Pointeurs, Structures de données et Chaines

Module INFO 0401

2ème année informatique

Reims – Sciences Exactes

Cours n°3

A.HEBBACHE

Les Pointeurs

1. Adressage
    1.1. Adressage direct
    1.2. Adressage indirect

2. Les Pointeurs
    2.1. Les opérateurs de base
    2.2. Les opérations élémentaires sur pointeurs

Les Pointeurs

1. Adressage

L'importance des pointeurs en C

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de ***pointeurs***, c.-à-d. à l'aide de variables auxquelles on peut attribuer les ***adresses*** *d'autres variables*.

1. Adressage

1.1. Adressage direct

Dans la programmation, nous utilisons des variables pour stocker des informations.

La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur.

Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

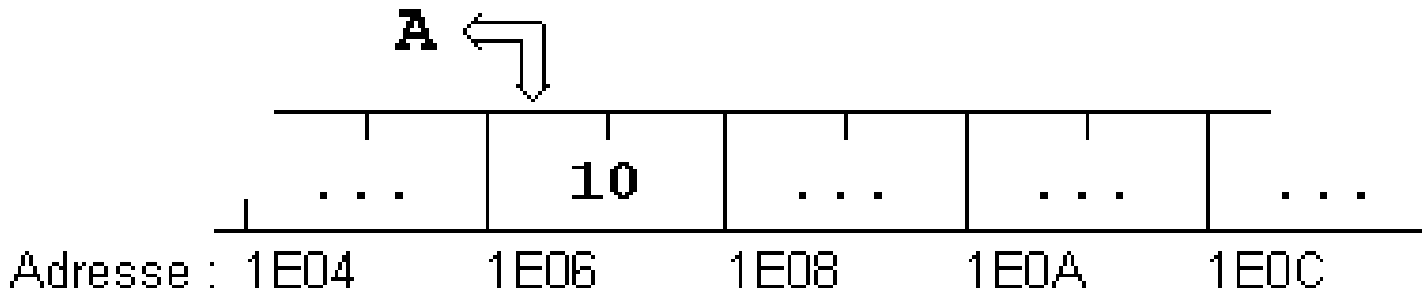
Adressage direct

Accès au contenu d'une variable par le nom de la variable.

1. Adressage

Exemple

```
short A;
A = 10;
```



1. Adressage

1.2. Adressage indirect

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, *nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée* ***pointeur*****.** Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

Adressage indirect

Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

2. Les Pointeurs

Définition

*Un **pointeur** est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.*

En C, chaque pointeur est limité à un type de données.

Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

Si un pointeur P contient l'adresse d'une variable A, on dit que :

'P pointe sur A'

2. Les Pointeurs

Remarque

Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur.

Il faut quand même bien faire la différence:

-Un ***pointeur*** est une variable qui peut 'pointer' sur différentes adresses.

- Le ***nom d'une variable*** reste toujours lié à la même adresse.

2. Les Pointeurs

2.1. Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin

- d'un opérateur 'adresse de' : **&** pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de' : * pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

2. Les Pointeurs

L'opérateur 'adresse de' : **`&`**

`&<NomVariable>` fournit l'adresse de la variable **`<NomVariable>`**
L'opérateur **`&`** nous est déjà familier par la fonction **`scanf`**, qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

Par exemple : **`scanf("%d", &N);`**

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

2. Les Pointeurs

Représentation schématique

Soit P un pointeur non initialisé

et A une variable contenant la valeur 10.

Alors l'instruction

```
P = &A;
```

affecte l'adresse de la variable A à la variable P.

2. Les Pointeurs

L'opérateur 'contenu de' : *******

****<NomPointeur>*** désigne le contenu de l'adresse référencée par le pointeur **<NomPointeur>**

Exemple

Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé :

Après les instructions,

```
P = &A;    B = *P;    *P = 20;
```

P pointe sur A,
Le contenu de A (référencé par *P) est affecté à B et mis à 20.

2. Les Pointeurs

Déclaration d'un pointeur

`<Type> *<NomPointeur>` déclare un pointeur **`<NomPointeur>`** qui peut recevoir des adresses de variables du type **`<Type>`**

Une déclaration comme `int *PNUM;`
peut être interprétée comme suit:

	PNUM est du type **int
Ou	
	*PNUM est un pointeur sur **int***
Ou	
	*PNUM peut contenir l'adresse d'une variable du type **int***

2. Les Pointeurs

Exemple

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit:

`Main()`	ou	`main()`
`{`		`{`
`/* déclarations */`		`/* déclarations */`
`Short A = 10;`		`short A, B, *P;`
`Short B = 50;`		`/* traitement */`
`Short *P;`		`A = 10;`
`/* traitement */`		`B = 50;`
`P = &A;`		`P = &A;`
`B = *P;`		`B = *P;`
`*P = 20;`		`*P = 20;`
`Return 0;`		`return 0;`
`}`		`}`

2. Les Pointeurs

Remarque

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données.

Ainsi, la variable PNUM déclarée comme pointeur sur **int** ne peut pas recevoir l'adresse d'une variable d'un autre type que **int**.

2. Les Pointeurs

2.2. Les opérations élémentaires sur pointeurs

Priorité de ***** et **&**

➢Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incrémentation ++, la décrémentation --).

Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

➢Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

2. Les Pointeurs

Exemple

Après l'instruction `P = &X;`
les expressions suivantes, sont équivalentes :

`Y = *P+1`	⇔	`Y = X+1`
`*P = *P+10`	⇔	`X = X+10`
`*P += 2`	⇔	`X += 2`
`++*P`	⇔	`++X`
`(*P)++`	⇔	`X++`

2. Les Pointeurs

Exemple

Après l'instruction `P = &X;`
les expressions suivantes, sont équivalentes :

`Y = *P+1`	⇔	`Y = X+1`
`*P = *P+10`	⇔	`X = X+10`
`*P += 2`	⇔	`X += 2`
`++*P`	⇔	`++X`
`(*P)++`	⇔	`X++`

Dans le dernier cas, les parenthèses sont nécessaires :
Comme les opérateurs unaires `*` et `++` sont évalués *de droite à gauche*, sans les parenthèses le *pointeur* P serait incrémenté, *non pas l'objet* sur lequel P pointe.

2. Les Pointeurs

Le pointeur NUL

On peut uniquement affecter des adresses à un pointeur.

Seule exception:
La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.

```
int *P;
P = 0;
```

Finalement, les pointeurs sont aussi des variables et peuvent être utilisés comme telles.

Soit P1 et P2 deux pointeurs sur **int**, alors l'affectation

```
P1 = P2;
```

copie le contenu de P2 vers P1.
P1 pointe alors sur le même objet que P2.

2. Les Pointeurs

Résumons

Après les instructions:

```
int A;   int *P;   P = &A;
```

`A`	désigne le contenu de A
`&A`	désigne l'adresse de A
`P`	désigne l'adresse de A
`*P`	désigne le contenu de A

En outre:

`&P`	Désigne l'adresse du pointeur P
`*A`	est illégal (puisque A n'est pas un pointeur)

3. Pointeurs et tableaux

En C, il existe une relation très étroite entre tableaux et pointeurs.

Ainsi, chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.

En général, les versions formulées avec des pointeurs sont plus compactes et plus efficientes, surtout à l'intérieur de fonctions.

Mais, du moins pour des débutants, le 'formalisme pointeur' est un peu inhabituel.

3. Pointeurs et tableaux

3.1. Adressage des composantes d'un tableau

Le nom d'un tableau représente l'adresse de son premier élément.
En d'autre termes:

`&tableau[0]` et `tableau`

sont une seule et même adresse.

En simplifiant, nous pouvons retenir que *le nom d'un tableau est un* ***pointeur constant*** *sur le premier élément du tableau*.

3. Pointeurs et tableaux

Exemple

Si on déclare :

```
int A[10];
int *P;
```

l'instruction :

```
P = A;
```

est équivalente à

```
P = &A[0];
```

Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante.

3. Pointeurs et tableaux

Plus généralement,

`P+i`	Pointe sur la i-ème composante derrière P et
`P-i`	Pointe sur la i-ème composante devant P.

Ainsi, après l'instruction, `P = A;` le pointeur P pointe sur A[0], et

`*(P+1)`	Désigne le contenu de A[1]
`*(P+2)`	Désigne le contenu de A[2]
`...`	...
`*(P+i)`	Désigne le contenu de A[i]

3. Pointeurs et tableaux

Remarque

Il est bien surprenant que P+i n'adresse pas le i-ème ***octet*** derrière P, mais la i-ème ***composante*** derrière P ...

Ceci s'explique par la stratégie de programmation 'défensive' du langage C :

Si on travaille avec des pointeurs, les erreurs les plus perfides sont causées par des pointeurs mal placés et des adresses mal calculées.

En C, le compilateur peut calculer automatiquement l'adresse de l'élément P+i en ajoutant à P la grandeur d'une composante multipliée par i.

Ceci est possible, parce que :

- chaque pointeur est limité à un seul type de données, et
- le compilateur connaît le nombre d'octets des différents types.

3. Pointeurs et tableaux

Exemple

Soit A un tableau contenant des éléments du type **float** et P un pointeur sur **float**:

```
float A[20], X;
float *P;
```

Après les instructions,

```
P = A;
X = *(P+9);
```

X contient la valeur du 10-ème élément de A, (c.-à-d. celle de A[9]). Une donnée du type **float** ayant besoin de 4 octets, le compilateur obtient l'adresse P+9 en ajoutant 9 * 4 = 36 octets à l'adresse dans P.

3. Pointeurs et tableaux

assemblons les constatations ci dessus :
Comme A représente l'adresse de A[0],

`*(A+1)`	Désigne le contenu de A[1]
`*(A+2)`	Désigne le contenu de A[2]
`...`	
`*(A+i)`	Désigne le contenu de A[i]

3. Pointeurs et tableaux

Attention

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

- Un *pointeur* est une variable,
donc des opérations comme P = A ou P++ sont permises.

- Le *nom d'un tableau* est une constante,
donc des opérations comme A = P ou A++ sont impossibles.

Lors de la première phase de la compilation, toutes les expressions de la forme **A[i]** sont traduites en ***(A+i).** En multipliant l'indice i par la grandeur d'une composante, on obtient un indice en octets:

```
<indice en octets> = <indice élément> * <grandeur élément>
```

3. Pointeurs et tableaux

Cet indice est ajouté à l'adresse du premier élément du tableau pour obtenir l'adresse de la composante i du tableau.

Pour le calcul d'une adresse donnée par une adresse plus un indice en octets, on utilise un mode d'adressage spécial connu sous le nom ***'adressage indexé'***:

```
<adresse indexée> = <adresse> + <indice en octets>
```

Presque tous les processeurs disposent de plusieurs registres spéciaux *(registres index)* à l'aide desquels on peut effectuer l'adressage indexé de façon très efficace.

3. Pointeurs et tableaux

Résumons

Soit un tableau A d'un type quelconque et i un indice pour les composantes de A, alors :

`A`	désigne l'adresse de	`A[0]`
`A+i`	désigne l'adresse de	`A[i]`
`*(A+i)`	désigne le contenu de	`A[i]`

Si P = A, alors

`P`	pointe sur l'élément	`A[0]`
`P+i`	pointe sur l'élément	`A[i]`
`*(P+i)`	désigne le contenu de	`A[i]`

3. Pointeurs et tableaux

Formalisme tableau et formalisme pointeur

Il nous est facile de 'traduire' un programme écrit à l'aide du *'formalisme tableau'* dans un programme employant le *'formalisme pointeur'.*

Exemple

Les deux programmes suivants copient les éléments positifs d'un tableau T dans un deuxième tableau POS.

3. Pointeurs et tableaux

Formalisme tableau

```
main()
{
  int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
  int POS[10];
  int I,J;  /* indices courants dans T et POS */

  for (J=0,I=0 ; I<10 ; I++)
      if (T[I]>0)
      {
            POS[J] = T[I];
            J++;
      }
  return 0;
}
```

3. Pointeurs et tableaux

Formalisme pointeur

Nous pouvons remplacer systématiquement la notation tableau[I] par *(tableau + I), ce qui conduit à ce programme:

```
main()
{
  int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
  int POS[10];
  int I,J;  /* indices courants dans T et POS */

  for (J=0,I=0 ; I<10 ; I++)
    if (*(T+I)>0)
    {
      *(POS+J) = *(T+I);
      J++;
    }
  return 0;
}
```

3. Pointeurs et tableaux

3.2. Arithmétique des pointeurs

➢ **Affectation par un pointeur sur le même type**

Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction

```
P1 = P2;
```

fait pointer P1 sur le même objet que P2

➢ **Addition et soustraction d'un nombre entier**

Si P pointe sur l'élément A[i] d'un tableau, alors

P+n	pointe sur **A[i+n]**
P-n	pointe sur **A[i-n]**

3. Pointeurs et tableaux

➢ **Incrémentation et décrémentation d'un pointeur**

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

`P++;`	P pointe sur **`A[i+1]`**
`P+=n;`	P pointe sur **`A[i+n]`**
`P--;`	P pointe sur **`A[i-1]`**
`P-=n;`	P pointe sur **`A[i-n]`**

3. Pointeurs et tableaux

Domaine des opérations

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies *à l'intérieur d'un tableau*.

Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

Seule exception: Il est permis de 'pointer' sur le premier octet derrière un tableau (à condition que cet octet se trouve dans le même segment de mémoire que le tableau).

Cette règle, introduite avec le standard ANSI-C, légalise la définition de boucles qui incrémentent le pointeur *avant* l'évaluation de la condition d'arrêt.

3. Pointeurs et tableaux

Exemple

`int A[10];`	
`int *P;`	
`P = A+9;`	/* dernier élément : légal */
`P = A+10;`	/* dernier élément + 1 : légal */
`P = A+11;`	/* dernier élément + 2 : illégal */
`P = A-1;`	/* premier élément - 1 : illégal */

3. Pointeurs et tableaux

➢**Soustraction de deux pointeurs**

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau* :

`P1-P2`	fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction **P1-P2** est

négatif,	si P1 précède P2
zéro,	si P1 = P2
positif,	si P2 precède P1
indéfini,	si P1 et P2 ne pointent pas dans le même tableau

3. Pointeurs et tableaux

➢ **Comparaison de deux pointeurs**

On peut comparer deux pointeurs par `<, >, <=, >=, ==, !=`

La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants.

(Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

3. Pointeurs et tableaux

3.3. Pointeurs et chaînes de caractères

De la même façon qu'un pointeur sur **int** peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur **char** peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères.

Un pointeur sur **char** peut en plus contenir *l'adresse d'une chaîne de caractères constante* et il peut même être *initialisé* avec une telle adresse.

A la fin de ce chapitre, nous allons anticiper avec un exemple et montrer que les pointeurs sont les éléments indispensables mais effectifs des fonctions en C.

3. Pointeurs et tableaux

➢ **Pointeurs sur char et chaînes de caractères constantes**

Affectation

On peut attribuer *l'adresse d'une chaîne de caractères constante* à un pointeur sur **char**.

Exemple

```
char *C;
C = "Ceci est une chaîne de caractères constante";
```

Nous pouvons lire cette chaîne constante (par exemple pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

3. Pointeurs et tableaux

Initialisation

Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:

```
char *B = "Bonjour !";
```

3. Pointeurs et tableaux

Remarque

Il existe une différence importante entre les deux déclarations:

```
char A[] = "Bonjour !";   /* un tableau  */
char *B  = "Bonjour !";   /* un pointeur */
```

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.

3. Pointeurs et tableaux

Modification

Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante.

D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur :

Exemple

```
char *A = "Petite chaîne";
char *B = "Deuxième chaîne un peu plus longue";
A = B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue.

3. Pointeurs et tableaux

Remarque
Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères :

Exemple

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];

A = B;              /* IMPOSSIBLE -> ERREUR */
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR */
```

Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C.

Ces opérations sont impossibles et illégales parce que ***l'adresse représentée par le nom d'un tableau reste toujours constante***.

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (p.ex. dans une boucle) ou déléguer cette charge à une fonction de <*stdio*> ou <*string*>.

3. Pointeurs et tableaux

➢**Avantages des pointeurs sur char**

Comme la fin des chaînes de caractères est marquée par un symbole spécial, nous n'avons pas besoin de connaître la longueur des chaînes de caractères.

Nous pouvons même laisser de côté les indices d'aide et parcourir les chaînes à l'aide de pointeurs.

Cette façon de procéder est indispensable pour traiter de chaînes de caractères dans des fonctions.

D'où les avantages des pointeurs dans la définition de fonctions traitant des chaînes de caractères :

> Pour fournir un tableau comme paramètre à une fonction, il faut passer *l'adresse du tableau* à la fonction. Or, les *paramètres des fonctions sont des variables locales*, que nous pouvons utiliser comme variables d'aide.

Bref, une fonction obtenant une chaîne de caractères comme paramètre, dispose d'une *copie locale de l'adresse de la chaîne*. Cette copie peut remplacer les indices ou les variables d'aide du formalisme tableau.

3. Pointeurs et tableaux

Discussion d'un exemple

Reprenons l'exemple de la fonction **strcpy**, qui copie la chaîne CH2 vers CH1. Les deux chaînes sont les arguments de la fonction et elles sont déclarées comme *pointeurs sur **char***.

La première version de **strcpy** est écrite entièrement à l'aide du formalisme tableau:

```
void strcpy(char *CH1, char *CH2)
{
  int I;

  I = 0;
  while ( (CH1[I]=CH2[I]) != '\0' ) I++;
}
```

3. Pointeurs et tableaux

Dans une première approche, nous pourrions remplacer simplement la notation tableau[I] par *(tableau+I), ce qui conduirait au programme:

```
void strcpy(char *CH1, char *CH2)
{
  int I;

  I = 0;
  while ( (*(CH1+I)=*(CH2+I)) != '\0' ) I++;
}
```

3. Pointeurs et tableaux

Cette transformation ne nous avance guère, nous avons tout au plus gagné quelques millièmes de secondes lors de la compilation.

Un 'véritable' avantage se laisse gagner en calculant directement avec les pointeurs CH1 et CH2 :

```
void strcpy(char *CH1, char *CH2)
{
  while ((*CH1=*CH2) != '\0')
  {
    CH1++;
    CH2++;
  }
}
```

Un vrai professionnel en C escaladerait les 'simplifications' jusqu'à obtenir :

```
void strcpy(char *CH1, char *CH2)
{
  while (*CH1++ = *CH2++);
}
```

3. Pointeurs et tableaux

3.4. Pointeurs et tableaux à deux dimensions

L'arithmétique des pointeurs se laisse élargir avec *toutes* ses conséquences sur les tableaux à deux dimensions.

Voyons cela sur un exemple:

3. Pointeurs et tableaux

Exemple

Le tableau M à deux dimensions est défini comme suit:

```
int M[4][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}  };
```

Le nom du tableau M représente l'adresse du premier élément du tableau et pointe sur le ***tableau*** M[0] qui a la valeur :

```
{0,1,2,3,4,5,6,7,8,9}
```

L'expression (M+1) est l'adresse du deuxième élément du tableau et pointe sur M[1] qui a la valeur :

```
{10,11,12,13,14,15,16,17,18,19}
```

3. Pointeurs et tableaux

Problème

Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c.à-d.: aux éléments

`M[0][0], M[0][1], ... , M[3][9]` ?

3. Pointeurs et tableaux

Discussion

Une solution consiste à convertir la valeur de M (qui est un pointeur sur *un tableau du type **int**)* en un pointeur sur ***int***. On pourrait se contenter de procéder ainsi:

```
int M[4][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}   };
int *P;
P = M;   /* conversion automatique */
```

Cette dernière affectation entraîne une conversion automatique de l'adresse `&M[0]` dans l'adresse `&M[0][0]`.

(Remarquez bien que l'adresse transmise reste la même, seule la nature du pointeur a changé).

Cette solution n'est pas satisfaisante à cent pour-cent: Généralement, on gagne en lisibilité en explicitant la conversion mise en œuvre par l'opérateur de conversion forcée ("cast"), qui évite en plus des messages d'avertissement de la part du compilateur.

3. Pointeurs et tableaux

Solution

Voici finalement la version que nous utiliserons:

```
int M[4][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}  };
int *P;
P = (int *)M;  /* conversion forcée */
```

Dû à la mémorisation ligne par ligne des tableaux à deux dimensions, il nous est maintenant possible traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 4*10.

3. Pointeurs et tableaux

Exemple

Les instructions suivantes calculent la somme de tous les éléments du tableau M:

```
int M[4][10] = {  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                  {10,11,12,13,14,15,16,17,18,19},
                  {20,21,22,23,24,25,26,27,28,29},
                  {30,31,32,33,34,35,36,37,38,39}  };
int *P;
int I, SOM;

P = (int*)M;
SOM = 0;
for (I=0; I<40; I++) SOM += *(P+I);
```

3. Pointeurs et tableaux

Remarque

Lors de l'interprétation d'un tableau à deux dimensions comme tableau unidimensionnel ***il faut calculer avec le nombre de colonnes indiqué dans la déclaration*** du tableau.

Exemple

Pour la matrice A, nous réservons de la mémoire pour 3 lignes et 4 colonnes, mais nous utilisons seulement 2 lignes et 2 colonnes:

```
int A[3][4];A[0][0]=1;A[0][1]=2;A[1][0]=10;A[1][1]=20;
```

Dans la mémoire, ces composantes sont stockées comme suit :

L'adresse de l'élément `A[I][J]` se calcule alors par:

```
A + I*4 + J
```

4. Tableaux de pointeurs

Si nous avons besoin d'un ensemble de pointeurs du même type, nous pouvons les réunir dans un tableau de pointeurs.

Déclaration d'un tableau de pointeurs

`<Type> *<NomTableau>[<N>]` déclare un tableau `<NomTableau>` de `<N>` pointeurs sur des données du type `<Type>.`

4. Tableaux de pointeurs

Exemple

```
double *A[10];
```

déclare un tableau de 10 pointeurs sur des rationnels du type **double** dont les adresses et les valeurs ne sont pas encore définies.

Remarque

Le plus souvent, les tableaux de pointeurs sont utilisés pour mémoriser de façon économique des *chaînes de caractères de différentes longueurs*.

Dans la suite, nous allons surtout considérer les tableaux de pointeurs sur des chaînes de caractères.

4. Tableaux de pointeurs

Initialisation

Nous pouvons initialiser les pointeurs d'un tableau sur **char** par les adresses de chaînes de caractères constantes.

Exemple

```
char *JOUR[] = { "dimanche", "lundi", "mardi", "mercredi",
                 "jeudi", "vendredi", "samedi" };
```

déclare un tableau `JOUR[]` de 7 pointeurs sur **char**.

Chacun des pointeurs est initialisé avec l'adresse de l'une des 7 chaînes de caractères.

4. Tableaux de pointeurs

On peut afficher les 7 chaînes de caractères en fournissant les adresses contenues dans le tableau JOUR à **printf** (ou **puts**) :

```
int I;
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Comme JOUR[I] est un pointeur sur **char**, on peut afficher les premières lettres des jours de la semaine en utilisant l'opérateur 'contenu de' :

```
int I;
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```

L'expression JOUR[I]+J désigne la J-ième lettre de la I-ième chaîne. On peut afficher la troisième lettre de chaque jour de la semaine par:

```
int I;for (I=0; i<7; I++)
printf("%c\n",*(JOUR[I]+2));
```

5. Allocation dynamique de mémoire

Nous avons vu que l'utilisation de pointeurs nous permet de mémoriser économiquement des données de différentes grandeurs.

Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin.

Nous parlons alors de *l'allocation dynamique* de la mémoire.

Revoyons d'abord de quelle façon la mémoire a été réservée dans les programmes que nous avons écrits jusqu'ici.

5. Allocation dynamique de mémoire

5.1. Déclaration statique de données

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire.

Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des données.

Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation.

Nous parlons alors de la ***déclaration statique*** des variables.

5. Allocation dynamique de mémoire

Exemples

```
float A, B, C;                  /* réservation de 12 octets */

short D[10][20];                /* réservation de 400 octets */

char E[] = {"Bonjour !"};  /* réservation de 10  octets */

char F[][10] = {"un", "deux", "trois", "quatre"};
                                /* réservation de 40  octets */
```

5. Allocation dynamique de mémoire

Pointeurs

Le nombre d'octets à réserver pour un *pointeur* dépend de la machine et du 'modèle' de mémoire choisi, mais il est déjà connu lors de la compilation.

Un pointeur est donc aussi déclaré statiquement.

Supposons dans la suite qu'un pointeur ait besoin de p octets en mémoire. (En DOS: p =2 ou p = 4)

Exemples

```
double *G;      /* réservation de p    octets */
char *H;        /* réservation de p    octets */
float *I[10];   /* réservation de 10*p octets */
```

5. Allocation dynamique de mémoire

Chaînes de caractères constantes

L'espace pour les *chaînes de caractères constantes* qui sont affectées à des pointeurs ou utilisées pour initialiser des pointeurs sur **char** est aussi réservé automatiquement :

Exemples

```
char *J = "Bonjour !";   /* réservation de p+10 octets */

float *K[] = {"un", "deux", "trois", "quatre"};
                         /* réservation de 4*p+3+5+6+7 octets */
```

5. Allocation dynamique de mémoire

5.2. Allocation dynamique

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.

5. Allocation dynamique de mémoire

Exemple

Nous voulons lire 10 phrases au clavier et mémoriser les phrases en utilisant un tableau de pointeurs sur **char**.

Nous déclarons ce tableau de pointeurs par:

```
char *TEXTE[10];
```

Pour les 10 pointeurs, nous avons besoin de 10*p octets.

Ce nombre est connu dès le départ et les octets sont réservés automatiquement.

Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...

5. Allocation dynamique de mémoire

Allocation dynamique

La réservation de la mémoire pour les 10 phrases peut donc seulement se faire *pendant l'exécution du programme*.

Nous parlons dans ce cas de l'***allocation dynamique*** de la mémoire.

5. Allocation dynamique de mémoire

5.3. La fonction malloc et l'opérateur sizeof

La fonction **malloc** de la bibliothèque *<stdlib>* nous aide à localiser et à réserver de la mémoire au cours d'un programme.

Elle nous donne accès au tas *(heap)*; c.-à-d. à l'espace en mémoire laissé libre une fois mis en place le système d'exploitation (Windows, Linux, …) le programme lui-même et la pile *(stack)*.

5. Allocation dynamique de mémoire

La fonction malloc

`malloc( <N> )` fournit l'adresse d'un bloc en mémoire de <N> octets libres ou la valeur zéro s'il n'y a pas assez de mémoire.

Exemple

L'instruction: `T = malloc(4000);`

fournit l'adresse d'un bloc de 4000 octets libres et l'affecte à T.
S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

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.

5. Allocation dynamique de mémoire

***L'opérateur unaire* `sizeof`**

`sizeof <var>` fournit la grandeur de la variable `<var>`

`sizeof <const>` fournit la grandeur de la `constante <const>`

`sizeof(<type>)` fournit la grandeur pour un objet du type `<type>`

5. Allocation dynamique de mémoire

Exemple

Après la déclaration,

```
short A[10];
char B[5][10];
```

nous obtenons les résultats suivants sur un PC 16 bits :

`sizeof A`	s'évalue à 20
`sizeof B`	s'évalue à 50
`sizeof 4.25`	s'évalue à 8
`sizeof "Bonjour !"`	s'évalue à 10
`sizeof(float)`	s'évalue à 4
`sizeof(double)`	s'évalue à 8

5. Allocation dynamique de mémoire

Exemple

Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X est lue au clavier:

```
int X;
int *PNum;

printf("Introduire le nombre de valeurs :");
scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

5. Allocation dynamique de mémoire

5.4. La fonction free

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de `malloc`, alors nous pouvons le libérer à l'aide de la fonction `free` de la bibliothèque `<stdlib>`.

`free( <Pointeur> )` libère le bloc de mémoire désigné par le `<Pointeur>;` n'a pas d'effet si le pointeur a la valeur zéro.

5. Allocation dynamique de mémoire

Remarque

➢La fonction **free** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par **malloc**.

➢La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.

➢Si nous ne libérons pas explicitement la mémoire à l'aide **free**, alors elle est libérée automatiquement à la fin du programme.