

# Langage C

ENSTA - TC 1ère année

François Pessaux

U2IS

2017-2018

`francois.pessaux@ensta-paristech.fr`

## Approfondissement des types de base de C

# Les entiers (naturels et relatifs)

- Plusieurs **tailles**.
- **Signés** ou **non**.
- Par défaut : un entier est **signé**.
- Ex : 0 -3 +0 15657

	Taille (bits)	Valeur	
		Min	Max
short	16	-32768	32767
unsigned short	16	0	65535
int	32	$-2^{31}$	$2^{31} - 1$
unsigned int	32	0	$2^{32} - 1$
long	32 ou 64	dépend de l'architecture	
unsigned long	32 ou 64	dépend de l'architecture	
long long	64	$-2^{63}$	$2^{63} - 1$
unsigned long long	64	0	$2^{64} - 1$

- Entiers codés en binaire : base 2.
  - ▶ 421    0000000110100101 (sur 16 bits)
  - ▶  $= 2^8 + 2^7 + 2^5 + 2^2 + 2^0$
- Base 2 :
  - ▶ Pratique au niveau électronique mais encombrant !
  - ▶ On préfère la base 16 (hexadécimal).
  - ▶    0000 0001 1010 0101  
0x    0     1     A     5

# Codage des entiers signés

- ⚠ Le bit de poids le **plus fort** sert à représenter le **signe**.
  - 0 positif.
  - 1 négatif.
- Pour obtenir l'**opposé** d'un nombre : **complément à 2**.
  - Inverser les bits de l'écriture binaire de sa valeur absolue,
  - puis ajouter 1 au résultat.
  - Calcule  $2^l - x$  où  $l$  est la longueur de représentation d'un entier.
  - 1 seule représentation de 0.
  - + et - identiques sur signés / non-signés, positifs / négatifs.

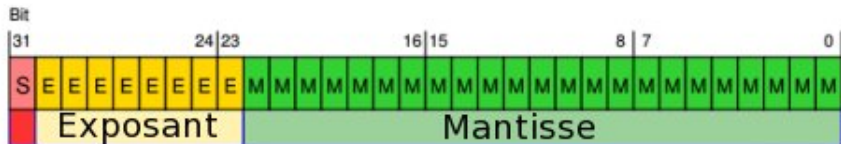
● 4      -4 :

	4	0	1	0	0	(*)
not (*)		1	0	1	1	(**)
(**) + 1		1	1	0	0	

- Arithmétique **entière** et **modulo** la taille des mots machine.
- Pour les exemples suivants, taille = 4 bits.
- Perte de plein de « *bonnes* » propriétés mathématiques ☹
  - ▶ Débordement de non signé :  $9 + 7 = 16$  ?  
 $1001 + 0111 = 1\ 0000$  = sur 4 bits ...  $0000 = 0$
  - ▶ Débordement de signé :  $-8 - 1 = -9$  ?  
 $1000 - 0001 = 1\ 1111$  = sur 4 bits ...  $1111 = -1$
  - ▶ Conversion de signé non signé :  $-4$  ?  
 $1100 = 12$
  - ▶ Conversion de non signé signé :  $15$  ?  
 $1111 = -1$
  - ▶ Associativité de et :  $(1 / 4) * 4 = (1 * 4) / 4$  ?  
 $1 / 4 = 0.25$  En entiers ... 0. Donc,  $* 4 = 0$   
 $1 * 4 = 4$  En entiers ... 4. Donc,  $/ 4, = 1$

- Couramment appelés « flottants » (« *floating point numbers* »).
- Toujours signés.
- 2 tailles 2 précisions.
- Ex : 1.0 3.14159 -6.14 +1.6e-15

	Taille (bits)	Valeur		Précision max
		Min	Max	
float	24 + 8	$1.1 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	$10^{-6}$
double	53 + 11	$2.2 \cdot 10^{-308}$	$1.8 \cdot 10^{308}$	$10^{-15}$



- Nombres flottants pratiques pour approximer les réels mathématiques.
- Pourtant bien différents de la « beauté » mathématique :
  - Notion de précision :
    - ★  $\Rightarrow$  Tous les réels ne sont pas représentables entre 2 réels.
    - ★ Il existe des « trous » entre 2 réels.
    - ★ Ex :  $48431.1231 = 48431.1250$  (c.f. slide suivant).
  - Comme pour les entiers : notion de réels min et max représentables.
  - Mauvais conditionnement :
    - ★ Opérations entre grand et petit flottants parfois incohérentes.
    - ★ Ex :  $100.0 + 4e+15 = \dots 4e+15$
  - Arrondis lors des calculs.
    - ★ Test d'égalité == **proscrit**.
    - ★ Test « modulo un  $\epsilon$  ».
    - ★ Attention,  $\epsilon$  n'est pas absolu et dépend du problème !
  - Conversion réel entier : 0, débordement, troncature.



# Des réels différents égaux. . .

---

## thiskillsme.c

---

```
#include <stdio.h>

int main ()
{
    float fl1 = 48431.1231 ;
    float fl2 = 48431.1239 ;
    float fl3 = 48431.1250 ;

    printf ("fl1: %f fl2: %f fl3: %f\n", fl1, fl2, fl3) ;
    printf ("fl1 == fl2 ? %d\n", (fl1 == fl2)) ;
    printf ("fl2 == fl3 ? %d\n", (fl2 == fl3)) ;
    printf ("fl1 == fl3 ? %d\n", (fl1 == fl3)) ;
    fl1 = fl1 + 0.0001 ;
    printf ("fl1 + 0.0001: %f\n", fl1) ;
    return (0) ;
}
```

---

```
> ./thiskillsme
fl1: 48431.125000 fl2: 48431.125000 fl3: 48431.125000
fl1 == fl2 ? 1
fl2 == fl3 ? 1
fl1 == fl3 ? 1
fl1 + 0.0001: 48431.125000
```

## ... et des réels égaux différents

---

### thiskillsmeagain.c

---

```
#include <stdio.h>
#include <math.h>
#define EPSILON (1e-6) // Tout petit epsilon...

int main ()
{
    double fl1 = 0.1 ;
    double fl2 = 0.2 ;
    double fl3 = 0.3 ;
    double fl4 = fl1 + fl2 ;

    printf ("fl1: %f fl2: %f fl3: %f fl4: %f\n", fl1, fl2, fl3, fl4) ;
    printf ("fl3 = fl4 ? %d\n", (fl3 == fl4)) ;
    printf ("fl3 ~ fl4 ? %d\n", (fabs (fl3 - fl4) < EPSILON)) ;
    return (0) ;
}
```


---

```
> ./thiskillsmeagain
```

```
fl1: 0.100000 fl2: 0.200000 fl3: 0.300000 fl4: 0.300000
```

```
fl3 = fl4 ? 0
```

```
fl3 ~ fl4 ? 1
```

- Caractère : type prédéfini `char` = entiers sur 8 bits.
  - ▶ Par ex : `'a'` = 97 (= 01100001 en binaire).
- Chaîne = tableau de caractères terminé par le caractère `'\0'`.
- Fichier d'en-tête requis : `#include <string.h>`
- Peuvent être copiées : fonction `strcpy` (*dest*, *source*).
- Longueur d'une chaîne : fonction `strlen` (*str*).
  - ▶  Ne compte pas le `'\0'` final!
  - ▶ `strlen ("Damned")` 6.
  - ▶ Pourtant : 

D	a	m	n	e	d	0
---	---	---	---	---	---	---

## Quelques formes particulières de caractères

---

- Utilisables en tant que caractères individuels ou dans des chaînes.
  - ▶ `'\n'` : Retour à la ligne.
  - ▶ `'\\'` : Le caractère `\`.
  - ▶ `'\''` : Le caractère `'`.
  - ▶ `'\"'` : Le caractère `"`.
  - ▶ `'\t'` : La tabulation.
  - ▶ `'\x'` ●● : Le caractère dont le code en **hexadécimal** suit.
  - ▶ `'\'` ●●● : Le caractère dont le code en **octal** suit.
  - ▶ ... et quelques autres que l'on passe sous silence.
- Ex : `printf ("foo\n\"'\bar\\\x65\046");`


foo

"'\bar\&

## Plus de constructions

## Faire $n$ fois : la boucle `for`

---

- `for (instruction1 ; expression ; instruction2) { instruction3(s) }`
-  Des **points-virgules** ! Pas des virgules !
- Sémantique :
  - 1 Initialisation : exécuter *instruction<sub>1</sub>*.
  - 2 Si *expression* s'évalue en `true`
    - 2.1 Exécuter *instruction<sub>3</sub>(s)*.
    - 2.2 Exécuter *instruction<sub>2</sub>* (« post-traitement »).
    - 2.3 Retourner en 2.

---

```
#include <stdio.h>
int main ()
{
    int i ;
    for (i = 0; i < 10; i++) {
        printf ("%d ", i) ;
    }
    printf ("\n") ;
    return (0) ;
}
```

---

```
mymac: /tmp$ gcc enumerate.c -o enumerate
mymac: /tmp$ ./enumerate
0 1 2 3 4 5 6 7 8 9
```

# La boucle `for` dégénérée

---

- Possibilité d'omettre des parties de la construction.
- Initialisation, condition, post-traitement, voire corps !
- Poussé à l'extrême : formes étranges voire illisibles 😞. **À éviter !**

---

```
i = 0 ; j = 0 ;  
for (**/; i < 10; i++) {  
    j += i ;  
}
```

```
j = 0 ;  
for (i = 0; i < 10; **/) {  
    j += i ;  
    i++ ;  
}
```

---

---

```
i = 0 ; j = 0 ;  
for (**/; i < 10; **/) {  
    j += i ;  
    i++ ;  
}
```

```
i = 0 ; j = 0 ;  
for (**/; **/; **/) {  
    j += i ;  
    if (i == 9) break ;  
    i++ ;  
}
```

---

# Équivalence entre boucles `for` et `while`

---

- Remplacement de `for` en `while` :

---

```
for (init; test; postlude) {  
    instruction(s) ;  
}
```

---

---

```
init ;  
while (test) {  
    instruction(s) ;  
    postlude ;  
}
```

---

- Remplacement de `while` en `for` :

---

```
while (test) {  
    instruction(s) ;  
}
```

---

---


```
for ( ; test; ) {  
    instruction(s) ;  
}
```

---



## Faire « tant que » : la boucle `do-while`

---

- Boucle `while` avec le test à la fin du traitement.
- Traitement toujours exécuté au moins 1 fois.
- `do { instruction(s) } while (expression);`
-  Syntaxe : n'oubliez pas le “;” final!

---

```
#include <stdio.h>

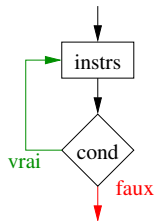
int main ()
{
    char c ;
    do {
        printf ("Entrez un chiffre: ") ;    // Lecture...
        scanf ("%c", &c) ;
    } while ((c < '0') || (c > '9')) ; // Jusqu'à succès.

    return (0) ;
}
```

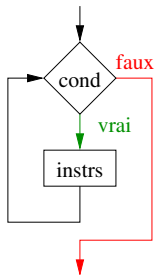
---

# En résumé des boucles

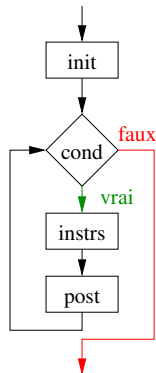
do { instrs } while (cond) ;



while (cond) { instrs }



for (init; cond; post) { instrs }



# Construction conditionnelle « généralisée » (1)

---

- Besoin de tester une valeur contre plusieurs cas **mutuellement exclusifs**.
- Ex : `x == 0` ou `x == 1` ou `x == 4` ou ... ou « sinon ».
- On peut cascader des `if-else`

---

```
if (x == 0) printf ("x: 0\n") ;
else {
    if (x == 1) printf ("x: 1\n") ;
    else {
        if (x == 4) printf ("x: 4\n") ;
        else {
            if ... else ... if ... else ...
            ... printf ("Autre cas\n") ;
        } } ... }
    }
}
```

---

- Ni pratique ni lisible ☹️

## Construction conditionnelle « généralisée » (2)

---

- Discrimination par cas (sur **constantes** se « réduisant » à des entiers).
- Cas doivent être **mutuellement exclusifs**.
- Construction `switch (expression) { ... }`
- Suite de `case constante : instructions`
- Les *instructions* exécutées : celles correspondant au cas où *constante* est égal à la valeur de *expression*.
- Fin de chaque cas par `break ;`
- Cas « *autre* », par défaut : `default`.

---

```
...
switch (x) {
  case 0: printf ("x: 0\n") ; break ;
  case 1: printf ("x: 1\n") ; break ;
  case 4: printf ("x: 4\n") ; break ;
  default: printf ("Autre cas\n") ; break ;
}
```

---

## Quelques utilitaires (autour) de C

# Constantes, macros et préprocesseur (1)

---

- Définir des constantes littérales dans le source : **pas maintenable!**
- Nécessité de changer **partout** la valeur.
- Utilisation de **macros**.

```
#define SIZE (42)
...
int i = SIZE ;
if (j < SIZE) ...
```


- Remplacement **textuel** avant compilation par le **préprocesseur C** (cpp).
- Mieux vaut parenthéser :

```
#define SIZE 42 + 1
...
int i = SIZE * 10 ;
```

▶ **i = 52 et non 430!**

## Constantes, macros et préprocesseur (2)

---

- Possibilité de macros « paramétrées » :  
`#define TWICE(x)((x)* (x))`
- Remplacement **textuel** en substituant `x` par le texte argument lors de l'utilisation.
- `Bla TWICE(ah bon ?)`  
`Bla ((ah bon ?) * (ah bon ?))`
-  Syntaxe : pas d'espace entre le nom de la macro et la parenthèse ouvrante (définition **et** utilisation).
- Possibilité d'arguments multiples : `#define MUL(a,b)((a)* (b))`
- `Bli MUL(((15 )) , autre truc)`  
`Bli (((((15 ))) * ( autre truc))`

- Vous en avez assez de taper `unsigned long int` ?
- Vous devez changer partout `int` en `"unsigned int"` ?
- Définissez un **alias**.
- `typedef unsigned long int uli_t ;`
  - ▶ `uli_t` : nom, abréviation de `unsigned long int`.
  - ▶ Utilisable ensuite comme nom de type : `uli_t i = 42 ;`
- Ne définit **pas** un nouveau type !
- De manière générale : `typedef <type existant> <nom>`
- Fonctionnera avec les types que nous verrons plus tard.



## Contrainte de type (transtypage, cast)

---

- Parfois besoin de changer le type selon lequel une donnée est « vue ».
- `float int, unsigned int long int, ...`



Utilisation d'un **cast** : **(nom-type)** expression.

---

```
float v = 3.14159 ;
int i ;
unsigned long j = 3000000 ;

i = (int) v ;
j = 15 * ((unsigned long) (i + 1)) + j ;
```

---

-  Possibilité de **perte** / **corruption** d'information ! (c.f. sides 5 et 7).
-  « *Conversion* » entier flottant : **change** la représentation interne.
- Autres « *conversions* » : ne changent pas la représentation, juste la façon de **considérer** cette représentation.