

TP10 C Memoire

Published on 2022-01-06 by **Marc de Falco**

Remarque On a fait du 0CamL récemment, vous retournez au C. Comme à chaque changement de contexte, vous allez prendre beaucoup de temps. C'est normal. Plus on alternera souvent, plus vous retrouverez vos repères rapidement.

Développement en C

Environnement

Pour ce retour au C, je vous propose d'utiliser :

- soit votre ordinateur directement avec un éditeur de texte et gcc ou clang dans le terminal pour compiler.
- soit dans la machine virtuelle
- soit, et je pense que c'est une bonne solution, directement depuis info.classeprepa.net :
 - Dans le menu en haut à droite New vous choisissez Terminal
 - Il suffit de créer un répertoire pour votre tp et un fichier C :
 - `mkdir tp10`
 - `cd tp10`
 - `touch main.c`
 - Ensuite, naviguez vers le fichier `main.c` vide dans le Jupyter, cliquez dessus et vous avez un éditeur, il suffit de faire Sauvergarder ou Ctrl-S puis de compiler.

Compilation

Pour compiler, on va directement lancer le compilateur dans le terminal. Pour cela, en tenant compte du fait que votre fichier est `main.c`. On pourra écrire `gcc -o main main.c` puis `./main` pour le lancer.

Vous risquez d'avoir des programmes qui font plein d'erreurs de mémoire ou de crash. Je vous recommande donc de compiler avec la commande suivante :

```
gcc -Wall -Werror -g -fsanitize=address -o main main.c
```

Découpons cette commande :

- gcc est le compilateur, on peut le remplacer par clang qui est plus récent
- -Wall signifie qu'on active tous les avertissements warning.
- -Werror signifie qu'on traite les avertissements comme des erreurs de compilations.
- -g qu'on compile avec les informations de débogage, c'est-à-dire le lien entre votre code source et l'endroit où une erreur se produit.
- -fsanitize=address signifie qu'on active le vérificateur d'erreurs de gestion mémoire.

Comme c'est un peu long à taper, vous pouvez tirer partie de l'historique du shell, c'est-à-dire qu'en utilisant la touche fléchée vers le haut, on retrouve la commande précédente, en appuyant deux fois, la commande qui précède la précédente et ainsi de suite. Il suffit alors d'appuyer sur la touche Entrée pour exécuter la bonne commande ainsi retrouvée.

Rendu et tests

Pas de rendu pour ce TP et pas de tests automatiques. Je vais vous demander de faire vos propres tests progressivement. La démarche sera alors d'écrire les fonctions dans votre fichier `main.c` et de rajouter des tests dans votre fonction `main`, éventuellement, ces tests seront groupés par thème dans des fonctions de tests comme dans les TP précédents.

Tableaux non statiques

On va définir ici une structure de donnée pour gérer des tableaux dont la taille ne sera connu qu'à l'exécution. Pour simplifier, ce seront des tableaux d'entiers `int` mais la méthode s'adapte naturellement pour des tableaux de n'importe quoi.

Type array

On définit le type suivant :

```
struct array {  
    int *elements;  
    unsigned int size;  
};  
typedef struct array array;
```

Un array est donc une structure qui contient :

- un pointeur vers un tableau `elements` d'entiers
- un entier indiquant la taille de ce tableau

Remarque importante quand on copie un pointeur, on copie uniquement un entier qui est l'adresse pointée. Ainsi, quand on copie un array, ce n'est finalement qu'un couple d'entiers qu'on copie. C'est pourquoi, dans la suite, on passe tous les array par valeur. C'est-à-dire que pour définir une fonction de recherche d'un élément dans un array on va écrire :

```
int search(array t, int x)
{
    for(int i = 0; i < t.size; i++)
    {
        if(t.elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

Quand on va appeler la fonction, on va avoir une copie de l'argument, c'est-à-dire du array, mais le tableau elements lui sera toujours à la même place. L'alternative serait de passer les valeurs array par pointeurs comme dans la variante suivante :

```
int search(array *t, int x)
{
    for(int i = 0; i < t->size; i++)
    {
        if(t->elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

On n'a rien à gagner au surplus de complexité induit par cela. L'unique avantage serait de permettre de modifier les paramètres de la structure passée en argument, mais on réservera ça aux fonctions qui le nécessitent.

Allocation, Initialisation, Libération

A l'aide de malloc et free (dans stdlib.h vous pouvez taper man malloc pour avoir la doc), on vous demande

Question 1

Écrire une fonction de prototype array array_alloc(unsigned int size) qui alloue un tableau de size entiers avec malloc et renvoie le array pointant dessus.

Conformément à ce qui est écrit au-dessus, on peut créer un array en variable locale et le renvoyer avec return, ce qui compte c'est la mémoire pointée.

Question 2

Écrire une fonction de prototype void array_free(array t) qui libère le tableau pointé par t avec free. A partir de ce moment, on ne peut plus utiliser cette adresse sans faire d'erreurs.

Question 3

Le tableau alloué n'est pas initialisé, écrire une fonction de prototype array array_make(unsigned int size, int def) qui alloue un tableau et initialise toutes les valeurs de celui-ci à def.

Question 4

Il est possible de mesurer le temps pris par un programme à l'aide de la fonction time :

```
$> time ./main
./main  0.38s user 0.00s system 99% cpu 0.379 total
```

Comparer le temps d'exécution d'un programme qui

- alloue puis libère 100 fois un tableau d'un million d'entiers
- alloue *en initialisant* à 0 avec make puis libère 100 fois un tableau d'un million d'entiers.

Que peut-on en conclure ?

Explorons la libc

Il existe d'autres fonctions que malloc comme

```
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

qui permettent :

- pour `calloc` d'allouer un tableau de `nmemb` membres qui sont chacun de taille `size` et **d'initialiser les octets à 0**. Cette fonction prend donc un temps linéaire en le nombre d'octets.
- pour `realloc` de déplacer en mémoire un tableau en changeant sa taille (en plus ou en moins). Si jamais il y a de la place on ne bouge pas le tableau. C'est donc forcément plus efficace que de le faire à la main.

Ces deux fonctions sont **hors programme** !

Affichage, copie

Question 5

Écrire une fonction `void array_print(array a)` qui affiche `[a1,a2,...,an]` si `a` contient les éléments de `a1` à `an`.

Remarque cette fonction vous sera utile pour tester ce que vous avez fait dans la suite.

Question 6

Écrire une fonction de prototype

```
void array_blit(array src, int src_start, array dst, int dst_start, int len)
```

qui copie `len` éléments depuis le tableau `src` à partir de l'indice `src_start` dans le tableau `dst` à partir de l'indice `dst_start`.

Remarque on a déjà écrit cette fonction en 0Cam1, vous pouvez vous en inspirer.

Implémentation d'une pile non bornée

Question 7

Écrire une fonction de prototype `void array_push(array *a, int x)` qui rajoute `x` à la fin du array `a`, c'est-à-dire :

- on va allouer un nouveau array `b` dont le tableau contient `a->size+1` éléments
- on va copier les anciennes valeurs de `*a` dans `b`
- écrire `x`
- libérer `*a`
- remplacer avec `*a = b`.

Remarque On peut tout à faire écrire `*a = b` pour que `a` ait les mêmes valeurs que `b` en dehors de la fonction;

Rappel pour une `struct` par pointeurs, on utilise `->` pour accéder aux champs. Donc `a->elements` et `a->size` ici.

Remarque Faites des tests comme en créant un tableau de taille 0 et en empilant les entiers de 1 à 10 et en affichant le résultat. Faites les tests avec `sanitize` et tout.

Question 8

Écrire une fonction `int array_pop(array *a)` qui retire le dernier élément du tableau `a` et renvoie sa valeur. Il faudra donc :

- allouer un nouveau array `b` dont le tableau contient `a->size-1` éléments
- copier les anciennes valeurs sauf une
- récupérer dans une variable `x` la valeur de la dernière case de `a` (*parce qu'il n'est pas possible d'y accéder après l'étape suivante*)
- libérer l'ancien tableau
- remplacer avec `*a = b`.

Tableaux dynamiques

Question 9

Reprendre toutes les questions précédentes en changeant la structure de donnée pour permettre de faire des tableaux dynamiques. Il faudra ainsi avoir deux entiers : `int p_size` qui contiendra la taille *physique* en mémoire et `int l_size` qui contiendra la taille *logique* c'est-à-dire les éléments signifiants stockés.

On fera des tests !