# TP n° 2 – Listes chaînées

https://www.lri.fr/∼schevalier/teaching.html

28 février 2019

## 1 Rappels sur les pointeurs

Lorsque l'on exécute un programme, chacune des variables que l'on a définie est stockée en mémoire dans une zone réservée (de sa déclaration jusqu'à la fin de "son bloc"). Concrètement, la mémoire est constituée d'une multitude de blocs de 8 bits (un octet donc). La taille qu'occupe une variable en mémoire varie selon son type : par exemple, un type `char` occupera un seul bloc. Chaque bloc mémoire possède un identifiant, c'est-à-dire un numéro que l'on appelle *adresse*.

Partant de ce constat, on peut donc envisager d'accéder à une variable de deux façons : par son nom, ou par l'adresse du premier bloc alloué à la variable.

Un *pointeur* est une variable qui peut contenir l'adresse d'une autre variable, c'est-à-dire l'adresse du premier bloc alloué à la variable. Ainsi il suffit de manipuler des pointeurs pour manipuler (indirectement) des variables.

### 1.1 Connaître l'adresse d'une variable

Bien entendu, nous n'allons jamais écrire "en dur" des adresses de variables (exprimées le plus souvent en hexadécimal). Il existe donc un moyen permettant de connaître l'adresse d'une variable. Pour cela, il suffit d'ajouter le symbole `&` devant le nom d'une variable pour accéder directement à son adresse :

```
1 int ma_variable = 10;
2 printf("%p", &ma_variable); /* Affiche en hexadécimal l'adresse de la variable */
```

### 1.2 Déclaration et initialisation des pointeurs

Pour déclarer un pointeur, nous devons indiquer le type de la variable pointée. En effet l'adresse mémoire ne désignant que l'identifiant du premier des blocs dans lesquels se trouve la variable, préciser son type permettra de savoir sur combien de blocs supplémentaires s'étend la variable. Il faut ensuite préciser que la variable est un pointeur en utilisant le symbole `*`, et enfin donner un nom au pointeur :

```
1 type *nom_pointeur;
```

Par la suite, et comme pour toute autre variable, il faut initialiser le pointeur avec l'adresse d'une autre variable. Par exemple :

```
1 int ma_variable = 10 ;
2 int *mon_pointeur = &ma_variable; /* Déclaration et initialisation du pointeur */
```

Dans cet exemple, on dira que `mon_pointeur` pointe vers `ma_variable`, ou encore que `ma_variable` est pointée par `mon_pointeur`. Pour initialiser un pointeur à une adresse nulle, on utilisera :

```c
int *mon_pointeur = NULL; /* Déclaration et initialisation du pointeur */
```

Notez que, si on initialise un pointeur après sa déclaration, on n'utilise plus le symbole `*` :

```c
int ma_variable;             /* Déclaration d'une variable de type entier */
int *mon_pointeur;           /* Déclaration d'un pointeur d'entiers */
ma_variable = 10;            /* Initialisation d'une variable de type entier */
mon_pointeur = &ma_variable; /* Initialisation d'un pointeur d'entiers */
```

### 1.3 Accéder à la variable pointée

Après avoir initialisé un pointeur (à autre chose que `NULL`), il est possible d'accéder à la variable pointée. Pour cela on réutilisera le symbole `*` suivi du nom du pointeur : c'est ce qu'on appelle *déréférencement*. Ainsi dans le code suivant les deux `printf` sont équivalents :

```c
int ma_variable = 10;
int *mon_pointeur = &ma_variable; /* Déclaration et initialisation du pointeur */
printf("%d", ma_variable);
printf("%d", *mon_pointeur);
```

Sur le même principe, à la fin du code suivant, `ma_variable` vaudra 5 :

```c
int ma_variable = 10;
int *mon_pointeur = &ma_variable; /* Déclaration et initialisation du pointeur */
*mon_pointeur = 5 ;
```

## 2 Allocation dynamique de mémoire

En C, l'allocation dynamique de mémoire constitue la gestion manuelle de la mémoire en utilisant les fonctions dédiées `malloc`, `realloc`, `calloc` et `free`, déclarées dans la bibliothèque `stdlib.h`. Au cours de ces TPs, nous allons utiliser uniquement les fonctions `malloc` et `free`.

On a besoin de faire de l'allocation dynamique :

- quand on ne connaît pas à l'avance (au moment de la compilation) la taille des données ;
- quand il faut pouvoir modifier ces données (par exemple redimensionner un tableau) ;
- quand les données doivent pouvoir être manipulées en dehors de leur portée.

## 2.1 Allouer de l'espace mémoire

La fonction `malloc` va nous permettre d'allouer de l'espace mémoire pour nos variables. Voici son prototype :

```
1  void* malloc(size_t size);
```

La fonction `malloc` :

- *input :* une variable de type `size_t` représentant la taille de l'espace mémoire à allouer en octets (`size_t` est un alias pour désigner un entier non signé, et c'est précisément le type renvoyé par `sizeof`).
- *rôle :* essaie de réserver la quantité de mémoire demandée.
- *output :* renvoie l'adresse de début de l'espace mémoire nouvellement alloué, ou un pointeur nul si l'allocation a échoué. Le pointeur générique (`void*`) est un type "joker" qui peut être converti en pointeur de n'importe quel type.

En récupérant la valeur renvoyée par `malloc` dans un pointeur de type approprié, on va donc s'assurer que ce pointeur représente une adresse valide dans la mémoire ; si ce n'est pas le cas, on affiche un message d'erreur sur le flux standard d'erreur et on arrête l'exécution avec un code de retour différent de 0. Il ne faut pas perdre de vue, dans ce cas, les éventuelles variables déjà allouées dynamiquement et les libérer (voir ci-dessous).

## 2.2 Libérer l'espace mémoire alloué

Pour les variables allouées dynamiquement, la mémoire ne se libère pas automatiquement à la fin d'un bloc. L'espace est alloué pour toute la durée du programme (ce qui peut être très pratique), **mais avant la fin du programme il va falloir libérer la mémoire allouée en utilisant la fonction** `free`. En effet, ne pas libérer l'espace mémoire alloué dynamiquement entraîne des fuites de mémoire. Même si certains systèmes d'exploitation sont capables de récupérer cet espace mémoire, ne pas le faire soi-même est hasardeux et non professionnel.

La fonction `free` prend en argument **un pointeur sur l'espace mémoire alloué**.

Prenons l'exemple d'une structure représentant un point dans le plan :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct {
5      double x;
6      double y;
7  } Point2D;
8
9  int main() {
10     Point2D *p = NULL;
11
12     p = (Point2D *)malloc(sizeof(Point2D));
13     if (!p) {           /* Si p vaut NULL */
14         fprintf(stderr, "L'allocation de mémoire a échoué !\n");
15         exit(EXIT_FAILURE);
16     }
17
18     scanf("%lf %lf", &p->x, &p->y);
19     printf("x = %lf, y = %lf\n", p->x, p->y);
20
21     free(p);
22
23     return EXIT_SUCCESS;
24 }
```

Remarque :
à la différence d'une variable de type structure où on accède aux membres de la structure via l'opérateur "`.`" (par exemple `p.x`), **pour une variable de type pointeur de structure on utilise l'opérateur `->` (par exemple `p->x`)**.

# 3 Les listes chaînées

## 3.1 Les structures

Créez un fichier `liste.h` et placez-y les structures suivantes :

- `struct Maillon` : structure composée d'un champ `suivant` représentant un pointeur vers l'élément suivant, et d'un champ `valeur` de type entier ;
- `struct Liste` : structure composée d'un champ `tete` représentant la tête de la liste.

## 3.2 Les fonctions

Créez ensuite les fonctions suivantes dans un fichier `liste.c` avec les prototypes associés dans le fichier `liste.h`.

Le but dans un premier temps est d'écrire les fonctions `nouvelle_liste`, `nouveau_maillon`, `afficher_liste` et `free_liste`. Les autres fonctions seront minimales, c'est-à-dire qu'elles auront des corps vides (pour les fonctions de type `void`) ou vont juste se contenter de renvoyer 0 (pour les fonctions de type `int`).

1. `Liste *nouvelle_liste()` : crée (avec allocation mémoire) une nouvelle liste et renvoie un pointeur vers cette liste.
2. `Maillon *nouveau_maillon(int val)` : crée (avec allocation mémoire) un nouveau maillon qui a pour valeur l'entier `val` et renvoie un pointeur vers ce maillon.
3. `void afficher_liste(Liste *L)` : affiche la liste reçue en argument.
4. `void free_liste(Liste *L)` : libère la mémoire allouée à la liste (ainsi que pour tous ses maillons).
5. `void ajout_element_debut(Liste *L, int val)` : ajoute un élément de valeur `val` au début de la liste.
6. `void ajout_element_fin(Liste *L, int val)` : ajoute un élément de valeur `val` à la fin de la liste.
7. `int ajout_element(Liste *L, int val, int k)` : essaie d'ajouter un élément de valeur `val` à la $k^{\text{ème}}$ place de la liste (la première position valide étant 1) ; renvoie 1 si c'est possible et 0 sinon.
8. `void supp_element_debut(Liste *L)` : supprime le premier élément de la liste.
9. `void supp_element_fin(Liste *L)` : supprime le dernier élément de la liste.
10. `int supp_element(Liste *L, int k)` : essaie de supprimer le $k^{\text{ème}}$ élément de la liste (la première position valide étant 1) ; renvoie 1 si c'est possible et 0 sinon.

**Attention !** Votre fichier `liste.c` ne doit pas comporter de fonction `main` (voir ci-dessous).

## 3.3 Le programme de test

Téléchargez le programme de test depuis la page `https://www.lri.fr/~schevalier/teaching.html` ou directement via `https://www.lri.fr/~schevalier/doc/teaching/M1BIBS/A-et-C/2018/TP2/main_liste.c`. Ce programme possède une fonction `main` qui crée une liste à laquelle seront ajoutés et supprimés des éléments afin de tester les cas limites, comme par exemple l'ajout ou la suppression d'un élément à une position invalide, la suppression d'un élément alors que la liste est vide, etc.

À ce stade-là, vous pouvez compiler le tout avec :

```
1 gcc -Wall -g -o main_liste liste.c main_liste.c
```

L'option `-g` instruit `gcc` à produire l'exécutable avec des informations de *debug*, utiles lorsqu'on va utiliser des instruments de débogage tels que `valgrind` ou `gdb` qu'on va voir plus bas dans le TP.

Une fois cette étape finie, votre exécutable `main_liste` doit fonctionner sans erreurs (même si pour l'instant on ne peut pas réellement ajouter et supprimer des éléments).

Écrivez maintenant les autres fonctions. N'oubliez pas de compiler et d'exécuter le programme de test à chaque fois que vous avez fini d'écrire une fonction. Les résultats attendus sont fournis dans le fichier `out_liste.txt` disponible sur la page `https://www.lri.fr/~schevalier/teaching.html` ou directement via `https://www.lri.fr/~schevalier/doc/teaching/M1BIBS/A-et-C/2018/TP2/out_liste.txt`.

### 3.4 Ça marche... mais est-ce correct ?

Les *segfaults*, les erreurs *double free or corruption* et les fuites de mémoire sont les problèmes incontournables lorsqu'on manipule directement la mémoire en C.

Un des outils pouvant servir à identifier la source des erreurs, ainsi que les fuites de mémoire *même quand tout a l'air de fonctionner correctement*, est `valgrind`. Si vous avez compilé en passant l'option `-g` à `gcc` comme indiqué dans la section précédente, il n'y a rien de plus simple pour lancer cet outil. En supposant que votre exécutable s'appelle `main_liste`, lancez-le via `valgrind` comme ceci :

```
1 valgrind --track-origins=yes --leak-check=full ./main_liste
```

Si tout se passe bien, vous devriez voir quelque chose comme ceci (en plus du résultat de l'exécution de votre programme) :

```
==21980== HEAP SUMMARY:
==21980==     in use at exit: 0 bytes in 0 blocks
==21980==   total heap usage: 19 allocs, 19 frees, 1,304 bytes allocated
==21980==
==21980== All heap blocks were freed -- no leaks are possible
==21980==
==21980== For counts of detected and suppressed errors, rerun with: -v
==21980== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Sinon, c'est que vous avez des fuites de mémoire (*leaks*) et/ou des erreurs. *Welcome to the Dark Side !*