Exercise — My memory recycler

version **#893e9bbbaee094ba0fa281e11988040afee0f2d6**

Assistants C/Unix 2025 <assistants@tickets.assistants.epita.fr>

Copyright

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

Contents

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

File Tree

```
my_recycler/
├── recycler.c  (to submit)
└── recycler.h  (to submit)
```

Authorized functions : You are only allowed to use the following functions

- malloc(3)
- free(3)

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

Compilation : Your code must compile with the following flags

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

1 Goal

In this exercise you will implement functions and structure to manage memory.

We will design a memory recycler to manage memory and make you even more comfortable with the notion of pointers. We need:

- Create and destroy functions to allocate a chunk of memory large enough to fit a given amount of blocks of the same size.
- Functions handling allocations and frees inside this chunk.

A header (`recycler.h`) containing all the required functions is provided.

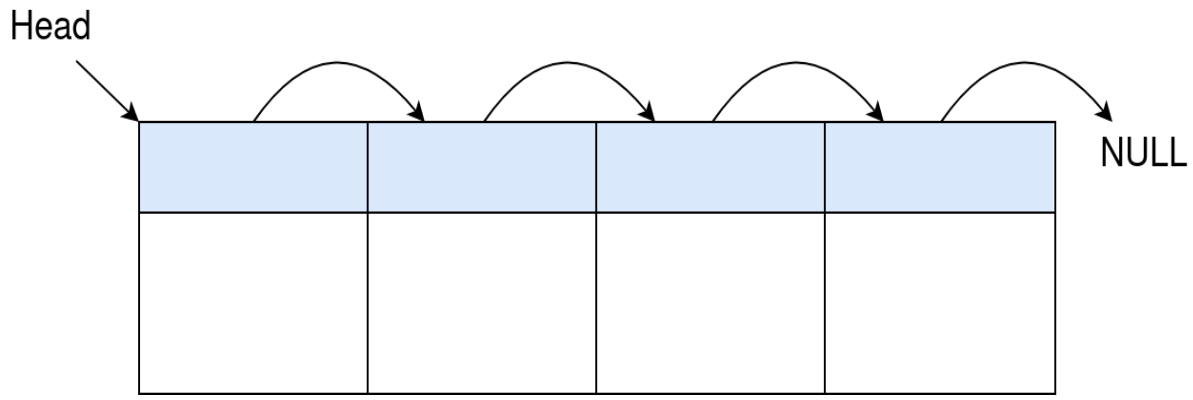
You are allowed to add this header to your submission, in the `my_recycler/` folder. Keep in mind that while we test your submission, we will use **our** `recycler.h`, so any modification will not be kept.

2 free list

Because you will split your allocated chunk in smaller blocks that you will deliver during a request for allocation, you need to keep track of all the free blocks. To do so, you will create a list where each element represents a block that is free: the free list.

```
struct free_list
{
    struct free_list *next; // next free block
};
```

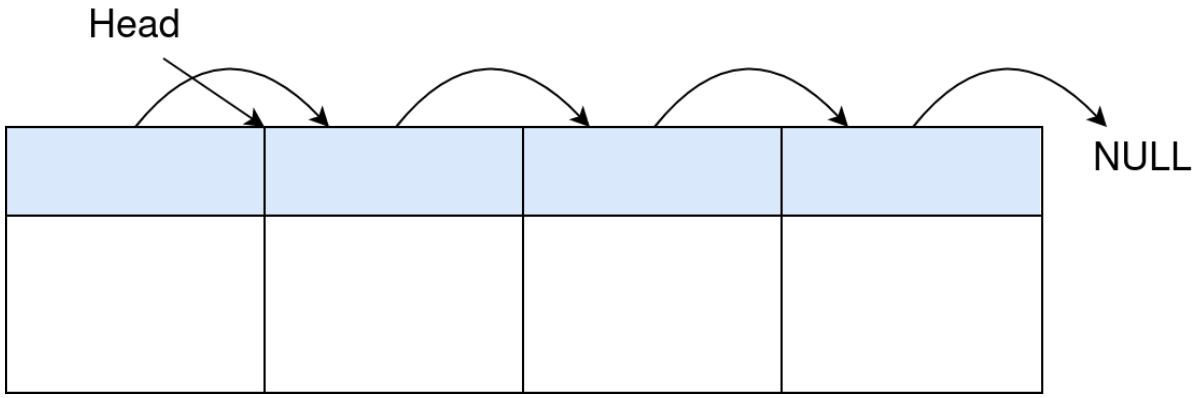
We will manage the free list by storing at the beginning of each free block a free list element that stores the address of the next free block. Let us consider you are using a recycler that addresses to the next free block and delivers blocks of 32 bytes, we will use the first bytes to store a pointer to the next free block.



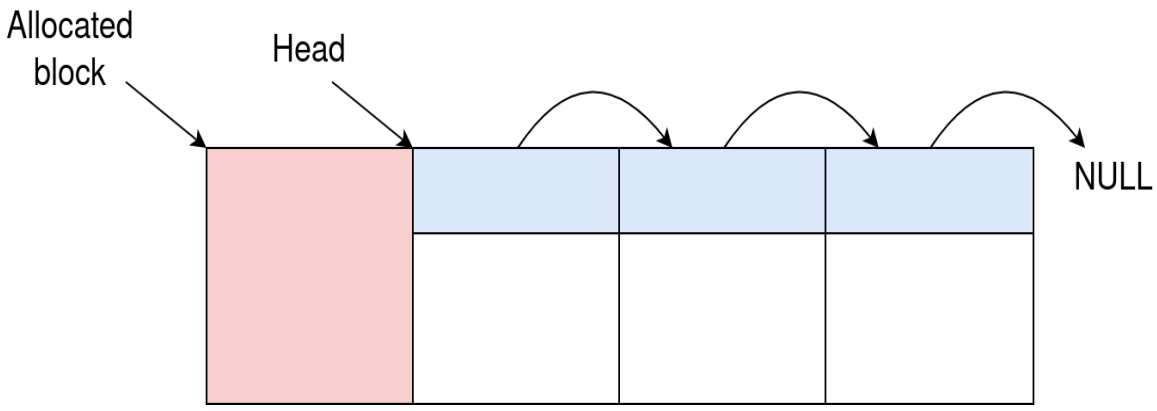
In the example above, all blocks are free. The address of the beginning of the block will be used as the address of a pointer to the next free block. If the address of the first block is `X`, the value at `*X` will be `X + 32`, the value at `*(X + 32)` will be `X + 64` and so on.

2.1 Allocate a block:

When you want to allocate a new block, you will first make the head pointing at the second element of the list:

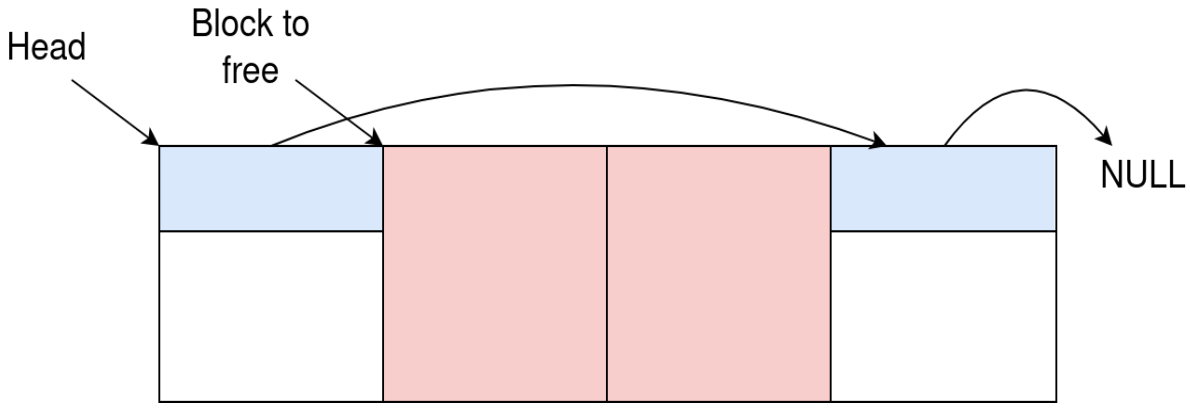


Then, the element previously pointed by the head can be returned.

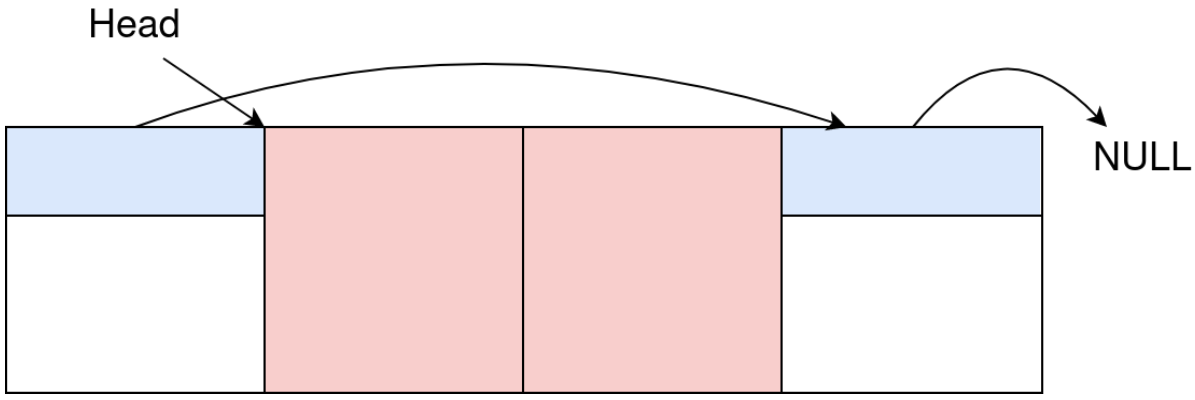


2.1.1 Free a block:

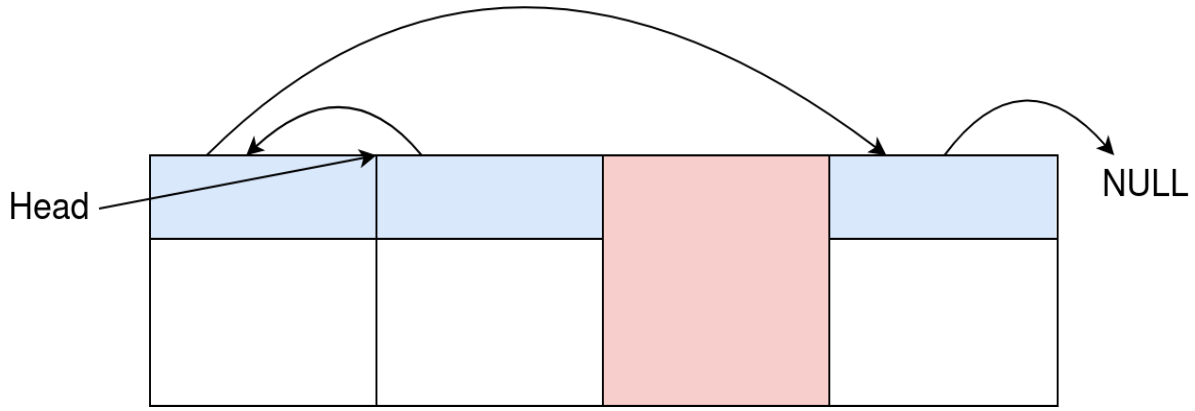
When you want to free a block, its address will be passed as an argument, and you will have to add the block in the free list.



You will first make the head pointing to the block.



Then create a new free list element at the beginning of the block. Finally, make the new free list element point to the block previously pointed by the head.



Be careful!

The first free block is not necessarily the first in the memory chunk; it is just the **head** of the free list.

3 given code

```c
struct recycler
{
    size_t block_size;
    size_t capacity; // number of blocks in the chunk
    void *chunk; // memory chunk containing all blocks
    void *free; // address of the first free block of the free list
};
```

3.1 recycler_create

- **Authorized functions:** `malloc(3)`, `free(3)`

```c
struct recycler *recycler_create(size_t block_size, size_t total_size);
```

Create a recycler structure, construct it (allocate the chunk, setup the free list ...) and return it.

Return `NULL` if:

- `block_size` of values that are not multiple of `sizeof(size_t)`: we want to keep memory align
- `block_size` or `total_size` are equal to zero
- If `total_size` is not a multiple of `block_size`
- If `malloc` fail

3.2 recycler_destroy

- **Authorized functions:** `free(3)`

```
void recycler_destroy(struct recycler *r);
```

Destroy the recycler by freeing all blocks and the recycler structure. Do not do anything if `r` is `NULL`.

3.3 recycler_allocate

- **Authorized functions:** *none*

```
void *recycler_allocate(struct recycler *r);
```

Find a free block and return it. Return `NULL` if `r` is `NULL` or if no block is available. Do not forget to update your free list.

3.4 recycler_free

- **Authorized functions:** *none*

```
void recycler_free(struct recycler *r, void *block);
```

Mark the given block as free. Do not do anything if `r` or `block` is `NULL`. Do not forget to update your free list.

`block` is guaranteed to be a part of the memory chunk allocated by `r`.

Seek strength. The rest will follow.