

Valisp séance 1 : ALLOCATEUR MÉMOIRE

L2 Informatique – Université Côte d’Azur

On suppose que vous êtes venu au CM. Rappel : la mémoire sera vue comme une succession de régions. Une région correspond à une liste de cases en mémoire qui peuvent être libres ou occupées. Les cases sont des mots de 32 bits appelées blocs. Chaque région commence par une case de type `bloc` contenant quatre informations :

1. La marque pour le ramasse-miette (sur 1 bit)
2. l’indice du bloc précédent (sur 15 bits)
3. La disponibilité du bloc (sur 1 bit)
4. l’indice du bloc suivant (sur 15 bits)

Les blocs de début de régions forment ainsi une liste doublement chaînée dans laquelle chaque bloc pointe à la fois vers le précédent et le suivant. Les zones mémoires entre les blocs de début de région seront ignorées par l’allocateur.

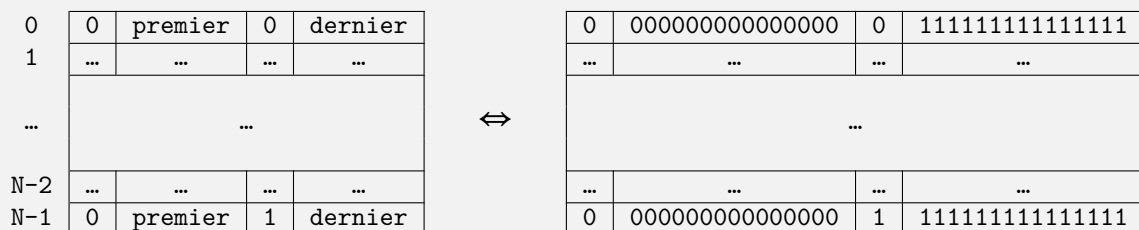
Exercice 1 – Préambule

1. Récupérez sur la page du cours le fichier `valisp.zip` et désarchivez le fichier dans votre répertoire `ProgC/` dans lequel vous faites vos TP de C depuis le début du semestre. Avant de pouvoir compiler, il reste quelques modifications à faire.
2. Commencez par créer les deux fichiers utiles pour ce TP : `allocateur.c` et `allocateur.h`. Dans le fichier `header` il faudra ajouter la déclaration des trois fonctions marquées par le cadre `[.h]` au fur et à mesure que vous les écrivez ; on fera attention à bien mettre les directives `#ifndef ALLOCATEUR_H`. Le reste du TP sera à faire dans le fichier `allocateur.c`.
3. Commencez par créer un type `bloc` correspondant au type `uint32_t`.
4. Créer une macro pour définir la constante `TAILLE_MEMOIRE_DYNAMIQUE` à 2^{15} .
5. Créez de manière globale le tableau `MEMOIRE_DYNAMIQUE` contenant `TAILLE_MEMOIRE_DYNAMIQUE` blocs.
6. Vous pouvez enfin compiler votre projet et exécuter le programme `./valisp`.¹ Le programme lance automatiquement des tests. Le premier échoue logiquement car la fonction `initialiser_memoire_dynamique` n’est pas encore écrite. Vite, passons à l’exercice 2 !

Exercice 2 – Mise en place de la mémoire

Au commencement, la mémoire ne contient que deux blocs situés dans la première et dernière case de notre mémoire. Toutes les autres cases représentent les cases libres de la mémoire que l’on pourra allouer plus tard.

Les adresses étant codés sur 15 bits, notre tableau contient $N = 2^{15}$ cases.



1. `[.h]` Écrivez une procédure `void initialiser_memoire_dynamique()` qui initialise les première et dernière cases de la mémoire avec les bonnes valeurs. N’oubliez pas d’ajouter la déclaration de la fonction dans le fichier `header` `allocateur.h` (comme indiqué par le case `[.h]` en début de question)
2. Relancez le programme `./valisp` puis tester votre fonction. Corrigez si nécessaire avant de passer à l’exercice 3.

1. Le projet utilise la bibliothèque GNU readline. Sous Debian/Ubuntu, vous pouvez l’installer avec `sudo apt install libreadline-dev`

Exercice 3 – Création des blocs

On rappelle que chaque bloc est sur 32 bits de la forme

b0	precedant	b1	suivant
----	-----------	----	---------

1. Écrivez la fonction `bloc_cons_bloc(int rm, int precedant, int libre, int suivant)` qui construit un entier de 32 bits en se basant sur quatre entiers de tailles respectives 1 bit, 15 bits, 1 bit et 15 bits.
2. Écrivez maintenant les quatres fonctions inverses permettant de décomposer un bloc. En paramètre, les fonctions utilisent l’indice d’un bloc dans la mémoire dynamique créée globalement dans l’exercice précédent.
 - (a) `int bloc_suivant(int indice)` (c) `int usage_bloc(int indice)`
 - (b) `int bloc_precedant(int indice)` (d) `int rm_bloc(int indice)`
3. Écrivez une fonction `int taille_bloc(int indice)` qui calcule l’espace disponible dont le bloc d’indice `i` est responsable. L’unité sera le nombre de cases dans le tableau.
4. Nous allons pouvoir commencer à jouer avec le REPL (une interface interactive dans laquelle on peut entrer des commandes et observer le résultat). Dans la fonction `tp1_main()` du fichier `tp1.c`, ajoutez avec le `return` 0 un appel à la fonction `repl_tp1()`. Faites `@mem` pour afficher le détail de la mémoire. Vous êtes censée observer un résultat de deux lignes comme ci-dessous.

```
0 → [ 0 ] → 32767 [ ] 0x56068aeab9e0 [ 0 ] (32766)
0 → [32767] → 32767 [x] 0x56068aecb9dc [32767] (0)
```

Chaque ligne affichée correspond à une région mémoire. Les trois premières valeurs correspondent respectivement au bloc précédent, au bloc courant (entre crochets) et au bloc suivant. La notation `[]` ou `[x]` indique si la région est libre ou non. Le reste de la ligne indique l’adresse mémoire du premier bloc de la région, rappelle son indice et précise la taille de la région entre parenthèses.

5. Pour voir le reste des directives (les commandes du REPL commençant par `@`) faites `@aide`. Pour pouvoir plus tard de tester vos fonctions, vous pouvez charger une image prérempli de la mémoire en tapant `@load exemple.img`. Affichez le résultat avec `@mem`. Combien y a t’il de régions libres ? Quelles sont leur tailles et leur indices ?

Exercice 4 – La fonction malloc

Nous pouvons maintenant implémenter notre propre `malloc` qui devra avoir la même signature que la fonction d’origine : `void * malloc(size_t size)`. Le type `size_t` est un type d’entier permettant de coder une taille de tableau (codé concrètement avec la même longueur qu’un pointeur, 64 bits sur une machine moderne).

1. Écrivez la fonction `int rechercher_bloc_libre(int nombre_blocs)` qui parcourt la liste doublement chaînée des blocs et trouve une région libre de taille suffisante pour contenir `nombre_blocs` cases. On renverra -1 si la mémoire ne contient aucun espace suffisamment grand et l’indice du premier bloc de la région sinon.
2. En déduire une fonction `int allouateur_balloc(int nombre_blocs)` qui alloue dans la mémoire une région libre de taille `nombre_blocs`. On renverra là aussi -1 en cas d’échec. Vous pouvez tester votre fonction avec la directive `@balloc` (par exemple `@balloc 12` pour allouer 12 blocs).
3. `malloc` utilise un paramètre `size` en octets quand notre tableau est organisé en blocs. Écrivez une fonction `int octets_vers_blocs(size_t size)` qui renvoie le nombre de cases minimale pour allouer `size` octets.
4. `.h` Déduire des deux fonctions précédentes la fonction `void *allouateur_malloc(size_t size)` qui renvoie un pointeur (et non un indice) vers la zone de mémoire trouvée par la fonction précédente et `NULL` s’il n’y a pas assez d’espace disponible. Le pointeur ne devra pas renvoyer vers le bloc utilisé par l’allouateur mais sur le suivant. Vous pouvez tester votre fonction avec la directive `@malloc`.

Félicitation, vous avez atteint l’objectif de ce TP. l’implémentation de `free` est facultative. En effet, la mémoire sera libérée automatiquement sans qu’il soit nécessaire d’y faire appel.

Exercice 5 – La fonction free

1. Écrivez la fonction `void allouateur_bree(int i)` qui libère la région dont le premier bloc a pour indice `i`. On fera attention de fusionner si nécessaire le bloc avec les blocs libres adjacents. On pourra tester avec `@bree`.
2. `.h` En déduire `void allouateur_free (void *ptr)` qui libère la région correspondante à l’adresse fourni en paramètre. Attention `ptr` pointe sur le deuxième bloc de la région (le premier étant utilisé par l’allouateur). À tester avec `@free`.