TD : Programmation en C — semaine 4

2 mars 2018

Objectif(s)

- ✯ Utilisation des pointeurs.
- ✯ Allocation dynamique.

Exercice 1 – Remise en forme

— **Note 1** : Pour tous vos programmes, vous utiliserez le script *run.sh* pour compiler.
— **Note 2** : Pour tous vos programmes, vous ferez des fonctions de 5 lignes maximum.

1. Lire la correction du mini-jeu du TD3 disponible sur le site web du cours.
2. Faire une fonction `min` et une fonction `max` prenant 2 arguments que vous appellerez dans un programme demandant à l'utilisateur de rentrer n entiers et qui affichera le plus petit et le plus grand des n.

Exercice 2 – Tu pointes ou tu tires ?

La mémoire est un grand bloc de cases tel que chaque case contient une valeur et une adresse. Un pointeur est une variable comme une autre, sauf que sa *valeur est une adresse*. Une analogie est une boite au lettre : elle a une adresse et contient du courrier (les valeurs). Si vous déménagez, vous demandez à la poste une redirection : tout le courrier envoyé vers la première adresse sera redirigé vers la deuxième. Dans ce cas, la valeur de la boite au lettre est une adresse. En C, on sait si une variable contient une adresse ou une valeur si sa déclaration est précédée par une étoile, par exemple avec `int* x;`, la variable `x` contiendra une adresse. Vous ferez les exercices suivants **en dessinant sur une feuille de papier** la mémoire des programmes suivants.

1. ```
   int x = 1;
   int* y = &x;
   int z = *y;
   ```

2. ```
   int* x;
   x = &x;
   int y = *x;
   ```

3. ```
   int x = 1;
   int* y = &x;
   int** z = &y;
   int w = **z;
   ```

4. ```
   int* x;
   x = &x;
   int y = ****x;
   ```

5.
```
int* x = 0;
int y = *x;
```

6.
```
int x[2] = {1, 2};
int* y = &x[1];
int z = *y;
```

7.
```
int* x[2];
x[0] = &x[1];
x[1] = &x[0];
int z = *(x[0]);
int z2 = **(x[1]);
```

8.
```
int x[2][2] = {{1,2},{3,4}};
int* y = &x[1];
int z = y[1];
```

En C, un pointeur est considéré comme une valeur : effectivement, une adresse c'est un entier. Par contre, un `int` a une taille de 32 bits, alors qu'un pointeur peut avoir une taille de 32 ou 64 bits suivant que votre machine tourne en 32 bits ou 64 bits.

1. Si votre ordinateur est en 64 bits, quels sont les exemples ci-dessus qui ne marcheront pas ? En effet, on a parfois utilisé une astuce qui consiste à convertir une adresse vers un entier et vice-versa, mais si ils n'ont pas la même taille, c'est la cata.
2. Si z est situé juste après x en mémoire, que contient w à la fin de l'exécution du programme suivant ?

```
int x[2] = {1, 2};
int z = 3;
int w = x[2];
```

3. Et dans le cas suivant ? Est-ce que ça pose un problème ?

```
int x[2] = {1, 2};
int w = x[2];
```

Exercice 3 – Encore plus de pointeurs

Lorsqu'on utilise des tableaux, nous ne pouvons pas les retourner d'une fonction ! En effet, considérer la fonction suivante :

```
void f() {
  int t[2] = { 1, 2 };
  int z = 3;
}
```

Lorsqu'on exécute f, on crée un bloc de 3 cases en mémoire (2 cases pour le tableau t et 1 pour z). Ces variables ont une durée de vie limitée ! Lorsqu'on sort de la fonction f, les cases mémoires sont automatiquement libérées afin d'être ré-utilisées ultérieurement—et heureusement, sinon on manquerait très vite de mémoire lorsqu'on lance un programme. Ce mécanisme est automatique. Si on essaye d'accéder à une case mémoire qui ne nous est plus réservée, on obtient la fameuse `segfault` qui signifie qu'on essaye d'accéder à une case mémoire invalide.

1. Que se passe t-il dans le programme suivant ?

```
int* f() {
  int t[2] = {1 , 2};
  return t;
}
int main() {
  int* t = f();
  printf("%d", t[0]);
}
```

2. Si la mémoire est libérée automatiquement, pourquoi est-ce qu'on peut accéder à y dans le programme suivant ?

```
int f() {
  int x = 1;
  return x;
}
int main() {
  int y = f();
  printf("%d", y);
}
```

En C, le passage de paramètres et retour de fonction s'effectuent *par valeur*. Concrètement cela veut dire que C va copier les cases mémoires, mais attention : si la case mémoire contient un pointeur, C va copier le pointeur mais pas la valeur pointée ! ! C'est comme un facteur feignant qui ne suit pas les redirections des boites aux lettres mais copie l'adresse de la première boite au lettre.

3. Est-ce que l'accès à `t[0]` dans `f` et dans le `main` peut générer une erreur de segmentation (`segfault`) ?

```
void f(int* t, int size) {
  if(size > 0) {
    t[0] = 1;
  }
}
int main() {
  int t[2] = {1, 2};
  f(t, 2);
  printf("%d", t[0]);
}
```

4. Qu'affiche le programme précédent ? Quel est l'adresse de `t` dans le `main` et dans la fonction `f` ? Est-ce la même ? Pourquoi ?

Exercice 4 – Allocation dynamique

Le problème avec les tableaux qu'on déclare manuellement est qu'ils ont une taille fixe ! Que faire si c'est l'utilisateur qui choisit la taille du tableau ? Sans le savoir, certains d'entre vous ont déjà utilisé des tableaux à taille dynamique appelés *Variable Length Array (VLA)* qui ne sont disponibles que à partir de `C99` (donc pas sans le flag de compilation `-std=c99` présent dans le `run.sh`). Vous pouvez donc faire :

```
int main() {
  int size;
  scanf("%d", &size);
  int notes[size];
}
```

qui va déclarer un tableau de taille $size$ où $size$ est choisi par l'utilisateur. Par contre, ça n'enlève pas le problème que la mémoire est automatiquement libérée lorsqu'on sort de la portée de cette variable, par exemple :

```
int* f() {
  int size;
  scanf("%d", &size);
  int notes[size];
  return notes;
}
int main() {
  int* notes = f();
  printf("%d", notes[0]);
}
```

est **totalement invalide** puisque la mémoire a été libérée à la sortie de `f`. Donc comment faire ?

La solution à ce problème est d'utiliser l'allocation dynamique à l'aide de deux fonctions `malloc` et `free`. Par exemple :

```
int* f() {
  int size;
  scanf("%d", &size);
  int* notes = (int*) malloc(sizeof(int) * size);
  return notes;
}
int main() {
  int* notes = f();
  printf("%d", notes[0]);
  free(notes);
}
```

est tout à fait valide ! Ces deux fonctions nous donne la main sur la mémoire et c'est **le programmeur** qui décide quand allouer ou libérer la mémoire. Tant qu'on appelle pas `free`, on peut continuer à utiliser le bloc de mémoire correspondant. Règle générale : il faut toujours qu'un `malloc` soit accompagné de son `free`, sinon il est triste.

1. Demander à l'utilisateur la taille d'un tableau et puis des nombres pour remplir ce tableau. Vous ferez cela en utilisant `malloc` et `free` avec une fonction retournant le résultat. Comment faire pour retourner le tableau et sa taille ?
2. Créer une fonction qui multiplie les éléments d'un tableau par une valeur v et retourner un nouveau tableau avec les valeurs multipliées (en laissant l'original intact !).
3. Pareil mais avec une fonction qui ajoute c à un tableau.
4. En utilisant les deux fonctions précédentes, créer une fonction qui crée un nouveau tableau u tel que $u[i] = t[i]*v+c$.

Exercice 5 – Valgrind

Soyons clair : gérer sa mémoire à la main n'est pas facile et de nombreux problèmes peuvent se poser dont :
— *Double free* : appeler `free` sur une mémoire déjà libérée,
— *Segfault* : utiliser la zone mémoire après l'appel à `free`,
— *Memory leak* : ne jamais libérer la mémoire.

Un outil très important pour détecter ces problèmes est `valgrind`. Vous pouvez lancer le programme avec valgrind en faisant `valgrind ./mon_prog` et il vous donnera des informations.

À partir de maintenant, **pour tous vos programmes**, il faudra que vous utilisiez `valgrind` et qu'il vous dise :

```
All heap blocks were freed -- no leaks are possible
```

et qu'il n'émette aucun avertissement.

1. Soit le programme suivant :

```
int main() {
  int* t = malloc(sizeof(int)*10);
  printf("%d", t[0]);
  return 0;
}
```

Passer le dans `Valgrind` et corriger les problèmes.

2. Passer les programmes de la question précédente dans `Valgrind` et corriger les problèmes potentiels.

Exercice 6 – Recherche dichotomique

1. Lorsqu'on a un tableau trié, on peut dire si le tableau contient ou non un élément en $O(log(n))$, ce qui signifie que pour un tableau de taille n, on parcourt au maximum $log(n)$ éléments. Cet algorithme s'appelle la *recherche dichotomique*, le principe est simple, soit un tableau t et l'élément à rechercher x :
   1. On garde l'intervalle (l, u) tel que $t[l] \leq x \leq t[u]$,
   2. Ensuite, on réduit cet intervalle en prenant le milieu, par exemple si $t[mid] < x$, on recommence avec $(mid + 1, u)$.

   Implémentez l'algorithme correspondant avec la fonction suivante :

```
// Return the index of the element `x` if it is present in `array`,
// otherwise return `-1`.
int binary_search(int* array, int x);
```

2. Suivant le même principe, on vous demande d'écrire une fonction récursive qui n'utilise pas de boucles. Son entête sera :

```
// Return the index of the element `x` if it is present in `array`,
// otherwise return `-1`.
int binary_search(int* array, int x);
// Look for `x` for each `i` of `t[i]` such that `idx_lower <= i <= idx_upper`.
int binary_search_aux(int* array, int x, int idx_lower, int idx_upper);
```

3. Basé sur le même principe, créer une fonction :

```
void slice(int* array, int size, int min, int max,
  int* idx_lower, int* idx_upper);
```

qui donne la tranche du tableau `array` tel que tout élément entre `array[idx_lower]` et `array[idx_upper]` soit compris entre `min` et `max`. Cet algorithme doit fonctionner en $O(log(n))$ également.

Exercice 7 – Tri rapide

1. Soit le pseudo-code donné par la page Wikipédia :
   `https://fr.wikipedia.org/wiki/Tri_rapide#Description_de_l'algorithme`
   Implémentez l'algorithme du tri rapide en C.