Chapitre 5

Structures de données dynamiques et pointeurs

Nous avons vu la notion de *structures de données* et nous avons étudié les structures de données dont la manipulation est prévue en tant que telle dans la plupart des langages de programmation. Nous avons fait, en particulier, la distinction entre les *structures de données statiques* et les *structures de données dynamiques*. De plus les seules structures de données dynamiques que nous ayons rencontrées jusqu'à maintenant sont les fichiers, qui concernent la mémoire de masse.

Les autres structures de données dont on peut avoir besoin sont appelées **structures de données abstraites**. L'adjectif fait référence au fait qu'elles ne sont pas implémentées dans le langage de programmation : il faut d'abord constituer un cahier des charges précis sur leur manipulation puis ensuite les implémenter. L'étude des structures de données abstraites les plus courantes, parmi celles qui ont été dégagées au cours de l'histoire de la programmation, fait l'objet d'un cours à part entière, de taille analogue au cours d'initiation à la programmation. Nous renvoyons donc à un tel cours pour leur étude.

L'implémentation des structures de données dynamiques est facilitée par la gestion de la mémoire. Nous avons déjà fait remarquer à plusieurs reprises que le programmeur n'a pas à entrer dans les détails du matériel mais ne doit considérer que les besoins de la programmation. Un bon compromis, en ce qui concerne la programmation, entre les structures de données dynamiques toutes faites et la gestion directe de la mémoire (dépendant, en particulier, du matériel utilisé) est l'utilisation des *pointeurs*. Nous allons étudier cette notion de pointeur dans ce chapitre.

En fait les pointeurs sont déjà apparus plus tôt à propos du passage par adresse des arguments des fonctions. Ce dernier point est une spécificité du langage C.

5.1 Pointeur et indexation indirecte

L'inconvénient avec une structure de données statique telle qu'un tableau est qu'il faut prévoir une taille maximum dès le départ. Une exécution du programme occupera la même place mémoire quelle que soit la session de travail. Le concept d'*indexation indirecte* permet une première amélioration.

5.1.1 Notion générale

Comme nous venons de le rappeler nous avons déjà rencontré les pointeurs, puisque nous illustrons ce cours d'initiation à la programmation par le langage C, à propos du passage des arguments par adresse. Nous rappelons cependant ici les notions essentielles les concernant. De plus nous abordons la notion d'*appel de pointeur*.

5.1.1.1 Intérêt des pointeurs

Principe.- Nous avons vu, en ce qui concerne la désignation des entités, la notion de *référence directe* grâce à une *variable*. Ceci correspond en gros à une boîte noire dans laquelle on peut placer certaines entités, le *type* de la variable indiquant la taille des entités que l'on peut y placer.

L'**indexation indirecte** repose sur le principe suivant. On utilise une nouvelle sorte de variables, dites variables **pointeur**. Le contenu d'une telle variable ne sera pas une entité du type voulu mais l'*adresse* de cette entité (dite entité **pointée**).

L'intérêt de cette façon de faire est que le contenu d'une telle variable pointeur a une taille fixe (celle d'une adresse), quel que soit le type de l'entité pointée (qui peut être de taille très grande).

Types pointeurs.- À tout type correspond un type pointeur. L'intérêt d'un type pointeur est le suivant. Le contenu d'une variable d'un type pointeur est de taille fixe, il suffirait donc *a priori*, d'un seul type pointeur. Cependant lorsqu'on réservera l'emplacement pour l'entité pointée, il faudra en connaître la taille, qui est indiquée par le type du type pointeur.

Déclaration d'une variable de type pointeur.- Une variable de type pointeur, ou plus concisément un **pointeur**, se déclare comme n'importe quelle variable.

Accès au contenu pointé.- La référence à une variable pointeur donnera l'adresse de l'entité voulue et non l'entité elle-même. Il faut donc un moyen pour désigner cette entité. Celui-ci dépend du langage considéré.

Dans beaucoup de langages de programmation, l'adresse n'est pas accessible directement. Le langage C est, à cet égard, un langage à part.

Pointeur nul.- On a quelquefois besoin d'une valeur spéciale de pointeur qui ne pointe sur rien : on l'appelle le **pointeur nul**. Il y a *a priori* une valeur nulle par type de pointeurs mais, dans la plupart des langages de programmation, tous ces pointeurs nuls portent le même nom.

5.1.1.2 Appel et libération des pointeurs

Initialisation d'un pointeur.- Une variable d'un type simple s'initialise directement, soit par lecture, soit par affectation. Il est dangereux d'attribuer directement une adresse à un pointeur, car celle-ci peut avoir été choisie par le système d'exploitation et le compilateur pour une autre variable ou, pire, pour une partie du programme. Cela peut cependant se faire dans certains langages orientés programmation système, tel que le langage C (pour pouvoir accéder aux cartes

son ou graphique, par exemple). En général cependant, en ce qui concerne la programmation pure, l'initialisation d'un pointeur se fait par **appel de pointeur** : on indique que l'on veut une initialisation du pointeur ; une recherche est effectuée automatiquement par le système d'exploitation pour déterminer une région de la mémoire vive dans laquelle un certain nombre d'octets consécutifs ne sont pas réservés pour autre chose et correspondent à la taille du type de l'entité pointée ; l'adresse du premier octet est placée dans la variable pointeur ; le nombre adéquat d'octets (connu par le type du pointeur) est réservé par le système d'exploitation (qui ne pourront donc pas être utilisés par autre chose).

Libération des pointeurs.- Une fois qu'on n'a plus besoin d'utiliser un pointeur initialisé par appel, il faut penser à libérer la place réservée, sinon on risque d'arriver très rapidement à une saturation de la mémoire vive pour rien (et de façon inexplicable à l'utilisateur).

En effet, l'utilisation des pointeurs permet d'accéder à un nombre d'emplacements mémoire dépendant de la session d'utilisation du programme. Ceci permet d'accéder à beaucoup de mémoire. Mais la mémoire n'est pas infinie. On risque de dépasser la capacité du matériel, surtout si on ne fait pas attention. Or certains emplacements peuvent être utiles temporairement, sans qu'on en ait besoin tout au long de la session. On aimerait donc pouvoir les réserver lorsqu'on en a besoin, et les laisser libres pour d'autres activités à d'autres moments.

Une instruction spécifique permet la **libération** de l'emplacement pointé par un pointeur donné. On obtient ainsi une véritable **gestion dynamique de la mémoire**.

5.1.2 Mise en place des pointeurs en langage C

5.1.2.1 Type et variable pointeurs

Type pointeur.- À tout type τ (dit alors **type de base**) est associé un nouveau type dont les éléments sont des pointeurs sur des éléments du premier type. Pour désigner ce type il suffit de faire suivre le nom du type de base du symbole '*' pour obtenir $\tau*$.

Exemple.- Le type pointeur sur les entiers sera `int*`.

Variable pointeur.- Une variable pointeur se déclare comme toute autre variable.

Exemple.- On a par exemple :

```
int* ptr ;
```

pour indiquer que `ptr` est une variable de type pointeur sur un entier. On remarquera la désignation traditionnelle `ptr` de la variable, évidemment pour *PoinTeuR*, ou plus exactement pour *PoinTeR*.

Déplacement de l'astérisque.- En fait il est traditionnel d'écrire :

```
int *ptr ;
```

en déplaçant le symbole '*'.
Cette façon de faire permet de faire des déclarations multiples telles que :

```
int *ptr, n ;
```

indiquant que `ptr` est une variable de type pointeur sur un entier et `n` une variable de type entier.

Accès au contenu pointé.- Pour accéder à l'entité pointée par un pointeur, on fait précéder le nom du pointeur par le symbole '*'.

Exemple.- L'affectation :

```
*ptr = 5 ;
```

permet d'initialiser l'entité pointée par la variable `ptr`.

Pointeur nul.- Les pointeurs nuls sont tous notés `NULL` en langage C, quel que soit leur type.

5.1.2.2 Appel de pointeur

Introduction.- La fonction `malloc()` (pour l'anglais *memory allocation*, c'est-à-dire allocation de mémoire) permet l'appel de pointeur. La déclaration de cette fonction se trouve dans le fichier en-tête `stdlib.h` (pour *STandarD LIBrary*), non utilisé jusqu'à maintenant.

Syntaxe.- La syntaxe de cette fonction est :

```
ptr = malloc(entier) ;
```

où `entier` est une expression entière (positive) et `ptr` un pointeur.

Sémantique.- La signification de cette instruction est la suivante : un bloc mémoire de `entier` octets disponibles (consécutifs) est réservé et son adresse (un pointeur) est renvoyée à `ptr`. Bien entendu cela se déroule correctement s'il reste suffisamment de place disponible en mémoire, sinon le pointeur `NULL` est retourné.

5.1.2.3 Libération des pointeurs

Introduction.- La libération d'un pointeur se fait en langage C grâce à la fonction `free()`. La déclaration de cette fonction se trouve également dans le fichier en-tête `stdlib.h`.

Syntaxe.- La syntaxe de cette fonction est :

```
free(ptr) ;
```

où `ptr` est un pointeur.

5.1.3 Un exemple d'utilisation : tableau de pointeurs

5.1.3.1 Principe

Le problème.- On veut disposer en mémoire centrale d'un répertoire téléphonique. Chaque composant de ce répertoire comportera un nom et un numéro de téléphone.

Analyse du problème.- Les structures de données à lesquelles on peut penser pour ce programme sont un type structuré avec les champs `nom` et `telephone`, et un tableau `repertoire` dont les éléments sont de ce type structuré.

Il faut décider d'une dimension maximum pour ce tableau, par exemple 500.

Inconvénients.- Cette façon de faire réserve beaucoup de place mémoire, ce qui risque de laisser peu de place pour faire autre chose. Ceci peut, par exemple, ralentir considérablement une impression dans le cas d'une utilisation de l'ordinateur en mode multitâche.

Utilisation d'un tableau de pointeurs.- Une autre façon de faire, permettant d'économiser la mémoire non réellement nécessaire, est d'utiliser un tableau dont le type d'un élément est, non pas une structure, mais un type pointeur sur une telle structure. On ne fera pointer de façon effective que pour les emplacements nécessaires.

5.1.3.2 Un programme

Ceci nous conduit au programme suivant, pour l'initialisation de ce répertoire (en s'arrêtant sur un nom commençant par '#') puis pour l'affichage des données ainsi initialisées :

```
/* malloc.c */
#include <stdio.h>
#include <stdlib.h>

void main(void)
     {
     struct donnee
            {
            char nom[30];
            char tel[20];
            };
     struct donnee *repertoire[500], individu;
     int    i, j;

/* initialisation */

     i = -1;
     do
        {
        printf("\nNom       : ");
        scanf("%s", individu.nom);
        if (individu.nom[0] != '#')
           {
           printf("Telephone : ");
           scanf("%s", individu.tel);
           i++;
           repertoire[i] = (struct donnee *) malloc(50);
           *(repertoire[i]) = individu;
           };
        }
     while ((individu.nom[0] != '#') && (i < 500));

/* affichage */

     for (j = 0; j <= i; j++)
     printf("%s : %s\n", repertoire[j]->nom,
                         repertoire[j]->tel);
     }
```

5.1.4 Détermination de la taille d'un type

Introduction.- L'allocation de mémoire que nous avons réservée pour un `individu` dans le programme ci-dessus est de 50 octets puisque l'on sait qu'un caractère occupe un octet. Mais nous ne sommes pas toujours au fait des procédés de codage employés par tel ou tel compilateur, et donc de la taille associée à un type donné. La fonction `sizeof()` ('taille de' en anglais) permet d'obtenir la place nécessaire.

Syntaxe.- La syntaxe de cette fonction est :

```
entier = sizeof(variable);
```

où `variable` est une variable (et non un type comme on pourrait s'y attendre) et `entier` une variable entière.

Sémantique.- La variable `entier` contient alors le nombre d'octets nécessaire pour coder une entité du type celui de la variable `variable`.

Exemple d'utilisation.- On peut reprendre le programme ci-dessus en changeant juste la ligne faisant intervenir la fonction `malloc` :

```
/* sizeof.c */
- - - - - - - - - - - - -
   repertoire[i] = (struct donnee *) malloc(sizeof(individu));
- - - - - - - - - - - - -
```

5.2 Structures auto-référentes

La première façon d'utiliser les pointeurs, vue à la section précédente, nous permet de gagner un peu de place en mémoire centrale. Mais nous n'avons pas encore vraiment de structure de données dynamique (nous avons toujours un problème s'il y a plus de 500 éléments dans notre répertoire). La notion de *structure auto-référente* va nous permettre de construire de vraies structures de données dynamiques.

Voyons ceci sur un exemple, celui des *listes chaînées*.

5.2.1 Notion de liste chaînée

Si nous reprenons le problème précédent de manipulation d'un répertoire en mémoire centrale, une structure de donnée plus intéressante que celle de tableau (dont la dimension est fixée une fois pour toute) est celle de **liste chaînée**. L'idée de base est très simple.

Une telle *liste* est constituée d'*éléments*, comme pour un tableau. Un élément contient des informations utiles à l'utilisateur, comme pour un tableau, mais aussi des *informations structurelles* pour construire la liste. Plus exactement on aura un pointeur désignant l'élément suivant.

On peut résumer ceci par le schéma suivant :

Bien entendu si on fait ainsi on a une liste infinie. On a donc besoin d'une valeur spéciale de pointeur qui ne pointe sur rien (ce qui permet de terminer la liste) : c'est le **pointeur nul**. Le dessin ci-dessus se réécrit alors plutôt de la façon suivante, le pointeur nul étant traditionnellement représenté par une croix.

5.2.2 Mise en place en langage C

Problème de la déclaration récursive.- Essayons de déclarer le type d'une telle liste chaînée, par exemple en langage C. Nous pouvons penser à la déclaration suivante :

```
struct donnee {
              char nom[30] ;
              char tel[20] ;
              struct donnee *suivant ;
              } ;
```

Il y a cependant un cercle vicieux (appelé plus particulièrement ici du nom technique de *récursivité*) car on utilise le type pointeur de `donnnee` dans la déclaration de `donnee`.

Syntaxe.- On utilise cependant quand même la définition récursive du genre de celle que nous venons de voir. C'est au niveau de la conception du compilateur qu'il faudra résoudre ce problème de récursivité.

Définition.- *Une* **structure auto-référente** *est une structure dans laquelle un ou plusieurs champs ont pour type un pointeur sur cette structure.*

5.2.3 Un exemple d'utilisation

Le programme suivant crée une liste chaînée dont les éléments sont des structures comprenant un nom, un numéro de téléphone et un pointeur permettant d'accéder à l'élément suivant. Le répertoire lui-même est un pointeur, de nom `debut`, permettant d'accéder au premier élément. Pour simplifier, la liste est créée en partant du dernier élément, et non pas du premier.

On affiche, dans une seconde étape, les valeurs de la liste. Celle-ci sera affichée dans l'ordre inverse de l'ordre d'entrée.

```
/* liste_1.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
     {
     struct donnee
            {
            char nom[30];
            char tel[20];
            struct donnee *suivant;
            };
     struct donnee item, *debut, *ptr;
     char name[30], telephone[20];

/* Initialisation de la liste */

     debut = NULL;
     do
        {
        printf("\nNom       : ");
        scanf("%s",name);
        if (name[0] != '#')
           {
           printf("Telephone : ");
           scanf("%s",telephone);
           ptr = (struct donnee *) malloc(sizeof(item));
           strcpy(item.nom, name);
           strcpy(item.tel,telephone);
           item.suivant = debut;
           *ptr = item;
           debut = ptr;
           }
        }
     while (name[0] != '#');
```

```
/* Affichage de la liste */

     printf("\nLa liste dans l\'ordre inverse est :\n");
     while (debut != NULL)
           {
           printf("%s : %s\n",debut->nom, debut->tel);
           debut = debut->suivant;
           }

/* Liberation des pointeurs */

     while (debut != NULL)
           {
           ptr = debut;
           debut = debut->suivant;
           free(ptr);
           }
     }
```

EXERCICES

Comme d'habitude, lorsqu'on demande une fonction, on écrira également un programme complet permettant de tester ces sous-programmes.

Exercice 1.- *Écrire un programme C déterminant les n premiers nombres premiers, la valeur de n étant fournie en donnée. On constituera une liste chaînée des nombres premiers, au fur et à mesure de leur découverte.*

Exercice 2.- (**Recherche dans une liste chaînée**)

Écrire une fonction C qui, étant données une liste chaînée (par exemple un répertoire) et une valeur d'une clé (par exemple un nom), permet d'afficher les autres informations de l'élément (par exemple le numéro de téléphone), s'il existe.

Exercice 3.- (**Insertion dans une liste chaînée triée**)

Écrire une fonction C qui, étant donnés une liste chaînée (par exemple un répertoire) triée suivant les valeurs d'une clé (par exemple les noms) et un élément, insère ce nouvel élément dans la liste de façon à obtenir une nouvelle liste chaînée triée.

Exercice 4.- (**Tri d'une liste chaînée**)

Écrire une fonction C qui, étant donnée une liste chaînée, fournit une nouvelle liste chaînée, obtenue à partir de la précédente, en la triant suivant une clé (par exemple le nom pour un répertoire).

[On pourra s'aider de la fonction de l'exercice précédent.]

Exercice 5.- (**Fichier et liste chaînée**)

Une liste chaînée est très facile à manipuler mais ne concerne que la mémoire volatile alors qu'un fichier est moins facile à manipuler mais concerne la mémoire de masse. Il est donc important de pouvoir passer d'une structure de donnée à l'autre.

- 1^o) *Écrire une fonction C qui lit un fichier (par exemple un répertoire) et crée une liste chaînée possédant les mêmes éléments.*
- 2^o) *Écrire une fonction C qui lit une liste chaînée (par exemple un répertoire) et place ses éléments dans un fichier.*

Exercice 6.- (**Tri d'un fichier**)

Utiliser les fonctions des exercices 4 et 5, ou mieux s'en inspirer, pour trier un fichier.

Exercice 7.- *Écrire une fonction C qui, étant donnés une liste chaînée d'entiers et deux entiers P et G, affiche à l'écran les entiers de la liste qui sont plus grands que P et plus petits que G.*

Exercice 8.- *Écrire une fonction C qui considère une liste chaînée et renverse l'ordre des éléments de cette liste.*