Exercises — Ring Buffer

version **#7be580532266ed398481e31366afcc24b1950c2a**

Assistants C/Unix 2023 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

File Tree

```
ring_buffer/
├── main.c
├── ring_buffer.c   (to submit)
└── ring_buffer.h
```

Authorized headers : You are only allowed to use the functions defined in the following headers

- err.h
- errno.h
- assert.h
- stddef.h
- stddef.h
- stdlib.h
- string.h

Compilation : Your code must compile with the following flags

- -std=c99 -pedantic -Werror -Wall -Wextra -Wvla

1 Goal

Even though lists and vectors are good data structures, it's important to be comfortable with *buffers*. In this exercise, you will implement a **ring buffer**, a fixed-size buffer connected end-to-end. This structure is usefull to understand about buffers and streams.

We will use a *ring buffer* of size 10 for all our examples:

Be careful!

Before going further, we advise you to take a closer look on `memcpy(3)` and *void genericity*.

Tips

If you want to have a more detailed look of what a *Ring Buffer* is, we encourage you to take a look on the Circular Buffer Wikipedia.

2 Implementation

You will be asked to implement a simple **ring_buffer** struct. In order to be more guided, these struct needs to respect these declaration:

```
struct ring_buf
{
    char elements[RING_BUF_CAPACITY];
    size_t start;
    size_t end;
    size_t size;
    size_t element_size;
};
```

The structure contains the following fields:

- `elements`: the buffer you will manipulate
- `start`: the start position of your first element
- `end`: the end position of your last element
- `size`: the occupied size of the elements in your buffer
- `element_size`: the size of a single element

`RING_BUF_CAPACITY` being the static size of your buffer.

3 Functions

In order to create a fully functional *ring buffer*, you will have to implement some functions to respect some behavior:

```
struct ring_buf *rb_create(size_t element_size);
void rb_clear(struct ring_buf *rb);
int rb_push(struct ring_buf *rb, void *el);
void *rb_pop(struct ring_buf *rb, void *pop);
```

3.1 *rb_create*

`rb_create` allocates a `ring_buffer` with a fixed `element_size`, setting all other fields to 0.

If it fails to allocate, `NULL` must be returned.

Be careful!

`rb_create` allocate a `struct ring_buffer`, but it should be freeed by the user.

3.2 *rb_push*

`rb_push` copies an element at the `end` of your buffer, the first available place.

If the element is `NULL` or there is not enough room to fit it, you must return `1`, otherwise `0` must be returned

3.3 *rb_pop*

`rb_pop` returns the first element in your buffer and sets the `pop` pointer to its value.

If there is no element in the buffer, `pop` shouldn't be set and you must return `NULL`.

Be careful!

The element doesn't need to be suppressed in your buffer. The memory zone being not accessible anymore, it will be later overwritten if a new element is pushed at this index.

3.4 *rb_clear*

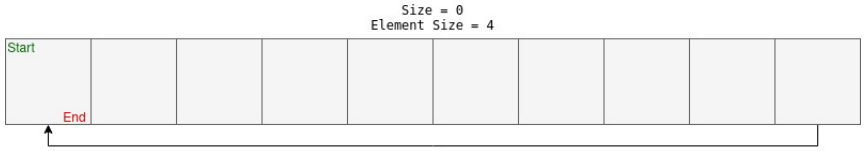
`rb_clear` reset the buffer, thus its `size`, `start` and `end` position must be set to 0.

4 Example

Those following examples considers that `RING_BUF_CAPACITY` is set to 10.

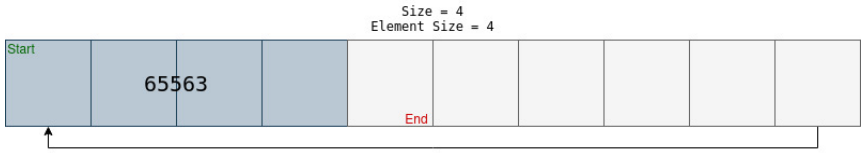
- We first create a `ring_buffer`, we can observe that its `size` is 0 and `start` and `end` are both at index 0.

```
rb_infos(rb);
```



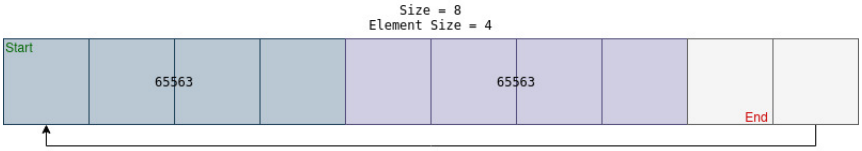
- We now push an element in it, `x` with the value 65563, `size` is now 4 and `end` is now at index 4.

```
assert(!rb_push(rb, &x)); // No error
rb_infos(rb);
```



- Pushing the same element in it, `size` is now 8 and `end` is now at index 8.

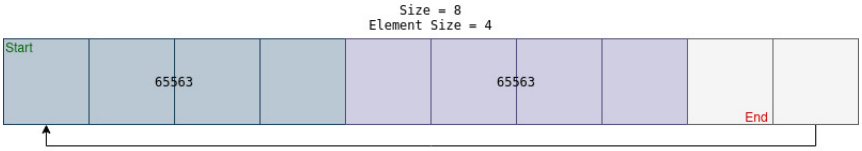
```
rb_infos(rb);
```



- Pushing an element is not possible since the size of an `integer` is 4. The vector state remains the same.

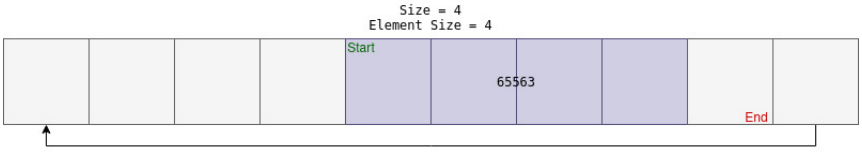
```

```



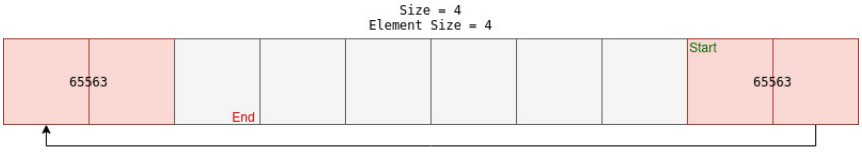
- Popping an element will change the `size` of the buffer to 4, `start` will now be at index 4.

```
int *z = rb_pop(rb, &y);
assert(x == y); // Supposed to be equal
```



- Popping and pushing an element will increment `start` and `end`. Since it is a ring buffer, `end` will be at index 2.

```
rb_push(rb, &x);
rb_infos(rb);
```



Tips

You will notice that when adding an element, we increment `end`, whereas popping an element, we increment `start`. We never decrement.

Be careful!

While pushing and popping elements, be careful with elements being between the junction of the end and the start of the buffer, such as the last example. You may require to copy twice.

The way is lit. The path is clear. We require only the strength to follow it.