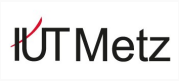
R3.05 - Programmation Système

Bases de C

Cyril Grelier

Université de Lorraine - IUT de Metz



Langage C

- Inventé au début des années **1970** par **Dennis Ritchie** aux Bell Labs.
- Utilisé initialement pour réécrire le système **Unix**.
- Langage **impératif**, **compilé**, proche du matériel.
  - un langage de **bas niveau** comparé aux langages modernes ;
  - un langage de **haut niveau** comparé à l'assembleur.
- Toujours très employé : systèmes d'exploitation/embarqués, pilotes, ...
  Et le sera pour de nombreuses années
- Voir Zig et Rust qui se présentent comme des alternatives modernes au C
- Rust étant le 3ème langage à rejoindre le développement du noyau Linux après l'assembleur et le C

Dennis Ritchie

Brian Kernighan

Hello World !

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

int main(void) {
    printf("Hello World!\n"); // écrit sur stdout
    return EXIT_SUCCESS; // signale une fin normale : EXIT_SUCCESS ou 0
    // depuis C99, le return 0 n'est plus obligatoire à la fin du main
}
```

```
$ # Compilation avec GCC
$ gcc -std=c2x -Wall -Wextra -pedantic hello.c -o hello
$ # Exécution
$ ./hello
Hello World!
```

Variables : déclaration / initialisation - allocation automatique

```c
// Déclaration
type nom_variable;
// Déclaration + initialisation (recommandé)
type nom_variable = valeur;
// Déclaration variable constante avec initialisation
const type nom_variable = valeur;
```

Les variables sont composées de :

- un **type** : qui détermine la nature et la taille de la donnée en octets (1 octet = 8 bits),
- un **nom** : lettre ou _, puis lettres/chiffres/_ ; pas de mot-clé du C,
- un **espace mémoire réservé** : dont la taille dépend du type, identifié grâce à son adresse,
- une **valeur** : optionnelle à l'initialisation, mais obligatoire avant toute utilisation, stockée dans l'espace mémoire réservé.

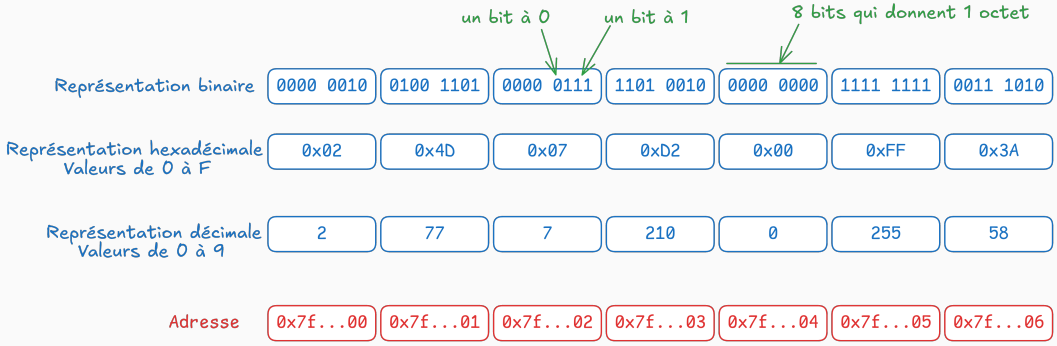
Une variable déclarée dans une fonction est allouée automatiquement sur le **stack** (la pile) puis automatiquement libérée à la fin de la fonction → on parle d'**allocation automatique**.

Types courants et sizeof

```c
#include <stdbool.h> // import bool et true/false
#include <stdint.h>  // import de int32_t int64_t

sizeof (type)   // retourne la taille
sizeof variable // de la variable/type

bool b = true;           // 1 octet  = 8  bits
char c = 65; // ou = 'A' // 1 octet  = 8  bits
int i = 42;              // 4 octets = 32 bits
int32_t i32 = 1500;      // 4 octets = 32 bits
int64_t i64 = 345623132; // 8 octets = 64 bits
float f = 3.1415926f;    // 4 octets = 32 bits
double d = -12345.6789;  // 8 octets = 64 bits
int *p = &i;             // 8 octets = 64 bits
```

La mémoire - Les octets

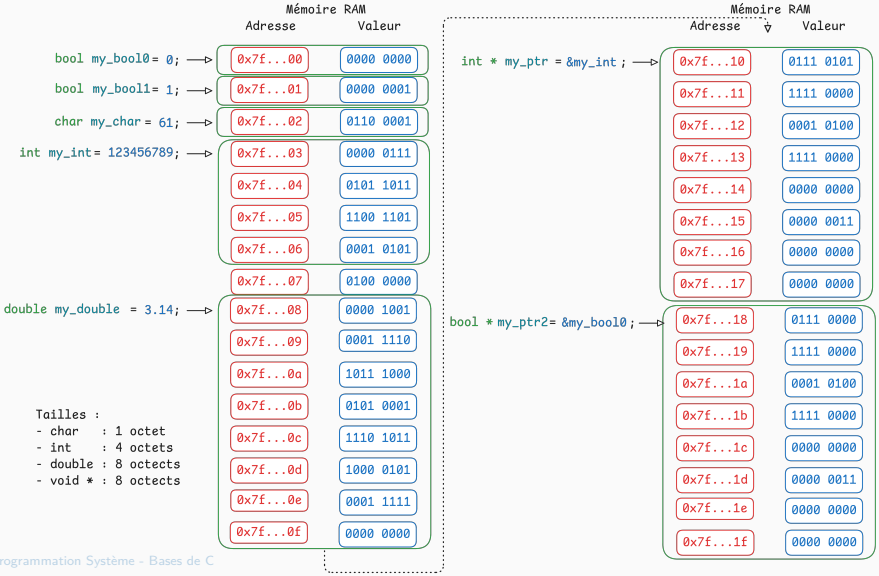
La mémoire est un ensemble de blocs de 8 bits appelés octets repérés par une adresse



Les adresses sont généralement affichées sous forme hexadécimale :

- En décimal, un octet va de 0 à 255, 1 à 3 caractères pour le représenter
- En hexadécimal, un octet va de 0x00 à 0xff, 2 caractères tout le temps

La mémoire - Les variables en mémoire



Sorties (printf, puts) & Entrées (scanf, fgets)

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

int main(void) {
    char nom[100];
    puts("Votre nom ?");
    fgets(nom, sizeof nom, stdin);  // fgets pour chaînes
    nom[strcspn(nom, "\n")] = '\0'; // retire le \n

    int age;
    puts("Quel est votre âge ?");
    scanf("%d", &age); // scanf pour nombres

    printf("Bonjour %s, %d ans.\n", nom, age);
}
```

Spécificateurs utiles : %d (int), %f (float/double), %s (char*), %zu (size_t), %p (void*).

Contrôle du flux & Boucles

```c
int a = 5;
if (a != 8) {
    printf("a n'est pas un 8");
}
if (a % 2 == 0) {
    printf("a est pair\n");
} else {
    printf("a est impair\n");
}
if (a > 10) {
    printf("a est plus grand que 10\n");
} else if (a > 3) {
    printf("a est entre 4 et 10\n");
} else {
    printf("a est 3 ou moins\n");
}
```

```c
for (int i = 0; i < 5; i++) {
    printf("%d ", i);
}

int j = 0;
while (j < 3) {
    printf("%d ", j);
    j++;
}

int k = 0;
do {
    printf("%d ", k);
    k++;
} while (k < 2);
```

Fonctions : prototypes et définitions

```c
#include <stdio.h>

// prototypes (éventuellement dans un .h)
void bonjour(void);
int get_number(void);
int addition(int a, int b);

// définitions (éventuellement dans un .c)
void bonjour(void) {
    puts("Bonjour !");
}
int get_number(void) {
    return 42;
}
int addition(int a, int b) {
    return a + b;
}
```

```c
// appels
bonjour();
printf("Numéro : %d\n", get_number());
printf("3 + 4 = %d\n", addition(3,4));
```

- Fonctions sans paramètres
  $\rightarrow$ préciser `void`
- Avant C23 : sans `void`, un appel comme `bonjour("coucou");` compile sans erreur

main et arguments

```c
#include <stdio.h>

// si pas d'arguments :
int main (void) {
    // ...
}
// sinon :
// int main(int argc, char *argv[])
// int main(int argc, char **argv)
// ou depuis C23 (+1 car un pointeur NULL est ajouté à la fin des arguments) :
int main(int argc, char *argv[argc + 1]) {
    printf("argc = %d\n", argc);
    for (int i = 0; i < argc; ++i) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
}
```

Tableaux : 1D et 2D

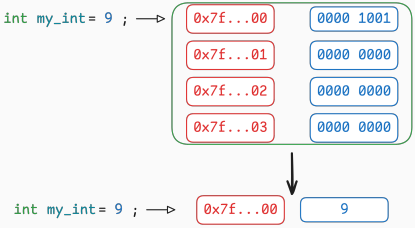
```c
// -------- Tableau 1D --------
#define TAILLE 5
int notes[TAILLE] = {8, 5, 7, 1, 4};
for (int i = 0; i < TAILLE; ++i) {
    printf("%d ", notes[i]);
}

// -------- Tableau 2D --------
// lignes x colonnes
int matrice[3][2] = {
    {1, 2},
    {3, 4},
    {5, 6}
};
printf("%d\n", matrice[1][0]); // 3
```

```c
// -------- Fonctions 1D --------
void afficher_tab(const int *t, size_t n);
// ou
void afficher_tab(const int t[], size_t n);

// appel :
afficher_tab(notes, TAILLE);

// -------- Fonctions 2D --------
// On doit préciser la taille de la 2e dimension :
void afficher_mat(const int m[][2], size_t lignes);
// ou (équivalent) :
void afficher_mat(const int (*m)[2], size_t lignes);

// appel :
afficher_mat(matrice, 3);
```

Pas de vérification d'indices : hors bornes → **comportement indéfini** (*Undefined Behavior* - UB).

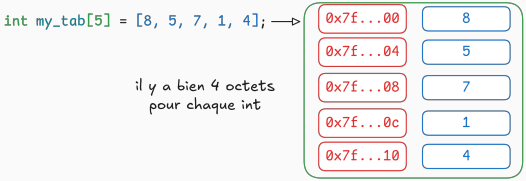
Tableaux

Simplification la représentation d'un int



my_int prend 4 octects qui
commencent à l'adresse 0x7f...00

Représentation d'un tableau d' int



il y a bien 4 octets
pour chaque int

L'adresse du tableau est 0x7f...00
Le programme sait qu'en mémoire
il y a 5 cases de taille 4 octects qui se suivent

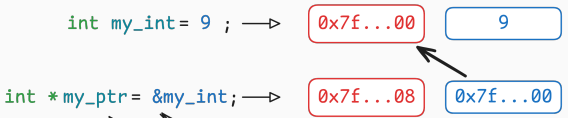
Pointeurs : bases

Un **pointeur** est une variable qui contient l'**adresse** mémoire d'une autre variable.

```c
int var = 42;
int *ptr = &var; // ptr pointe vers var

// cast des adresses en void * pour %p
printf("%p\n", (void *)&var); // 0x7ffc75afed9c
printf("%p\n", (void *)ptr);  // 0x7ffc75afed9c
printf("%d\n", *ptr);         // 42

*ptr = 5;             // modifie var via ptr grâce au déréférencement
printf("%d\n", var); // 5
```

Peu importe le type pointé, sur un système 64 bits, un pointeur aura toujours une taille de 64 bits/8 octets. La taille réservée pour le pointeur correspond à celle d'une adresse en mémoire, pas à la donnée pointée.

Pointeurs



my_int prend 4 octects qui commencent à l'adresse 0x7f...00

my_ptr enregistre sur 8 octects l'adresse de my_int

& : donne l'adresse d'une variable

* deux cas :

- *À* la déclaration : il indique que la variable est un pointeur vers un type

- Devant une variable pointeur (hors déclaration), il permet de déréférencer le pointeur, c'est-à-dire accéder à la valeur stockée à l'adresse pointée

```
*my_ptr = 10
```

↓

```
my_int == 10
```

Passage par adresse

Les pointeurs permettent :

- de modifier directement une variable déclarée dans une autre fonction
- de retourner plusieurs résultats en modifiant les variables passées en paramètres

```c
void doubler(int *n) {
    *n *= 2;
}
void carre_cube(int n, int *carre, int *cube) {
    *carre = n * n;
    *cube  = n * n * n;
}
// main ...
int a = 10;
doubler(&a);       // on passe l'adresse de a
printf("%d\n", a); // affiche 20
int n = 3;
int carre;
int cube;
carre_cube(n, &carre, &cube);
printf("%d^2 = %d, %d^3 = %d\n", n, carre, n, cube);
// affiche : 3^2 = 9, 3^3 = 27
```

Tableaux et fonctions

```c
// Deux formes équivalentes (le tableau devient un pointeur) :
void inverser_signes(int *tab, size_t n);
// ou :
void inverser_signes(int tab[], size_t n);

void inverser_signes(int *tab, size_t n) {
    for (size_t i = 0; i < n; ++i)
        tab[i] = -tab[i];
}

int tab[4] = {1, -2, 3, -4};
inverser_signes(tab, 4);
```

Bonnes pratiques : utiliser `size_t` pour les tailles et `const` si on ne modifie pas le tableau.

Structures

```c
typedef struct {
    int x; // champs ou
    int y; // membres
} Point;

// ancienne déclaration :
struct Point{
    int x;
    int y;
};

// passage par adresse :
void deplacer(Point *p) {
    p->x++; // == (*p).x++
    p->y++; // == (*p).y++
}
```

```c
// Déclaration puis initialisation
Point p1;
// struct Point p1; si ancien type de déclaration
p1.x = 3;
p1.y = 4;
// Déclaration et initialisation
Point p2 = {5, 6};
// Déclaration et initialisation avec noms de champs
Point p3 = {.x = 5, .y = 4};
printf("(%d,%d)\n", p1.x, p1.y);
deplacer(&p);
printf("(%d,%d)\n", p1.x, p1.y);
```

Fin Partie 1

Vous pouvez faire les exo 1 et 2 de la feuille d'exo sur ARCHE

Séance 2 :

- Chaînes de caractères
- Rappels des pointeurs et passage par adresse
- Allocation dynamique
- Tableaux 2D dynamique
- Erreurs
- Aléatoire

Chaînes de caractères (<string.h>)

```c
#include <string.h>

char nom[4] = "Bob";
// équivalent à :
// char nom[4] = { 'B', 'o', 'b', '\0' };
char nom_complet[40];

strcpy(nom_complet, nom);
strcat(nom_complet, " Dupont");

size_t len = strlen(nom);
if (strcmp(nom, "Bob") == 0) {
    printf("C'est Bob !");
}
```

Toujours réserver la place pour le `\0`.

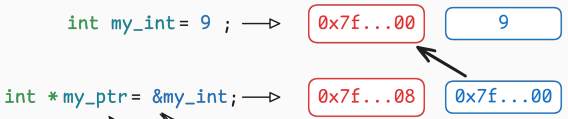
Pointeurs : bases

Un **pointeur** est une variable qui contient l'**adresse** mémoire d'une autre variable.

```c
int var = 42;
int *ptr = &var; // ptr pointe vers var

// cast des adresses en void * pour %p
printf("%p\n", (void *)&var); // 0x7ffc75afed9c
printf("%p\n", (void *)ptr);  // 0x7ffc75afed9c
printf("%d\n", *ptr);         // 42

*ptr = 5;             // modifie var via ptr grâce au déréférencement
printf("%d\n", var); // 5
```

Peu importe le type pointé, sur un système 64 bits, un pointeur aura toujours une taille de 64 bits/8 octets. La taille réservée pour le pointeur correspond à celle d'une adresse en mémoire, pas à la donnée pointée.

Pointeurs



my_int prend 4 octects qui commencent à l'adresse 0x7f...00

my_ptr enregistre sur 8 octects l'adresse de my_int

& : donne l'adresse d'une variable

* deux cas :

- *À* la déclaration : il indique que la variable est un pointeur vers un type

- Devant une variable pointeur (hors déclaration), il permet de déréférencer le pointeur, c'est-à-dire accéder à la valeur stockée à l'adresse pointée

```
*my_ptr = 10
    ↓
my_int == 10
```

Passage par adresse

Les pointeurs permettent :

- de modifier directement une variable déclarée dans une autre fonction
- de retourner plusieurs résultats en modifiant les variables passées en paramètres
- d'optimiser certaines opérations en évitant des copies coûteuses
- de gérer des structures de données dynamiques (listes, tableaux dynamiques, etc.)

```c
void doubler(int *n) {
    *n *= 2;
} // doubler(&mon_int);
void carre_cube(int n, int *carre, int *cube) {
    *carre = n * n;
    *cube  = n * n * n;
} // carre_cube(numero, &carre, &cube);
void echanger(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
} // echanger(&x, &y);
void deplacer(Point *point) {
    point->x++; // (*point).x == point->x
    point->y++; // (*point).y == point->y
} // deplacer(&mon_point);
void inverser_signes(int *tab, size_t taille) {
    for (size_t i = 0; i < taille; ++i)
        tab[i] = -tab[i];
} // inverser_signes(tab, taille);
```

Allocation : automatique *vs* dynamique

Automatique (pile / *stack*)

- Déclarée dans une fonction
  → vit le temps du bloc
- Taille connue à la compilation
- Libération **implicite** à la sortie du bloc

Dynamique (tas / *heap*)

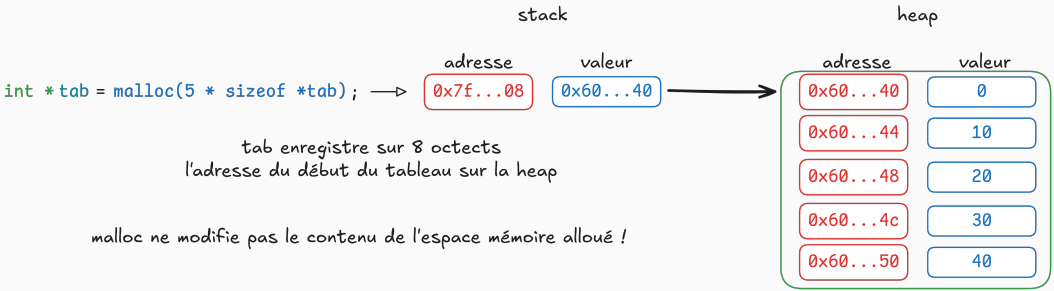
- Allouée **à l'exécution** avec `malloc/calloc/realloc`
  → vit jusqu'à la libération avec `free`
- Taille choisie à l'exécution
- **Libération manuelle** avec `free`

```c
// Allocation automatique : stack (pile)
void ma_fonction(void) {
    int a = 2;
    int t[5] = {1,2,3,4,5}; // taille fixe
    printf("%d %d\n", a, t[0]);
} // a et t sont libérés automatiquement ici

// Allocation dynamique : heap (tas)
int *init_tab_dynamique(size_t n) {
    int *tab = malloc(n * sizeof *tab);
    if (!tab) { /* gérer l'erreur */ return; }

    for (size_t i = 0; i < n; ++i)
        tab[i] = (int)i;

    return tab;
} // le free sera à faire par la fonction qui récupère tab

// À ne jamais faire !!!
// Ne pas retourner l'adresse d'une variable déclarée sur le stack
int *mauvais(void) {
    int x = 42; // vit sur le stack de la fonction
    return &x;  // dangling pointer après retour
} // x est libéré automatiquement ici
```

Allocation dynamique - Tableau 1D - malloc/free <stdlib.h>

```c
// allocation sur la heap d'une zone mémoire de taille 5 * int
int *tab = malloc(5 * sizeof *tab);
if (!tab) { // malloc retourne NULL si la mémoire n'est pas allouée
    perror("malloc");
    exit(EXIT_FAILURE);
}

for (int i = 0; i < 5; ++i) {
    tab[i] = i * 10;
}

free(tab); // on libère la mémoire allouée dynamiquement
tab = NULL; // on évite une utilisation accidentelle du pointeur libéré
// si on utilise tab ici -> segfault
// si on utilisait tab ici sans le tab = NULL
//    -> comportement indéfini (le programme fait n'importe quoi ou segfault)
```

Allocation dynamique - Tableau 1D



Un pointeur déclaré sans malloc est une variable comme les autres → il vit sur le stack (pile).

Avec malloc :

- le pointeur est toujours sur le stack
- mais la mémoire pointée est allouée sur la heap (tas)

Allocations dynamiques & paramètres de fonction

```c
// alloue et renvoie la mémoire (propriété -> appelant)
int *tab_creer(size_t n, int val) {
    int *t = malloc(n * sizeof *t);
    if (!t) return NULL;
    for (size_t i = 0; i < n; ++i) t[i] = (int)val;
    return t;  // l'appelant devra free(t)
}

// la fonction utilise la structure
// pas de transfert de propriété
void tab_remplir(int *t, size_t n, int val) {
    for (size_t i = 0; i < n; ++i) t[i] = val;
}

// la fonction redimensionne (double pointeur + realloc)
// pas de transfert de propriété
// mais retour indique si le redimensionnement a
↪  fonctionné
int tab_redimensionner(int **t, size_t nouveau_n) {
    int *tmp = realloc(*t, nouveau_n * sizeof **t);
    if (!tmp) return 0;  // échec: *t inchangé
    *t = tmp;            // succès: *t mis à jour
    return 1;
}
```

```c
size_t n = 5;
int *t = tab_creer(n, 7);
if (!t) { perror("malloc"); return EXIT_FAILURE; }

tab_remplir(t, n, 42); // t déjà alloué par l'appelant

if (!tab_redimensionner(&t, 10)) {
    free(t);
    return EXIT_FAILURE;
}
// ... utiliser t[0..9] ...

free(t); // propriété appelant
t = NULL;
```

Règle d'ownership : celui qui alloue (`malloc`/`calloc`/`realloc`) est responsable du `free`, sauf si la fonction documente explicitement un transfert de propriété.

Allocation dynamique - struct

```c
typedef struct {
    int x;
    int y;
} Point;

// Fabrique : alloue et initialise
Point *point_creer(int x, int y) {
    Point *p = malloc(sizeof *p);
    if (!p) { perror("malloc Point"); return NULL; }
    p->x = x;
    p->y = y;
    return p; // propriété transférée à l'appelant
}

void point_afficher(const Point *p) {
    printf("(%d,%d)\n", p->x, p->y);
}

void point_detruire(Point **pp) {
    if (pp && *pp) {
        free(*pp);
        *pp = NULL; // éviter le dangling pointer
    }
}
```

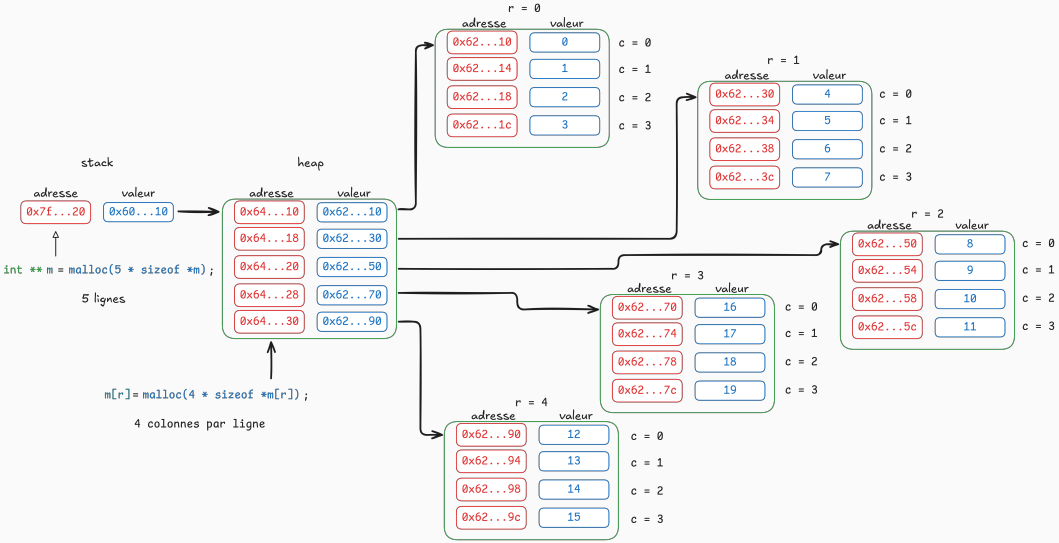
```c
int main(void) {
    Point *p = point_creer(3, 4);
    if (!p) return EXIT_FAILURE;

    point_afficher(p);  // (3,4)

    p->x++;     // modification
    p->y += 10; // via pointeur
    point_afficher(p);  // (4,14)

    // on donne l'adresse du pointeur pour
    // permettre la mise à NULL
    point_detruire(&p); // free + mise à NULL
}
```

- point_creer → transmet la propriété (ownership)
- point_detruire → libère et met à NULL
- le main gère le cycle de vie complet

Tableau 2D dynamique : tableau de pointeurs

```
1  size_t R=5, C=10;
2  int **m = malloc(R * sizeof *m);
3  if (!m) { perror("malloc row"); exit(EXIT_FAILURE); }
4  for (size_t r = 0; r < R; ++r) {
5    m[r] = malloc(C * sizeof *m[r]);
6    if (!m[r]) { perror("malloc col");
7      for (size_t i = 0; i < r; ++i) free(m[i]);
8      free(m); exit(EXIT_FAILURE);
9    }
10 }
11 for (size_t r=0;r<R;++r)
12   for (size_t c=0;c<C;++c)
13     m[r][c] = (int)(c + r*C);
14
15 for (size_t r=0;r<R;++r) free(m[r]);
16 free(m);
```

Tableau 2D dynamique : tableau de pointeurs



Gestion des erreurs : valeurs de retour / errno

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

int *tab = malloc(5 * sizeof *tab);
if (!tab) { // malloc retourne NULL si la mémoire n'est pas allouée
    fprintf(stderr, "Erreur %d : %s\n", errno, strerror(errno));
    // Affiche dans stderr : Erreur 12 : Cannot allocate memory
    perror("malloc");
    // Affiche dans stderr : malloc: Cannot allocate memory
    return EXIT_FAILURE;
}
// utilisation de tab ...
free(tab);  // on libère la mémoire allouée dynamiquement
tab = NULL; // on évite une utilisation accidentelle du pointeur libéré
```

Toujours vérifier `malloc`, `fopen`, `fork`, etc.

`perror` plus simple à utiliser.

Aléatoire : rand/srand

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

int main(void) {
    // On initialise une graine (seed) à partir du temps courant
    // time(NULL) renvoie le nombre de secondes écoulées depuis le 1er janvier 1970
    // Comme cette valeur change à chaque seconde, on obtient une graine différente à chaque exécution
    unsigned int seed = (unsigned int) time(NULL);
    // srand() initialise le générateur de nombres pseudo-aléatoires avec la graine donnée
    // Si on ne l'appelle pas, le générateur utilise par défaut la graine 1
    // ce qui produit toujours la même séquence de nombres à chaque exécution
    srand(seed); // à faire 1 seule fois !
    // La graine contrôle le point de départ de la séquence pseudo-aléatoire
    // Deux programmes lancés avec la même graine produiront exactement les mêmes "nombres aléatoires"
    for (int i = 0; i < 10; ++i) {
        // rand() renvoie un entier pseudo-aléatoire entre 0 et RAND_MAX
        // En prenant rand() % 100, on limite la valeur à l'intervalle [0, 99]
        printf("%d ", rand() % 100);
    }
    // Avec la graine 0, on obtiendra toujours la même suite : 83 86 77 15 93 35 86 92 49 21
    // Avec une graine basée sur le temps (time(NULL)), la suite changera à chaque exécution
}
```