Abstract Classes - 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. Filters

You are given code that reads a line from the console, then reads the definition of a filter, applies the filter (which removes some of the symbols from the input), and prints the output. The filter types are:

- **`A-Z`** – removes all capital letters
- **`a-z`** – removes all non-capital letters
- **`0-9`** – removes all digits

However, the code only defines the **`main()`** function and a base **`Filter`** class. Your task is to implement the specific filter in each case.

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.

Examples

Input	Output
`Where Eagles Dare` `A-Z`	`here agles are`
`don't ever go into a ALL CAPS RAAAAGE` `a-z`	`'  ALL CAPS RAAAAGE`
`one 2 three` `0-9`	`one  three`

2. IDs

You are given code that reads information about Company objects from the console, maps and sorts them by id, then prints info about each on the console.

The provided code only contains the implementation, without the declarations for the classes & members. Your task is to study the provided code and create files with the appropriate declarations so that the code compiles and accomplishes the task described.

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.

Examples

Input	Output
`42 uni (I.K.,S.N.)`	`13 joro (G.G.)`

13 joro (G.G.) end	42 uni (I.K.,S.N.)
188 icyha (B.Q.,H.P.,F.S.) 58 uadel (S.A.,C.H.,L.T.) end	58 uadel (S.A.,C.H.,L.T.) 188 icyha (B.Q.,H.P.,F.S.)
13 joro (G.G.) end	13 joro (G.G.)

3. Files

You are given code that reads information **File** and **Directory** objects in a file system, each of which has an id, and each of which has a parent – the object which contains it.

NOTE: For this task, only files will have parents (but similar logic will be used in other tasks in this homework assignment).

Operations with the files and directories are done with reference to their **id**, and there are several types of operations:

- **file** – create a **File** object with a **filename** and **contents** (a sequence of characters, stored in a **string**)
- **directory** – create a Directory object with a name
- **copy** – move a file into a directory, only 1 such operation will be done for any file, and it will always contain a file id to move and a directory id to move to
- **size** – prints the **size in bytes** of a **File** or **Directory**. The size of a **File** is equal to the size (number of characters) of its **contents**. The size of **Directory** is equal to the **sum of the sizes** of the **File** objects in it.
- **path** – prints the path of a **File** or **Directory**. The path is the sequence of parents for the **File**/**Directory**, separated by **"/"**, followed by the name of the **File**/**Directory**. For this task, only **File**s will have parents, meaning that paths for a directory will always be just its name.
- **print** – prints the contents of a **File** – will only be called with ids of already existing **File**s

The provided code is missing the definitions for the **File** and **Directory** classes – you should implement them. Also, you should study the code and see what inheritance hierarchy is used to represent the file system and implement any other necessary classes/interfaces and functions.

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

The input will always contain correct operations – i.e. any object used by an operation will have already been created by the file or directory operations. There will be no invalid ids, and no **copy** or **print** operations on **Directory** objects (but the destination of a copy operation will always be a **Directory**).

The provided code handles input/output and operation management – you should focus on implementing the classes it uses.

Examples

Input	Output
file 1 example.txt some example bytes as text file 2 otherFile.txt other text	example.txt examples examples/example.txt

path 1 directory 3 examples path 3 copy 1 3 path 1 print 1 path 2 end	some example bytes as text otherFile.txt
file 1 example.txt some example bytes as text file 2 otherFile.txt other text size 1 size 2 directory 3 examples size 3 copy 1 3 size 3 copy 2 3 size 3 end	26 10 0 26 36

4. 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

Input	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

5. 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**
  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**)
  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**, **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`

6. Tree

Like in **Task 3**, you are given code that reads information **`File`** and **`Directory`** objects in a file system, each of which has an id, and each of which has a parent – the object which contains it.

Operations with the files and directories are done with reference to their **id**, and there are several types of operations:

- **`file`** – create a **`File`** object with a **`filename`** and **contents** (a sequence of characters, stored in a **`string`**)
- **`directory`** – create a Directory object with a name
- **`move`** – move an object (File or Directory) into a Directory
- **`shortcut`** – creates a "shortcut" to a file or directory. Shortcuts do not move the object (i.e. the object remains in the directory it was originally, but it also appears in the shortcuts)

The provided code is missing the definitions for the **`File`** and **`Directory`** classes – you should implement them.

Note: there are some minor changes to the requirements for file system objects (getters and range-based for loop usability for **`FileSystemObjectsContainer`**).

Your task (in addition to implementing **`File`** and **`Directory`**) is to implement a "tree view" for the file system entered on the input.

A tree view is a layered representation of hierarchical objects. Objects on the **first level are printed without indentation**. Objects on the **second level** (i.e. objects contained inside directories from the first level) are printed with **1 level of indentation** after their **parent** objects. And so on – **objects on each following level are printed with an additional level of indentation, compared to their parents**, and are printed **on the line after their parents**. Additionally, objects on **each level** should be **sorted lexicographically** (using **`operator<`** of the **`string`** class). The shortcuts are printed as if they were a directory named **`[shortcuts]`**.

For this task, one level of indentation should be represented by the string **`"--->"`** (three dashes and a "greater than" sign, i.e. an arrow). See the examples below for more details on how to represent the tree view.

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

The input will always contain correct operations – i.e. any object used by an operation will have already been created by the file or directory operations. There will be no invalid or duplicate ids, and no **`move`**/**`shortcut`** operations referencing ids not yet created. No object (**`File`** or **`Directory`**) will have the same name as another object.

The provided code handles input/output and operation management – you should focus on implementing the classes it uses and on implementing the construction of the tree view.

Examples

Input	Output

`file 1 example.txt some example bytes as text`
`file 2 otherFile.txt other text`
`directory 3 examples`
`move 1 3`
`directory 4 nested`
`move 4 3`
`move 2 4`
`file 5 rootFile.txt this file is in the file system root`
`directory 6 rootDir`
`end`
	`examples`
	`--->example.txt`
	`--->nested`
	`--->--->otherFile.txt`
	`rootDir`
	`rootFile.txt`

`file 1 example.txt some example bytes as text`
`file 2 otherFile.txt other text`
`shortcut 2`
`directory 3 examples`
`move 1 3`
`directory 4 nested`
`shortcut 4`
`move 4 3`
`move 2 4`
`file 5 rootFile.txt this file is in the file system root`
`directory 6 rootDir`
`file 7 noDot can't use name to check if directory or file :)`
`move 7 4`
	`[shortcuts]`
	`--->nested`
	`--->--->noDot`
	`--->--->otherFile.txt`
	`--->otherFile.txt`
	`--->rootDir`
	`examples`
	`--->example.txt`
	`--->nested`
	`--->--->noDot`
	`--->--->otherFile.txt`
	`rootDir`
	`rootFile.txt`

shortcut 6 end	

7. Explorer

Like in **Task 3** and **Task 4** you are given code that reads information **`File`** and **`Directory`** objects in a file system, however, it uses an **`Explorer`** class to create, cut & paste, create shortcuts and navigate between them.

You are tasked with implementing the **`Explorer`** class so that it supports the operations below. After all operations by the explorer are done, the tree view logic from Task 4 is used to print the resulting file system.

The **`Explorer`** supports the following operations:

- **`mf`** – create a **`File`** object with a **`filename`** and **contents** (a sequence of characters, stored in a **`string`**), in the current directory
- **`md`** – create a **`Directory`** object with a name, in the current directory
- **`cut`** – prepare and object from the current directory to be moved. Can be called multiple times and each time it adds an object to be moved to a "clipboard"
- **`paste`** – moves the objects from the clipboard to the current directory. The object is removed from its current parent and placed in the current directory (shortcuts to the object remain unchanged)
- **`sc`** – creates a "shortcut" to a file or directory. Shortcuts do not move the object (i.e. the object remains in the directory it was originally, but it also appears in the shortcuts). Shortcuts are listed the same way as in Task 4, and no other operations will access shortcuts (i.e. no navigation to them, no copying, etc.)
- **`cd`** – changes the current directory. Receives a single **`path`** parameter, which indicates the **name of a directory**, inside the **current directory**, to which to navigate, or the string **`".."`**, which indicates **the parent of the current directory** (NOTE: this is like the DOS **`cd`** command, however, you do not need to implement complex path parsing)

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

The input will always contain correct operations – i.e. any object used by an operation will have already been created. There will be no invalid or duplicate names, and no invalid **`cd`**/**`cut`**/**`sc`** operations.

The provided code handles input/output and operation management – you should focus on implementing the classes it uses and on implementing the **`Explorer`** class.

Examples

Input	Output
mf example.txt some example bytes as text md examples cut example.txt cd examples paste md nested cd nested mf otherFile.txt other text cd .. cd .. md rootDir	examples --->example.txt --->nested --->--->otherFile.txt rootDir rootFile.txt

mf rootFile.txt this file is in the file system root end	
mf example.txt some example bytes as text md examples cut example.txt cd examples paste md nested sc nested cd nested mf otherFile.txt other text sc otherFile.txt mf rootFile.txt this file is in the file system root cut rootFile.txt cd .. cd .. md rootDir paste sc rootDir mf noDot can't use name to check if directory or file :) cut noDot cd examples cd nested paste end	[shortcuts] --->nested --->--->noDot --->--->otherFile.txt --->otherFile.txt --->rootDir examples --->example.txt --->nested --->--->noDot --->--->otherFile.txt rootDir rootFile.txt

8. Warcraft IV

You are given 7 files: main.cpp, Defines.h, Structs.h, Hero.h, Archmage.h, DeathKnight.h and DrawRanger.h.

The classes 'Archmage', 'DeathKnight' and 'DrawRanger' represents your 3 heroes.

Each hero has the following attributes:

- name – the name of the character;
- maxMaxa – the character mana pool for casting spells. (If you don't know what 'mana' is – think of it as a currency required to cast a spell).
- baseManaRegenRate – tell you how much mana points your character restores when an ActionType::REGENERATE_MANA is performed. Keep in mind that your character can **NOT** have more mana points than his "maxMaxa". Your character can restore mana points **UP** to his "maxMaxa".

NOTE: The **Archmage** class has a special bonus attribute: "manaRegenModifier", which scales up his mana regeneration (multiplies baseManaRegenRate to manaRegenModifier) each time the character performs an ActionType::REGENERATE_MANA.

Each character has his unique BASIC and ULTIMATE spells that are already predefined.

```
struct Spell {
	std::string name;    //name of the spell
	int         manaCost; //mana requirement to cast this spell
};
```

You are given the **`main()`** function, which first populates Archmange, DeathKnight, and DrawRanger classes constructors and then reads a single integer value of memory (N).

- The next N whitespace separated integer are special ActionType commands;

```
enum ActionType {
        CAST_BASIC_SPELL,
        CAST_ULTIMATE_SPELL,
        REGENERATE_MANA
};
```

  - "0" or ActionType::CAST_BASIC_SPELL command – all heroes should **TRY** to casts their BASIS spells (if they have enough mana points);
  - "1" or ActionType::CAST_ ULTIMATE _SPELL command – all heroes should **TRY** to cast their ULTIMATE spells (if they have enough mana points);
  - "2" or ActionType::REGENERATE_MANA command – all heroes should use their ability to regenerate mana;

Your task is to study the provided Skeleton and implement the missing functionalities for Archmage.cpp, DeathKnight.cpp and DrawRanger.cpp files with a few things in mind:

After each ActionType::CAST_BASIC_SPELL or ActionType::CAST_ULTIMATE_SPELL each hero should print to the console a result of his actions.

- For a successful cast you should print: 'spell name' casted for 'spell mana' followed by a **newline**.
- For unsuccessful cast you should print: 'spell name' – not enough mana to cast 'spell name' followed by a **newline**.

Note: ActionType::REGENERATE_MANA does **NOT** print any result to the console.

Special hero abilities:

- Archmage – if SpellType::ULTIMATE is successfully casted the Archmage gets **immediately** a free ActionType::REGENERATE_MANA.
- DeathKnight – if SpellType::ULTIMATE is successfully casted the DeathKnight gets **immediately** a free ActionType::CAST_BASIC_SPELL. **Important note**: on the free basic cast spell you should print to the console – 'spell name' casted for 0 mana.
- DrawRanger – if SpellType::BASIC is successfully casted the DrawRanger gets **immediately** a free ActionType::CAST_BASIC_SPELL. **Important note**: on the free basic cast spell you should print to the console – 'spell name' casted for 0 mana.

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

All heroes **at any time** can have mana points from **[0, individual 'maxMana']** inclusively;

Examples

Input	Output
`Archmage 480 80 2`	`Archmage casted Water Elemental for 120 mana`

DeathKnight 420 70 DrawRanger 360 60 0 1	DeathKnight casted Death Coil for 75 mana DrawRanger casted Silence for 90 mana DrawRanger casted Silence for 0 mana Archmage casted Mass Teleport for 180 mana DeathKnight casted Animate Dead for 200 mana DeathKnight casted Death Coil for 0 mana DrawRanger casted Charm for 150 mana
Values 180 50 3 Are-not 220 80 Hardcoded 160 90 1 1	Values casted Mass Teleport for 180 mana Are-not casted Animate Dead for 200 mana Are-not casted Death Coil for 0 mana Hardcoded casted Charm for 150 mana Values - not enough mana to cast Mass Teleport Are-not - not enough mana to cast Animate Dead Hardcoded - not enough mana to cast Charm
ConjurusRex 280 80 1 Arthas 320 40 Sylvanas 160 50 1 2 2 1	ConjurusRex casted Mass Teleport for 180 mana Arthas casted Animate Dead for 200 mana Arthas casted Death Coil for 0 mana Sylvanas casted Charm for 150 mana ConjurusRex casted Mass Teleport for 180 mana Arthas casted Animate Dead for 200 mana Arthas casted Death Coil for 0 mana Sylvanas - not enough mana to cast Charm