CSE 3150 — Essentials of C++ — Yufeng Wu — Spring 2024
Project Part 3: Build a simple console user interface (UI) Due: 04/24/24 (note: this is a Wednesday).

In this part of the project, you are to implement a simple user interface, which is to be used as the *front end* of the airline reservation system.

The standard C++ doesn't offer user interface beyond some very basic input/output. During the past 50 years, graphical user interface (GUI) is getting more and more popular. Nowadays, users would expect some form of intuitive UI even for console-based applications (i.e., those running in a terminal window). Please note that while programs running on terminal windows may seem like some ancient legacy code, such programs are actually still very useful (e.g., those running in servers/clouds). Therefore, it would be useful to construct a reusable code base for supporting simple user interface in a console-based environment. There are existing C++ console-based UI libraries, e.g., *ncurses*). In this assignment, however, we are not going to use these libraries. Instead, I am asking you to implement user interface (almost) from scratch. Well, I am not asking you to develop a C++ console UI library that can make code like ncurses obsolete. Your code likely will look far worse than the production-quality libraries. The purpose of this exercise is letting you practice C++ programming with object orientation (especially by applying design patterns). As I said repetitively in the class, the way to learn C++ object orientation is by working on a project that has some level of complexity. Only working on a relatively large task will let you appreciate the power of object orientation/design patterns.

You are given a skeleton of UI code, which can function but only in a very limited way. Your task is implementing a few more classes to make the UI more useful.

1 User interface: classes

There are two kinds of user interface classes.

1. Various user interface items. You must have seen something like these before. Each UI item occupies a specific region of the window.

   (a) *This is given. ECConsoleUIItem*: this is the abstract base class for all the UI items in the following. While this class is simple, it is the foundation of the entire UI code design. Briefly,

      i. Each ECConsoleUIItem occupies a segment of the window. There is a single constructor of this class that takes the coordinates of the upper-left and bottom-right corners. Note: coordinates are in the rows (not in pixels). Our UI only knows about rows/columns of texts. It has no concepts of pixels at all!

      ii. *GetTextAtRow* function is meant to return the text within the UI item at a specific row (i.e., the offset of row within the UI, not the row in the entire window). This will decide how the UI will look like in the text view.

      iii. *GetHighlightPosition* function: some UI item (e.g., list box) has some items to highlight (e.g., the selected item in list box). This is meant to return what row is to be highlighted.

      iv. *IsHandlingKeys*: return true if the UI item is active, otherwise return false. Some UI is static. For example, a text label will always stay the same; other UIs (e.g., list box) are active (i.e., responding to user keys).

      v. Several functions to handle specific keys, e.g., arrows and enter.

   (b) *This is given.* Text label (class: *ECConsoleUITextLabel*). This is a simple text label (with a single text row and is located at a specific position on the screen).

(c) *You need to implement.* List box (class: *ECConsoleUIListBox*). A list box contains multiple rows (each for a choice of something). By default, the first row is chosen. You can change the choices using "UP" or "DOWN" keys. Note: there may be more choice rows than the size of the list box. In this case, you need to "scroll" up or down if possible. You don't need to implement "wrap-around" (that is, at the first row, you can ignore "UP"; at the last choice, you can ignore "DOWN" key.

(d) *You need to implement.* Text field (class: *ECConsoleUITextField*). A text field displays a ":" at the beginning (thus you know where the text field starts. By default, text field is not responding to keyboard input. Once you press "ENTER", the cursor will be placed inside the text field and you can type to enter the text. You can use "BACKSPACE" to erase the entered text. You can use "CTRL-C" to escape from the edit mode and back to the default no-edit mode.

(e) *This is given.* Button *ECConsoleUIButton*: this class is meant to respond to the "ENTER" key and perform some action. For now, you don't need to implement what to do if a user hits "ENTER".

You can highlight a UI item. The texts in a highlighted UI item are displayed in the color RED. For list box, the current choice should be highlighted. By default, text labels are not highlighted (unless the user makes it to be).

2. A text UI window (implemented in the class *ECConsoleUIView*. This is a main user interface which occupies the entire console window. It contains a list of user interface items (as described above). That is, ECConsoleUIView works as a placeholder for working with multiple user interface.

(a) A text UI window keeps track the currently active UI item. By default, the first added UI item is considered to be active. The texts belong to the first UI item are to be colored differently (i.e., in Blue color) so you know which UI item is active.

(b) A user uses the "CTRL-A" key to move to the next active UI item. Note: you should ignore UI items that don't take user inputs. For example, you should skip any text labels. Also note: you should loop through UI items in the order when they are added to the UI window.

2 What you need to do?

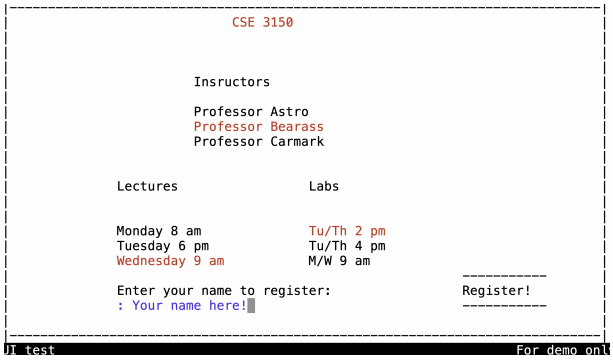

Figure 1: Five text labels (in addition to the borders, which are also text labels). Three list boxes. One text field: which starts with a ":". One button with label "Register!".

You need to complete the implementation of the list box class. For **extra credits**, you can also implement the text field class. Then you should create a UI on your own that looks like the following Fig. 2. This is to demonstrate that you have made the UI work. Please note: the text field aspect of the UI is for extra credits; you don't have to implement it in your code. I would suggest you to read the provided test code to see how the user interface is built and used. To help you gain some intuition about the UI, here are some screenshots of the UI I made.

Note: your UI doesn't have to be exactly like the one shown here. As long as your UI allows interaction, e.g., selecting different items in the list box, that is OK.

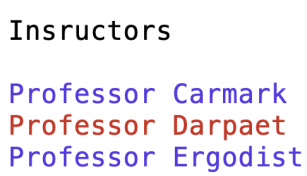

Figure 2: Scrolling of a list box to show more choices of professors. The list box only shows three rows at the same time. Note: the list box must be active (colored as blue) to change settings.

The following figures show how the list box and text field items should work.

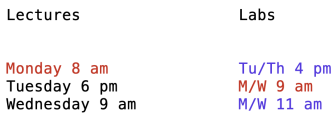

Figure 3: UI items can be placed side by side. A user uses CTRL-A key to change the active item (the item to edit). Here, the "Labs" list box is active.

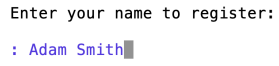

Figure 4: Text field. After this text field becomes active, press "ENTER" to enable editing. Then you can type the text (or use BACKSPACE to erase). Note the cursor position should be set properly to allow intuitive UI.

3 What to submit?

Submit the followiing to Gradescope.

1. Submit your source code.
2. A short text file, which contains (i) how to build an executable from your code, (ii) a statement about whether you have implemented the list bost and how well it works, and (iii) whether you have mplemented the text field (for extra credits) and how well it works.
3. Submit screenshots in some common image format that show you have achieved the functionalities as you claimed.