Problem Description

With the help of STL, we can create many things. This time, let's create a simple text editor called `TextEditor`. Its behavior is similar to typical text editors like Notepad on Windows, VSCode, etc. If you have any confusion, you can try these editors for reference, but **everything should be based on the problem description**.

Basic Structure of `TextEditor`

- Text: The object being edited. The text can be loaded into `TextEditor` for operations or output by `TextEditor`. Each character's position is determined by its row and column numbers: `(row, col)`, with everything starting from 0.
  - For a file with n lines, $\texttt{row} \in [0, n)$ (note that it's a half-open interval);
  - For a line with k characters, $\texttt{col} \in [0, k]$ (note that both ends are closed intervals). $\texttt{col} = 0$ indicates the position of the first character, and $\texttt{col} = k$ indicates the position of the last newline character in that line.

  For example:

  ```
  abc
  hello
  ```

  $(0, 0)$ is the position of the **character 'a'**, and $(1, 5)$ is the position **after 'o'** (the newline character).
- Cursor: Indicates the position being edited, also determined by row and column numbers, i.e., `(row, col)`, representing the position **before** the character at `(row, col)`. For example:

  ```
  abc
  h*ello
  ```

  The cursor is at $(1, 1)$.

Functions to be Implemented in `TextEditor`

- Create Editor:
  - Constructor (`TextEditor`): Create the editor. Initially, the text is empty (i.e., no text, represented as a blank line when printed), the cursor is at $(0, 0)$ (i.e., at the beginning of this blank line), and it's in a non-selected state.
- Cursor Operations:
  - Move Cursor (`MOVE`): Accepts a position `dest`, moving the cursor to that position (you can assume moving the row first, then the column). If the position is out of the current data range, move the cursor to the closest boundary to `dest`. For example:

    ```
    // Initial
    ```

```
*abc
hello
// MOVE 1 4
abc
hell*o
// MOVE 5 5
abc
hello*
// MOVE -5 10
abc*
hello
// MOVE 5 2
abc
he*llo
```

- Text Editing:
  - Write (`WRITE`): Accepts a string `data` and inserts it at the cursor position. After writing, the cursor moves to the end of the inserted content. For example:

    ```
    // Initial
    *abc
    // WRITE hello
    hello*abc
    ```

  - New Line (`NEWLINE`): Inserts a new line below the current cursor line, moving the characters after the cursor to the new line. After inserting, the cursor is at the beginning of the new line. For example:

    ```
    // Initial
    abc*
    // NEWLINE
    abc
    *
    ```

  - Delete (`DELETE`): Deletes the character before the cursor, with the cursor position remaining unchanged. If the cursor is at the beginning of a line, it will delete the newline character of the previous line and merge the current line with the previous one. If the cursor is at $(0, 0)$, **no operation** will be performed. For example:

    ```
    // Initial
    a*bc
    // DELETE
    *bc
    ```

```
// Initial
ab
c
*d
// DELETE
ab
c*d
```

- Undo (`UNDO`): Undoes the most recent text editing operation, reverting **both text and cursor** to their state before the last editing operation. Note: `UNDO` can be used multiple times to undo multiple steps, so you may need to keep track of the entire history of the text editor. For example:

```
// Initial
*abc
// WRITE hello
hello*abc
// MOVE 0 2
he*lloabc
// UNDO
*abc
```

- View Text:
  - Screenshot (`SCREEN`): Outputs the current text in the editor line by line to the standard output, marking the cursor position with `*`.

    Output format: first output `### SCREENSHOT BEGIN ###`, then output the current text in the editor line by line. Finally, output `### SCREENSHOT END ###`.

    For example:

    When the cursor is at $(0, 2)$:

```
### SCREENSHOT BEGIN ###
ab*c
hello
### SCREENSHOT END ###
```

The initial state should output:

```
### SCREENSHOT BEGIN ###
*
### SCREENSHOT END ###
```

Input and Output Format

The first line of the input contains a positive integer `m`, indicating the number of operations;

The next `m` lines contain a string `op`, representing one of the aforementioned functions:

- For the cursor move operation `MOVE`: input two positive integers `row` and `col`, indicating the position to move the cursor, ensuring the cursor position is valid;
- For the write operation `WRITE`: input a string `text`, representing the string to be written, ensuring the string is non-empty, contains no newline characters, and no `*`;
- For other operations: no additional input is needed.

For each screenshot operation, output the current text in the text editor as described above.

Data Scale and Constraints

- Test Points 1-2: Only creation of editor, `MOVE`, `WRITE`, and `SCREEN` operations.
- Test Points 3-4: Only creation of editor, `MOVE`, `WRITE`, `DELETE`, and `SCREEN` operations.
- Test Points 5-6: Only creation of editor, `MOVE`, `WRITE`, `NEWLINE`, `DELETE`, and `SCREEN` operations.
- Test Points 7-10: All operations are tested comprehensively.
- For 100% of the data, $\mathtt{m} \leq 200$, with each new string length not exceeding 20.
- Ensure that the input for `WRITE` operation consists of combinations of numbers, uppercase, and lowercase letters.
- This problem does not require optimization of algorithm time or space complexity. A straightforward implementation can pass the evaluation.

Sample

See the files `exp1.in` to `exp4.in` and `exp1.out` to `exp4.out` provided in the downloaded files together with this problem description.

Requirements

1. You cannot modify `main.cpp` and `Makefile`, even if you modify them, they will be overwritten :(
2. The download file provides some existing code for your reference, but you are free to choose whether to use it. These files also contain some helpful instructions. Please further develop `editor.h` and `editor.cpp`.
3. File download: the required files are downloaded together with this problem description.

Submission Format

Please package `editor.h` and `editor.cpp` into a zip format and upload. **Note: Your files should be in the root directory of the zip, not in a subfolder. In other words, after unzipping your submission, the cpp and h files should be directly available, not in a subfolder. The OJ will compile and execute the provided files by pasting them into your directory.**