Assignment 6: State-Based Interface

This homework assignment, worth 20 marks and 5% of your final grade, is due by 4:00 PM on Friday, Oct. 31st. Submit your solution (a printed and stapled/bound copy of `Edit.cpp`) to the ECE 312 LEC A1 assignment box outside the ECERF reception (2nd floor). Include your name and your student ID, the percentage of the submission (excluding code and ideas given with the assignment) original by you, and the names of all other contributors to your submission (indicate the percent taken from each).

Electronic and wrong-box submissions will not be accepted. If any part of the assignment is submitted late, the whole assignment will be penalized 2 marks, i.e., 10% of the maximum mark, per business day. Submissions received, in whole or in part, after 4:00 PM on Tuesday, Nov. 4th, will receive 0 marks. Do not cheat, do not plagiarize, do not misrepresent facts, and do not participate in an offence!

Objectives

Interfaces, such as user interfaces, may be purposefully limited in mechanical and electrical terms for cost, size, reliability, and/or other reasons. Interface sophistication may be provided using software instead, which can interpret the pattern of limited input choices over time. In this assignment, you will program, document, and test state-based code, using static local variables, to implement part of the multi-tap interface of a classic Nokia handset. For simplicity, hardware is not involved.

Getting Started

In Visual Studio (Express) 2012 or 2013, create a new one-project solution with `Main.cpp` and `Edit.cpp` as source files. Put `Input1.txt` in the project's working directory. Set it and `Output1.txt` as input and output files, respectively, as follows. Right click on your project in the Solution Explorer and select Properties. Choose Configuration Properties ≫ Debugging in the left pane. In the right pane, by Command Arguments, enter "`Input1.txt Output1.txt`" without quotes and click on OK.

Build and run the project. Verify that it creates an `Output1.txt` file successfully. If you cannot build and run the code, try the ETL E5-012 lab, where Visual Studio 2012 is available on Linux via VMware Player. Visual Studio 2012 may also be available on the Windows PCs in the ETL E4-013 lab.

In addition to this document, please review (if the in-class review did not suffice): the Wikipedia page on multi-tap input; the Cprogramming page on the `static` keyword ("inside a function" usage only); and all code and comments given in `Main.cpp` and `Edit.cpp`. Further information is linked below.

Technical Specifications

Write/edit the `multiTap` function (`Edit.cpp`). Although some code is given there for Getting Started, you may delete/revise it. Do not change any other code, including the `multiTap` function interface. Do not add `include` statements or modify existing ones. Your entire solution should be programmed into the `Edit.cpp` file. You may add other functions if that will help you develop a solution. Functions you write/edit are allowed to have static local variables. Do not use any global variables.

Each time `multiTap` is called, it is given an integer, `key`, that either ranges from `0` to `9` or equals `PAUSE` (defined as `-1`). This means, respectively, that either the corresponding key was tapped or no key was tapped over a short time period, i.e., there was a pause. Based on the sequence of tapped keys and pauses, `multiTap` should output a sequence of characters, as early as possible, to the file stream `fout`. This output is based on the number of times a key is tapped, modulo the total number of symbols it represents, before either a different key is tapped or there is a pause. See the table below.

Taps − 1:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	Total
Key 0:	␣	0	␣	0	␣	0	␣	0	␣	0	␣	0	␣	0	2
Key 1:	,	.	'	?	!	"	1	-	(	)	@	/	:	_	14
Key 2:	A	B	C	2	A	B	C	2	A	B	C	2	A	B	4
Key 3:	D	E	F	3	D	E	F	3	D	E	F	3	D	E	4
Key 4:	G	H	I	4	G	H	I	4	G	H	I	4	G	H	4
Key 5:	J	K	L	5	J	K	L	5	J	K	L	5	J	K	4
Key 6:	M	N	O	6	M	N	O	6	M	N	O	6	M	N	4
Key 7:	P	Q	R	S	7	P	Q	R	S	7	P	Q	R	S	5
Key 8:	T	U	V	8	T	U	V	8	T	U	V	8	T	U	4
Key 9:	W	X	Y	Z	9	W	X	Y	Z	9	W	X	Y	Z	5

Presentation Specifications

Format your code professionally. Compare your placement of curly braces, i.e., `{` and `}`, to that of the given code. Use a similar approach or another professional approach. Code of the same code block should be left aligned and the indentation should consistently match the nesting level. In Visual Studio, selecting EDIT ≫ Advanced ≫ Format Document from the menu may correct alignment errors.

Ensure that long indented lines do not overflow without indentation when printed. Either break them into multiple indented lines (the compiler ignores line breaks) or print in landscape mode.

Write suitable comment headers for all functions you write/edit, which clearly describe the purpose, inputs, and outputs of each function. Functions that use static local and/or global variables are not fully defined this way. Describe, in the comment header, the state-driven behaviour of functions that use such variables. This is one good reason to minimize/simplify the use of state variables.

Review your working solution and look for ways to simplify it so that it would be easier to follow. For example, use refactoring to reduce code smell. This is another good reason to minimize/simplify the use of state variables. Furthermore, give helpful comments beyond those in the comment header.

Notes and Hints

Your code needs to incorporate the data given in the Technical Specifications. Global constants are okay – you can put them after the `#define` statement in `Edit.cpp` – but local constants are preferred. Use the `const` keyword, e.g., `const char string[] = ",.'?!\"1-()@/:_"` defines a constant string (note: `\"` is just the character `"`). Although you can use a 2D character array for the data, a superior choice is to use a 1D array of strings, where each string has a different length. Individual characters in the 1D array of strings may be accessed the same way as with a 2D character array.

Do *not* submit the output file, `Output1.txt`, generated by your code, which corresponds to the given input file, `Input1.txt`. Your code is marked against the specifications, above and beyond any output. However, your solution has (a) technical error(s) if your output file does not match the given one. Debug and correct the errors. If your output does match, your solution may still contain technical errors.

Reread the Technical Specifications and verify that you have met all the requirements exactly. Try other test cases. For example, put `Input2.txt` in the project's working directory and, using the Getting Started instructions, set it and `Output2.txt` as input and output files, respectively. Check your output against the given one etc. Note: The specifications do not place an upper bound on the number of times a key may be pressed repeatedly. Will your code work with a very high number of repetitions?