ICS 45C Spring 2019
Project #4: *People Just Love to Play with Words*

Due date and time: *Friday, June 7, 11:59pm*

Introduction

Among a variety of new experiences you've no doubt had in this course, one experience that has likely been new for a lot of you is spending more of your time in the text-based world of the command line. One of the things you may have noticed is that it is possible to be quite productive, even without the kinds of graphical user interfaces you may be accustomed to. When you want to edit code, there are text editors that are based entirely on text — where there are no "windows" or "menus" in the sense that you might be used to them, but where the keyboard can be used to issue a broader variety of commands, taking the place of what you might otherwise do with a mouse or touchpad.

Of course, getting used to an editor like that can be a big adjustment, and you may have ultimately decided that you want nothing more to do with editors like this, but it's good for you to have seen them and interacted with them a little bit, because there are plenty of professionals who spend the vast majority of their time working in these kinds of environments. The learning curve can be daunting, but, once conquered, the tools can actually be surprisingly powerful; I've seen people do things with **vim** that boggle my mind, in terms of how quickly they could accomplish complex edits.

If we stop and think about text editing for a minute, it's actually more complicated than it sounds. We take for granted being able to move the cursor around, press keys on our keyboard that do things like skip words, delete lines, or select text to be copied and pasted. More importantly, we take for granted the idea that if we make a misstep, there is a way to issue an "undo" command that lets us undo whatever we most recently did, and, quite often, a way to "redo" it if we issued "undo" on a change that we actually wanted to keep.

This project asks you to implement a terminal-based text editor in C++ called **BooEdit**. Ours is decidedly simpler than editors like **vim**, **emacs**, or **nano**, but it does have some of the key features we expect to see:

- A cursor displays the current position where we're editing
- The cursor can be moved around within the text using the keyboard
- Text typed into the editor is inserted at the point where we typed it
- Text can be deleted using a "backspace" key on the keyboard
- Every change made in the editor (cursor movements, changes to the text) can be undone with one keypress, and can be redone after being undone (in case you "undo" too far)

Our editor doesn't load or save the contents of files, and it doesn't have the same rich array of commands that you might expect to see in a "real" text editor, but it does form the basis of our exploration of a few key C++ and object-oriented programming concepts:

- Using classes in the C++ Standard Library, such as **std::vector**, to avoid the difficulty of manually managing memory
- Throwing and catching exceptions in C++, including the need to ensure that they are handled safely (e.g., they don't cause memory leaks)
- Using abstract base classes with multiple derived classes to allow us to customize how parts of our program are implemented in different circumstances (e.g., being able to unit test how our text editor behaves without requiring user interaction or displaying the editor on the screen)
- Implementing the well-known object-oriented **Command** pattern, which is the typical way you would implement a sequence of user interactions that can be undone and redone

Because a text editor is actually a fairly daunting project to implement from scratch, BooEdit is already partially implemented. You'll implement the parts that are interesting from the perspective of what we're learning about in this course; I've provided the rest. This will give you the opportunity to participate in a very common scenario in real-world software development: contributing to a project for which much of the code already exists. Learning how to find your way around a program you've never seen — gradually building a mental model of how it's built, then determining where you need to make changes and how — is a hugely important real-world skill that this project will give you some experience with.

Getting started

As with the previous project, you'll need to complete a couple of chores before you can start working on this project. *Be sure you read these instructions and follow them completely before proceeding.*

Refreshing your ICS 45C VM environment

Even if you previously downloaded your ICS 45C VM, you may need to refresh its environment before proceeding with this project, so that you have a copy of the **project4** project template that you'll need for this project.

Log into your VM and issue the command **ics45c version** to see what version of the ICS 45C environment you currently have stored on your VM. Note, in particular, the timestamp; if you see a version with a timestamp older than the one listed below, you'll want to refresh your environment by running the command **ics45c refresh** to download the latest one before you proceed with this project.

```
2019-05-23 16:16:39
Project #4 template added
```

Note that you can instead use the **ics45c refresh_local** technique described in the Project #1 write-up, if you're unable to make your outgoing Internet connection work from within the ICS 45C VM.

Creating your project directory on your ICS 45C VM

A project template has been created specifically for this project. It includes a **gather** script for preparing your files for submission when you're finished, as well as the usual array of scripts for running the program with and without the Memcheck tool, and a directory in which you can (optionally) write unit tests to assist in your testing.

Decide on a name for your project directory, then issue the command **ics45c start PROJECT_NAME**

project4 to create your new project using the **project4** template.

Do not use other project templates, like ***basic*** *or* ***project3****, for this project!*

The BooEdit text editor

The text editor you'll be implementing in this project is called **BooEdit**. It's a dramatically simplified version of an editor like **vim** or **emacs**, completely based around a textual interface that runs in a Linux shell window. When running, BooEdit looks like this:

```
16 |It was just a glimpse of you, like looking through a window
17 |Or a shallow sea
18 |Could you see me?
19 |And after all this time■
20 |It's like nothing else we used to know
21 |After all the hangers-on are done hanging on to the dead lights
22 |Of the afterglow
23 |I've gotta know
24 |
25 |Can we work it out?
26 |Scream and shout 'till we work it out?
    ----------------------------------------------------------------------
                                                              Ln 19 Col 24
```

At any given time, the lion's share of the space is dedicated to the text being edited, which we'll call the *editor area*. A cursor is displayed somewhere in that area — depicted above as a black square — and the location of the cursor, relative to the entire text being edited, is indicated in the bottom-right corner. Line numbers are displayed along the left-hand edge of the window. Because the text may be longer than the available space in the editor area, the text scrolls vertically and horizontally as necessary, so that the cursor will always be visible in the editor area. Occasionally, short error messages are displayed (e.g., when trying to move the cursor above the topmost line of the text); in that case, the error messages are shown just below the dashes under the last line of text (so, in the example above, they would be displayed just below the dashes underneath the words **Scream and shout**).

BooEdit always launches with no text in the editor. It supports neither loading of text files nor saving of them.

Giving input to BooEdit

The only way to provide input to BooEdit is via the keyboard. To maintain maximum compatibility with everyone's setup — you may be using any number of ways of connecting to your VM (e.g., using the ICS 45C VM's GUI in VirtualBox, via SSH through PuTTY on Windows or a Terminal window on a Mac), BooEdit is quite conservative about what keys it pays attention to, because special keys (like function keys, arrow keys, **Home**, **End**, etc.) are not handled consistently across systems. So the only keys that BooEdit officially supports are these:

- *Printable characters* in the ASCII character set (those with ASCII codes between 32 and 126), a list of which you can find many places online.

- *Control characters*, meaning the characters you get when you hold down the **Control** key on your keyboard and press a letter (A through Z).

Other keys may be ignored or may trigger other strange behavior, but this is not something we'll be concerned with in this project. (You can feel free to add support for anything else you'd like, but we'll be testing only with what's listed below.)

The following is how different keys are handled by BooEdit:

Key	***Meaning***	***Details***
Any printable character	Insert Character	Inserts a character where the cursor is currently positioned. Any characters at or beyond the cursor on the same line are shifted to the right. The cursor moves one cell to the right afterward.
Ctrl+I	Cursor Up	Moves the cursor up one cell (i.e., to the same column on the previous line). If the previous line is so short that this would place the cursor beyond the end of the previous line, the cursor is placed in the column just beyond the end of the previous line instead. If the cursor is already on line 1, there is no cell above, so this command fails and an error message is displayed.
Ctrl+K	Cursor Down	Moves the cursor down one cell (i.e., to the same column on the next line). If the previous line is so long that this would place the cursor beyond the end of the next line, the cursor is placed in the column just beyond the end of the next line instead. If the cursor is already on the last line, there is no cell below, so this command fails and an error message is displayed.
Ctrl+U	Cursor Left	Moves the cursor left one cell (i.e., to the previous column on the same line). If the cursor is at the beginning of a line already, it instead moves just beyond the end of the previous line instead. If the cursor is at the beginning of line 1, this command fails and an error message is displayed.
Ctrl+O	Cursor Right	Moves the cursor right one cell (i.e., to the next column on the same line). If the cursor is just beyond the end of a line already, it instead moves to the beginning of the next line instead. If the cursor is just beyond the end of the last line, this command fails and an error message is displayed.
Ctrl+Y	Cursor Home	Moves the cursor to the beginning of the current line. No error message is shown if the cursor is already at the beginning of the current line.
Ctrl+P	Cursor End	Moves the cursor just beyond the end of the current line. No error message is shown if the cursor is already at the end of the current line.
Ctrl+J Ctrl+M	New Line	Creates a new line just under the current line. Any text at or after the cursor on the current line is moved to the beginning of the new line. The cursor is moved to the beginning of the new line. (Note that there are two different keys that are interpreted the same way; either Ctrl+J or Ctrl+M behaves this way.)
Ctrl+H	Backspace	Deletes the character to the left of the current cursor position, sliding subsqeuent characters backward to fill the empty space. Moves the cursor one character to the left. If the cursor is already in column 1, the entire current line of text is instead moved to the end of the previous line and the cursor is placed at the beginning of that moved text. If the cursor is already on line 1 column 1, this command fails and an error message is displayed.
Ctrl+D	Delete Line	Deletes the entire current line of text, with all subsequent lines moving up to fill the empty space. The cursor remains in its current location *unless* the cursor would be beyond the end of the line of text that now occupies the cursor's line number, in which case the cursor is placed just beyond the end of that line of text instead. If there is only one line of text, it is

		cleared and the cursor is placed at line 1 column 1. (In other words, the editor must always have at least one line of text, even if it's empty.)
Ctrl+Z	Undo	Reverts the previously-executed command, putting the editor back into the state it was in before that command was executed. All commands listed above are "undoable", including cursor movements and text changes, and individual commands are undone individually (i.e., each "undo" command undoes a single cursor movement, a single character inserted, a single line deleted, etc.).
Ctrl+A	Redo	Re-executes the most-recently-undone command. Note that Undo/Redo work the way that Back and Forward buttons have traditionally worked in a web browser; once you execute a command other than Undo or Redo successfully, there are no commands available to be redone until a command is subsequently undone.
Ctrl+X	Quit/Exit	Exits BooEdit immediately.

How BooEdit works when you first start this project

When you first create your project directory from the **project4** project template, you'll already have a version of BooEdit that you can compile and run, though it will be severely limited. It will always display the text **BooEdit!** on line 1, placing the cursor on line 1 column 1. The only key that's handled is **Ctrl+X**, which allows you to exit the program; every other key is ignored.

Experimenting with a completed example implementation of BooEdit

The **project4** project template also contains a compiled version of a complete implementation of BooEdit, so you can experiment with the program as it is meant to behave when you're done. This will help you to understand its behavior, as well as help you to catch some of the nuances and edge cases that may not have been conveyed completely in the English descriptions above.

You can run the completed example implementation of BooEdit by issuing this command from a shell prompt within the project directory:

```
./run example
```

A brief tour of the project directory

When you create your project directory from the **project4** project template, you'll have a fair amount of code in the **app** and **gtest** directories. Much of it can safely be ignored; others will need to be modified. As you would any time you need to begin working on a program that's partially implemented already, you should start by gaining a basic understanding of what code is present and roughly what each part does and how the parts fit together, then focus on learning more details on an as-needed basis.

Change into the **app** directory and list its contents by issuing the **ls** shell command. You should see a number of files already present. What follows is a high-level tour of what's there.

- There is a **main.cpp** file, which contains the program's **main()** function, along with some supporting functions that do some necessary tasks, such as setting up handling for exceptions (and also for signals that the program has crashed, so it can still dismiss the user interface even when there's a crash).
- The **BooEdit** module implements a class called **BooEdit**, which implements BooEdit's outermost

layer. An instance of BooEdit is actually a combination of three things:
    - An **EditorModel**, which stores the text editor's underlying data (e.g., the lines of text, the current position of the cursor) and allows it to be manipulated.
    - An **EditorView**, which visualizes the current state of the text editor in some way, e.g., by displaying it in a Linux shell window.
    - An **InteractionReader**, which reads user interfaces, e.g., by waiting for the user to press keys on the keyboard.

  The latter two of those things are implemented as an inheritance hierarchy, with an abstract base class specifying the set of virtual member functions, and derived classes supplying the implementations. This allows us to do unit testing on BooEdit without a user interface being displayed. (This is described in more detail below.)
- The **EditorModel** module implements a class called **EditorModel**, which is responsible for maintaining and modifying the current state of the editor (the current cursor location, all of the text currently in the editor, and the error message (if any) that should be displayed). As provided, it contains a handful of member functions that are used to gather enough information to draw the screen, though they're generally implemented as stubs (i.e., their implementations are not correct, but at least compile and run). You'll need to add additional member functions to this class as you find the need, so you can retrieve and modify the state of the editor.
- The **EditorView** module provides an abstract base class for views, which implement the "look and feel" of BooEdit. There are two derived classes of the **EditorView** class provided: **NcursesEditorView** and (in your **gtest** directory) **InvisibleEditorView**. You will not need to modify either of these.
- The **InteractionReader** module provides an abstract base class for reading user interfaces. There are two derived classes of the **InteractionReader** class provided: **KeypressInteractionReader** and (in your **gtest** directory) **MockInteractionReader**. You will not need to modify **InteractionReader** or **MockInteractionReader**, though you will need to modify **KeypressInteractionReader**..
- The **Keypress** module implements a class called **Keypress**, which represents the result of a user pressing a key on the keyboard, possibly with the Ctrl key depressed.
- The **KeypressReader** module provides an abstract base class for reading keypresses. There are two derived classes of **KeypressReader** provided: **NcursesKeypressReader** and (in your **gtest** directory) **MockKeypressReader**.
- The **InteractionProcessor** module handles user interactions, waiting for the user to press relevant keys and then executing the corresponding commands. This is also where the "undo" and "redo" features are to be implemented. You'll need to make changes here to implement undo/redo and whatever else you find necessary.
- **Command.hpp** declares an abstract base class from which you'll derive classes to implement each command.
- The **BooEditLog** module provides a logging capability, described in more detail later in this write-up.

Note that there is also a **gtest** directory in your project, in which you can opt to write unit tests using Google Test if you would find it helpful. You might find unit tests of your **EditorModel** class particularly useful, because they'll allow you to nail down whether the modifications you're making to your editor are correct as you go. You may also want to write unit tests of other functionality, which you can do using the provided "mock" implementations of **InteractionReader**, **KeypressReader**, and/or **EditorView**. There is no explicit requirement that you write unit tests and we will not be grading the tests that you submit; however, you may find that writing unit tests is a useful tool in helping you to isolate issues that are otherwise difficult to diagnose in the context of the whole program, so I'm providing a **gtest** directory and a

./run gtest command that will allow you to run any unit tests you write. Additionally, there is a file called **BooEditTests.cpp** that contains a small handful of tests already, along with tools to make it easier for you to build your own.

The Command pattern

In lecture, we briefly discussed the concept of *design patterns*. Design patterns are well-understood solutions to commonly-occurring programming problems.

There is a well-known pattern that you'll need to understand for this project. Many programs take user input in the form of *commands*. There are different ways to pass commands to a program, of course. A textual user interface, like the one provided by BooEdit, accepts commands by allowing the user to type them within the terminal. A graphical user interface accepts commands from the user by allowing the user to click buttons, select menu items, drag objects around, and so on. Some programs take commands from the user by accepting command-line arguments. But regardless of the style of user interface, the program's execution can be viewed as a sequence of commands from the user, each followed by the appropriate response from the program. This is a commonly-occurring design requirement, so it makes sense that we ought to think of a common design that addresses it.

The Command pattern is exactly that common solution. In a program that accepts input commands from the user — be it commands typed into the console, mouse clicks and menu selections, or whatever — it is sensible for each command to be represented by a *command object*. Different kinds of commands are represented by different kinds of objects. Command objects know how to do two things: be *executed* (i.e., affect whatever change they're supposed to) and be *undone* (i.e., remove whatever change they made previously).

When it's time to execute a command, we can call an execute() method on the command object. So, we can define one class for each kind of command, and derive all of them from an abstract base class called Command. The Command class will have a pure virtual execute() function (we'll make it pure virtual, since we don't know how to execute a command unless we know specifically what kind of command it is), a pure virtual undo() function, and probably little or nothing else. The classes derived from Command provide appropriate overrides of the execute() and undo() functions.

There are a couple of advantages to using this pattern. First of all, the program's main loop becomes very simple indeed:

```
while the user hasn't quit the program yet
{
    read one command from the keyboard
    build the corresponding Command object to represent it
    execute the command by calling its execute() method
}
```

There is a clear separation between the code that oversees the execution of the program — you might call this the *command loop* or *event loop* — and the code that does other jobs (such as parse the input or execute the commands). Adding a new kind of command does not require touching this command loop at all, though you would obviously have to teach the "read one command" part to understand how it is to be entered as input, and you'd have to write a new subclass of Command that would know how to execute it.

Implementing undo and redo

It turns out that our program's main loop can't be quite as simple as the one described above if we want to support undo and redo features. However, the Command pattern has the big advantage that it becomes very easy to keep track of all of the commands that have been executed, by keeping them after they've been executed. Implementing an "undo" feature would require finding the most-recently executed command and calling its undo() function. So long as Command objects have the knowledge about what change they made (e.g., what character was inserted, where the cursor moved from, etc.), they will be able to undo their changes.

Note that in a program like a text editor — which might be dealing with a very large amount of text — it's not acceptable to store complete copies of the entire text being edited on every change. Instead, it becomes necessary to know just enough about what happened (e.g., what character was added and where) that we can undo the effect of a Command with the most minimal changes possible. So our Command objects have to be smart enough to do that, though it turns out that it often doesn't take a lot of smarts to do it; we just need to track some information in member variables, so it will be available later.

To implement an undo feature, every time a command is executed, its corresponding command object can be pushed on to a stack. When the undo feature is used, the top object can be popped from the stack and its undo() function can be called. To implement a corresponding "redo" feature, you could store the commands in two stacks, in much the same way that web links are stored in two stacks to implement the "back" and "forward" buttons in a web browser.

One important conceptual note here: *undo and redo are not commands!* You can execute a command and you can undo a command. Redoing a command is simply to execute it again.

What does this have to do with me?

The reason I've explained the Command pattern to you is that you'll be required to use it. I've provided an abstract base class called Command, which will form the basis of your various command implementations. Each kind of modification that the user can make in the editor — cursor movements, inserting characters, deleting lines, etc. — would be implemented as a separate class that inherits from Command. In that class, you would implement an **execute()** member function that makes the changes necessary for the command (e.g., moving the cursor or inserting a character) and an **undo()** member function that reverts the change. Redoing a command simply means calling **execute()** on the command again.

General advice on approach

You may have become quite accustomed to working on programs that you largely "own," where you've been given little or no code to start with. This project is different; some of the challenge lies in understanding what code has been provided to you, what parts of that code you'll need to understand and modify, and what parts of it you can mostly ignore. This may seem like an uncomfortable place to start, but this is intensely practical; unless you're involved at the moment of inception, work on most projects involves modifying and extending an existing code base, as opposed to building a new one from scratch.

The big question, then, is "Where do I start?" As with most projects, there are multiple reasonable paths you could follow to complete this project, but I'll give you an idea of what I might do if confronted with this problem. If you'd rather do something different, you can certainly feel free; but if you're not sure where to start, consider starting down the path that I've described, though you can certainly deviate from it if you get to a point where something different would feel more comfortable for you.

- Since the program, as provided, starts up with the bogus text **BooEdit!** — you'll want to remove this eventually, but it's handy at the beginning to have some text to work with — I'd consider starting by implementing the command to move the cursor to the right. Skip the undo feature initially and just do what you need to do to make the command work, which is roughly this:
    - Derive a class from Command to implement your command to move the cursor to the right
    - Implement its **execute()** method to call a function on the **EditorModel** object (passed to **execute()** as a parameter) that moves the cursor.
    - You'll notice that there is no function in **EditorModel** yet that moves the cursor, so you'll need to add it. Note, too, that **EditorModel** currently hard-codes the cursor's position (by always returning 1 from its **cursorLine()** and **cursorColumn()** functions); you'll need to change that, too.
    - In **KeypressInteractionReader.cpp** is a member function called **KeypressInteractionReader::nextInteraction()**, which is where keypresses are turned into user interactions. Add code to accept Ctrl+O and return an Interaction containing an object of your new command class.
    - At this point, if you've done everything correctly, you should be able to move the cursor to the right. Now add code to handle the case of getting to the end of the line by displaying an error message. (The way to display an error message is for your command's **execute()** function — or whatever function(s) it calls — to throw an **EditorException**.)
- Next, I'd probably implement the command for moving the cursor to the left, since it would be so similar to the command I'd just implemented. The steps would be the same.
- Next, I'd consider working on the undo/redo feature in **InteractionProcessor.cpp**, then implementing the **undo()** function in my two command classes. At this point, I should be able to move the cursor to the left and right *and* undo/redo those cursor movements.
- At this point, a fair amount of infrastructure would be in place, so I'd consider adding one command at a time — probably inserting characters and then handling new lines, which would allow me to add text to a line *and* have text longer than a single line, so I could continue on to moving the cursor up and down. To keep things simple, I might only support new lines when the cursor is at the end of the line, then worry about the "line-splitting" version of it later.

I'd proceed in this fashion, one command at a time. The order in which I'd implement the commands would be determined by my ability to implement and test them (e.g., I wouldn't implement moving the cursor down until I could have multiple lines of text in the editor). My goal would be to find relatively small tasks that will lead to stable ground as often as possible.

As I said, there are many perfectly good paths that lead to a completed version of this project, but the above is how I would approach it.

The BooEdit log

You may notice that BooEdit creates a file called **booedit.log** in whatever directory you ran it from. The purpose of the log file is to allow you to write any debug output you'd like, without that debug output interfering with the normal output that BooEdit displays in the shell window. This turns out to be critical, because BooEdit takes complete control of the shell window, handling cursor movements, "repainting" the screen as it changes, and so on. This makes it impossible for you to write to **std::cout** normally without causing a variety of problems.

You can write anything to the log that you'd like at any time you'd like; there is no requirement about when and what to write, and you don't have to use the log at all if you don't want to. But it provides you an easy way to generate and view debug output, which you may find to be critical to your ability to diagnose certain kinds of problems.

If you want to write to the BooEdit log from any of your source files, there are only two things you'll need to do:

- Include the file **BooEditLog.hpp** in that source file, which includes the declaration of the function **booEditLog()** that writes a message to the log.
- Call the function **booEditLog()** wherever you need it. The function takes a single **const std::string&** parameter and appends it, on its own line, into the **booedit.log** file. Each message in the log file is preceded by a timestamp that indicates when it was written — using your VM's system clock to determine the time. (Two example calls to **booEditLog()** appear in the file **main.cpp**, if you want to see what they look like.)

BooEdit, as provided, writes two messages to the log every time it runs: one when the program starts and another when the program stops. So a typical run of BooEdit might generate a log that looks something like this:

```
2019-05-19 14:50:30  Started /home/ics45c/projects/p4/out/bin/a.out.app
2019-05-19 14:50:33  Stopped /home/ics45c/projects/p4/out/bin/a.out.app
```

Anything you write will appear between those two lines in the log. After you exit BooEdit, you can open **booedit.log** to view what you wrote to it. Alternatively, you can open a separate Terminal window in your VM, change into the directory where you started BooEdit (i.e., wherever you were when you executed the **./run** script), and issue this command *after* starting BooEdit:

```
tail -f booedit.log
```

This will cause the contents of **booedit.log** to be displayed as they're added to the file, so you can see them play out "live" as they happen. When your program ends — or when you no longer want to watch this file — you can stop the **tail** program by pressing **Ctrl+C**. Note, too, that if you used **tail** to watch the file change, the file will still be there when you're done, so you can still open the file in a text editor later if you'd like.

Be aware also that the **booedit.log** file is not deleted each time the program begins or ends; every write to it simply makes it longer. You can delete the file manually if you ever want to start it over. (You can use the timestamps to narrow down the output from a single run of your program.)

A couple of notes about using Memcheck

As you no doubt discovered in your work on previous projects, Memcheck can be an invaluable tool for diagnosing not only memory leaks, but program crashes (e.g., segmentation faults) and other hard-to-diagnose memory-related issues. It does this by watching every use of memory — not only every allocation and deallocation, but also every time you read and write from memory — and is able to tell you when you've used memory that isn't allocated, read the value of a variable that hasn't yet been initialized, and so on. If you had a clean, complete implementation of previous projects, there's a good chance that you needed to use Memcheck in order to get to that point.

However, Memcheck presents us with two problems in the context of this project that we'll need to overcome in order to use it effectively.

Separating Memcheck's output from the BooEdit display

As you've seen before, Memcheck's output will be displayed in your shell window alongside your program's standard output. Anything you write to **std::cout** and anything that Memcheck writes will be intermixed. In our work so far this quarter, this has never posed much of a problem, because our use of the standard input and output has tended to be line-based (i.e., reading and writing whole lines), so that it becomes easy to distinguish Memcheck's output because of the prefix that it prints before each line of its output (e.g., **==12345==**).

But because BooEdit uses the entire shell window, including taking control of cursor movements, clearing the screen, and so on, it becomes necessary to avoid having Memcheck write its output to the shell window alongside BooEdit's output, because we'll quickly have a mess on our hands. For this reason, we'll need to find a way to separate Memcheck's output from what is displayed ordinarily by BooEdit.

We've seen previously that it is possible to save a program's standard output to a file, instead of displaying it in your shell window, by using a technique called redirection. However, that technique poses an interesting problem for us in the context of this program; the standard output can't be redirected, because it's where BooEdit is displayed! If you don't understand what I mean, try these two commands in your project directory:

```
./build
./run >test.out
```

What you'll see is that the program will appear to hang — no output will be shown in your shell window. But, interestingly, if you press **Ctrl+X**, which is the BooEdit "exit" command, you'll find that the program ends, just as it would normally. Now open the **test.out** file in an editor like **vim**:

```
vim test.out
```

and you'll see what appears to be a mess, but if you look at that mess more carefully, you'll notice that you'll really see all of the output (including a funny-looking representation of cursor movements) needed to generate BooEdit's initial, empty display. So the output really was redirected to a file, but that's not what we really want here; what we'd like is to redirect Memcheck's output to a file, while continuing to send BooEdit's output to the shell window.

The good news is that Memcheck actually doesn't send its output to the standard output (**std::cout**) at all. Instead, it writes its output to a separate stream called the *standard error* (written in C++ as **std::cerr**). To verify this, watch what happens when you do this:

```
./build
./run --memcheck >test.out
```

Notably, the Memcheck preamble is displayed in the shell window, then the program appears to hang indefinitely. But, once again, BooEdit is running in the background — it's there, but is sending its output to a file — so you can exit using **Ctrl+X**, at which time you'll see Memcheck's final report. If you open the file **test.out** in an editor, you'll once again see BooEdit's output — including cursor movements, etc. — but you won't see anything from Memcheck. This is because the **>** operator we've been using for redirection actually redirects only the standard output; the standard error still displays in the shell window.

Luckily for us, we can redirect the standard error to a file instead, without redirecting the standard output. The trick is to add a single character to our command:

```
./build
./run --memcheck 2>test.out
```

The **2** character denotes that we want to send the program's second output — its standard error — into a file, while allowing the standard output to be sent to the shell window as normal. Try that and open the **test.out** file that got created; what you'll see is Memcheck's output, without any of BooEdit's normal output. This gives you the best of both worlds: you can interact with BooEdit normally, but Memcheck's report will be available in a file when you're done. You can open that file in a text editor to see the results. Alternatively, you can open a separate Terminal window in your VM, change into your project directory, and issue this command *after* starting BooEdit:

```
tail -f test.out
```

This will cause the contents of **test.out** to be displayed as they're added to the file, so you can see them play out "live" as they happen. When your program ends — or when you no longer want to watch this file — you can stop the **tail** program by pressing **Ctrl+C**. Note, too, that if you used **tail** to watch the file change, the file will still be there when you're done, so you can still open the file in a text editor later if you'd like.

Using this technique, you'll still be able to use Memcheck in your work on this project. As before, you'll be required to deallocate any memory that you allocate, but you'll also find that Memcheck can be invaluable in diagnosing other program crashes, etc., so it's wise to get a handle on how to do this early, so you can use it throughout your work on this project.

Avoiding reports of memory leaks that are not your responsibility

BooEdit uses a well-known open source library called **ncurses** for managing cursor movements, windowing, and text output. Programs that use **ncurses** will appear to leak memory when run under Memcheck, though all of these leaks will be in the category called *still reachable* (i.e., memory still reachable from pointers stored in global variables when the program ended). I've made my best effort to eliminate the reporting of these leaks by including a file **memcheck.supp**, which instructs Memcheck not to report on the kinds of leaks emanating from **ncurses** that would appear, for our purposes, to be memory leaks caused by us.

You should not modify **memcheck.supp**. If you do change it, you may find that you start to receive reports of memory leaks that do not appear to emanate from your own code, so your best bet is to leave this file as-is. The **./run** script instructs Memcheck to use **memcheck.supp** in its analysis automatically, so you shouldn't see leaks that emanate from **ncurses** in your work on this project. If you do believe you're seeing such leaks, let me know; while I think I've got this part dialed in, it's possible that this will need a tweak or two to get it completely right.

Deliverables

After using the **gather** script in your project directory to gather up your C++ source and header files (including your unit tests) into a single **project4.tar.gz** file, submit that file (and only that file!) to Checkmate. (Note that the filename is different this time; the tarball is now called **project4.tar.gz** to avoid confusion.)

Follow this link for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally. (It's not a bad idea to look at the contents of your tarball on your host operating system before submitting it.)

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course which is described in the section titled *Late work* at this link.

Unit test improvements by Alex Thornton, Spring 2019.
Additional details added to the write-up, plus some adjustments to the implementation by Alex Thornton, Fall 2016.
Re-implementation of program (to support unit testing of various components) and additional explanation and background added by Alex Thornton, Fall 2015
An example implementation, support for logging, notes on using Memcheck effectively, various clarifications, and several BooEdit bug fixes by Alex Thornton, Fall 2014
Originally written by Alex Thornton, Winter 2014