# 10. Exercises on Rule of Three, Inheritance and Polymorphism

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

Submit your solutions here: <https://judge.softuni.bg/Contests/1290/10-Exercises-on-Rule-of-Three-Inheritance-and-Polymorphism>

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 – MaxSumArray

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

|  |  |
| --- | --- |
| Example Input | Expected Output |
| 3  4 1 -2 3 4  1 505  2 13 42 | 505 |

**Task 2 – IndexedSet**

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 chose 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 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 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**

|  |  |
| --- | --- |
| Example Input | Expected 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 |

**Task 3 – Extractor**

You are given code that reads a line from the console and extracts certain items from it. The provided code uses an Extractor class that selects the items to extract. There are 3 types of extractors:

* **digits** – extracts each digit from the string as a separate item
* **numbers** – extracts sequences of digits
* **quotes** – extracts sequences of symbols between two quote (") marks

Your task is to implement the necessary **Extractor** classes and initialization logic.

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**

All quotes will be “closed”, i.e. there will always be an even number of " symbols in the input.  
There will be no “negative numbers” to extract.

**Examples**

|  |  |
| --- | --- |
| Example Input | Expected Output |
| hello 123 "bye" 4 bye  digits | 1  2  3  4 |
| hello 123 "bye" 4 bye  numbers | 123  4 |
| hello 123 "bye" 4 bye  quotes | bye |

**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 |