# Rapport Allocateur Mémoire RUST

Projet CSE-PC

TERESE Niels,
OPAROWSKI Corentin

11 octobre 2024

https://github.com/NielsTRS/allocateur-memoire-rust

Tuteurs : LEPERS Baptiste, MARANGOZOVA Vania

# Contents

# 1 Introduction

Ce rapport décrit la création d'un allocateur de mémoire en Rust dans le cadre du projet CSE-PC. L'accent a été mis sur l'utilisation des pointeurs bruts pour une gestion manuelle, afin de contourner les limitations des pointeurs intelligents de Rust, jugés trop contraignants pour ce type de projet.

# 2 Choix d'implémentations

## 2.1 Differentes methodes de gestion de memoire en RUST

Rust implémente un système de gestion de la mémoire distinct de celui du C, basé sur un modèle de propriété (ownership) et d'emprunt (borrowing) grace a des "smart pointers". Ce système permet à Rust d'assurer la sécurité mémoire sans avoir besoin d'un ramasse-miettes (garbage collector).

Par ailleurs, Rust permet également la gestion de pointeurs bruts (raw pointers), similaire à celle du C. Cependant, l'utilisation de ces pointeurs n'est pas "safe" par défaut et doit être entourée de blocs *unsafe* , qui délimitent explicitement les sections de code où les vérifications de sécurité de Rust ne s'appliquent pas.

## 2.2 L'utilisation de "raw pointers"

Dans le cadre de ce projet d'allocateur mémoire, nous avons opté pour l'utilisation exclusive de pointeurs bruts. En effet, ils offrent une gestion manuelle de la mémoire, permettant un contrôle direct sur les adresses de stockage. Bien que nous ayons envisagé l'usage des smart pointers de Rust, ceux-ci se sont révélés trop restrictifs pour ce type de projet, car ils imposent des contraintes qui ne sont pas adaptées aux exigences d'un allocateur mémoire nécessitant un contrôle total sur la manipulation des ressources.

## 2.3 Structures de contrôles

### 2.3.1 MemHeaderBlock

Placé au début de la mémoire, ce bloque (de taille 24) permet de stocker la méthode de tri des zones libres ainsi qu'un pointeur vers la première zone libre.

### 2.3.2 MemFreeBlock

Placé au début de chaque zones libres, ce bloque (de taille 24) permet de savoir la taille de la zone libre ainsi que la prochaine zone libre.

### 2.3.3 MemMetaBlock

Placé au début de chaques zone allouées, ce bloque (de taille 8) permet de savoir la taille de la zone allouée.

## 2.4 Problemes rencontrés

- écrasement de la mémoire: Nous avont decidé de fixer une taille minimum d'allocation de 16 octets, afinde palier au probleme de depassement des bornes des blocs lors de liberation/allocation multiples
- stockage de la head (FreeMemBlock)
- alignement: Rust nous oblige a avoir des adresses de memoire multiple de 8, nous avont donc du rajouter une fonction modulo, qui transforme les adresses en multiple de 8

### 2.5 Affichage de la memoire

La logique de parcours et d'affichage de la zone mémoire est implémentée dans la fonction memshow. Pour cette implémentation, nous avons choisi d'utiliser un pointeur mutable. Celui-ci part de l'adresse de début de la mémoire et s'incrémente au fur et à mesure en fonction des blocs rencontrés. Cette approche permet de parcourir efficacement toute la zone mémoire, garantissant que chaque bloc est pris en compte, tout en optimisant le temps de parcours pour s'assurer qu'aucun bloc n'est oublié.

## 3 Tests realisées

Nous avons dû dans un premier temps "traduire" les tests fournis en C vers le Rust. Nous avons ajouté en plus des tests unitaires (pour notre fonction modulo) et des tests pour la fonction realloc. Ces derniers sont disponibles dans le dossier "tests" du projet.

### 3.1 Allocation de taille 0

a0
q

Allocation d'une zone de taille 0.
Résultat attendu :

```
Zone occupee, Adresse : 32, Taille : 16
Zone libre, Adresse : 48, Taille : 127944
```

La zone occupée est un bloc de taille 8, et donc de taille 16 en incluant les metadonnées (alignement a 8)

### 3.2 Allocation complete de la memoire

a127000
a960
q

Allocation totale de la mémoire (en 2 blocs)
Résultat attendu :

```
Zone occupee, Adresse : 32, Taille : 127000
Zone occupee, Adresse : 127040, Taille : 960
```

il n'y a plus de zones libres, la memoire est bien totalement occupé.

### 3.3 Allocation négative

a−1
q

Allocation d'une zone de taille négative.
Résultat attendu :
Erreur : la taille ne peut pas etre négative

### 3.4 Désallocations multiples

```
a32
a16
f1
f1
f1
f1
q
```

Désallocation plusieurs fois de la même zone.
Résultat attendu :

```
Zone libre, Adresse : 24, Taille : 32
Zone occupee, Adresse : 72, Taille : 16
Zone libre, Adresse : 88, Taille : 127904
```

La liberation multiples d'une meme zone ne cause aucun probleme.

### 3.5 Memoire completement vide apres allocation

```
a12568
f1
q
```

Allocation en une zone de la mémoire puis désallocation de celle-ci.
Résultat attendu :

```
Zone libre, Adresse : 24, Taille : 127976
```

### 3.6 Allocation trop grande

```
a128000
q
```

Allocation d'une zone mémoire trop grande par rapport à la taille totale de la mémoire .
Résultat attendu :

```
Echec de l'allocation
```

### 3.7 Alignement de la memoire

```
a7
a8
a9
a17
q
```

Allocation de plusieurs zones mémoires de taille differentes
Résultat attendu :

```
Zone occupee, Adresse : 32, Taille : 16
Zone occupee, Adresse : 56, Taille : 16
Zone occupee, Adresse : 80, Taille : 16
Zone occupee, Adresse : 104, Taille : 24
Zone libre, Adresse : 128, Taille : 127864
```

la taille d'allocation minimum de taille 16

### 3.8 Reallocation plus petite

```
a 64
r 1 32
```

Réallocation d'une mémoire plus petite que celle de départ
Résultat attendu :

```
Apres l'allocation
Zone occupee, Adresse : 32, Taille : 64
Zone libre, Adresse : 96, Taille : 127896

Apres la reallocation
Zone occupee, Adresse : 32, Taille : 32
Zone libre, Adresse : 64, Taille : 127928
```

### 3.9 Reallocation plus grande

```
a 32
r 1 64
```

Réallocation d'une mémoire plus grande que celle de départ
Résultat attendu :

```
Apres l'allocation
Zone occupee, Adresse : 32, Taille : 32
Zone libre, Adresse : 64, Taille : 127928

Apres la reallocation
Zone libre, Adresse : 24, Taille : 32
Zone occupee, Adresse : 72, Taille : 64
Zone libre, Adresse : 136, Taille : 127856
```

## 4 Conclusion

Malgré la réussite de l'implémentation, il apparaît que Rust n'est pas parfaitement adapté à ce type de projet. Son système de sécurité mémoire, bien qu'efficace pour les applications traditionnelles, impose des restrictions qui compliquent la manipulation manuelle de la mémoire. L'utilisation de pointeurs bruts, nécessaire pour un contrôle total, annule en grande partie les avantages de la sécurité de Rust, bien qu'il permette une gestion de la mémoire sans ramasse-miettes, ses mécanismes de sécurité rendent difficile la conception d'un allocateur qui exige une liberté complète sur la gestion des ressources.

## 5 Compilation et éxécution

Pour la compilation et l'éxécution, nous utilisons l'outil "cargo". Le détail des commandes est disponible dans le fichier README.md du projet.