# Le langage C

## Table des matières



Le langage C est un langage de bas niveau, inventé en 1972 lors du développement du système d'exploitation Unix. Ce langage étant proche de la machine, la compilation d'un programme C en assembleur, seul langage directement compris par le processeur, est relativement aisée. Le C est un langage assez minimaliste et n'est pas spécialisé dans un domaine applicatif. Si on ajoute le fait qu'il est disponible sur toutes les plateformes, on comprend qu'il soit devenu si populaire. Le langage C est défini très précisément par un standard, qui possède plusieurs versions successives : C89, C99, C11, C18. Conformément au programme de MP2I, nous utiliserons dans ce cours le standard C99.

# 1 Premiers pas

## 1.1 Généralités

Depuis le livre « The C programming language » écrit par B. Kernighan et D. Ritchie, qui ont inventé le langage C, la tradition est d'écrire un premier programme qui affiche `hello, world` sur la console. Pour cela, commençons par créer le fichier text `hello.c` contenant le programme suivant.

```c
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

La première ligne de ce programme permet d'avoir accès aux fonctions de la bibliothèque `stdio` (bibliothèque standard entrées/sorties). Elle définit ensuite une fonction appelée `main` qui affiche le texte demandé à l'aide de la fonction `printf` de la bibliothèque `stdio`. Cette fonction renvoie enfin l'entier 0, qui est une convention en C pour signifier que l'exécution du programme s'est bien déroulée. Afin de compiler ce programme, nous ouvrons un terminal et après s'être déplacé dans le dossier contenant notre programme, on écrit.

```
$ gcc hello.c -o hello
```

On a demandé au compilateur de générer un exécutable `hello` à partir du programme `hello.c`. Une fois le programme compilé, on peut lancer cet exécutable grâce à la commande

```
$ ./hello
hello, world
```

Dans la suite du cours, conformément au programme de MP2I, nous supposerons que les bibliothèques suivantes sont chargées au début de chaque fichier.

```c
#include <assert.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
```

Il conviendra donc de rajouter ces lignes en en-tête de vos fichiers afin que les exemples de ce cours puissent compiler sur une machine.

Par rapport au langage Python, signalons quelques différences.

— Un programme C contient toujours une fonction appelée `main`. L'exécution du programme revient à appeler cette fonction.

— En C, les retours à la ligne et l'indentation ne sont pas significatifs. Bien entendu, les programmes étant lus par des humains, il vous sera demandé d'écrire du code dont l'indentation met en valeur la structure du code. La plupart des éditeurs de code permettent une indentation automatique et certains programmes comme `clang-format` permettent d'indenter un fichier en accord avec la structure du code.

— Contrairement au Python, le langage C est compilé et non interprété. Autrement dit, lors de la compilation, le programme est traduit en langage machine. Il est ensuite exécuté. Cela a plusieurs conséquences.

  — Le code machine généré par le compilateur C est extrêmement efficace. Dans de nombreux cas, un programme C est de l'ordre de 100 fois plus rapide qu'un programme Python.

  — Des erreurs peuvent être détectées à la compilation. Par exemple, si on avait écrit un `print` à la place de `printf`, le compilateur aurait refusé de compiler le programme en évoquant le fait qu'une telle fonction n'existe pas. Plus généralement, si on fait référence à une variable qui n'existe pas, si on appelle une fonction avec le mauvais nombre d'arguments avec les mauvais types, alors le compilateur échouera, même si l'erreur se trouve dans une partie du code qui ne sera jamais exécutée. On appelle cela le *typage statique*. De son côté, un programme Python possèdant le même type d'erreur échouera lors de l'exécution au moment ou l'instruction erronée est atteinte. On parle de *typage dynamique*. Plus généralement, l'adjectif « statique » fait référence à tout ce qui est connu avant l'exécution d'un programme, notamment au moment de sa compilation. L'adjectif « dynamique » fait quant à lui référence à ce qui n'est connu que pendant l'exécution d'un programme.

## 1.2 Types et opérations élémentaires

Une des particularités du C est que certaines opérations ont un comportement qui est non défini (*undefined behavior* en anglais). Contrairement à ce qui se passe en Python où une division par 0 génère une exception, une division par 0 en C peut avoir des conséquences très variées. Le programme peut s'arrêter de manière brutale, mais peut aussi continuer en ayant un comportement totalement erratique. C'est au programmeur de s'assurer qu'aucune instruction de son programme n'aura de comportement qui n'est pas défini. En échange, le compilateur peut supposer que ces opérations n'arriveront jamais et optimiser le code produit de manière agressive.

### 1.2.1 Les entiers

Il existe de nombreux types d'entiers définis en C qui diffèrent selon leur taille et leur caractère signé ou non.

— *Entiers non signés.* Rappelons qu'un type entier non signé, codé sur $n$ bits, peut représenter tous les entiers de l'ensemble $E \coloneqq [\![0, 2^n - 1]\!]$. Les opérations usuelles `+`, `-` et `*` sont définies modulo $2^n$. Autrement dit, si le résultat d'une de ces opérations n'est pas dans $E$, il est remplacé par le reste de sa division euclidienne par $2^n$. Par exemple, si $n = 8$, alors $E = [\![0, 255]\!]$ et `255 + 1` s'évalue en `0`. L'opération `/` représente la division entière et la division par 0 est « undefined behavior ». L'expression `a % b` est utilisé pour calculer $a$ modulo $b$ lorsque $b$ est non nul. Le C définit les types non signés `uint8_t`, `uint16_t`, `uint32_t` et `uint64_t` qui sont respectivement codés sur 8, 16, 32 et 64 bits.

| type | valeur minimale | valeur maximale |
|---|---|---|
| `uint8_t` | 0 | 255 |
| `uint16_t` | 0 | 65 535 |
| `uint32_t` | 0 | 4 294 967 295 |
| `uint64_t` | 0 | 18 446 744 073 709 551 615 |

— *Entiers signés.* Un type entier signé codé sur $n$ bits, peut représenter tous les entiers de l'ensemble $E \coloneqq [\![-2^{n-1}, 2^{n-1}-1]\!]$. Les opérations usuelles `+`, `-`, `*` ne sont définies que lorsque le résultat est dans $E$. Si on effectue une opération dont le résultat n'est pas dans $E$ (on parle de débordement arithmétique), le programme entre en « undefined behavior ». Concernant la division, `a / b` et `a % b` ne sont définis que lorsque $b$ est strictement positif. La division entière se fait en arrondissant le nombre vers 0, ce qui donne lieu à un comportement différent de Python. Par exemple $(-3)/2$ s'evalue en $-1$. Le lien avec la division euclidienne est donc plus délicat. Si $a \in \mathbb{Z}$ et $b \in \mathbb{N}^*$ et $q, r \in \mathbb{Z}$ désignent respectivement `a / b` et `a % b`, on a toujours $a = qb + r$, mais :
  — Si $a \geqslant 0$, alors $q = \lfloor a/b \rfloor$ donc $r \in [\![0, b[\![$. On en déduit que $q$ et $r$ sont respectivement le quotient et le reste de la division euclidienne de $a$ par $b$.
  — Sinon, $a < 0$. Dans ce cas, $q = \lceil a/b \rceil$ donc $r \in ]\!]-b, 0]\!]$.
    — Si $r = 0$, alors $r \in [\![0, b[\![$ donc $q$ et $r$ sont le quotient et le reste de la division euclidienne de $a$ par $b$.
    — Si $r < 0$, alors $a = (q-1)b + (b+r)$ et $0 < b + r < b$. Donc $q - 1$ est le quotient de la division euclidienne de $a$ par $b$ et $b + r$ est son reste.

  Le C définit les types signés `int8_t`, `int16_t`, `int32_t` et `int64_t`, codés respectivement sur 8, 16, 32 et 64 bits.

| type | valeur minimale | valeur maximale |
|---|---:|---:|
| `int8_t` | -128 | 127 |
| `int16_t` | -32 768 | 32 767 |
| `int32_t` | -2 147 483 648 | 2 147 483 647 |
| `int64_t` | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 807 |

Lorsque la taille de l'entier n'est pas vraiment importante, nous utiliserons les types `int` et `unsigned int` qui codent des entiers de même taille, respectivement signés et non signés. Sur la plupart des plateformes 32 et 64 bits avec lesquelles nous travaillerons, ces entiers sont codés sur 32 bits.

En plus des opérations arithmétiques usuelles en mathématiques (addition, multiplication, division), il est naturel de définir des opérations *bit à bit* (ou *bitwise*) qui opèrent directement sur la représentation binaire : en effet, pour un processeur, ces opérations sont très simples à réaliser (beaucoup plus qu'une division par exemple).

| **Opération** | **Notation C** | **Exemple** |
|---|---|---|
| Conjonction | `&` | 00001101<br>`&` 00101001<br>00001001 |
| Disjonction | `\|` | 00001101<br>`\|` 00101001<br>0101101 |
| Ou exclusif | `^` | 00001101<br>`^` 00101001<br>00100100 |
| Négation | `~` | `~` 00101001<br>11010110 |
| Décalage à gauche | `n << i` | `0b1001 << 3 = 0b1001000` |
| Décalage à droite | `n >> i` | `0b1001011 >> 3 = 0b1001` |

Opérations bit à bit

Quelques remarques sur ces opérations :
— L'argument `i` des décalages doit obligatoirement être positif ou nul et inférieur ou égal à la largeur du type.
— L'opération `n << i` décale la représentation binaire de `n` de `i` chiffres vers la gauche en rajoutant des zéros à droite. S'il n'y a pas de dépassement de capacité, cela revient à multiplier $n$ par $2^i$.
— L'opération `n >> i` décale la représentation binaire de `n` de `i` bits vers la droite. Si `n` est un entier non signé, ou signé et positif, cela revient à calculer $\lfloor n/2^i \rfloor$. Si `n` est négatif, nous n'utiliserons jamais cette opération.
— Évitez de remplacer tous les `n = n * 2` par des `n = n << 1` en espérant gagner en performance : le compilateur est assez grand pour faire ça tout seul.
— Le résultat d'une négation dépend de la largeur de l'entier ! Il faut être très prudent si l'on utilise cette opération.

### 1.2.2 Les booléens

Le type des booléens est `bool` et ses deux valeurs sont `true` et `false`. On peut comparer les valeurs numériques avec les opérations `==`, `!=`, `<`, `>`, `<=` et `>=`. Les opérations logiques disponibles sont `!` (négation), `&&` (conjonction) et `||` (disjonction). Ces deux dernières opérations sont paresseuses. Autrement dit, elles n'évaluent la seconde opérande que lorsque l'évaluation de la première opérande ne suffit pas pour déterminer le résultat.

### 1.2.3 Les flottants

Le C fournit un type `double` qui correspond sur la plupart des plateformes à un nombre flottant 64 bits, conforme à la norme IEEE 754. Les opérations se notent `+`, `-`, `*` et `/` comme pour l'arithmétique entière. Ce sont cependant des opérations différentes. On parle alors de *surcharge* d'opérateur. Comme en Python, les opérations sur ces nombres flottants nécessitent de faire des arrondis ce qui rend leur arithmétique délicate.

### 1.2.4 Les caractères et les chaines de caractères

Le type des caractères est `char`. Il s'agit d'un type codé sur un octet (8 bits). En pratique, on l'utilise uniquement pour représenter les caractères de la table ASCII. Les caractères se notent entre apostrophes : `'f'`, `'+'`, etc. Le caractère \ est une séquence d'échappement qui permet d'écrire les caractères spéciaux suivants.

| Caractère | Signification |
|---|---|
| `'\n'` | retour charriot |
| `'\t'` | tabulation |
| `'\0'` | caractère nul |

Les séquences `'\\'` et `'\''` permettent de désigner respectivement les caractères \ et '. Une chaîne de caractère est notée entre guillemets, comme `"hello, world"`. Les mêmes séquences d'échappement que pour les caractères peuvent être utilisées, ainsi que `\"` pour noter le caractère ". Le type des chaînes de caractères est `char *`.

## 1.3 Structure de contrôle

### 1.3.1 Variables et blocs

En C, les variables ne peuvent contenir que des valeurs d'un type déterminé lors de leur déclaration.

— *Déclaration* : Il est possible de déclarer une variable sans l'initialiser. Cependant, l'utilisation de la valeur d'une variable avant de l'avoir initialisée est « undefined behavior ».

```
int a;
// Pour l'instant, on ne peut pas utiliser la valeur de a
a = 42;
```

Il est possible de déclarer plusieurs variables d'un même type sur une même ligne. Par exemple

```
int a, b;
```

déclare deux variables `a` et `b`.

— *Définition* : Il est plus courant cependant de déclarer et d'initialiser une variable sur une même ligne ; on parle alors de définition. Pour cela, on utilise la syntaxe suivante.

```
int a = 42;
a = 2 * a;
```

Si l'on souhaite empêcher la modification ultérieure d'une variable, on utilise le mot clé `const`.

```
const int a = 42;
```

Une séquence d'instructions peut être regroupée dans un *bloc*, délimité par des accolades. Un bloc peut contenir la déclaration d'une ou plusieurs variables, leur portée s'étendant jusqu'à la fin du bloc dans lequel elle est déclarée. Étant considéré comme une instruction, un bloc peut contenir des sous-blocs. Une variable visible dans un bloc est visible dans un sous-bloc.

```
{
    int a = 42;
    {
        int b = 2 * a;
        b = b + 1;
        // Ici, a et b sont visibles
    }
    // Ici, seul a est visible
}
// Ici, ni a ni b ne sont visibles
```

Étant donné que les instructions du type `a = a + b` sont très courantes, on peut les écrire aussi `a += b`. L'écriture `a += 1` est d'ailleurs si courante qu'on peut aussi l'écrire `a++` ou `++a`. De la même manière, il est possible de décrémenter une variable avec les instructions `a--` et `--a`. Remarquons que les instructions `a -= b`, `a *= b` et `a /= b` sont disponibles.

### 1.3.2 Conditionnelle

Une conditionnelle est introduite avec le mot clé `if` et une expression booléenne entre parenthèses représentant une condition. Si cette expression s'évalue en `true`, l'instruction qui suit est exécutée.

```
if (condition) instruction
```

Le plus souvent, même lorsque l'instruction est élémentaire, on utilise un bloc d'instruction. Par exemple, le code suivant transforme l'entier $n$ en sa valeur absolue.

```
if (n < 0) {
    n = -n;
}
```

La forme la plus générale d'une conditionnelle est la suivante.

```
if (condition) instruction1 else instruction2
```

Si `condition` s'évalue en `true`, la première instruction est exécutée. Sinon, c'est la seconde instruction qui est exécutée. Par exemple si `n` est un terme de la suite de Syracuse, le prochain terme est calculé de la manière suivante.

```
if (n % 2 == 0) {
    n = n / 2;
} else {
    n = 3 * n + 1;
}
```

Il est possible d'enchainer les `if`. Par exemple, le code suivant affiche le nombre de solutions réelles du trinôme $P \coloneqq aX^2 + bX + c \in \mathbb{Z}[X]$.

```
int delta = b * b - 4 * a * c;
int nb_solutions;
if (delta > 0) {
    nb_solutions = 2;
} else if (delta == 0) {
    nb_solutions = 1;
} else {
    nb_solutions = 0;
}
```

Il existe également une construction d'expression conditionnelle, qui se note

```
condition ? expression1 : expression2
```

Si `condition` est évalué en `true`, c'est `expression1` qui est évalué et c'est sa valeur qui est utilisée. Dans le cas contraire, c'est `expression2` qui est évalué et sa valeur qui est utilisée. Par exemple

```
int x = a > b ? a : b;
```

permet d'initialiser `x` à la valeur du maximum de $a$ et $b$.

Remarquons qu'en C, il est possible de remplacer une condition par une valeur numérique. Cette valeur est alors considérée comme vraie lorsqu'elle est non nulle. Ainsi, si `n` est une variable de type `int`, dans le code

```
if (n) instruction
```

l'instruction sera executée uniquement lorsque $n$ sera non nul. Les booléens sont d'ailleurs des valeurs numériques qui valent 1 pour `true` et 0 pour `false`. Attention, l'utilisation d'une valeur numérique en lieu et place d'un booléen n'est pas autorisée par le programme de MP2I. Cependant, c'est quelque chose d'idiomatique en C que vous rencontrerez parfois dans des programmes.

### 1.3.3 Boucle while

Une boucle conditionnelle est introduite avec le mot clé `while` et son test s'écrit entre parenthèses, comme pour une conditionnelle.

```
while (condition) instruction
```

Par exemple, si on souhaite parcourir tous les entiers de 0 à $n-1$ avec une variable `i`, on peut le faire de façon élémentaire avec une boucle `while` comme ceci.

```c
int i = 0;
while (i < n) {
    [corps de boucle]
    i++;
}
```

Cependant, il existe une construction plus idiomatique pour cela, en utilisant le mot-clé `for`.

```c
for (int i = 0; i < n; i++) {
    [corps de boucle]
}
```

Notons que la portée de la variable `i` est limitée au corps de la boucle. La construction

```c
for (initialisation; condition; incrementation) instruction
```

est donc équivalente à

```c
{
    initialisation
    while (condition) {
        instruction
        incrementation
    }
}
```

Que ce soit dans une boucle `while` ou dans une boucle `for`, l'instruction `break` permet de sortir de la boucle et l'instruction `continue` permet de sauter immédiatement à l'itération suivante. Bien entendu, dans le cas d'une boucle `for`, l'instruction d'incrémentation est effectuée avant de passer à l'itération suivante.

**Exercice 1**

⇨ Si `n` est un entier non signé, non nul, que va faire le code suivant ?

```c
for (unsigned int i = n - 1; i >= 0; i--) {
    ...
}
```

## 1.4 Fonction

Une fonction est définie en donnant son type de retour, son nom et la liste de ses paramètres avec leurs types. Le corps de la fonction est un bloc. Voici par exemple une fonction quotient prenant en entrée deux paramètres `a` et `b` de types `int` et renvoyant un résultat de type `int`. Si $a \geqslant 0$, et $b > 0$, le résultat renvoyé est le quotient de la division euclidienne de $a$ par $b$.

```c
int quotient(int a, int b) {
    int q = 0;
    while (a >= b) {
        a -= b;
        q++;
    }
    return q;
}
```

La fonction renvoie un résultat avec l'instruction `return`. Ici, on renvoie la valeur de la variable `q`. L'instruction `return` peut apparaître à plusieurs endroits de la fonction, y compris à l'intérieur d'une branche conditionnelle ou d'une boucle.

Si une fonction ne renvoie pas de valeur, elle est déclarée avec le mot clé `void` à la place du type de retour.

```c
void say_hello(bool is_morning) {
    if (is_morning) {
        printf("Good morning.\n");
    } else {
        printf("Good afternoon.\n");
    }
}
```

Attention, `void` n'est pas un type. En particulier, on ne peut pas déclarer une variable de type `void`. Il est possible de sortir de telles fonctions, soit en utilisant une instruction `return` sans argument, soit lorsque la fin du corps de la fonction est atteinte.

Il est aussi possible de définir des fonctions qui ne prennent aucun argument. Dans ce cas, on utilise aussi le mot clé `void` dans la liste de ses arguments.

```c
void say_hello_world(void) {
    printf("hello, world\n");
}
```

Attention à la construction `int f()` qui ne déclare pas une fonction qui ne prend aucun argument, mais plutôt une fonction qui prend un nombre variable d'arguments. La définition de telles fonctions n'est pas au programme de MP2I.

Une fonction peut être récursive. Par exemple, la fonction `factorial` peut être définie de la manière suivante.

```c
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

## 1.5 Le préprocesseur

Le langage C est muni d'un *préprocesseur*, qui permet notamment d'insérer d'autres fichiers dans notre fichier source, d'effectuer de la compilation conditionnelle et de définir des constantes. Le préprocesseur fait son travail avant le compilateur et n'a aucune connaissance du langage C : il se contente d'effectuer des transformations textuelles sur le fichier.

- *Include* : Il est possible d'insérer le contenu d'un fichier à l'aide de `#include <nom_du_fichier.h>`. Lorsque le nom du fichier est entre chevrons comme ici, le compilateur va chercher ce fichier dans les différents repértoires utilisés par les bibliothèques du système. Lorsqu'il est entre guillemets, il va le chercher dans le repértoire courant.
- *Constantes* : Il est possible de définir une constante à l'aide de l'instruction `#define SIZE 16`. Si l'on fait cela en en-tête de fichier, avant que le compilateur ne lise le fichier, chaque occurrence de `SIZE` sera remplacé par `16`. Cependant, on préférera, lorsque c'est possible, l'utilisation d'une variable `const int size = 16` à l'usage du processeur.

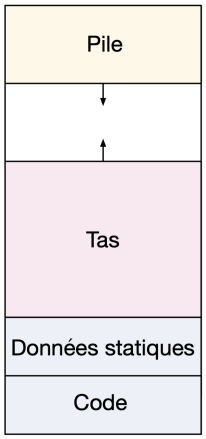
## 1.6 Modèle d'exécution

### 1.6.1 La pile et le tas

Afin de bien utiliser un langage, il est nécessaire de comprendre son modèle d'exécution. Cela est nécessaire d'une part pour la correction de nos programmes (cette donnée est-elle copiée ou partagée ?) mais également pour comprendre leur performance.

Comme dans la majorité des langages de programmation, les appels de fonctions en C sont imbriqués : si une fonction `f` appelle une fonction `g`, cet appel à la fonction `g` termine avant que l'appel à la fonction `f` ne termine. Cette propriété permet une compilation très efficace des appels de fonctions, en utilisant une *pile*, que l'on appelle *pile d'appels*. Lors de l'appel d'une fonction `f` le processeur doit stocker les données relatives à cet appel : l'adresse de retour, les paramètres, les variables locales, etc. L'ensemble de ces données forment ce qu'on appelle un *tableau d'activation* (stackframe en anglais) et sont placées en haut de la pile. Si le code de la fonction `f` vient à appeler la fonction `g`, le tableau d'activation de `g` viendra se placer au dessus de celui de celui de `f`. Lorsque l'appel à la fonction `g` se termine, le tableau d'activation de `g` est supprimé et on retrouve celui de `f` au sommet de la pile. En particulier, la pile est d'autant plus grande qu'il y a d'appels de fonctions imbriquées en cours d'exécution.

Le compilateur C va organiser les différents espaces mémoire nécessaires à l'exécution du programme selon le schéma ci-dessous.




Le code du programme est placé dans les adresses basses de la mémoire. Au dessus, on trouve les données statiques, c'est-à-dire les données connues au moment de la compilation comme par exemple une chaine de caractères contenue dans le code source. Le reste de la mémoire va être utilisée dynamiquement, c'est-à-dire pendant l'exécution du programme. Il se partage entre la pile d'appels, située en haut de la mémoire et le *tas*, qui est constitué de blocs de mémoire alloués avec la fonction `malloc` de la bibliothèque standard. Avec cette organisation, la pile n'interfère pas avec le tas. Notons que ce schéma nous donne l'impression qu'un programme utilise toute la mémoire pour lui seul. C'est grâce au mécanisme de *mémoire virtuelle*, qui traduit les adresses virtuelles que nous manipulerons dans nos programmes, en adresses physiques. La pile croit vers les adresses basses. Autrement dit, le fond de la pile est situé tout en haut de la mémoire, et le sommet de la pile est plus bas. Il est nécessaire de comprendre cela afin de comprendre les nombreuses illustrations données plus loin dans ce chapitre. En particulier, les adresses croissent vers le haut lorsque nous dessinerons des fragments de la pile.

Sur la plupart des systèmes, la pile d'appel a une taille maximale. Sous Linux, par exemple, par défaut, cette taille est de 8 Mo. Si on dépasse cette taille, on va provoquer ce qu'on appelle un débordement de pile (stackoverflow en anglais) ce qui entrainera un accès à une zone de mémoire qui n'est pas autorisée (erreur de segmentation, ou segmentation fault en anglais). Le système d'exploitation provoquera alors l'arrêt du programme. C'est souvent un phénomène que l'on observera lorsqu'une récursion ne termine pas.

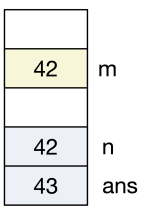
### 1.6.2 Passage par valeur

Le langage C fonctionne avec ce que l'on appelle un *passage par valeur*. Cela signifie que, lors d'un appel de fonction, non seulement les paramètres effectifs de l'appel sont d'abord évalués, mais aussi que ces valeurs sont *copiées* dans autant de nouvelles variables qu'il y a de paramètres avant que le code de la fonction ne soit exécuté. Illustrons cela avec les fonctions suivantes.

```c
int g(int n) {
    int ans = n + 1;
    return ans;
}

void f(void) {
    int m = 42;
    m = g(m);
    // m vaut 43
}
```

Juste avant l'exécution de `return ans`, voici l'état de la pile d'appels.




En particulier, puisque la valeur de `m` est copiée lors de l'appel de `g`, on en déduit que le code suivant

```c
void g(int n) {
    n = n + 1;
}
```

```c

void f(void) {
    int m = 42;
    g(m);
    // m vaut 42
}
```

ne change pas la valeur de `m`. Une telle fonction `g` est donc inutile.

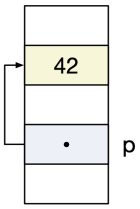
Il est aussi bon de comprendre que, contrairement à ce qui se passe en Python, les noms des variables `m`, `n` et `ans` ne sont nulle part représentés en mémoire. Le compilateur a déterminé des emplacements mémoire pour ces variables et il produit du code qui y fait référence. Ce sont uniquement nos schémas qui matérialisent ces noms de variables, afin de mieux comprendre l'évolution du programme.

# 2 Pointeur, tableau et structure

Afin de construire des données arbitrairement grandes, le langage C fournit des tableaux. Pour agréger ensemble des données de types différents, le C fournit ce qu'on appelle des structures. Mais nous commencerons par présenter les pointeurs qui sont intimement liés aux tableaux en C.

## 2.1 Pointeur

Un *pointeur* est une variable, et plus généralement une expression dont la valeur est une adresse mémoire. À cette adresse, se trouve la représentation binaire d'une valeur d'un certain type. Prenons par exemple l'exemple d'une variable `p` qui contient l'adresse d'un emplacement mémoire où se trouve la valeur `42` de type `int`. Dans ce cours, on se représente cette situation comme ceci.




En pratique, sur une machine 64 bits, la variable `p` contient une adresse mémoire stockée sur 8 octets. L'usage est de représenter cette valeur en hexadécimal. Un exemple de valeur de pointeur est `0x7ffee0b58774`. Notre schéma est une abstraction de cette réalité, car la valeur effective de l'adresse nous importe peu.

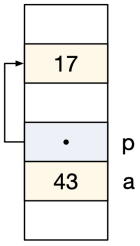
On accède à la valeur pointée par le pointeur `p` avec la construction `*p`. On dit qu'on *déréférence* le pointeur. Ainsi, l'instruction

```c
int a = *p + 1;
```

va initialiser la variable `a` avec la valeur 43. De même, on peut modifier la valeur pointée par `p` avec une affectation de `*p`. Par exemple, après l'instruction

```c
*p = 17;
```

nous serons dans la situation suivante.




Attention, la valeur de `p` n'a pas changée, puisqu'il pointe toujours vers la même adresse mémoire, mais c'est la valeur pointée qui a été modifiée.

Le type d'un pointeur vers un entier se note `int *`. Plus généralement, un pointeur `p` vers une valeur de type `T` a le type `T *` et l'expression `*p` a le type `T`. C'est pourquoi, le langage C a choisi la syntaxe suivante pour déclarer notre pointeur `p`.

```c
int *p;
```

Notons que comme les espaces entre `int`, `*` et `p` ne sont pas significatifs pour le compilateur C, on aurait pu aussi bien déclarer notre pointeur de la manière suivante.

```c
int* p;
```

Les programmeurs C ont tendance à utiliser la première notation alors que les programmeurs C++ ont plutôt tendance à utiliser cette seconde notation.

Voici un exemple de fonction qui prend en argument un pointeur `p` vers un entier et qui incrémente cet entier.

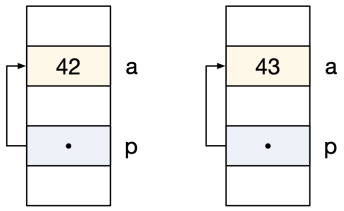
```c
void incr(int *p) {
    *p = *p + 1;
}
```

On peut lui passer notre pointeur `p` en argument en écrivant `incr(p)`.

Il nous reste à expliquer comment construire des pointeurs. Une première façon d'en obtenir est d'utiliser l'opérateur `&` du langage C. Pour une expression `e` de type `T` qui désigne un emplacement mémoire, `&e` est un pointeur de type `T *` vers cet emplacement mémoire. Voici par exemple une portion de code qui déclare une variable `a` qui vaut 42, et qui passe un pointeur vers cette variable à la fonction `incr` ci-dessus afin de l'incrémenter.

```c
int a = 42;
int *q = &a;
incr(q);
```

Et voici l'état de la pile juste avant puis, juste après l'incrémentation dans la fonction `incr`.




Le passage d'un pointeur à une fonction peut être utile pour communiquer certains résultats à l'appelant. En effet, le langage C ne possède pas nativement de $n$-uplet, et il n'est pas simple d'écrire une fonction renvoyant deux résultats. On pourrait renvoyer une structure comme nous le verrons plus loin, mais une autre solution consiste à passer un pointeur à la fonction pour que cette dernière écrive la réponse à cette adresse mémoire. Supposons par exemple que l'on souhaite écrire une fonction qui renvoie le résultat de la division euclidienne de $a \in \mathbb{Z}$ par $b \in \mathbb{N}^*$. La fonction

```c
int division_euclidienne(int a, int b, int *r) {
    int rem = a % b;
    int quo = a / b;
    if (a >= 0) {
        *r = rem;
        return quo;
    } else {
        if (rem == 0) {
            *r = 0;
            return quo;
        } else {
            *r = rem + b;
            return quo - 1;
        }
    }
}
```

permet cela. Si l'on souhaite obtenir le reste et le quotient de $a \in \mathbb{Z}$ par $b \in \mathbb{N}^*$, on appelle alors la fonction de la manière suivante.

```c
int r;
int q = division_euclidienne(a, b, &r);
```

Le quotient et le reste sont alors respectivement contenus dans les variables `q` et `r`.

Une autre manière d'obtenir des pointeurs est d'allouer de la mémoire sur le tas avec la fonction `malloc`. En écrivant

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

on demande l'allocation d'une nombre d'octets permettant de stocker un entier. Sur la plupart des plateformes, `int` représente un entier 32 bits, soit 4 octets et l'expression `sizeof(int)` s'évalue en 4. La fonction `malloc` de la bibliothèque standard alloue donc 4 octets sur le tas. Il est alors possible d'y stocker une valeur de la manière suivante :

```
*p = 42;
```

Lorsque nous n'avons plus besoin de cette mémoire, il est nécessaire de la désallouer à l'aide de la fonction `free`.

```
free(p);
```

Une fois cette désallocation effectuée, toute tentative de déréférencer le pointeur `p` est « undefined behavior ».

Il existe un pointeur spécial, appelé *pointeur nul* et noté `NULL`. Il ne pointe pas vers un emplacement mémoire valide. Toute tentative de le déréférencer est « undefined behavior » et se soldera le plus souvent par une erreur de segmentation et une interruption du programme. Le pointeur nul reste cependant utile et est utilisé en C comme une valeur spéciale signifiant l'absence de vrai pointeur. Par exemple, si `malloc` est incapable d'allouer la mémoire demandée, ce qui peut arriver lorsque l'ordinateur ne dispose plus de mémoire, il renverra le pointeur `NULL`. Il est possible de tester si un pointeur vaut `NULL` avec les opérations `==` et `!=`. Nous avons vu précédemment que les constructions `if` et `while` qui sont naturellement utilisées avec un booléen peuvent aussi être utilisées avec un pointeur. Le pointeur sera interprété comme étant vrai si il est non nul et comme étant faux si il est nul. Ainsi le code

```
if (p) instruction
```

est équivalent au code

```
if (p != NULL) instruction
```

Dans ce cours, conformément au programme, nous utiliserons la seconde forme, mais il est bon de savoir que la première construction est idiomatique en C et se rencontre fréquemment en dehors des classes préparatoires.

Remarquons enfin que la manipulation explicite des pointeurs rend le langage C à la fois puissant et dangereux. En particulier, toute tentative de déréférencer un pointeur qui n'est pas valide, comme le pointeur nul où un emplacement mémoire de la pile qui n'est plus valide, est « undefined behavior ». Par exemple, la fonction suivante

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

ne peut être qu'incorrectement utilisée. En effet, le code suivant

```
int *p = f();
*p = *p + 1
```

est « undefined behavior » puisque la fonction `f` renvoie un pointeur vers son tableau d'activation. Une fois que ce pointeur est assigné à `p`, on a quitté la fonction `f`, donc l'adresse vers laquelle le pointeur `p` pointe n'est plus valide. On parle alors de *pointeur fantôme*. L'utilisation d'un tel pointeur pourrait faire planter notre programme, mais pourrait aussi provoquer des comportements non souhaités, voire malicieux.

Notons enfin qu'il existe un type de pointeur particulier appelé `void *`. Bien que `void` ne soit pas un type à proprement parler, le type `void *` est utilisé pour désigner un pointeur vers une valeur dont on ne connait pas le type. La fonction `malloc` de la bibliothèque standard renvoie d'ailleurs un pointeur de ce type, mais les règles de typage du C permettent d'utiliser un tel pointeur là où une valeur d'un certain type de pointeur est attendue. Le code

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

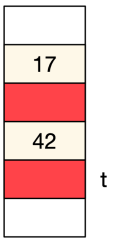
est donc accepté.

## 2.2 Tableau

Un tableau est un ensemble de valeurs stockées consécutivement en mémoire, auxquelles on peut accéder en temps constant avec un indice. En C, comme dans de nombreux langages de programmation, les tableaux sont indexés à partir de 0. On peut allouer un tableau localement à un bloc de la manière suivante

```
int t[4];
t[1] = 42;
t[3] = 17;
```

Juste après ces instructions, la pile est dans l'état suivant.




Dans cet exemple, on déclare un tableau de 4 entiers qui est alloué sur la pile. Son contenu n'est pas initialisé. Dans le cas où les entiers `int` sont 32 bits, notre tableau occupe 16 octets. En interne, la variable `t` contient l'adresse où le tableau a été alloué. Le premier élément du tableau est stocké a l'adresse `t`, le second est stocké 4 octets plus loin, etc. De manière générale, l'expression `t[i]` fait référence à la case $i$ du tableau $t$ pour une valeur de $i$ telle que $0 \leqslant i < n$ où $n$ désigne la taille du tableau. En C, il n'y a aucune protection contre un accès en dehors du tableau, que ce soit en lecture ou en écriture. Autrement dit, l'accès en dehors des bornes du tableau est « undefined behavior ». Si on lit une valeur avec un indice en dehors des bornes du tableau, le programme peut continuer en utilisant une valeur non prédictible, ou s'arrêter brutalement à cause d'une erreur de segmentation. Si on écrit en dehors d'un tel tableau alloué sur la pile, il est même possible de corrompre la pile et ainsi faire face à un comportement imprédictible plus tard dans l'exécution du programme.

Lors de la déclaration d'un tableau, il est possible de l'initialiser, en donnant des éléments entre accolades.

```
int t[3] = { 42, 17, 21 };
```

Attention, en C, il est de la responsabilité du programmeur de garder une variable contenant la taille du tableau. C'est pourquoi on déclarera le plus souvent une variable contenant la taille du tableau juste avant de déclarer notre tableau.

```
const int n = 3;
int t[3] = { 42, 17, 21 };
```

Notons que la taille d'un tableau doit être définie par un entier litéral. C'est pourquoi, même si cela peut paraitre surprenant, il n'est pas possible d'écrire

```
const int n = 3;
int t[n] = { 42, 17, 21 };
```

Il est donc idiomatique d'utiliser une constante litérale à l'aide du préprocesseur.

```
#define N 3

int t[N] = { 42, 17, 21 };
```

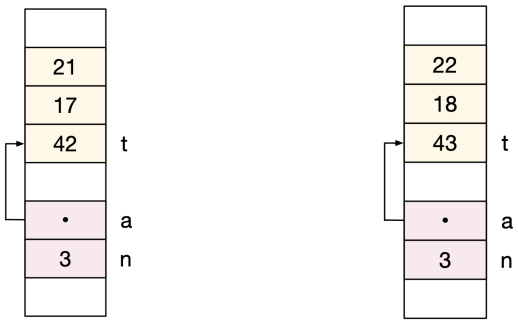
Lorsqu'on passe un tableau en paramètre à une fonction, on ne fait que passer que l'*adresse de ce tableau*. En particulier, le contenu du tableau n'est pas copié, mais partagé. Comme il n'est pas possible de connaitre la taille du tableau uniquement à partir de son adresse, l'usage veut qu'on passe aussi en argument la taille de ce dernier. Par exemple, dans cet exemple, la fonction `g` incrémente de 1 toutes les valeurs du tableau `t`.

```
void g(int a[], int n) {
    for (int i = 0; i < n; i++) {
        a[i] += 1;
    }
}

void f(void) {
    int t[3] = { 42, 17, 21 };
    g(t, 3);
```

```
    ...
}
```

Une fois à l'intérieur de la fonction `g`, l'état de la pile est le suivant. À gauche, nous avons l'état de la pile avant la boucle, et à droite, l'état de la pile après cette même boucle.




Une fois que l'on sort de la fonction `g`, les modifications effectuées par la fonction `g` dans le tableau sont donc observables à l'intérieur de la fonction `f`.

Notons que la fonction `g` a pour signature `void g(int a[], int n)`, mais `a` n'est rien d'autre qu'un pointeur de type `int *`. Il aurait d'ailleurs été strictement équivalent d'écrire `void g(int *a, int n)`. Cette relation entre pointeurs et tableaux s'explique par le fait que, pour le langage C, l'expression `a[i]` n'est rien d'autre qu'un raccourci syntaxique pour l'expression `*(a + i)`, c'est-à-dire un déréférencement du pointeur `a + i`. Dans cette expression, où on ajoute un entier à un pointeur, l'opérateur `+` est une *arithmétique de pointeur*. Cette opération ajoute à l'adresse mémoire représentée par `a`, un décalage représentant la case `i` d'un tableau d'entiers. Sur une machine où le type `int` occupe 32 bits, on obtient un décalage de $4i$ octets. Le compilateur est capable de calculer ce décalage en utilisant le type de `a`, ici `int *`.

On peut allouer un tableau sur le tas plutôt que sur la pile avec la fonction `malloc`.

```
int n = 3;
int *t = malloc(n * sizeof(int));
```

Comme nous l'avons vu plus haut, le pointeur `t` peut être passé à la fonction `g` qui attend un tableau, ou un pointeur puisque c'est la même chose. Une fois notre tableau utilisé, il faudra libérer la mémoire à l'aide de la fonction `free`.

```
free(t);
```

L'avantage d'allouer un un tableau sur le tas est que contrairement à ce qui se passe pour les tableaux alloués sur la pile, il n'est pas nécessaire de connaitre sa taille à la compilation. Une allocation sur le tas est aussi le seul moyen que l'on dispose pour qu'une fonction renvoie un tableau. En effet, le code suivant

```
int *f(void) {
    int t[] = { 42, 17, 21 };
    return t;
}
```

renvoie un pointeur vers un élément du tableau d'activation de la fonction `f` qui va disparaitre dès que l'on va sortir de la fonction. La personne ayant appelé cette fonction aura donc à sa disposition un pointeur fantôme. Si l'on souhaite renvoyer un tableau, il est donc nécessaire en C de l'allouer sur le tas de la manière suivante.

```
int *f(void) {
    int *t = malloc(3 * sizeof(int));
    t[0] = 42;
    t[1] = 17;
    t[2] = 21;
    return t;
}
```

Il sera de la responsabilité de la personne ayant appelé la fonction `f` de libérer la mémoire une fois que le tableau `t` ne sera plus utilisé.

## 2.3 Tableau multidimensionnel

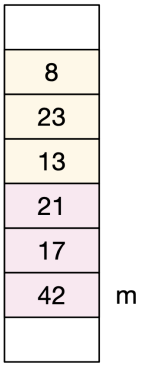
Si l'on souhaite construire un tableau à plusieurs dimensions, comme une matrice à 2 lignes et 3 colonnes, on écrit

```c
int m[2][3];
```

Le tableau sera alloué sur la pile, et on accèdera à la ligne $i$ et à la colonne $j$ pour $0 \leqslant i < 2$ et $0 \leqslant j < 3$ à l'aide de la syntaxe `m[i][j]`. Il est possible d'initialiser un tableau de la manière suivante

```c
int m[2][3] = {{42, 17, 21}, {13, 23, 8}};
```

où les éléments du tableau `m` sont énumérés lignes après lignes. Par exemple `m[1][0]` est la valeur de la seconde ligne et de la première colonne, c'est-à-dire 13. En mémoire, les éléments du tableau `m` vont être placés sur la pile, ligne après ligne.




Autrement sur une machine où les entiers de type `int` occupent 4 octets, l'accès à l'élément `m[i][j]` va se faire en effectuant un décalage de $4 \times (3i + j)$ octets par rapport au premier élément. Le nombre de colonnes de notre tableau doit donc être connu par le compilateur. Si on doit passer un tableau multidimensionnel en paramètre d'une fonction, il est donc nécessaire de spécifier le nombre de colonnes. On passera en général le nombre de lignes dans un autre paramètre. Ainsi, la fonction

```c
void f(int m[][3], int n) {
    ...
}
```

pourra être utilisée avec n'importe quel tableau possédant $n$ lignes 3 colonnes.

Si le nombre de lignes et de colonnes n'est pas connu à la compilation, il est nécessaire de placer ce dernier sur le tas. Il existe deux manières de faire cela. La première consiste à stocker les éléments dans un tableau unidimensionnel.

```c
void f(int nb_lines, int nb_cols) {
    int *m = malloc(nb_lines * nb_cols * sizeof(int));
    ...
}
```

On accèdera ensuite à l'élément d'indice $(i, j)$ en à l'aide de `m[i * nb_cols + j]`.

Il existe une autre manière de procéder qui consiste à créer un tableau unidimensionnel de pointeurs vers des tableaux alloués sur le tas. Pour cela, on écrit

```c
void f(int nb_lines, int nb_cols) {
    int **m = malloc(nb_lines * sizeof(int *));
    for (int i = 0; i < nb_lines; i++) {
        m[i] = malloc(nb_cols * sizeof(int));
    }
    ...
}
```

On accèdera ensuite à l'élément d'indice $(i, j)$ en à l'aide de `m[i][j]`.

Bien entendu, il est possible de créer de la même manière des tableaux à 3 dimensions. Si leur dimension est connue à la compilation, on peut les allouer sur la pile de la manière suivante.

```c
int t[2][3][4];
```

On accèdera ensuite à l'élément d'index $(i, j, k)$ pour $0 \leqslant i < 2$, $0 \leqslant j < 3$ et $0 \leqslant k < 4$ à l'aide de la syntaxe `t[i][j][k]`. Les éléments du tableau vont être placés les uns à la suite des autres de sorte que pour accéder à l'élément d'indice $(i, j, k)$, sur une machine où un `int` occupe 4 octets, le compilateur devra effectuer un décalage de $4 \times [(i \times 3 + j) \times 4 + k]$ octets. Afin de passer un tableau à une fonction, on utilisera la syntaxe

```c
void f(int t[][3][4], int n) {
    ...
}
```

où $n$ est la première dimension, ici 2.

### 2.3.1 Chaine de caractères

En C, une chaîne de caractère n'est rien d'autre qu'un tableau de caractères, de type `char`, dont le dernier élément est le caractère nul `'\0'`. Chaque caractère est stocké sur un octet. Par exemple l'instruction

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

va créer un tableau de taille 6 dont les éléments sont successivement `'h'`, `'e'`, `'l'`, `'l'`, `'o'` et `'\0'`. Comme pour un tableau, la longueur d'une chaine n'est pas stockée en mémoire, mais contrairement au tableau, la présence du caractère nul en fin de chaine (on appelle cette valeur une sentinelle) permet de calculer sa longueur. Tout comme les tableaux, il est possible de créer sur le tas des chaines de caractères dont la taille n'est pas connue à la compilation.

La bibliothèque C fournit des fonctions pour comparer deux chaines pour l'ordre lexicographique (`strcmp`), convertir une chaine en entier (`atoi`) ou calculer la longueur d'une chaine (`strlen`). D'autres fonctions permettent de modifier une chaine en place, par exemple en l'écrasant par une autre (`strcpy`) ou en lui concaténant une autre chaine (`strcat`). Bien entendu, il est nécessaire pour cela que l'espace de destination soit assez grand.

## 2.4 Structure

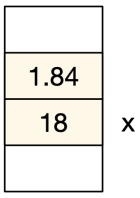
Une *structure* permet d'agréger plusieurs valeurs, en nommant les différentes composantes. On déclare ainsi une structure `S` contenant deux composantes `age` et `height` de types respectifs `int` et `double` qu'on appelle *champs* de la structure, de la manière suivante.

```c
struct info {
    int age;
    double height;
}
```

On dispose maintenant d'un nouveau type qui se note `struct info`. Il y a plusieurs façons de construire une valeur de ce type. Si on déclare une variable locale de type `struct info`, la structure est alors allouée sur la pile. Il est d'ailleurs possible de l'initialiser de la manière suivante.

```c
struct info x = {.age = 18, .height = 1.84};
```

Sur la pile, les éléments d'une structure sont situés ainsi.




On a ainsi initialisé la structure en donnant des valeurs aux champs `age` et `height` entre accolades. On accède à la valeur d'un champ avec la notation `struture.champ`. De plus, la valeur d'une champ peut être modifiée par une affectation.

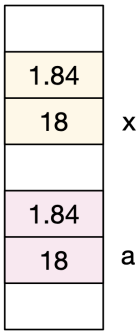
```c
x.age = 19;
```

Lorsqu'une structure est passée à une fonction, elle est intégralement copiée dans la variable qui est le paramètre de la fonction. Par exemple, avec le code

```c
void g(struct info a) {
    ...
}

void f(void) {
    struct info x = {.age = 18, .height = 1.84};
    g(x);
    ...
}
```

une fois à l'intérieur de la fonction `g`, la pile sera dans l'état suivant.




Toute modification de `a` n'aura donc aucune influence sur la valeur de `x`. De la même manière, la valeur d'une structure est entièrement copiée lors d'une affectation de structures ou lorsqu'une fonction renvoie une structure avec `return`.

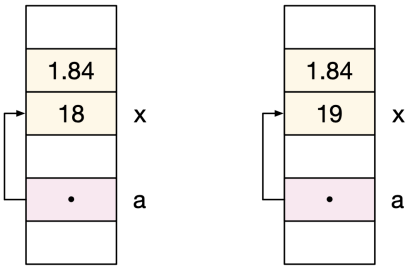
Lorsque l'on souhaite qu'une fonction modifie la valeur d'une structure, on passe à la fonction un pointeur vers une structure. Si la variable `a` est un pointeur vers une structure `s`, c'est-à-dire si elle est de type `s *`, alors on accède au champ `age` de la structure en commençant par déréférencer le pointeur, pour ensuite accéder au champ. On écrit donc `(*a).age`. Les pointeurs vers les structures étant très utilisés, le langage C propose le raccourci syntaxique `a->age` pour cela. Ainsi, la fonction

```
1 void increase_age(struct info *a) {
2     a->age += 1;
3 }
```

pourra être utilisée par le code

```
1 struct info x = {.age = 18, .height = 1.84};
2 increase_age(&x);
```

pour ajouter un an à l'age de la personne. Voici l'état de la pile avant et après l'exécution de l'instruction `a->age += 1` dans la fonction `g`.




Il est également possible d'allouer une structure sur le tas avec `malloc`. Le nombre d'octets à allouer est donné par `sizeof(struct info)`, c'est-à-dire la place allouée occupée par une structure de type `struct info`.

```
1 struct info *x = malloc(sizeof(struct info));
2 x->age = 18;
3 x->height = 1.84;
```

Bien entendu, il faudra prendre soin de libérer plus tard l'espace mémoire alloué par `malloc` avec la fonction `free`. Il est aussi possible de définir des tableaux de structures, qu'ils soient alloués sur la pile ou sur le tas.

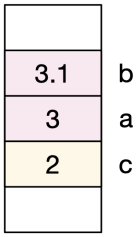
Enfin, il est tout à fait possible d'imbriquer des structures, c'est-à-dire d'avoir un champ de structure dont le type est une structure, un tableau de structures ou un pointeur vers une structure. Par exemple si

```
1 struct x {
2     int a;
3     double b;
4 }
5
6 struct y {
7     int c;
8     struct x d;
9 }
```

on peut initialiser une structure de type `struct y` de la manière suivante.

```
struct y v = {.c = 2, .d = {.a = 3, .b = 3.1}};
```

Dans ce cas, la mémoire est à plat et les éléments de la structure sont les uns à la suite des autres.



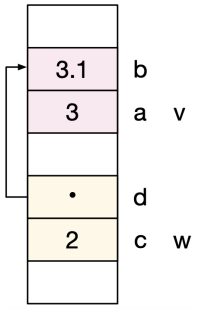

Ce comportement est d'ailleurs différent du cas où notre seconde structure possède un pointeur vers la première structure

```
struct z {
    int c;
    struct x *d;
}
```

ce qui permet l'initialisation suivante.

```
struct x v = {.a = 3, .b = 3.1};
struct z w = {.c = 2, .d = &v};
```

Dans ce cas, la structure de la pile est la suivante.




Enfin, comme il est souvent fastidieux de toujours taper `struct`, il est possible de définir un raccourci à l'aide de `typedef`.

```
typedef struct info info;
```

On pourra alors utiliser `info` en lieu et place de `struct info`.

# 3 Entrée/Sortie

On présente ici quelques éléments simples pour écrire un programme interagissant avec le monde extérieur.

## 3.1 Affichage sur la sortie standard

D'une façon très élémentaire, on peut afficher un caractère sur la sortie standard avec la fonction `putchar` ou encore une chaine de caractères, suivie d'un retour charriot, avec la fonction `puts`. C'est rapidement limité, notamment si on cherche à imprimer des nombres entiers ou flottants. Pour cela, on utilise plutôt la fonction `printf`, que nous avons déjà croisée brièvement.

La fonction `printf` reçoit en premier argument une chaine de caractères appelée *chaine de format*. C'est la chaine qui va être imprimée. Cette chaine peut contenir des directives commençant par le caractère `%`, comme `%d`, `%f` ou `%s`, qui indiquent que les arguments suivants de `printf` doivent être imprimés à cet endroit-là, tout en indiquant leur type. Ainsi, on peut écrire

```
printf("La dimension est %d x %d\n", n, m);
```

avec $n$ et $m$ deux entiers (ici ce sont deux variables, mais il pourrait s'agir de deux expressions quelconques de type `int`) et on obtient alors un affichage sur la sortie standard de la forme suivante :

```
La dimension est 34 x 55
```

De la même façon, on peut afficher une valeur de type `double` avec `%f` ou une chaine de caractères de type `char *` avec `%s`. Pour un nombre flottant, on peut spécifier le nombre de chiffres après la virgule ; par exemple `%.5f` affichera 5 décimales. On consultera la documentation de `printf` pour plus de précisions.

### 3.2 Lecture sur l'entrée standard

Pour lire sur l'entrée standard, on dispose d'une fonction très élémentaire `getchar` qui lit un unique caractère, mais aussi d'une fonction plus puissante, `scanf`, analogue à la fonction `printf`. Comme pour `printf`, on indique une chaine de format qui décrit la forme de l'entrée attendue, avec des éléments comme `%d`, `%f` ou `%s` pour spécifier la reconnaissance d'entiers, de flottants ou de chaines. Ainsi, on peut écrire

```
1 scanf("%d/%d/%d\n", &day, &month, &year);
```

pour lire une ligne sur l'entrée standard, contenant exactement trois entiers séparés par deux caractères `/`. Les arguments de la fonction `scanf` au-delà de la chaine format sont des pointeurs vers les emplacements mémoire qui recevront les valeurs reconnues. Ici, on a pris l'adresse de trois variables `day`, `month` et `year` de type `int` pour recevoir les trois entiers reconnus. Pour une chaine de caractères, l'emplacement mémoire indiqué doit être suffisamment grand pour recevoir la chaine et son caractère nul final.

Si la fin de l'entrée est atteinte, la fonction `scanf` renvoie la valeur spéciale `EOF`. Voici par exemple un programme qui lit des entiers sur l'entrée standard, un par ligne, et affiche leur somme une fois que la fin de l'entrée est atteinte (par exemple avec la saisie de Ctrl-D dans un terminal).

```
1 int x = 0;
2 int s = 0;
3 while (true) {
4     if (scanf("%d\n", &x) == EOF) {
5         break;
6     }
7     s += x;
8 }
9 printf("La somme vaut %d\n", s);
```

On consultera la documentation de `scanf` pour plus de précisions.

### 3.3 Lecture de la ligne de commande

Les arguments de la ligne de commande d'un programme C sont passés à la fonction `main` sous la forme de deux paramètres : un entier donnant le nombre d'éléments sur la ligne de commande et un tableau de chaines de caractères.

```
1 int main(int argc, char **argv)
```

Il faut savoir que le nom de l'exécutable fait partie de la ligne de commande, comme premier élément. Voici par exemple un programme qui lit deux entiers sur la ligne de commande et affiche leur somme :

```
1 int main(int argc, char **argv) {
2     printf("La somme vaut %d\n", atoi(argv[1]) + atoi(argv[2]));
3
4     return 0;
5 }
```

On le compile et on le lance comme ceci :

```
$ gcc somme.c -o somme
$ ./somme 34 55
La somme vaut 89
```

## 4 Modularité

Lorsqu'un programme C commence à devenir gros, il est intéressant de découper son code en plusieurs fichiers. Par ailleurs, certains de ces fichiers pourront être réutilisés dans d'autres programmes ; on les appelle bibliothèques. Supposons ainsi qu'on écrive une partie de notre programme dans un premier fichier, `math.c`, contenant la définition d'une fonction

```
int power(int x, int n) { ... }
```

et le reste de notre programme dans un second programme, `main.c`, qui analyse la ligne de commande, fait des calculs en utilisant la fonction `power` et affiche des résultats. On peut alors compiler notre programme en passant ces deux fichiers au compilateur C.

```
$ gcc math.c main.c -o main
```

Le programme est effectivement compilé et son exécution n'est pas différente de celle d'un programme qui aurait été écrit dans un seul fichier. Cependant, le compilateur s'est plaint, avec un avertissement, de l'utilisation dans `main.c` d'une fonction `power` qu'il ne connait pas.

```
main.c:8:18: warning: implicit declaration of function 'power'
```

En effet, tout se passe ici comme si on compilait successivement et indépendamment, les deux fichiers `math.c` et `main.c`, avant de réaliser une édition de liens avec les deux codes compilés `math.o` et `main.o`.

```
$ gcc -c math.c
$ gcc -c main.c
$ gcc math.o main.o -o main
```

On appelle cela de la *compilation séparée*. C'est dans la deuxième commande, c'est-à-dire la compilation de `main.c`, que le compilateur se plaint de ne pas connaitre la fonction `power`. Pour y remédier, il faut déclarer la fonction `power` dans le fichier `main.c` avant de l'utiliser. Pour cela, on écrit la ligne

```
int power(int x, int n);
```

qui déclare l'existence d'une fonction `power` et donne son type. On note que la ligne se termine par un point-virgule, sans corps pour la fonction `power`. Avec cette déclaration, le compilateur C dispose de toute l'information nécessaire (nombre et types des arguments, type du résultat) pour compiler le fichier `main.c`.

Lors de l'édition de liens, c'est-à-dire notre troisième commande qui construit l'exécutable à partir des deux fichiers `math.o` et `main.o`, le compilateur C va vérifier que la fonction `power` promise dans `main.c` est bien présente, en l'occurrence dans `math.o`. Si elle venait à manquer, la compilation échouerait avec un message du type suivant :

```
main.c:(.text+0x46): undefined reference to 'power'
```

Le compilateur accepte, modulo un avertissement, de compiler un fichier qui fait référence à une fonction nulle part déclarée (même s'il n'est pas conseillé de le faire) mais il refuse en revanche de construire un exécutable dans lequel il manquerait une fonction.

## 4.1 Fichier d'en-tête

On comprend que si on commence à ajouter d'autres fonctions dans notre fichier `math.c`, il va falloir les déclarer partout où elles seront utilisées. Si en particulier on utilise `math.c` dans plusieurs programmes, il faudra dupliquer d'autant les déclarations des fonctions fournies par `math.c`, à minima celles qui sont effectivement utilisées. Et si le nom ou le type de l'une de ces fonctions vient à changer, il faudra mettre à jour toutes les déclarations de cette fonction. Ce n'est pas très satisfaisant.

Une solution à ce problème consiste à écrire les déclarations des fonctions fournies par le fichier `math.c` dans un unique fichier, une fois pour toutes. On appelle cela un *fichier d'en-tête* et il porte le siffuxe `.h` (de l'anglais *header*). Dans notre cas, on écrit donc un fichier `math.h` contenant une seule ligne, à savoir

```
int power(int x, int n);
```

On peut alors *inclure* ce fichier d'en-tête dans notre fichier `main.c` en utilisant la directive `#include`.

```
#include "math.h"
```

C'est identique à l'inclusion des fichiers d'en-têtes de bibliothèques, comme `stdlib.h`, la seule petite différence étant ici l'utilisation de guillemets autour de `math.h`, plutôt que de chevrons, pour signifier que le fichier doit être cherché localement, dans le répertoire courant, plutôt que dans la bibliothèque du compilateur.

Supposons maintenant que nous augmentons notre fichier `math.c`, et donc notre fichier `math.h`, avec d'autres déclarations, comme par exemple un type de paires d'entiers et une fonction effectuant la division euclidienne.

```c
struct euclidean {
    int quotient;
    int remainder;
}
typedef struct euclidean euclidean;

euclidean division(int a, int b);
```

Pour éviter d'écrire la définition de ce type à la fois dans les fichiers `math.c` et `math.h`, il suffit d'inclure le fichier `math.h` dans le fichier `math.c`.

```c
#include "math.h"

euclidean division(int a, int b) { ... }
int power(int x, int n) { ... }
```

Incidemment, le compilateur va maintenant vérifier que les définitions contenues dans `math.c` sont conformes aux déclarations contenues dans `math.h`.

Si notre programme est composé de nombreux fichiers, qui incluent à chaque fois les fichiers d'en-têtes nécessaires, on peut vite se retrouver à inclure plusieurs fois un même fichier d'en-tête, par transitivité. Ainsi, si on utilise à la fois les fichiers `foo` et `bar`, et qu'on inclut donc les en-têtes `foo.h` et `bar.h`, il se peut que le fichier `bar.h` inclut lui-même l'en-tête `foo.h`, car il a besoin d'un type qui y est défini. Le compilateur se retrouve alors avec le type défini deux fois, ce qui provoque une erreur du type `error: redefinition of 'struct ...'`.

Une solution à ce problème consiste à rendre l'inclusion d'un en-tête idempotente, c'est-à-dire sans effet la seconde fois, en se servant des directives `#ifndef` et `define` de la manière suivante (ici sur l'exemple de notre fichier `math.h`).

```c
#ifndef MATH
#define MATH
...
#endif
```

La première fois que le fichier est inclus, la macro `MATH` n'est pas définie et tout le contenu du fichier est donc considéré. En particulier, `#define` defini la macro `MATH`, avec un contenu vide, en l'occurrence. Si le fichier est inclus de nouveau par la suite, la macro étant maintenant définie, tout le bloc entre `#ifndef` et `#endif` est ignoré, ce qui est l'effet attendu.

## 4.2 Espace de noms

Le découpage d'un programme en plusieurs fichiers, et plus généralement la construction de bibliothèques pose un autre problème, un peu plus gênant. Il s'agit de la gestion de l'*espace de noms* qui, dans le langage C, est complètement à plat. Deux fonctions d'un même programme, même contenues dans deux fichiers différents, ne peuvent pas porter le même nom. Ainsi, on ne pourrait pas avoir à côté de `math.c` un autre fichier, disons `modular.c`, introduisant une autre fonction

```c
int power(int x, int n, int m) { ... }
```

pour calculer $x$ à la puissance $n$ modulo $m$, quand bien même elle n'a pas le même nombre d'arguments que la fonction `power` de `math.c`. Il n'y a pas de surcharge des opérateurs en C.

La même contrainte d'unicité de nom existe pour la définition d'une structure (`struct`) ou d'un type (`typedef`). Si on a la maitrise de l'ensemble des fichiers composant le programme, on peut s'en sortir facilement, en donnant des noms uniques à toutes nos fonctions, nos structures et nos types. Mais lorsqu'on développe une bibliothèque, qui sera utilisée dans d'autres programmes sur lesquels on n'a pas la maitrise, alors il devient impossible de savoir avec quel ensemble de noms on risque d'entrer en conflit. Un pis-aller consiste à préfixer les noms avec leur origine. Ainsi, il est préférable d'appeler nos deux fonctions `math_power` et `math_division` pour éviter tout conflit avec d'autres fonctions `power` ou `division`.