

Introduction au langage C - Partie 3

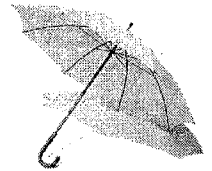
et protection contre quelques bugs

Nancy-Université
Université
Henri Poincaré

Brigitte Wrobel-Dautcourt

Octobre 2007

*Département Informatique - Faculté des Sciences
Université Henri Poincaré - Nancy 1*



Partie3

- 11. Les structures de données (tableaux)
 - 12. Adresses et pointeurs
 - 13. La gestion de la mémoire
-

11. Les structures de données (tableaux)

Introduction au langage C
└ Les structures de données (tableaux)

Les tableaux

- ▶ un tableau est caractérisé par :
 - ▶ identificateur
 - ▶ type
 - ▶ dimension
 - ▶ classe d'allocation
- ▶ syntaxe de la déclaration


```
type identificateur [ taille ] ( [ taille ] ) * ;
```
- ▶ taille doit être un nombre entier positif constant


```
char tab[5] ;
int n = 5 ; char tab[n] ;
void f(int n)
{ char tab[n] ;
```

127 / 237

Introduction au langage C
└ Les structures de données (tableaux)

- ▶ les indices commencent **obligatoirement** à 0
- ▶ la déclaration d'un tableau de 10 entiers d'indices 0 à 9


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

 se traduit par :
 - ▶ réservation d'une zone de (10 * taille(int)) octets consécutifs en mémoire
 - ▶ pas de **new** !! :=)
 - ▶ association du nom **tab** à l'adresse de début de la zone mémoire réservée

tab	0	1	2	3	4	5	6	7	8	9
	□	□	□	□	□	□	□	□	□	□

128 / 237

Un tableau est une suite ordonnée d'éléments de **même type**.

La déclaration d'un tableau s'effectue au moyen de l'opérateur [] :

```
float tab[10] ;
```

Chaque élément d'un tableau est identifié par un indice représentant sa position dans la suite des éléments. Le premier élément est toujours à l'indice 0.

On accède aux éléments d'un tableau par l'opérateur [] ; ainsi **tab[5]** permet d'accéder au 6^{ème} élément du tableau.

Lors de la déclaration d'un tableau, on peut préciser sa taille (constante ou paramètre d'une fonction), cette allocation de mémoire est gérée par le compilateur, on parle alors d'**allocation statique** de mémoire).

Si l'utilisateur préfère gérer lui-même l'allocation et la libération de la mémoire, on utilise alors un autre mécanisme d'allocation : l'**allocation dynamique** de mémoire, présenté page 128.

Introduction au langage C
└ Les structures de données (tableaux)

```
short tab[10], i ;
char nom[] = "chat" ;

for (i = 0 ; i < 10 ; i++) /* version correcte */
  tab[i] = -1 ;

for (i = 1 ; i <= 10 ; i++)
  tab[i] = -1 ;

tab[12] = -1 ;
```

→ indices non valides non détectés

tab											i	nom
	0	1	2	3	4	5	6	7	8	9		

129 / 237

Introduction au langage C
└ Les structures de données (tableaux)

L'initialisation d'un tableau monodimensionnel

- ▶

```
int t[] = {0, 1, 2, 1, 0} ;
```


le tableau **t** possède 5 éléments
- ▶

```
short pairs[5] = {2, 4, 6} ;
```


les éléments suivants sont alors initialisés à 0. ce qui est équivalent à :


```
pairs[0] = 2 ;
pairs[1] = 4 ;
pairs[2] = 6 ;
pairs[3] = pairs[4] = 0 ;
```
- ▶

```
#define N 100
int tableau[N] = { 0 } ;
```


permet d'initialiser tous les éléments du tableau à 0 (écriture rapide !)

130 / 237

Introduction au langage C
 ↳ Les structures de données (tableaux)

Les chaînes de caractères

- ▶ une chaîne = un tableau de caractères
- ▶ instruction de déclaration d'un tableau de 50 caractères :

```
char nom[50] ;
```

- ▶ utilisation sous forme d'un tableau de caractères : 50 cases au maximum.
- ▶ utilisation sous forme d'une chaîne : 49 cases au maximum.

Contrainte : le dernier caractère de la chaîne doit être le caractère nul ('\0') afin de permettre l'utilisation des fonctions C manipulant les chaînes.

131 / 237

Introduction au langage C
 ↳ Les structures de données (tableaux)

exemple :

```
for (i = 0 ; i < N ; i++)
  tab[i] = i + 0x30 ;

for (i = 0 ; i < N ; i++)
  printf("%c", tab[i]) ;
printf("\n") ;

printf("%s\n", tab)
```

Faux car aucun élément de tab ne contient le caractère '\0'.
 → le contenu de la mémoire va être affiché, en considérant chaque octet comme le code ASCII d'un caractère, à partir de l'adresse donnée par tab et jusqu'à rencontrer un octet nul.

132 / 237

Introduction au langage C
 ↳ Les structures de données (tableaux)

- ▶ Les deux initialisations suivantes sont équivalentes :

```
(a) char mess[] = {'m','e','s','s','a','g','e','\0'} ;
(b) char mess[] = "message"
```

représentation en mémoire :

```
'm' 'e' 's' 's' 'a' 'g' 'e' '\0'
```

133 / 237

Introduction au langage C
 ↳ Les structures de données (tableaux)

Les fonctions C de manipulation de chaînes de caractères

- ▶ déclaration et initialisation d'une chaîne de caractères :

```
char chaine[] = "bonjour" ;
```

- ▶ affectation d'une chaîne de caractères :

```
chaine = "bonjour"
```

il faut utiliser la fonction `strcpy()` de copie d'une chaîne dans une autre :

```
strcpy(chaine, "bonjour") ;
```

134 / 237

Introduction au langage C
 Les structures de données (tableaux)

Les fonctions C de manipulation de chaînes de caractères - quelques exemples

<code>strcpy</code>	copie une chaîne dans une autre
<code>strncpy</code>	copie seulement les <i>n</i> premiers caractères d'une chaîne dans une autre
<code>strlen</code>	retourne la longueur d'une chaîne
<code>strcmp</code>	compare deux chaînes
<code>strcat</code>	concatène deux chaînes

136 / 237

Voir aussi page 121, la déclaration et l'initialisation des chaînes de caractères.

Remarque : l'affectation

```
char chaine[100] ;
...
chaine = (const char*) "bonjour" ;
```

ne fonctionne pas malgré tous les souhaits et incantations des étudiants...

Les fonctions C de manipulation de chaînes de caractères

```
#include <string.h>
```

Il faut inclure le fichier `string.h` contenant entre autre, les déclarations des profils des fonctions et la définition du type `size_t` :

```
typedef unsigned int size_t;
```

```
char *strcat(char *dest, char *src)
```

concatène les chaînes *dest* et *src*, (ajoute la chaîne *src* à la fin de la chaîne *dest*)
 retourne le résultat dans *dest*

```
char *strncat(char *dest, char *src, int n)
```

concatène *dest* et les *n* premiers caractères de *src*,
 retourne le résultat dans *dest*

```
strcmp(char *str1, char *str2)
```

compare les deux chaînes *str1* et *str2*,
 retourne une valeur : < 0 si *str1* < *str2*
 0 si *str1* = *str2*
 > 0 si *str1* > *str2*

```
char strncmp(char *str1, char *str2, int n)
```

compare les *n* premiers caractères des deux chaînes *str1* et *str2*

retourne une valeur : < 0 si *str1* < *str2*
 0 si *str1* = *str2*
 > 0 si *str1* > *str2*

```
char *strcpy(char *dest, char *src)
```

copie *src* dans *dest*,
 retourne le résultat dans *dest*

```
char *strncpy(char *dest, char *src, int n)
```

copie les *n* premiers caractères de *src* dans *dest*,
 retourne *dest*

```
size_t strlen(char *src)
```

retourne la longueur de *src*

```
char *strchr(char *src, int c)
```

retourne un pointeur sur la première occurrence de *c* dans *src*,
 retourne `NULL` si absence

c est bien un caractère passé sous forme d'un entier à la fonction `strchr()`

```
char *strrchr(char *src, int c)
```

retourne un pointeur sur la dernière occurrence de *c* dans *src*,
 retourne `NULL` si absence

c est bien un caractère passé sous forme d'un entier à la fonction `strrchr()`

```
char *strtok(char *src, char *d)
```

retourne un "token" de *src*,
 le "token" est délimité par *d*

```
size_t strspn(char *src, char *spanset)
```

recherche les caractères de *spanset* dans *src*,
 retourne le nombre de caractères trouvés

```
size_t strcspn(char *src, char *spanset)
```

recherche les *n* caractères n'appartenant pas à *spanset* dans *src*,
 retourne le nombre de caractères trouvés

```
char *strpbrk(char *src, char *spanset)
```

recherche les caractères de *spanset* dans *src*,
 retourne le pointeur sur la première occurrence d'un caractère de *spanset*

```
char *strstr(char *src, char *pat)
```

retourne un pointeur sur le début de *pat* dans *src*

Exemple :

```

char nom[20] ;
char prenom[30] ;
char adresse[100] ;

printf("nom : ") ;
scanf("%s", nom) ;
printf("prenom : ") ;
scanf("%s", prenom) ;

strcpy(adresse, nom) ;
strcat(adresse, ".") ;
strcat(adresse, prenom) ;
strcat(adresse, "depinfo.uhp-nancy.fr) ;

```

La variable `adresse` devrait ainsi contenir l'adresse électronique de la personne dont on vient de préciser le nom et le prénom.

Les fonctions `sscanf()` et `sprintf()`

```

sprintf(chaine, format, listeVariables)
sscanf(chaine, format, listeAdressesVariables)

```

► travaillent sur une chaîne de caractères et non sur un fichier standard (`stdin` ou `stdout`)

exemples :

```

char chaine[] = "29 31034" ;
int x, y ;

sscanf(chaine, "%d", &x) ;
sscanf(chaine, "%d", &y) ;

```

Les fonctions `sscanf()` et `sprintf()`

```

sprintf(chaine, format, listeVariables)
sscanf(chaine, format, listeAdressesVariables)

```

► travaillent sur une chaîne de caractères et non sur un fichier standard (`stdin` ou `stdout`)

exemples :

```

strcpy(chaine, "nombre de pays = 29") ;
sscanf(chaine, "nombre de pays = %d", &nbP) ;

strcpy(chaine, "population = 31034") ;
sscanf(chaine, "population = %d", &pop) ;

```

Les fonctions `sscanf()` et `sprintf()`

Contenu du fichier :

```

nombre de pays = 29
population = 31034
superficie = 25

```

Fgets(chaine, 80, fichier);

Introduction au langage C
Les structures de données (tableaux)

Les fonctions sscanf() et sprintf() (suite)

chaîne nombre de pays = 29

Idee : se placer après le caractère = pour récupérer la valeur numérique

*char * strchr (char *src, int c) :*
retourne un pointeur sur la dernière occurrence de c dans src, retourne NULL si absence

sscanf (src, "%d", &mbp) :

138 / 237

Les fonctions sscanf() et sprintf()

Il existe des fonctions voisines des fonctions *scanf()* et *printf()* (pages 63 et 66) nommées *sscanf()* et *sprintf()* qui réalisent des conversions similaires, mais qui travaillent sur une chaîne de caractères et non sur un fichier standard (*stdin* ou *stdout*).

- *syntaxe* : `sprintf(chaîne, format, listeVariables)`

sprintf() met en format les arguments contenus dans la *liste* en fonction de la chaîne de *format* indiquée, comme la fonction *printf()*, mais elle place le résultat dans *chaîne* au lieu de le mettre directement sur la sortie standard. Il est souhaitable que la chaîne de caractères, définie comme un tableau de caractères, soit assez grande pour contenir le résultat souhaité.

Exemple :

```
char nom[100] ;
int x ;

x = 54 ;
sprintf(nom, "image%d", x) ;
```

la fonction *sprintf()* place dans la variable *nom* la chaîne de caractères "image54" terminée naturellement par '\0'.

- *syntaxe* : `sscanf(chaîne, format, listeAdressesVariables)`
- sscanf()* réalise la transformation inverse : elle analyse la *chaîne* selon le *format* indiqué et place les résultats dans la *liste* des variables dont on a précisé les adresses.

Exemple :

```
char nom[100] ;
int x ;

strcpy(nom, "image1034");
...
sscanf(nom, "image%d", &x) ;
```

la fonction *sscanf()* place la valeur 1034 (qui suit la constante "image") dans la variable *x*.

Exemple d'utilisation des fonctions de chaînes

```
void strnins(char *s, char *t, int i)
{
    /* insère la chaîne t dans la chaîne s à la position i */
    char string[MAX_SIZE], *temp = string ;

    if ((i < 0) && (i > strlen(s)))
    {
        fprintf(stderr, "la position (i) déborde\n", i) ;
        exit(1) ;
    }

    if (!strlen(s))
        strcpy(s, t) ;
    else
        if (strlen(t))
        {
            strncpy(temp, s, i) ;
            strcat(temp, t) ;
            strcat(temp, (s+i)) ;
            strcpy(s, temp) ;
        }
}
```

Introduction au langage C
└ Les structures de données (tableaux)

Quelques fonctions de classification de caractères

<code>isalpha(c)</code>	c est une lettre
<code>isupper(c)</code>	c est une lettre majuscule
<code>isdigit(c)</code>	c est un chiffre
<code>isnum(c)</code>	c est un caractère numérique

Exemple :

```
int fonction(char c)
{
    return((c == '_' || isalnum(c)) ? 1 : 0) ;
}
```

139 / 237

Les fonctions de classification de caractères

```
#include <ctype.h>
```

Il faut inclure le fichier `ctype.h` qui contient les déclarations des entêtes des fonctions de classification et de conversion de caractères.

`isalpha(c)` : retourne vrai (c'est-à-dire une valeur $\neq 0$) si `c` est une lettre

`isupper(c)` : retourne vrai si `c` est une lettre majuscule

`islower(c)` : retourne vrai si `c` est une lettre minuscule

`isdigit(c)` : retourne vrai si `c` est un chiffre

`isxdigit(c)` : retourne vrai si `c` est un chiffre hexadécimal [0-9], [A-F] ou [a-f]

`isalnum(c)` : retourne vrai si `c` est un caractère alphanumérique

`isspace(c)` : retourne vrai si `c` est un espace, une tabulation, RETURN, NEW LINE, FORMFEED, ou un caractère de tabulation verticale

`ispunct(c)` : retourne vrai si `c` est un caractère de ponctuation

`isprint(c)` : retourne vrai si `c` est un caractère imprimable

`iscntrl(c)` : retourne vrai si `c` est un caractère DEL, ou un caractère de contrôle

`isascii(c)` : retourne vrai si `c` est un caractère codé en ASCII (code < 0200)

`isgraph(c)` : retourne vrai si `c` est un caractère graphique visible

Introduction au langage C
└ Les structures de données (tableaux)

Les fonctions de conversion de caractères

<code>toascii(c)</code>	masque le caractère <code>c</code> pour forcer sa représentation dans l'intervalle [0 - 0x7F]
<code>toupper(c)</code>	transformation du caractère <code>c</code> en majuscule
<code>tolower(c)</code>	transformation du caractère <code>c</code> en minuscule
<code>atoi(str)</code>	transformation d'une chaîne <code>str</code> en un entier, les espaces en début de chaîne sont ignorés
<code>atof(str)</code>	transformation d'une chaîne <code>str</code> en un réel, les espaces en début de chaîne sont ignorés

Exemple :

```
if (isalnum(c[0]))
    x = atoi(c) ;
```

140 / 237

Pour utiliser les fonctions `atoi()` et `atof()`, il est nécessaire d'inclure le fichier `stdlib.h`.

Exemple :

```
#include <stdlib.h>

int main(void)
{
    char str[100] ;
    int x ;

    printf("entier : ") ; scanf("%s",str) ;

    x = atoi(str) ;

    printf("entier lu : %d\n", x) ;
    return(0) ;
}
```

Résultat d'exécution :

```
entier : 54.y
entier lu : 54
entier : er43
entier lu : 0
```

Voir aussi les tableaux de chaînes de caractères page 121.

Introduction au langage C
 Les structures de données (tableaux)

Les tableaux multidimensionnels

- ▶ en C, un tableau multidimensionnel est un tableau de tableaux
- ▶ la déclaration :

```
int mat[3][2];
```

définit un tableau bidimensionnel (3 lignes, 2 colonnes), de nom `mat`

C'est en fait un tableau comportant 3 éléments, chacun d'entre eux étant un tableau de 2 éléments :

1	2
3	4
5	6

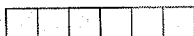
lequel est stocké en mémoire de la façon suivante :

1	2	3	4	5	6
---	---	---	---	---	---

141 / 237

Introduction au langage C
 Les structures de données (tableaux)

- ▶ accès à un élément par : `mat[i][j]`
- ▶ la déclaration : `int mat[3][2];`
- ▶ peut se traduire par :

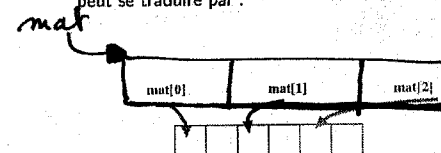


- ▶ réservation d'une zone mémoire de (3 * 2 * taille d'un entier) octets consécutifs

142 / 237

Introduction au langage C
 Les structures de données (tableaux)

- ▶ accès à un élément par : `mat[i][j]`
- ▶ la déclaration : `int mat[3][2];`
- ▶ peut se traduire par :



- ▶ réservation d'une zone mémoire de (3 * 2 * taille d'un entier) octets consécutifs
- ▶ association des noms `mat[0]`, `mat[1]`, ... aux adresses de début des tableaux monodimensionnels

142 / 237

Déclaration d'un tableau multidimensionnel

La ligne :

```
int tab[nbl][nbc];
```

déclare un tableau de `nbl` tableaux de `nbc` éléments (entiers) chacun.

L'initialisation d'un tableau multidimensionnel

La déclaration :

```
long t[5][4] = { {0, 1, 2, 3},
                 {4, 5, 6, 7},
                 {8, 9, 10, 11} };
```

est équivalente à la déclaration :

```
t[0][0] = 0;
t[0][1] = 1;
t[0][2] = 2;
...
```


Introduction au langage C
Les structures de données (tableaux)

liste

```
char liste[2][7];
```

liste : adresse de début d'implantation du tableau en mémoire

- ▶ liste[i][j] est un caractère

```
printf("%c", liste[0][0]);
```

```
scanf("%c", &liste[0][0]);
```
- ▶ liste[k] est une adresse en mémoire

```
printf("%s", liste[k]);
```

```
printf("%s", &liste[k]);
```
- ▶ liste est une adresse en mémoire, c'est l'adresse des tableaux
aucun sens

143 / 237

Introduction au langage C
Les structures de données (tableaux)

Tableaux statiques

```
#define N 10
```

```
int t[N][100];
```

- ▶ le nombre de dimensions et le nombre d'éléments par dimension du tableau sont connus par le compilateur à la lecture du texte source
- ▶ les tableaux statiques peuvent être des variables globales ou des variables locales

144 / 237

Introduction au langage C
Les structures de données (tableaux)

Tableaux dynamiques

```
void fonc(int a)
{ int t[a][a+2];
  ...
}
```

- ▶ le nombre de dimensions du tableau est connu par le compilateur à la lecture du texte source
- ▶ le nombre d'éléments par dimension du tableau est connu à l'exécution
- ▶ les tableaux dynamiques ne peuvent être que des variables locales

145 / 237

Introduction au langage C
Les structures de données (tableaux)

Tableaux flexibles

- ▶ pas de tableaux flexibles (dont les bornes varient dans les dimensions après la déclaration du tableau)

Remarque :

▲ en aucun cas, le nombre de dimension peut varier !!

146 / 237

Seuls les **tableaux statiques** dont la taille est connue à la compilation et les **tableaux dynamiques** dont la taille est connue à l'exécution sont autorisés en langage C.

Les **tableaux flexibles** dont la taille varie lors de l'exécution du programme ne sont pas implantés en C (contrairement aux langages Algol60, Eiffel).

12. Adresses et pointeurs

Introduction au langage C
└─ Adresses et pointeurs

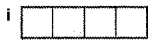
Les adresses

► la déclaration :

```
int i ;
```

se traduit par :

réserve de 4 octets consécutifs en mémoire pour stocker ultérieurement un entier et association du nom i à la zone allouée



147 / 237

Introduction au langage C
└─ Adresses et pointeurs

l'opérateur & permet d'accéder à l'adresse mémoire d'une variable :

- `int i` : association du nom `i` à la zone allouée
`&i` : adresse de début de la zone
- `int tab[N]` : association du nom `tab` à l'adresse du début de la zone allouée
`&tab[0]` : adresse du premier élément du tableau identique à `tab`
- `int mat [N] [M]` : association du nom `mat` à l'adresse de début du tableau des `N` tableaux monodimensionnels de `M` éléments chacun
`&mat [0] [0]` : adresse du premier élément du tableau identique à `mat [0]`
`mat` correspond à l'adresse d'implantation du tableau des tableaux

148 / 237

Introduction au langage C
└─ Adresses et pointeurs

```
int i ;
int tab[10] ;
char nom[50] ;
```

- `&i` représente l'adresse à partir de laquelle est rangé l'entier `i`
- `&tab[i]` représente l'adresse à partir de laquelle est rangé l'entier d'indice `i`

Lecture d'un entier :

```
scanf("%d", &i) ;
scanf("%d", &tab[i]) ;
```

Lecture d'un caractère :

```
scanf("%c", &nom[i]) ;
```

Lecture d'une chaîne de caractères :

```
scanf("%s", &nom[0]) ;
scanf("%s", nom) ;
```

149 / 237

Introduction au langage C
└─ Adresses et pointeurs

Les pointeurs

- 1 variable "pointeur"
 - identificateur
 - type de l'objet pointé
 - valeur
= 1 adresse (codée sur 4 octets, sur votre machine)
- déclaration :

```
type *p ;
```

150 / 237

Introduction au langage C
Adresses et pointeurs

attention : le type "pointeur" n'existe pas, il faut connaître le type de l'objet dont le pointeur va contenir l'adresse, c'est-à-dire le type de l'objet pointé.
La déclaration :

```
char *p ;
```

se traduit par :

- réservation de 4 octets consécutifs en mémoire pour y stocker ultérieurement une adresse
- association du nom `p` à la zone mémoire allouée

`p`

--	--	--	--

151 / 237

Un pointeur est une variable ou une constante dont la valeur est une adresse en mémoire. Il peut s'agir de l'adresse d'un objet ou d'une fonction.

Une variable "pointeur" conduit à la réservation d'une zone mémoire contenant une adresse et nécessite 4 octets, quel que soit le type de l'objet pointé.

- Si `i` est un entier et `p` contient l'adresse de `i`, alors `p` est déclaré comme un pointeur sur un entier :

```
int i, *p ;
```

- Si `n` est une structure et `p` contient l'adresse de `n`, alors `p` est déclaré comme un pointeur sur une structure :

```
struct noeud n, *p ;
```

- Si `t` est un tableau de caractères et `p` contient l'adresse de ce tableau, alors `p` est déclaré comme un pointeur sur un caractère :

```
char t[N], *p ;
```

La notation `void *` existe, elle est utilisée par les fonctions, voir page 141.

Le type de l'objet pointé est utilisé lors des additions et des soustractions sur les variables "pointeurs" (page 118).

Introduction au langage C
Adresses et pointeurs

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

`p` est un pointeur
attention :
type `*p` ; déclare seulement une variable de type "pointeur sur type"

Pour que ce pointeur contienne l'adresse mémoire d'une autre variable, il faut que cette variable existe et que le lien entre les deux variables ait été créé.

`p = &i ;`
Pour accéder à la variable `i` on a 2 façons
`*i`
`*p`

152 / 237

Introduction au langage C
Adresses et pointeurs

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

`q`

--

`p`

--

`i`

--

► déclaration de `q` : `int **q ;`
► accès à `i` `*p **q`
Dans ce cas, il est plus lisible de définir un nouveau type.

153 / 237

Petite remarque :

L'affectation :

```
y = x/*p ;
```

considère `/*` comme le début d'un commentaire...

Il est préférable d'écrire :

```
y = x/( *p ) ; ou y = x/ *p ;
```

Introduction au langage C
Adresses et pointeurs

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

p = (int *) i
p = 35 ;
p = 0 ;
p = NULL ;
```

→ impossible
→ aucun sens.

un pointeur est une adresse
la seule constante que l'on peut affecter à un pointeur est 0 (NULL)
#define NULL 0 dans `stdio.h`
La constante NULL désigne le pointeur nil.
On ne s'intéresse jamais à la valeur de l'adresse mais au lien ainsi créé.

154 / 237

La définition :

```
#define NULL 0
```

se trouve dans les fichiers `stdio.h`, `stdlib.h`, `string.h` ...

Si `p` est un pointeur NULL, il ne faut pas tenter d'utiliser le contenu de `p`.

Par exemple, l'instruction `printf("%s", p)` ; est fausse.

Introduction au langage C
Adresses et pointeurs

Seules l'addition et la soustraction sont autorisées sur les pointeurs :

```
p = p + 1 ;
p = p + 256 ;

p = p - 4 ;
p = t + N ;
```

La valeur additionnelle est convertie en multiple de la taille de l'objet pointé.

```
int i, j, *p ;
p = &i ;
p = p + 1 ; (ou p++)
```

155 / 237

Introduction au langage C
Adresses et pointeurs

```
short tab[5], *q ;
q = &tab[0] ;
q++ ;
```

156 / 237

```
char *p ;
p++ ;
```

équivalent à calculer $adresse = adresse + 1 \text{ octet}$

```
float *p ;
p++ ;
```

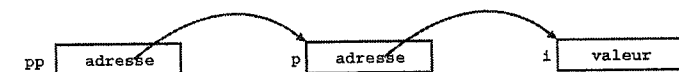
équivalent à calculer $adresse = adresse + 4 \text{ octets}$
si un réel de type float est représenté sur 4 octets

```
p += 4 ;
```

équivalent à calculer $adresse = adresse + 2 * 4 \text{ octets}$

Les calculs sont similaires si `p` pointe sur une structure, la quantité qui est ajoutée est la taille de la structure en octets, ceci est transparent à l'utilisateur.

Tout pointeur a une adresse qui peut être, elle aussi, contenue dans une autre variable de type pointeur :



```
int x, *p, **pp ;
```

Dans ce cas, il est préférable de passer par des définitions de type pour ne pas alourdir l'écriture en ajoutant des * :

```
typedef int* P_ENTIER ;
```

```
int x ;
P_ENTIER p, *pp ;
```

Introduction au langage C
└ Adresses et pointeurs

Les tableaux monodimensionnels et les pointeurs

```
int t[100], *p;
```

mise à 0 des éléments d'un tableau :

```
for (i = 0 ; i < 100 ; i++)
    t[i] = 0;
```

```
for (p = &t[0] ; p < &t[100] ; p++)
    *p = 0;
```

```
for (p = t ; p < t + 100 ; p++)
    *p = 0;
```

*for (p = t ; p < t + 100 ; *(p++) = 0*

157 / 237

Exemples de fonction calculant la longueur d'une chaîne de caractères

Version 1 :	<pre>unsigned int strlen(char s[]) { unsigned int n ; for (n = 0 ; s[n] != 0 ; n++) ; return(n) ; }</pre>
Version 2 :	<pre>unsigned int strlen(char *s) { unsigned int n ; for (n = 0 ; *s != 0 ; s++) n++ ; return(n) ; }</pre>
Version 3 :	<pre>unsigned int strlen(char *s) { unsigned int n = 0 ; while (*s++) n++ ; return(n) ; }</pre>

Version 4 :

```
unsigned int strlen(char *s)
{
    char *p = s ;

    while (*p++) ;
    return((unsigned int)(p-s-1)) ;
}
```

Introduction au langage C
└ Adresses et pointeurs

Les tableaux multidimensionnels et les pointeurs

```
int t[NBLIGS][NBCOLS] ;
```

L'affectation :

```
i = t[2][4] ;
```

est équivalente aux trois écritures suivantes :

```
i = *(t[2] + 4) ;
i = *(*t + 2) + 4) ;
i = *(t + 2*NBCOLS + 4) ;
```

159 / 237

Introduction au langage C
└ Adresses et pointeurs

Les tableaux multidimensionnels et les pointeurs (suite)

```
for (i = 0 ; i < nbLignes ; i++)
    for (j = 0 ; j < nbCols ; j++)
        accès à tab[i][j]
```

► le compilateur traduit chaque accès à `tab[i]` par :

```
*tab + i * nbCols + j)
```

```
for (p = tab[0] ; p < tab[0]+nbLignes*nbCols ; p++)
    accès à *p
```

► accès direct, sans calcul supplémentaire, au contenu de la case du tableau et ce, quelle que soit la nature des éléments du tableau

`p++` ↔ `p += (taille en octet des éléments de tab)`

159 / 237

Autres écritures équivalentes :

<code>&t[0][0]</code>	adresse du premier élément du tableau <code>t</code> et non pas <code>t !</code>
<code>t</code>	tableau <code>t</code>
<code>&t[0]</code>	adresse du premier élément du tableau <code>t[0]</code>
<code>&t[i][0]</code>	adresse du premier élément du tableau <code>t[i]</code>
<code>t[i][j]</code>	est traduit par le compilateur en <code>*(t + i*NBCOLS + j)</code>

```
char nom [10][50] ;
```

```
scanf("%s", &nom[6][0]) ; (ou) scanf("%s", nom[6]) ;
```

La déclaration :

```
int t[10][20] ;
```

peut se traduire par :

- réservation de 10 tableaux de (20 * taille(int)) octets chacun
- association des noms `t[0]`, `t[1]`, ... aux adresses de début des tableaux monodimensionnels
- association du nom `t` à l'adresse de début du tableau des 10 tableaux monodimensionnels

Le compilateur traduit les indices des cases en pointeurs.

```
int t[NBLIGS][NBCOLS] ;
```

`t[i][j]` se traduit en `*(t + i*NBCOLS + j)` (calculs peu rapides, nécessitant 2 additions et une coûteuse multiplication)

- Pour un tableau à `n` dimensions, il faut toujours connaître au moins (`n-1`) dimensions (voir passage des tableaux en paramètres de fonction page 135).
- Soit la déclaration :

```
char chaine[nb] ;
```

Les deux instructions suivantes sont équivalentes :

```
printf("%s\n", chaine) ;
printf("%s\n", &chaine[0]) ;
```

- Pour avancer de `NBCOLS` en `NBCOLS` dans le tableau `t` :

```
int *pointeur[NBCOLS] ;
for (pointeur = t ; pointeur < &t[NBLIGS] ; pointeur++)
    ...
```

L'incréméntation `pointeur++` permet "d'avancer" de `NBCOLS` entiers.

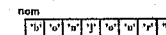
Introduction au langage C
Adresses et pointeurs

Les déclarations/initialisations de chaînes

Soient les 3 déclarations :

- (a) `char nom[] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'} ;`
 (b) `char nom[] = "bonjour" ;`
 (c) `char *nom = "bonjour" ;`

- (a) et (b) sont identiques et déclarent et initialisent un tableau de 8 caractères :



Le contenu de ce tableau est modifiable à tout instant.

160 / 237

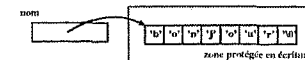
Introduction au langage C
Adresses et pointeurs

Les déclarations/initialisations de chaînes

Soient les 3 déclarations :

- (a) `char nom[] = {'b', 'o', 'n', 'j', 'o', 'u', 'r', '\0'} ;`
 (b) `char nom[] = "bonjour" ;`
 (c) `char *nom = "bonjour" ;`

- (c) déclare un pointeur (zone mémoire de 4 octets) de caractères, initialisé avec l'adresse de la constante "bonjour". Dans ce cas, seul le contenu du pointeur (une adresse) peut être modifié, la chaîne de caractères est ici une constante, rangée dans une partie de la mémoire protégée en écriture.



160 / 237

Les tableaux de pointeurs

L'intérêt est d'utiliser un tableau bidimensionnel pour stocker des informations sans perdre de place.

Introduction au langage C
Adresses et pointeurs

```
static char *jour[] = { "lundi",
                       "mardi",
                       ...,
                       "dimanche" } ;
```

- réservation d'un tableau de 7 pointeurs,
- chaque élément de ce tableau est initialisé avec l'adresse de la constante chaîne de caractères définie,
- association du nom `jour` au début de la zone mémoire du tableau.

161 / 237

Introduction au langage C
Adresses et pointeurs

```
static char *jour[] = { "lundi",
                       "mardi",
                       ...,
                       "dimanche" } ;
```

► de quel type est `jour[0]`? `char*`
`printf("%s", jour[0]);` → lundi
 ► de quel type est `*jour[0]`? `char`
`printf("%c", *jour[0]);` → l
 de quel type est `jour[0]+2`? `char*`
`printf("%s", jour[0]+2)` → mardi

161 / 237

Pour faire écrire mercredi:
`jour[2]`
`*(jour+2)`

Introduction au langage C
Adresses et pointeurs

Pour faire écrire 'c', la 4^{ème} lettre de "mercredi" :

- `jour` : adresse de début du tableau
- `(jour+2)` : adresse de `jour[2]`
- `*(jour+2)` : adresse de la chaîne "mercredi"
- `*(jour+2)+3` : adresse de la chaîne "mercredi"
- `*(*(jour+2)+3)` : caractère 'c'

```
printf("%c", (*(jour+2)+3));
```

163 / 237

Le programme :

```
char **p ;
p = jour ;
printf("%s\n", *p) ;
p++ ;
printf("%s\n", *p) ;
```

affiche :

```
lundi
mardi
```

Introduction au langage C
Adresses et pointeurs

Tableaux, pointeurs, structures...

```
struct noeud { int val ;
               struct noeud *suiv ;
               } ;
```

164 / 237

Introduction au langage C
Adresses et pointeurs

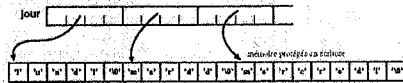
```
static char *jour[] = { "lundi",
                       "mardi",
                       ...,
                       "dimanche" };
```

- réservation d'un tableau de 7 pointeurs,
- chaque élément de ce tableau est initialisé avec l'adresse de la constante chaîne de caractères définie,
- association du nom `jour` au début de la zone mémoire du tableau.

161 / 237

Introduction au langage C
Adresses et pointeurs

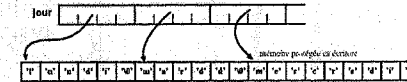
```
static char *jour[] = { "lundi",
                       "mardi",
                       ...,
                       "dimanche" };
```



- de quel type est `jour[0]`? `char*`
`printf("%s", jour[0]);` → lundi
- de quel type est `*jour[0]`? `char`
`printf("%c", *jour[0]);` → l
- de quel type est `jour[0]+2`? `char*`
`printf("%s", jour[0]+2);` → ndi

Pour faire écrire mercredi:
`jour[2]`
`*(jour+2)`

Introduction au langage C
Adresses et pointeurs



Pour faire écrire 'c', la 4^{ème} lettre de "mercredi" :

- `jour` : adresse de début du tableau

- `(jour + 2)` : adresse de `jour[2]`
- `*(jour + 2)` : adresse de la chaîne "mercredi"
- `*(jour + 2) + 3` : adresse de la chaîne "mercredi"
- `*(*(jour + 2) + 3)` : caractère 'c'

```
printf("%c", (*(jour+2)+3));
```

163 / 237

Le programme :

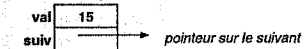
```
char **p ;
p = jour ;
printf("%s\n", *p) ;
p++ ;
printf("%s\n", *p) ;
```

affiche :

lundi
mardi

Introduction au langage C
Adresses et pointeurs

Tableaux, pointeurs, structures...

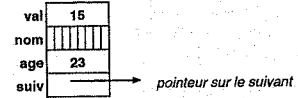


```
struct noeud { int val ;
               struct noeud *suiv ;
             } ;
```

164 / 237

Introduction au langage C
Adresses et pointeurs

Tableaux, pointeurs, structures...

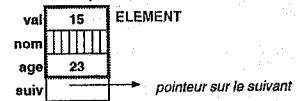


```
struct noeud { int val ;
               char nom[10] ;
               int age ;
               struct noeud *suiv ;
             } ;
```

164 / 237

Introduction au langage C
Adresses et pointeurs

Tableaux, pointeurs, structures...



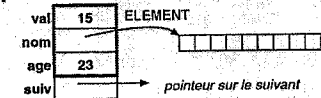
```
typedef struct { int val ;
                 char nom[10] ;
                 int age ;
                 } ELEMENT ;

struct noeud { ELEMENT elt ;
               struct noeud *suiv ;
             } ;
```

164 / 237

Introduction au langage C
Adresses et pointeurs

Tableaux, pointeurs, structures...



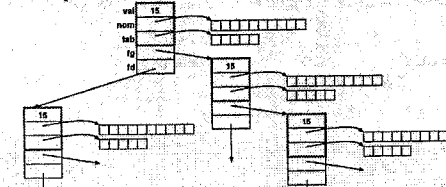
```
typedef struct { int val ;
                 char *nom ;
                 int age ;
                 } ELEMENT ;

struct noeud { ELEMENT elt ;
               struct noeud *suiv ;
             } ;
```

164 / 237

Introduction au langage C
Adresses et pointeurs

Tableaux, pointeurs, structures...



```
typedef struct { int val ;
                 char *nom ;
                 int *tab ;
                 } ELEMENT ;

struct noeud { ELEMENT elt ;
               struct noeud *suiv ;
             } ;
```

**18, *fd;*

164 / 237