2000`** (inclusive).

There will be between **`1`** and **`5000`** indices in the input. No index will be larger than the number of elements in the smallest array after all its duplicate elements have been removed. That is, there is no need to add range checks and validation logic to the **`IndexedSet`** class.
The total running time of your program should be no more than **`0.3s`**
The total memory allowed for use by your program is **`8MB`**

Submission Instructions

Submit only a single file – **`IndexedSet.cpp`**, containing the implementation of the **`IndexedSet`** class, as declared in **`IndexedSet.h`**.

Example I/O

Input	Output
5 1 3 7 9 3 2 1 4 12 10 9 8 10 100 15 2 3 4 end 0 1 2	8 9 10 12
1 1 1 2 3 2 1 1 3 3 3 end 1 1 1 1	1 3

7. Memory Allocator Reforged

Your task is to write a memory allocator, which does not introduce a memory leak. You are given the **`main()`** function, which reads a single value (as an integer number) of memory followed by N command lines.

- The integer value indicates the number of following command lines (N) you need to process and execute (in the range **`[0, INT_MAX]`** inclusive).
- The next **N** lines indicate the command that you should process and execute.
  The commands have the following syntax:
  - "Allocate Single INDEX" – allocate memory for a single integer on INDEX in your memory allocator;
  - "Allocate Multiple INDEX" – allocate memory for ALLOCATION_MULTIPLE_SIZE (see Defines.h) on INDEX in your memory allocator;
  - "Deallocate Single INDEX" – deallocate memory for a single integer on INDEX in your memory allocator;
  - "Deallocate Multiple INDEX" – deallocate memory for multiple integers on INDEX in your memory allocator;
    Where INDEX can be any integer in the range (in the range **`[0, 9]`** inclusive);

You should implement the functions **executeCommand()** and **printResult()** in another .cpp file. (For example MemoryAllocatorReforged.cpp)

For each executed command in the **executeCommand()** – you should print a status message depending on the received **ErrorCode** in **printResult()**. Every call to **printResult()** should end with a **newline**.

You should print:

- For successful allocation/deallocation (not introducing memory leak or crashing the problem) – "**command** - success"
- For preventing a memory leak – "**command** - memory leak prevented, will not make allocation"
- For preventing a system crash – "**command** - system crash prevented, will skip this deallocation"
- For trying to deallocate a 'Single' memory node with deallocating 'Multiple' command and vice versa – "**command** – Warning, allocate/deallocate mismatch, will skip this deallocation"
  Where "**command**" is the exact same string that is passed to the function.

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

You are free to implement another function/functions that are used internally by the **executeCommand()** and **printResult().**

Each command buffer INDEX will always be in the range **[0-9]** inclusive.

Your program is NOT allowed to leak memory on an Allocate/Deallocate action on the same INDEX.

Examples

Input	Output
4 Allocate Single 3 Deallocate Single 3 Allocate Multiple 1 Deallocate Multiple 1	Allocate Single 3 - success Deallocate Single 3 - success Allocate Multiple 1 - success Deallocate Multiple 1 - success
4 Allocate Single 1 Deallocate Multiple 3 Allocate Multiple 2 Deallocate Multiple 1	Allocate Single 1 - success Deallocate Multiple 3 - system crash prevented, will skip this deallocation Allocate Multiple 2 - success Deallocate Multiple 1 - Warning allocate/deallocate mismatch, will skip this deallocation
5 Allocate Single 1 Allocate Multiple 1 Deallocate Single 1 Allocate Multiple 1 Deallocate Multiple 1	Allocate Single 1 - success Allocate Multiple 1 - memory leak prevented, will not make allocation Deallocate Single 1 - success Allocate Multiple 1 - success Deallocate Multiple 1 - success

8. Snake

Your task is to write a program that implements the famous **Snake** game. Most of the game is already coded.
Your only role is to **provide an implementation for the Snake class**.

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

- The field (simple 2D array of chars) size will be read from the console. Field size is represented by ROWS and COLS variables.
- Starting position of the snake (startRowIndex and startColIndex)
- Number of following obstacles **N**
- **N** rows of obstacles (each containing obstacleRow and obstacleCol)
- Number of following powerUps **M**
- **M** rows of powerUps (each containing powerUpRow and powerUpCol)

The next part of the input consists of commands that your Game should execute:

```
enum InputCommands
{
    MOVE_SNAKE         = 0,
    GENERATE_OBSTACLE = 1,
    GENERATE_POWER_UP = 2
};
```

The GENERATE_OBSTACLE generates a new obstacle and run-time and GENERATE_POWER_UP respectively a power-up at run-time. Both those commands are already provided in the Skeleton.
You are **assured** that an obstacle or power-up will **NOT** be generated on top of your snake at run-time.
Your role is to implement the functionality behind the MOVE_SNAKE command.

The MOVE_SNAKE command takes as an argument a Direction.

```
enum class Direction
{
    UP    = 0,
    RIGHT = 1,
    DOWN  = 2,
    LEFT  = 3
};
```

Where your snake should move 1 tile up or 1 tile right or 1 tile down or 1 tile left (depending on the provided Direction).

The MOVE_SNAKE command can have different outcomes, which are described by the rules of the game (provide them as a return value of the method).

```
enum class StatusCode
{
    SNAKE_MOVING   = 0,
    SNAKE_GROWING  = 1,
    SNAKE_DEAD     = 2,

    STATUS_UNKNOWN = 255
};
```

- Return SNAKE_MOVING StatusCode - when the snake moves freely in the provided direction (which means no obstacles, powerUps, or snake body parts should be present in the designated tile).
  Additionally, this tile **must be within the field boundaries**.

- Return SNAKE_DEAD StatusCode – when the snake would make an illegal move (hit an obstacle, hit its own body part, or goes out of field bounds).
  NOTE: when making a move **first move the head** of the snake and just **after that move its remaining body parts.**
  If SNAKE_DEAD StatusCode is received – the game immediately is stopped (no further commands are executed).

- Return SNAKE_GROWING StatusCode - when the snake encounters a powerUp on the field.
  In that case, the snake should grow its body part by 1 node to its tail(the previous snake's last body position before the move).
  NOTE: every powerUp that is collected by the snake should be **removed** from the powerUps container!

On each iteration of the game – the Field is automatically printed for you.

The symbols used for markers can be found in **Defines.h**

```
enum FieldMarkerDefines
{
    EMPTY_FIELD_MARKER = '.',
    OBSTACLE_MARKER    = 'o',
    POWER_UP_MARKER    = '*',
    SNAKE_HEAD_MARKER  = '@',
    SNAKE_BODY_MARKER  = 'x'
};
```

As you can see the snake HEAD has a different print symbol than any other snake body part. This has nothing to do with the snake functionalities and only has the purpose to help you visualize the game field better.

Important reminder: You can make helper functions in your Snake.cpp file that is not part of the class.

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
3 3 1 1 0 0 2 0 0 0 0	Printing initial Field state: ... .@. ... MOVE_SNAKE in dir: UP, status: SNAKE_MOVING Printing Field: .@. ... ... MOVE_SNAKE in dir: UP, status: SNAKE_DEAD

3 3 2 2 1 1 1 2 0 2 0 0 8 0 0 1 2 2 2 2 1 0 0 0 3 0 3 0 2 0 1	Printing initial Field state: *.* .o. ..@ MOVE_SNAKE in dir: UP, status: SNAKE_MOVING Printing Field: *.* .o@ ... GENERATE_OBSTACLE at row: 2, col: 2 Printing Field: *.* .o@ ..o GENERATE_POWER_UP at row: 2, col: 1 Printing Field: *.* .o@ .*o MOVE_SNAKE in dir: UP, status: SNAKE_GROWING Printing Field: *.@ .ox .*o MOVE_SNAKE in dir: LEFT, status: SNAKE_MOVING Printing Field: *@x .o. .*o MOVE_SNAKE in dir: LEFT, status: SNAKE_GROWING Printing Field: @xx .o. .*o MOVE_SNAKE in dir: DOWN, status: SNAKE_MOVING Printing Field:

	xx. @o. .*o MOVE_SNAKE in dir: RIGHT, status: SNAKE_DEAD
4 4 2 3 0 3 1 3 0 2 0 0 13 0 0 0 0 0 3 0 3 0 3 0 2 0 1 1 3 2 2 3 3 0 0 0 1 0 1 0 1	Printing initial Field state: *.*. ...* ...@ .... MOVE_SNAKE in dir: UP, status: SNAKE_GROWING Printing Field: *.*. ...@ ...x .... MOVE_SNAKE in dir: UP, status: SNAKE_MOVING Printing Field: *.*@ ...x .... MOVE_SNAKE in dir: LEFT, status: SNAKE_GROWING Printing Field: *.@x

```
...x
....
....

MOVE_SNAKE in dir: LEFT, status: SNAKE_MOVING
Printing Field:
*@xx
....
....
....

MOVE_SNAKE in dir: LEFT, status: SNAKE_GROWING
Printing Field:
@xxx
....
....
....

MOVE_SNAKE in dir: DOWN, status: SNAKE_MOVING
Printing Field:
xxx.
@...
....
....

MOVE_SNAKE in dir: RIGHT, status: SNAKE_MOVING
Printing Field:
xx..
x@..
....
....

GENERATE_OBSTACLE at row: 3, col: 2
Printing Field:
xx..
x@..
....
..o.

GENERATE_POWER_UP at row: 3, col: 3
Printing Field:
xx..
x@..
```

	.... ..o* MOVE_SNAKE in dir: UP, status: SNAKE_DEAD