Adresses mémoire et allocation dynamique

Commençons par rappeler l'existence des options `-Wall` et `-Wextra` pour `gcc` . En ajoutant ces options à la compilation, on obtient de meilleurs avertissements qui aident à déboguer.

A La notion de pointeurs

Fondamentalement, votre mémoire RAM est un immense « tableau » de taille environ 2Go. [1] Une adresse mémoire est un « indice » de ce tableau. Nous n'avons jamais manipulé ces adresses directement : nous manipulions des identifiants (noms de variables), sans avoir besoin de se demander où ces variables étaient rangées dans la mémoire.

A.1 Pointeurs

Définition 1 (Pointeur).
Un pointeur est une adresse mémoire. Le nom provient du fait que l'on peut dessiner une adresse comme une flèche qui « pointe » vers une case.
Soit `type` un type. Le type des pointeurs qui pointent vers une case mémoire contenant un élément de type `type` est `type*` . Par exemple, un pointeur pointant vers le contenu d'une variable `int` est de type `int*` .

Définition 2 (Manipulation de pointeurs).
L'opérateur `&` permet d'obtenir l'adresse mémoire d'une variable. On peut l'utiliser pour définir un pointeur à partir d'une variable :

```c
int x = 42;
int* pointeur = &x; // pointeur contient l'adresse de x
```

Réciproquement, l'opérateur `*` permet d'accéder à la valeur du contenu pointée par un pointeur. On parle de déréférencement :

```c
int y = *pointeur; // y contient la valeur pointée par le pointeur, donc x
printf("%d\n", y); // affichera donc 42
```

Dans l'exemple précédent, on a utilisé un déréférencement pour lire ce qui se trouve à une adresse mémoire donnée. On peut aussi l'utiliser pour modifier ce qui se trouve à cette adresse :

```c
*pointeur = 5; // y contient la valeur pointée par le pointeur, donc x
printf("x : %d; y : %d\n", y); // affichera 5 et 42
```

1. Les exemples précédents proviennent de `partie-A/ptr-base.c` . Compiler le fichier, et le lancer pour vérifier que les valeurs affichées sont bien celles attendues. Me demander si vous ne comprenez pas pourquoi les valeurs attendues sont celles que l'on attend.
2. Sortir un papier, une règle et un crayon. Faire un schéma de la pile mémoire aux débuts des lignes 8, 11 et 14. Me demander validation !!!!!!!!!!

1. Ou plus, je ne sais pas combien de Go de RAM vous avez.

Pour les prochaines questions de la partie A, il faut compléter le code des fonctions dans `partie-A/ptr.c`, puis compiler ensemble `partie-A/ptr.c` et `partie-A/main.c` afin de lancer les tests. Rappelons qu'il est demandé d'ajouter `-Wall` et `-Wextra` comme options à la compilation.

NB : Vous verrez beaucoup d'avertissement « unused parameters ». C'est normal, tant que les corps des fonctions sont vides, leurs arguments ne servent effectivement à rien.

3. **a.** Complétez la fonction `int dereference_int(int* ptr)` qui prend en argument un pointeur vers un entier et renvoie la valeur de cet entier. Testez-la.
   b. Complétez de même `double dereference_double(double* ptr)`. C'est la même fonction que la précédente, sauf qu'elle manipule des nombres à virgule flottante double précision plutôt que des `int`.
   c. Complétez `deref_et_somme`. Elle doit déréférencer les deux entiers non-signés (de type `unsigned int`), les sommer, et renvoyer leur somme.
4. Écrire une fonction `void met_a_0(int* ptr)` qui met à 0 le contenu pointé par `ptr`.
5. Écrire une fonction `void incr(int *ptr)` qui incrémente de 1 l'entier pointé par `ptr`.
6. Écrire une fonction `void swap(int* a, int* b)` qui échange le contenu des cases pointées par `a` et `b`.
   On fera attention à ne pas confodnre « échanger les pointeurs » avec « échanger le contenu des cases pointées ».0
7. Écrire une fonction `void min_et_max(int T[], int len, int* mini, int* maxi)`. Elle prend en entrée un tableau, sa longueur, et deux pointeurs. Elle doit calculer le minimum du tableau et le stocker dans l'entier pointé par `mini`, et faire de même pour le maximum.

Remarque. Les quatre fonctions précédentes modifient leur « monde extérieur » grâce à un pointeur : elles ont des effets secondaires. En fait, les pointeurs permettent de passer une donnée par référence. On parle aussi de... passage par pointeur ![2]

8. Dans les quatre questions précédentes, quels étaient les effets secondaires ?

A.2 const

On peut garantir l'absence d'effets secondaires en ajoutant le mot clef `const` au type :

Définition 3 (const).
`type const*` est le type des pointeurs pointant vers des contenus non-modifiables de type `type`.
On peut également appliquer `const` aux tableaux. Par exemple, `int const tab[]`.

NB : On peut aussi le noter `const type*`. C'est un style plus ancien, mais qui peut rendre des types moins lisibles (si `type` contient lui-même des const).

9. Que fera la fonction suivante (devinez avant de tester !) ?

```c
void mouhaha(int const* x) {
        *x = 666;
        return;
}
```

A.3 Retour sur scanf

10. Dans un nouveau fichier, écrire une fonction `void assigne(int x, int* ptr)` qui assigne la valeur `x` à l'entier pointé par `ptr`.

Notre fonction `assigne` est une version simplifiée de `scanf` :

Proposition 4.
`scanf` prend en arguments :

- Une chaîne de caractères qui est l'entrée attendue, contenant des spécificateurs `%` (comme `%d`, `%lf`, etc) .
- Des *pointeurs* qui indiquent où seront stockées les valeurs lues par les spécificateurs.

2. Vous verrez peut-être un jour une distinction entre le passage par référence et le passage par pointeur. À notre niveau, il n'y en a pas.

C'est pour cela que l'on mettait des & dans les `scanf` ! Quand on fait `scanf(" %d", &x)` , on demande de lire un entier (`%d`) et on indique *où* le stocker : à l'adresse `&x` . Retenez l'idée :

- **printf** : pour *afficher*, il faut donner les valeurs qu'il faut afficher.
- **scanf** : pour *lire*, il faut donner *l'adresse* où stocker ce qui est lu.

A.4 Pointeurs et tableaux

Dissipons une confusion usuelle, qui est souvent proférée sur les internets mondiaux ou par des quidams ayant (mal) appris le C :

Proposition 5 (tableau ≠ pointeur).
Les tableaux ne sont pas des pointeurs, et les pointeurs ne sont pas des tableaux !

Il existe par contre un lien entre les deux.

Définition 6 (Afaiblissement en pointeur).
Quand on passe un tableau en argument à une fonction, C passe en fait un pointeur vers le début du tableau. Ce qui explique que le contenu du tableau se comporte comme si passé par référence.
On dit qu'un tableau s'**affaiblit en pointeur.**

C'est pour cela que dans un schéma mémoire, lorsqu'un tableau est passé en argument on représente dans le bloc de la fonction appelé uniquemement un pointeur vers le tableau d'origine.

B Allocation mémoire sur le tas

B.1 Compléments sur les pointeurs

Définition 7 (`NULL`).
La librairie `stdlib` contient un pointeur très particulier : `NULL` . C'est le pointeur qui ne pointe sur *rien*. On peut assigner ce pointeur à n'importe quel type : `type* ptr = NULL` fonctionne pour tous les types.

Définition 8 (`void*` et conversions).
Parfois, on veut définit des pointeurs sans connaitre le type de ce qui est pointé. On leur donne alors le type `void*` : c'est le type d'un pointeur qui pointe vers *quelque chose*, sans plus d'informations.
On peut convertir des pointeurs depuis et vers le type `void*` . la syntaxe est `(nouveau_type) pointeur` .

```c
void* ptr = NULL;
... // du code qui modifie ptr
int* nouveau_ptr = (int*) ptr; // nouveau_ptr pointe vers la même chose que ptr
                               // MAIS il sait que ce sur quoi il pointe est un
                                ↪ entier
printf("%d\n", *nouveau_ptr); // affiche l'entier pointé par nouveau_ptr
```

B.2 Allocation dynamique

Vous utiliserez désormais `partie-B/main.c` . Il est très peu pré-complété, vous devrez faire la plupart des choses vous-même.

Définition 9 (`sizeof`).
La fonction `sizeof` (rangée dans `stdlib.h`) prend le nom d'un type en argument et renvoie le nombre d'octets utilisés pour stocker un élément de ce type.
Par exemple, `int` prend généralement 4 octets (32 bits) donc `sizeof(int)` vaut 4.

Remarque.

- En réalité, ce n'est pas une fonction : la valeur est calculée à la compilation et écrite en dur dans l'exécutable créé (contrairement à une fonction où la valeur est écrite à l'exécution).
- On peut aussi donner à `sizeof` une variable en argument. Cela revient à donner le type de la variable en argument.

11. Affichez les tailles de `int`, `unsigned int`, `bool` et `double`. Commenter.

Il faut utiliser `%lu` *dans* `printf` *pour afficher la valeur renvoyée par* `sizeof`.

12. (Bonus, à faire uniquement si vous êtes plutôt en avance) Créez un tableau et affichez sa taille. Ensuite, passez le tableau en argument à une fonction, et affichez sa taille une fois dans la fonction. Commentez.

Définition 10 (`malloc` et `sizeof`).
La fonction `malloc` (rangée dans `stdlib.h`) permet de réserver un certain nombre de cases mémoires adjacentes, et renvoie un pointeur vers la première de celles-ci :

```c
void* ptr = malloc(4);
```

Dans l'exemple ci-dessus, on réserve 4 cases mémoires c'est à dire 4 octets. On peut convertir le pointeur renvoyé par `malloc` :

```c
int* ptr_int = (int*) malloc(4);
```

La zone réservée peut être assez grande pour stocker plusieurs valeurs du type indiqué :

```c
int* ptr_int = (int*) malloc(40*sizeof(int)); // peut stocker 40 int
```

Dans ce cas, la zone réservée fonctionne comme un tableau. Ainsi, dans l'exempe précédent, on peut utiliser `ptr_int[0]`, ..., `ptr_int[39]`.

13. Écrire une fonction `tuto_alloc(int n)`. Elle doit réserver une zone mémoire assez grande pour stocker `n` entiers. Ensuite, elle doit écrire `42` dans chaque case. Enfin, elle doit renvoyer le pointeur vers cette zone.
Pour la tester, dans le `main`, récupérez le pointeur renvoyé par la fonction et affichez le contenu de ses `n` cases.

14. Appelez-moi en cas de besoin d'aide ! Appelez-moi aussi pour validation.

Remarque.

- On dit que `malloc` *alloue* des cases mémoires. D'où le nom de la fonction : **m**emory **alloc**ation.
- Il ne faut jamais hésiter à faire des dessins pour représenter les pointeurs et comprendre ce qu'il se passe. Je recommande chaudement le site `https://pythontutor.com/c.html#` qui fait ces dessins pour vous. Servez-vous à volonté !
- Les cases mémoires en question sont allouées dans le tas mémoire. C'est une zone séparée de la pile (et des données statiques). Les cases ne sont pas libérées tant que l'on ne le demande pas : il faut indiquer explicitement la fin de la durée de vie avec `free` (cf ci-dessous).

Définition 11 (`free`).
La fonction `free` (rangée dans `stdlib`) prend en argument un pointeur vers une zone allouée par `malloc`, et libère (c'est à dire désalloue) cette zone.

Théorème 12 (Important !!!!).
Toute zone crée par `malloc` doit être libérée dès lors qu'elle n'est plus utile.
Ne pas le faire mène à ce que l'on nomme des *fuites mémoires* : le programme consomme de plus en plus de mémoire, pour rien.

Remarque.

- Je n'insisterai jamais assez sur à quel point il est important de penser à libérer la mémoire. C'est vraiment une erreur classique qui peut rendre catastrophique un programme plutôt bon au demeurant [3].

3. Le corrigé d'une épreuve de concours de l'an dernier avait une fuite de mémoire, qui faisait qu'une fonction consommait une quantité exponentielle de mémoire et atteignait donc vite la limite de la RAM...

- Si jamais vous oubliez de libérer la mémoire - ce qu'il ne faut jamais jamais faire [4] -, et si le programme parvient à atteindre sa fin, le système d'exploitation libère alors de force toute la mémoire encore allouée.
  C'est pour cela que quand vous avez un programme qui consomme trop de mémoire (ex : un navigateur internet), le fermer de force suffit à regagner sa mémoire.

15. Modifier la partie du `main` qui teste `tuto_alloc` pour qu'elle n'oublie pas de libérer les pointeurs.

Vous pouvez ajouter `-fsanitize=address` aux options de `gcc` . Cela *modifie votre code*, afin d'essayer de détecter des erreurs liées aux adresses mémoire (comme une fuite de mémoire). Ses messages d'erreurs sont un peu durs à lire (car ils affichent des adresses mémoire), demandez-moi de l'aide au besoin.

Il existe également `-fsanitize=undefined` , qui lui aussi modifie votre code pour essayer de détecter des erreurs usuelles (lire une variable avant de lui avoir donné une valeur, par exemple). Ainsi, la ligne de compilation que l'on utilise désormais ressemble à :

```
gcc -Wall -Wextra -fsanitize=undefined -fsanitize=address  <vos fichiers .c>
```
Terminal

On peut raccourcir en :

```
gcc -Wall -Wextra -fsanitize=undefined,address  <vos fichiers .c>
```
Terminal

Remarque.

- Dans le terminal, à l'aide des flèches haut et bas vous pouvez parcourir l'historique des commandes. Cela vous évitera de retaper à chaque fois cette ligne à rallonge...
- Parfois (rarement), les sanitizers `undefined` et/ou `address` peuvent créer des bugs [5]. Si c'est le cas, ne les mettez pas.

Dans les prochaines questions, je vous donne des petits problèmes qui demandent de calculer des valeurs. Vous devrez écrire ces fonctions [6], et les tester. La difficulté peut venir de l'allocation mémoire, ou d'ailleurs. Vous n'oublierez pas de libérer la mémoire.

16. Écrire une fonction `int* fibo(int n)` qui calcule les `n` premières valeurs de la suite de Fibonnacci, et renvoie un pointeur vers une zone allouée qui les contient.
Cette suite est définie comme suit :

$$\forall n \in \mathbb{N}, f_n = \begin{cases} f_{n-1} + f_{n-2} & \text{si } n \geq 2 \\ 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \end{cases}$$

Les premières valeurs de cette suite sont 0, 1, 1, 2, 3, 5, 8, 13, ...
(Bonus :) trouvez le premier `n` pour lequel f_n déborde d'un `int`.

17. Écrire une fonction `bool* est_premier(int n)` calcule pour les entiers de $[\![0; n[\![$ s'ils sont premiers ou non. Elle renvoie un pointeur vers une zone contenant des booléens telle que la case d'indice `i` de cette zone contient `true` si `i` est premier, et `false` sinon.
Un entier premier si et seulement si il admet exactement deux diviseurs distincts : 1 et lui-même.
Ainsi, 0 n'est pas premier, 1 n'est pas premier, 2 est premier, 3 est premeir, 4 n'est pas premier, 5 est premier, 6 n'est pas premier, 7 est premier, 8 n'est pas premier, 9 n'est pas premier, 10 n'est pas premier, etc.

18. Écrire une fonction `int* base10(int n, int* nb_chiffres)` qui décompose `n` en base 10. Elle renvoie un pointeurs vers une zone contenant les chiffres de l'écriture décimale de `n`, du chiffre de poids faible jusqu'au chiffre de poids fort. Elle stocke le nombre de chiffres dans `nb_chiffres` .

Par exemple, sur l'entrée 521, elle doit renvoyer un pointeur vers une zone dont le contenu est `[|1; 2; 5|]` , et modifier `*nb_lettres` pour qu'il contienne 3.

19. (Plus difficile) Écrire une fonction `void affiche_triangle_pascal(int n)` qui affiche les `n` premières lignes du triangle de Pascal.

C M'sieur j'ai fini !

20. Allez débloquer le niveau 3 de France-IOI (plus facile à dire qu'à faire).

21. Quoi, ça aussi c'est fini ? Bah débloquez le niveau 4.

4. Et j'enleverai tous les points que je veux sur vos DS pour faire rentrer cette leçon.
5. Cela dépend de votre machine en fait. Ils ne fonctionnent pas sur toutes les architectures d'ordinateur.
6. Et éventuellement des fonctions supplémentaires si vous en ressentez le besoin.