TP #3 – APL2

21 novembre 2018
Erwan Kerrien

Rappel : si vous avez des questions sur ce TP ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr (je consulte plus rarement mon mail Erwan.Kerrien@univ-lorraine.fr).

Le TP est à rendre pour demain matin 8h. Les questions notées sont en pages 6. Vous devez me renvoyer un fichier nommé `liste.c` qui répond à ces questions. Vous pouvez, si vous le souhaitez et si vous le jugez utile, m'envoyer un texte de commentaires. Ne pas le faire ne vous exposera à aucun retrait de points. En revanche, ne pas commenter vos codes d'une manière ou d'une autre pourrait le faire. Ce fichier est à déposer sur arche.

1 Objectifs du TP

Ce TP a pour objectif d'implémenter la sorte `Liste` vue lors du TD2.

Au préalable, il est nécessaire de voir comment coder une **structure** en C. En effet, la cellule d'une liste chaînée doit accueillir deux informations différentes : l'information à stocker d'une part (l'`Elément`) et la référence vers la cellule suivante (un pointeur).

La partie notée ne concerne que la fin du TP, et les listes chaînées (voir p. 6).

2 Structures

Les structures permettent de rassembler plusieurs éléments de types différents (soit un type de base, soit une structure) en un seul nouveau type complexe. Ces éléments sont appelés *membres*. Les structures sont ainsi des embryons d'objets, tels qu'on peut les retrouver dans des langages vraiment objets comme Java ou Python[1].

Nous avons vu lors du dernier TP tous les types dits simples définis par défaut par C. Ces types permettent de définir les membres d'une structure, auxquels on peut ensuite accéder par l'opérateur . , comme le montre l'exemple suivant (compilez et lancez-le) :

```c
// fichier struct1.c
#include <stdio.h>

struct temps
{
  int heures;
  int minutes;
  float secondes;
};
typedef struct temps temps;

void TempsAfficher(temps t)
{
  printf("Il est %d heures, %d minutes, %f secondes\n",t.heures, t.minutes, t.secondes);
}

temps TempsCreer(int Heures, int Minutes, float Secondes)
{
  temps t;
  t.heures = Heures;
  t.minutes = Minutes;
  t.secondes = Secondes;
  return t;
}
```

1. Ce ne sont pas de vrais objets en C car on ne peut y stocker des méthodes. On peut certes définir des membres de types fonction, mais d'une part c'est une notion qui dépasse le cadre de ce cours d'introduction au C, et d'autre part un membre fonctionnel n'est pas une méthode.

```
25
26 int main()
27 {
28   temps t1;
29   t1.heures=12;
30   t1.minutes=46;
31   t1.secondes=38.64;
32   printf("(Sans fonction) Il est %d heures, %d minutes, %f secondes\n",t1.heures, t1.minutes, t1.second
33
34   temps t2={12,46,38.64};
35   TempsAfficher(t2);
36
37   temps t3=TempsCreer(12,46,38.64);
38   TempsAfficher(t3);
39
40   return 0;
41 }
```

Ici une structure est employée pour stocker un instant (*temps*) en heures, minutes et secondes (ces dernières en nombre décimal afin de pouvoir tout gérer). La structure `temps` est définie lignes 4 à 9. Notez le ; en fin de ligne 9 qui doit être présent pour conclure la définition de la structure.

Une structure définie de cette manière définit un nouveau type. Son nom est `struct temps`. Normalement, une variable, nommée par exemple `t`, devrait être déclarée comme suit :

```
1 struct temps t;
```

Cela fait une lourdeur de notation qui peut être vite pénible. C propose heureusement un mécanisme de déclaration d'alias de types grâce à l'instruction `typedef` qui a pour syntaxe `typedef <AncienType> <NouveauType>`. On peut donc l'utiliser pour définir `temps` comme type à la place de `struct temps`, de la manière suivante (voir aussi l. 10 du code ci-dessus) :

```
1 typedef struct temps temps;
```

Il n'y a aucune obligation à reprendre le nom que vous avez donné à la structure. Vous auriez tout aussi bien pu définir un alias `instant` pour la `struct temps`, de la manière suivante

```
1 typedef struct temps instant;
```

Avec les lignes 4 à 10, on définit donc un nouveau type, nommé `temps`, que l'on peut utiliser comme n'importe quel autre type, sauf que les opérations qui sont disponibles de base sur les types habituels (affichage, somme et produit pour les types numériques, etc...) ne le sont pas pour ce nouveau type qui vient "nu". C'est donc de la responsabilité du programmeur de les écrire. Par exemple, les lignes 12 à 15 définissent la fonction d'affichage (équivalent du `printf`).

Le `main`, l. 26 à 41, montre trois manières d'initialiser une variable de type `temps`.

- De la ligne 28 à 31, on déclare une variable `t1` de type `temps`, puis on initialise chaque membre de la structure en utilisant les affectations classiques puisque ces membres sont de types classiques (`int` et `float`) prédéfinis par le langage C. Puis en ligne 32 on affiche la structure avec une instruction `printf` qui est du coup assez compliquée. La longueur et complexité de ces lignes en sont le souci.
- La ligne 34 déclare une nouvelle variable `t2` de type `temps` et l'initialise. Vous noterez l'usage d'accolades. La difficulté de cette manière de procéder est qu'il faut lister les valeurs d'initialisation des membres, dans l'ordre de leur déclaration dans la structure `temps`. Ensuite, l. 35, on en affiche la valeur, cette fois-ci en utilisant la fonction d'affichage que nous avons définie l. 12 à 15. Vous remarquerez que le nouveau type `temps` est utilisé comme n'importe quel autre type pour déclarer le paramètre `t` de cette fonction (l. 12).
- Enfin, l.37, on définit une dernière variable `t3`, encore de type `temps`, sauf que cette fois-ci, nous appelons une fonction `TempsCreer`, dédiée à l'initialisation de cette variable. Ceci améliore la lisibilité du code. Vous remarquerez à cette occasion, que la fonction `TempsCreer`, définie l. 17 à 24, renvoie une valeur de type `temps` (l. 17).

Cette dernière manière de faire est la plus lisible. De plus, on voit ici comment on définit un nouveau type à travers une spécification (définition de la structure), et des opérations dédiées (`TempsAffichage` et `TempsCreer`, mais on pourrait en écrire plus, voir les extensions possibles ci-dessous...). Ces opérations permettent de manipuler aisément et de manière naturelle les variables du nouveau type défini.

Ceci fonctionne très bien, mais peut poser problème lorsque la structure est “lourde”, c'est-à-dire que l'espace de stockage d'une variable de ce type devient important. En effet, les variables étant passées par valeur, tout appel de la fonction `TempsAffichage` par exemple va impliquer la création d'une variable locale (nommée `t`) et la recopie de la valeur passée lors de l'appel de cette fonction. Il y a donc allocation d'un certain espace mémoire et recopie de valeurs, ce qui peut être coûteux à la longue.

Une façon de réduire ces coûts est d'utiliser des pointeurs. Cela permet un passage par variable pour la fonction `TempsAfficher` par exemple, ce qui implique la simple recopie d'un pointeur. La fonction `TempsCreer` quant à elle impose de passer par une allocation mémoire explicite si on veut qu'elle renvoie un pointeur. Ceci donne le code suivant :

```
1  // fichier struct2.c
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  struct temps
6  {
7    int heures;
8    int minutes;
9    float secondes;
10 };
11 typedef struct temps temps;
12
13 temps * TempsCreer(int Heures, int Minutes, float Secondes)
14 {
15   temps *t;
16   t=malloc(sizeof(temps));
17   t->heures = Heures;
18   t->minutes = Minutes;
19   t->secondes = Secondes;
20   return t;
21 }
22
23 temps * TempsDetruire(temps *t)
24 {
25    if (t) free(t);
26    return (temps*)NULL;
27 }
28
29 void TempsAfficher(temps *t)
30 {
31   printf("Il est %d heures, %d minutes, %f secondes\n",t->heures, t->minutes, t->secondes);
32 }
33
34 int main()
35 {
36   temps *t1=malloc(sizeof(temps));
37   t1->heures=12;
38   t1->minutes=46;
39   t1->secondes=38.64;
40   printf("(Sans fonction) Il est %d heures, %d minutes, %f secondes\n",t1->heures, t1->minutes, t1->sec
41   free(t1);
42   printf("Après free: %p\n",t1);
43
44   temps *t2=TempsCreer(0,0,0);
45   *t2=(temps){12,46,38.64};
46   TempsAfficher(t2);
47   TempsDetruire(t2);
48   printf("Après Detruire sans affectation: %p\n",t2);
49
50   temps *t3=TempsCreer(12,46,38.64);
51   TempsAfficher(t3);
52   t3=TempsDetruire(t3);
53   printf("Après Detruire avec affectation: %p\n",t3);
```

```
54
55    return 0;
56  }
```

La première remarque concerne `TempsCreer` dans laquelle je dois appeler `malloc`. En effet, on aurait pu penser retourner un pointeur en écrivant

```
1  temps * TempsCreer(int Heures, int Minutes, float Secondes)
2  {
3    temps t={Heures,Minutes,Secondes};
4    return &t;
5  }
```

Cette façon de procéder ne fonctionne pas puisqu'on renvoie un pointeur vers la variable `t` qui est locale à la fonction et qui est donc détruite dès la fin de cette fonction : tenter d'y accéder depuis le `main` échouera puisque cette zone de la mémoire n'est plus réservée au programme (ou si elle l'est, c'est pour autre chose...).

Par ailleurs, l'usage de `malloc` dans la fonction `TempsCreer` impose un usage en miroir de la fonction `free` pour libérer cette mémoire. C'est ce qui est fait dans la fonction `TempsDetruire`. Dans cette fonction, je teste si le pointeur passé n'est pas nul, autrement dit si le pointeur est valide puisque `malloc` renvoie `NULL` en cas d'échec. Ceci assure que la fonction `free` va bien agir sur une zone mémoire allouée au programme. Afin de rendre cette convention valable partout dans mon programme, une bonne pratique est de renvoyer le pointeur `NULL`, donc invalide, par la fonction de libération de la mémoire, ce qui permet de bien réinitialiser à `NULL` tout pointeur que je viens de détruire (et donc ne pas avoir d'erreur si je tente de le détruire deux fois). Ceci est fait ligne 52, les lignes précédentes montrant ce qui se passe si on ne le fait pas (ni `t1`, ni `t2` ne sont automatiquement remis à `NULL` lors de leur déallocation lignes 41 et 47).

Exercice : Extensions possibles (Non noté)

Cette partie n'est pas notée, et n'est donc pas obligatoire, mais si vous souhaitez approfondir la notion de structure, vous pouvez repartir du code de `struct1` et écrire :

1. Une fonction `TempsCreerSecondes` qui prend en argument un `float` : ce nombre est un nombre de secondes qui sera traduit en heures, minutes et secondes. La fonction renverra un `temps` qui stockera cette information.
2. Une fonction `TempsEnSecondes` qui fait le contraire : elle prend en argument une variable de type `temps` et traduit le temps donné en heures, minutes et secondes en un temps en secondes qui est renvoyé en sortie sous forme de `float`.
3. Une fonction `TempsPrecede` qui prend en entrée deux variables `t1` et `t2` de type `temps` et renvoie un booléen (de type `unsigned char` en C : Vrai (1) si `t1` est un temps qui précède `t2`, Faux (0) sinon.
4. Rien ne m'empêche de donner de mauvais arguments à la fonction `TempsCreer`. Écrivez une fonction `TempsValider` qui prend en entrée une variable de type `temps` et la transforme pour que ses informations soient valides : les secondes et les minutes sont des nombres positifs compris entre 0 et 60 (strictement). Les heures négatives sont autorisées, permettant de coder une durée négative.
5. On peut alors écrire des fonctions `TempsAjouter` et `TempsSoustraire` qui permet de faire des manipulations algébriques sur les `temps`...

3 Listes chaînées

Afin d'implanter une liste chaînée, je vous fournis le fichier `liste.c` qui contient un squelette de code avec tout le préalable nécessaire, et notamment les opérations de la sorte `Liste`, ainsi que le prototype des fonctions à implémenter. Le code est intégralement commenté, mais nous pouvons le passer brièvement en revue.

Nous commençons par deux instructions `#include` qui permettent de déclarer les fonctions externes qui sont utiles, et habituelles : `stdio.h` donne accès à la fonction `printf` et `stdlib.h` donne accès aux fonctions `malloc`, `free`, et `strtol` (voir la fonction `main`), et enfin `#include` définit la constante `NAN` employée pour définir l'élément invalide (l. 19).

```
1  // déclaration des fonctions externes
2  #include <stdio.h> // nécessaire pour printf
3  #include <stdlib.h> // nécessaire pour malloc, free, et strtol
4  #include <math.h> // nécessaire pour NAN
```

Pour plus de clarté du code et plus de facilité d'écriture, nous définissons très rapidement un type `Booleen` comme un `unsigned char` qui encode `Vrai` par 1 et `Faux` par 0.

```
6  // DEFINITION D'UN TYPE BOOLEEN : un unsigned char qui encode Vrai par 1 et Faux par 0
7  typedef unsigned char Booleen; // après cette ligne Booleen est un nouveau type équivalent
8                                 // à unsigned char
9  Booleen Vrai=1; // Encodage de Vrai par 1
10 Booleen Faux=0; // Encodage de Faux par 0
11 Booleen Non(Booleen b) // fonction NOT (utilisée dans EstDans())
12 {
13   return b==Vrai ? Faux : Vrai; // Note: emploi de l'opérateur ternaire
14                                 // x ? y : z <=> si x alors y sinon z
15 }
```

Ensuite, également par souci de clarté, nous définissons, un type `Element`. Ce type encode un `element_invalide` en tant que `NAN`. Nous définissons aussi une fonction pour afficher cet `Element` : ceci peut sembler inutile, mais je testerai votre code en changeant cette définition d'un élément, et j'utiliserai la fonction `AfficherElement` pour afficher mon élément personnel.

```
17 // DEFINITION D'UN TYPE ELEMENT : un float
18 typedef float Element; // après cette ligne Element est un nouveau type équivalent à float
19 Element element_invalide=NAN; // Encodage de l'element_invalide par NAN=Not a Number (voir math.h)
20 void AfficherElement(Element E) // fonction d'affichage d'un élément (utilisée dans Afficher())
21 {
22   if (E == element_invalide)
23     printf("INVALID\n");
24   else
25     printf("%f\n", E);
26 }
```

Les lignes suivantes définissent les types nécessaires à l'implémentation de la sorte liste chaînée, que je nomme `Liste`. On commence par définir, l. 30 à 34, la cellule par une structure qui contient un `Element`, stocké dans le membre `data` et un pointeur vers la cellule suivante, stocké dans le membre `next`. La dernière ligne, l. 35, rend le nouveau type `Cellule` équivalent au type `struct Cellule` précédemment défini.

```
28 // DEFINITION DES SORTES POUR LA LISTE CHAINEE
29 // La cellule
30 struct Cellule
31 {
32   Element data;
33   struct Cellule *next;
34 };
35 typedef struct Cellule Cellule;
36
37 // la liste
38 typedef Cellule *Liste;
39 Liste liste_vide=(Liste)NULL;
```

Nous définissons ensuite, l. 38, le type `Liste` comme un pointeur vers une `Cellule` : ainsi le champ `next` d'une `Cellule` est bien une `Liste`. La ligne 39 définit la constante `liste_vide`, de type `Liste`, comme un pointeur nul (`NULL`=0)

```
37 // la liste
38 typedef Cellule *Liste;
39 Liste liste_vide=(Liste)NULL;
```

Ensuite les lignes 41 à 81 définit toutes les opérations sur la sorte `Liste`, telles que nous les avons vues en TD : `Creer` lignes 42 à 50, `EstVide` lignes 52 à 59, `Contenu` lignes 61 à 66, `Succ` lignes 68 à 72 et enfin `Detruire` liges 74 à 80. Vous pourrez vérifier que ces opérations obéissent aux propriétés et préconditions données dans le TD. Notez aussi les commentaires...

Enfin, chaque fonction vue en TD est commentée. Je donne leur prototype, mais charge à vous d'écrire le corps de ces fonctions, afin d'implanter ce que nous avons vu en TD (et ce que nous n'avons pas eu le temps de voir, mais dont vous avez le corrigé). Les fonctions à écrire sont : `Afficher`, `Longueur`, `Rechercher`, `Supprimer`, `Ajouter`, `Inverser` et `Vider`. La fonction `EstDans` est écrite mais commentée afin de ne pas gêner vos premières compilations. Dès lors qu'une fonction est écrite et validée, vous pouvez l'employer, si vous le désirez, dans l'écriture d'une autre fonction.

Enfin, la fonction `main` (ligne 146 à fin) est un exemple de programme minimal que vous pouvez utiliser pour tester vos fonctions.

Ce code ne compile pas : vous devez écrire a minima les fonctions `Ajouter`, `Longueur` et `Afficher` pour que cela marche. Vous pouvez bien évidemment modifier ce code en fonction de vos besoins de test. J'évaluerai votre code en insérant mon propre code de fonction `main`, en le compilant, puis en l'exécutant.

4 Question notée

Écrivez toutes les fonctions sur les listes qui sont commentées. Un bon ordre à suivre est celui donné dans le fichier `liste.c`, mais vous pouvez suivre le vôtre. Basez-vous sur le travail d'algorithmique fait pendant le TD2. Vous me renverrez votre version du fichier `liste.c` (écrit comme cela).

Éléments de notation :

- le nom du fichier (`liste.c`) et celui des fonctions, ainsi que leur prototype (type de sortie et nombre et types des paramètres) ne doivent pas être changés : mes scripts automatiques seront impitoyables.
- assurez-vous a minima que le fichier que vous m'envoyez compile. Faites attention en particulier aux typos que vous insérez parfois dans votre code en le commentant au dernier moment.
- enfin, j'attends un code commenté : la qualité et la pertinence des commentaires entrera dans la notation.