

# Paradigmes de programmation

Cours 1 : Rappels sur les types de base

---

Benoît Montagu — `benoit.montagu@inria.fr`

Préparation à l'agrégation d'informatique — Automne 2022



Chaque langage de programmation possède des **types de base** (entiers, booléens, flottants, chaînes de caractères...).

Mais ils peuvent être différents d'un langage à l'autre :

- ▶ Les choix d'implémentation sont différents
  - ▶ Représentation mémoire ?
  - ▶ Complexité asymptotique des opérations ?
- ▶ L'interface de programmation est différente aussi !
  - ▶ Données mutables ou immuables ?
  - ▶ Comportement en cas de dépassement de bornes ?

## Quelques rappels :

- ▶ Représentation en machine des entiers
- ▶ Entiers en C/OCaml/Python
- ▶ Booléens en C/OCaml/Python
- ▶ Rappels rapides sur les flottants IEEE-754
- ▶ Chaînes de caractères en C/OCaml/Python

## Rappels sur les entiers

---

## Représentation binaire des entiers

- ▶ Les entiers sont représentés sur  $n$  bits :  $b_{n-1} \cdots b_1 b_0$  où  $b_i \in \{0, 1\}$
- ▶ Sur les machines actuelles :  $n \in \{32, 64\}$  généralement
- ▶ Vocabulaire : un octet (byte) est un entier sur 8 bits

# Représentation binaire des entiers

- ▶ Les entiers sont représentés sur  $n$  bits :  $b_{n-1} \dots b_1 b_0$  où  $b_i \in \{0, 1\}$
- ▶ Sur les machines actuelles :  $n \in \{32, 64\}$  généralement
- ▶ Vocabulaire : un octet (byte) est un entier sur 8 bits
- ▶ **Interprétation non signée** :  $[[\cdot]]_u : \{0, 1\}^n \rightarrow \mathbb{N}$

$$[[b_{n-1} \dots b_1 b_0]]_u = \sum_{i=0}^{n-1} b_i \times 2^i$$

- ▶ Intervalle :  $0 \leq [[b_{n-1} \dots b_1 b_0]]_u < 2^n$      $\text{MIN}_u = 0 \dots 0$      $\text{MAX}_u = 1 \dots 1$

# Représentation binaire des entiers

- ▶ Les entiers sont représentés sur  $n$  bits :  $b_{n-1} \dots b_1 b_0$  où  $b_i \in \{0, 1\}$
- ▶ Sur les machines actuelles :  $n \in \{32, 64\}$  généralement
- ▶ Vocabulaire : un octet (byte) est un entier sur 8 bits
- ▶ **Interprétation non signée** :  $\llbracket \cdot \rrbracket_u : \{0, 1\}^n \rightarrow \mathbb{N}$

$$\llbracket b_{n-1} \dots b_1 b_0 \rrbracket_u = \sum_{i=0}^{n-1} b_i \times 2^i$$

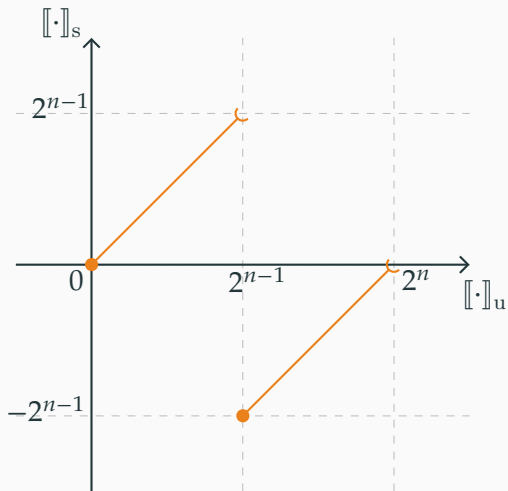
- ▶ Intervalle :  $0 \leq \llbracket b_{n-1} \dots b_1 b_0 \rrbracket_u < 2^n$      $\text{MIN}_u = 0 \dots 0$      $\text{MAX}_u = 1 \dots 1$
- ▶ **Interprétation signée, en complément à 2** :  $\llbracket \cdot \rrbracket_s : \{0, 1\}^n \rightarrow \mathbb{Z}$

Le bit  $b_{n-1}$  indique le signe

$$\begin{aligned} \llbracket b_{n-1} \dots b_1 b_0 \rrbracket_s &= -b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i \\ &= -b_{n-1} \times 2^n + \llbracket b_{n-1} \dots b_1 b_0 \rrbracket_u \end{aligned}$$

- ▶ Intervalle :  $-2^{n-1} \leq \llbracket b_{n-1} \dots b_1 b_0 \rrbracket_s < 2^{n-1}$      $\text{MIN}_s = 10 \dots 0$      $\text{MAX}_s = 01 \dots 1$   
 $\llbracket 00 \dots 0 \rrbracket_s = 0$      $\llbracket 11 \dots 1 \rrbracket_s = -1$

# Entiers binaires : non signés et signés



En complément à 2 :

- ▶ Les signés **positifs** ont la même représentation que les non signés
- ▶ Les signés **négatifs** sont représentés comme les « grands » non signés



# Opérations arithmétiques

- ▶ Arithmétique non signée : les opérations se font modulo  $2^n$

Exemple pour l'addition :

$$\llbracket a + b \rrbracket_u = \begin{cases} \llbracket a \rrbracket_u + \llbracket b \rrbracket_u & \text{si } \llbracket a \rrbracket_u + \llbracket b \rrbracket_u < 2^n \\ \llbracket a \rrbracket_u + \llbracket b \rrbracket_u - 2^n & \text{sinon} \end{cases}$$

- ▶ Arithmétique signée : même implémentation que les opérations non signées!

Exercice : démontrer que

$$\llbracket a + b \rrbracket_s = \begin{cases} \llbracket a \rrbracket_s + \llbracket b \rrbracket_s & \text{si } -2^{n-1} \leq \llbracket a \rrbracket_s + \llbracket b \rrbracket_s < 2^{n-1} \\ \llbracket a \rrbracket_s + \llbracket b \rrbracket_s - 2^n & \text{si } 2^{n-1} \leq \llbracket a \rrbracket_s + \llbracket b \rrbracket_s \\ \llbracket a \rrbracket_s + \llbracket b \rrbracket_s + 2^n & \text{si } \llbracket a \rrbracket_s + \llbracket b \rrbracket_s < -2^{n-1} \end{cases}$$

- ▶ Identités remarquables :

$$\begin{aligned} 0_u &= 0_s \\ \text{MAX}_s + 1 &= \text{MIN}_s \\ \text{MAX}_s \times \text{MAX}_s &= 1 \end{aligned}$$

$$\begin{aligned} \text{MAX}_u &= -1_s \\ -\text{MIN}_s &= \text{MIN}_s \\ \text{MIN}_s \times \text{MIN}_s &= 0 \end{aligned}$$

Chapitre 2.2 « Integer Representations »

Chapitre 2.3 « Integer Arithmetic »



Randal E. BRYANT et David R. O'HALLARON (fév. 2018).  
Computer Systems: A Programmer's Perspective, Global Edition.  
Pearson. 1120 p. ISBN : 1292101768

- ▶ Se lit comme un roman !
- ▶ Représentation : endianness, ones' complement, ...
- ▶ Opérations : opérations logiques, xor, shifts (logique/arithmétique), ...
- ▶ Comment tester l'absence d'overflow
- ▶ Faites les exercices !

## Les entiers en C : types

Type C	Intervalle typique	Intervalle minimum (standard C)
<b>char</b>	$[-2^7, 2^7 - 1]$	$[-(2^7 - 1), 2^7 - 1]$
<b>unsigned char</b>	$[0, 2^8 - 1]$	$[0, 2^8 - 1]$
<b>short</b>	$[-2^{15}, 2^{15} - 1]$	$[-(2^{15} - 1), 2^{15} - 1]$
<b>unsigned short</b>	$[0, 2^{16} - 1]$	$[0, 2^{16} - 1]$
<b>int</b>	$[-2^{31}, 2^{31} - 1]$	$[-(2^{15} - 1), 2^{15} - 1]$
<b>unsigned int</b>	$[0, 2^{32} - 1]$	$[0, 2^{16} - 1]$
<b>long</b>	$[-2^{31}, 2^{31} - 1]$	$[-(2^{31} - 1), 2^{31} - 1]$
<b>unsigned long</b>	$[0, 2^{32} - 1]$	$[0, 2^{32} - 1]$
<b>int32_t</b>	$[-2^{31}, 2^{31} - 1]$	$[-2^{31}, 2^{31} - 1]$
<b>uint32_t</b>	$[0, 2^{32} - 1]$	$[0, 2^{32} - 1]$
<b>int64_t</b>	$[-2^{63}, 2^{63} - 1]$	$[-2^{63}, 2^{63} - 1]$
<b>uint64_t</b>	$[0, 2^{64} - 1]$	$[0, 2^{64} - 1]$

$2^{31} \approx 2,14 \times 10^9$        $2^{32} \approx 4,29 \times 10^9$        $2^{63} \approx 9,22 \times 10^{18}$        $2^{64} \approx 18,4 \times 10^{18}$

## Les entiers en C : subtilités (1)

### Entiers non signés :

- ▶ Dépassements de capacité : **définis par le standard C**
- ▶ Opérations arithmétiques sont définies sur  $\mathbb{Z}/2^n\mathbb{Z}$
- ▶ **Cast** vers un type entier non signé : représentant dans le  $\mathbb{Z}/2^n\mathbb{Z}$  cible
- ▶ Exemple :

```
#include <stdint.h>
int64_t x1 = (((int64_t) 1) << 32) + 1;
// x1 = 4294967297      (= 2^32 + 1)
uint32_t y1 = (uint32_t) x1;
// y1 = 1              (= x1 - 2^32)
// c'est la troncation de x1 depuis 64 bits vers 32 bits

int64_t x2 = -x1;
// x2 = -4294967297    (= -2^32 - 1)
uint32_t y2 = (uint32_t) x2;
// y2 = 4294967295    (= 2^32 - 1)
```

### Entiers signés :

- ▶ Dépassements de capacité arithmétique : **comportement indéfini**
  - ▶ Exemples :  $-\text{MIN}_s$     $\text{MAX}_s + 1$

## Les entiers en C : subtilités (2)

### Entiers signés :

- ▶ Dépassements de capacité arithmétique : **comportement indéfini**
  - ▶ Exemples :  $-\text{MIN}_s$   $\text{MAX}_s + 1$
- ▶ Cast vers entier signé, sans dépassement de capacité : défini comme étant l'identité
- ▶ Cast vers entier signé, avec dépassement de capacité : **comportement indéfini**
  - ▶ Exemple : cast de  $2^{31}$  depuis **uint32\_t** vers **int32\_t**
  - ▶ Exemple : cast de  $-2^{31} - 1$  depuis **int64\_t** vers **int32\_t**

## Les entiers en C : subtilités (2)

### Entiers signés :

- ▶ Dépassements de capacité arithmétique : **comportement indéfini**
  - ▶ Exemples :  $-\text{MIN}_s$   $\text{MAX}_s + 1$
- ▶ Cast vers entier signé, sans dépassement de capacité : défini comme étant l'identité
- ▶ Cast vers entier signé, avec dépassement de capacité : **comportement indéfini**
  - ▶ Exemple : cast de  $2^{31}$  depuis **uint32\_t** vers **int32\_t**
  - ▶ Exemple : cast de  $-2^{31} - 1$  depuis **int64\_t** vers **int32\_t**
- ▶ Division par zéro : exception matérielle!
- ▶ Division sur opérandes négatives :  $a / b$  est la **troncation vers 0** de  $\frac{a}{b}$ 
  - ▶ Exemple :  $(-5) / 3 = -1$  et  $5 / (-3) = -1$
- ▶ Reste sur opérandes négatives (C99) : défini tel que  $a = (a / b) * b + (a \% b)$ 
  - ▶ Exemple :  $(-5) \% 3 = -2$  mais  $5 \% (-3) = 2$
- ❗ **Pour l'agrégation :** pas de division/reste avec des négatifs!

## Que des entiers signés :

- ▶ **int** : entiers signés 31 bits / 63 bits
  - ❗ 1 bit est réservé pour utilisation par le runtime (GC)
- ▶ **int32** : entiers signés 32 bits
- ▶ **int64** : entiers signés 64 bits
- ▶ **nativeint** : entiers signés 32 bits / 64 bits

## Opérations arithmétiques :

- ▶ Définies comme les opérations en complément à 2
- ▶ Division/reste : comme en C99 (troncation vers 0 + identité d'Euclide)
- ▶ Division par zéro : **exception `Division_by_zero`**



## Entiers en précision arbitraire :

- ▶ Nous avons accès à tout  $\mathbb{Z}$
- ▶ Pas de dépassement de capacité
- ▶ Capacité limitée par la mémoire disponible
- ▶ Pas d'entier maximal, pas d'entier minimal
- ❗ En OCaml : entiers en précision arbitraire avec `Zarith.t`

## Division/reste :

- ▶ Division par zéro : `raise ZeroDivisionError()`
- ▶ « floor division » : `a // b` est la troncation vers  $-\infty$  de  $\frac{a}{b}$ 
  - 👉 Comportement différent de C ou OCaml sur les entiers négatifs!

⚠ **Attention :** l'expression `a / b` renvoie un flottant!

```
x = 3 // 2    # x = 1
y = 3 / 2     # y = 1.5
```

- ▶ Reste : `a % b` défini via `a = (a // b) * b + (a % b)`

## Interlude : recherche dichotomique (C)

```
/* Assumes [n >= 0] and [t] is an array of size [n] and is sorted.
   Returns an index [i] such that [0 <= i < n] and [t[i] == v],
   or returns [-1] if [t] does not contain [v]. */
int bsearch(int *t, int n, int v) {
    int l = 0; // lower bound of the searched range
    int u = n-1; // upper bound of the searched range
    while (l <= u) {
        int i = (l + u) / 2; // middle point
        if (v < t[i]) { // search after [i]
            u = i-1;
        } else if (v > t[i]) { // search before [i]
            l = i+1;
        } else { // we have found [v] at index [i]
            return i;
        }
    }
    return -1; // [v] was not found
}
```

## Interlude : recherche dichotomique (C)

```
/* Assumes [n >= 0] and [t] is an array of size [n] and is sorted.
   Returns an index [i] such that [0 <= i < n] and [t[i] == v],
   or returns [-1] if [t] does not contain [v]. */
int bsearch(int *t, int n, int v) {
    int l = 0; // lower bound of the searched range
    int u = n-1; // upper bound of the searched range
    while (l <= u) {
        int i = (l + u) / 2; // middle point ⚠ possible overflow!
        if (v < t[i]) { // search after [i]
            u = i-1;
        } else if (v > t[i]) { // search before [i]
            l = i+1;
        } else { // we have found [v] at index [i]
            return i;
        }
    }
    return -1; // [v] was not found
}
```

## Interlude : recherche dichotomique (C)

```
/* Assumes [n >= 0] and [t] is an array of size [n] and is sorted.
   Returns an index [i] such that [0 <= i < n] and [t[i] == v],
   or returns [-1] if [t] does not contain [v]. */
int bsearch(int *t, int n, int v) {
    int l = 0; // lower bound of the searched range
    int u = n-1; // upper bound of the searched range
    while (l <= u) {
        int i = l + (u - l) / 2; // middle point 👍 overflow impossible
        if (v < t[i]) { // search after [i]
            u = i-1;
        } else if (v > t[i]) { // search before [i]
            l = i+1;
        } else { // we have found [v] at index [i]
            return i;
        }
    }
    return -1; // [v] was not found
}
```

## Interlude : quelques notes historiques sur la recherche dichotomique

- ▶ Code publié pour la 1<sup>re</sup> fois en 1946 (John Mauchly)
- ▶ Codes incorrects publiés dans de nombreux livres
- ▶ En 1988, seulement 5 sur 20 textbooks contenaient une version correcte de l'algorithme (source)

“ Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried. ”

Donald E. Knuth,  
The Art of Computer Programming, Vol. 3, p. 407

## Interlude : quelques notes historiques sur la recherche dichotomique

- ▶ Code publié pour la 1<sup>re</sup> fois en 1946 (John Mauchly)
- ▶ Codes incorrects publiés dans de nombreux livres
- ▶ En 1988, seulement 5 sur 20 textbooks contenaient une version correcte de l'algorithme (source)

“ Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried. ”

Donald E. Knuth,  
The Art of Computer Programming, Vol. 3, p. 407

- ▶ Le bug d'overflow est resté présent dans la bibliothèque Java pendant 9 ans, corrigé seulement en 2006 (source)
- ▶ Qu'en est-il dans votre langage de programmation préféré ?

# Les booléens

---

« En C, un entier est un booléen comme un autre. »

- ▶ 0 est le booléen « faux »
- ▶ Tout entier différent de 0 est un booléen « vrai »

Il y a donc strictement **plus que deux valeurs booléennes...**

```
if (42) { x = 128; } else { x = -1; } // on a x = 128
```



« En C, un entier est un booléen comme un autre. »

- ▶ 0 est le booléen « faux »
- ▶ Tout entier différent de 0 est un booléen « vrai »

Il y a donc strictement **plus que deux valeurs booléennes...**

```
if (42) { x = 128; } else { x = -1; } // on a x = 128
```

Depuis C99 :

- ▶ `#include <stdbool.h>`
- ▶ Déclare le type `_Bool` et les constantes `true` (`== 1`) et `false` (`== 0`)
- ▶ Cast d'un type entier vers `_Bool` :

```
(_Bool) 0 == false      (_Bool) n == true si n != 0
```

⚠ `(true + true) != false` et `(true + true) != true`

Le calcul n'est pas dans  $\mathbb{Z}/2\mathbb{Z}$  : l'addition promeut les booléens en entiers!

- ❗ **Pour l'agrégation :** autant que possible ne mélangez pas entiers et booléens

## Interlude : opérateurs logiques en C

Nom	Exemple	Remarque
Test d'égalité	<code>e1 == e2</code>	Renvoie un booléen
Affectation	<code>x = e</code>	Affecte x avec la valeur de e, et renvoie la valeur de e

## Interlude : opérateurs logiques en C

Nom	Exemple	Remarque
Test d'égalité	<code>e1 == e2</code>	Renvoie un booléen
Affectation	<code>x = e</code>	Affecte x avec la valeur de e, et renvoie la valeur de e
« Ou » logique	<code>e1   e2</code>	Opération bit à bit Les expressions e1 et e2 sont <u>toujours</u> évaluées
Expression conditionnelle	<code>e1    e2</code>	Opérateur de contrôle, équivalent à <code>(e1)?true:(e2)</code> L'expression e1 est <u>toujours</u> évaluée, mais e2 ne l'est que si e1 s'évalue à <code>false</code>

## Interlude : opérateurs logiques en C

Nom	Exemple	Remarque
Test d'égalité	<code>e1 == e2</code>	Renvoie un booléen
Affectation	<code>x = e</code>	Affecte x avec la valeur de e, et renvoie la valeur de e
« Ou » logique	<code>e1   e2</code>	Opération bit à bit Les expressions e1 et e2 sont <u>toujours</u> évaluées
Expression conditionnelle	<code>e1    e2</code>	Opérateur de contrôle, équivalent à <code>(e1)?true:(e2)</code> L'expression e1 est <u>toujours</u> évaluée, mais e2 ne l'est que si e1 s'évalue à <code>false</code>
« Et » logique	<code>e1 &amp; e2</code>	Opération bit à bit Les expressions e1 et e2 sont <u>toujours</u> évaluées
Expression conditionnelle	<code>e1 &amp;&amp; e2</code>	Opérateur de contrôle, équivalent à <code>(e1)?(e2):false</code> L'expression e1 est <u>toujours</u> évaluée, mais e2 ne l'est que si e1 s'évalue à <code>true</code>

- ▶ Il n'y a que **deux booléens** en OCaml : **true** et **false**
- ▶ Leur représentation est cachée : ce sont des types abstraits
- ▶ Les booléens peuvent être manipulés par :

```
if b                match b with                b1 || b2
then ...            | true  -> ...            b1 && b2
else ...            | false -> ...            not b
```

- ▶ Les booléens peuvent être produits par :

```
e1 = e2            e1 <> e2    (* égalité structurelle *)
e1 == e2           e1 != e2    (* /\ \ égalité d'adresses *)
e1 < e2            e1 <= e2    e1 > e2            e1 >= e2
```

- ❗ En pratique, le compilateur OCaml *fait le choix* de représenter **false** par **0**, et de représenter **true** par **1** (choix compatible avec C)
- ❓ Exercice : écrire une fonction OCaml qui teste si deux valeurs ont la même représentation (sans utiliser le module **Obj**). L'utiliser sur les booléens.

## Solution de l'exercice : testeur de représentation en OCaml

```
(* Type existentiel  $\exists \alpha. \alpha$  implémenté à l'aide d'un GADT
   (type de données algébrique généralisé) *)
type any = Ex: 'a -> any

(* Testeur de représentation, utilisant l'égalité polymorphe *)
let same_representation a b =
  (Ex a) = (Ex b)
(* val same_representation : 'a -> 'b -> bool *)

same_representation 0 false (* résultat : true *)

same_representation 1 true (* résultat : true *)
```

- ▶ Deux constantes booléennes : **True** et **False**
- ▶ Opérateurs : `e1 and e2`, `e1 or e2`, `e1 == e2`, `e1 != e2`  
Attention : `e1 is e2` teste l'égalité d'adresses (est-ce le même objet?)
- ▶ Les booléens sont des objets de la `<class 'bool'>`
- ▶ De manière similaire à C : on confond entiers et booléens

```
>>> if 0:  
...     print("YES")  
... else:  
...     print("NO")  
...  
NO
```

```
>>> False + True
```

1

```
>>> if 42:  
...     print("YES")  
... else:  
...     print("NO")  
...  
YES
```

```
>>> True + True
```

2

- ⚠ **Mais aussi** : la liste/ensemble/map/chaîne vide sont équivalents à **False**, et à **False** dans le cas non vide. Exemples : `bool([])`, `bool((0,0))`

# Booléens en Python : règles de conversion

D'après la documentation :

(source : <https://docs.python.org/3/library/stdtypes.html#truth>)

By default, an object is considered true unless its class defines either a `__bool__()` method that returns **False** or a `__len__()` method that returns zero, when called with the object. Here are most of the built-in objects considered false :

- ▶ constants defined to be false : **None** and **False**.
- ▶ zero of any numeric type : `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- ▶ empty sequences and collections : `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

**i** La conversion vers le type booléen est donc programmable, et peut être redéfinie par héritage.



## Les flottants

---

# Nombres flottants

- ▶ Nombres binaires fractionnaires (pour  $x \in [0, 1)$ )  
À maîtriser : l'exemple de 0.1
- ▶ Norme IEEE-754
  - ▶  $s$  : bit de signe
  - ▶  $m$  : mantisse
  - ▶  $e$  : exposant
  - ▶ Valeur :  $(-1)^s \times m \times 2^e$
  - ▶ Normalisés, dénormalisés, +0, -0,  $+\infty$ ,  $-\infty$ , NaN
  - ⚠ Égalité sur les flottants : en pratique, tester l'égalité « à  $\epsilon$  près »
- ▶ Simple précision, double précision (en C : **float** / **double**)
- ▶ À lire : [Computer Systems : A Programmer's Perspective](#)  
Chapitre 2, Section 4 « Floating Point »
- ❗ Agrégation : les détails de la norme IEEE sont hors programme  
Mais : connaître les idées principales est essentiel!

# Les chaînes de caractères

---

## Chaînes de caractères en C : « string »

- ▶ C'est une suite de caractères situés les uns à la suite des autres...  
... et terminée par le caractère nul `'\0'`
- ▶ Les strings en C sont mutables : elles peuvent être **modifiées en place**
- ⚠ **L'accès en dehors des bornes est indéfini! (bugs de sécurité)**
- ▶ Implémenté par le type **char** \* (pointeur vers un caractère)
- ▶ En C : identique à un tableau de caractères avec un `'\0'` de plus à la fin

```
1 char *s = "abcde"; // char s[] = { 'a','b','c','d','e','\0' };
2 char zeroth = s[0]; // zeroth = 'a', les indices commencent à 0
3 char zeroth = *s; // idem: s pointe vers le 1er élément
4 char ith = s[i]; // accès au i-ème caractère
5 char ith = *(s+i); // idem, par arithmétique de pointeurs
6 char last = s[5]; // last = '\0'
7 char unknown = s[6]; // /\ access hors des bornes
8 char unknown = s[-1]; // idem (comportement indéfini)
9 s[3] = 'D'; // s est maintenant la chaîne "abcDe"
```

## Chaînes de caractères en C : « string »

- ▶ C'est une suite de caractères situés les uns à la suite des autres...  
... et terminée par le caractère nul `'\0'`
- ▶ Les strings en C sont mutables : elles peuvent être **modifiées en place**
- ⚠ **L'accès en dehors des bornes est indéfini! (bugs de sécurité)**
- ▶ Implémenté par le type **char** \* (pointeur vers un caractère)
- ▶ En C : identique à un tableau de caractères avec un `'\0'` de plus à la fin

```
1 char *s = "abcde"; // char s[] = { 'a','b','c','d','e','\0' };
2 char zeroth = s[0]; // zeroth = 'a', les indices commencent à 0
3 char zeroth = *s; // idem: s pointe vers le 1er élément
4 char ith = s[i]; // accès au i-ème caractère
5 char ith = *(s+i); // idem, par arithmétique de pointeurs
6 char last = s[5]; // last = '\0'
7 char unknown = s[6]; // /\! accès hors des bornes
8 char unknown = s[-1]; // idem (comportement indéfini)
9 s[3] = 'D'; // s est maintenant la chaîne "abcDe"
```

🔗 Le code de gauche produit une erreur de segmentation à la ligne 9. Pourquoi? <sup>21/27</sup>

## Exemple : calcul de la longueur d'un chaîne en C

```
1  #include <stdio.h>
2  // Retourne le nombre de caractères de la chaîne s
3  unsigned int str_len(char *s) {
4      unsigned int n = 0;
5      for (char *tmp = s; (*tmp) != '\0'; tmp++) {
6          n++;
7      }
8      return n;
9  }
10 int main() {
11     char *s = "abcde";
12     printf("La chaîne %s contient %u caractères.\n", s, str_len(s));
13     return 0;
14 }

$ gcc -std=c99 -Wpedantic -Wall -Wextra string_test.c -o string_test
$ ./string_test
La chaîne abcde contient 5 caractères.
```

## Chaînes de caractères en OCaml

- ▶ **type string** L'implémentation est cachée : c'est un type abstrait
- ▶ Les chaînes sont immuables en OCaml : elle ne peuvent pas être modifiées
- ▶ Littéraux : comme en C. Exemple : "OCaml\tis fun.\n"

- ▶ Module **String** (extraits) :

```
val init : int -> (int -> char) -> string
```

```
val length : string -> int
```

```
val get : string -> int -> char (* s.[i] identique à get s i *)  
(* get lance l'exception Invalid_argument si accès hors bornes *)  
(* Il n'y a pas de fonction "set": les strings sont immuables *)
```

```
val mapi : (int -> char -> char) -> string -> string
```

```
1 open String
```

```
2 let s = init 5 (fun i -> Char.chr (i + Char.code 'a'))
```

```
3 (* s = "abcde" *)
```

```
4 let bad = try s.[6] with Invalid_argument _ -> '!'
```

```
5 (* bad = '!' *)
```

```
6 let s2 = mapi (fun i ci -> if i = 3 then 'D' else ci) s
```

```
7 (* s = "abcde" et s2 = "abcDe" *)
```

## Chaînes de caractères en Python (1)

- ▶ Une chaîne de caractères est un object de la classe `str`
- ▶ Les chaînes sont immuables
- ▶ Les éléments constituant une string Python sont des caractères UTF8 (qui peuvent tenir sur plus d'1 octet)
- ▶ Exemple : le caractère € est encodé en UTF8 sur 3 octets
- ⚠ Un « caractère » est une chaîne de taille 1

```
>>> s = 'abcde'
>>> (s[0], s[4]) # accès valide à un indice
('a', 'e')
>>> s[5] # accès invalide
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> prix = '2€99'
>>> prix[1]
'€'
```



## Chaînes de caractères en Python (2)

- ▶ Indices négatifs : accès « par la fin »

```
>>> (s[-1], s[-5]) # accès valide "par la fin"
('e', 'a')
```

```
>>> s[-6] # accès invalide "par la fin"
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: string index out of range
```

- ▶ Extraire des sous-chaînes à l'aide de « *slices* »

```
>>> s[2:4] # la sous-chaîne entre les indices 2 et 4 (exclu)
'cd'
```

```
>>> s[2:] # la sous-chaîne démarrant à l'indice 2
'cde'
```

```
>>> s[:4] # la sous-chaîne s'arrêtant à l'indice 4 (exclu)
'abcd'
```

- ▶ Exercice : coder l'équivalent de la fonction OCaml `String.mapi` en Python

## Solution de l'exercice : écrire `mapi` en Python

```
def mapi(f, s):  
    i = 0  
    t = ""  
    for c in s:  
        t = t + f(i,c)  
        i = i + 1  
    return t
```

```
def f(i, c):  
    if i == 3:  
        return 'D'  
    else:  
        return c
```

```
s = "abcde"  
t = mapi(f, s)  
# s = "abcde" et t = "abcDe"
```

## Solution de l'exercice : écrire `mapi` en Python

```
def mapi(f, s):
    i = 0
    t = ""
    for c in s:
        t = t + f(i,c)
        i = i + 1
    return t

def f(i, c):
    if i == 3:
        return 'D'
    else:
        return c

s = "abcde"
t = mapi(f, s)
# s = "abcde" et t = "abcDe"
```

# Version alternative, plus "pythonesque"

```
def mapi(f, s):
    t = ""
    for i, c in enumerate(s):
        t = t + f(i,c)
    return t
```

## Conclusion

---

- ▶ Des types de base parfois très différents suivant les langages
- ▶ Des spécifications différentes
- ▶ Des implémentations non présentées à l'utilisateur
- ▶ Premiers exemples de types abstraits
- ▶ Premier contact avec mutabilité/immuabilité
- 👉 C'est le sujet du cours suivant