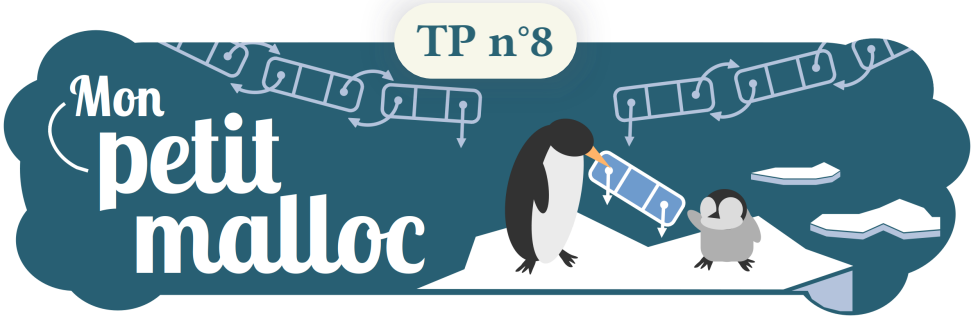
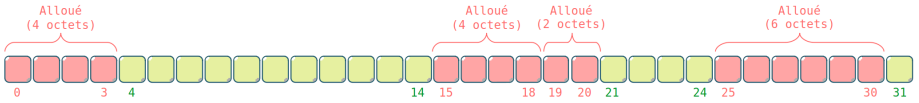


Objectif du TP

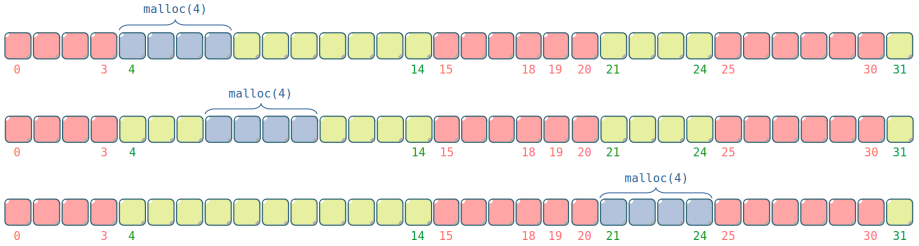
Manipuler des listes doublement chaînées afin de réimplémenter une petite version simplifiée de `malloc`.

Allocateur dynamique de mémoire

Lorsque l'on utilise `malloc` dans un programme en C, tout se passe automatiquement pour nous : on récupère un pointeur vers une zone mémoire contigue, de la taille souhaitée, sans effort. Mais sous le capôt, c'est un allocateur dynamique de mémoire qui travaille. Ce dernier doit se rappeler des zones qui ont déjà été allouées dans le tas, et des zones qui sont libres et pourront être utilisées plus tard. Il doit donc retenir l'état actuel de la mémoire :



À chaque appel de `malloc`, c'est aussi à l'allocateur dynamique de choisir quel emplacement utiliser pour l'allocation. Ce n'est pas toujours évident : ces choix pourront avoir des conséquences importantes dans le futur. Par exemple, si l'on souhaite allouer 4 octets, plusieurs choix s'offrent à nous :



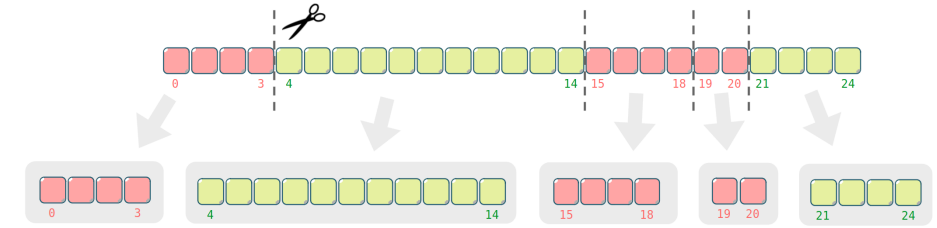
Nous avons donc besoin d'une structure de données permettant de retenir les blocs alloués et les blocs libres de la mémoire, tout en permettant de rapidement mettre à jour celle-ci lors d'une allocation ou une libération.

1 Pourquoi utiliser un grand tableau pour représenter la mémoire et les blocs alloués n'est ici pas une bonne option ? Que pensez-vous des complexités des opérations d'allocation et de libération ?

I. Modélisation par une liste doublement chaînée

La mémoire comme une liste de blocs

Nous proposons ici de représenter la mémoire par des blocs contigus qui peuvent être libres ou alloués.

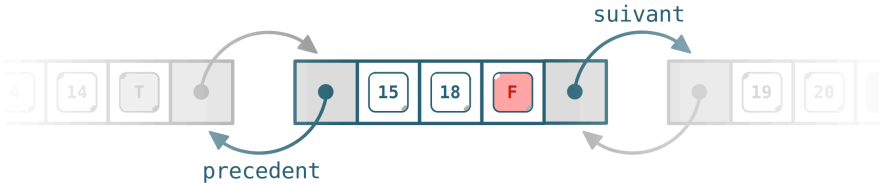


Comme le nombre de blocs n'est pas connu à l'avance, et que nous allons fréquemment insérer de nouveaux blocs, nous allons ici utiliser une structure de liste chaînée. Celle-ci sera **doublement chaînée** pour faciliter la suppression d'un bloc donné.



Le maillon d'une liste doublement chaînée

Dans le cas d'une liste doublement chaînée, un maillon dispose d'un pointeur vers le maillon suivant mais également d'un pointeur vers le maillon précédent. Cela permet de facilement supprimer un maillon donné, car il n'est pas nécessaire de parcourir toute la liste pour retrouver son prédecesseur : on peut y accéder directement.



Pour ce TP, chaque maillon contiendra également deux entiers correspondant aux adresses de début et de fin du bloc (incluses), ainsi qu'un booléen `est_libre` indiquant si ce bloc est considéré comme libre ou alloué.

2 En vous basant sur la structure d'un maillon de liste simplement chaînée définie au TP précédent, proposer et implémenter une structure de maillon de liste doublement chaînée, que l'on nommera `bloc` car il représentera un bloc mémoire.

3 Implémenter une fonction de prototype `bloc* nouveau_bloc(int debut, int fin)`, qui alloue dynamiquement un tel maillon sur le tas, et l'initialise. Ses pointeurs doivent être NULL, il doit être libre et ses valeurs `debut` et `fin` doivent correspondre aux valeurs données en argument.

II. Manipulation de la mémoire

4 Implémenter une fonction de prototype `bloc* nouvelle_memoire(int taille)` qui alloue et renvoie une liste doublement chaînée d'un seul bloc, dont le début est fixé à 0 et la taille correspond à celle donnée en argument. Cette liste chaînée représente notre mémoire entière et ne possède pour le moment qu'un seul bloc libre.

5 Implémenter une fonction de prototype `void libere_memoire(bloc* b)` qui permet de libérer une liste doublement chaînée de blocs représentant notre mémoire. Attention, on parle ici de la vraie libération de la structure de données, il faut utiliser le vrai `free` pour libérer chaque maillon.

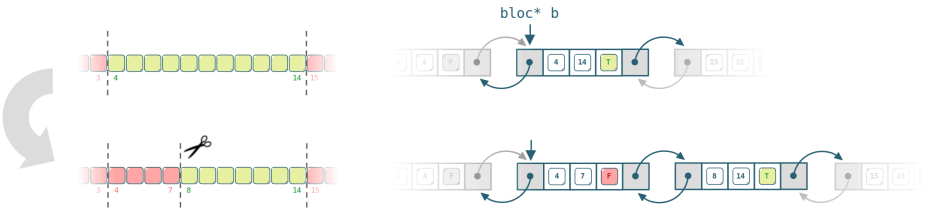
6 Implémenter une fonction de prototype `int taille_bloc(bloc* b)` qui renvoie la taille de l'espace mémoire représenté par un bloc donné. Attention, les bornes `debut` et `fin` sont toutes deux incluses.

7 Implémenter une fonction de protoype `espace_libre(bloc* b)` qui parcourt une liste de blocs représentant une mémoire, et renvoie l'espace libre total qui s'y trouve.

8 Implémenter une fonction de prototype `affiche_memoire(bloc* b)` qui permet d'afficher l'état actuel de la mémoire comme illustré dans ce TP. Vous pouvez copier les émojis suivants : 🟥 et 🟩.

III. Insertion dans une liste doublement chaînée

Allouer dans un bloc libre en le découpant en deux



9 Implémenter une fonction de prototype `void coupe_bloc(bloc* b, int taille)` qui réalise l'opération de découpe du bloc `b` illustrée ci-dessus, en allouant un nouveau bloc et en l'insérant dans la liste doublement chaînée. On pourra s'aider d'un dessin. Combien de pointeurs doit-on modifier ? Que pensez-vous du cas où `b->suivant` est `NULL` ? Quelle est la complexité de cette opération ?

10 Implémenter une petite fonction de test qui alloue une nouvelle mémoire, la coupe quelques fois en affichant les états successifs de la mémoire, puis qui la libère. Vérifier ainsi le bon comportement de vos fonctions implementées jusque-là.

IV. Mon petit malloc

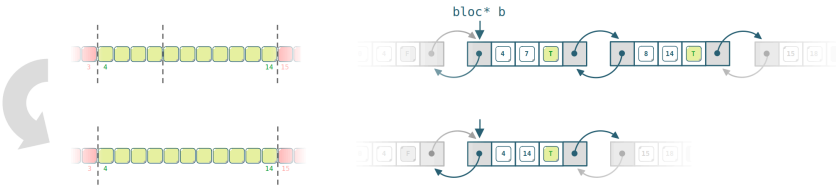
11 Implémenter une fonction de prototype `bloc* mon_petit_malloc(bloc* b, int taille)` qui parcourt la liste doublement chaînée en cherchant le premier bloc libre qui soit plus grand que `taille`. Une fois celui-ci trouvé, on le découpe à l'aide de la fonction `coupe_bloc`, et on renvoie un pointeur vers ce bloc alloué, pour pouvoir le libérer plus tard. Cette stratégie s'appelle First Fit. Si aucun bloc ne convient, on renvoie le pointeur `NULL`, comme le ferait `malloc` si l'allocation échoue.

12 Implémenter une fonction de prototype `void mon_petit_free(bloc* b)`, qui libère un bloc mémoire précedemment alloué, en remettant simplement son booléen `est_libre` de nouveau à `true`.

13 Implémenter un fichier header mon_petit_malloc.h, en y ajoutant les prototypes des fonctions `nouvelle_memoire`, `libere_memoire`, `affiche_memoire`, `mon_petit_malloc` et `mon_petit_free`. Cela permet à un autre fichier .c d'utiliser votre allocateur dynamique, comme mon_petit_test_1.c. Lancez-le : quel problème voyez-vous après avoir réalisé cette suite d'allocations et libérations ?

V. Fusion de blocs

Fusionner deux blocs libres en un seul



14 Implémenter une fonction de prototype `void fusionne_bloc_avec_ses_voisins(bloc* b)` qui permet à un bloc de fusionner avec ses blocs voisins s'ils sont aussi libres. Cela nécessite de libérer un ou deux maillons, sans casser la liste chaînée. Un dessin vous aidera. Mettre à jour `mon_petit_free` pour éviter d'avoir plusieurs blocs libres adjacents suite à une libération.

Le problème de la fragmentation mémoire

Lorsque l'on réalise de nombreuses opérations d'allocation et de libération, on peut atteindre des états ou la mémoire est dite fragmentée, c'est-à-dire avec de nombreux petits espaces inutilisables. Par exemple, il est impossible d'allouer une zone de 4 octets dans la mémoire suivante, alors qu'elle dispose de 12 octets libres.



15 Expérimenter et comparer les stratégies First Fit (choisir le premier bloc assez grand), Best Fit (choisir le bloc assez grand le plus petit) et Worst Fit (choisir le bloc le plus grand) sur des longs enchaînements d'allocations et de libérations, comme ceux proposés dans mon_petit_test_2.c et mon_petit_test_3.c. Comment pourrait-on lutter contre la fragmentation mémoire ?