NWEN 241 Assignment 1

(Weeks 1–2 Topics)

Release Date: **4 March 2024**

Submission Deadline: **18 March 2024, 23:59**

In this assignment, you will be asked implement C functions, submitted in a file named `editor.c`.

You must submit the required file to the Assessment System (`https://apps.ecs.vuw.ac.nz/submit/NWEN241/Assignment_1`). Any assignment submitted up to 24 hours after the deadline will be penalised by 20%, and any assignment submitted between 24 and 48 hours after the deadline will be penalised by 40%. Any assignment submitted 48 hours or more after the deadline will not be marked and will get 0 marks.

Important: The Assessment System is configured **not to accept submissions that do not compile.** So please test that your code compiles before submitting it.

Full marks is 100. The following table shows the overall marks distribution:

Criteria	**Marks**	**Expectations for Full Marks**
Compilation	5	Compiles without warnings
Comments	10	Sufficient and appropriate comments
Code Quality	15	Efficient code and use of consistent coding style
Correctness	70	Handles all test cases correctly (see marks distribution below)
Total	**100**	

For the **Correctness** criteria, the following table shows the marks distribution over the different task types:

Task Type	**Marks**
Core	45
Completion	15
Challenge	10
Total	**70**

Introduction

This assignment will test your application of the conceptual knowledge of C fundamentals to solve practical programming tasks. You may only use the Standard C Library to perform the tasks in this part. You must implement the functions in file named `editor.c`.

The programming tasks involve the implementation of several basic **text editor** operations: insert, delete, replace, etc. An important component of a text editor is the *editing buffer* which can be viewed as one-dimensional array of characters. The functions you will be implementing in deal with manipulating the contents of the editing buffer: (*i*) for Core (Tasks 1 and 2), you will implement `editor_insert_char` and `editor_delete_char`; (*ii*) for Completion (Task 3), you will implement `editor_replace_str`; and (*iii*) for Challenge (Task 4), you will implement `editor_view`.

Sample code showing an example on how you can test your code are provided under the `files` directory in the archive that contains this file.

Commenting

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 marks.

Code Quality

Code quality refers to (*i*) the use of efficient coding techniques and (*ii*) use of consistent coding style or standard.

When writing your source code, strive for *coding efficiency* which covers the following aspects: (*i*) elimination of unessential operations and variables; (*ii*) creation of functions to contain blocks of code that are used repeatedly; (*iii*) minimization of loop iteration; and (*iv*) avoiding the use of global variables (i.e. variables that are declared outside any function.)

In addition, you should follow a consistent coding style when writing your source code. Coding style (aka coding standard) refers to the use of appropriate indentation, proper placement of braces, proper formatting of control constructs, and many others. Following a particular coding style consistently will make your source code more readable. There are many coding standards available (search "C coding style"), but we suggest you consult the *lightweight* Linux kernel coding style (see `https://www.kernel.org/doc/html/v4.10/process/coding-style.html`). The relevant sections are Sections 1, 2, 3, 4, 6, and 8. Note that you do not have to follow every recommendation you can find in a coding style document, you just have to apply that style consistently.

Task 1.

Core [20 Marks]

Implement a function with the prototype

```
int editor_insert_char(char editing_buffer[], int editing_buflen,
                       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.

For example, if `editing_buflen` is 16 and the contents of `editing_buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	e	l	l	o	,		W	o	r	l	d	!	\0	\0	\0

after executing

```
int r = editor_insert_char(editing_buffer, 16, 's', 12);
```

the value of `r` should be 1 and contents of `editing_buffer` should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	e	l	l	o	,		W	o	r	l	d	s	!	\0	\0

You can test your implementation by compiling `editor.c` together with `t1test.c` (provided under the `files` directory). To do this, just type

```
gcc editor.c t1test.c -o t1test
```

in the terminal. Make sure that `editor.c` and `t1test.c` are in the same directory. If everything goes well, this will generate an executable file `t1test`. To see if your implementation is correct, run `t1test` and compare the expected and actual buffer contents and return values. If they match, it means your implementation passes the test. You are free to modify `t1test.c` if you want to add in more test cases.

Task 2.

Core [25 Marks]

Implement a function with the prototype

```
int editor_delete_char(char editing_buffer[], int editing_buflen,
                       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 (`'\0'`) is inserted at the end of the buffer. The function should return 1 if the character deletion occurred, otherwise it should return 0.

For example, if `editing_buflen` is 16 and the contents of `editing_buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	e	l	l	o	,		W	o	r	l	d	!	\0	\0	\0

after executing

```
int r = editor_delete_char(editing_buffer, 16, 'o', 6);
```

the value of `r` should be 1 and the contents of `editing_buffer` should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	e	l	l	o	,		W	r	l	d	!	\0	\0	\0	\0

You can test your implementation by compiling `editor.c` together with `t2test.c` (provided under the `files` directory). To do this, just type

`gcc editor.c t2test.c -o t2test`

in the terminal. Make sure that `editor.c` and `t2test.c` are in the same directory. If everything goes well, this will generate an executable file `t2test`. To see if your implementation is correct, run `t2test` and compare the expected and actual buffer contents and return values. If they match, it means your implementation passes the test. You are free to modify `t2test.c` if you want to add in more test cases.

Task 3.

Completion [15 Marks]

Implement a function with the prototype

```
int editor_replace_str(char editing_buffer[], int editing_buflen,
                       const char *str, const char *replacement,
                       int offset);
```

which will replace the first occurrence of the string `str` with `replacement`. The search for the first occurrence 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 and when moving characters to the left, insert null characters in the vacated positions.

If `str` is empty (regardless of the value of `replacement`), no string replacement should occur. If `replacement` is empty, then this is tantamount to deleting the string `str`.

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`.

For example, if `editing_buflen` is 16 and the contents of `editing_buffer` are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	e	l	l	o	,		W	o	r	l	d	!	\0	\0	\0

After executing

```
int r = editor_replace_str(editing_buffer, 16, "World!", "there", 0);
```

the value of `r` should be 11 (which is the index of the last 'e' in "there") and the contents of `editing_buffer` should be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
H	e	l	l	o	,		t	h	e	r	e	\0	\0	\0	\0

You can test your implementation by compiling `editor.c` together with `t3test.c` (provided under the `files` directory). To do this, just type

```
gcc editor.c t3test.c -o t3test
```

in the terminal. Make sure that `editor.c` and `t3test.c` are in the same directory. If everything goes well, this will generate an executable file `t3test`. To see if your implementation is correct, run `t3test` and compare the expected and actual buffer contents and return values. If they match, it means your implementation passes the test. You are free to modify `t3test.c` if you want to add in more test cases.

Task 4.

Challenge [10 Marks]

Implement a function with the prototype

```
void editor_view(int rows, int cols,
                 char viewing_buffer[rows][cols],
                 const char editing_buffer[],
                 int editing_buflen, int wrap);
```

which will copy the contents of the `editing_buffer` to the `viewing_buffer` for display to the user. Note that the `viewing_buffer` is a two-dimensional array, with dimensions `cols` columns and `rows` rows. 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 in `editing_buffer`, the text after the newline character is copied to the next row in `viewing_buffer`. Note that the newline character is not copied to `viewing_buffer`.
- 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 `cols-1`), the rest of the text in the `editing_buffer` are discarded until a newline is encountered which will cause the rest of the text after the newline to be copied to the next row. Note that column `cols-1` in `viewing_buffer` is never filled and will retain the null character.
- 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 `cols-1` in `viewing_buffer`), the text after is copied to the next row. Note that column `cols-1` in `viewing_buffer` is never filled and will retain the null character.

The copying process should terminate when a null character in the `editing_buffer` is encountered.

For example, if `editing_buflen` is 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	\0	\0	\0	\0	

and `cols` and `rows` are 11 and 8, respectively. After executing

```
editor_view(8, 11, viewing_buffer, editing_buffer, 48, 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

Alternatively, after executing

```
editor_view(8, 11, viewing_buffer, editing_buffer, 48, 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

You can test your implementation by compiling `editor.c` together with `t4test.c` (provided under the `files` directory). To do this, just type

```
gcc editor.c t4test.c -o t4test
```

in the terminal. Make sure that `editor.c` and `t4test.c` are in the same directory. If everything goes well, this will generate an executable file `t4test`. To see if your implementation is correct, run `t4test` and compare the expected and actual buffer contents and return values. If they match, it means your implementation passes the test. You are free to modify `t4test.c` if you want to add in more test cases.