NWEN 241 Assignment 1

"Text Editor"

Release Date: **11 March 2019**

Submission Deadline: **25 March 2019, 23:59**

A **text editor** is a systems program that facilitates the creation and modification of files that contain strings of characters.

In this assignment, you will implement several basic text editor operations such as insert, delete, replace, etc. You will implement some of the operations in pure C (Part I), and some in C++ (Part II). In Part III, you will be asked to design another subsystem of a text editor program using C++.

You only need to submit the required implementations. You do not need to write a `main()` function, but you would need one if you want to test your code. In any case, do not submit code with the `main()` function.

Full marks is 100.

Instructions and Submission Guidelines:

- You should provide appropriate comments to make your source code readable. If your code does not work and there are no comments, you may lose all the marks. See the marking criteria at the end of this document for details about the marks for commenting.
- You should follow a consistent coding style when writing your source code. There is a short discussion about coding style below. See the marking criteria at the end of this document for details about the marks for coding style.
- Submit the required files to the Assessment System (`https://apps.ecs.vuw.ac.nz/submit/NWEN241/Programming_Assignment_1`) on or before the submission deadline.
- Late submissions (up to 48 hours from the submission deadline) will be accepted but will be penalized. No submissions will be accepted 48 hours after the submission deadline.

"Skeleton" Files

In this assignment, you will be given the following skeleton files which should be under the `files` directory in the archive that contains this file.

1. `editor.h` – C header file that contains macro definitions and all the required function prototypes for Part I. *Do not modify this file!*
2. `editor.hh` – C++ header file that contains macro and class definitions for Part II. *Do not modify this file!*
3. `myeditor.hh` – Empty C++ header file that you will use to define your class for Part II.
4. `editor.c` – Empty C file that you will use to implement functions for Part I. You are free to implement other functions within this file that you think are needed to fulfil the tasks.
5. `myeditor.cc` – Empty C++ file that implements your class for Part II.

To get a better understanding of the tasks, you should read this document together with the provided skeleton code.

Coding Style

Coding style (also known as coding standard) refers to the use of appropriate indentation, proper placement of braces, proper formatting of control constructs, etc. Following a particular coding style consistently will make your source code more readable.

There are many coding standards available (search "C/C++ coding style"). Most of these standards are dense and will take you many days (even weeks) to read and understand. If you want to follow a *lightweight* coding style, consult the Linux kernel coding style (`https://www.kernel.org/doc/html/v4.10/process/coding-style.html`). You only need to read sections 1–3 of this document.

Note that you do not have to follow every recommendation you can find in a coding style document. If you change, for instance the tab size from 8 to 4, that is fine. You just have to apply that style consistently.

Editing Buffer

An important component of a text editor is the *editing buffer* which is essentially a block of computer memory that contains part of the text document being edited. It can be viewed as one-dimensional array of characters from the perspective of the programmer.

Part I: Pure C Programming

You may only use the Standard C Library to perform the tasks in this part. You should implement the functions in `editor.c`.

Task 1.

Basics [10 Marks]

In this task, you will implement a function with prototype

```
int editor_insert_char(char *editing_buffer, char
    to_insert, int pos)
```

which will insert the character `to_insert` at index `pos` of `editing_buffer`. The size of `editing_buffer` is `EDITING_BUFLEN`. When a character is inserted at index `pos`, each of the original characters at index `pos` until the end of buffer must be moved by one position to the right. The last character is thrown out.

The function should return 1 if the character insertion occurred, otherwise it should return 0.

To clarify, suppose that `EDITING_BUFLEN` is defined to be 21 and the contents of `editing_buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		b	r	o	w	n		f	o	x	\0	\0

After invoking

```
int r = editor_insert_char(editing_buffer, 's', 9);
```

the contents of `editing_buffer` should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k	**s**		b	r	o	w	n		f	o	x	\0

and the value of `r` is 1.

Task 2.

Basics [10 Marks]

In this task, you will implement a function with prototype

```
int editor_delete_char(char *editing_buffer, char
    to_delete, int offset)
```

which will delete the first occurrence of the character `to_delete`. The search should start from index `offset` of `editing_buffer`. The size of `editing_buffer` is `EDITING_BUFLEN`. When a character is deleted at index `pos`, each of the original characters at index `pos` until the end of buffer must be moved by one position to the left. A null character is inserted at the end of the buffer.

The function should return 1 if the character deletion occurred, otherwise it should return 0.

To clarify, suppose that `EDITING_BUFLEN` is defined to be 21 and the contents of `editing_buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		b	r	o	w	n		f	o	x	\0	\0

After invoking

```
int r = editor_delete_char(editing_buffer, 'f', 10);
```

the contents of `editing_buffer` should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		b	r	o	w	n		o	x	\0	\0	\0

and the value of `r` is 1.

Task 3.

Completion [10 Marks]

In this task, you will implement a function with prototype

```
int editor_replace_str(char *editing_buffer, const char
    *str, const char *replacement, int offset)
```

which will replace the first occurrence of the string `str` with `replacement`. The search should start from index `offset` of `editing_buffer`. The size of `editing_buffer` is `EDITING_BUFLEN`.

The replacement should not overwrite other contents in the buffer. This means that if `replacement` is longer than `str`, there is a need move the characters after `str` to the right. Likewise, if `replacement` is shorter than `str`, there is a need move the characters after `str` to the left. When moving characters to the right, throw out characters that will not fit in the buffer. When moving characters to the left, insert null characters in the vacated positions.

If the replacement text will go beyond the limits of `editing_buffer`, then replacement should only occur until the end of `editing_buffer`.

If the string replacement occurred, the function should return the index corresponding the last letter of `replacement` in `editing_buffer`, otherwise, it should return -1. If the replacement text will go beyond the limits of `editing_buffer`, the function should return `EDITING_BUFLEN-1`.

To clarify, suppose that `EDITING_BUFLEN` is defined to be 21 and the contents of `editing_buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		b	r	o	w	n		f	o	x	\0	\0

After invoking

```
int r = editor_replace_str(editing_buffer, "brown",
   "blue", 0);
```

the contents of `editing_buffer` should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		**b**	**l**	**u**	**e**		f	o	x	\0	\0	\0

and the value of `r` should be 13 (which is the index of 'e' in "blue").

Task 4.

Challenge [10 Marks]

In this task, you will implement a function with prototype

```
void editor_view(char **viewing_buffer, const char
    *editing_buffer, int wrap)
```

which will essentially copy the contents of the `editing_buffer` to the `viewing_buffer` for display to the user. Note that the latter is a two-dimensional array, with dimensions `VIEWING_COLS` columns and `VIEWING_ROWS` rows. `VIEWING_COLS` and `VIEWING_ROWS` are defined in `editor.h`.

Prior to the copying, the function must set every character in the `viewing_buffer` to the null character.

The argument `wrap` controls the behaviour of the copying process from `editing_buffer` to `viewing_buffer` as follows:

- Regardless of the value of `wrap`, whenever a newline character is encountered, subsequent text after the newline charater is copied to the next row. Note that the newline character itself is not copied to `viewing_buffer`.
- When `wrap` is non-zero, the text must be wrapped. This means that when the newline character is *not* encountered before the end of the current row (at column `VIEWING_COLS-1` in `viewing_buffer`), subsequent text is copied to the next row. Note that column `VIEWING_COLS-1` in `viewing_buffer` is never filled and will retain the null character.
- When `wrap` is 0, the text is not wrapped. This means that when the newline character is *not* encountered before the end of the current row (at column `VIEWING_COLS-1`), succeeding text in the `editing_buffer` are discarded until a newline is encountered which will cause the succeeding text to be copied to the next row. Note that column `VIEWING_COLS-1` in `viewing_buffer` is never filled and will retain the null character.

The copying process should terminate when a null character is encountered.

To clarify, suppose that `EDITING_BUFLEN` is defined to be 48 and the contents of `editing_buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
T	h	e		q	u	i	c	k		b	r	o	w	n	\n	...

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
f	o	x		j	u	m	p	s		o	v	e	r	\n	\n	...

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
t	h	e		l	a	z	y		d	o	g	\n	\0	\0	\0

Suppose that `VIEWING_COLS` and `VIEWING_ROWS` are 11 and 8, respectively. After invoking

```
editor_view(viewing_buffer, editing_buffer, 1);
```

the resulting contents of `viewing_buffer` should be

	0	1	2	3	4	5	6	7	8	9	10
0	T	h	e		q	u	i	c	k		\0
1	b	r	o	w	n	\0	\0	\0	\0	\0	\0
2	f	o	x		j	u	m	p	s		\0
3	o	v	e	r	\0	\0	\0	\0	\0	\0	\0
4	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
5	t	h	e		l	a	z	y		d	\0
6	o	g	\0	\0	\0	\0	\0	\0	\0	\0	\0
7	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

If the function is instead called with

```
editor_view(viewing_buffer, editing_buffer, 0);
```

the resulting contents of `viewing_buffer` should be

	0	1	2	3	4	5	6	7	8	9	10
0	T	h	e		q	u	i	c	k		\0
1	f	o	x		j	u	m	p	s		\0
2	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
3	t	h	e		l	a	z	y		d	\0
4	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
5	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
6	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
7	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

Part II: C++ Programming

You may use the C++ Standard Library to perform the tasks in this part.

Task 5.

Basics [10 Marks]

In this task, you will define a class that extends the `EditingBuffer` class defined in `editor.hh`. You should declare the class in `myeditor.hh` and name it `MyEditingBuffer`. This class should be in same namespace as `EditingBuffer`. The access mode of the inherited members should be preserved.

You are free to declare additional member variables and functions that are necessary to perform the subsequent tasks. Provide sufficient comment, and ensure that these additional member variables and functions are *not* publicly accessible.

Task 6.

Basics [10 Marks]

In this task, you will implement the member function

```
bool replace(char c, char replacement, int offset)
```

which will replace the first occurrence of the character `c` with `replacement` in the `buffer`. The search should start from index `offset` of `buffer`. The length of `buffer` is `BUFFER_LEN` which is defined in `editor.hh`.

The function should return `true` if the character insertion occurred, otherwise `false`.

To clarify, suppose that `BUFFER_LEN` is defined to be 21 and the contents of `buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		b	r	o	w	n		f	o	x	\0	\0

After invoking

```
MyEditingBuffer meb;
...
bool r = meb.replace('b', 'B', 5);
```

the resulting contents of the buffer should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		**B**	r	o	w	n		f	o	x	\0	\0

and the value of `r` should `true`.

You should implement this member function in `myeditor.cc`.

Task 7.

Completion [10 Marks]

In this task, you will implement the member function

```
int replace(std::string str, std::string replacement, int
    offset)
```

which will replace the first occurrence of the string `str` with `replacement`. The search should start from index `offset` of `buffer`.

The replacement should not overwrite other contents in the buffer. This means that if `replacement` is longer than `str`, there is a need move the characters after `str` to the right. Likewise, if `replacement` is shorter than

`str`, there is a need move the characters after `str` to the left. When moving characters to the right, throw out characters that will not fit in the buffer. When moving characters to the left, insert null characters in the vacated positions.

If the replacement text will go beyond the limits of `buffer`, then replacement should only occur until the end of `buffer`.

If the string replacement occurred, the function should return the index corresponding the last letter of `str` in `editing_buffer`, otherwise, it should return -1. If the replacement text will go beyond the limits of `buffer`, the function should return `BUFFER_LEN-1`.

To clarify, suppose that `BUFFER_LEN` is defined to be 21 and the contents of `buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		b	r	o	w	n		f	o	x	\0	\0

After invoking

```
MyEditingBuffer meb;
...
std::string s("brown");
std::string t("violet");
int r = meb.replace(s, t, 8);
```

the resulting contents of the buffer should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	h	e		q	u	i	c	k		**v**	**i**	**o**	**l**	**e**	**t**		f	o	x	\0

and the value of `r` should be 15 (which is the index of 't' in "violet").

You should implement this member function in `myeditor.cc`.

Task 8.

Challenge [15 Marks]

In this task, you will implement the member function

```
void justify(char **viewingBuffer, int rows, int cols)
```

which will justify the contents of `buffer` by adjusting the space in be-

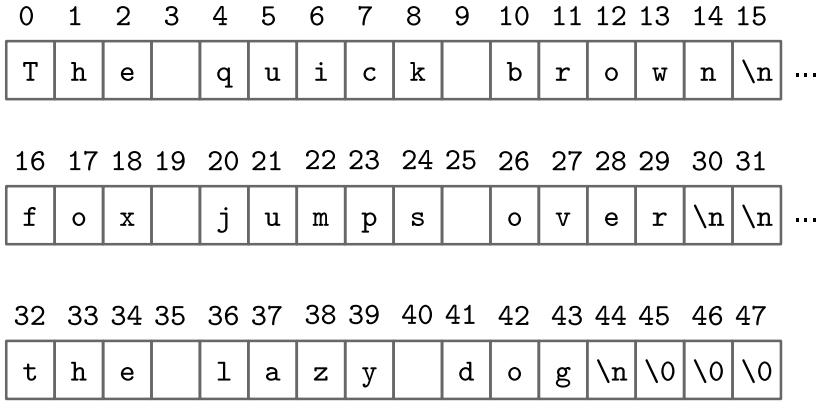
tween words. The justified text should be stored in `viewingBuffer`, a two-dimensional array which has `rows` rows and `cols` columns.

To implement the text justification, the contents of `buffer` must first be copied to `viewingBuffer` such that the copied text is wrapped. You may reuse your implementation in Task 4 for this purpose.

After copying, the contents of `viewingBuffer` must be justified, i.e., **the character at column `cols-2` of every row must not be a whitespace if possible**. This can be done by adding whitespaces in between words. When justifying, follow these guidelines:

1. Do not justify empty rows (rows consisting of null characters) and single-word rows.
2. Do not split words, *i.e.*, do not insert spaces in between the letters of a word.
3. You can split a single-word row if the single word is too long to fit one row.
4. Do not merge words.
5. If necessary, discard characters that will not fit in the buffer.

To clarify, suppose that `BUFFER_LEN` is defined to be 48 and the contents of `buffer` are



After invoking

```
MyEditingBuffer meb;
...
meb.justify(viewingBuffer, 8, 11);
```

the resulting contents of `viewingBuffer` should be

	0	1	2	3	4	5	6	7	8	9	10
0	T	h	e			q	u	i	c	k	\0
1	b	r	o	w	n	\0	\0	\0	\0	\0	\0
2	f	o	x			j	u	m	p	s	\0
3	o	v	e	r	\0	\0	\0	\0	\0	\0	\0
4	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0
5	t	h	e		l	a	z	y		d	\0
6	o	g	\0	\0	\0	\0	\0	\0	\0	\0	\0
7	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0	\0

You should implement this member function in `myeditor.cc`.

Part III: Systems Program Design

In Parts I and II, you implemented some functionality for manipulating the contents of the editing buffer. In this part, you will *design* a C++ program for manipulating the *viewing buffer*.

Task 9.

Challenge [15 Marks]

The *viewing buffer* is a block of computer memory that contains part of the text document shown in the computer display. It can be viewed as a two-dimensional array of characters, where the each row corresponds to a line of text. In practice, the number of rows and columns should be determined by the size of the display/terminal. For simplicity, fix the number of rows and columns to 25 and 80, respectively.

In designing the program, assume that a *main buffer* contains the entire text document being edited. Assume that the main buffer variable is global and is declared as `char main_buffer[1000000]` somewhere.

The following functionality are required to manipulate the viewing buffer:

- Load parts of text document from main buffer to the viewing buffer (from a given line/row position).
- Scroll up viewing buffer content by a given number of lines/rows.
- Scroll down viewing buffer content by a given number of lines/rows.
- Enable wrapping
- Disable wrapping
- Show line numbering
- Hide line numbering

What You Need To Do:

Create the necessary files (header and source) to define a class that would implement the above set of functionality. For every member variable and function that you specify, provide sufficient comments about their purpose.

You do not need to implement the member functions. However, you will need to write the *member function header* in the source file, followed by an empty body implementation.

Put the header and source files in a ZIP file called `part3.zip`.

Marking Criteria for Tasks 1, 2, 3, 4, 6, 7, and 8:

Criteria	Weight	Expectations for Full Marks
"Compilability"	10%	Source code compiles without warnings
Commenting	10%	Source code contains sufficient and appropriate comments
Coding Style	10%	Source code is formatted, readable and uses a coding style consistently
Correctness	70%	Handles all possible cases correctly
	100%	

Marking Criteria for Tasks 5 and 9:

Criteria	Weight	Expectations for Full Marks
Commenting	10%	Source code contains sufficient and appropriate comments
Coding Style	10%	Source code is formatted, readable and uses a coding style consistently
Correctness	40%	Addresses all specifications and correctly uses syntax in the declarations and/or definitions
Completeness	30%	Declaration and/or definition of all required member variables and functions
	100%	