Les pointeurs

MP2I - Informatique

Anthony Lick
Lycée Janson de Sailly

Les pointeurs

Pointeur

Toute variable manipulée dans un programme est stockée quelque part en mémoire.

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 (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets).

Les pointeurs

Pointeur

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.

Adresse et valeur d'un objet

Adresse et valeur d'un objet

Left value

On appelle **Lvalue** (**left value**) tout objet pouvant être placé à gauche d'un opérateur d'affectation.

Une **Lvalue** est caractérisée par :

- son **adresse**, c'est à dire l'adresse mémoire à partir de laquelle l'objet est stocké ;
- sa **valeur**, c'est à dire ce qui est stocké à cette adresse.

Adresse et valeur d'un objet

Adresse

L'**adresse** d'un objet étant le numéro d'un octet de la mémoire, il s'agit d'un entier (quelque soit le type d'objet à cette adresse).

Le format interne de cet entier (16 bits, 32 bits, ou 64 bits) dépend de l'architecture de la machine utilisée.

Sur les ordinateurs récents, une adresse a le format d'un entier long (64 bits).

Adresse et valeur d'un objet

Opérateur &

L'opérateur **&** permet d'accéder à l'adresse d'une variable.

Toutefois, si **i** est une variable, **&i** n'est pas une **Lvalue** mais une **constante** : on ne peut pas faire figurer **&i** à gauche d'une opération d'affectation.

Pour pouvoir manipuler les adresses, on doit donc recourir à un nouveau type d'objets : les **pointeurs**.

Adresse et valeur d'un objet

Exemple

```c
int main()
{
  int i, j;
  i = 3;
  j = i;
  printf("variable i : adresse = %p ; valeur = %d\n", &i, i);
  printf("variable j : adresse = %p ; valeur = %d\n", &j, j);
  return 0;
}
```

Output

```
variable i : adresse = 0x7fff071f720c ; valeur = 3
variable j : adresse = 0x7fff071f7208 ; valeur = 3
```

Exemple

Le compilateur va par exemple placer la variable **`i`** à l'adresse mémoire `0x7fff071f720c`, et la variable **`j`** à l'adresse `0x7fff071f7208`.

Deux variables différentes ont des adresses différentes.

L'affectation **`i = j;`** n'opère que sur les valeurs des variables.

Adresse et valeur d'un objet

Exemple

```c
int main()
{
  int i, j;
  i = 3;
  j = i;
  printf("variable i : adresse = %p ; valeur = %d\n", &i, i);
  printf("variable j : adresse = %p ; valeur = %d\n", &j, j);
  return 0;
}
```

Output

```
variable i : adresse = 0x7fff071f720c ; valeur = 3
variable j : adresse = 0x7fff071f7208 ; valeur = 3
```

Exemple

Les variables **i** et **j** étant de type **int**, elles sont stockées sur 4 octets.

Ainsi, la variable **j** est stockée sur les octets d'adresse `0x7fff071f7208` à `0x7fff071f720b`.

Notion de pointeur

Les pointeurs

```
1  type *nom_du_pointeur;
```

Pointeur

Un **pointeur** est un objet (**Lvalue**) dont la valeur est égale à l'adresse d'un autre objet. Comme pour n'importe quelle **Lvalue**, sa valeur est modifiable.

On déclare un pointeur via l'instruction ci-dessus, 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.

Les pointeurs

Type d'un pointeur

Même si la valeur d'un pointeur est toujours un entier, le type d'un pointeur dépend du type de l'objet vers lequel il pointe.

Si l'objet pointé est de type **`type`**, le pointeur est de type **`type*`**.

Cette distinction est indispensable à l'interprétation de la valeur du pointeur, car on doit connaître le type de l'objet pointé pour savoir combien d'octets il faut lire dans la mémoire à partir de l'adresse stockée dans le pointeur pour obtenir la valeur de l'objet pointé.

Les pointeurs

Exemple

- Un pointeur de type `char*` pointe vers un objet de type `char`. Sa valeur donne l'adresse de l'octet où cet objet est stocké.
- Un pointeur de type `int*` pointe vers un objet de type `int`. Sa valeur donne l'adresse du premier des 4 octets où l'objet est stocké.

Les pointeurs

Exemple

```c
#include <stdio.h>

int main()
{
  int i = 3;
  int *p;
  p = &i;
  printf("variable i : adresse = %p ; valeur = %d\n", &i, i);
  printf("variable p : adresse = %p ; valeur = %p\n", &p, p);
}
```

Output

```
variable i : adresse = 0x7ffd472c61cc ; valeur = 3
variable p : adresse = 0x7ffd472c61c0 ; valeur = 0x7ffd472c61cc
```

Exemple

Le code ci-dessus définit un pointeur **p** qui pointe vers un entier **i**.

La **valeur** de **p** est donc l'**adresse** de **i**.

Les pointeurs

Opérateur d'indirection

L'**opérateur unaire d'indirection** `*` permet d'accéder directement à la valeur de l'objet pointé.

Exemple

Si **`p`** est un pointeur vers un entier **`i`**, **`*p`** désigne la valeur de **`i`**.

Les pointeurs

Exemple

```c
#include <stdio.h>

int main()
{
  int i = 3;
  int *p;
  p = &i;
  printf("variable i : adresse = %p ; valeur = %d\n", &i, i);
  printf("variable p : adresse = %p ; valeur = %p\n", &p, p);
  printf("*p = %d\n", *p);
}
```

Output

```
variable i : adresse = 0x7ffd472c61cc ; valeur = 3
variable p : adresse = 0x7ffd472c61c0 ; valeur = 0x7ffd472c61cc
*p = 3
```

Exemple

Si **p** est un pointeur vers un entier **i**, ***p** désigne la valeur de **i**.

Les pointeurs

Exemple

```c
#include <stdio.h>

int main()
{
  int i = 3;
  int *p;
  p = &i;
  printf("variable i : adresse = %p ; valeur = %d\n", &i, i);
  printf("variable p : adresse = %p ; valeur = %p\n", &p, p);
  printf("*p = %d\n", *p);
}
```

Output

```
variable i : adresse = 0x7ffd472c61cc ; valeur = 3
variable p : adresse = 0x7ffd472c61c0 ; valeur = 0x7ffd472c61cc
*p = 3
```

Exemple

Dans ce programme, les objets **i** et ***p** sont identiques : ils ont mêmes adresse et valeur.

Cela signifie en particulier que toute modification de ***p** modifie **i**.

Les pointeurs

Exemple

```c
#include <stdio.h>

int main()
{
  int i = 3;
  int *p;
  p = &i;
  printf("variable i : adresse = %p ; valeur = %d\n", &i, i);
  printf("variable p : adresse = %p ; valeur = %p\n", &p, p);
  printf("*p = %d\n", *p);
}
```

Output

```
variable i : adresse = 0x7ffd472c61cc ; valeur = 3
variable p : adresse = 0x7ffd472c61c0 ; valeur = 0x7ffd472c61cc
*p = 3
```

Exemple

À la fin du programme ci-dessus, si l'on ajoute l'instruction `*p = 0;` la valeur de **`i`** devient nulle.

Les pointeurs

p vs *p

On peut donc dans un programme manipuler à la fois les objets **p** et ***p**.

Mais ces deux manipulations sont très différentes.

Les pointeurs

```c
#include <stdio.h>

int main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
  return 0;
}
```

```c
#include <stdio.h>

int main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
  return 0;
}
```

objet	adresse	valeur
i	0x7fff8dd69a0c	3
j	0x7fff8dd69a08	6
p1	0x7fff8dd69a00	0x7fff8dd69a0c
p2	0x7fff8dd699f8	0x7fff8dd69a08

Exemple

Dans chacun des programmes, la configuration de la mémoire est la même avant la ligne 9.

Les pointeurs

```c
#include <stdio.h>

int main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;
  return 0;
}
```

```c
#include <stdio.h>

int main()
{
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  p1 = p2;
  return 0;
}
```

objet	adresse	valeur
i	0x7fff8dd69a0c	6
j	0x7fff8dd69a08	6
p1	0x7fff8dd69a00	0x7fff8dd69a0c
p2	0x7fff8dd699f8	0x7fff8dd69a08

objet	adresse	valeur
i	0x7fff8dd69a0c	3
j	0x7fff8dd69a08	6
p1	0x7fff8dd69a00	0x7fff8dd69a08
p2	0x7fff8dd699f8	0x7fff8dd69a08

Exemple

Cependant, l'instruction de la ligne 9 n'a pas le même effet sur la mémoire.

Arithmétique des pointeurs

Arithmétique des pointeurs

Opérateurs arithmétiques

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 du même type que le pointeur de départ ;
- la **soustraction** d'un entier à un pointeur : le résultat est un pointeur du même type que le pointeur de départ ;
- la **différence** entre deux pointeurs de même type : le résultat est un entier.

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

Arithmétique des pointeurs

Fonctionnement

Si **`i`** est un entier, et **`p`** est un pointeur de type **`type*`**, alors l'expression **`p + i`** désigne un pointeur 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.

Arithmétique des pointeurs

Pointeur sur int

```c
#include <stdio.h>

int main()
{
  int i = 3;
  int *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %p\n", p1);
  printf("p2 = %p\n", p2);
  return 0;
}
```

Output

```
p1 = 0x7ffff2d82710
p2 = 0x7ffff2d82714
```

Pointeur sur double

```c
#include <stdio.h>

int main()
{
  double i = 3;
  double *p1, *p2;
  p1 = &i;
  p2 = p1 + 1;
  printf("p1 = %p\n", p1);
  printf("p2 = %p\n", p2);
  return 0;
}
```

Output

```
p1 = 0x7ffff2d82710
p2 = 0x7ffff2d82718
```

Exemple

p1 + 1 augmente l'adresse pointée par **p1** :

- de 4 si **p1** est de type **int***;
- de 8 si **p1** est de type **double***.

Arithmétique des pointeurs

Opérateurs de comparaison

Les **opérateurs de comparaison** sont également applicables aux pointeurs, à condition de comparer deux pointeurs de même type.

Tableaux

L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux : si `p` pointe vers la première case d'un tableau, alors `p + i` pointe vers la case d'indice i.

Arithmétique des pointeurs

```c
#include <stdio.h>
#define N 5

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

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

Output

```
ordre croissant:
1
2
6
0
7

ordre decroissant:
7
0
6
2
1
```

Exemple

Le programme ci-dessus imprime les éléments du tableau **tab** dans l'ordre croissant puis décroissant des indices.

Arithmétique des pointeurs

Différence entre deux pointeurs

Si p et q sont deux pointeurs de type **`type*`**, l'expression **`p - q`** désigne un entier dont la valeur est égale à $(\mathtt{p}-\mathtt{q})/$**`sizeof`**$(\mathtt{type})$.

Cette valeur correspond au nombre d'objets de type **`type`** que l'on peut stocker de manière contiguë entre les adresses **`p`** et **`q`**.

Allocation dynamique

Allocation dynamique

Allocation de mémoire

Avant de manipuler un pointeur, et notamment de lui appliquer l'opérateur d'indirection `*`, il faut l'initialiser par une adresse mémoire.

Si on ne connaît pas encore son adresse, il est conseillé d'initialiser la valeur d'un pointeur par la constante symbolique notée `NULL`, définie dans `stdio.h`. En général, cette constante vaut 0.

Le test `p == NULL` permet alors de savoir si le pointeur **p** pointe ou non vers un objet.

Allocation dynamique

Allocation de mémoire

On peut initialiser **p** par une affectation sur **p**. Par exemple, on peut affecter à **p** l'adresse d'une variable **var** via **p = &var**.

Mais il n'est pas obligatoire que l'adresse pointée par **p** soit l'adresse d'une variable existante. Dans ce cas, on peut directement travailler avec la valeur de ***p**, mais il faut d'abord réserver à **p** un espace mémoire de taille adéquate.

Allocation dynamique

Cette opération consistant à réserver un espace mémoire pour stocker l'objet pointé s'appelle **allocation dynamique**. Elle se fait en C via la fonction **`malloc`** de la librairie standard `stdlib.h`. Sa signature est la suivante :

```
void* malloc(size_t nombre_octets)
```

Le type **`size_t`** est un type entier utilisé pour désigner une taille en mémoire (en nombre d'octets). On peut obtenir la taille d'un **`type`** à l'aide de **`sizeof(type)`**.

Puisque la fonction **`malloc`** revoie des adresses de types différents selon la situation (**`int*`**, **`double*`**, **`char*`**, ...), son type de retour est **`void*`** : ce type joue un rôle spécial et le type de la valeur de retour s'adaptera au type de pointeur utilisé.

Allocation dynamique

```c
#include <stdlib.h>

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

Exemple

Le code ci-dessus initialise un pointeur vers un entier.

Remarque

On aurait pu écrire également `p = malloc(4);` si l'on sait qu'un objet de type `int` est stocké sur 4 octets.

Mais on préfèrera utiliser `sizeof`, ce qui produit un code **portable** (qui ne dépend pas de la machine utilisée).

Allocation dynamique

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

int main()
{
  int i = 3;
  printf("adresse de i = %p\n", &i);
  int *p = NULL;
  printf("valeur de p avant initialisation = %p\n", p);
  p = malloc(sizeof(int));
  printf("valeur de p apres initialisation = %p\n", p);
  *p = i;
  printf("valeur de *p = %d\n", *p);
  return 0;
}
```

Output

```
adresse de i = 0x7ffccad3f934
valeur de p avant initialisation = (nil)
valeur de p apres initialisation = 0x55f7abddb6b0
valeur de *p = 3
```

Exemple

Le programme ci-dessus définit un pointeur **p** vers un objet ***p** de type **int**, et affecte à ***p** la valeur de la variable **i**.

Allocation dynamique

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

int main()
{
  int *p;
  int i = *p;
  return 0;
}
```

Output

```
Segmentation fault (core dumped)
```

Exemple

Si on essaie d'accéder à une adresse mémoire invalide (par exemple, en oubliant d'initialiser un pointeur à l'aide de **`malloc`**), on obtient une **erreur de segmentation** à l'exécution du programme.

Allocation dynamique

Espace mémoire contigu

La fonction **`malloc`** permet également d'allouer un espace pour plusieurs objets contigus en mémoire.

Allocation dynamique

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

int main()
{
  int i = 3;
  int j = 6;
  int *p;
  p = malloc(2 * sizeof(int));
  *p = i;
  *(p + 1) = j;
  printf("p = %p \t *p = %d \np+1 = %p \t *(p+1) = %d \n",p,*p,p+1,*(p+1));
}
```

Output

```
p = 0x5632161c22a0      *p = 3
p+1 = 0x5632161c22a4    *(p+1) = 6
```

Exemple

Dans le programme ci-dessus, on a réservé, à l'adresse pointée par **p**, 8 octets en mémoire, qui permettent de stocker 2 objets de type **int**.

Allocation dynamique

```
1  free(nom_du_pointeur);
```

`free`

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

Ceci se fait à l'aide de la fonction **`free`**, dont la syntaxe est celle ci-dessus.

Attention

À toute instruction **`malloc`** doit être associée une instruction **`free`**.

Si on ne le fait pas, notre programme peut provoquer des **fuites de mémoire** (**memory leak** en anglais).

Pointeurs et tableaux

Pointeurs et tableaux

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

Pointeurs et tableaux à une dimension

Pointeur vers un tableau

Tout tableau en C est en fait un pointeur constant (non modifiable), dont la valeur est l'adresse du premier élément du tableau.

Exemple

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

Exemple

Dans la déclaration ci-dessus, **tab** est un pointeur dont la valeur est **&tab[0]**, c'est-à-dire l'adresse de **tab[0]**.

Pointeurs et tableaux à une dimension

Sucre syntaxique

On a vu dans le chapitre précédent que **`tab[i]`** permet d'accéder au i-ème élément de **`tab`**.

En fait, si **`p`** est un pointeur, **`p[i]`** accède à l'élément stocké à l'adresse **`p + i`**.

Autrement dit, l'instruction **`p[i]`** est équivalente à l'instruction **`*(p + i)`**.

Pointeurs et tableaux à une dimension

Exemple

```c
#define N 5

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

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

Exemple

```c
#define N 5

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

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

Exemple

Les deux programmes ci-dessus sont équivalents.

Cependant, on préfèrera utiliser la version de droite, qui est plus claire.

Allocation dynamique de tableaux

Allocation dynamique

Toutefois la manipulation de tableaux, et non de pointeurs, possède des inconvénients dûs au fait qu'un tableau est un pointeur constant :

- on ne peut pas créer de tableaux dont la taille est une variable du programme (taille du tableau non connue à l'avance) ;
- on ne peut pas créer du tableaux bidimensionnels dont les lignes n'ont pas tous le même nombre d'éléments.

Ces opérations deviennent possibles dès qu'on manipule des pointeurs **alloués dynamiquement**.

Allocation dynamique de tableaux

Allocation dynamique

```c
#include <stdlib.h>

int main()
{
  int n;

  /* calcul de n */

  // Création d'un tableau d'entiers de taille n
  int *tab;
  tab = malloc(n * sizeof(int));

  /* utilisation de tab */

  // Ne pas oublier de libérer la mémoire
  free(tab);
}
```

Exemple

Dans le code ci-dessus, on peut utiliser **`tab`** comme n'importe quel tableau (accès au i-ème élément avec **`tab[i]`**, ...).

Pointeurs et tableaux à plusieurs dimensions

Tableaux multidimensionnels

Un tableau à deux dimensions est, par définition, un tableau de tableaux. Il s'agit donc en fait d'un pointeur vers un pointeur !

Pointeurs et tableaux à plusieurs dimensions

```
1  int tab[M][N];
```

Tableaux multidimensionnels

Dans le code ci-dessus, **tab** est un pointeur, qui pointe lui-même vers un objet de type **int*** (type pointeur d'entier).
Le pointeur **tab** contient l'adresse de **tab[0]** (**&tab[0]**), et **tab[0]** contient l'adresse de **tab[0][0]** (**&tab[0][0]**).
De même, pour $i \in [\![0, M-1]\!]$, **tab[i]** contient l'adresse de **tab[i][0]** (**&tab[i][0]**).

Pointeurs et tableaux à plusieurs dimensions

```
1  int tab[M][N];
```

Tableaux multidimensionnels

Autrement dit, **tab** est un tableau d'éléments de type **int***, et chaque **tab[i]** est un tableau d'éléments de type **int**.

tab étant un pointeur pointant vers un élément de type **int***, **tab** est donc de type **int****.

Pointeurs et tableaux à plusieurs dimensions

Allocation dynamique

Exactement comme pour les tableaux à une dimension, les pointeurs de pointeurs ont de nombreux avantages sur les tableaux multi-dimensionnés.

Un pointeur qui pointe sur un objet de type **type*** (deux dimensions) est de type **type****. On le déclare donc via :

```
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;
```

Pointeurs et tableaux à plusieurs dimensions

Exemple

```
1  #include <stdlib.h>
2
3  int main()
4  {
5    int m, n;
6
7    /* calcul de m et n */
8
9    int **tab;
10   tab = malloc(m * sizeof(int*));
11
12   int i;
13   for (i = 0; i < m; i++)
14   {
15     tab[i] = malloc(n * sizeof(int));
16   }
17
18   /* utilisation du tableau */
19
20   // Libération de la mémoire
21   for (i = 0; i < m; i++)
22   {
23     free(tab[i]);
24   }
25   free(tab);
26 }
```

Exemple

Le code ci-contre déclare dynamiquement un tableau à m lignes et n colonnes.

À la ligne 10, une première allocation dynamique réserve pour l'objet pointé par **tab** l'espace mémoire correspondant à m pointeurs sur des entiers (un par ligne).

Pointeurs et tableaux à plusieurs dimensions

Exemple

```
1  #include <stdlib.h>
2
3  int main()
4  {
5    int m, n;
6
7    /* calcul de m et n */
8
9    int **tab;
10   tab = malloc(m * sizeof(int*));
11
12   int i;
13   for (i = 0; i < m; i++)
14   {
15     tab[i] = malloc(n * sizeof(int));
16   }
17
18   /* utilisation du tableau */
19
20   // Libération de la mémoire
21   for (i = 0; i < m; i++)
22   {
23     free(tab[i]);
24   }
25   free(tab);
26 }
```

Exemple

Aux lignes 12 à 16, on alloue dynamiquement, pour chaque pointeur **`tab[i]`**, l'espace mémoire nécessaire pour stocker n entiers.

Pointeurs et tableaux à plusieurs dimensions

Flexibilité

Contrairement aux tableaux déclarés statiquement, on peut choisir des tailles différentes pour chaque ligne d'un tableau alloué dynamiquement.

Exemple

```
1  for (i = 0; i < m; i++)
2  {
3    tab[i] = malloc((i+1) * sizeof(int));
4  }
```

Exemple

Le code ci-dessus déclare un tableau *triangulaire* (et non *rectangulaire*), où la i-ème ligne possède $i + 1$ cases.

Pointeurs et chaînes de caractères

Pointeurs et chaînes de caractères

Chaînes de caractères

On a vu précédemment qu'un chaîne de caractères était un tableau à une dimension d'objets de type **`char`**, se terminant par le caractère "`\0`".

On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur de type **`char`**`*`.

Pointeurs et chaînes de caractères

Exemple

```
1  #include <stdio.h>
2
3  int main()
4  {
5    char *chaine;
6    chaine = "chaine de caracteres";
7
8    int i;
9    for (i = 0; *chaine != '\0'; i++)
10   {
11     chaine++; // saute au prochain caractère de la chaine
12   }
13   printf("nombre de caracteres = %d\n",i);
14 }
```

Output

```
nombre de caracteres = 20
```

Exemple

Le programme ci-dessus mesure la longueur d'une chaîne de caractères. L'instruction de la ligne 11 fait pointé **chaine** vers le caractère suivant, et la boucle **for** s'arrête quand **chaine** pointe vers "\0".

Pointeurs et chaînes de caractères

string.h

La librairie standard **string.h** possède une fonction **strlen** qui procède de manière identique à l'exemple précédent.

Ainsi, l'instruction ci-dessous calcule la longueur d'une chaîne de caractères.

```
1  strlen(chaine);
```

Pointeurs et chaînes de caractères

Exemple

Le code suivant crée une chaîne de caractères correspondant à la concaténation de deux autres chaînes.

Pointeurs et chaînes de caractères

Exemple

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

int main()
{
  char *chaine1, *chaine2;
  chaine1 = "chaine ";
  chaine2 = "de caracteres";
  char *res; // res pointe vers le début d'une nouvelle chaine
  res = malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));
  char *p; // p pointe vers le premier caractère de res
  p = res;
  int i;
  for (i = 0; i < strlen(chaine1); i++) // on recopie chaine1
  {
    *p = chaine1[i];
    p++;
  }
  for (i = 0; i < strlen(chaine2); i++) // on recopie chaine2
  {
    *p = chaine2[i];
    p++;
  }
  printf("%s\n",res);
}
```

Output

```
chaine de caracteres
```

Pointeurs et chaînes de caractères

Remarque

Il faut veiller à ne pas modifier la valeur de **`res`**, car sinon on perdrait l'adresse du début de la chaîne.

C'est pour cela qu'on utilise un autre pointeur **`p`**, qu'on initialise à **`res`**, et qu'on incrémente au fur et à mesure du programme pour pointer successivement vers les différents caractères de la chaîne **`res`**.

Attention

Si on n'utilise pas de pointeur intermédiaire **`p`**, et qu'on modifie directement **`res`**, **`res`** pointe donc vers la fin de la chaîne de caractère à la fin du programme.

Pointeurs et chaînes de caractères

Exemple

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

int main()
{
  char *chaine1, *chaine2;
  chaine1 = "chaine ";
  chaine2 = "de caracteres";

  char *res;
  res = malloc((strlen(chaine1) + strlen(chaine2)) * sizeof(char));

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

Output

```
nombre de caracteres de res = 0
```

Pointeurs et structures

Pointeurs et structures

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 champ de la structure.

Pointeurs et structures

Exemple (1/2)

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

struct eleve
{
  char nom[20];
  int date;
};
typedef struct eleve *classe;
```

Output

```
nombre d'eleves de la classe = 3
saisie de l'eleve numero 0
nom de l'eleve = Janson
date de naissance JJMMAA = 010203
saisie de l'eleve numero 1
nom de l'eleve = Sailly
date de naissance JJMMAA = 040506
saisie de l'eleve numero 2
nom de l'eleve = Bob
date de naissance JJMMAA = 121212
Entrez un numero :2
Eleve numero 2:
nom = Bob
date de naissance = 121212
```

Exemple (2/2)

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

Pointeurs et structures

Pointeur vers une structure

Si **p** est un pointeur sur une structure, on peut accéder à un champ de la structure pointée via :

```
(*p).champ
```

Attention

L'usage de parenthèses est ici indispensable car l'opérateur d'indirection `*` à une priorité plus élevée que l'opérateur de membre de structure.

Pointeurs et structures

Sucre syntaxique

```
(*p).champ
```

La notation ci-dessus peut être simplifiée grâce à l'opérateur **pointeur de champ de structure**, noté **->**.

Ainsi, l'expression ci-dessus est strictement équivalente à :

```
p->champ
```