CS 15 Project 2: `typewriter`

"Half a dozen monkeys provided with typewriters would, in a few eternities, produce all the books in the British Museum. Strictly speaking, one immortal monkey would be sufficient."

- Jorge Luis Borges, *The Total Library*

Contents

Introduction

Preamble

Step aside, `nano`. There's a newer, simpler editor in town.

For this project you will implement `typewriter`, a text editor that runs within your terminal! `typewriter` supports a lot of the functionality you would expect from a basic editor. Crucially, it will also support the ability to undo and redo actions, which you will implement using two stacks. But we'll get in to all of these details soon.

Do not try to do this project in one sitting.

Read it right away. Jot down and draw ideas. It's better to work on it in 90 minute or 2 hour chunks of time. If you do that, and you're organized, it will go smoothly. The stack implementation itself should be doable in a single sitting. Be sure to note the multiple phases of the project.

Running `typewriter`

There are two ways to run `typewriter` from your terminal. First:

```
./typewriter textFile
```

If `textFile` is an ASCII file that already exists, it will be opened and its contents will be displayed for the user to edit. If `textFile` does not exist, then `typewriter` will display an empty document for the user to edit; later, if the user chooses to save their file, it will be saved to a file named `textFile`.

The only other way to run `typewriter` is with one additional argument naming a log file:

```
./typewriter textFile logFile
```

When this additional argument is used, `typewriter` is run in *log mode*. We will get into the details of log mode later—it will be very useful when testing your program.

`typewriter` makes two assumptions:

1. If the given `textFile` *does* exist, then it will be a file that the user has permission to read from and write to.
2. Any files edited in `typewriter` will only contain ASCII characters. Non-ASCII characters are not supported.

If the user attempts to run `typewriter` in any other way (i.e., with more or fewer arguments), the following message should be printed to `cerr` (ending with a newline), and your program should return with exit code 1:

```
Usage: ./typewriter textFile [logFile]
```

We have provided a reference implementation along with the starter files. To get a feel for how things work, you should play around extensively with this implementation before writing any code of your own.

Starter Files

Run the following command on the server to copy the starter files:

```
/comp/15m1/files/proj_typewriter/setup
```

Note that you should **not** run `cp`.

We discuss the starter files you receive in greater detail later in the spec, but for now, notice the following files:

- `the_typewriter`: this is our reference implementation.
- `mindfulness.text`, `maggiesfarm.text`, and `threelonglines.text`: these are ASCII text files that we are providing to you for testing purposes. Eventually, you'll want to write your own test files as well.

You now have enough to information to run the reference implementation on some existing text files. Try it out! Press `Escape` followed by `x` when you are ready to exit the program (see Inputs to typewriter for a full description of `typewriter`'s inputs).

Program Structure

User Interface and the `TextUI` Class

`typewriter` will work a bit differently from the other programs you've written for this course. That's because it features a *dynamic, interactive* text user interface.

In MetroSim, user interaction was sequential: the user typed some input and hit "return" to send it to `cin`, at which point the updated state of the simulation was printed to `cout`. This process would repeat until the program was terminated.

With `typewriter`, however, the program will respond to a *single* key press, and the terminal screen will be updated *in place*, rather than sending a whole new output to `cout`. This gives `typewriter` the seamless, dynamic feel that we expect from our text editors.

To help you achieve this, we've provided you with the interface and implementation for a class named `TextUI`. This class includes a number of public functions that you may want to use—you should take a look at the header file to familiarize yourself with them—but there are at least three functions that you will definitely need:

- ```
  void render(const std::vector<std::string> &lines,
              const int &cursorCol, const int &cursorLine)
  ```
  ```

The `render` function tells the interface what to display on the screen. It takes three arguments. The first is a `vector<string>` named `lines`. Each element of this vector is a line of the text file that is being edited. Of course, if the text file is very large, not all lines can be displayed on the screen simultaneously. Moreover, if an individual line is longer than the width of your terminal, then it must wrap around to the subsequent line. You don't need to worry about these details—the TextUI class has some internal logic that determines exactly how to display the lines of text. All you need to do is provide all lines in the document at a given time as a `vector<string>`.

In addition to the text, the interface must also display the cursor, which highlights the exact position within the text that is being edited. The second and third arguments to the `render` function tell the interface the position of the cursor. `cursorCol` is the column that the cursor is currently located, and `cursorLine` is the line it is located (i.e., the index in the vector `lines`). In a sense, these are like the `x` and `y` coordinates for the cursor location.

Every time `typewriter`'s screen must be updated, you should call the `render` function to do so, providing the new state of the text and location of the cursor with each call.

- `int getChar()`
  The `getChar` function is what you should use to receive input from the user. We no longer get input from `cin`, as we don't want the user to hit the "return" key for each character that they press. Instead, call `getChar` each time you want to listen for a single key press from the user. For most inputs, it returns the ASCII number corresponding to the key that was pressed. You can find more info in the Inputs to `typewriter` section later in the spec.

- `void startLogMode(std::string logfile)`
  When called, this function will put the interface in "log mode," where extra details are saved to a file named by the argument `logfile`. We discuss log mode in the Log Mode section later in the spec. You should call this function when the user provides two command line arguments, one of which is the name of the log file (see Running `typewriter` in the Introduction).

- `void close()`
  This function closes the text user interface. Call it when you are done with the interface.

The high-level structure of user interactions in your program will probably look similar to MetroSim: you will have a "command loop" that receives input then displays output. But now instead of using `cin`/`cout`, you should use `TextUI`'s `getChar` and `render` for your input/output.

Files

Your final program should include *at least* the files listed below.

- `TextUI.h` and `TextUI.o`: these are provided for you with the starter files. **You should not edit them.** As described above, `TextUI` is the class responsible for generating the text editor's user interface. Notice that we give you the pre-compiled `.o` file for the implementation, *not* the source `.cpp` file. We will not provide you with the implementation source code. Everything you need to know about this class is contained within the interface in `TextUI.h`—inspect this interface closely to understand what is available to you.

- `Editor.h` and `Editor.cpp`: the interface and implementation for the `Editor` class. This class contains the bulk of the functionality for your editor, including the logic for processing key presses. We describe it in greater detail in section `Editor` Interface below.

- `main.cpp`: this file contains your `main` function, which receives any command line arguments and kicks things off by creating and running an instance of the `Editor` class.
- `ActionStack.h` and `ActionStack.cpp`: the interface and implementation for the `ActionStack` class. This is a classic stack data structure, which will be necessary for implementing the editor's undo and redo functionality. We provide you with a *partially* complete `ActionStack.h` with the starter files. You will be responsible for completing this header file, and writing the full `.cpp` file. We give more detail on the `ActionStack` class below.

You are welcome (but not required) to include other files/classes in your program, but at a minimum you must have the files listed above.

The `ActionStack` Class

Introduction

The `ActionStack` class implements the standard functionality of a stack. The stack will contain `Action` instances, where each `Action` captures an action performed by the user. The `ActionStack` will be necessary for implementing undo and redo functionality. Think about it: when you undo an action in an editor, the *last performed* action is undone. That is, actions are undone in a *last in, first out* order, making the stack data structure the natural choice for implementation. The same is true of redoing actions.

The `Action` struct

The `ActionStack` will store instances of the `Action` struct. This struct definition is already provided for you in the `public` portion of `ActionStack.h`. It looks as follows:

```
struct Action {
    char character;
    bool deleted;
    size_t line;
    size_t column;
};
```

An `Action` instance captures a single ASCII character that was entered or deleted by the user. The `character` member variable stores what this character was. The `deleted` member variable is `true` when `character` was deleted by the user, or `false` when `character` was entered by the user. And the `line` and `column` member variables, both of type `size_t`, store the line and column within the text where `character` was entered or deleted. (Remember, `size_t` is the data type for integers that are greater than or equal to 0.)

You will need to create instances of the `Action` struct outside of the `ActionStack` class, likely in your `Editor` class or in tests you write. Because it is part of the public portion of the class, you can do so using `::`, the scope resolution operator. For example, in any file that uses '`#include ActionStack.h`', you can create an `Action` instance as follows:

```
ActionStack::Action a;
```

`ActionStack` Interface

The public interface for `ActionStack` has been provided for you in `ActionStack.h`. It includes the following functions:

- `bool isEmpty() const`: returns `true` when the stack is empty, and `false` otherwise.
- `int size() const`: returns the number of elements currently on the stack.
- `Action top() const`: returns the top element on the stack *without* removing it. If this function is called on an empty stack, it should throw an `std::runtime_error` with the message `"empty_stack"`, not terminated by a newline.
- `void pop()`: removes the top element on the stack. If this function is called on an empty stack, it should throw an `std::runtime_error` with the message `"empty_stack"`, not terminated by a newline.
- `void push(Action elem)`: pushes the given element onto the stack.
- `void push(char c, bool was_delete, size_t line, size_t column)`: Another way of pushing an element onto the stack. It creates an `Action` instance containing the values of the given arguments, and pushes that instance onto the stack.
- `void clear()`: Removes all items from the stack. No error is thrown when this function is called on an empty stack.
- `ActionStack()` and `~ActionStack()`: the constructor and destructor. Depending on what private member variables you include in your class, you may or may not need to actually initialize or recycle anything. Even if you don't need to do anything, you should define these functions, possibly with empty bodies.

You are welcome to edit `ActionStack.h` **only** to add private member variables or functions as needed. You **should not** edit the public portion of the interface.

You are responsible for creating `ActionStack.cpp` and defining the above functions.

Note that the use of `std::stack` is strictly forbidden, as it would make the implementation of this class trivial.

The `Editor` Class

`Editor` Interface

The `Editor` class will contain most of the functionality for `typewriter`. It has a very simple public interface comprising only four functions:

- `Editor(std::string filename)`: the constructor. It takes as input the filename to be edited—either an existing file which will be opened, or a new file which will be created—which will be passed in from `main()`. It should initialize the member variables of your class.
- `Editor(std::string filename, std::string logfile)`: A second constructor. Use this when the user provides *two* command line arguments, the second of which is the log file. In addition to initializing member variables, this constructor should also call `TextUI`'s `startLogMode()` function.
- `~Editor()`: the destructor. It frees any heap-allocated memory used by your class.
- `void run()`: the main input-processing function of your editor. It receives input from the user using `TextUI`'s `getChar()` function, processes it, then updates the interface using `TextUI's render` function.

That's it! You should not add anything else to the public interface for `Editor`. You will, of course, need a number of private member variables and functions. These design details are left entirely up to you.

Inputs to `typewriter`

As stated previously, the `run()` function continuously (1) receives an input using `getChar()`, (2) processes it in some way, and (3) calls the `render()` function passing it the updated state of the edited file, which comprises: a vector of lines in the file, and the location of the cursor as captured by its column and line numbers. For most inputs, the `getChar()` function returns the ASCII number corresponding to the key that was pressed. You can compare its output to the ASCII codes from the ASCII table (e.g., the char 'A' has code 65, "escape" has code 27, etc.).

However, there are also a few characters that `typewriter` supports that do not have corresponding ASCII codes. For these cases, when you `#include "TextUI.h"`, you will get access to the keypad constants that are listed here. In particular, you will want to make use of the following constants:

Key	Constant
Backspace/Delete	`KEY_BACKSPACE`
Left arrow	`KEY_LEFT`
Right arrow	`KEY_RIGHT`
Down arrow	`KEY_DOWN`
Up arrow	`KEY_UP`

For example, when `getChar() == KEY_LEFT`, that means the user pressed the left arrow key.

You will also need to maintain two `ActionStacks`: an undo stack, which keeps track of actions performed by the user so that they can be undone; and a redo stack, which keeps track of actions *undone* by the user so that they can be *redone*. Processing inputs will require updating these stacks. We won't tell you explicitly when to update the stacks, but you should use the following guidance:

- Push to the undo stack when the user performs an action that changes the text, including when the user redoes an action.
- Push to the redo stack when the user undoes an action.

- *Clear* the redo stack when the user performs an action that changes the text (*except* for undo). Think about it: the redo stack is for keeping track of undone actions that we may later want to redo. However, once the user alters the text in a new way, the actions stored on the redo stack are no longer relevant to the updated state of the text, so we clear the stack.

With all that said, these are the inputs that your editor must support:

- **The printable ASCII characters.** These characters are named "printable" because they can be displayed on screen, and they correspond to ASCII numbers 32 through 126. They include lower and upper case letters, the digits 0-9, and punctuation marks, among other symbols.

  When the user enters a printable ASCII character, you should insert it into the text at the cursor's location, and update the cursor's location. Note that the cursor always remains one position ahead of the last entered character (try this out in the reference implementation).

- **The left and right arrow keys.** Pressing these keys does not change the state of the text in any way. They only update the location of the cursor. You should consider various edge cases when implementing this functionality. Try using the reference implementation if ever you are unsure as to what behavior should occur.

- **The up and down arrow keys.** These keys also change the position of the cursor, not the state of the text. We list these separately from the left and right arrow keys, because implementing up and down is a fair bit trickier. As such, you may wish to save implementing up and down for after you have implemented some other functionality. Make sure to consider various cases here, including: when the current line the cursor is on is long enough to wrap around the terminal, and when the line you are moving too is long enough to wrap. See how the reference behaves in these cases. As a hint: you will need to use other public functions in the `TextUI` class for getting the user's terminal width.

- **The backspace key.** If a character exists prior to the cursor's current location, pressing the backspace key will delete this character and update the cursor's location Once again, think about edge cases! And consult the reference implementation when you are unsure.

- **The line feed (return) character (i.e., '\n' or "newline").** You know the drill: insert a new line at the cursor's location and update the cursor location Note that you **should not** insert a literal '\n' character into the text; rather, you must think about how to update the vector of lines that you pass to the `render` function. Once again, make sure to think through edge cases.

  Note that we do not require you to handle carriage returns (i.e., '\r'), which are typically only used on Windows machines. Because you are doing all work on the Unix-based CS department servers, this is not a concern.

- **The escape character.** When the user presses escape, your program should enter into "command mode." In this mode, the editor will wait for a *second* character input from the user, which lets `typewriter` know which command to carry out. We list the commands that `typewriter` supports in the next section.

If the user enters any input other than those listed above, the program state will remain unchanged and `typewriter` will await the next input.

Command mode

Once in command mode, the user can enter the following characters:

- **'s'**: the "save" command. When entered, the current state of the file is saved. Then, the user can continue entering inputs as normal. When the file is saved, you should also call `TextUI`'s `displaySaveMessage()` function, which will briefly display the message "Saved!" to the user.
- **'x'**: the "quit" command. When entered, use `TextUI`'s `savePrompt()` function, which asks the user if they wish to save, and returns `true` if they do or false if not. If they do wish to save, go ahead and do so.

  Either way, you should then close the interface using `TextUI`'s `close()` function, and `typewriter` should end. You should **not** call the `render()` function after the user enters the quit command.
- **'u'**: the "undo" command. When entered, the `undo` stack you've been maintaining is used to undo the last series of key presses. If the last action was the entering of a character, that character is deleted. If it was the deleting of a character, that character is re-entered. Even if your undo stack is empty, your program should not throw an error—it should continue to run normally.

  Note: Each element of the `ActionStack` corresponds to a single key press. However, typically when the user enters `undo`, `typewriter` should undo *multiple* key presses. We will not tell you the algorithm determining how many key presses to undo—this is for you to figure out! You should play around extensively with the reference implementation to figure out exactly how much gets undone for each key press.
- **'r'**: the "redo" command. When entered, the `redo` stack is used to redo the last series of undone actions. As with undo, your program should not throw an error in the case that the redo stack is empty. Once again, we will not tell you exactly how many actions from the stack should be redone. Use the reference implementation to determine this.

If the user enters any other character while in command mode, then command mode is exited and the user goes back to entering inputs in the standard way.

Do not attempt to implement all of this functionality at once. To get up and running with the `Editor` class,

1. Implement file handling: open the text file that the user provided and read its lines into a `vector<string>`; or if the file doesn't exist, create an initially empty `vector<string>`.
2. Start implementing the `run()` function. Get the basic structure down: a loop that receives an input then updates the user interface.
3. Start implementing input handling. First, just implement "save file" functionality, so that you have an output file to use in testing.
4. Next, implement handling of the printable ASCII characters, which should be easier.
5. From here, you have some choices. Choose the next form of input to handle: backspace, new lines, arrow keys, etc.

You should add a small bit of functionality, then test the new functionality out. Then repeat. As long as you have a clear design and you work in small stages, everything will be doable.

Testing

Testing `typewriter` will be more involved than the testing you've done for previous assignments, for two main reasons. First, as discussed earlier, `typewriter` relies extensively on outputting to a dynamic user interface, whereas earlier assignments exclusively output to streams like `cout`, `cerr`, and file streams. While outputs sent to streams can be `diff` tested against the reference, the interface used by `typewriter` cannot be `diff` tested.

The second challenge for testing is that, unlike your previous assignments, `typewriter` must handle non-printable characters that are input by the user. For example: the escape character enters command mode; backspace deletes characters; and the arrow keys move the cursor around. The complication for testing arises when you try to create test input files for your program. Typically, non-printable characters are not captured in the files we write. Think about it: if you press "escape" or an arrow key while writing to a file, those key presses are not actually entered into the file. This makes it challenging to create test input files that capture *all* the inputs that you want to test.

To account for these challenges in a way that allows you to still thoroughly test against the reference implementation, you can make use of two things: `TextUI`'s *log mode*, and the `keylogger` program that we provide to you.

Log Mode

The `TextUI` class that we provide to you includes a "log mode." In this mode, every time the `render()` function is called, the program state is written to a file. This way, you can `diff` test your implementation's sequence of program states against the reference implementation.

As stated earlier in the spec, the `TextUI` class includes the following function:

```
void startLogMode(std::string logFile)
```

You should call this function when the user provides two command line arguments. It will put `TextUI` in log mode. Once in to log mode, every time the `render()` function is called, the program state is written to a file named by `logFile`. The program state includes: some text capturing the current position of the cursor, followed by all of the lines of text that are passed to the `render()` function. You will likely find the resulting log file to be unwieldy and ugly to look at. Nevertheless, it will be *very* useful for `diff` testing to ensure that your implementation's behavior matches the reference's exactly.

A note on using log mode: because the full text is dumped into the `logFile` with each call to `render()`, the `logFile` can grow very quickly. For that reason, you should typically only run log mode on text files that are relatively small, and provide a smaller number of inputs. This will keep the resulting `logFile` manageable in size.

Non-Printable Characters

With log mode, you have a way to `diff` test against the reference. But you'll still need a way to save non-printable characters to a test input file that you can use for `diff` testing.

For this purpose, we have provided you with the executable program named `keylogger`, which you received with your starter files. This is a very simple program that you can use to generate test input files. When you run it (`./keylogger`), you are prompted to give the name of the file you'd like to generate. Then, you can enter as many keys as you'd like to save to the resulting file—*including* keys like the arrows, backspace, etc. You can press "escape" twice to exit the `keylogger` program, then you should have the generated test file. Run the test file through typewriter using redirection (i.e., `./typewriter textFile logFile < YOURTESTFILE`), then you can use the results to diff test against the reference!

Important Notes:

- The test files generated by `keylogger` will only work when `typewriter` is in log mode. So make sure to use log mode for testing!
- `typewriter` will automatically close when it is done processing a test input file. By default, it will close *without* saving. So, if you want to save the results of processing a test input file (presumably you do when you are testing), make sure to explicitly enter a save command in your test input file!

Using `valgrind`

As always, you are strongly encouraged to test your program using `valgrind` to uncover any memory issues. However, be aware of one quirk this time: the `TextUI` class uses the C++ library `ncurses`, which does not free all of the memory it uses by the time the program terminates. This means that when you run `valgrind`, you can expect it to report some leaked memory. Specifically, if `valgrind` reports "47,482 bytes in 331 blocks" of memory that is still reachable at the time the program terminates, you do not need to worry about it—this is internal to `ncurses`. If a larger amount of memory is leaking, or if any other memory errors are reported, then there are issues in your own code and you should do further debugging.

`cout` and `cerr`

The `TextUI` class writes directly to your terminal's output, so it can obscure, or sometimes entirely replace, outputs that you send to `cout`. Therefore, for the sake of this project, we advise avoiding the use of `cout` entirely.

If you would still like to print custom outputs (e.g., for debugging purposes), we would recommend relying on `cerr` instead. However, you may still find your `cerr` outputs are obscured by `TextUI`'s interface. So to guarantee that you see any debugging outputs, you should *redirect the stderr stream* to a file so that you can observe those outputs. You can see our guide to diff testing, or revisit the

make lab, for a reminder on how to do this.

Makefile and README

Makefile

You must write and submit a `Makefile` along with your code. When we run `make` and/or `make typewriter`, your `Makefile` should build your program and produce an executable named `typewriter` that we can run.

Two important notes on your `Makefile`:

- You must include the special flag `-lncurses` in your `LDFLAGS` variable in your `Makefile`. This tells `make` to link your program with the `ncurses`, library, a special library used by the `TextUI` class to build the user interface.
- Recall that we provide you with an *already compiled* `TextUI.o` file. This means that you **should not** have a `TextUI.o` rule in your `Makefile`, as you do not need to create this file. However, you *should* include `TextUI.o` among the object files that are linked to build your `typewriter` program.

README

In addition to your program and testing files, you should submit a `README` file that includes the following sections:

A. The title of the assignment and your name.

B. The purpose of the program.

C. Acknowledgments for any help you received.

D. A list of the files that you provided and a short description of what each file is and its purpose.

E. Instructions on how to compile and run your program.

F. An outline of the data structures and algorithms that you used. Given that this is a data structures class, you need to always discuss the **ADT** that you used and the **data structure** that you used to implement it and justify why you used it. Please discuss the features of the data structure and also include (with some justification/explanation) two other situations/circumstances/problems where you could utilize it. The **algorithm** overview is always relevant. Please pick a couple interesting/complex algorithms to discuss in the `README`

G. Details and an explanation of how you tested the various parts of assignment and the program as a whole. You should reference the testing files that you submitted to aid in your explanation.

I. Please let us know approximately how many hours you spent working on this project. This should include both phases one and two.

Each of the sections should be clearly delineated and begin with a heading that describes the content of the section.

Deliverables and Submitting

By the end of phase one (you can check the exact date on Gradescope), you must submit and conduct your design checkoff meeting, and submit your `ActionStack` class.

Deliverable #1: Design Checkoff

First, complete the required design checkoff questions given in the starter file `typewriter_design_checkoff.txt`, and submit your answers on Gradescope under the assignment "typewriter Design Checkoff." **You must submit this file prior to your design meeting.**

Then, attend your design meeting and discuss your plan. You should be prepared to discuss the answers you submitted. You are welcome to bring other materials as well, though you are not required to: drawings, pseudocode, etc.

The design checkoff helps twofold: you plan out your project and get your brain working on it in the background, and you also get design feedback before it's too late. We will check off your design, but reserve the right to not check off your design if we believe your design was not thoroughly mapped out enough.

Deliverable #2: `ActionStack` Class

Write and submit the `ActionStack` class. This involves completing `ActionStack.h`, and creating/-completing `ActionStack.cpp`. When these are ready, submit the following files to the assignment "proj_typewriter_phase1" on Gradescope:

```
ActionStack.h, ActionStack.cpp
README
(... any other files...)
```

You should only include other C++ files if your solution to `ActionStack` depends on them. Do not submit `Editor.cpp` for example. Submit these files directly, do not try to submit them in a folder. The `README` doesn't have to be the final `README`. Just document anything that you feel we should know about `ActionStack` or your project in general.

Final Submission

By the end of phase two, you must provide your final submission. For this part you will submit all of the files required to compile your `typewriter` program, including a `Makefile`. Be sure to include any files you used for testing, including test input files that you created. You only need to include testing input files, not the output files that resulted from them.

You will need to submit *at least* the following files:

```
ActionStack.h, ActionStack.cpp
Editor.h, Editor.cpp
main.cpp
unit_tests.cpp
Makefile
README
(...all the C++ .h, .cpp files that implement your program...)
(...any testing files, including those generated by keylogger...)
```

Again, if your program comprises any other files, make sure to submit them.
Notice: you **do not** need to submit `TextUI.h` or `TextUI.o`, as we have these files already.

Happy typewriting!