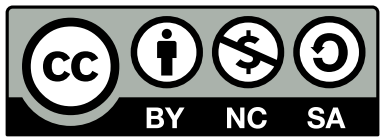
INF1135
Programmation procédurale
Chapitre 7 — Tableaux et chaînes

Alexandre Blondin Massé

Université du Québec à Montréal

v263



Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Rappel : tableaux et pointeurs

```
int t[5] = {3, 1, 4, 1, 5};
t[2] = 9;                          // 3e element
```

- Éléments de même type, **contigus** en mémoire
- Indices de `0` à `taille - 1`
- `t` vaut `&t[0]` ; `t[i]` est un raccourci pour `*(t + i)`
- Une fonction reçoit l'**adresse** du premier élément, mais pas la taille
- Aucune vérification des indices : un débordement est un comportement **indéfini**

Dans ce chapitre

On approfondit : initialisation, taille, algorithmes classiques, puis les **chaînes de caractères**, qui sont des tableaux particuliers, et enfin les tableaux à plusieurs dimensions.

Motivation

Calculer la moyenne d'un groupe, puis compter les notes au-dessus de la moyenne :

```
int note1 = 72, note2 = 85, note3 = 64,
    note4 = 91, note5 = 58, note6 = 77;
double moyenne = (note1 + note2 + note3
                  + note4 + note5 + note6) / 6.0;
int au_dessus = (note1 > moyenne) + (note2 > moyenne) + ...
```

- Impossible d'écrire une **boucle** sur `note1`, ..., `note6`
- Et pour un groupe de 120 étudiants ?
- Il faut **conserver** toutes les valeurs : on les parcourt deux fois

Et le texte ?

Un nom, un message, une ligne saisie au clavier : une **suite de caractères**. En C, c'est aussi un tableau.

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Exemple : moyenne d'un groupe

Fichier `average.c` :

```c
#include <stdio.h>

#define N 6

int main(void) {
  // Notes d'un groupe a un examen (sur 100)
  int notes[N] = {72, 85, 64, 91, 58, 77};

  int somme = 0;
  for (int i = 0; i < N; i++) {
    somme += notes[i];
  }
  double moyenne = (double) somme / N;
  printf("Moyenne : %.2f\n", moyenne);

  // Deuxieme parcours
  int au_dessus = 0;
  for (int i = 0; i < N; i++) {
    if (notes[i] > moyenne) au_dessus++;
  }
  printf("Au-dessus : %d/%d\n", au_dessus, N);
  return 0;
}
```

```
$ ./average
Moyenne : 74.50
Au-dessus : 3/6
```

- `N` : taille du tableau, définie **une seule fois**
- Les boucles s'écrivent avec `N` : passer à 120 notes ne change qu'une ligne
- Les notes restent disponibles pour le **deuxième** parcours

Exemple : initialisations

Fichier `init.c` :

```c
void afficher(const int t[], int n) {
  printf("{");
  for (int i = 0; i < n; i++) {
    printf(i == 0 ? "%d" : ", %d", t[i]);
  }
  printf("}\n");
}

int main(void) {
  int a[5] = {1, 2, 3, 4, 5};
  int b[5] = {1, 2};
  int c[5] = {0};
  int d[] = {7, 8, 9};
  int e[5] = {[1] = 4, [3] = 6};
  printf("a = "); afficher(a, 5);
  printf("b = "); afficher(b, 5);
  printf("c = "); afficher(c, 5);
  printf("d = "); afficher(d, 3);
  printf("e = "); afficher(e, 5);
  return 0;
}
```

```
$ ./init
a = {1, 2, 3, 4, 5}
b = {1, 2, 0, 0, 0}
c = {0, 0, 0, 0, 0}
d = {7, 8, 9}
e = {0, 4, 0, 6, 0}
```

- `b` : les éléments non mentionnés valent 0
- `d` : taille **déduite** des valeurs
- `e` : on désigne les indices à initialiser

Déclaration et initialisation

- Déclaration : `type nom[taille];`
- Taille : une **constante** entière, de préférence nommée (`#define N 6`) plutôt qu'un nombre « magique »

Initialisation	**Contenu**
`int a[5] = {1, 2, 3, 4, 5};`	complète
`int b[5] = {1, 2};`	`{1, 2, 0, 0, 0}`
`int c[5] = {0};`	que des zéros
`int d[] = {7, 8, 9};`	taille 3
`int e[5] = {[1] = 4, [3] = 6};`	`{0, 4, 0, 6, 0}`

Attention

- Sans initialisation, un tableau **local** contient des valeurs **indéterminées** (comme toute variable locale)
- Les accolades ne servent qu'à l'**initialisation**, dans la déclaration : plus loin, l'**affectation** `a = {1, 2};` est une erreur

Taille d'un tableau avec `sizeof`

Fichier `arraysize.c` :

```c
#include <stdio.h>

void taille(int t[]) {
  printf("taille : %zu\n", sizeof t);
}

int main(void) {
  int t[] = {3, 1, 4, 1, 5, 9, 2, 6};
  size_t n = sizeof t / sizeof t[0];
  printf("sizeof t    = %zu\n", sizeof t);
  printf("sizeof t[0] = %zu\n", sizeof t[0]);
  printf("n           = %zu\n", n);
  taille(t);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o arraysize
      arraysize.c
arraysize.c:4:35: warning: 'sizeof' on array
      function parameter 't' will return size of
      'int *' [-Wsizeof-array-argument]
$ ./arraysize
sizeof t    = 32
sizeof t[0] = 4
n           = 8
taille : 8
```

- `sizeof t` : taille du tableau **entier**, en octets
- `sizeof t / sizeof t[0]` : nombre d'**éléments**
- Dans `taille`, `t` n'est qu'un **pointeur** : `sizeof t` donne 8. Le compilateur nous prévient

Copier et comparer des tableaux

Fichier copy.c :

```c
int main(void) {
  int a[N] = {1, 2, 3, 4};
  int b[N];

  // Copie element par element
  for (int i = 0; i < N; i++) {
    b[i] = a[i];
  }

  // Comparaison element par element
  bool egaux = true;
  for (int i = 0; i < N && egaux; i++) {
    if (a[i] != b[i]) egaux = false;
  }

  printf("a == b : %d\n", a == b);
  printf("egaux  : %d\n", egaux);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o copy copy.c
copy.c:20:29: warning: comparison between two arrays
      [-Warray-compare]
copy.c:20:29: note: use '&a[0] == &b[0]' to compare
     the addresses
$ ./copy
a == b : 0
egaux  : 1
```

Et si on écrit `b = a;` ?

- Pas d'**affectation** entre tableaux : on copie élément par élément
- `a == b` compare les **adresses**, et non le contenu

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Exemple : valeur maximale

Fichier `maximum.c` :

```c
/**
 * Retourne l'indice du plus grand element d'un tableau.
 *
 * @param t  Le tableau (non vide)
 * @param n  Le nombre d'elements (n >= 1)
 * @return   L'indice du (premier) maximum
 */
int indice_max(const int t[], int n) {
  int imax = 0;
  for (int i = 1; i < n; i++) {
    if (t[i] > t[imax]) imax = i;
  }
  return imax;
}

int main(void) {
  int t[] = {-4, -2, -7, -2, -9};
  int i = indice_max(t, 5);
  printf("maximum : t[%d] = %d\n", i, t[i]);
  return 0;
}
```

```
$ ./maximum
maximum : t[1] = -2
```

- On part du **premier** élément, puis on compare avec les suivants
- Retourner l'**indice** donne à la fois la position et la valeur
- `>` plutôt que `>=` : on garde le **premier** maximum
- **Précondition** : `n >= 1`, documentée

Exemple : tableau trié ?

Fichier `sorted.c` :

```c
/**
 * Indique si un tableau est trie en ordre croissant.
 *
 * @param t  Le tableau
 * @param n  Le nombre d'elements
 * @return   true si t est trie, false sinon
 */
bool est_trie(const int t[], int n) {
  for (int i = 0; i + 1 < n; i++) {
    if (t[i] > t[i + 1]) return false;
  }
  return true;
}

int main(void) {
  int a[] = {1, 3, 3, 8};
  int b[] = {1, 5, 2, 8};
  int c[] = {42};
  printf("a : %d\n", est_trie(a, 4));
  printf("b : %d\n", est_trie(b, 4));
  printf("c : %d\n", est_trie(c, 1));
  printf("vide : %d\n", est_trie(a, 0));
  return 0;
}
```

```
$ ./sorted
a : 1
b : 0
c : 1
vide : 1
```

- On compare chaque élément à son **voisin** de droite
- `i + 1 < n` : `t[i + 1]` reste dans le tableau
- Dès qu'une paire est mal placée, on sait la réponse : `return false`
- Un tableau vide ou d'un seul élément est trié

Exemple : recherche linéaire

Fichier `search.c` :

```c
/**
 * Cherche une valeur dans un tableau.
 *
 * @param t  Le tableau
 * @param n  Le nombre d'elements
 * @param x  La valeur cherchee
 * @return   La premiere position de x, ou -1
 */
int chercher(const int t[], int n, int x) {
  for (int i = 0; i < n; i++) {
    if (t[i] == x) return i;
  }
  return -1;
}

int main(void) {
  int t[] = {4, 8, 15, 16, 23, 42};
  printf("15 : %d\n", chercher(t, 6, 15));
  printf("42 : %d\n", chercher(t, 6, 42));
  printf("7  : %d\n", chercher(t, 6, 7));
  return 0;
}
```

```
$ ./search
15 : 2
42 : 5
7  : -1
```

- On parcourt le tableau jusqu'à trouver `x`
- `-1` : valeur **impossible** pour un indice, elle signale l'échec
- Au pire, n comparaisons

Exemple : tri par sélection

Fichier `selection.c` (extrait) :

```
/**
 * Trie un tableau (tri par selection).
 *
 * @param t  Le tableau a trier
 * @param n  Le nombre d'elements
 */
void trier(int t[], int n) {
  for (int i = 0; i + 1 < n; i++) {
    // Indice du minimum de t[i..n-1]
    int imin = i;
    for (int j = i + 1; j < n; j++) {
      if (t[j] < t[imin]) imin = j;
    }
    echanger(&t[i], &t[imin]);
    // Equivalent: echanger(t + i, t + imin);
    afficher(t, n);
  }
}
```

```
$ ./selection
 29  10  14  37  13
 10  29  14  37  13
 10  13  14  37  29
 10  13  14  37  29
 10  13  14  29  37
```

- À chaque tour, on **sélectionne** le minimum de la partie non triée...
- ... et on l'échange avec le premier élément de cette partie
- `echanger` : le passage par adresse vu précédemment

Tri par sélection : trace

i	t[0]	t[1]	t[2]	t[3]	t[4]	imin	**Échange**
–	29	10	14	37	13	–	
0	**10**	29	14	37	13	1	t[0] et t[1]
1	**10**	**13**	14	37	29	4	t[1] et t[4]
2	**10**	**13**	**14**	37	29	2	aucun effet
3	**10**	**13**	**14**	**29**	**37**	4	t[3] et t[4]

- En gras : la partie **triée**, qui grandit d'un élément par tour
- Le dernier élément est forcément à sa place : `n - 1` tours suffisent
- Environ $n^2/2$ comparaisons : simple, mais lent pour de grands tableaux

Tableaux en paramètre

```
int indice_max(const int t[], int n);
int indice_max(const int *t, int n);   // identique
```

- En paramètre, `int t[]` et `int *t` sont **équivalents** : la fonction reçoit une adresse
- `t[]` rappelle au lecteur qu'on attend un tableau
- La taille doit toujours être **transmise**
- `const` : la fonction s'engage à ne pas modifier le tableau (`indice_max`, `est_trie`, `chercher`) ; sans `const`, elle peut le modifier (`trier`)

Bonnes pratiques

- Une fonction par algorithme : elle se teste et se réutilise facilement
- Documenter les **préconditions** (`n >= 1`) et la valeur retournée en cas d'échec (`-1`)

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Exemple : une chaîne en mémoire

Fichier `string.c` :

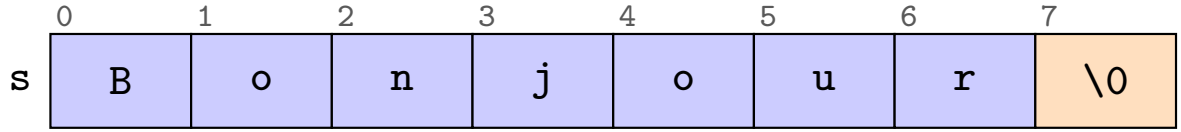
```c
#include <stdio.h>

int main(void) {
  char s[] = "Bonjour";
  printf("%s\n", s);
  for (int i = 0; i < (int) sizeof s; i++) {
    if (s[i] == '\0') {
      printf("s[%d] = '\\0' (%d)\n", i, s[i]);
    } else {
      printf("s[%d] = '%c'  (%d)\n", i, s[i], s[i]);
    }
  }
  printf("sizeof s = %zu\n", sizeof s);
  return 0;
}
```

```
$ ./string
Bonjour
s[0] = 'B'  (66)
s[1] = 'o'  (111)
s[2] = 'n'  (110)
s[3] = 'j'  (106)
s[4] = 'o'  (111)
s[5] = 'u'  (117)
s[6] = 'r'  (114)
s[7] = '\0' (0)
sizeof s = 8
```

- 7 lettres, mais `sizeof s` vaut **8**
- Le dernier élément est un caractère de code 0

Le caractère nul '\0'



- **Chaîne de caractères** : tableau de `char` terminé par le caractère nul `'\0'` (code 0)
- C'est le `'\0'`, et non la taille du tableau, qui marque la **fin** de la chaîne
- Le littéral `"Bonjour"` ajoute automatiquement le `'\0'` : `char s[] = "Bonjour";` équivaut à `char s[] = {'B', 'o', …, 'r', '\0'};`
- Une chaîne de n caractères occupe $n + 1$ octets

Ne pas confondre

`'\0'` (code 0), `'0'` (le chiffre, code 48) et `"0"` (une chaîne de 2 octets).

Taille du tableau et longueur de la chaîne

```
char a[] = "abc";      // 4 octets : a b c \0
char b[10] = "abc";    // 10 octets : a b c \0 \0 ... \0
char c[3] = "abc";     // 3 octets : a b c (pas de \0 !)
```

- Le tableau peut être **plus grand** que la chaîne : `b` peut contenir jusqu'à 9 caractères
- `c` est accepté sans avertissement, mais n'est **pas** une chaîne : `printf("%s", c)` lit au-delà du tableau

À retenir

Toujours prévoir **un octet de plus** pour le `'\0'`. Une chaîne de 20 caractères au plus se range dans un `char s[21]`.

Exemple : modifier une chaîne

Fichier `literal.c` :

```c
#include <stdio.h>

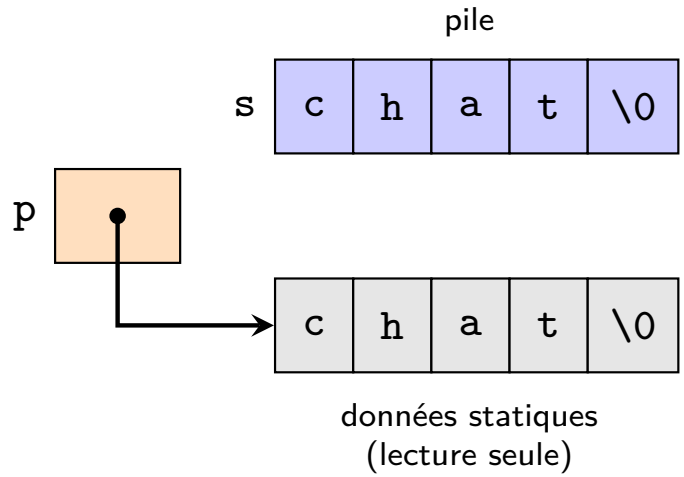
int main(void) {
  char s[] = "chat";  // tableau
  char *p = "chat";   // pointeur
  s[0] = 'p';
  printf("s = %s\n", s);
  p[0] = 'p';
  printf("p = %s\n", p);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o literal
      literal.c
$ ./literal
s = phat
Segmentation fault (core dumped)
```

- `s[0] = 'p'` : fonctionne
- `p[0] = 'p'` : plantage
- Aucun avertissement à la compilation !

Tableau ou pointeur vers un littéral

```
char s[] = "chat";
const char *p = "chat";
```



- `s` : tableau local sur la **pile**, où le littéral est **copié** ; la copie est modifiable
- `"chat"` : durée de vie **statique**, rangé avec les **données statiques** vues précédemment, dans une partie en **lecture seule** chargée depuis l'exécutable
- `p` : pointeur local qui contient l'**adresse** du littéral, sans copie
- Modifier un littéral : comportement **indéfini** ; ici, plantage (zone protégée)
- Toujours `const char *` pour un littéral : le compilateur refusera alors `p[0] = 'p'`

Exemple : parcourir une chaîne

Fichier count.c :

```c
/**
 * Indique si c est une voyelle minuscule.
 *
 * @param c  Le caractere
 * @return   true si c est une voyelle
 */
bool est_voyelle(char c) {
  switch (c) {
    case 'a': case 'e': case 'i':
    case 'o': case 'u': case 'y':
      return true;
    default:
      return false;
  }
}

/**
 * Retourne le nombre de voyelles d'une chaine.
 *
 * @param s  La chaine (non modifiee)
 * @return   Le nombre de voyelles de s
 */
int nb_voyelles(const char *s) {
  int n = 0;
  for (int i = 0; s[i] != '\0'; i++) {
    if (est_voyelle(s[i])) n++;
  }
  return n;
}
```

```c
/**
 * Convertit en majuscules les lettres
 * minuscules d'une chaine.
 *
 * @param s  La chaine, modifiee sur place
 */
void mettre_en_majuscules(char *s) {
  for (char *p = s; *p != '\0'; p++) {
    if (*p >= 'a' && *p <= 'z') {
      *p = *p - 'a' + 'A';
    }
  }
}

int main(void) {
  char s[] = "programmation procedurale";
  printf("%d voyelles\n", nb_voyelles(s));
  mettre_en_majuscules(s);
  printf("%s\n", s);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o count count.c
$ ./count
10 voyelles
PROGRAMMATION PROCEDURALE
```

Parcourir une chaîne

```
// Avec un indice
for (int i = 0; s[i] != '\0'; i++) { ... s[i] ... }
// Avec un pointeur
for (const char *p = s; *p != '\0'; p++) { ... *p ... }
```

- La condition porte sur le **caractère courant**, pas sur une taille
- `'\0'` vaut 0, donc faux : on voit souvent `s[i]` ou `*p` seul comme condition
- Les caractères sont des **entiers** : `'a' <= c && c <= 'z'`, `c - 'a' + 'A'`
- En paramètre : `const char *s` si la fonction ne modifie pas la chaîne, `char *s` sinon

Exemple : lecture avec `fgets`

Fichier `readline.c` :

```c
#include <stdio.h>

#define TAILLE 8

int main(void) {
  char nom[TAILLE];
  printf("Nom : ");
  if (fgets(nom, TAILLE, stdin) == NULL) return 1;

  // Retrait du '\n' final, s'il est present
  for (int i = 0; nom[i] != '\0'; i++) {
    if (nom[i] == '\n') {
      nom[i] = '\0';
      break;
    }
  }
  printf("Bonjour, [%s]!\n", nom);
  return 0;
}
```

```
$ ./readline
Nom : Ada
Bonjour, [Ada]!
$ ./readline
Nom : Margaret
Bonjour, [Margare]!
```

- `Ada` : lu en entier, `'\n'` compris, qu'on retire
- `Margaret` : **tronqué** à `TAILLE - 1` = 7 caractères

Lire une ligne avec `fgets`

`fgets(s, n, stdin)`, vu précédemment :

- Lit au plus `n - 1` caractères et ajoute toujours le `'\0'` : **aucun débordement** possible
- Conserve le `'\n'` final s'il y a la place : on le retire au besoin
- Retourne `NULL` en cas d'échec (fin de l'entrée)
- Si la ligne est trop longue, le reste demeure dans l'entrée et sera lu au prochain appel

Appel typique

```
char ligne[100];
if (fgets(ligne, sizeof ligne, stdin) != NULL) {
   ...
}
```

Lire un mot avec `scanf`

```
char mot[20];
scanf("%19s", mot);
```

- Pas de `&` : `mot` est déjà une adresse
- `%s` s'arrête au premier **espace** : lit un mot, pas une ligne
- Sans largeur (`"%s"`), aucune limite : risque de **débordement**
- Largeur = taille du tableau -1, pour le `'\0'`

Règle

Pour lire une ligne saisie par l'utilisateur, préférer `fgets`. Avec `scanf`, toujours indiquer une largeur.

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Exemple : manipuler un nom

Fichier `strings.c` :

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

int main(void) {
  char prenom[] = "Ada";
  char complet[30];

  strcpy(complet, prenom);
  strcat(complet, " Lovelace");
  printf("%s\n", complet);
  printf("strlen = %zu, sizeof = %zu\n",
         strlen(complet), sizeof complet);

  printf("%d\n", strcmp(prenom, "Ada"));
  printf("%d\n", strcmp(prenom, "Alan") < 0);
  printf("%s\n", strchr(complet, 'L'));
  printf("%s\n", strstr(complet, "lace"));
  return 0;
}
```

```
$ ./strings
Ada Lovelace
strlen = 12, sizeof = 30
0
1
Lovelace
lace
```

- `#include <string.h>`
- `strcpy` : copie ; `strcat` : ajoute à la fin
- `strlen` : 12 caractères, dans un tableau de 30 octets
- `strcmp` : 0 si égales
- `strchr`, `strstr` : recherche

Longueur et copie

`size_t strlen(const char *s)`

- Nombre de caractères **avant** le `'\0'` (format %zu)
- À ne pas confondre avec `sizeof`, la taille du **tableau**
- Parcourt toute la chaîne : éviter `for (i = 0; i < strlen(s); i++)`, qui la parcourt à chaque tour

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

- Copie `src`, `'\0'` compris, dans `dest`
- `dest` doit être **assez grand** : rien n'est vérifié
- Retourne `dest`

Copie avec limite

```
char d[4];
strncpy(d, "Bonjour", sizeof d);  // d : B o n j  (pas de \0 !)
d[sizeof d - 1] = '\0';           // d : B o n \0
```

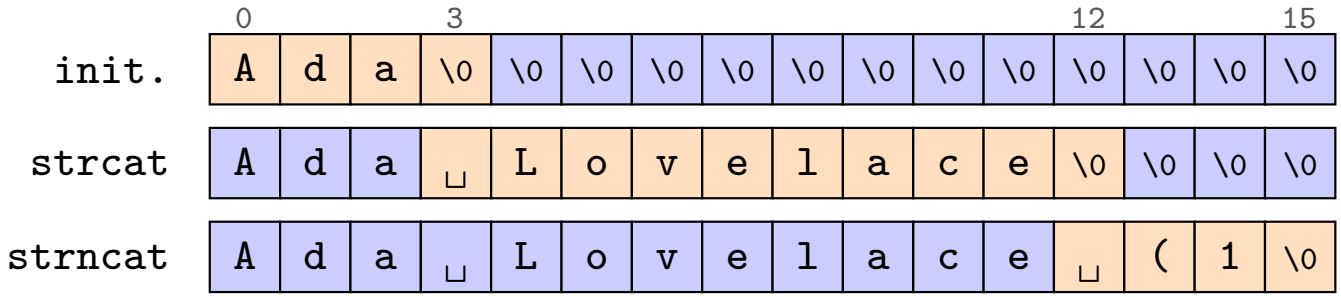
- `strncpy(dest, src, n)` : copie au plus `n` caractères
- Si `src` est trop longue, **aucun** `'\0'` n'est ajouté : `dest` n'est pas une chaîne
- Il faut donc ajouter le `'\0'` à la main

Alternative plus simple

`snprintf(d, sizeof d, "%s", src)` : copie au plus `sizeof d - 1` caractères et termine **toujours** la chaîne.

Concaténation

```
char s[16] = "Ada";
strcat(s, " Lovelace");
strncat(s, " (1815)", sizeof s - strlen(s) - 1);
```



en orange : octets écrits par l'instruction

- `strcat(dest, src)` : ajoute `src` à la fin de `dest`, en écrasant son `'\0'` ; `dest` doit avoir la place pour **les deux**
- `strncat(dest, src, n)` : ajoute au plus `n` caractères, puis **toujours** un `'\0'`. Ici, place restante : $16 - 12 - 1 = 3$
- Alternative : `snprintf(s, sizeof s, "%s %s", prenom, nom)` construit la chaîne en une fois, sans déborder

Exemple : comparer des chaînes

Fichier `compare.c` :

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

int main(void) {
  char reponse[] = "oui";
  if (reponse == "oui") {
    printf("== : egales\n");
  } else {
    printf("== : differentes\n");
  }
  if (strcmp(reponse, "oui") == 0) {
    printf("strcmp : egales\n");
  }
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o compare compare.c
compare.c:6:15: warning: comparison with string
      literal results in unspecified behavior [-
      Waddress]
$ ./compare
== : differentes
strcmp : egales
```

- Même contenu, et pourtant == répond « différentes »
- == compare deux **adresses** : celle du tableau `reponse` et celle du littéral

Comparaison

`int strcmp(const char *s1, const char *s2)` retourne une valeur **négative**, **nulle** ou **positive** selon que `s1` vient avant, est égale à ou vient après `s2`.

```
if (strcmp(reponse, "oui") == 0) ...    // egalite
if (strcmp(a, b) < 0) ...               // a avant b
if (strncmp(ligne, "GET ", 4) == 0) ...  // 4 premiers caracteres
```

- Ordre **lexicographique** selon les codes des caractères : `"Zoe"` vient avant `"ada"` ($\texttt{'Z'} = 90 < \texttt{'a'} = 97$)
- Seul le **signe** compte, pas la valeur exacte
- `strncmp(s1, s2, n)` : au plus `n` caractères

Piège

`if (strcmp(a, b))` est vrai quand les chaînes sont **différentes**.

Exemple : rechercher dans une chaîne

Fichier `find.c` :

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

int main(void) {
  const char *s = "capteur=thermo;unite=C";

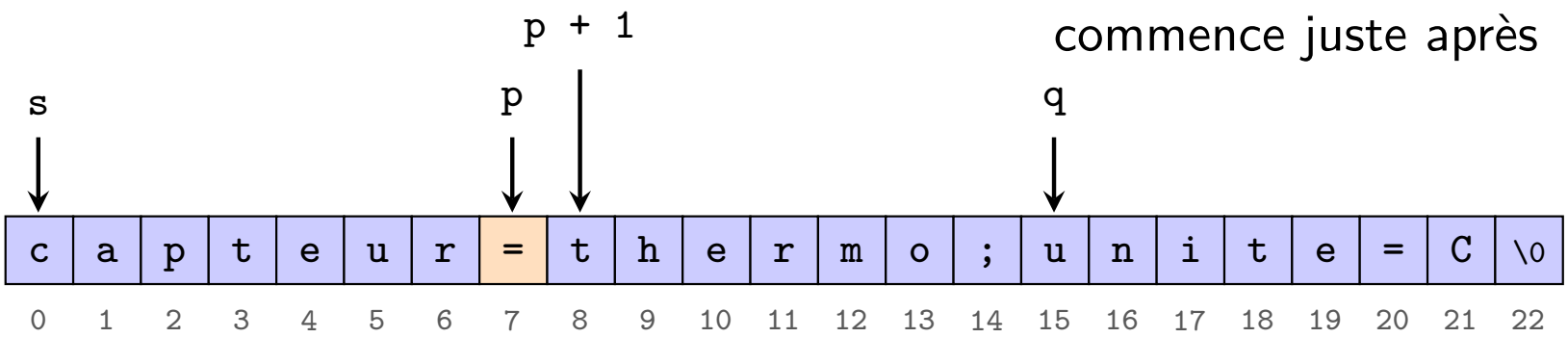
  const char *p = strchr(s, '=');
  printf("premier '=' : position %td\n", p - s);
  printf("valeur      : %s\n", p + 1);

  const char *q = strstr(s, "unite");
  printf("strstr      : %s\n", q);

  if (strchr(s, '#') == NULL) {
    printf("aucun '#'\n");
  }
  return 0;
}
```

```
$ ./find
premier '=' : position 7
valeur      : thermo;unite=C
strstr      : unite=C
aucun '#'
```

- `p` : adresse du `'='` **à l'intérieur** de `s`
- `p - s` : sa position (soustraction de pointeurs, vue précédemment)
- `p + 1` : la chaîne qui commence juste après



Recherche

- `strchr(s, c)` : adresse de la **première** occurrence du caractère `c` dans `s`
- `strstr(s, sous)` : adresse de la première occurrence de la chaîne `sous` dans `s`
- Si rien n'est trouvé : `NULL`

```
const char *p = strchr(s, '=');
if (p != NULL) {
  printf("position %td\n", p - s);
}
```

Attention

Toujours tester le résultat avant de l'utiliser : déréférencer `NULL` provoque un plantage.

Résumé de <string.h>

Fonction	Rôle	Retour
`strlen(s)`	longueur, sans le `'\0'`	`size_t`
`strcpy(d, s)`	copie `s` dans `d`	`d`
`strncpy(d, s, n)`	copie au plus `n` caractères	`d`
`strcat(d, s)`	ajoute `s` à la fin de `d`	`d`
`strncat(d, s, n)`	ajoute au plus `n` caractères	`d`
`strcmp(s1, s2)`	compare	< 0, 0 ou > 0
`strncmp(s1, s2, n)`	compare au plus `n` caractères	< 0, 0 ou > 0
`strchr(s, c)`	cherche un caractère	adresse ou `NULL`
`strstr(s, t)`	cherche une sous-chaîne	adresse ou `NULL`

Responsabilité du programmeur

- Les chaînes reçues doivent être terminées par `'\0'`
- La destination doit être assez grande : aucune de ces fonctions ne connaît la taille des tableaux

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Exemple : une matrice

Fichier `matrix.c` :

```c
#define LIGNES 3
#define COLONNES 4

int main(void) {
  int m[LIGNES][COLONNES] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
  };
  m[1][2] = 0;

  for (int i = 0; i < LIGNES; i++) {
    for (int j = 0; j < COLONNES; j++) {
      printf("%4d", m[i][j]);
    }
    printf("\n");
  }
  printf("sizeof m    = %zu\n", sizeof m);
  printf("sizeof m[0] = %zu\n", sizeof m[0]);
  return 0;
}
```

```
$ ./matrix
   1   2   3   4
   5   6   0   8
   9  10  11  12
sizeof m    = 48
sizeof m[0] = 16
```

- 3 lignes de 4 colonnes
- `m[1][2]` : ligne 1, colonne 2
- Deux boucles **imbriquées** : lignes, puis colonnes
- `sizeof m` $= 3 \times 4 \times 4$ octets ; `m[0]`, une ligne, occupe 4×4 octets

Déclaration et accès

```
int m[LIGNES][COLONNES];
double image[480][640];
char grille[3][3] = {
  {'X', 'O', ' '},
  {' ', 'X', 'O'},
  {' ', ' ', 'X'},
};
```

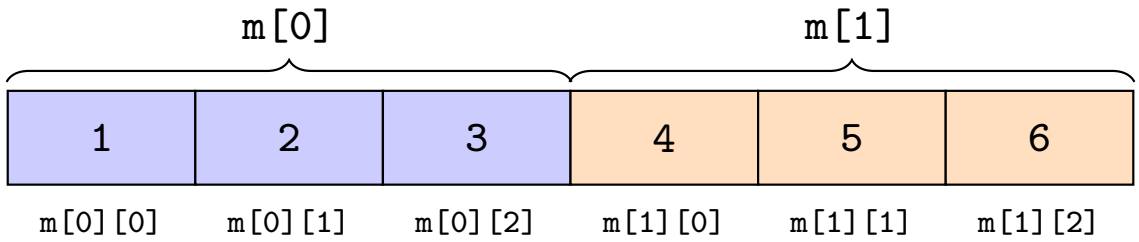
- `m[i][j]` : élément de la ligne `i`, colonne `j`
- Un tableau 2D est un **tableau de tableaux** : `m[i]` est la ligne `i`, un tableau de `COLONNES` éléments
- Initialisation ligne par ligne, avec des accolades imbriquées ; les valeurs manquantes valent 0
- Seule la **première** dimension peut être déduite : `int m[][3] = {{1, 2, 3}, {4, 5, 6}};`

Disposition en mémoire

Fichier `matrixmem.c` :

```c
int main(void) {
  int m[2][3] = {{1, 2, 3}, {4, 5, 6}};
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
      printf("&m[%d][%d] = %p\n",
             i, j, (void *) &m[i][j]);
    }
  }
  printf("m[1]      = %p\n", (void *) m[1]);
  return 0;
}
```

```
$ ./matrixmem
&m[0][0] = 0x7ffec3a12230
&m[0][1] = 0x7ffec3a12234
&m[0][2] = 0x7ffec3a12238
&m[1][0] = 0x7ffec3a1223c
&m[1][1] = 0x7ffec3a12240
&m[1][2] = 0x7ffec3a12244
m[1]     = 0x7ffec3a1223c
```



- Les lignes sont placées **l'une après l'autre**, sans espace
- `m[i][j]` se trouve `i * 3 + j` éléments après `m[0][0]` : le compilateur doit connaître le nombre de **colonnes**

Exemple : passer une matrice à une fonction

Fichier `matparam.c` :

```c
#define COLONNES 3

/**
 * Affiche la somme de chaque ligne d'une matrice.
 *
 * @param m       La matrice
 * @param lignes  Le nombre de lignes
 */
void sommes_lignes(const int m[][COLONNES], int lignes) {
  for (int i = 0; i < lignes; i++) {
    int s = 0;
    for (int j = 0; j < COLONNES; j++) {
      s += m[i][j];
    }
    printf("ligne %d : %d\n", i, s);
  }
}

int main(void) {
  int a[2][COLONNES] = {{1, 2, 3}, {4, 5, 6}};
  int b[4][COLONNES] = {
    {1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {1, 1, 1},
  };
  sommes_lignes(a, 2);
  sommes_lignes(b, 4);
  return 0;
}
```

```
$ ./matparam
ligne 0 : 6
ligne 1 : 15
ligne 0 : 1
ligne 1 : 1
ligne 2 : 1
ligne 3 : 3
```

- Le nombre de lignes est un **paramètre**
- Le nombre de colonnes est **fixé** dans le type : `COLONNES`
- a et b ont des nombres de lignes différents, mais le même nombre de colonnes

Matrices en paramètre

```
void f(int m[][COLONNES], int lignes);   // correct
void g(int m[][], int lignes, int colonnes);  // erreur
```

- Toutes les dimensions **sauf la première** doivent figurer dans le type du paramètre
- Raison : pour trouver `m[i][j]`, il faut savoir de combien d'éléments sauter pour passer d'une ligne à la suivante
- Comme en 1D, la fonction reçoit une adresse : elle peut modifier la matrice, sauf avec `const`

Limite

Une même fonction ne peut pas traiter des matrices de 3 colonnes **et** de 5 colonnes. Pour gagner en souplesse, on a besoin d'une nouvelle notion : les **pointeurs de pointeurs**.

Exemple : pointeur vers un pointeur

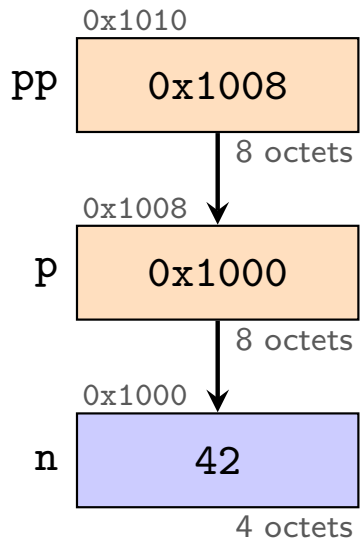
Fichier `pointer2.c` :

```c
int main(void) {
  int n = 42;
  int *p = &n;
  int **pp = &p;

  printf("&n  = %p\n", (void *) &n);
  printf("p   = %p\n", (void *) p);
  printf("&p  = %p\n", (void *) &p);
  printf("pp  = %p\n", (void *) pp);
  printf("*pp = %p\n", (void *) *pp);
  printf("**pp = %d\n", **pp);

  **pp = 7;
  printf("n   = %d\n", n);
  return 0;
}
```

```
$ ./pointer2
&n  = 0x7ffd2e19eb14
p   = 0x7ffd2e19eb14
&p  = 0x7ffd2e19eb18
pp  = 0x7ffd2e19eb18
*pp = 0x7ffd2e19eb14
**pp = 42
n   = 7
```



(adresses simplifiées)

Exemple : modifier un pointeur dans une fonction

Fichier `ptrparam.c` :

```c
/**
 * Fait pointer *pmax vers le plus grand.
 *
 * @param a     L'adresse du premier entier
 * @param b     L'adresse du deuxieme entier
 * @param pmax  L'adresse du pointeur a modifier
 */
void pointer_max(int *a, int *b, int **pmax) {
  *pmax = *a >= *b ? a : b;
}

int main(void) {
  int x = 3, y = 8;
  int *p = NULL;
  pointer_max(&x, &y, &p);
  printf("plus grand : %d\n", *p);
  return 0;
}
```

```
$ ./ptrparam
plus grand : 8
```

- La fonction doit modifier `p`, une variable de `main`
- Passage par adresse : on transmet `&p`, de type `int **`
- `*pmax = ...` : modifie `p`, qui pointe ensuite vers `y`

Pointeurs de pointeurs

Expression	Type	Désigne
`n`	`int`	l'entier
`p`, `*pp`	`int *`	l'adresse de `n`
`pp`	`int **`	l'adresse de `p`
`*p`, `**pp`	`int`	l'entier `n`

- `int **pp` : `pp` est un pointeur vers un `int *`
- Chaque `*` « suit une flèche » de plus
- Un pointeur est une variable comme les autres : il a lui-même une **adresse**

Deux usages

- Permettre à une fonction de **modifier un pointeur** de l'appelant
- Manipuler un **tableau de pointeurs**, par exemple un tableau de lignes

Exemple : lignes de longueurs différentes

Fichier `jagged.c` :

```
/**
 * Affiche un tableau de lignes de longueurs differentes.
 *
 * @param m         Les adresses des lignes
 * @param tailles   Le nombre d'elements de chaque ligne
 * @param lignes    Le nombre de lignes
 */
void afficher(int **m, const int tailles[], int lignes) {
  for (int i = 0; i < lignes; i++) {
    for (int j = 0; j < tailles[i]; j++) {
      printf("%3d", m[i][j]);
    }
    printf("\n");
  }
}

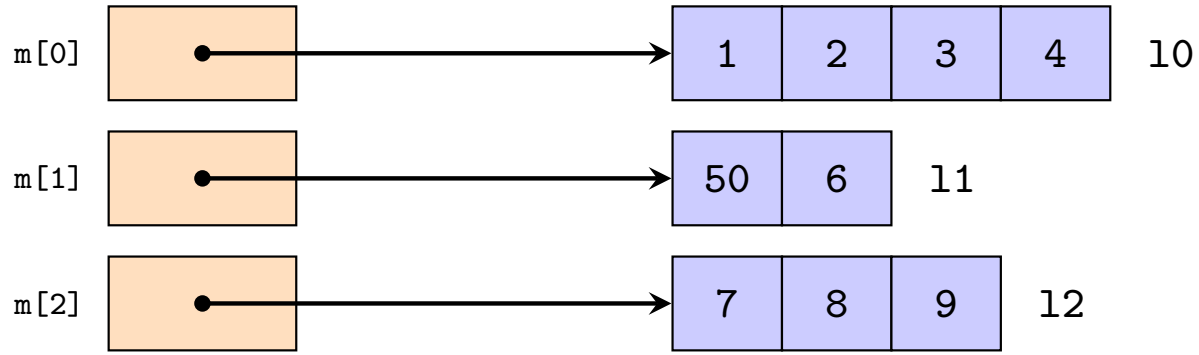
int main(void) {
  int l0[] = {1, 2, 3, 4};
  int l1[] = {5, 6};
  int l2[] = {7, 8, 9};
  int *m[3] = {l0, l1, l2};
  int tailles[3] = {4, 2, 3};

  m[1][0] = 50;  // modifie l1[0]
  afficher(m, tailles, 3);
  return 0;
}
```

```
$ ./jagged
  1   2   3   4
 50   6
  7   8   9
```

- `m` : tableau de 3 **pointeurs**, chacun vers une ligne
- Les lignes n'ont pas la même longueur
- `afficher` reçoit un `int **` : aucune dimension n'est fixée
- La syntaxe `m[i][j]` est **inchangée**

Tableau de pointeurs en mémoire



- `int *m[3]` : les crochets s'appliquent à `m` ; c'est un tableau de 3 `int *`
- Passé à une fonction, `m` devient l'adresse de son premier élément, soit un `int **`
- `m[i]` : adresse de la ligne `i` ; `m[i][j]` : élément `j` de cette ligne, soit `*(*(m + i) + j)`
- Les lignes peuvent être **n'importe où** en mémoire : il faut suivre une flèche de plus
- Les tailles des lignes ne sont pas connues : on les transmet (`tailles`)

Une matrice n'est pas un pointeur de pointeur

Fichier `notptr2.c` :

```c
#include <stdio.h>

void afficher(int **m, int nl, int nc) {
  for (int i = 0; i < nl; i++) {
    for (int j = 0; j < nc; j++) {
      printf("%3d", m[i][j]);
    }
    printf("\n");
  }
}

int main(void) {
  int m[2][3] = {{1, 2, 3}, {4, 5, 6}};
  afficher(m, 2, 3);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o notptr2 notptr2.c
notptr2.c:14:12: warning: passing argument 1 of '
      afficher' from incompatible pointer type [-
      Wincompatible-pointer-types]
notptr2.c:3:21: note: expected 'int **' but argument
       is of type 'int (*)[3]'
$ ./notptr2
Segmentation fault (core dumped)
```

- `int m[2][3]` ne contient **aucun pointeur** : seulement 6 entiers contigus
- `afficher` lit `m[0]` comme une adresse, faite en réalité des entiers 1 et 2 : plantage
- `int (*)[3]` : pointeur vers un tableau de 3 `int`, le vrai type de `m` passé en argument

Exemple : tableau de chaînes

Fichier `days.c` :

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

int main(void) {
  const char *jours[] = {
    "lundi", "mardi", "mercredi", "jeudi",
    "vendredi", "samedi", "dimanche",
  };
  int n = sizeof jours / sizeof jours[0];

  for (int i = 0; i < n; i++) {
    printf("%d : %-8s (%zu)\n",
           i, jours[i], strlen(jours[i]));
  }
  printf("initiale de jeudi : %c\n", jours[3][0]);
  printf("sizeof jours = %zu\n", sizeof jours);
  return 0;
}
```

```
$ ./days
0 : lundi    (5)
1 : mardi    (5)
2 : mercredi (8)
3 : jeudi    (5)
4 : vendredi (8)
5 : samedi   (6)
6 : dimanche (8)
initiale de jeudi : j
sizeof jours = 56
```

- `jours` : tableau de 7 **pointeurs** (`sizeof` $= 7 \times 8$)
- `jours[3]` : la chaîne `"jeudi"`
- `jours[3][0]` : son premier caractère

Tableaux de chaînes

```
const char *a[] = {"lundi",
                   "mercredi"};
```

- Tableau de **pointeurs** vers des littéraux
- Chaque chaîne a sa propre longueur
- En paramètre : `const char **`

```
char b[][9] = {"lundi",
               "mercredi"};
```

- **Matrice** de caractères : 2×9 octets
- Chaque ligne a la taille de la plus longue
- Les chaînes sont **modifiables**

Lien avec les matrices

On retrouve les deux représentations des matrices : un tableau de pointeurs vers des lignes de longueurs différentes, ou une vraie matrice dont toutes les lignes ont la même taille. Dans les deux cas, `a[i][j]` et `b[i][j]` désignent le caractère `j` de la chaîne `i`.

Plus de deux dimensions

```
// 480 lignes, 640 colonnes, 3 canaux (R, V, B)
unsigned char image[480][640][3];
image[10][20][0] = 255;  // rouge au maximum
```

- Même principe : un tableau de tableaux de tableaux
- Stockage contigu, la **dernière** dimension varie le plus vite
- En paramètre, toutes les dimensions sauf la première : `void f(unsigned char img[][640][3], int lignes);`
- Taille : $480 \times 640 \times 3$ octets, soit environ 900 Ko. Attention aux grands tableaux **locaux** : la pile est limitée

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Exemple : tester la recherche du maximum

Une autre version, qui retourne la **valeur** maximale plutôt que son indice :

```
// ---------- Definitions ---------

int maximum(const int t[], int n) {
  int max = 0;
  for (int i = 0; i < n; i++) {
    if (t[i] > max) max = t[i];
  }
  return max;
}

bool tester(const char *nom, const int t[], int n,
            int attendu) {
  int obtenu = maximum(t, n);
  bool ok = obtenu == attendu;
  printf("%-5s : %-10s -> %d (attendu %d)\n",
         ok ? "OK" : "ECHEC", nom, obtenu, attendu);
  return ok;
}
```

```
int main(void) {
  int a[] = {3, 1, 4, 1, 5};
  int b[] = {9, 2, 6};
  int c[] = {2, 7};
  int d[] = {42};
  int e[] = {-4, -2, -7};
  int reussis = 0;
  reussis += tester("normal", a, 5, 5);
  reussis += tester("premier", b, 3, 9);
  reussis += tester("dernier", c, 2, 7);
  reussis += tester("un seul", d, 1, 42);
  reussis += tester("negatifs", e, 3, -2);
  printf("%d/5 tests reussis\n", reussis);
  return reussis == 5 ? 0 : 1;
}
```

Exécution des tests

```
$ gcc -Wall -Wextra -std=c23 -o testmax testmax
      .c
$ ./testmax
OK    : normal      -> 5 (attendu 5)
OK    : premier     -> 9 (attendu 9)
OK    : dernier     -> 7 (attendu 7)
OK    : un seul     -> 42 (attendu 42)
ECHEC : negatifs    -> 0 (attendu -2)
4/5 tests reussis
$ echo $?
1
```

- Maximum au début, à la fin, un seul élément : **réussis**
- Échec avec des valeurs toutes **négatives** : `max` commence à 0, une valeur absente du tableau

```
int max = 0;        // incorrect
int max = t[0];     // correct, puisque n >= 1
```

Constat

Avec des valeurs positives seulement, l'erreur passait **inaperçue**. Initialiser avec le premier élément, comme dans `indice_max`.

Cas limites

Tableaux

- Valeur cherchée en **première** et en **dernière** position, ou absente
- Tableau d'**un seul** élément ; tableau **vide** (`n = 0`), s'il est permis
- Valeurs **égales**, valeurs **négatives**, tableau déjà trié ou trié à l'envers

Chaînes

- Chaîne **vide** `""` (un seul octet : `'\0'`)
- Chaîne d'un seul caractère
- Chaîne qui remplit **exactement** le tableau, ou qui le dépasse (saisie trop longue)
- Caractères inattendus : espaces, majuscules, chiffres, ponctuation

Erreur de un

Les erreurs se cachent souvent aux **bornes** : `<` au lieu de `<=`, indice `n` au lieu de `n - 1`, oubli de la place du `'\0'`. Les cas limites sont là pour les débusquer.

Plan

1. Rappel et motivation
2. Tableaux à une dimension
3. Algorithmes classiques
4. Chaînes de caractères
5. La bibliothèque `<string.h>`
6. Tableaux à plusieurs dimensions
7. Tests
8. Débogage

Inspecter un tableau avec gdb

```
$ gdb -q matrix
Reading symbols from matrix...
(gdb) break 14
Breakpoint 1 at 0x11ff: file matrix.c, line 14.
(gdb) run
Breakpoint 1, main () at matrix.c:14
14          for (int i = 0; i < LIGNES; i++) {
(gdb) print m
$1 = {{1, 2, 3, 4}, {5, 6, 0, 8}, {9, 10, 11, 12}}
(gdb) print m[1]
$2 = {5, 6, 0, 8}
(gdb) ptype m
type = int [3][4]
```

- `print m` : affiche **tout** le tableau, ligne par ligne
- `print m[1]` : une seule ligne
- `ptype m` : le type complet, dimensions comprises
- Pour un tableau reçu en paramètre (un pointeur), il faut préciser le nombre d'éléments : `print *t@5`

Inspecter une chaîne avec gdb

```
$ gdb -q string
Reading symbols from string...
(gdb) break 5
Breakpoint 1 at 0x11b2: file string.c, line 5.
(gdb) run
Breakpoint 1, main () at string.c:5
5         printf("%s\n", s);
(gdb) print s
$1 = "Bonjour"
(gdb) print sizeof s
$2 = 8
(gdb) print s[7]
$3 = 0 '\000'
(gdb) x/8cb s
0x7fffffffd6a0: 66 'B'  111 'o' 110 'n' 106 'j' 111 'o' 117 'u' 114 'r' 0 '\000'
```

- `print s` : la chaîne, jusqu'au `'\0'`
- `x/8cb s` : 8 unités d'**un octet** (b) au format caractère (c) : le `'\000'` final est visible
- Utile pour vérifier qu'une chaîne est bien **terminée**

Exemple : débordement de chaîne

Fichier `overflow.c` :

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

int main(void) {
  char ligne[100];
  char nom[8];
  printf("Nom : ");
  if (fgets(ligne, sizeof ligne, stdin) == NULL) return 1;
  strcpy(nom, ligne);
  printf("nom    : %s", nom);
  printf("ligne : %s", ligne);
  return 0;
}
```

```
$ gcc -Wall -Wextra -std=c23 -o
      overflow overflow.c
$ ./overflow
Nom : Margaret Hamilton
nom    : Margaret Hamilton
ligne :  Hamilton
```

- Aucun avertissement, aucun plantage
- `nom` semble correct…
- … mais `ligne` a été **écrasée** : `strcpy` a débordé de `nom` sur sa voisine

Détecter les débordements

```
$ gcc -std=c23 -g -fsanitize=address -o overflow overflow.c
$ ./overflow
Nom : Margaret Hamilton
==832201==ERROR: AddressSanitizer: stack-buffer-overflow on address ...
WRITE of size 19 at 0x79cfe0d00038 thread T0
    #0 0x79cfe30a7922 in strcpy ...
    #1 0x654256a9438e in main overflow.c:9
  ...
    [48, 56) 'nom' (line 6)
```

- `-fsanitize=address` : le compilateur ajoute des **vérifications** autour de chaque accès mémoire
- Arrêt au **premier** débordement : 19 octets écrits par `strcpy` (ligne 9) dans `nom`, qui n'en a que 8
- Programme plus lent : option réservée au **débogage**. D'autres outils existent ; on y reviendra
- Correction : `snprintf(nom, sizeof nom, "%s", ligne)`