TRAVAUX PRATIQUES X

Allocation mémoire & Tableaux dynamiques

On rappelle que tous les codes doivent être compilés avec les options `-Wall` , `-Wextra` , `--std=c17` , et que `-fsanitize=undefined` est conseillé. On rappelle que le compilateur a généralement raison lorsqu'il donne un avertissement : vous devez compiler sans avertissement. On rappelle aussi que vos fonctions doivent être *testées*.

A Allocation mémoire

A.1 `sizeof` et `size_t`

Le type `size_t` est un type d'entiers non-signé assez grand pour parler de la taille (en nombre d'octets) de n'importe quel objet représentable en mémoire. Dans l'idée, `size_t` est assez grand pour que le nombre d'octets de votre RAM y soit stockable.

Si toutefois vous utilisez à la place un type entier plus petit, C devrait faire automatiquement les conversions *ad hoc* . Ne vous prenez donc pas la tête avec cela.

Définition 1 : sizeof

`sizeof` est un opérateur du langage C qui prend « en argument » un type ou une variable. À la compilation, le compilateur va remplacer le `sizeof` par la taille en nombre d'octets de ce type / cette variable.
On dit donc que `sizeof` est *statiquement* calculé.
`sizeof` « renvoie » une taille, c'est à dire un `size_t` .

Exemple : `sizeof(int)` est probablement remplacé par `4` sur votre machine. Probablement, car il est seulement spécifié que `int` doit faire au moins deux octets, mais sur presque tous les systèmes modernes il en fait excatement 4 (32 bits).
Remarque : `sizeof` N'est *pas* une fonction, car il est calculé à la compilation et non à l'exécution ! D'où les guillements précédents sur « argument » et « sortie ».

A.2 Rappels et compléments sur les pointeurs

Rappel : un pointeur est une adresse mémoire. Étant donnée une variable `n`, on peut obtenir un pointeur sur `n` avec l'opérateur `&` (« *address of* ») : `&n` .

Étant donné un pointeur `p` pointant vers une variable, on peut accéder à cette variable en déréférençant avec `*` : `*p` .

À partir d'aujourd'hui, on va complexifier en créant des pointeurs qui ne pointent plus vers une variable mais vers une zone du Tas mémoire où sont rangées plsuieurs données/variables. L'idée reste cependant la même : on manipule des *adresses*.

Fondamentalement, une adresse n'a pas de type supplémentaire : ce n'est qu'une adresse mémoire. Ainsi tous les types `int*` , `double*` , `bool*` , etc fonctionnent de la même façon : ce ne sont que des adresses, et celles-ci fonctionnent pareil pour toutes les données [1].

Connaître le type de ce qui est pointé permet « juste » [2] de pouvoir déréférencer correctement en sachant combien de bits occupe la donnée pointée (un `char` n'est pas aussi long qu'un `int` !) et comment les interpréter lorsque l'on déréférence. Si l'on écrit `3 + *p` , savoir comment interpréter les bits de `*p` est nécessaire.

Nous sommes toutefois parfois amenés à parler d'adresse sans connaitre ou sans préciser le type pointé. On utilise pour cela le type `void*` :

1. Métaphore immobilière : l'adresse d'une maison ne dépend pas de la couleur de sa façade ou de son nombre d'étages. Une adresse c'est une adresse, point.
2. C'est un gros «juste»...

Définition 2

`void*` est un type de pointeurs. C'est le type des pointeurs qui ne donne aucune information sur la donnée pointée.

On peut croiser ce type dans deux grandes situations :

— si l'on n'a pas d'autre information que l'adresse de la donnée.

— si l'on veut faire une fonction très générique qui marche pour n'importe quel pointeur : on fait alors une fonction qui prend un pointeur sans plus d'informations, donc un `void*` .[3]

Par exemple, un programmeur C expérimenté pourrait faire une fonction `void swap(void* a, void* b, size_t taille)` qui échange deux données de même taille, donnée par `taille` (en octets) et situées respectivement aux adresses `a` et `b` .

Définition 3 : Transtypage

On peut demander à C de considérer un pointeur « avec information de type » comme un `void*`, avec une simple conversion de type `(type) var` . La syntaxe est donc la suivante :

C
```c
type0* p; // Soit p un pointeur avec infos. Ici, un ptr vers type
...
type1* p_bis = (type1*) p; /* p_bis pointe vers la même adresse
                           mais fait comme si ce qui est pointé
                           est de type type1. */
...
```

Ainsi, si `p_void` est un pointeur de type `void*` , alors `(type*) p_void` est un pointeur vers la même adresse mais qui considère que la donnée pointée est de type `type` .
On appelle ces conversions des *transtypages* .

Remarque : les transtypages en C permettent de faire bien plus que cela, mais le programme se restreint explicitement aux transtypages vers et depuis `void*`.

Il existe un pointeur spécial `NULL` , défini dans `stdlib.h` . C'est le pointeur qui pointe vers rien du tout, une sorte de "0" pour les pointeurs. Il peut par exemple servir à détecter des erreurs (si on renvoie l'adresse NULL c'est qu'il y a eu erreur), à initialiser un pointeur (c'est une valeur valide pour un pointeur et il est clair pour le programmeur que c'est une valeur d'initialisation car ce n'est l'adresse de rien du tout), ou à marquer la fin d'une structure de données (un peu comme `'\0'` marque la fin d'une chaine de caractères)

A.3 Allocation sur le tas

C permet de créer des objets à durée de vie allouée : il s'agit d'allouer une zone mémoire d'une taille donnée sur le tas. Charge alors au programmeur de gérer lui-même le contenu de cette zone, et de **libérer la mémoire** avant la fin du programme.

Définition 4 : malloc

La fonction `void* malloc(size_t size)` (**m**emory **alloc**ation) de `stdlib.h` permet de réserver `size` octets consécutifs en mémoire. Cette allocation a lieu sur le tas. La fonction renvoie un pointeur `void*` vers le début de la zone réservée.

La fonction `malloc` est un très bon exemple d'utilisation de `void*` pour manque d'information : elle ne demande pas le type des données que l'on mettra dans la zone, elle demande simplement le nombre d'octets. Elle n'a donc pas assez de connaissances pour renvoyer autre chose qu'une adresse sans informations de type.

Un exemple :

C
```c
int* p = (int*) malloc(size_of(int)); /* p pointe vers une zone exactement assez grande
                                      pour un entier */
  *p = 42; /* on peut donc manipuler p comme un pointeur vers un entier ! */
```

1. Écrire une fonction `unsigned int* alloue_int(unsigned int init)` qui alloue sur le tas (à l'aide de `malloc`) une zone mémoire juste assez grande pour un entier `unsigned int`, initialise cette zone à la valeur `init` puis renvoie un pointeur vers cette zone. Testez-la.

3. C'est ce qui s'approche le plus du polymorphisme en C .

Si `*p` est un pointeur vers une zone assez grande pour contenir plusieurs valeurs, alors `p[0]` est la première de ces valeurs, `p[1]` la seconde, etc etc. (Cf exemple fait en classe.)

A.4 Libération

Un grand pouvoir implique de grandes responsabilités. Puisqu'il ordonne lui-même l'allocation, le programmeur doit ordonner lui-même la libération de la mémoire en question. S'il oublie de le faire, alors le programme peut avoir une utilisation excessive de la RAM : il alloue de la mémoire, oublie qu'il l'a allouée, puis en alloue encore, etc etc, jusqu'à ce que la RAM soit pleine à craquer. On parle de fuites de mémoire [4].

Proposition 5 : free

La fonction `void free(void* ptr)` prend en argument un pointeur vers une zone allouée et libère la zone en question. Il n'est pas nescessaire de transtyper explicitement son argument vers `void*`.

Toute mémoire allouée doit être libérée.

Attention toutefois à ne pas libérer trop tôt : une fois libérée, les données contenues dans une zone peuvent tout à fait être supprimée. La mémoire peut même appartenir à quelqu'un d'autre. En bref, un pointeur vers une zone libérée doit être considéré comme invalide et à ne plus utiliser.

2. Reprenez votre test de la fonction `alloue_int` . Faites en sorte de libérer toute la mémoire allouée une fois les tests finis..
3. Répétez après moi : « *Toute mémoire allouée doit être libérée.* » . Oublier de libérer la mémoire est une erreur commune et couteuse.

Parenthèse historique

En 1969, Appolo 11 envoie des hommes sur la Lune. À son bord, trois êtres humains s'apprêtant à entrer dans la légende, mais aussi un ordinateur de bord qui doit s'assurer qu'ils rentrent (sains et saufs). Cet ordinateur avait 4ko de RAM. Il fonctionnait si bien qu'il a réussi à fonctionner malgré un dysfonctionnement grave de la part d'un instrument de mesure qui donnait de fausses informations.

On a envoyé des humains sur la Lune avec 4ko de RAM…
Quatre. Kilo. Octets.
Sur. La. Lune.
Ne vous plaignez *jamais* du manque de mémoire de vos machines.

L'ingénieure en chef du développement logiciel de la mission était Margaret Hamilton. C'est une pionnière et une légende de l'informatique [5] [6]. Ce TP étant déjà bien assez long, je n'ai pas la place de lister les contributions et les récompenses de Margaret Hamilton. Pour faire court : il y a un avant et un après Hamilton dans le développement logiciel.

4. Les utilisateur·rices de certains navigateurs internet ne me comprendront que trop bien. Ou de jeux vidéos. Ou de… la liste est trop longue.
5. Et elle est encore en vie ! L'informatique est une science encore assez jeune pour que vous puissiez un jour croiser des fondateurs. C'est hélas aussi une science assez vieille pour que vous croisiez des nécrologies de fondateurs.
6. Note culture G : la composante logicielle de l'informatique était à l'époque d'Hamilton méprisée par les hommes (masculins). C'est pour cela que l'on a « toléré » qu'Hamilton étudie la programmation, dans un milieu très sexiste et discriminant, car ce n'était pas considérée comme une « vraie » science. Et même ainsi, Hamilton a dû se battre et se démener.

B Tableaux dynamiques

Passons maintenant au coeur de ce TP : implémenter des tableaux dynamiques.
Pour les représenter, on utilisera le type suivant :

C : dynarray.h

```c
struct dynarray_s {
 int* tab; // pointeur vers les cases du tableau dynamique
 int n;   // nombre de cases du tableau (utilisées ou non)
 int len; // nombre de cases utilisées du tableau
};
```

.

Comme nous l'avons vu la semaine dernière, cela défini un type structure nommé `struct dynarray_s`. On voudrait le renommer. Pour cela, on utilise :

C : dynarray.h

```c
typedef struct dynarray_s dynarray;
```

.

Cela crée un nouvel identifiant de type, nommé `dynarray`, qui correspond au même type que `struct dynarray_s`.

C Librairie

Dans ce TP, vous allez principalement coder dans le fichier `dynarray.c` . Vous y implémenterez les tableaux dynamiques. Le but est qu'ensuite n'importe quel code puisse venir « utiliser » les fonctions de `dynarray.c` : on dit que l'on crée une librairie.

4. Vous avez déjà utilisé des librairies. Par exemple, `stdlib` est une librairie (d'où le `lib` dans son nom). Nommez d'autres librairies que vous utilisiez déjà.

Le fichier `dynarray.c` ne doit *pas* contenir de `main`. Il doit uniquement contenir des fonctions de manipulation des tableaux dynamiques. Le `main` du jour est rangé dans... `main.c`. Il est pré-codé : il se contente d'appeler les fonctions de `dynarray` et de les coder.

Rappel : un programme C doit avoir un et un seul `main`. Si `dynarray` ET le fichier qui l'appelent avaient tous deux un `main`, il y aurait un problème.

Quand un code appelle on librairie, on dit qu'il est **code client** de cette librairie.

C.1 Header

En plus des `.c` que vous reconnaissez, il y a un petit nouveau : `dynarray.h`. C'est un `header`, aussi appelé **interface**. Dedans, sont rangés :

- Les inclusions de librairies dont `dynarray.c` et un code qui s'en sert a besoin.
- Les déclarations de type dont `dynarray.c` et un code qui s'en sert a besoin.
- Le prototype des fonctions implémentées par `dynarray.c`, accompagnés de leur documentation. Remarquez que cela ne donne aucune information sur *comment* les fonctions sont implémentées !
  Si jamais on écrit une fonction dans le `.c` mais qu'on ne la mentionne pas dans son interface, elle n'est pas utilisable par un code client : elle est « cachée ».

À retenir : l'**interface** (`.h`) simplifie et abstrait l'implémentation (.c). C'est le fichier que l'on va lire pour savoir ce qu'une librairie propose.
Votre travail dans ce TP est donc d'implémenter la librairie de tableaux dynamiques décrite par le `.h`.

C.2 Compilation

Pour compiler, on pourra simplement compiler ces différents fichiers ensemble :

```
gcc -o exe dynarray.c main.c -Wall -Wextra --std=c17 -fsanitize=undefined
```

Plus tard dans l'année, nous verrons comment compiler ces fichiers *séparément*.

C.3 Petit complément sur les tableaux

Pour prendre un tableau en argument sans préciser son nombre de cases en toutes lettres (i.e. pour ne pas écrire `type f(type T[10000])`), on peut utiliser `T[]` (i.e. écrire `type f(type T[])`). Cela permet de prendre un tableau de n'importe quelle taille.

Nous en reparlerons au S2, mais cela peut vous servir si vous avancez assez dans le TP.

C.4 C'est parti !

Sauf mention contraire, vous devez coder dans `dynarray.c` . Dans tout ce TP, on manipulera des pointeurs vers des tableaux dynamiques.

Nous allons commencer par créer et supprimer des tableaux dynamiques. Il sera très confortable qu'un tableau ait un nombre de cases qui soit une puissance de deux.

5. Écrire une fonction `uint64_t puiss_2(int n)` qui renvoie la première puissance de deux supérieures à `n`.
   On pourra exceptionnellement faire des comparaisons entre entiers de type distincts. On admet qu'il y a conversion implicite, et qu'elle ne devrait pas poser (trop) de problèmes.

Nous sommes parés pour le début !

6. Complétez la fonction `dyn_create` . Elle prend en argument `len` et `x`. Elle doit renvoyer un pointeur vers un tableau dynamique contenant au moins `len` cases, toutes initialisées à `x`.
   On allouera en fait `puiss_2(n)` cases, afin que le nombre de cases soit une puissance de deux.
   On fera attention à ne pas renvoyer un pointeur vers de la mémoire morte !!!!! Il faut peut-être faire plus d'un appel à `malloc`...
7. Complétez la fonction `dyn_free` . Elle prend en argument un pointeur vers un tableau dynamique, et doit libérer *toute* la mémoire allouée pour un tableau dynamique.

Dorénavant, je ne décrirai plus systématiquement les entrées/sorties des fonctions : ce serait paraphraser le header.

8. Complétez la fonction `dyn_affiche` qui affiche les éléments d'un tableau dynamique. Testez pour vérifier (vous pouvez voir les tests dans `main.c` ; n'hésitez pas à en ajouter ou à en enlever).
9. Complétez la fonction `dyn_len` . Comme toujours, testez.
10. Complétez la fonction `dyn_acces` qui renvoie la valeur d'une case d'un tableau.
    Vous utiliserez `assert` pour garantir que la case à laquelle on veut accéder est une case valide (c'est à dire que l'indice donné est bien l'indice d'une case remplie).
11. Complétez la fonction `dyn_remplace` qui modifie une case d'indice donné.
    Vous utiliserez à nouveau `assert` pour vous garantir la cohérence de la requête.

Passons au plat de résistance.

12. Complétez la fonction `dyn_ajoute` qui ajoute un élément à un tableau dynamique. Vous devrez doubler le tableau dynamique s'il n'est pas assez grand.
    Pour cela, vous allouerez une zone deux fois plus grande (qui servira à remplacer `tab`), recopierez l'ancienne dans la nouvelle, modifierez le tableau dynamique pour qu'il utilise dorénavant la nouvelle, et supprimez l'ancienne.
    Testez !
13. Complétez la fonction `dyn_retire` . Elle doit retirer le dernier élément et le renvoyer. Vous appliquerez la contraction vue en TD.

Et pour terminer :

14. Écrire les derniers fonctions décrites dans `dynarray.h` que vous n'auriez pas encoreimplémentées.