**Informatique** *Cours*
**Programmation impérative avec C - Pointeurs, tableaux, structures** Ipesup

## Objectifs

— Comprendre le modèle d'exécution du C : pile d'appels, tas, passage des arguments par valeur.
— Manipuler pointeurs, tableaux et structures ; allocation statique et dynamique (`malloc`/`free`).

# 1 Modèle d'exécution

Comprendre le modèle d'exécution d'un langage est essentiel, à la fois pour garantir la correction d'un programme (une donnée est-elle copiée ou partagée ?) et pour en évaluer l'efficacité (quelle est la complexité d'une instruction ?).

Le modèle d'exécution du C est plus subtil que celui d'OCaml ou de Python.

Comme dans la plupart des langages, les appels de fonctions en C sont *imbriqués* : si une fonction `f` appelle `g`, alors `g` s'exécute entièrement avant le retour de `f`. Cette propriété permet une implémentation efficace à l'aide d'une *pile d'appels*.

Lorsqu'une fonction `f` est appelée, ses informations d'exécution (paramètres, adresse de retour, variables locales, etc.) sont placées au sommet de la pile dans un *tableau d'activation* (ou *stack frame*). Si `f` appelle `g`, un nouveau tableau d'activation est empilé au-dessus. Quand `g` se termine, son tableau est dépilé et l'exécution reprend dans `f`. La taille de la pile reflète donc la profondeur d'imbrication des appels.

Le compilateur C répartit la mémoire d'un programme comme suit :

— le code exécutable occupe les adresses basses ;
— les données statiques (connues à la compilation, comme les chaînes constantes) se situent juste au-dessus ;
— le reste de la mémoire est utilisé dynamiquement :
  — en haut, la *pile d'appels*, qui croît vers les adresses basses ;
  — en bas, le *tas*, réservé aux allocations via `malloc`.

Cette organisation empêche toute interférence entre pile et tas.

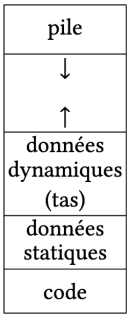



FIGURE 1 – Organisation de la mémoire d'un programme en C.

**Mode de passage.** En C, le passage des arguments se fait *par valeur*. Lors d'un appel, les arguments sont évalués, puis leurs valeurs copiées dans les variables locales de la fonction.

```
void incr(int x) {
x = x + 1;
}
void f() {
int v = 41;
incr(v);
// v vaut toujours 41
}
```

Ici, `incr` reçoit une copie de `v` dans la variable locale `x`. L'incrémentation modifie `x`, pas `v`. À la fin de la fonction, `x` disparaît et la valeur de `v` reste inchangée. Les deux variables occupent donc des zones mémoire distinctes.

Les noms `v` et `x` ne sont pas stockés en mémoire : le compilateur leur associe simplement des emplacements (pile, registre, etc.) et produit le code correspondant. Les noms n'existent que pour notre compréhension.

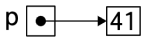
**Débordement de la pile.** La pile d'appels a une taille limitée (environ 8 Mo sous Linux). Un nombre excessif d'appels imbriqués provoque un *débordement de pile* (*stack overflow*), entraînant une *erreur de segmentation* et l'arrêt du programme. Ce phénomène traduit souvent une récursion non terminante due à l'absence de cas de base.

# 2 Pointeurs, tableaux et structures

Pour construire des données arbitrairement grandes, le langage C fournit des tableaux et une notion de types enregistrements appelés structures. Avant de les présenter, on commence par les pointeurs, car pointeurs et tableaux sont intimement liés dans le langage C.

## 2.1 Pointeurs

Un *pointeur* est une variable (ou expression) dont la valeur est une adresse mémoire contenant une donnée d'un certain type. Par exemple, si `p` contient l'adresse d'un entier valant 41, on représente cette situation ainsi :




En pratique, `p` contient une adresse sur 8 octets (sur une machine 64 bits), par exemple `0x7fff770117cc`, mais la valeur exacte importe peu.

L'opérateur `*p` permet d'accéder à la valeur pointée : on dit qu'on *déréférence* le pointeur.

```
printf("%d\n", *p); // affiche 41
*p = 42; // modifie la valeur pointee
```

Après cette affectation, la variable `p` contient toujours la même adresse, mais la valeur stockée à cet emplacement est devenue 42.

Le type d'un pointeur vers un entier est `int*`. En général, un pointeur `p` vers une valeur de type $\tau$ a le type $\tau*$, et l'expression `*p` est de type $\tau$.
On peut déclarer un pointeur de la manière suivante :

```
int* p; // ou int *p;
```

Les deux formes sont équivalentes, mais la seconde est préférable. Elle se lit naturellement comme "`*p` est un entier" et évite les pièges tels que :

```
int* x, y; // declare un pointeur x et un entier y
```

### Exemple

Voici une fonction qui incrémente l'entier pointé par `x` :

```
void incr(int *x) {
  *x = *x + 1;
}
```

On peut l'appeler avec un pointeur `p` en écrivant `incr(p)`.

### Création de pointeurs

Une première manière d'obtenir un pointeur consiste à allouer dynamiquement de la mémoire :

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

`malloc` réserve sur le tas un espace suffisant pour un entier et renvoie un pointeur `void*` converti ici en `int*`.
Pour vérifier que l'allocation de mémoire a réussi, on utilise `assert(p != NULL)`.
Une autre méthode consiste à utiliser l'opérateur `&`, qui renvoie l'adresse d'une variable :

```
void f() {
  int v = 41;
  incr(&v); // v vaut maintenant 42
}
```

Ici, dans `incr` on a un pointeur vers la variable `v`, allouée sur la pile.

### Application

Le langage C n'a pas de type de $n$-uplet, et une fonction ne peut renvoyer qu'une seule valeur. Pour retourner plusieurs résultats, on utilise des pointeurs.
Par exemple, pour ecrire une fonction qui calcule quotient et reste de la division euclidienne renvoie le quotient et écrit le reste dans l'adresse donnée par `r` :

```
int division(int a, int b, int *r);
int r;
int q = division(89, 3, &r);
```

On obtient alors le quotient dans `q` et le reste dans `r`.

### Pointeur nul

Le pointeur `NULL` ne désigne aucun emplacement mémoire valide. Le déréférencer provoque une erreur de segmentation. Il reste toutefois utile, notamment dans les structures chaînées. On teste l'égalité avec `NULL` via `==`.
On sait déjà que les constructions `if` et `while` permettent de tester si un entier est non nul en écrivant directement `if (n)` plutôt que `if (n != 0)`. De même, on peut écrire `if (p)` à la place de `if (p != NULL)`, mais nous préférons la forme explicite, plus claire.

**Attention**

La manipulation des pointeurs est dangereuse : le C ne protège pas contre le déréférencement d'un pointeur `NULL`, invalide ou de type incorrect. Exemple :

```
1 int* f() {
2   int v = 41;
3   return &v; // ERREUR : v disparait a la fin de la fonction
4 }
```

Ce code crée un *pointeur fantôme* vers une zone de pile réutilisée ultérieurement, entraînant des comportements indéfinis ou des plantages.

### Le type `void*`

Bien que `void` ne soit pas un type, `void*` représente un pointeur vers une valeur de type inconnu. La fonction `malloc` renvoie une valeur de ce type :

```
1 int *x = malloc(sizeof(int));
```

Le C autorise la conversion implicite entre `void*` et tout autre type de pointeur. Ce mécanisme sert à écrire du code polymorphe (par exemple des fonctions sur des tableaux génériques), mais nous nous limiterons ici au code monomorphe, réservant le polymorphisme à OCaml.

## 2.2 Tableaux

Comme dans la plupart des langages, un *tableau* est une suite de valeurs stockées consécutivement en mémoire, accessibles par un indice. En C, les indices commencent à 0. On peut allouer un tableau local à une fonction ainsi :

```
1 void f() {
2   int a[4];
3   a[1] = 41;
4 }
```

Ici, `a` est un tableau de quatre entiers, local à `f`, donc alloué sur la pile. Son contenu n'est pas initialisé. Sur une machine à entiers 32 bits, il occupe 16 octets. La variable `a` désigne l'adresse de la première case : le premier élément est à `a`, le second à `a + 4`, etc.

L'expression `a[i]` accède à la case `i` du tableau, pour $0 \leq i < n$, où `n` est la taille du tableau. Le C ne contrôle pas ces bornes : un accès hors limites, en lecture ou écriture, produit un comportement indéfini.

On peut initialiser un tableau dès sa déclaration :

```
int b[3] = { 34, 55, 89 };
```

ou laisser le compilateur déduire la taille :

```
int b[] = { 34, 55, 89 };
```

Pour initialiser toutes les cases à 0 :

```
int b[10] = { 0 };
```

La taille d'un tableau n'est jamais stockée en mémoire. Même connue du compilateur, elle ne protège pas contre un accès hors bornes.

**Passage en paramètre.** Lorsqu'un tableau est passé à une fonction, c'est en réalité son *adresse* qui est transmise.

Exemple :

```
void incra(int x[]) {
  x[1] = x[1] + 1;
}
void f() {
  int a[4];
  a[1] = 42;
  incra(a);
}
```

On peut préciser une taille dans la déclaration :

```
void incra(int x[4]) { }
```

mais cette information est purement décorative : le compilateur traite `int x[4]` et `int x[]` de la même façon, c'est-à-dire comme un pointeur vers le premier élément du tableau.

En C, `a[i]` n'est qu'un raccourci pour `*(a + i)`, soit le déréférencement du pointeur décalé de `i` éléments.

Si la taille du tableau est nécessaire à la fonction, il faut la passer explicitement :

```
void reset(int a[], int n) {
  for (int i = 0; i < n; i++) {
    a[i] = 0;
  }
}
```

**Tableaux dynamiques.** On peut aussi allouer un tableau sur le tas avec `calloc` :

```
int *a = calloc(4, sizeof(int));
a[1] = 41;
```

Le premier argument indique le nombre d'éléments, le second la taille de chaque élément (ici `sizeof(int)`). `calloc` renvoie un pointeur vers la zone allouée, ici stocké dans `a`. Ce pointeur peut être utilisé comme un tableau ou passé à une fonction prenant un pointeur : c'est équivalent.

**Tableaux multidimensionnels.** Pour créer un tableau à plusieurs dimensions, par exemple une matrice de 5 lignes et 8 colonnes d'entiers, on peut utiliser un tableau de 5 pointeurs, chacun pointant vers un tableau de 8 entiers. L'allocation dynamique se fait ainsi :

```
int **mat = calloc(5, sizeof(int*));
for (int i = 0; i < 5; i++) {
  mat[i] = calloc(8, sizeof(int));
}
```

On accède à l'élément $(i, j)$ avec `mat[i][j]`. L'expression `mat[i]` renvoie le pointeur vers la ligne `i`, et `[j]` sélectionne le `j`-ième entier de cette ligne.

Le C propose également une forme primitive de tableau multidimensionnel, déclarée directement par :

```
int mat[5][8];
```

Cela alloue sur la pile un bloc de $5 \times 8$ entiers contigus, organisés par ligne. Contrairement au cas précédent, il ne s'agit pas d'un tableau de pointeurs, mais d'un seul bloc mémoire. L'accès à `mat[i][j]` reste le même, mais le compilateur calcule l'adresse par un décalage de $4 \times (8 \times i + j)$ octets. C'est le type de `mat` qui détermine si le tableau est un bloc continu ou un ensemble de pointeurs.

Lorsqu'on passe un tableau multidimensionnel à une fonction, il faut indiquer toutes les dimensions sauf la première, pour permettre le calcul des adresses :

```
void f(int mat[5][8]) {
  ...
}
```

La première dimension (ici 5) est optionnelle et sert de documentation, mais les suivantes (ici 8) sont obligatoires pour le compilateur.

Pour des tableaux dont les dimensions ne sont pas connues statiquement, on utilisera plutôt la version dynamique avec pointeurs, en passant les tailles en paramètres :

```
void f(int n, int m, int **mat) { ... }
```

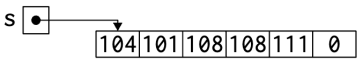
Ici, `mat` est supposé être un tableau de `n` pointeurs, chacun représentant un tableau de `m` entiers.

## 2.3 Chaînes de caractères

En C, une chaîne de caractères n'est rien d'autre qu'un tableau de caractères, de type `char`, dont le dernier élément est le caractère nul `'\0'` de code 0. Ainsi, si on déclare :

```
char *s = "hello";
```

on se retrouve avec la situation suivante :




où chaque caractère est stocké sur un octet.

La bibliothèque C fournit notamment les opérations suivantes :

— `strlen(s)` renvoie la longueur de la chaîne `s`. Comme pour un tableau, la longueur n'est pas stockée en mémoire. Mais à la différence d'un tableau, la présence du caractère nul permet le calcul de la longueur.
— `strcmp(s1, s2)` compare les deux chaînes `s1` et `s2` pour l'ordre lexicographique, en renvoyant un résultat strictement négatif, nul ou strictement positif. En particulier, l'expression `strcmp(s1, s2) == 0` teste l'égalité des deux chaînes. La variante `strncmp(s1, s2, n)` se limite à au plus `n` caractères.
— `atoi(s)` convertit la chaîne `s` en un entier de type `int`, sans chercher à détecter d'erreur.

D'autres fonctions permettent de modifier une chaîne en place, par exemple en l'écrasant par une autre (`strcpy`) ou encore en lui concaténant une autre chaîne (`strcat`). Il faut pour cela que l'espace de destination, préalloué, soit assez grand.

Les chaînes de caractères littérales contenues dans le code source, comme `"hello"` ci-dessus, sont préallouées par le compilateur (dans le segment de données) et sont immuables. Ce caractère immuable permet notamment au compilateur d'allouer une seule occurrence de chaque chaîne distincte. Une erreur classique lorsque l'on débute avec C consiste à comparer deux chaînes avec `==` plutôt qu'avec `strcmp`. Mais si on utilise uniquement des chaînes littérales dans les tests, on risque de passer à côté de cette erreur du fait du partage réalisé par le compilateur.

**Constantes.**

On peut ajouter le qualificatif `const` devant la déclaration d'un tableau ou d'une chaîne. Cela signifie alors que les éléments ne peuvent être modifiés. C'est typiquement le cas pour spécifier qu'un paramètre de fonction ne va pas être modifié. Ainsi, la fonction `strcpy` dont nous venons de parler se déclare avec le type :

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

pour préciser que la chaîne `src` n'est pas modifiée par la fonction.

## 2.4 Structures

Une *structure* agrège plusieurs valeurs, appelées *champs*, chacune ayant un nom et un type. On déclare par exemple une structure `S` composée de deux champs entiers `a` et `b` :

```
1 struct S { int a; int b; };
```

Ce code définit un nouveau type `struct S`. Une variable locale de ce type est allouée sur la pile :

```
1 void f() {
2   struct S s = { .a = 1, .b = 2 };
3   ...
4 }
```

On accède à un champ par `structure.champ` et on le modifie par affectation :

```
1 s.b = 3;
2 printf("b = %d\n", s.b);
```

**Passage et copie.** Lorsqu'une structure est passée en paramètre, elle est entièrement copiée :

```
void incr1(struct S x) {
  x.b = x.b + 1;
}
void f() {
  struct S s = { .a = 1, .b = 2 };
  incr1(s);
  // s.b vaut toujours 2
}
```

Ici, le champ `b` de `x` est modifié, mais la structure `s` reste inchangée, car copiée. Le même comportement vaut pour l'affectation.

**Pointeurs vers structures.** Pour éviter les copies coûteuses, on passe souvent un pointeur vers une structure. Si `x` est de type `struct S *`, on accède à un champ avec `(*x).b`, ou plus simplement `x->b` :

```
void incr2(struct S *x) {
  x->b = x->b + 1;
}
void f() {
  struct S s = { .a = 1, .b = 2 };
  incr2(&s);
  // s.b vaut maintenant 3
}
```

**Allocation dynamique.** Une structure peut être allouée sur le tas avec `malloc`, en réservant `sizeof(struct S)` octets :

```
void g() {
  struct S *s = malloc(sizeof(struct S));
  s->a = 1;
  s->b = 2;
  incr2(s);
  free(s);
}
```

**Structures imbriquées.** Les structures peuvent contenir d'autres structures comme champs :

```
struct T { int c; struct S d; int e; };
void f() {
  struct T t = { .c = 1, .d = { .a = 2, .b = 3 }, .e = 4 };
}
```

La représentation est *plate* : ici, quatre entiers consécutifs. En revanche, si le champ est un pointeur vers une structure, la mémoire n'est plus contiguë :

```
struct T { int c; struct S *d; int e; };
void f() {
```

```
  struct S *s = malloc(sizeof(struct S));
  s->a = 1; s->b = 2;
  struct T t = { .c = 1, .d = s, .e = 4 };
}
```

Le type et l'organisation mémoire diffèrent.

**Alias de types.** Pour éviter d'écrire `struct S` à chaque fois, on peut définir un raccourci avec `typedef` :

```
typedef struct S s;
```

On peut ensuite utiliser simplement `s` comme type.

### Valeurs gauches

Dans une affectation `e1 = e2`, l'expression `e1` désigne un emplacement mémoire modifié par l'opération. Une telle expression est appelée *valeur gauche* (*lvalue*). Le C admet trois formes de valeurs gauches :

- une variable ;
- une expression `*e` ;
- une expression `e.x`, où `e` est une valeur gauche.

Deux autres formes équivalentes s'y ajoutent :

- `e->x`, équivalente à `(*e).x` ;
- `e[i]`, équivalente à `*(e+i)`.

Toute affectation sur une expression non gauche provoque une erreur :

```
error: lvalue required as left operand of assignment
```

L'opérateur `&` exige également une valeur gauche comme opérande : on ne peut donc obtenir l'adresse que d'une variable, d'un élément de tableau ou d'un champ de structure. Sinon, le compilateur signale :

```
error: lvalue required as unary '&' operand
```