Chapitre 5

Les pointeurs

Chapitre 5 : Les pointeurs

I Introduction

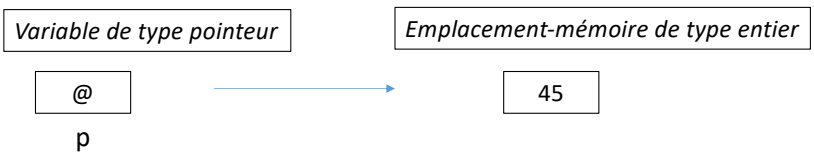
- A toute variable utilisée dans un programme est réservé un emplacement-mémoire, *ensemble d'octets*, identifié par une ***adresse*** sans **équivoque** (deux variables différentes ont des adresses mémoire différentes).
- Il est plus simple de faire référence à un emplacement-mémoire par un identifiant (*nom de variable*) que par son **adresse** (*exprimée en hexadécimal*).

Chapitre 5 : Les pointeurs

- **L'adresse** d'une variable est un numéro d'octet.
- Ce qui implique que quel que soit le type de la variable, son **adresse** est un **entier**.
- Selon la plate-forme utilisée, elle est codée sur 2, 4 ou 8 octets (*en général pour un type entier long : 64 octets*).
- Dans le cas d'une variable occupant plusieurs octets contigus, son adresse est **l'adresse du premier octet**.
- Une *adresse-mémoire* est gérée à l'aide d'une variable spécifique.
- Cette variable sera de type ***pointeur*** et aura pour valeur (ou contenu) une **adresse.**
- **L'adresse** d'une variable est un numéro d'octet.
- Ce qui implique que quel que soit le type de la variable, son **adresse** est un **entier**.
- Selon la plate-forme utilisée, elle est codée sur 2, 4 ou 8 octets (*en général pour un type entier long : 64 octets*).
- Dans le cas d'une variable occupant plusieurs octets contigus, son adresse est **l'adresse du premier octet**.
- Une *adresse-mémoire* est gérée à l'aide d'une variable spécifique.
- Cette variable sera de type ***pointeur*** et aura pour valeur (ou contenu) une **adresse.**

Chapitre 5 : Les pointeurs

- Une variable de type **pointeur** permet de **référencer** une variable. En pratique, cela signifie qu'une variable de type **pointeur** contient **l'adresse** d'une variable. L'accès au contenu de la variable passe par la variable pointeur.
- Une valeur de type **pointeur** correspond à une *adresse-mémoire*.
- Exemple : Soit p une variable qui **référence** un entier.



Chapitre 5 : Les pointeurs

II Déclaration de variables de type pointeur

Syntaxe **:** type_objet_référencé. ∗nom_variable_pointeur**;**

∗ : opérateur **unaire d'indirection**. Il permet d'accéder directement au contenu de l'emplacement-mémoire pointé (*référencé* ou *adressé*) par la variable de type **pointeur**.

Exemple1 **: int** ∗ p**;**

Interprétation : p est une variable de type **pointeur** sur un entier : c.à.d contient *l'adresse d'un emplacement-mémoire destiné à contenir un* **entier**.

Chapitre 5 : Les pointeurs

Exemple2 :

```
int main ()
{
 int i = 2 ;
 int  * p ;
 p = &i ; /* contenu de p : adresse de la variable i */
 printf ( " Contenu adresse référencée par p :
          %d\n ", * p ) ;
 printf ( " Contenu de i :%d\n " , i ) ;
}
```

- ✓ ∗ p : contenu de l'adresse pointée par p.
- ✓ i et ∗p sont deux objets identiques : même adresse et même valeur.
- ✓ &i et p sont deux objets identiques.
- ✓ Toute modification de ∗p modifiera la variable i.
- ✓ Les deux instructions afficheront : **2**.

Chapitre 5 : Les pointeurs

III Pointeurs et opérateurs additifs

- On peut utiliser les opérations d'addition (**+**) ou de soustraction (**-**) d'un entier sur des variables de type **pointeur**.
- Ces deux opérations n'ont de sens que si le **pointeur** référence (*pointe*) un *élément d'un tableau*.

Soient **p** un **pointeur** vers des objets de type **T** et un *tableau* dont les éléments sont de type **T**.

- Si **p** *pointe* le **$i^{ième}$** élément du tableau alors **p+j** est une ***valeur de type pointeur vers T*** et référence le **$(i+j)^{ième}$** élément du *tableau* (si cet élément existe).
- Il en est de même pour la différence.

Remarque :

- Il est possible de faire la soustraction de deux variables pointeurs à condition que les deux *pointent* des objets de même type.
- De plus, l'opération n'a de sens que s'ils *pointent* des éléments d'un même tableau par exemple.

Chapitre 5 : Les pointeurs

IV Pointeurs et allocation dynamique

- Pour manipuler un **pointeur** **p** il faut l'initialiser pour qu'il *référence* un emplacement-mémoire.
- A défaut, il a pour valeur la constante (symbolique) **NULL** qui est définie dans le fichier d'entêtes **stdio.h** et qui a généralement pour valeur 0.
- Pour savoir si un **pointeur** **p** *référence* un objet, il suffit de le comparer à **NULL**.
- L'initialisation de **p** se fait :
  - soit par *affectation* : on lui affecte l'adresse d'une autre variable
  - soit par *réservation* d'un emplacement-mémoire pouvant contenir ***p**

  (la valeur *référencée* par **p**).
- Ce principe qui consiste à réserver de la place mémoire pour stocker un objet pointé s'appelle ***allocation dynamique***.
- La place réservée doit être conforme au type de l'objet.

Chapitre 5 : Les pointeurs

- Les fonctions ***malloc*** et ***free*** de la librairie standard **stdlib** permettent respectivement **d'allouer** de la mémoire à un pointeur ou de la **dés-allouer**.
- La fonction ***sizeof***, de la librairie C, permet de calculer la ***taille*** de l'emplacement occupé par le ***type*** de l'objet *référencé*.
- Cela évite à l'utilisateur de connaître, pour chaque machine, le nombre d'octets utilisés pour chaque *type*.

Syntaxe de l'allocation **: malloc** (*nbre_octets_objet_pointé*)

nbre_octets_objet_pointé : calculé par la fonction ***sizeof***.

Sémantique :

- le résultat de cette fonction est un pointeur de type ∗*char*.
- Pour initialiser des **pointeurs** vers des objets qui ne sont pas de type *char*, on convertit explicitement le résultat de la fonction ***malloc*** à l'aide d'un ***cast*.**

Chapitre 5 : Les pointeurs

Exemple **:**

```
#include <stdlib.h>
int main ( )
{
 int *p;                                  /* déclaration de la variable pointeur */
 p = (int *) malloc (sizeof ( int ) ) ;   /* allocation mémoire */
 *p = 5 ;
...
 }
```

Un entier de type **int** étant codé sur 4 octets, on peut également écrire :

```
 p = ( int *) malloc ( 4 ) ;
```

Chapitre 5 : Les pointeurs

✓ La fonction **malloc** permet également d'allouer la place mémoire pour plusieurs objets.

Exemple :

```
#include <stdlib.h>
int main ( )
{
int i = 5 , j = 10 ;
/* déclaration de la variable pointeur*/
 int * p ;
/* allocation mémoire */
 p = (int *) malloc (2 * sizeof ( int ) ) ;
 * p = i ;
* (p + 1) = j ;
...
 }
```

✓ *p contient 5 et *(p+1) contient 10.

✓ La différence entre les deux adresses p et p+1 est égale à 4 qui correspond à la taille de l'emplacement mémoire occupé par le second entier.

Dés-allocation :

free (nom_pointeur) => *on libère l'emplacement référencé par le pointeur s'il n'y en a plus besoin.*

Chapitre 5 : Les pointeurs

Autre fonction d'allocation :

La fonction ***calloc*** de la librairie **stdlib** joue le même rôle que la fonction ***malloc*** et permet, en plus, d'initialiser l'objet pointé * p à 0
(*tous les bits de l'emplacement mémoire réservé sont à 0*)

Syntaxe : **calloc (*nbre_objets*, *taille_objets*)**

Exemple :

```
#include <stdlib.h>
int main ( )
{
int * p , n ;
.....
 p = (int *) calloc (n, sizeof ( int ) ) ;
.....
 }
```

est équivalent à :

```
#include <stdlib.h>
int main ( )
{
int * p , n , i ;
.....
 p = (int *) malloc (n * sizeof ( int ) ) ;
for ( i = 0 ; i < n ; i++ )
* (p + i) = 0 ;
.....
 }
```

Chapitre 5 : Les pointeurs

Une application des pointeurs :

En **C**, le passage de paramètres à une fonction est toujours fait par **valeur**.

Pour simuler le passage de paramètres par **adresse** ou **référence** (équivalents des modes *sortie* et *entrée-sortie* vus en algo), on utilise les **pointeurs**.

Exemple1 :

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

```
void Permut ( int * a, int * b )
{
int mem = * a ;
 * a = * b ;
 * b = mem ;
}
int main ()
{
 int x = 3, y = 6 ;
 Permut ( &x, &y ) ;
 printf ( "x = %d, y = %d\n", x, y) ;
}
```

Résultats affichés : **x = 6, y = 3**

Chapitre 5 : Les pointeurs

Exemple2 :

```
#include <stdio.h>
#include <stdlib.h>
void Maximum ( int a, int b, int *m )
{
 if ( a > b ) * m = a ;
  else * m = b ;
}
 int main ()
{
 int x = 3, y = 8, max ;
 Maximum ( x, y, &max) ;
 printf ( " Max( %d, %d ) = %d \n", x, y, max) ;        /* résultat affiché : Max(3,8) = 8 */
}
```