

Éléments de syntaxe du langage C, partie 1

Sébastien Jean

IUT de Valence
Département Informatique

v1.0, 1^{er} octobre 2025

Le langage C

- Le langage C a été proposé en 1972 par Dennis Ritchie et Kenneth Thompson (puis Brian Kernighan)
- Il a été standardisé en 1989, il évolue sensiblement depuis

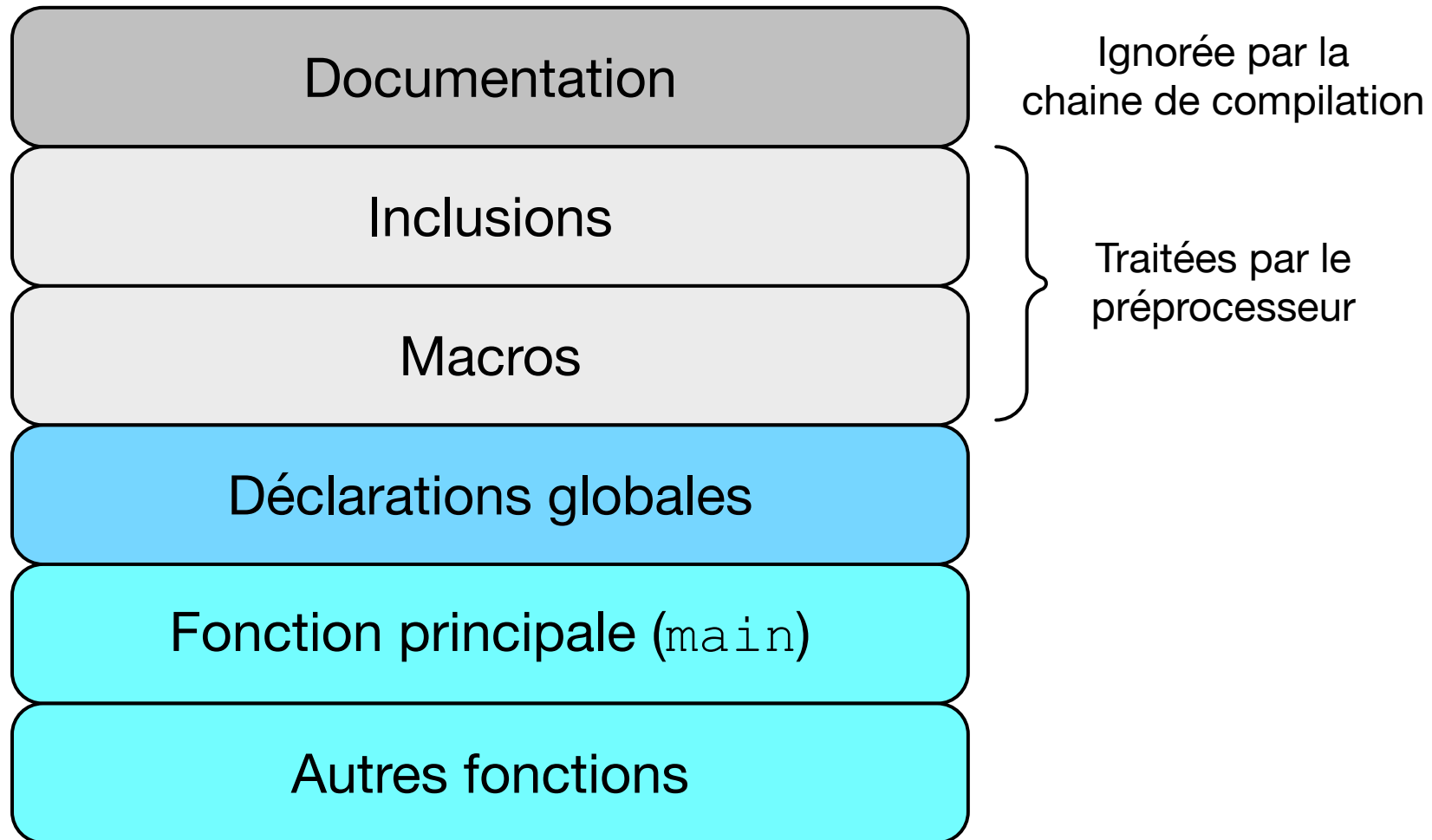
Timeline of C language

Year	Informal name	Official standard
1972	first release	—
1978	K&R C	—
1989, 1990	ANSI C, C89, ISO C, C90	ANSI X3.159-1989 ISO/IEC 9899:1990
1999	C99, C9X	ISO/IEC 9899:1999
2011	C11, C1X	ISO/IEC 9899:2011
2018	C17, C18	ISO/IEC 9899:2018
2024	C23, C2X	ISO/IEC 9899:2024
TBA	C2Y	



Structure d'un programme C

- Un **programme C** est structuré en **6 sections**



Structure d'un programme C : exemple

Documentation

Inclusions

Macros

Déclarations globales

Fonction principale (main)

```
/**  
 * fichier: classe_americaine.c  
 * auteur: Georges Abitbol  
 * description: affichage avec classe.  
 */  
  
#include <stdio.h>  
  
#define OUICHE 41  
  
int lorraine = 1;  
  
int main(void) {  
    printf("%d ", OUICHE + lorraine);  
    return 0;  
}
```

Documentation

```
/**  
 * fichier : classe_americaine.c  
 * auteur : Georges Abitbol  
 * description : affichage avec classe.  
 */
```

- Donner des **informations sur le code** (description, auteur, version, ...)
- **Syntaxe particulière** pour pouvoir plus tard être traitée par des **outils de documentation** (*Doxygen*, ...)



Interlude : commentaires

- Les **commentaires** sont **utiles aux développeurs**, ils ne sont **pas traités** par la chaîne de compilation
- Commentaire sur **ligne unique** :



Comments please!

Exemple

```
// Ceci est un commentaire sur une seule ligne
```

- Commentaire sur **plusieurs lignes** :

Exemple

```
/*  
Ceci est un commentaire  
sur plusieurs lignes  
*/
```

Inclusions

- N.B. : en C, toute ligne commençant par **#** n'est **pas une instruction valide** mais une **directive** destinée à être traitée par le **préprocesseur**
- Une **directive d'inclusion** permet d'intégrer dans le code les **signatures de fonctions** définies dans un **autre fichier**
 - N.B. : un fichier **.h** est un **fichier d'en-tête** (*header* ne contenant **que** des signatures de fonctions)

Exemple

```
#include <stdio.h>
```

- ici, `stdio.h` contient notamment la signature de la fonction `printf` (le code de cette fonction est dans une bibliothèque appelée *libc*)

Macros

- La **directive de définition de macro** respecte la syntaxe

```
#define a b
```

- Elle indique au préprocesseur de **remplacer dans le code toutes les occurrences** de a par b
 - Cela peut servir à définir des constantes

Exemple

```
#define OUICHE 41
```

Déclarations globales

- Les **déclarations globales** sont de 2 types :
 - Déclarations de **variable** ou de **constante**
 - **Signatures de fonctions** définies après la fonction principale main
- Une variable ou une constante définie **globalement** peut être vue/utilisée depuis **n'importe quelle** fonction du programme
- Une fonction dont la signature est définie **globalement** peut être appelée depuis **n'importe quelle** fonction du programme

Déclarations de variables et de constantes

- N.B. : toute séquence se terminant par ; est une **instruction**
- Une **déclaration de variable** respecte la syntaxe **type identifiant;**
 - Il est possible de lui **affecter** une valeur (avec =) lors de sa déclaration (← en pseudo code)
- Une constante est définie avec le mot clé **const**

Exemple

```
int lorraine;           // déclaration de variable  
  
double temperature = -3.5; // idem avec affectation  
  
const double PI = 3.14157; // déclaration de constante
```

Règles sur la construction des identifiants

- Un **identifiant** est une **suite de caractères** parmi un ensemble de caractères **autorisés** :
 - Les lettres **minuscules** ou **majuscules** (non accentuées),
 - Majuscules et minuscules différenciées (sensible à la casse)
 - Les **chiffres**,
 - le caractère **_** (*underscore*)
- Le **premier caractère ne peut pas être un chiffre**
 - De préférence cela ne doit pas être **_**
- Il est déconseillé d'avoir un identifiant de plus de 31 caractères

Types primitifs en C : entiers

- **4 types entiers** (principaux)
 - Les valeurs peuvent être considérées comme **signées** (signed) ou **non signées** (unsigned)

Type	Description	Taille (en bits)	Intervalle (signé)	Intervalle (non signé)	Par défaut
char	caractère / octet	8	$[-128, +127]$	$[0, 255]$	non signé
short int	entier court	16	$[-32768, +32767]$	$[0, 65535]$	signé
int	entier	32	$[-2^{31}, +2^{31}-1]$	$[0, 2^{32}-1]$	signé
long int	entier long	64	$[-2^{63}, +2^{63}-1]$	$[0, 2^{64}-1]$	signé

- Le type char sert à la fois à représenter des **caractères** et des **octets**
- N.B. : ici les tailles occupées par une valeur d'un type donné sont celles les plus courantes et minimales, elles peuvent varier

Types primitifs en C : entiers

- Les **entiers** peuvent être exprimés :
 - En **décimal** : -3, 127
 - En **binaire** : 0b00001111, 0b0101010101010101
 - En **hexadécimal** : 0x42, 0xABCD
- N.B. : le type des littéraux entiers est choisi par le compilateur par convention, mais on peut le fixer avec un suffixe (cf. doc)
- Les **caractères** sont exprimés soit par leur **code** (jeu de caractères) soit entre **apostrophes** ('a')
- Caractères spéciaux (extrait)
 - '\r' → fin de ligne
 - '\n' → saut de ligne
 - '\t' → tabulation

Types primitifs en C : flottants

- **2 types flottants** (principaux)
 - Les valeurs sont en général représentées selon la **norme IEEE754**

Type	Description	Taille (en bits)
float	flottant simple précision	32
double	flottant double précision	64

- N.B. : tous les flottants ne sont pas représentables, l'intervalle des valeurs possède des **trous** (entre les valeurs, autour du zéro, au delà d'une certaine
- Conséquence. : l'arithmétique flottante génère des erreurs de calculs

Types primitifs en C : bool et void

- 2 autres types

- bool → représente un **booléen** (1 bit) qui vaut soit **true** soit **false**
 - Avant la norme C23, il est nécessaire d'**inclure stdbool.h**
- void → qui ne s'applique (pour l'instant) qu'au **type de retour d'une fonction** pour indiquer qu'elle ne retourne **pas de résultat**

Opérateurs arithmétiques

Opérateur	Type opérande	Type opérande	Type résultat	Description
- (unaire)	entier		entier	opposé
	flottant		flottant	
+, -, *	entier	entier	entier	addition, soustraction, multiplication
	flottant	flottant ou entier	flottant	
/	entier	entier	entier	quotient de la division entière
	flottant	flottant ou entier	flottant	division flottante
%	entier	entier	entier	reste de la division entière

- Pour l'évaluation des expressions (ex : $3 + 4 * 6 * 2 - 7 \% 2$) :
 - *, / et % sont prioritaires sur + et -
 - Les opérateurs de même priorité sont évalués de gauche à droite

Déclarations de variables locales

- Une **paire d'accolades** délimite un **bloc**
- Un **bloc** peut contenir :
 - Des **déclarations de variables ou de constantes**
 - Des **instructions**

Signatures des fonctions

- La **signature d'une fonction** respecte la syntaxe :

`type-de-retour nom(paramètre1, paramètre2, ...)`

- Les **paramètres** respectent la syntaxe `type nom`
- Le **corps d'une fonction** est écrit dans un bloc

Exemple

```
int somme(int a, int b) {  
  
    return a + b;  
  
}
```

- L'instruction **return** permet de **retourner une valeur** (expression)

Appel de fonctions

- L'**appel d'une fonction** s'effectue en passant une **expression** comme valeur pour chacun des paramètres
- L'appel d'une fonction est une **expression** dont la **valeur** est du type déclaré comme type de retour

Exemple

```
int resultat = appelDeFonction(3.14, 'c');  
  
// ici la signature de la fonction peut être  
// int appelDeFonction(float f, char c);
```

Portée des variables

- Les **variables et constantes** peuvent être définies **globalement** (avant les fonctions) ou **localement** (dans une fonction ou structure de contrôle)

Exemple

```
double r; // globale

void f(int c ) { // paramètre local à la fonction
    int x;      // locale à la fonction
    double d;   // locale à la fonction
    ...
}

int main() {
    float x;    // locale à la fonction
    bool b;     // locale à la fonction
    ...
}
```

Portée des variables

- Une variable/constante **globale** est **visible (accessible) partout**
- Une variable/constante **locale** n'est visible que dans le bloc où elle est déclarée

```
double r;                                portée de r

void f(int c) {
    int x;                                portée de x et d
    double d;                             portée de c (paramètre)
    ...
}

int main() {
    float x;
    bool b;                               portée de x et b
    ...
}
```

- N.B. : les mêmes identifiants peuvent être utilisés si leur portée est disjointe

Passage de paramètres

- Les paramètres sont passés par copie de la valeur

Exemple

```
int v = 0;

int maFonction(int n) {
    n = n + 1;
    return n;
}

int main() {
    int x = maFonction(v);
    return v;
}
```

- Que retourne le main ? Que vaut x ?

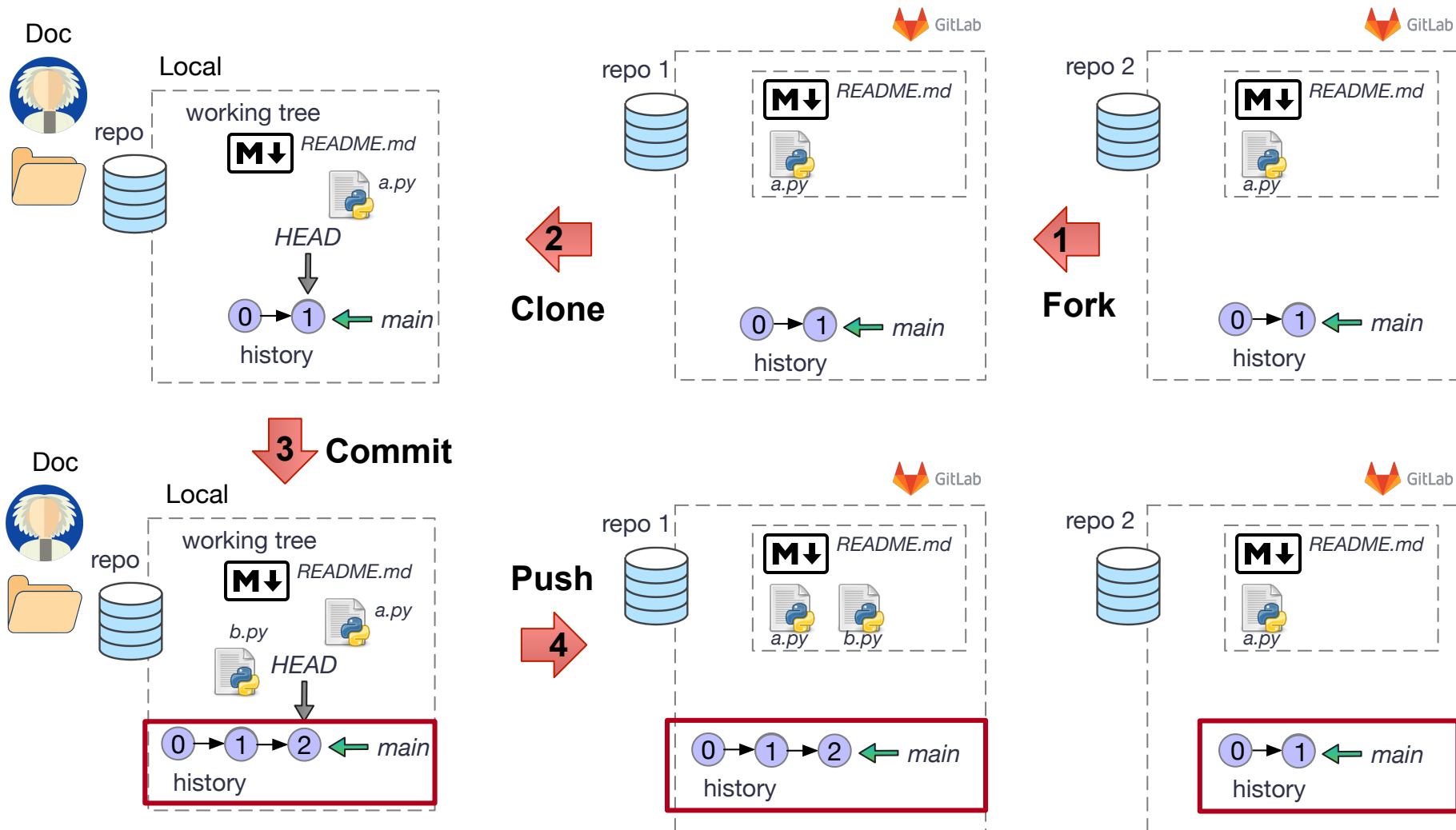
Affichage de valeurs sur la console

Exemple

```
printf(" un entier : %d , un réel : %f", i+1, r/3);  
// i est un entier  
// r est un flottant
```

- La fonction **printf** :
 - Prend en paramètres :
 - Une **chaîne de caractères** (exprimée entre guillemets) contenant des **marqueurs de format** pour l'insertion de valeurs typées
 - Un paramètre ayant pour valeur une **expression à insérer** dans chaque marqueur, dans l'ordre
 - **Affiche sur la console la chaîne de caractères (remplie) sans passer à la ligne**
- cf. page de manuel pour la syntaxe des marqueurs

Interlude : Duplication *Fork* d'un projet GitLab



- Un *fork* est un **duplicata d'un projet** (d'un compte Gitlab vers un autre compte Gitlab) qui devient **indépendant**

Interlude : Duplication *Fork* d'un projet GitLab



- **Faire un fork du projet**

`https://gitlab.iut-valence.fr/INFO-BUT/S1/R1-01/hello-c`

- **Cloner** le **fork** depuis VsCode et **ouvrir le dépôt**

Fichier .gitignore

- Dans l'historique d'un dépôt, on **exclut** les **ressources sensibles** (fichiers de configuration contenant des mots de passe) et les **ressources qui peuvent être re-générées**
- Pour **exclure des ressources**, on peut indiquer des **motifs d'exclusion** dans un fichier **.gitignore** placé à la racine du dépôt
- Avec le motif ****/build/**** on exclut toutes les ressources situés dans et sous un répertoire build, ce répertoire pouvant être situé à n'importe quel niveau de l'arborescence

Fichier .gitignore



- Créer dans HelloC un sous-répertoire build
 - Au même niveau que src
- Ouvrir le terminal *VSCode* et se placer dans HelloC
- Compiler `main.c` en faisant en sorte que l'exécutable soit produit dans `build` et soit nommé `helloC`
 - cf. page de manuel de gcc, option `-o`
- Vérifier que le fichier exécutable est bien ignoré

Fin !

