TP 02 - Les pointeurs en langage C -

1 Définition

Un pointeur est une variable qui contient l'adresse d'une autre variable. Il possède le même type que la variable. Il est déclaré précédé d'une étoile.

2 Déclaration

Syntaxe

```
<type> *<nom_variabel>
```

Exemple

```
float *x;
int *a, *b;
char *t;
```

x est un pointeur qui va contenir l'adresse d'un réel, a et b sont des pointeurs qui vont contenir chacun l'adresse d'un entier, et t est un pointeur qui va contenir l'adresse d'un caractère ou d'une chaîne de caractères.

3 Les opérateurs de base

Lorsqu'on travaille avec des pointeurs, nous avons besoins:

- D'un opérateur *adresse de*: **&** (pour obtenir l'adresse d'une variable);
- D'un opérateur *contenu de*: ***** (pour accéder au contenu d'une adresse).

4 L'opérateur *adresse de*: &

Syntaxe

```
&<nom_variable>
```

Fournit l'adresse de la variable.

Exemple L'opérateur & nous est déjà famillier par la fonction **scanf**, qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

```
int n;
printf("Introduire n: ");
scanf("%d", &n);
```

Remarque L'opérateur & peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c'est à dire à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

5 L'opérateur *contenu de*: *

Syntaxe

`*<nom_pointeur>`

Désigne le contenu de l'adresse référencée par le pointeur.

Exemple

```
p = &a;
b = *p;
*p = 20;
```

- p pointe sur a;
- Le contenu de a (référencé par $*p$) est affecté à b;
- Le contenu de a (référencé par $*p$) est mis à 20.

6 Exemple

```
#include<stdio.h>

void main() {
  int a = 5, *b;

  b = &a;

  printf("a = %d et *b = %d", a, *b);
}
```

À l'exécution, le programme affiche ce qui suit:

```
a = 5 et *b = 5
```

$*b = 5$, car le pointeur b contient l'adresse de a, et à cette adresse se trouve la valeur de a (égale à 5).

7 Passage de paramètres à une fonction

En langage C, le passage des paramètres dans une fonction se fait toujours par valeur, c'est à dire, les fonctions n'obtiennent que les *valeurs* de leurs paramètres et non pas d'accès aux variables elles-mêmes. Pour changer la valeur d'une variable de la fonction appelante, nous allons procéder comme suit:

- La fonction appelante doit fournir l'adresse de la variable;
- La fonction appelée doit décalrer le paramètre comme pointeur.

Exemple Programme C qui contient une fonction *permuter* qui reçoit deux entiers par adresse, et permute le contenu des deux variables.

```
#include<stdio.h>

void permuter(int*, int*);

void main() {
  int a = 5, b = 7;

  permuter(&a, &b);

  printf("a = %d, b = %d", a, b);
}

void permuter(int* a, int* b) {
  int tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Après exécution, le programme affiche les résultats suivants:

```
a = 7, b = 5
```

8 Pointeurs et tableaux

Le nom d'un tableau contient en réalité l'adresse de son premier élément.

Exemple

```
int a[10];
```

- Le nom du tableau contient l'adresse de son premier élément, c'est à dire, $a = \&a[0]$;
- $a + i$ pointe sur la i^{ieme} composante de a;
- $*a$ désigne le contenu de $a[0]$;
- $*(a + 1)$ désigne le contenu de $a[1]$;
- ...
- $*(a + i)$ désigne le contenu de la i^{ieme} composante de a ($a[i]$).

8.1 Différence entre un pointeur et un nom de tableau

Il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau:

- Un pointeur est une variable, donc les opérations comme $a = p$ ou $a{+}{+}$ sont permises;
- Le nom d'un tableau est une constante, donc des opérations comme $a = p$ ou $a{+}{+}$ ne sont pas permises.

8.2 Passage d'un tableau à une fonction

Le passage d'un tableau de données comme paramètre à une fonction est toujours un passage par adresse, car le nom d'un tableau contient en réalité son adresse.

9 Allocation dynamique

9.1 Problème

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

Un des principaux intérêts de l'allocation dynamique est de permettre à un programme de réserver la place nécessaire au stockage d'un tableau en mémoire dont il ne connaissait pas la taille avant la compilation. En effet, jusqu'ici, la taille de nos tableaux était fixée dans le code source.

9.2 La fonction *malloc* et l'opérateur *sizeof*

malloc La fonction *malloc* nous permet de réserver de la mémoire à un programme en cours d'exécution. Elle nous donne accès au *tas* (*heap*).

Syntaxe

```
#include<stdlib.h>

void* malloc(N);
```

Retourne l'adresse d'un bloc mémoire de N octets libres ou la valeur 0 s'il n'y a pas assez de mémoire.

sizeof Si nous voulons réserver de la mémoire pour des données d'un type dont la grandeur varie d'une machine à l'autre, nous avons besoin de la grandeur effective d'une donnée de ce type. L'opérateur *sizeof* nous aide alors à préserver la portabilité du programme.

Exemple

```
int taille_int;
taille_int = sizeof(int);
printf("Un int sur cette machine est codé sur %d octets", taille_int);
```

Exemple (Allocation dynamique d'un tableau d'entiers)

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

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

```
  printf("Introduire la taille du tableau: ");
  scanf("%d", &n);

  tab = (int*) malloc(n * sizeof(int));
  if(tab == NULL) exit(EXIT_FAILURE);

  printf("Introduire les éléments du tableau\n");
  for(i = 0; i < n; i++) {
    printf("Introduire tab[%d]: ", i);
    scanf("%d", &tab[i]);
  }

  printf("Affichage du tableau\n");
  for(i = 0; i < n; i++) printf("%d ", tab[i]);

  putchar('\n');

  return EXIT_SUCCESS;
}
```