

# Langage C

ENSTA - TC 1ère année

François Pessaux

U2IS

2017-2018

francois.pessaux@ensta-paristech.fr

## Introduction

## But de ce cours

---

- Vous avez vu :
  - la programmation,
  - en Python,
  - de l'algorithmique. ...
- Nous allons voir :
  - un nouveau langage : C,
  - encore un peu d'algorithmique.

## Règles d'or

---

- Ne pas hésiter à poser des questions. Même en amphi.
- Dites-vous que si vous avez une question « bête », au moins 1/4 de la promo se pose la même.
- Ne s'avouer ni perdu ni vaincu. Demander des explications.
- Ne pas hésiter à dire « *M'sieur, vous vous z'êtes pas trompé sur bla ?* »
- Mon bureau et mon mail sont facilement accessibles.

## Architecture

## « Vous avez dit ordinateur ? »

---

- *Mais pour quoi faire ?* Pour traiter de l'**information**.
- « *Information ?* »
  - C'est un **message**.
  - Écriture avec des **symboles** : **code**.
  - Monde numérique : code binaire.
- Architecture d'un ordinateur :
  - **Mémoire(s)** : pour stocker de l'information.
  - **Microprocesseur** (CPU) : pour travailler l'information.
  - **Périphériques** : pour échanger avec « l'extérieur ».
- « *Traitement* » ⇒ description de « *comment* » : **programme**.
- Architecture de Von Neumann :
  - Le programme est stocké dans la mémoire...
  - ... dans la même mémoire que les données (informations).

## La mémoire

- Pourquoi en avoir besoin ?
  - Certains traitements peuvent se faire directement :  $x \mapsto x + 1$ .
  - Intervention de la notion de « *temps* » : exemple, un compteur.
  - $\Rightarrow$  Besoin de se rappeler de (d'un) « *état* » précédent.
- Différents **fonctionnements** de mémoire :
  - RAM (Random Access Memory) : lecture / écriture – Volatile.
  - ROM (Read Only Memory) : lecture – Persistante.
  - EPROM (Erasable Programmable Read Only Memory) : lecture (/ écriture mais en mode de fonctionnement particulier) – Persistante.
- Différents **types** de mémoire :
  - Mémoire de masse (disques mécaniques) :  $\sim 10$  ms
  - Mémoire de masse (disques SSD) :  $\sim 0.1 - 0.3$  ms
  - Mémoire vive (RAM) externe au CPU :  $\sim 40 - 50$  ns
  - Mémoire cache  $\pm$  interne au CPU :  $\sim 5 - 10$  ns
  - Registres :  $\sim 1$  ns

## Le microprocesseur

- C'est l'unité qui « *travaille* ».
- Idéalement segmentée en 2 parties :
  - L'unité **arithmétique et logique** (UAL / ALU) : effectue les calculs.
  - L'unité de **contrôle** : décodage et séquençage des instructions.
- Contient de la mémoire très rapide ( $\sim 1$  ns) : les **registres**.
- « *Instruction* » : opération **élémentaire** effectuée par le CPU (addition de 2 registres, copie registre  $\leftrightarrow$  mémoire, registre  $\leftarrow$  registre, décalages, sauts ...)
- Chaque CPU (x86, ARM, amd64 ...) est **différent**, a son **propre** jeu d'instructions.

## Le langage

# Le langage C

- Déjà vu : Python.
- Programmation, algorithmique ≠ langage.
- Langage = support, mots pour le dire.



- Maintenant... C.
- Créé B. Kernighan, D. Ritchie et K. Thompson aux Bell Labs en 1972.
- Une mine d'infos : <http://www.lysator.liu.se/c/>

## C pourquoi ?

[MODE publicité=ON]

- Très utilisé dans l'industrie malgré certains défauts/difficultés.
- « *Souple* » (programmation haut et bas niveau).
- Disponible quelle que soit (à 99.9999%) l'architecture.
- Seul moyen « simple » de piloter du matériel.
- Excellentes performances (sauf sur un algo pourri bien sûr).

[MODE publicité=OFF]

[MODE incon vénient=ON]

- Gestion de la mémoire explicite.
- Structures de données de base rudimentaires.
- Fait apparaître des détails de compilation du langage.

[MODE incon vénient=OFF]

## Python vs C

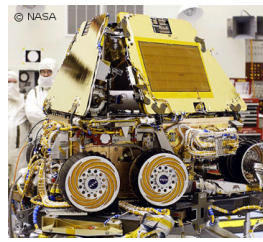
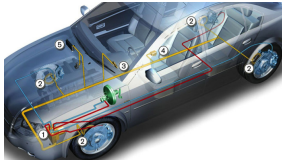
```
up_to_number = 10000
print("Prime numbers up to " + str(up_to_number) +
      ":")
for number in range(2, up_to_number + 1):
    isprime = True
    for divisor in range(2, number - 1):
        remainder = number
        while (remainder >= divisor):
            remainder = remainder - divisor
            if (remainder == 0):
                isprime = False
    if isprime:
        print(number)

#include <stdio.h>
#include <stdbool.h>
#define UP_TO_NUMBER (10000)
int main () {
    unsigned int number ;
    printf ("Prime numbers up to %d:\n", UP_TO_NUMBER) ;
    for (number = 2; number <= UP_TO_NUMBER; number++) {
        unsigned int divisor ;
        bool isprime = true ;
        for (divisor = 2 ; divisor < number; divisor++) {
            unsigned int remainder ;
            remainder = number ;
            while (remainder >= divisor) {
                remainder = remainder - divisor ;
            }
            if (remainder == 0) isprime = false ;
        }
        if (isprime == true) printf ("%d\n", number) ;
    }
    return (0) ;
}
```

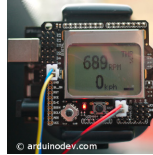
```
mymac:/tmp$ time python allprimes.py > /dev/null
real 0m42.976s
user 0m42.960s
sys 0m0.014s
mymac:/tmp$ time ./allprimes.x > /dev/null
real 0m0.887s
user 0m0.885s
sys 0m0.002s
```

## Contrôler des systèmes physiques

Matériel commandé par programme



(Loading gear.mp4)



## L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
    printf ("Eat at Joe's\n") ;
    return (0) ;
}
```

- La plus simple manière d'afficher une simple chaîne de caractères.
- Équivalent au programme Python :  
print "Eat at Joe's"

## L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
    printf ("Eat at Joe's\n") ;
    return (0) ;
}
```

- #include <> : Demander accès aux fonctions entrée/sortie (printf).
- /\* ... \*/ : Commentaires.
  - Ignorés par le langage.
  - **Mais** pas par les lecteurs!
  - Servent à la **documentation** du code.
  - **Documentez** vos programmes!
- Contrairement à Python, espaces non significatifs...
- mais continuez à **indenter** votre code!

## L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
    printf ("Eat at Joe's\n") ;
    return (0) ;
}
```

- Fonction main : **Le point d'entrée** du programme.
- Toujours **automatiquement** appelée en **premier**.
- Un programme en C doit **toujours** contenir **une** fonction **main**.
- Type int (entier) au début : type renvoyé par main.
- Pour le main : **toujours un entier**.

## L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
    printf ("Eat at Joe's\n") ;
    return (0) ;
}
```

- Fonction printf : affiche un message dans le terminal.
- "Eat at Joe's\n" : chaîne de caractères (à afficher).
  - ▶ \n : représente le caractère « retour à la ligne ».
- ; Point-virgule : **fin d'instruction**, marque la **séquence**.

## L'incontournable premier programme en C

joe.c

```
#include <stdio.h> /* Pour accéder à la fonction printf. */

int main ()
{
    printf ("Eat at Joe's\n") ;
    return (0) ;
}
```

- Retour de la fonction main :
  - ▶ Code de sortie du programme.
  - ▶ Existe pour toutes les commandes Unix
  - ▶ Permet de tester si le programme s'est exécuté normalement.
  - ▶ Convention : 0 ≡ « OK ».

## Compilation et exécution d'un programme

- Édition du source :
  - `> emacs joe.c &`
  - emacs : Éditeur de texte, permet de taper le source.
  - « *Human-readable* ».
  - Vous pouvez utiliser n'importe quel éditeur (vi, vim, joe...)
- Compilation :
  - `gcc joe.c` ou `gcc joe.c -o first.x`
  - gcc (compilateur) transforme le source en exécutable.
  - « *Computer-readable* » : compilateur ≡ traducteur.
  - Par défaut, nom de l'exécutable = « a.out ».
  - Option `-o` : permet de spécifier un nom d'exécutable.
- Exécution :
  - `./a.out` ou `./first.x`
  - ... en fonction du nom de votre exécutable.
  - Exécute votre programme dans le répertoire courant.
  - « `./` » devant car le répertoire courant n'est pas dans le PATH.

## Le compilateur gcc

- Compilateur : programme transformant un source en exécutable.
- Dispose de nombreuses options dont :
  - `-W` : Activer des warnings (`-Wall` : tous, **plus que recommandé**).
  - `-g` : Activer les infos de debug.
  - `-I` : Ajouter des répertoires où trouver les entêtes (`#include`).
  - `-c` : Compile sans « *linker* » (lorsque plusieurs fichiers sources).
  - `-L` : Transfère des options au « *linker* »
  - `-O0 -O1 -O2 -O3` : Optimise pas, peu, plus, beaucoup, (trop).
  - `-l` : Lier une bibliothèque (Ex : `-lm` pour la bibliothèque maths).
- Si tout s'est **bien passé**, le compilateur n'affiche **rien**.
- S'il affiche une **erreur**, l'exécutable n'est **pas** créé.
- S'il affiche un/des warning/s, l'exécutable **est** créé, mais **risque de ne pas fonctionner correctement**.

## Les messages d'erreur et d'avertissement

- **Erreurs** et **avertissements** sont détaillés par gcc !
  - **nature** de l'erreur,
  - **numéro de ligne** du source incriminée.



## Quelques constructions élémentaires

## Expressions (de base) du langage

- À partir d'expressions de base, on combine les expressions.
- Expressions de base : les variables et les littéraux.
  - Variables : toute variable **déclarée** et de portée accessible.
  - Entiers : `4 -5`
  - (~ Booléens : `true false`)
  - Caractères : `'U' 'n'`
  - Flottants : `-4.6 5e-12`
  - Chaînes : `"plop" "\tGlop\n"`

## Composition d'expressions

- Arithmétique : entre entiers et/ou flottants.
  - +, -, /, \* avec leur sens « habituel », % (modulo)
  - Ex : `4 - (5 * 7)`
- Relationnel (« tests ») : entre expressions de même type.
  - Rem : conversions implicite entre scalaires (entiers, flottants, caractères et booléens).
  - == (égalité), != (inégalité), <, >, <=, >=.
  - Ex : `y <= 5 (6 * 4) < x`
- Logique : entre booléens (donc implicitement, entiers).
  - && (et), || (ou), ! (négation).
  - Ex : `(y <= 5) && ((6 * 4) < x)`
  - && et || sont « paresseux ».
- Binaire (« bit-à-bit ») : entre entiers / caractères.
  - Fonction logique opérant **sur chacun des bits** de la représentation.
  - ~ (négation), ^ (ou exclusif), & (et), | (ou).
  - Ex : `(~x ^ y) ^ 0xFE`



## La mémoire

- Avant d'être exécuté, le programme est chargé en mémoire.
- Un « *pointeur* » indique où en est l'exécution.
- À chaque instruction il avance d'une « *case* ».
- Il y a parfois des sauts (appels de fonctions, tests, boucles. . .).
- Pendant l'exécution, le programme peut aussi réserver de la mémoire :
  - **Déclaration** (création) de **variables**, réceptacles d'information.
  - À chaque variable correspond une « *case mémoire* ».

## Nécessité de typage

- Mémoire : contient de l'information codée en **binaire**.
- Nous avons vu **quelques** formes de données.
- ⇒ un octet en mémoire peut avoir différentes significations.
- L'octet 01010111 peut représenter :
  - l'entier 87 ( $= 64 + 16 + 4 + 2 + 1$ ),
  - le flottant  $4.85 \cdot 10^{-270}$ ,
  - (le caractère 'W'),
  - une instruction en langage machine,
  - une adresse mémoire, etc . . .
- ⚠ En C, ce **n'est pas** comme en Python !
- Il faut associer un **type** à chaque case mémoire, chaque variable !
- C'est son **type** (lors de la **déclaration**) qui va définir sa **signification** réelle.

## Types de base en C

- **char** : caractères (on y reviendra).
- **int** : entiers (on y reviendra)..
- **float** : flottants (simple précision).
- **double** : flottants (double précision).
- Chaînes de caractères pas « *natives* » (on y reviendra).
- Pas de « vrais » booléens : utilisation d'entiers.
  - **0** ≡ « **faux** ».
  - **Autre ≠ 0** ≡ « **vrai** ».
  - Type **bool** avec les constantes **true** et **false** disponibles en utilisant le fichier d'entête **stdbool.h**.

## Déclaration et initialisation de variable (1)

```
int main ()
{
    float c ;
    int i, j = 42 ;
    double b ;
    b = 0.00345 ;
    c = 3.14159 ;
    return (0) ;
}
```

- On **déclare** une variable en donnant son type suivi de son nom.
- Cela réserve une case mémoire contenant *a priori* une valeur **quelconque**.
- L'**initialisation** fixe sa valeur.
- Cela peut se faire dès la déclaration, ou après.

- Le tout c'est de ne **pas lire** dans une variable **avant son initialisation** !
- Sa valeur est « ce qu'il y avait dans la mémoire ».
- Ça n'a aucun sens de l'utiliser, c'est une erreur.

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

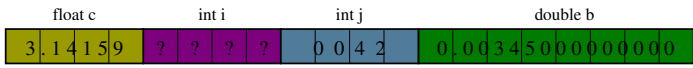
---

---

## Déclaration et initialisation de variable (2)

```
int main ()
{
    float c ;
    int i, j = 42 ;
    double b ;
    b = 0.00345 ;
    c = 3.14159 ;
    return (0) ;
}
```

- Avec ce code, la mémoire ressemble (+/-) à :



---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

## L'affectation

- Dénotée par le signe **=** (à ne pas confondre avec **==**).
- Donne une valeur à une variable.
- Besoin de « compatibilité » entre type de la case mémoire et type de la valeur à stocker !
- Si les types sont les mêmes, aucun problème.
- Si les types sont « compatibles », transformation implicite :
  - ▶ Entier ↔ entier avec signe et/ou taille différents,
  - ▶ Flottant ↔ entier,
  - ▶ Entier ↔ caractère.
- ... **mais** attention aux problèmes de **précision** et **débordement** !
- Et sinon, les types n'ont rien à voir (ex : chaîne à la place d'un entier) : erreur de typage par le compilateur.

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---


## Instructions d'affectation-arithmétique abrégées

- Il existe des versions courtes des opérations « courantes » :

```
int i = 0 ;

i++ ;           // i = i + 1 ;
i-- ;           // i = i - 1 ;
i += 3 ;        // i = i + 3 ;
i -= 3 ;        // i = i - 3 ;
i *= 3 ;        // i = i * 3 ;
i /= 3 ;        // i = i / 3 ;
i %= 3 ;        // i = i % 3 ;
i |= 3 ;        // i = i | 3 ;
i &= 3 ;        // i = i & 3 ;
i ^= 3 ;        // i = i ^ 3 ;
```

## Instruction conditionnelle (if)

- `if (expression) { instruction(s); }` :
  - `instruction(s)` exécutée(s) si `expression` renvoie  $\neq 0$ .
- `if (expression) { instruction1(s); } else { instruction2(s); }` :
  - `instruction1(s)` exécutée(s) si `expression` renvoie  $\neq 0$
  - sinon, `instruction2(s)` exécutée(s).
- Condition **toujours entre parenthèses!**
- Remarquez la construction `{ ... }` : **groupement de plusieurs instructions** : « bloc ».
  - ▶  **Différent** de Python où l'**indentation** jouait ce rôle!

## « Tant que » ... faire : la boucle while

- `while (expression) { instruction(s) }`
- Sémantique :
  - 1 Si `expression` s'évalue en « vrai »
    - 1.1 Exécuter `instruction(s)`.
    - 1.2 Retourner en 1.
- Ex : 
$$U_{n+1} = \begin{cases} \frac{U_n}{2} & \text{Si } U_n \text{ est pair} \\ 3 * U_n + 1 & \text{Sinon} \end{cases}$$

syracuse.c

```
int main ()
{
    int x = 134560 ;
    while (x != 1) { // Répéter tant que x est différent de 1.
        if (x % 2 == 0) // Si x est pair.
            x = x / 2 ; // On le divise par 2.
        else x = 3 * x + 1 ;
    }
    return (0) ;
}
```

## Sortie à l'écran printf (1)

- Nécessite : #include <stdio.h>
- printf ("Eat at Joe's\n");
- printf ("%d = %d + %d\n", x + y, x, y);
- Format= chaîne de caractères contenant (ou pas) des séquences %\.

%d un int  
%ld un long int en décimal  
%u un unsigned int en décimal  
%x un int en hexadécimal  
%f un float  
%lf un double  
%e un double en notation scientifique  
%.7lf un double avec 7 chiffres après la virgule  
%07d un int en décimal sur 7 digits (remplissage frontal avec des 0)  
% 7d un int en décimal sur 7 digits (remplissage frontal avec des ' ')  
%c un char (comme caractère ASCII, pas comme entier)

---

---

---

---

---

---

---

---

---

---

---

---

## Sortie à l'écran avec printf (2)

```
#include <stdio.h>

int main ()
{
    printf ("%d\n", 42) ;           42
    printf ("%ld\n", 42) ;         42
    printf ("%u\n", 42) ;          42
    printf ("%x\n", 42) ;          2a
    printf ("%f\n", 42.0) ;        42.000000
    printf ("%e\n", 42.0) ;        4.200000e+01
    printf ("% .7f\n", 42.0) ;     42.0000000
    printf ("%07d\n", 42) ;        0000042
    printf ("% 7d\n", 42) ;        42
    printf ("%c\n", 42) ;          *
    return (0) ;
}
```

---

---

---

---

---

---

---

---

---

---

---

---