Rappels sur les pointeurs et tableaux dans le langage C

L'utilisation des pointeurs dans le langage C est souvent orientée vers la manipulation de tableaux. Dans ce qui suit, un rappel concis sur ces deux concepts est présenté.

1 Pointeurs et tableaux à une dimension

Tout tableau en C est en fait un pointeur constant. Dans la déclaration

```
int tab[10];
```

`tab` est un pointeur constant (non modifiable) dont la valeur est l'adresse du premier élément du tableau. Autrement dit, `tab` a pour valeur `&tab[0]`. On peut donc utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau.

```
#define N 5
int tab[5] = {0, 1, 4, 20, 7};
main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    {
      printf(" %d \n",*p);
      p++;
    }
}
```

On accède à l'élément d'indice `i` du tableau `tab` grâce à l'opérateur d'indexation `[]`, par l'expression `tab[i]`. Cet opérateur d'indexation peut en fait s'appliquer à tout objet `p` de type pointeur. Il est lié à l'opérateur d'indirection `*` par la formule

```
p[i] = *(p + i)
```

Les pointeurs et tableaux se manipulent donc exactement de la même manière. Par exemple, le programme précédent peut aussi s'écrire

```
#define N 5
int tab[5] = {0, 1, 4, 20, 7};

main()
{
  int i;
  int *p;
  p = tab;
  for (i = 0; i < N; i++)
    printf(" %d \n", p[i]);
}
```

Cependant, la manipulation de tableaux, et non de pointeurs, possède certains inconvénients dus au fait qu'un tableau est un pointeur constant. Ainsi

- on ne peut pas créer de tableaux dont la taille est une variable du programme,
- on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement. Ainsi, pour créer un tableau d'entiers à `n` éléments où `n` est une variable du programme, on écrit :

```
#include <stdlib.h>
main()
{
  int n;
  int *tab;

   ...
  tab = (int*)malloc(n * sizeof(int));
   ...
  free(tab);
}
```

Si on veut en plus que tous les éléments du tableau `tab` soient initialisés à zéro, on remplace l'allocation dynamique avec `malloc` par

```
  tab = (int*)calloc(n, sizeof(int));
```

Les éléments de `tab` sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.

Les deux différences principales entre un tableau et un pointeur sont

- un pointeur doit toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i` ;
- un tableau n'est pas une Lvalue ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++;`).

2 Pointeurs et tableaux à plusieurs dimensions

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur. Considérons le tableau à deux dimensions défini par :
`int tab[M][N];`

`tab` est un pointeur, qui pointe vers un objet lui-même de type pointeur d'entier. `tab` a une valeur constante égale à l'adresse du premier élément du tableau, `&tab[0][0]`. De même `tab[i]`, pour `i` entre 0 et `M-1`, est un pointeur constant vers un objet de type entier, qui est le premier élément de la ligne d'indice `i`. l'élément `tab[i]` a donc une valeur constante qui est égale à `&tab[i][0]`.

Exactement comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de

nombreux avantages sur les tableaux multi-dimensionnés. On déclare un pointeur qui pointe sur un objet de type *type ** (deux dimensions) de la même manière qu'un pointeur, c'est-à-dire

```
type **nom-du-pointeur;
```

De même, un pointeur qui pointe sur un objet de type *type* `**` (équivalent à un tableau à 3 dimensions) se déclare par

```
type ***nom-du-pointeur;
```

Par exemple, pour créer avec un pointeur de pointeur une matrice à `k` lignes et `n` colonnes à coefficients entiers, on écrit :

```
main()
{
  int k, n;
  int **tab;

  tab = (int**)malloc(k * sizeof(int*));
  for (i = 0; i < k; i++)
    tab[i] = (int*)malloc(n * sizeof(int));
      ....

  for (i = 0; i < k; i++)
    free(tab[i]);
  free(tab);
}
```

La première allocation dynamique réserve pour l'objet pointé par `tab` l'espace mémoire correspondant à `k` pointeurs sur des entiers. Ces `k` pointeurs correspondent aux lignes de la matrice. Les allocations dynamiques suivantes réservent pour chaque pointeur `tab[i]` l'espace mémoire nécessaire pour stocker `n` entiers.

Si on désire en plus que tous les éléments du tableau soient initialisés à zéro, il suffit de remplacer l'allocation dynamique dans la boucle `for` par

```
    tab[i] = (int*)calloc(n, sizeof(int));
```

Contrairement aux tableaux à deux dimensions, on peut choisir des tailles différentes pour chacune des lignes `tab[i]`. Par exemple, si l'on veut que `tab[i]` contienne exactement `i+1` éléments, on écrit

```
   for (i = 0; i < k; i++)
    tab[i] = (int*)malloc((i + 1) * sizeof(int));
```

3 Pointeurs et chaînes de caractères

On a vu précédemment qu'une chaîne de caractères était un tableau à une dimension d'objets de type `char`, se terminant par le caractère nul `'\0'`. On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`. On peut faire subir à une chaîne définie par `char *chaine;`
des affectations comme `chaine = "ceci est une chaine";`
et toute opération valide sur les pointeurs, comme l'instruction `chaine++;`.

Ainsi, le programme suivant imprime le nombre de caractères d'une chaîne (sans compter le caractère nul).

```
#include <stdio.h>
main()
{
  int i;
  char *chaine;

  chaine = "chaine de caracteres";
  for (i = 0; *chaine != '\0'; i++)
    chaine++;
  printf("nombre de caracteres = %d\n",i);
}
```

La fonction donnant la longueur d'une chaîne de caractères, définie dans la librairie standard `string.h`, procède de manière identique. Il s'agit de la fonction `strlen` dont la syntaxe est `strlen(`*`chaine`*`);` où *`chaine`* est un pointeur sur un objet de type `char`. Cette fonction renvoie un entier dont la valeur est égale à la longueur de la chaîne passée en argument (moins le caractère `'\0'`). L'utilisation de pointeurs de caractère et non de tableaux permet par exemple de créer une chaîne correspondant à la concaténation de deux chaînes de caractères :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  int i;
  char *chaine1, *chaine2, *res, *p;

  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
  p = res;
  for (i = 0; i < strlen(chaine1); i++)
    *p++ = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++)
    *p++ = chaine2[i];
  printf("%s\n",res);
}
```

Remarquons l'indispensable utilisation d'un pointeur intermédiaire `p` dès que l'on effectue des opérations de type incrémentation. En effet, si on avait incrémenté directement la valeur de `res`, on aurait évidemment ``perdu'' la référence sur le premier caractère de la chaîne. Par exemple,

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  int i;
  char *chaine1, *chaine2, *res;
  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  res = (char*)malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
```

```
  for (i = 0; i < strlen(chaine1); i++)
    *res++ = chaine1[i];
  for (i = 0; i < strlen(chaine2); i++)
    *res++ = chaine2[i];
  printf("\n nombre de caracteres de res = %d\n",strlen(res));
}
```

imprime la valeur `0`, puisque `res` a été modifié au cours du programme et pointe maintenant sur le caractère nul.

6 Pointeurs et structures

6.1 Pointeur sur une structure

Contrairement aux tableaux, les objets de type structure en C sont des Lvalues. Ils possèdent une adresse, correspondant à l'adresse du premier élément du premier membre de la structure. On peut donc manipuler des pointeurs sur des structures. Ainsi, le programme suivant crée, à l'aide d'un pointeur, un tableau d'objets de type structure.

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

struct eleve{
  char nom[20];
  int date;
};

typedef struct eleve *classe;

main()
{
  int n, i;
  classe tab;

  printf("nombre d'eleves de la classe = ");
  scanf("%d",&n);
  tab = (classe)malloc(n * sizeof(struct eleve));
  for (i =0 ; i < n; i++)
    {
      printf("\n saisie de l'eleve numero %d\n",i);
      printf("nom de l'eleve = ");
      scanf("%s",&tab[i].nom);
      printf("\n date de naissance JJMMAA = ");
      scanf("%d",&tab[i].date);
    }
  printf("\n Entrez un numero  ");
  scanf("%d",&i);
  printf("\n Eleve numero %d:",i);
  printf("\n nom = %s",tab[i].nom);
  printf("\n date de naissance = %d\n",tab[i].date);
  free(tab);
}
```

Si `p` est un pointeur sur une structure, on peut accéder à un membre de la structure pointé par l'expression `(*p).`*`membre`*

L'utilisation de parenthèses est ici indispensable car l'opérateur d'indirection `*` à une priorité plus élevée que l'opérateur de membre de structure. Cette notation peut être simplifiée grâce à l'opérateur *pointeur de membre de structure*, noté `->`.
L'expression précédente est strictement équivalente à `p->membre` Ainsi, dans le programme précédent, on peut remplacer `tab[i].nom` et `tab[i].date` respectivement par `(tab + i)->nom` et `(tab + i)->date`.

6.2 Structures auto-référencées

On a souvent besoin en C de modèles de structure dont un des membres est un pointeur vers une structure de même modèle. Cette représentation permet en particulier de construire des listes chaînées. En effet, il est possible de représenter une liste d'éléments de même type par un tableau (ou un pointeur). Toutefois, cette représentation, dite *contiguë*, impose que la taille maximale de la liste soit connue a priori (on a besoin du nombre d'éléments du tableau lors de l'allocation dynamique). Pour résoudre ce problème, on utilise une représentation *chaînée* : l'élément de base de la chaîne est une structure appelée *cellule* qui contient la valeur d'un élément de la liste et un pointeur sur l'élément suivant. Le dernier élément pointe sur la liste vide `NULL`. La liste est alors définie comme un pointeur sur le premier élément de la chaîne.

Pour représenter une liste d'entiers sous forme chaînée, on crée le modèle de structure `cellule` qui a deux champs : un champ `valeur` de type `int`, et un champ `suivant` de type pointeur sur une `struct cellule`. Une liste sera alors un objet de type pointeur sur une `struct cellule`. Grâce au mot-clef `typedef`, on peut définir le type `liste`, synonyme du type pointeur sur une `struct cellule`.

```
struct cellule
{
  int valeur;
  struct cellule *suivant;
};

typedef struct cellule *liste;
```

Un des avantages de la représentation chaînée est qu'il est très facile d'insérer un élément à un endroit quelconque de la liste. Ainsi, pour insérer un élément en tête de liste, on utilise la fonction suivante :

```
liste insere(int element, liste Q)
{
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = Q;
  return(L);
}
```

Le programme suivant crée une liste d'entiers et l'imprime à l'écran :

```
#include <stdlib.h>
#include <stdio.h>
struct cellule
{
  int valeur;
  struct cellule *suivant;
```

```
};
typedef struct cellule *liste;


liste insere(int element, liste Q)
{
  liste L;
  L = (liste)malloc(sizeof(struct cellule));
  L->valeur = element;
  L->suivant = Q;
  return(L);
}

main()
{
  liste L, P;
  L = insere(1,insere(2,insere(3,insere(4,NULL))));
  printf("\n impression de la liste:\n");
  P = L;
  while (P != NULL)
    {
      printf("%d \t",P->valeur);
      P = P->suivant;
    }
}
```

On utilisera également une structure auto-référencée pour créer un arbre binaire :

```
struct noeud
{
  int valeur;
  struct noeud *fils_gauche;
  struct noeud *fils_droit;
};

typedef struct noeud *arbre;
```