7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.1. Introduction

- Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle ***adresse.***
- Pour retrouver une variable, il suffit donc de connaître l'adresse de l'octet où elle est stockée.
- Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse.
- C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire. Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.2. Adresse et valeur d'un objet

- On appelle *Lvalue* (left value) tout objet pouvant être placé à gauche d'un opérateur d'affectation. Une *Lvalue* est caractérisée par :
  - son adresse: l'adresse-mémoire à partir de laquelle l'objet est stocké;
  - sa valeur, c'est-à-dire ce qui est stocké à cette adresse.
- Dans l'exemple, int i, j; i = 3; j = i; Si le compilateur a placé la variable i à l'adresse 4831836000 en mémoire, et la variable j à l'adresse 4831836004, on a: Deux variables différentes ont des adresses différentes.

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.2. Adresse et valeur d'un objet

- L'affectation i = j; n'opère que sur les valeurs des variables. Les variables i et j étant de type int, elles sont stockées sur 4 octets. Ainsi la valeur de i est stockée sur les octets d'adresse 4831836000 à 4831836003.
- L'opérateur & permet d'accéder à l'adresse d'une variable.
- Toutefois &i n'est pas une Lvalue mais une constante ➔ on ne peut pas faire figurer &i à gauche d'un opérateur d'affectation.
- Pour pouvoir manipuler des adresses, on doit donc recourir un nouveau type d'objets, les pointeurs.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Notion de pointeur

- Un pointeur est un objet dont la valeur est égale à l'adresse d'un autre objet. On déclare un pointeur par l'instruction :

type **nom-du-pointeur*;

où ***type*** est le type de l'objet pointé.

- Cette déclaration déclare un identificateur, *nom-du-pointeur*, associé à un objet dont la valeur est l'adresse d'un autre objet de type ***type.***
- L'identificateur *nom-du-pointeur* est donc en quelque sorte un identificateur d'adresse. Comme pour n'importe quelle Lvalue, sa valeur est modifiable.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Notion de pointeur

- Dans l'exemple suivant, on définit un pointeur p qui pointe vers un entier i :

```
int i = 3;
int *p;
p = &i;
```

- On se trouve dans la configuration

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Notion de pointeur

- **L'opérateur unaire d'indirection *** permet d'accéder directement à la valeur de l'objet pointé. Ainsi, si p est un pointeur vers un entier i, *p désigne la valeur de i. Par exemple, le programme

main() { int i = 3; int *p; p = &i; printf("*p = %d \n",*p);} ➔*p = 3.

- Les objets i et *p sont identiques : ils ont mêmes adresse et valeur:

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

Cela signifie en particulier que toute modification de *p modifie i. Ainsi, si l'on ajoute l'instruction *p = 0; à la fin du programme précédent, la valeur de i devient nulle.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Notion de pointeur

On peut donc dans un programme manipuler à la fois les objets p et *p. Ces deux manipulations sont très différentes. Comparons par exemple les deux programmes suivants : P_1➔

```
main() { int i = 3, j = 6; int *p1, *p2; p1 = &i; p2 = &j; *p1 = *p2; }
```

Avant la dernière affectation

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

Après *p1 = *p2

objet	adresse	valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Notion de pointeur

P_2➔ main() { int i = 3, j = 6; int *p1, *p2; p1 = &i; p2 = &j; p1 = p2; }
Par contre, l'affectation p1 = p2 du second programme, conduit à la situation :

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Arithmétique des pointeurs

- La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques.

- Les seules opérations arithmétiques valides sur les pointeurs sont :

- L'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- La soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ; La différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

➔ Notons que la somme de deux pointeurs n'est pas autorisée.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Arithmétique des pointeurs

- Si i est un entier et p est un pointeur sur un objet de type *type*, l'expression p + i désigne un pointeur sur un objet de type *type* dont la valeur est égale à la valeur de p incrémentée de i * sizeof(*type*).

- Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation ++ et --.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Arithmétique des pointeurs

Exemple:

```
main() { int i = 3; int *p1, *p2; p1 = &i; p2 = p1 + 1;
 printf("p1 = %ld \t p2 = %ld\n",p1,p2); }
```

➔ affiche p1 = 4831835984 p2 = 4831835988.

Par contre, le même programme avec des pointeurs sur des objets de type double :

➔ affiche p1 = 4831835984 p2 = 4831835992.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.3. Arithmétique des pointeurs

- Les opérateurs de comparaison sont également applicables aux pointeurs, à condition de comparer des pointeurs qui pointent vers des objets de même type.

Exemple:

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
 main() { int *p;
printf("\n ordre croissant:\n");
for (p = &tab[0]; p <= &tab[N-1]; p++) printf(" %d \n",*p);
printf("\n ordre decroissant:\n");
for (p = &tab[N-1]; p >= &tab[0]; p--) printf(" %d \n",*p); }
```

- Si p et q sont deux pointeurs sur des objets de type *type*, l'expression p - q désigne un entier dont la valeur est égale à (p - q)/sizeof(*type*) .

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.4. Allocation dynamique

Avant toute utilisation, un pointeur doit être initialisé(sinon, il peut pointer sur n'importe quelle région de la mémoire!):

- Soit par l'affectation d'une valeur « nulle » à un pointeur : p=NULL;
- Soit par l'affectation de l'adresse d'une autre variable: p=&i;
- Soit par l'allocation dynamique d'un nouvel espace-mémoire.

Définition: L'allocation dynamique est l'opération qui consiste à réserver un espace-mémoire d'une taille définie.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.4. Allocation dynamique

L'allocation dynamique en C se fait par l'intermédiaire de la fonction dans la libraire standard stdlib.h:

Char * malloc(nombreOctets)

- Par défaut, cette fonction retourne un char * pointant vers une espace mémoire de taille **nombreOctets** octets.
- Pour initialiser des pointeurs vers des objets qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un cast.
- L'argument **nombreOctets** est souvent donné à l'aide de la fonction sizeof() qui renvoie le nombre d'octets utilisés pour stocker un objet.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.4. Allocation dynamique

Pour initialiser un pointeur vers un entier, on écrit :

```
#include <stdlib.h>
int *p;
p = (int*)malloc(sizeof(int));
```

On aurait pu écrire également

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

Puisqu'un objet de type int est stocké sur 4 octets. Mais on préfèrera la première écriture qui a l'avantage d'être portable.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.4. Allocation dynamique

- Le programme suivant:

```
#include <stdio.h>
#include <stdlib.h>
main(){int i = 3; int *p;
 printf("valeur de p avant initialisation = %ld\n",p);
p = (int*)malloc(sizeof(int));
printf("valeur de p apres initialisation = %ld\n",p);
*p = i; printf("valeur de *p = %d\n",*p);}
```

- Ce programme définit un pointeur p sur un objet *p de type int, et affecte à *p la valeur de la variable i. Il imprime à l'écran (avertissement):
  - valeur de p avant initialisation = 0
  - valeur de p apres initialisation = 5368711424
  - valeur de *p = 3

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.4. Allocation dynamique

La fonction malloc permet également d'allouer un espace pour plusieurs objets adjacents en mémoire. On peut écrire par exemple

```
#include <stdio.h>
#include <stdlib.h>
main()
{ int i = 3; int j = 6; int *p; p = (int*)malloc(2 * sizeof(int));
*p = i; *(p + 1) = j;
printf("p = %ld \t *p = %d \t p+1 = %ld \t *(p+1) = %d \n",p,*p,p+1,*(p+1));}
```

On a ainsi réservé, à l'adresse donnée par la valeur de p, 8 octets en mémoire, qui permettent de stocker 2 objets de type int. Le programme affiche : p = 5368711424 *p = 3   p+1 = 5368711428   *(p+1) = 6 .

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.4. Allocation dynamique

La fonction calloc de la librairie stdlib.h a le même rôle que la fonction malloc mais elle initialise en plus l'objet pointé *p à zéro. Sa syntaxe est

calloc(*nb-objets*,*taille-objets*)

Ainsi, si p est de type int*, l'instruction

```
p = (int*)calloc(N,sizeof(int));
```

est strictement équivalente à

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
*(p + i) = 0;
```

➔ L'emploi de calloc est simplement plus rapide.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.4. Allocation dynamique

- Lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement (c'est-à-dire quand on n'utilise plus le pointeur p), il faut libérer cette place en mémoire.

- Ceci se fait à l'aide de l'instruction free qui a pour syntaxe:

free(*nom-du-pointeur*);

- A toute instruction de type malloc ou calloc doit être associée une instruction de type free.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

L'usage des pointeurs en C est, en grande partie, orienté vers la manipulation des tableaux.

A. 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.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

A. Pointeurs et tableaux à une dimension

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
main()
{int i;
int *p;
p = tab;
for (i = 0; i < N; i++)
{printf(" %d \n",*p);
p++;}}
```

pour accéder un élément i du Tab : tab[i] ➔

p[i] = *(p + i)

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

A. Pointeurs et tableaux à une dimension

Pointeurs et tableaux se manipulent donc exactement de même manière.

Par exemple, le programme précédent peut aussi s'écrire

```
#define N 5

int tab[5] = {1, 2, 6, 0, 7};

 main() { int i; int *p; p = tab;

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

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

A. Pointeurs et tableaux à une dimension

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.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

A. Pointeurs et tableaux à une dimension

- 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.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

B. 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. tab[i] a donc une valeur constante qui est égale à &tab[i][0].

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

B. Pointeurs et tableaux à plusieurs dimensions

- 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*;

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

B. Pointeurs et tableaux à plusieurs dimensions

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);}
```

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

B. Pointeurs et tableaux à plusieurs dimensions

- 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.

7. Tableaux, Chaînes de caractères et pointeurs.

7.3. Les Pointeurs

7.3.5. Pointeurs et tableaux

B. Pointeurs et tableaux à plusieurs dimensions

- 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));
```