TP5: Pointeurs & Allocation Dynamique

Rémy Poulain & Gabriel Hondet

23 octobre 2019

Au menu de la séance : des *pointeurs*.

Prérequis : avoir lu ou entendu le chapitre "Pointeurs" de Jean (ici pour le poly).

1 Pointeurs

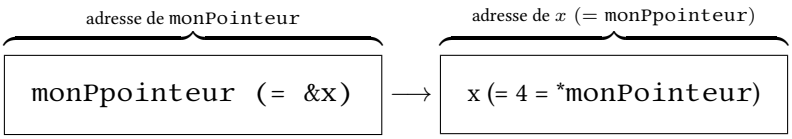
Le pointeur est un objet central dans la programmation en C. Il faudra vous assurez que vous comprenez bien comment ça fonctionne. Pour un petit rappel, consultez ce résumé et si ça ne suffit pas, lisez le poly de Jean.

Il est assez facile de motiver son utilisation : lorsque vous voulez par exemple réaliser le produiut matriciel de deux matrices de dimensions 100000, il est beaucoup plus simple de communiquer à votre fonction les emplacements mémoires de vos matrices au lieu de faire passer toutes ces données à la fonction !

• Un pointeur est une adresse *pointant* vers une valeur. Si vous avez bien suivi le cours de Jean, tout est fait de flèches dans la mémoire. Le pointeur de C permet de stocker et manipuler cette flèche. Des points de syntaxe :

- `int *pointeur;` déclare un pointeur vers un entier. Il faut comprendre que pour le moment, `pointeur` n'a pas encore été instancié par une adresse. Aucune case supplémentaire dans la mémoire n'a été allouée ;
- `int x = 1; pointeur = &x;` déclare un entier x (vous savez tout ça) puis instancie `pointeur` comme étant l'adresse de x. En d'autres termes, `pointeur` *pointe* maintenant vers la case mémoire de x. L'opérateur & devant une variable renvoie donc son adresse (au lieu de renvoyer sa valeur) ;
- `*pointeur` renvoie la valeur pointée par `pointeur` c'est à dire le contenu de la case dont l'adresse est `pointeur`. En d'autres termes, l'opérateur $*$ devant un pointeur renvoie la valeur stockée dans l'adresse contenue par le pointeur ;
- `*pointeur = 4` change le contenu de la case mémoire vers laquelle pointe `pointeur`. Hors cette case, c'est x. Donc x vaut maintenant 4 !

Avec un schéma ça donne (dans les boîtes on représente la valeur stockée par la case mémoire et au-dessus l'adresse de la case mémoire, la flèche nous aide juste à visualiser le lien) :



Vous devez comprendre ces deux opérateurs $*$ et & et avoir le schéma en tête. Les questions du prochain exercice vous aussi vous aider.

Exercice - 1 *Pointeurs*

1- Copiez-collez le code suivant :

```
#include <stdio.h>

void qu1()
{
  int x = 1;
  int *p;
  p = &x;
  printf("x vaut: %d\n", x);
  printf("p vaut: %d\n", p);
  printf("*p vaut: %d\n", *p);
  printf("&x vaut: %d\n", &x);
  *p = 2;
  printf("(après *p = 2) x vaut maintenant: %d\n", x);
  x = 5;
  printf("(après x = 5) *p vaut maintenant: %d\n", *p);
}

int main()
{
  qu1();
  return 0;
}
```

Compilez (malgrés les warnings), exécutez. Tout va bien ? Exécutez de nouveau : remarquez le changement d'adresse.

2- Ecrivez une fonction qui échange les valeurs pointées par deux pointeurs dont le prototype est `void swap (int *a, int *b)`.

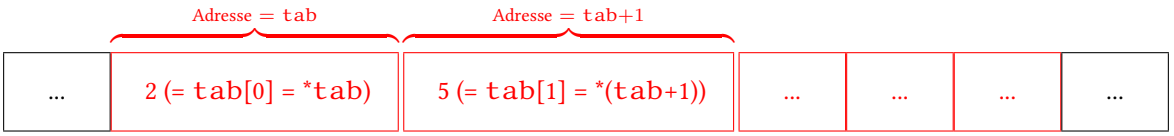
• La fonction `swap` renvoie `void` autrement dit, elle ne fait que des *effets de bords* (changement dans la mémoire) mais ne renvoie pas de valeurs. C'est bien ce qu'on attend de cette fonction. Attention, `int *a` dénote le type d'un pointeur vers un `int`.

3- La semaine dernière, on a déja manipulé des pointeurs sans le savoir (ou pas). `int tab[5]` déclare un tableau de taille 5. Plus précisemment, cette instruction alloue 5 cases mémoire côte à côte et déclare `tab` comme étant un pointeur vers un entier. `tab` est également instancié par l'adresse de la première case mémoire du tableau. Copiez-collez le code suivant et ajoutez `qu3();` à votre `main` :

```
void qu3()
{
  int tab[5];
  tab[0] = 2;
  tab[1] = 5;
  printf("\ntab[0] vaut: %d\n", tab[0]);
  printf("*tab vaut: %d\n", *tab);
  printf("tab[1] vaut: %d\n", tab[1]);
  printf("*(tab+1) vaut: %d\n", *(tab + 1));
}
```

Compilez, exécutez. Surpris ? Quelle est la différence entre `*tab + 1` et `*(tab+1)` ?

• Pour vous aider à comprendre, voici un schéma :



• Quand on passe un tableau en argument de cette façon : `type fonction (int tab[])` on peut écrire de façon équivalente `type fonction (int *tab)`. La seule différence est pour le lecteur qui sait à quoi s'attendre en lisant le premier prototype.

4- Ecrivez une fonction qui prend en argument un tableau d'entiers, sa taille, et deux pointeurs vers des entiers et qui met dans le premier (resp. second) pointeur le minimum (resp. maximum) des valeurs du tableau. Son prototype doit être : `void minMax (int tab[], int taille, int *min, int *max)`.

Exercice - 2 *Allocation dynamique*

Nous avons vu que lorsque l'on déclare un tableau `int tab[N]`, on allouait également N cases mémoires contiguës. C'est pour cette raison que le compilateur doit connaître la valeur N à l'avance. Heureusement, il est possible d'allouer de la mémoire dynamiquement avec `malloc`. Les tableaux déclarés de cette façon s'utilisent comme d'habitude.

• Pour allouer dynamiquement des cases mémoires, il faut expliciter la taille que l'on veut allouer. Par exemple pour un tableau d'entiers de taille n la taille du tableau est `sizeof(int) * n` (taille d'un entier fois n). La fonction `malloc` prend en argument la taille à allouer et renvoie un pointeur vers la première case. Exemple : pour déclarer et allouer de l'espace pour un tableau d'entiers `tab` de taille n, vous pouvez utiliser : `int *tableau = malloc(sizeof(int) * n)`. Attention, lorsque vous ne voulez plus utiliser cette espace vous devez manuellement demander sa suppression avec `free`. Exemple : `free(tab)` désalloue la mémoire. Si vous oubliez de le faire, vous risquez une *fuite mémoire* qui peut faire planter votre programme.

1- Ecrivez une fonction qui demande à l'utilisateur (dans le terminal) une taille de tableau puis autant d'entiers et construisez le tableau correspondant. Affichez le ensuite.

• Pour gérer les entrées dans le terminal, vous pouvez utiliser `scanf` qui s'utilise comme `printf` excepté que vous devez donner des adresses et non des valeurs après la chaîne de cractères. Exemple : pour demander un entier vous pouvez entrer : `int x; scanf("%d.", &x)`. La valeur entrée dans le terminal sera stockée dans x.

Si vous voulez en savoir plus sur les pointeurs, vous pouvez consulter le chapitre 3 du cours de B. Cassagne.

Le TP n'est pas terminé

2 Problèmes

Exercice - 3 *Tableaux dynamiques*

On se propose d'écrire une petite librairie implémentant des tableaux dynamiques. On fournit ici un fichier d'en tête (plus souvent appelés *headers*), à l'adresse `http://www.lsv.fr/~hondet/dynarray.h`. Le but est d'écrire le fichier source C correspondant. Les prototypes sont recopiés ci-dessous :

```
struct dynarray {
    int* data;
    size_t capacity; /// Maximal number of elements
    size_t size; /// Next cell index to write, number of elements
};

typedef struct dynarray dynarray;

dynarray* init();

int empty(dynarray*); /// True if array is empty
void pp(dynarray*);

/// Accessors
int at(dynarray*, size_t);
int front(dynarray*); /// Returns first element
int back(dynarray*); /// Returns last element

/// Modifiers
void push_back(dynarray*, int);
int pop_back(dynarray*); /// Delete last element and return it

/// Functional
void iter(void (*)(int), dynarray*);
dynarray* map(int (*)(int), dynarray*);

void clear(dynarray*);
```

La fonction "compliquée" est `push_back`, elle doit agrandir le tableau `data` si besoin s'en fait.

Exercice - 4 *Caesar cipher*

Vous devez écrire un programme qui permet le chiffrement et la déchiffrement de texte en utilisant le schéma de César que l'on va un peu modifier. La clé de chiffrement contient deux entiers k_1 et k_2. La première servira à modifier les lettres, par exemple $a \to c$ si $k_1 = 3$. La seconde servira à décaler les lettres dans le texte. Par exemple Coucou $\to$ uCouco si $k_2 = 1$.

1- Ecrivez une fonction de chiffrement prenant un tableau de char, sa taille, deux clés et un tableau de char (pour stocker le résultat) de prototype : `void chiffr (char text[], int taille, int k1, int k2, char res[])`.

2- Ecrivez sur le même principe une fonction de déchiffrement.

3- Ecrivez une fonction `main` implémentant le comportement suivant : `./a.out -c 10 45 "Chiffrez moi svp"` renvoie le texte chiffré pour les clés $k_1 = 10$ et $k_2 = 45$ et `./a.out -d 124 4 "Hjklhkjk hjk hjk"` renvoie le texte déchiffré pour les clés $k_1 = 124$ et $k_2 = 4$. Faites en ensuite une version interactive.

4- (Bonus) Ce schéma de chiffrement est très faible principalement à cause de l'espace des clés qui par sa taille peut être brut-forcé. Une façon de contourner cette attaque est de considérer une permutation des lettres au lieu d'un simple décalage. Implémentez cette solution en proposant une représentation agréable des clés.

5- (Bonus Bonus) Al-Kindi a trouvé au IXe s. une faille basée sur une analyse fréquentielle des lettres. Blaise de Vigenère propose une solution (seulement au XIe s.). Exploitez la faille et/ou Implémentez la solution.

Exercice - 5 *Listes chaînées* *reste du sujet d'après David Baelde*

Nous allons définir un type de donnée pour des listes chaînées. C'est comme les listes doublement chaînées du TP 2, mais avec une seule direction de chaînage, et en C.

Ouvre un nouveau fichier `.c`, et commencez par inclure les en-têtes suivant, respectivement pour les fonctions de gestion mémoire, d'affichage, et… assert () :

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
```

On définit ensuite le type enregistrement `struct _list`, et un alias `list` pour celui-ci :

```
typedef
  struct _list {
    int value;
    struct _list *next;
  }
  list;
```

Chaque cellule de nos listes chaînées sera représentée par un objet de type `list`, qui est une structure comportant deux champs : la valeur contenue dans la cellule, et un pointeur vers la prochaine cellule. Par convention, le pointeur NULL est utilisé en fin de liste, quand il n'y a plus de prochaine cellule. (En réalité, ce qu'on pensera comme une liste est plutôt un objet de type `list*` que `list`. La liste vide est représentée par NULL et une liste non-vide par un pointeur valide sur une cellule.)

Pour accéder à un champ d'un `struct`, on utilise simplement `.`, par exemple : `uneListe.value`.

1- Définissez une fonction qui "ajoute un élément à une liste". Plus précisemment, elle devra allouer une nouvelle cellule contenant la valeur donnée et pointant ensuite vers la liste pré-existante, qui n'est pas modifée. Votre fonction devra se conformer au prototype suivant :

```
list* cons(int hd, list* tl);
```

2- Créez une fonction pour afficher les éléments d'une liste, avec un nombre maximal n d'éléments à afficher. Elle pourra par exemple produire des résultats comme `1:2:3:nil` ou `1:2:3:...`. Quand n est négatif, tous les éléments devront être affichés quel que soit leur nombre. La fonction devra se conformer au prototype suivant, et l'on créera aussi une seconde fonction pour ne pas avoir à passer `n=-1` explicitement.

```
void show_max(list* hd, int n);
void show(list* l);
```

3- Testez avec la fonction principale suivante :

```
int main() {
  list* l2 = cons(2,NULL);
  show(l2);
  list* l1 = cons(1,l2);
  show(l1);
  show(l2);
  return 0;
}
```

Cela devrait afficher (modulo détails de présentation) `2:nil`, `1:2:nil` puis `2:nil`.

4- Il y a une fuite mémoire dans le programme précédent. Vous pouvez par exemple le détecter en lançant votre programme sous l'environnement de déboguage `valgrind` :

```
$ valgrind ./monprogramme
```

Cet outil supervise l'exécution de votre programme, et notamment son utilisation de la mémoire. Vous devriez voir un LEAK SUMMARY : il vous informe qu'une partie de la mémoire allouée dynamiquement n'est jamais libérée, c'est une *fuite mémoire*.

Il nous faut une fonction de libération de la mémoire : celle-ci devra libérer toutes les cellules d'une liste, et se conformer au prototype suivant :

```
void destroy(list* tl);
```

Utilisez cette fonction pour libérer la mémoire une fois qu'on n'a plus besoin des listes `l1` et `l2`.

5- On veut maintenant implémenter une fonction d'accès au $n^{ème}$ élément d'une liste, l'élément de tête étant considéré comme ayant l'index 0. Faites-le en style impératif avec une fonction `nth` qui utilise une boucle (`for` ou `while` selon les goûts) :

```
int nth(int n, list* l);
```

Pour tester, récupérer le fichier `tests.c` sur la page des TPs : il contient une fonction `main()` avec de nombreux tests pré-écrits et commentés, que vous pourrez décommenter au fur et à mesure. Remplacez votre fonction principale par celle-ci.
A la fin du test de la fonction `nth()`, la liste est détruite puis on essaie de nouveau d'accéder à son premier élément : expliquez ce qui se passe alors.

6- Implémenter une fonction de concaténation. Précisemment, étant donné deux listes `l1` et `l2`, elle doit faire pointer la dernière cellule de `l1` sur la première cellule de `l2`. Le prototype :

```
void glue(list* l1, list* l2);
```

Testez en décommentant le bloc suivant dans la fonction `main()` fournie.

7- Implémenter une fonction qui teste si deux listes sont "structurellement" égales, c'est à dire qu'elles ont même contenu. Comme avant, on respectera le prototype suivant et on vérifiera le résultat du test fourni :

```
int equals(list* l1, list* l2);
```

8- Implémenter une fonction qui fasse boucler une liste sur elle même : la dernière cellule de la liste originale doit, après appel de la fonction, pointer sur la première cellule. Par exemple, `1:2:nil` devient, après bouclage, `1:2:1:2:1:2:...` à l'infini.

```
void tie(list* l);
```

9- Pour pouvoir appeler `destroy()` sur une liste circulaire (résultant par exemple d'un appel à `tie()`) il faut briser la circularité. Implémenter pour cela la fonction `untie()` qui prend une liste, supposant que celle ci boucle sur son premier élément, et détruit ce lien. Sur l'exemple précédent, après "débouclage" on devrait récupérer `1:2:nil`.

```
void untie(list* l);
```

Décommenter la suite du test, qui déboucle et détruit la liste. Vérifier qu'il n'y a pas de fuite mémoire.

10- Une liste est dite circulaire s'il y a un cycle dans les liens entre ses noeuds. Il ne s'agit pas forcément d'un cycle qui revient vers la première cellule. Par exemple, la liste `1:2:3:4:3:4:3:4:3:4:...` (qu'on peut obtenir par `glue()` et `tie()`) est circulaire. On pourra noter qu'en fait une liste ne peut être infinie sans être circulaire. Implémenter une fonction testant si une liste est circulaire.

```
int is_circular(list* l);
```

Indice : cherchez un algorithme en espace constant, qui n'utilise aucune structure de donnée auxiliaire ; en particulier, ne cherchez pas à mémoriser tous les noeuds déja rencontrés.

11- Implémentez un test d'égalité entre deux listes, qui fonctionne aussi bien quand les listes sont circulaires :

```
int equals_circular(list* l1, list* l2);
```

Exercice - 6 *Bonus : comptage de références*

Comme vous l'avez peut être remarqué en faisant l'exercice précédent, il peut être difficile de savoir quelles listes peuvent êtres détruites. Si l'on concatène `l1` à `l2`, il ne faut pas détruire `l2` même si on ne l'utilise plus jamais directement. Inversement, détruire `l1` ne doit pas être fait tant qu'une portion de code a encore besoin de `l2`.

Dans certains cas, ces problèmes sont insurmontables, et il devient bon de penser à mettre en place un dispositif de gestion automatique de la mémoire. Un des moyens les plus primitifs pour cela est le *comptage de référence*. L'idée est d'enrichir votre type de données (ici, les noeuds de la liste) avec un compteur indiquant combien d'autres données pointent vers celle-ci. Par exemple, à la création *ex nihilo* d'une liste `l1` non vide, toutes les cellules de la liste auraient un compteur de références à 1. (Pour la première, cela peut se comprendre comme indiquant que la fonction en cours "pointe" sur le premier élément de la liste par le biais de la variable locale qui détient la liste.) Si l'on concatène une liste `l2` en tête de `l1`, alors la première de cellule passe à 2 réferences.

On ne détruit une liste que quand plus personne ne pointe dessus. Sinon, la "destruction" consiste juste à décrémenter le compteur de références du noeud de tête de la liste. Pour clarifier, on distinguera l'opération de *libération* et de *destruction*, la première n'entraînant la seconde que dans le cas où le compteur de références est nul. En reprenant l'exemple précédent, des appels successifs à destroy sur `l1` et `l2` entraîneront la libération de toute la mémoire, quel que soit l'ordre dans lequel on détruit ; de plus, la libération de `l1` n'entraîne pas la destruction de `l2`, qui reste utilisable.

Implémenter un tel mécanisme. Il serait utile de réfléchir à organiser votre code de façon à systématiquement bien mettre à jour les compteurs, typiquement en isolant les opérations ayant trait à ceux-ci.

On pourra aussi critiquer la portée d'un tel système.

Exercice - 7 *Bonus : tables de hachage*

Une fonction de hachage associe un entier $h(x)$ à toute valeur x d'un certain type, avec l'espoir que la répartition de ces entiers soit plutôt uniforme. Etant donné une telle fonction, une table de hachage est une structure de donnée permettant de représenter un ensemble (modifiable) de valeurs indexées par des clés. Nous prendrons ici `string` comme type des clés et `int` comme type des valeurs, une table de hachage permettra ainsi par exemple de stocker l'age d'un certain nombre de personnes désignées par leur nom de famille.

Si une valeur v doit être associée à une clé k, on va la stocker dans la case $h(k)\%n$ d'un tableau de taille n. Comme plusieurs clés peuvent donner le même $h(k)\%n$, le tableau doit en fait contenir une liste d'associations, chaque association indiquant le couple clé-valeur.

1- Déclarer un type de listes d'associations (disons `assoc`) pour des clés de type `char*` et des valeurs de type `int`.

2- Déclarer une structure `table` qui contienne un tableau de listes d'associations, ainsi qu'un entier indiquant la taille du tableau, et enfin le nombre total d'entrées dans la table.

3- Implémenter une fonction pour créer une table vide, prenant en argument la taille du tableau :

```
table* create(int n);
```

4- Implémenter une fonction de hachage de votre choix :

```
int hash(string* s);
```

On pourra par exemple prendre $5381 + \sum_i a_i 33^i$ (une fonction due à Dan Bernstrein, empiriquement efficace) où les a_i sont les caractères successifs.

5- Implémenter la fonction d'insertion :

```
void insert(table* t, char* key, int value);
```

6- Implémenter la fonction de recherche, qui renvoie la cellule de liste d'association trouvée, ou bien NULL :

```
assoc* lookup(table* t, char* key);
```

Tester sur un petit ensemble de chaînes saisies manuellement.

7- Quand il y a plus d'entrées dans la table que la taille du tableau, la probabilité de collisions devient trop grande, et il est raisonnable de redimensionner le tableau. Implémenter cette fonctionalité, en doublant la taille du tableau :

```
void resize(table* t);
```

Modifier ensuite la fonction d'insertion pour automatiquement redimensionner quand on atteint un seuil dans le nombre d'éléments.

8- Générer toutes les chaînes de 3 caractères de $\{a, \ldots, z\}$, les stocker dans une table de taille 23.