TP – DEV : Les structures

28 novembre 2023
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 4 et 6. Vous devez me renvoyer deux fichiers nommés `struct1.c` et `liste.c` qui répondent 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. Ces fichiers sont à déposer sur arche.

1 Objectifs du TP

L'objectif est d'implémenter la sorte Liste vue lors du TD9&10. Nous allons le faire sur deux TP. Le TP d'ajourd'hui a pour but de définir un type Liste et d'implémenter les opérations de la sorte.

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 (le ou les `Elément`) et la référence vers la cellule suivante (un pointeur). Ces informations étant de type différent, on ne peut pas les stocker ensemble dans un tableau. La structure est faite pour cela.

2 Structures

Dans cette partie, les questions notées sont p.4. Vous rendrez un fichier `struct1.c`.

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;
```

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.

```
23    t.secondes = Secondes;
24    return t;
25  }
26
27  int main()
28  {
29    temps t1;
30    t1.heures=12;
31    t1.minutes=46;
32    t1.secondes=38.64;
33    printf("(Sans fonction) Il est %d heures, %d minutes, %f secondes\n",
34           t1.heures, t1.minutes, t1.secondes);
35
36    temps t2={12,46,38.64};
37    TempsAfficher(t2);
38
39    temps t3=TempsCreer(12,46,38.64);
40    TempsAfficher(t3);
41
42    return 0;
43  }
```

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",
32          t->heures, t->minutes, t->secondes);
33 }
34
35 int main()
36 {
37   temps *t1=malloc(sizeof(temps));
38   t1->heures=12;
39   t1->minutes=46;
40   t1->secondes=38.64;
41   printf("(Sans fonction) Il est %d heures, %d minutes, %f secondes\n",
42          t1->heures, t1->minutes, t1->secondes);
43   free(t1);
44   printf("Après free: %p\n",t1);
45
46   temps *t2=TempsCreer(0,0,0);
47   *t2=(temps){12,46,38.64};
48   TempsAfficher(t2);
49   TempsDetruire(t2);
50   printf("Après Detruire sans affectation: %p\n",t2);
```

```
51
52    temps *t3=TempsCreer(12,46,38.64);
53    TempsAfficher(t3);
54    t3=TempsDetruire(t3);
55    printf("Après Detruire avec affectation : %p\n",t3);
56
57    return 0;
58  }
```

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 : Questions notées

Repartir du code de `struct1` et écrire les fonctions suivantes (vous rendrez le résultat sous la forme d'un fichier nommé `struct1.c`) :

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. On rappelle qu'il y a 3600 secondes dans une heure et 60 secondes dans une minutes.

   On pourra utiliser la fonction `floor` disponible dans `math.h` (directive `#include`). Attention : cette fonction renvoie un `double` et il faut donc le convertir en `int`. Il faut aussi compiler avec une nouvelle option pour pouvoir l'utiliser : `gcc struct1.c -o struct1 -lm`. Cette option indique d'utiliser la librairie mathématique `libm.so`.
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 `TempsComparer` qui prend en entrée deux variables `t1` et `t2` de type `temps` et renvoie -1 si `t1` vient après à `t2`, 0 si les deux temps sont égaux, et +1 si `t1` vient avant `t2`
4. Rien n'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. Par exemple, cette fonction transformera le temps (mauvais) 0h 63 min -2 sec en 1h 2 min et 58 sec. Autre exemple : 0h -63 min 0 sec sera transformé en -2 h 57 min 0 sec.
5. On peut alors écrire des fonctions `TempsAjouter` et `TempsSoustraire` qui permet de faire des manipulations algébriques sur les `temps` : par exemple, `TempsAjouter` permettrait de calculer combien font 1h 43 min 57 s et 4h 18 min et 18 secondes. Indice : on pourra utiliser avec profit les fonctions `TempsCreerSecondes` et `TempsEnSecondes`. Ces fonctions doivent renvoyer une valeur de type `temps`
6. **Bonus en option** : répondez aux mêmes questions mais en partant du fichier `struct2.c` (emploi des pointeurs).

3 Listes chaînées

Dans cette deuxième partie, il vous est demandé d'implémenter la sorte Liste et ses opérations, ce qui permettra d'implémenter toutes les fonctions vues en TD lors du prochain TP. Les questions sont p.6, et vous rendrez un fichier `liste.c`.

La sorte Liste utilise les sortes booléen et Element. Le type booléen est défini par inclusion du fichier header `stdbool.h` (`#include <stdbool.h>`). Il faut en revanche spécifier ce qu'est un Element. Nous allons ici utiliser un char[], autrement dit, notre liste chaînée servira à stocker des chaînes de caractères. Plus précisément, on va utiliser le type char*, qui va nous demander de gérer nous-mêmes la mémoire associée à ces chaînes de caractères. Les fonctions dont nous avons besoin sont données par la signature de la sorte Element, que je vous rappelle ici et auxquelles je rajoute les fonctions `ElementCopie` et `ElementDetruire` :

```
Sorte : Elément
Utilise : booléen
Opérations :
 élément_invalide : -> Elément
 ElémentEstValide : Elément -> booléen
 ElémentAfficher  : Elément ->
 ElémentComparer  : Elément x Elément -> Booléen
 ElémentCopie     : Elément -> Elément
 ElémentDétruire  : Elément -> Elément
```

Les spécifications des opérations sur un Element sont les suivantes :

- element_invalide doit renvoyer quelque chose de type Element qui soit bien identifié comme une valeur interdite. Comme le type utilisé est char*, l'élément invalide sera ici le pointeur `NULL`.
- `ElementEstValide` renverra Faux si l'Element est element_invalide, et Vrai sinon.
- `ElementAfficher` affichera l'Element passé en argument. On utilisera la fonction `printf`. Dans le cas où l'Element en arguent n'est pas valide, la fonction affichera "<INVALIDE>".
- `ElementComparer` compare deux Element et renvoie Vrai s'ils sont identiques, et Faux sinon. On utilisera ici la fonction `strcmp`.
- `ElementCopie` effectue une copie profonde de l'Element passé en argument, c'est-à-dire que la mémoire est allouée pour un nouvel Element, puis une copie de l'argument est faite, et le nouvel Element est renvoyé. Cette fonction n'alloue aucune mémoire, et renvoie element_invalide dans le cas où l'Elément passé en argument n'est pas valide. Indice, vous pourrez utiliser la fonction `strdup`.
- `ElementDetruire` désalloue la mémoire réservée pour l'Element si celui-ci est valide, et renvoie element_invalide dans tous les cas.

Ensuite, je vous rappelle la signature de la sorte Liste :

```
Sorte : Liste
Utilise : Elément, booléen
Opérations :
  liste_vide  :  -> Liste
  EstVide     : Liste -> booléen
  ContenuLire : Liste -> Elément
  ContenuModif: Liste x Elément ->
  SuccLire    : Liste -> Liste
  SuccModif   : Liste x Liste ->
  Créer       : Elément x Liste -> Liste
  Détruire    : Liste -> Liste
```

On commencera par définir une structure permettant de stocker une cellule, c'est-à-dire regroupant un Element (char*) et un pointeur vers une structure cellule comme successeur.

Puis on définira le type Liste comme un pointeur vers une cellule.

Par ailleurs, voici les spécifications pour ces opérations :

- liste_vide doit renvoyer une liste vide. Ici, ce sera un pointeur `NULL`.
- `EstVide` renvoie Vrai si la Liste passée en argument est liste_vide, et Faux sinon.

— `ContenuLire` va renvoyer l'Element contenu dans la première cellule de la liste. element_invalide sera renvoyé si la liste est vide.
— `ContenuModif` va écrire un Element dans la première cellule de la liste. Il ne se passe rien si la liste est vide ou si on essaie de stocker element_invalide. La copie doit être profonde. On veillera ici à bien gérer la mémoire, notamment si la cellule contient déjà un Element qu'il s'agira de désallouer au préalable.
— `SuccLire` va renvoyer la Liste suivante, c'est-à-dire celle qui commence avec la deuxième cellule de la liste. Cette fonction renverra liste_vide si la liste en entrée est vide.
— `SuccModif` va stocker la Liste passée en deuxième argument comme successeur de la première cellule de la liste. Cette fonction ne fait rien si la liste en premier argument est vide. En revanche, on peut bien stocker une liste vide et donc la passer en deuxième argument. Ici, il faut utiliser une copie superficielle pour que la fonctionnement en liste chaînée soit effectif.
— `Creer` va en effet créer une cellule, ce qui implique, ainsi que nous l'avons vu en cours, d'allouer la place mémoire nécessaire pour une cellule, pour ensuite l'initialiser avec les arguments : l'Element en contenu et la Liste en successeur). Le pointeur vers cette cellule sera renvoyé, c'est-à-dire une Liste.
— `Detruire` va faire l'opération inverse sur la mémoire, c'est-à-dire qu'elle va libérer la mémoire allouée pour la première cellule de la liste et renvoyer le successeur de cette cellule, c'est-à-dire le reste de la liste. Attention ici à bien libérer la mémoire réservée pour l'Element stocké s'il est valide.

Exercice : Questions notées

— définissez un type Element comme équivalent à un char* (avec une `typedef`) ;
— définissez une constant element_invalide comme un pointeur `NULL` de type Element ;
— écrivez les fonctions de la sorte Element : `ElementEstValide`, `ElementAfficher`, `ElementComparer`, `ElementCopie`, et `ElementDetruire` ;
— définissez une structure cellule qui permet de stocker un Element dans un champ `data`, et un pointeur vers une structure cellule dans un champ `next` ;
— définissez un type Liste comme équivalent à un pointeur sur une structure cellule (avec un `typedef`) ;
— définissez une constante liste_vide comme le pointeur `NULL` de type Liste ;
— écrivez toutes les opérations de la sorte Liste : `EstVide`,`Creer`,`Detruire`,`ContenuLire`,`ContenuModif`,`SuccLire`, `SuccModif`. Vous veillerez à bien respecter les spécifications données plus haut. Vous n'utiliserez que les opérations de la sorte Element pour gérer l'Element stocké.

Votre code sera écrit dans un seul fichier, nommé `liste.c`, qui sera rendu.