R5.04 - Qualité algorithmique

TP 3 - Gestion de la mémoire

2025-2026

Gestion de la mémoire

Pour le moment, nous n'avions vu que des méthodes d'allocation **statique** par le biai de déclaration d'objets plus ou moins complexe.

Une première difficulté d'une telle allocation est qu'il est nécessaire de connaitre la taille mémoire de l'objet dès le début. Prenons par exemple un tableau d'entier. Il est nécessaire de connaitre le nombre d'éléments du tableau pour le créer, par exemple comme ceci :

```
int tab[10];
```

La taille du tableau sera fixe au cours de l'exécution.

De plus, l'allocation statique utilise une partie de la mémoire du processus exécutant le programme appelée la *pile*. La difficulté est que la mémoire ainsi allouée n'est disponible qu'au cours de l'exécution de la fonction ayant initiée cette allocation mémoire. A la fin de l'exécution, toutes les variables ainsi crées ne sont plus utilisable.

A l'inverse, **l'allocation dynamique** permet d'allouer de la mémoire dans le **tas**, qui est accessible durant toute l'exécution du processus. Ce type d'allocation permet également de réallouer de la mémoire offrant ainsi plus de flexibilité et une gestion mémoire au plus près des besoins.

La commande permettant d'allouer dynamiquement de la mémoire est la commande `malloc()` dont voici le prototype complet.

```
void *malloc(size_t size);
```

Le paramètre `size` nous permet de préciser la quantité de mémoire (en octet) à allouer. En retour, nous obtenons un pointeur vers la zone mémoire allouée.

> **Note**
>
> La fonction 'sizeof(type)' permet de connaitre la quantité mémoire à utiliser pour un type donné. Elle est souvent utilisée conjointement à 'malloc'.

On notera que la mémoire allouée avec `malloc()` n'est pas initialisée! Son contenu n'est pas déterminé.

Une fois que l'objet créé n'est plus utilisé, il est nécessaire de désallouer la mémoire qui lui était réservé. Pour cela, il faut utiliser la commande suivante.

```
void free(void * ptr);
```

On fournit simplement en paramètre le pointeur vers la zone mémoire à libérer.

D'autres fonctions utiles existent également (comme `realloc()` ou `calloc()`). Vous pouvez consulter le `manuel` pour plus d'informations.

1. Récupérer sur celene le module `liste_dyn.c` qui fournit une implémentation des listes chainées en utilisant les mécanismes d'allocation dynamique. Repérer l'utilisation de `malloc()` et de `free()`. Analyser les différences avec la version que vous avez implémenté précédemment, uniquement avec de l'allocation statique.

Un premier test

L'étape suivante consiste à vérifier que la gestion de la mémoire a correctement été mise en place. Pour cela nous utiliserons l'outil `valgrind`.

L'option `memcheck` va vérifier l'utilisation du tas (mémoire dynamique) et détecter d'éventuelles erreurs mémoire. On retrouve par exemple :

— lire une zone mémoire non initialisée
— lire ou écrire une valeur dans une zone mémoire non allouée
— libérer une zone mémoire déjà libérée
— une zone mémoire n'est plus référencée par un pointeur (mémoire "perdue")

Pour plus de détails sur `valgrind`, vous pouvez bien sur lire sa documentation. Le livre **dive into system** permet également de se familiariser avec cet outil.

La commande à exécuter est la suivante :

```
valgrind --tool=memcheck --leak-check=full ./prog
```

1. Récupérer le fichier `valgrind.c` sur Celene, compilez le puis lancer `valgrind`. Tentez de repérer dans le rapport les différents types d'erreurs.

Occupation mémoire

Nous allons maintenant analyser à l'aide de différents outils la consommation mémoire réelle

Mesurer l'occupation mémoire du tas avec valgrind Massif

Le programme `valgrind` fournit également un outil nommé `Massif` permettant de visualiser l'utilisation du tas au cours de l'exécution.

Pour lancer le programme :

```
valgrind --tool=massif prog
```

`massif` va générer un rapport nommé `massif.out.pid` où `pid` correspond au pid de votre programme.

Pour pouvoir interpréter ce rapport, il faut l'annoter :

```
ms_print callgrind.out.pid
```

1. Réalisez un suivi de l'utilisation du tas avec l'outil `Massif` pour votre programme.
2. En vous aidant la documentation officielle, repérez dans le rapport la quantité maximum de mémoire utilisée par votre programme.

Analyser l'utilisation mémoire total d'un processus

L'analyse du tas ne permet pas de visualiser l'ensemble de la consommation mémoire, mais uniquement le résultat de l'allocation dynamique (avec `malloc/calloc/realloc/free`). Si l'on veut une vue globale de la consommation mémoire, d'autres outils sont nécessaires. Les exercices suivants sont fortement inspirés du livre Operating Systems: Three Easy Pieces, au chapitre 13.

1. Un premier outil disponible sur les systèmes unix est la commande `free`. Exécutez la commande `free -m` et observez la consommation mémoire de votre système.
2. Modifiez ensuite votre programme afin d'y intégrer une boucle infini. Compilez votre programme ainsi modifié, exécutez-le puis relancez ensuite la commande précédente. Quels changements constatez-vous ?
3. Utilisez maintenant la commande `pmap -X` et tentez de déterminer à partir de la sortie générée la proportion de la mémoire associée au tas, à la pile et au code de votre programme.