Université de Bordeaux 2026–27 — Licence et CMI OSIA 2e année
Programmation C — Feuille d'exercices no 1 : pointeurs et tableaux

⚠ Consignes

Pour cette séance sur feuille, sans ordinateur, il ne s'agit pas d'écrire du code mais de **lire** des programmes déjà écrits et de **dessiner l'état de la mémoire** à certains moments de leur exécution.

Pour ces dessins, on utilise la convention suivante : chaque variable est représentée par une case portant son nom et contenant sa valeur. Un pointeur est ainsi une case dont la valeur est une adresse. Cependant, plutôt que d'inventer pour l'adresse une fausse valeur numérique, on la représentera par une *flèche* allant vers la case pointée. Une case dessinée avec un point d'interrogation « ? » signifie que son contenu est indéterminé (variable locale non `static`, définie mais non initialisée). Un tableau se représente comme une suite de cases *contiguës*.

Par exemple, avec le code `int a = 5; int *p = &a;`, la case de `p` est laissée vide et une flèche part de `p` vers `a` : c'est ainsi que l'on représente « `p` contient l'adresse de `a` ».

Quand une zone mémoire est allouée dynamiquement (avec `malloc`, `calloc` ou `realloc`), indiquez à côté de cette zone « tas », pour bien mémoriser que cette zone n'est pas allouée sur la pile d'exécution.

Rappels essentiels

Un *pointeur* est une variable dont la valeur est une adresse mémoire : `T *p;` déclare un pointeur `p` destiné à pointer vers une valeur de type `T`. L'opérateur `&` donne l'adresse d'une variable, l'opérateur `*` accède (en lecture ou en écriture) à la valeur pointée. Un pointeur ne doit **jamais** être déréférencé (avec `*`) tant qu'il ne pointe pas vers un emplacement mémoire valide : soit l'adresse d'une variable existante (`&x`), soit une zone allouée dynamiquement (`malloc`). Déréférencer un pointeur non initialisé, non affecté ou déjà libéré est une erreur grave. Le comportement du programme est indéfini (même si le programme peut « sembler fonctionner »).

Une définition de *tableau* par `T tab[n];` réserve `n` cases contiguës de type `T`, toutes allouées dès cette définition. Dans (presque) toute expression, le nom `tab` d'un tableau est automatiquement converti en un pointeur vers sa première case : c'est pourquoi `tab[i]` et `*(tab + i)` désignent exactement la même case, que `tab` soit un « vrai » tableau ou un simple pointeur.

Points communs

- Même syntaxe d'indexation : `p[i]` ≡ `*(p + i)`, que `p` soit un tableau ou un pointeur.
- L'arithmétique de pointeurs avance de `sizeof(T)` octets[a], pas d'un seul. Autrement dit, `p + 1` pointe sur l'élément *suivant*, pas sur l'octet suivant en général.
- Un tableau passé en argument d'une fonction est « dégradé » en un simple pointeur vers sa première case (voir l'exercice 10).

a. L'unité de `sizeof` est le *byte*, une unité qui coïncide souvent avec l'octet, mais ce n'est pas imposé par la norme C.

Différences essentielles

- Un tableau désigne toujours une mémoire valide dès sa définition, alors qu'un pointeur peut être non initialisé, `NULL`, ou *pendant* (il a pointé vers une mémoire depuis libérée ou désallouée).
- Un tableau ne peut pas être réaffecté (c'est-à-dire que `tab = ...;` est interdit), alors que l'affectation est possible pour un pointeur (`p = ...;` est toujours valide).
- `sizeof(tab)` donne la taille *totale* du tableau, en octets ; `sizeof(p)` donne la taille d'*un* pointeur (souvent 8), quel que soit ce vers quoi il pointe.

Pictogrammes des exercices : ❗ Exercice important. ☆ à ★★★★ Difficulté. 🏠 À faire chez soi.

1 Pointeurs simples

Exercice 1 – ☆ ❗ Écritures via des pointeurs. On considère les deux programmes C suivants.

```c
#include <stdio.h>

int main(void) {
  int  a = 5, b = 10;
  int *p = &a;

  *p = 42;

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

```c
#include <stdio.h>

int main(void) {
  int  a = 5, b = 10;
  int *p = &a;

  p  = &b;
  *p = 42;

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

1. Dans chaque programme, dessiner l'état de la mémoire juste après la ligne `int *p = &a;`.
2. Dessiner à nouveau l'état de la mémoire juste avant l'appel à `printf`.
3. Qu'affiche chacun des deux programmes ?
4. En une phrase, expliquer ce que fait l'instruction `*p = 42;` et l'instruction `p = &b;`.

Exercice 2 – ★ ❗ Alias par pointeurs. On considère le programme C suivant :

```c
#include <stdio.h>

int main(void) {
  int a = 1, b = 2, c = 3, d = 4;
  int *p1 = &a;
  int *p2 = &b;
  int *p3 = &c;
  int *p4 = &d;
  int *p5;

  p5  = p3;
  *p3 = *p2;
  *p2 = *p5;
  *p4 = *p1;
  *p1 = *p4;
  printf("a, b ,c et d sont égaux à:"
         "%d, %d, %d et %d\n",
         a, b, c, d);
  return 0;
}
```

1. Dessiner l'état de la mémoire après que toutes les variables ont été déclarées.
2. Dessiner à nouveau l'état de la mémoire après l'instruction `p5 = p3;`.
3. Dessiner l'état de la mémoire après chacune des quatre affectations qui suivent.
4. Que produit le programme ?

2 Pointeurs et fonctions

Exercice 3 – ★★☆ ❗ Passage de paramètres. Que produisent les quatre programmes suivants ? Dessiner la mémoire au moment de l'appel des fonctions `swap1`, `swap2`, `swap3`, `swap4`, juste avant et juste après leur instruction `return`.

Rappel. La pile d'exécution contient un cadre par fonction actuellement appelée.

```c
#include <stdio.h>

void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}

int main(void) {
  int a = 1, b = 2;
  swap1(a, b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
```

```c
#include <stdio.h>

void swap2(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int main(void) {
  int a = 1, b = 2;
  swap2(&a, &b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
```

```c
#include <stdio.h>

void swap3(int *a, int *b) {
  int *temp = a;
  a = b;
  b = temp;
}

int main(void) {
  int a = 1, b = 2;
  swap3(&a, &b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
```

```c
#include <stdio.h>

void swap4(int *a, int *b) {
  int *temp;
  *temp = *a;
  *a = *b;
  *b = temp;
}

int main(void) {
  int a = 1, b = 2;
  swap4(&a, &b);
  printf("a = %d, b= %d.\n", a, b);
  return 0;
}
```

Exercice 4 – ★★☆ ❗ Incrémentation et pointeurs. On considère le programme suivant :

```c
#include <stdio.h>

void f(int *p) {
  (*p)++;
}

void g(int *p) {
  p++;
}

void h(int *p) {
  *p++;
}

int main(void) {
  int x = 10;

  f(&x);
  printf("Après f : x = %d\n", x);

  g(&x);
  printf("Après g : x = %d\n", x);

  h(&x);
  printf("Après h : x = %d\n", x);

  return 0;
}
```

1. Dessiner la mémoire juste avant l'entrée et la sortie des fonctions `f`, `g` et `h`.
2. Décrire pas à pas le comportement du programme.

Exercice 5 – ★★ ❗ Fonctions renvoyant des pointeurs. On considère les programmes suivants :

```c
#include <stdio.h>

int *point1(void) {
  int n = 42;
  return &n;
}

int main(void) {
  int *q = point1();
  printf("%d.\n", *q);
  return 0;
}
```

```c
#include <stdio.h>

int *point2(void) {
  int *p;
  *p = 42;
  return p;
}

int main(void) {
  int *q = point2();
  printf("%d.\n", *q);
  return 0;
}
```

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

int *point3(void) {
  int *p = malloc(sizeof(int));
  *p = 42;
  return p;
}

int main(void) {
  int *q = point3();
  printf("%d.\n", *q);
  free(q);
  return 0;
}
```

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

void point4(int *p) {
  p  = malloc(sizeof *p);
  *p = 42;
}

int main(void) {
  int *p;
  point4(p);
  printf("%d.\n", *p);
  return 0;
}
```

Pour chacun des quatre programmes :

1. Dessiner la mémoire juste avant l'instruction `return` (ou, pour `point4`, juste avant la fin de la fonction), en faisant apparaître le cadre de la fonction `main` *et* le cadre de la fonction appelée.
2. Dessiner à nouveau la mémoire juste après le retour dans `main`, avant l'exécution de `printf`.
3. L'affichage produit est-il garanti ? Sinon, pourquoi, et que risque-t-il de se produire à l'exécution (valeur imprévisible, plantage…) ?

3 Pointeurs sur pointeurs

Exercice 6 – ★★ Pointeurs sur pointeurs (1). On considère le programme C suivant :

```c
#include <stdio.h>

int main(void) {
  int a = 3, b = 7;
  int *p = &a;
  int **pp = &p;
  **pp = **pp + b;
  p    = &b;
  *p   = *p - 1;
  **pp = **pp * 2;

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

1. Dessiner l'état de la mémoire juste après les définitions, puis après chaque instruction.
2. Que produit le programme ?

Exercice 7 – ★★☆ Pointeurs sur pointeurs (2). On considère le programme C suivant :

```c
#include <stdio.h>

int main(void) {
  int a = 1, b = 2, c = 3, d = 4;
  int *p1  = &a;
  int **p2 = &p1;
  int *p3  = &c;
  int *p4  = &b;
  int **p5 = &p3;
  int *p6  = &d;

  *p3      = **p2 + 6;
  *p2      = p3;
  *p1      = *p4 * 5;
  p5       = &p6;
  **p5     = **p5 + **p2;
  *p4      = *p3 + 14;
  *p5      = p1;
  **p2     = 1 + **p2;
  **p5     = **p5 * **p2;

  printf("a, b ,c et d sont égaux à :"
         "%d, %d, %d et %d\n",a,b,c,d);
  return 0;
}
```

1. Dessiner l'état de la mémoire après que toutes les variables ont été déclarées.
2. Redessiner la mémoire après les 5 premières affectations, juste avant l'instruction `*p4 = *p3 + 14;`.
3. Redessiner la mémoire après que toutes les affectations ont été réalisées.
4. Que produit le programme ?

Exercice 8 – ★★★ Triple indirections. On considère le programme C suivant.

```c
#include <stdio.h>

int x1, y1, y2, z1, z2;
int *px1 = &x1;
int *py1 = &y1;
int *py2 = &y2;
int *pz1 = &z1;
int *pz2 = &z2;

void aux(int **p, int ***q) {
  **p  = 2;
  ***q = 99;
  *p   = pz1;
  **p  = 12;
  p    = &py2;
  *q   = &px1;
  **p  = 4;
  ***q = 42;
}
```

```c
int main(void) {
  int **p;
  int **q;
  int ***r;
  int s;
  int *t;

  p     = &py1;
  q     = &px1;
  r     = &q;
  *p    = &s;
  **q   = 1;
  **p   = 34;
  ***r  = 76;
  t     = **r;
  r     = &p;

  aux(p,&q);
  printf("%d %d %d %d %d \n",
         **q, **p, ***r, s, *t);
  return 0;
}
```

1. Dessiner l'état de la mémoire après que toutes les variables globales et locales de `main` ont été déclarées, jusqu'à `int *t;` inclus (avant la première affectation de `main`).
2. Redessiner la mémoire après les neuf premières instructions de `main`, jusqu'à `r = &p;` inclus.
3. Redessiner la mémoire après l'appel de la fonction `aux`, juste avant son `return`, en faisant apparaître le cadre de `main` et le cadre de `aux` pendant l'exécution de cette dernière.
4. Que produit le programme ?

4 Tableaux et pointeurs

Exercice 9 – ★ ❗ Tableaux et arithmétique de pointeurs. On considère le programme C suivant :

```c
#include <stdio.h>

int main(void) {
  int arr[5] = {10, 20, 30, 40, 50};
  int *p = arr;

  printf("arr[2]=%d *(arr+2)=%d *(p+2)=%d p[2]=%d\n",
         arr[2], *(arr + 2), *(p + 2), p[2]);

  *(p + 1) = 99;
  arr[3]   = *p + 5;
  p        = p + 2;
  *p       = 0;
  printf("%d\n", p[5]);
  return 0;
}
```

1. Dessiner l'état de la mémoire juste après `int *p = arr;`.
2. Que produit la première instruction `printf` ?
3. Dessiner l'état de la mémoire après les 4 instructions suivantes (de `*(p + 1) = 99;` à `*p = 0;`).
4. Que fait le programme ?

Exercice 10 – ★★☆ Un tableau passé en paramètre. On considère le programme C suivant :

```c
#include <stdio.h>

void fun(int *arr) {
  printf("Taille vue dans la fonction"
         " %s = %zu\n", __func__,
         sizeof(arr) / sizeof(arr[0]));
}

int somme(int *arr, int n) {
  int total = 0;
  for (int i = 0; i < n; i++)
    total += arr[i];
  return total;
}
```

```c
int main(void) {
  int t[6] = {1, 2, 3, 4, 5, 6};
  printf("Taille vue dans la fonction"
         " %s = %zu\n", __func__,
         sizeof(t) / sizeof(t[0]));
  fun(t);
  printf("Somme = %d\n", somme(t, 6));
  return 0;
}
```

1. Dessiner l'état de la mémoire dans `main`, juste après la définition de `t`.
2. Dessiner l'état de la mémoire pendant l'exécution de `fun(t)` : faire apparaître le cadre de `main` et le cadre de `fun`, et bien représenter ce que contient *réellement* le paramètre `arr`.
3. Que vaut `sizeof(t)` dans `main` ? Que vaut `sizeof(arr)` dans `fun` ? Ces deux tailles sont-elles comparables ?
4. En déduire l'affichage produit par le programme. Pourquoi la fonction `somme` a-t-elle besoin de recevoir `n` en argument, alors que `main` connaît la taille de `t` sans qu'on la lui donne explicitement ?

5 Allocation dynamique

Exercice 11 – ★★ ❗ Allocation dynamique. On considère la fonction `main` suivante, qui appelle une fonction `creer_tableau`.

```c
int main(void) {
  int *t = creer_tableau(5);
  printf("%d\n", t[3]);
  free(t);
  return 0;
}
```

On considère les 5 versions suivantes de la fonction `creer_tableau()`.

```c
// Version 1
int *creer_tableau(int n) {
  int *t = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)
    t[i] = i * i;
  return t;
}
```

```c
// Version 2
int *creer_tableau(int n) {
  int *t = malloc(n);
  for (int i = 0; i < n; i++)
    t[i] = i * i;
  return t;
}
```

```c
// Version 3
int *creer_tableau(int n) {
  int *t;
  for (int i = 0; i < n; i++)
    t[i] = i * i;
  return t;
}
```

```c
// Version 4
int *creer_tableau(int n) {
  int *t = malloc(n * sizeof(int));
  free(t);
  for (int i = 0; i < n; i++)
    t[i] = i * i;
  return t;
}
```

```c
// Version 5
int *creer_tableau(int n) {
  int *t = NULL;
  for (int i = 0; i < n; i++) {
    t = malloc(n * sizeof(int));
    t[i] = i * i;
  }
  return t;
}
```

Pour chacune des versions, expliquez le comportement du programme en dessinant la mémoire. Indiquez en particulier si le programme provoque une erreur (comportement indéterminé, erreur à l'exécution, ...).

Exercice 12 – ★★★★ Pointeurs et récursion. Déterminer et expliquer le comportement des exécutables qui correspondent aux 4 sources suivants, en dessinant l'évolution de la mémoire si besoin.

```c
// Code 1, #include omis
int resultat = 0;

void foo(int **p) {
  resultat++;
  *p = &resultat;
  main();
}

int m;
int *p[5] = {&m, &m, &m, &m, &m};

int somme(void) {
  int i, s = 0;
  for (i = 0; i < 5; i++)
    s += *p[i];
  return s;
}

int main(void) {
  static int n;
  if (n == 5)
    printf("%d\n", somme());
  else
    foo(p + n++);
  exit(EXIT_SUCCESS);
}
```

```c
// Code 2, #include omis
int resultat = 0;

void foo(int *p) {
  resultat++;
  p = &resultat;
  main();
}

int m;
int *p[5] = {&m, &m, &m, &m, &m};

int somme(void) {
  int i, s = 0;
  for (i = 0; i < 5; i++)
    s += *p[i];
  return s;
}

int main(void) {
  static int n;
  if (n == 5)
    printf("%d\n", somme());
  else
    foo(p[n++]);
  exit(EXIT_SUCCESS);
}
```

```c
// Code 3, #include omis
int resultat = 0;

void foo(int **p) {
  int resultbis = ++resultat;
  *p            = &resultbis;
  main();
}

int m;
int *p[5] = {&m, &m, &m, &m, &m};

int somme(void) {
  int i, s = 0;
  for (i = 0; i < 5; i++)
    s += *p[i];
  return s;
}

int main(void) {
  static int n;
  if (n == 5)
    printf("%d\n", somme());
  else
    foo(p + n++);
  exit(EXIT_SUCCESS);
}
```

```c
// Code 4, #include omis
int resultat = 0;

void foo(int **p) {
  int resultbis = ++resultat;
  *p            = &resultbis;
  main();
}

int m;
int *p[5] = {&m, &m, &m, &m, &m};

int somme(void) {
  int i, s = 0;
  for (i = 0; i < 5; i++) {
    printf("%d\n", *p[i]);
    s += *p[i];
  }
  return s;
}

int main(void) {
  static int n;
  int m = n;
  if (n != 5) {
    foo(p + (m = n++));
    if (m == 0)
      printf("%d\n", somme());
    else
      return 0;
  }
  exit(EXIT_SUCCESS);
}
```

Exercice 13 – ★★★★ Segfault. Expliquer le comportement du programme C suivant :

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

#define S(x) (char[]) {x}

int main(int argc, char *argv[]) {
  char *s[] = {S("Programmer en C"),
               S("apporte"),
               S("sagesse"),
               S("et satisfaction"),
               S("ultime."),
               "Michel Eyquem, seigneur de Montaigne, 1/4/1580"};

  char **ps[] = {s + 4, s + 3, s + 2, s + 1, s};

  char ***pps = ps;

  (*(++pps)[1])[1]          = 'e';                    // instruction 1
  (*pps++[1])[3]            = '\0';                   // instruction 2
  printf("[1]    %d %s", getpid(), **pps);          // instruction 3
  (*(++pps)[1])[9]          = '\0';                   // instruction 4
  (*(pps)[1])[13]           = '\0';                   // instruction 5
  printf("%s", (*(pps++[1]) + 7));                  // instruction 6
  printf("%s", (pps++[0][0] + 12));                 // instruction 7
  printf("%c%c", (*(pps[-5]))[2], (*(pps[-2]))[0]); // instruction 8
  printf("%s ", (*(pps--[-4])) + 11);               // instruction 9
  ((*(pps--[-3])) + 11)[-1] = '\0';                 // instruction 10
  printf("%s", (*(pps--[-2])) + 8);                 // instruction 11
  printf("%.3s %s\n", (*(pps--[-2])), argv[0]);     // instruction 12
  **(*pps + 2)              = 0;                    // instruction 13
  exit(0);
}
```

Note

La macro `S` sert à rendre les zones mémoires correspondantes modifiables. Autrement dit, on peut modifier les caractères des chaînes `s[0]` à `s[4]`. En revanche, la chaîne `s[5]` n'est pas modifiable.