

Cours Langage C

```
#include <stdio.h>

int main(int argc, char ** argv) {
    printf("Bonjour !\n");
    return 1;
}
```

Langage C – Contexte Historique

Inventé par

Kernigan et Ritchie

Année 1970

Contemporain de la
création d'Unix

C, Hériter de:

- Algol
- Successeur direct de B
- ...

C, Ancêtre de:

- C++
- Java
- Java script
- Php
- Python
- ...

C est toujours en vie:

- Utilisé pour développer le noyau **Linux**
- Utilisé pour faire des applications rapides/ temps réel
- ...

Avantages/Inconvénients

- Rapides
- Langages à la fois bas niveau/haut niveau
- Bien documenté
- Standardisé
- Dépendant de la plate-forme
- Pas un langage objet
- Structure rigide
- Gestion de la mémoire...

Syntaxe

Instructions

Commentaires

- Deux manières de commenter

- Bloc
- Fin de ligne

- Les commentaires blocs ne sont pas emboîtable :

- `/* /* */ */`

```
/* *
```

```
* commentaire
```

```
* bloc
```

```
*/
```

```
//commentaire
```

```
//ligne
```

```
int a=10 ;//a var
```

```
//resultat
```


Types de base

- Entiers:
 - Flottants :
 - Booléens:
 - Caractères:
 - Chaînes de caractères:
- `int`
 - `float`
 - `double`
 - N'existe pas !
 - `char`
 - Tableaux de caractères

Le cas des booléens:

- Un entier est considéré à vrai si il est **différent de 0**
- Un flottant est vraie si **différent de 0.0**
- Idem pour les caractères

Exemple déclarations:

- `int a=1 ;`
- `char c='a' ;`
- `int b,c,d,e ;`
- `b=c=d=e=12 ;`
- `float p=2.0 ;`
- `double p2=3.14 ;`
- a est de type entier
- c est un caractère qui est initialisé à 'a'
- On déclare b c d et e comme des entiers
- = se propage et est associatif à droite
- p est un flottant
- p2 est un double (un flottant avec plus de précision...)

Options pour les nombres

- Chaque type de variable `int` peut être spécifiée avec les modificateurs suivants :
 - `short`
 - `long`
 - `unsigned`
- `short` : la taille de l'entier est plus petit (2 octets)
- `Long` : entier plus long (8 octets)
- `unsigned` : Le nombre est ≥ 0

Opérations sur les entiers

- $+$ $-$ $*$ $/$

- $++$

- $--$

- $+=$ $-=$

- $*=$ $/=$

- Opérations classiques

- Incrémente de 1

```
int i=0 ; i++ ; ++i ;
```

- Décrémente de 1

- Additionne à la variable l'ancienne valeur

```
int i=0 ;  
i+=3 ; // équiv  
// i=i+3
```


Structures Conditionnelles

if

- If fonctionne comme en algo
- Utilise la « *lazy evaluation* »
-

```
if( condition1) {  
  bloc1  
}else if(condition2) {  
  bloc2  
}else if(condition3) {  
  bloc3  
}else {  
  bloc4  
}
```


i++ et ++i

- En C il est possible d'utiliser l'opérateur ++ en préfixe ou en postfixe.
- La différence entre les deux est faible mais peut causer des problèmes

```
int a=2 ; int b=2 ;  
int i=0 ;  
a= i++ ;  
//i vaut 1  
i-- ;  
//i vaut 0  
b=++i ;  
//i vaut 1
```

Combien valent a et b ?

++

- $a = i++$;
- Le scénario est le suivant :
 - La valeur de i est renvoyée
 - Puis, i est incrémenté
- a vaut donc 0
- $b = ++i$;
- L'ordre des opérations est inversé :
 - i est incrémenté
 - La valeur (nouvelle) de i est renvoyée
- Et b vaut 1 !

Structures Conditionnelles

switch

- Pour les entiers ou les caractères, il existe une manière de tester plusieurs cas :
- Comme switch nécessite l'emploi des commandes **break** et **continue**. Il est préférable de ne pas l'utiliser

```
int a=12 ;  
switch(a) {  
case 1 :  
    a++ ; break ;  
case 4 :  
    a-- ; break ;  
case 12 :  
    a+=a ; break ;  
default :  
    a=0 ;  
};
```

Opérateurs logiques

- ==
- <
- >
- <=
- >=
- !=

- Égalité
- Inférieur
- Supérieur
- Inférieur ou égal
- Supérieur ou égal
- différent

Opérateurs logiques (suite)

- & &
- | |
- !
- Et logique
- Ou logique
- Négation

Opérateurs bit à bit

- `&` binaire sur chaque bit
- `|` binaire sur chaque bit
- `^` xor sur chaque bit
- `~` négation bit à bit
- ```
int a,b ; a=1 ;
b=5 ;
```
- ```
int c= a|b ;
```
- ```
int d= a&b ;
```
- ```
int e= a^b ;
```
- c vaut 5
- d vaut 1
- Et e vaut 4

Opération bit à bit (suite)

décalage

- << Décalage à gauche
- >> Décalage à droite

```
int a = 15 ;  
int b = a<<2 ;  
int c = a>>2 ;
```

- b est a décalé à gauche 2 fois : $a * 2 * 2$: b vaut ?
- c est décalé à droite 2 fois : $a/2/2$ c vaut ?

Précédence des opérateurs

1 ++ -- (suffixe)

2 ++ Prefix increment

Right-to-left

-- Prefix decrement

+ Unary plus

- Unary minus

! Logical NOT

~ Bitwise NOT

(type) Type cast

* Indirection (unary)

& Address-of
sizeof Size-of

3

* Multiplication

/ Division

% Modulo (remainder)

4

+ addition

- subtraction

5

<< Bitwise left shift

>> Bitwise right shift

Précédence des opérateurs (suite)

6

< Less than

<=

> Greater than

>=

7

==

!=

8 & (bitwise)

9 ^ (bitwise)

10 | (bitwise)

11 && (logical)

12 || (logical)

13

= Direct assignment

+=

-=

*=

/=

%=

<<=

>>=

&=

^=

|=

for

- For est la première instruction de boucle en C
- Elle se compose en 2 parties
 - ()
 - Bloc d'instructions

for : exemple

```
int i, j ;  
int z=0 ;  
for (i=0, j=2; i<10; i++, j++) {  
    z+=i+j ;  
}
```

- `for (Initialisation ; Test ; Incrément) {
 Bloc
}`
- Initialisation des variables déjà déclarée (ici i et j)
- Test : tant que le test est vrai le bloc est exécuté
- Incrément à chaque exécution les variables dans la partie incrément sont modifiées comme défini.

for (suite)

- Chacun des blocs
 - Initialisation
 - Test
 - Ou incrément peuvent être vide :
- `for (; ;) { i=i+1 ; }`
boucle infinie
- `for (; i<100 ;) { i++ ; }`
équivalent à **tant que (i < 100)**

for (fin)

- Dans la mesure où `for` gère l'incrément il est peu recommandé (interdit en fait) de modifier les variables d'index à l'intérieur du bloc.
- Même si `for` autorise d'avoir des parties vides on préférera un `while (test) {}` à un `for (; test;) {}` car **plus explicite** !

while

- Le while en C fonctionne de la même manière qu'en algo :
 - Il y a une condition d'arrêt
 - Il faut gérer soi-même les variables à modifier (incrémenter/décrémenter)

While

```
while (condition)  
{  
    Bloc  
}
```

- **Exemple:**

```
int a=0 ;  
while (a<100) {  
    a+=6 ;  
}  
return a ;
```

- La **condition** est une expression logique valide
- Le **Bloc** est un bloc d'instruction quelconque
- On doit s'assurer que nous n'obtenons pas de boucle infinie

while (suite)

- ```
int i=0 ;
while (1) {
 i++ ;
}
```
- Boucle infinie car 1 est valide au sens « *booléen* » de C
- ```
int j=0 ;  
while (0) {  
    j-- ;  
}
```
- Dans ce cas 0 est « *faux* » donc `j--` n'est jamais exécuté.

Do While

- Do ... While
- Une autre structure du tant que
- La condition est évaluée après l'exécution du bloc

```
int i=0 ;  
do {  
    printf("i%d\n",i) ;  
}  
while(i>0) ;
```

- Dans ce exemple on va exécuter une fois le `printf`. Dans le cas du `while` on ne l'aurait jamais fait !

Fonctions

- Une fonction a :
 - Un nom
 - Des paramètres
 - Un type de retour
 - Un bloc d'instructions

Fonctions

```
int  m(int a, int b) {  
    int c ;  
    if (a<b) {  
        c=a ;  
    } else {  
        c=b ;  
    }  
    return c ;  
}
```

- Le type de retour est **int**
- Le nom de la fonction est *m*
- Les paramètres sont deux entiers *a* et *b*
- La fonction renvoie la valeur du plus petit des deux

Fonctions

- Une fonction doit être déclarée avant d'être utilisée
- t2 est déclarée après être utilisée : pas correct

```
int test() {  
    int a=2+t2() ;  
    return a ;  
}  
  
int t2() {  
    return 4 ;  
}
```

Fonctions sans type de retour:

- Si on souhaite seulement faire des affichages (print). Le type de retour n'est pas toujours nécessaire Pour cela on dispose de `void`

```
void Affiche(int a) {  
    int i ;  
  
    for (i=0; i<a; i++++) {  
        printf("%d\n", i) ;  
    }  
}
```

Compilation

Code source → Assemblage → Édition de
Liens → Exécutable

Compilation :

en pratique

- Le compilateur sous Linux s'appelle gcc
- Il permet de :
 - Transformer le code source en assembleur
 - Transformer l'assembleur en code objet
 - Effectuer l'édition de liens

gcc : Syntaxe

- `gcc -s fichier.c` • Transforme le code C en assembleur (lisible) produit **fichier.s**
- `gcc -c fichier.s` • Transforme le fichier assembleur (lisible) en assembleur machine produit **fichier.o**

gcc : Syntaxe

- gcc -o fichier fichier.o
- Prend fichier.o et le lie avec les autres fichiers (bibliothèque et autres)
- gcc -o fichier fichier.c
- Toutes les étapes en une seule

Exemple : gcc -s fichier.c

```
#include <stdio.h>
int main(int argc, char **
argv) {

    printf("Bonjour\n") ;

    return 1;
}
```

```
.file "t1.c"
.section .rodata
.LC0 :
.string "Bonjour"
.text
.globl main
.type main, @function
main :
.LFB0 :
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
movq %rsp, %rbp
.cfi_offset 6, -16
.cfi_def_cfa_register 6
subq $16, %rsp
movl %edi, -4(%rbp)
movq %rsi, -16(%rbp)
movl
$.LC0, %edi
call puts
movl $1, %eax
leaveret
.cfi_endproc
.LFE0 :
.size main, .-main
.ident "GCC: (Ubuntu 4.4.3-
4ubuntu5.1) 4.4.3"
```

Programme C

Structure d'un programme En langage C

Fichiers d'en têtes

(headers → .h)

- Comme nous devons déclarer une fonction avant de pouvoir l'utiliser on peut découper la déclaration et la définition :
 - Déclaration de la fonction :

prototype : type de retour ; nom de la fonction ; types de paramètre ;
 - Définition de la fonction : idem plus le bloc d'instructions
- Chaque fonction doit être déclaré qu'une seule fois

Fichiers d'en têtes

- On pourra donc stocker les déclarations dans un fichier spécial (le fichier .h)
 - Ainsi que les constantes comme
`float pi=3.14 ;`
- Et les définitions dans un fichier .c
- Ainsi le code est préservé dans un fichier .c (on peut changer l'implémentation)
- Par contre les définitions des fonctions sont accessibles dans un fichier .h

Prototype:

- **Prototype:**

```
void test(int a, int b) ;
```

- **Définition :**

```
void test(int x, int y) {  
    int z=x*y+x ;  
    printf(' '%d\n',' ',z) ;  
}
```

Organisation des fichiers

- Un programme pourra être constitué de :
 - Plusieurs fichiers .c
 - Les fichiers .h associés
 - Et faire appel à des bibliothèques
- Le programme exécute une seule fonction, la fonction `main()`

Organisation des fichiers

- Donc parmi tous les fichiers .c de votre programme, une seule fonction main doit être définie.
- Les fichiers .c servent à « ranger » les fonctions dans des fichiers bien identifiés :
 - De manière à rapidement retrouver la fonction concernée et éventuellement la modifier
 - De ne recompiler que la partie nécessaire

Utilisation des fichiers d'en-tête:

code.h

```
#include <stdio.h>

void test(int x) ;
void aff(int y) ;
```

code.c

```
#include "'code.h'"
#include <stdio.h>
void test(int a){
    return a+1 ;
}
void aff(int z){
    printf("'%d'",z) ;
}
```


Pré-processeur

- Le pré-processeur est une partie du compilateur qui va modifier le code source
- Toutes les directives qui commencent par # font parties du pré-processeur
- `#define MAX 100`
à chaque fois que MAX sera vu dans le programme il sera remplacé par 100

Pré-processeur

- Les commandes du préprocesseur :
 - `#define NOM valeur`
 - `#define SYMBOLE`
 - `#ifdef SYMBOLE`
 - `#ifndef SYMBOLE`
 - `#endif`
 - `#undef SYMBOLE`
 - `#include 'mon_fichier.h'`

Commandes pré-processeur

- `#define NOM valeur`
 - Crée un symbole NOM et remplace NOM par la valeur correspondante
- `#define SYMBOLE`
 - Crée simplement SYMBOLE on peut ensuite tester son existence avec les commandes `#ifdef` et `#ifndef`
 - Utile pour les fichier .h

Commandes pré-processeur (suite)

- `#ifdef SYMBOLE` **et** `#ifndef SYMBOLE`
permettent respectivement de savoir si
un symbole a été définie ou pas
- Chaque commande inclus un bloc de
choses à faire et se termine par :
- `#endif`

Commandes pré-processeur (suite)

- `#include` permet d'inclure entièrement le contenu d'un fichier
- `#include 'mon_fichier.h'` quand `mon_fichier.h` est dans le répertoire local (i.e. Le même que le fichier `c`)
- `#include <stdio.h>` car le fichier `stdio.h` est un fichier standard et il est stocké dans un répertoire où le compilateur sait le trouver.

Macros utiles

- `__FILE__` → donne le nom du fichier (chaîne de caractère).
- `__LINE__` → donne la ligne du fichier où la macro apparaît (entier).
- `__DATE__` → donne la date de la compilation
- `__TIME__` → donne l'heure de compilation

Macros Utiles

- `#define DEBUG printf(''fichier %s ; ligne %d'', __FILE__, __LINE__) ;`
- Ensuite dans le code on peut appeler la macro:
DEBUG
code problématique
DEBUG
- Ainsi si le programme « plante » on peut isoler d'où vient le problème.

Pré-processeur (suite)

- On peut voir le résultat du pré-processeur avec gcc
- `gcc -E fichier.c`
 - Affiche le fichier.c une fois l'étape du préprocesseur effectuée

Exemple d'utilisation

- Dans un fichier `math_const.h` on définit la constante `pi=3.14` ;
- Dans un fichier `cercle.h` on utilise `pi` pour la fonction :
 - `Perimetre()`
 - `Aire()`
- Dans `cercle.h` on a :

```
#include 'math_const.h'
```
- Dans un autre fichier on a besoin de `cercle.h` et `math_const.h` on a donc :

```
#include 'math_const.h'
#include 'cercle.h'
```
- Problème : `math_const` est déjà inclus dans `cercle.h` on va définir deux fois `pi`

- Pour palier le problème dans chaque fichier .h

- On ajoute au début :

```
#ifndef FICHER_H_  
#define FICHER_H_  
DEFINITION  
classique :  
#endif
```

- Dans l'exemple on a :

```
#include 'math_const.h'  
#include 'cercle.h'
```

- Du coup, lors du premier include :

```
MATH_CONST_H_  
est définie
```

- Lors du second include le bloc ne sera pas ré-inclus.

Entrées/Sorties

Fonctions `printf` et `scanf`

stdio.h

- stdio.h est le fichier en-tête qui définit toutes les fonctions pour les entrées/sorties basiques :
 - Afficher/Imprimer une variable dans le terminal/un fichier ...
 - `printf`
 - Lire et mettre le résultat dans une variable depuis le terminal/ un fichier...
 - `scanf`

printf

- La fonction **printf** permet d'afficher du texte dans le terminal ainsi que des variables :
 - Exemple simple :

```
printf(' 'Bonjour \n') ;
```
 - Exemple plus compliqué :

```
int i=0 ; char c='a' ;  
printf(' 'i vaut %d c vaut %c\n',i,c) ;
```

printf : modificateurs

- Tout ce qui est de la forme %x sera remplacé par la valeur de la variable et sera affiché selon la forme x.
- La liste des est la suivante :
 - d or i Signed decimal integer
 - u Unsigned decimal integer
 - o Unsigned octal
 - x Unsigned hexadecimal integer
 - X Unsigned hexadecimal integer (uppercase)
 - f Decimal floating point, lowercase
 - F Decimal floating point, uppercase

printf : modificateurs (suite)

- e Scientific notation (mantissa/exponent), lowercase
- E Scientific notation (mantissa/exponent), uppercase
- g Use the shortest representation: %e or %f
- G Use the shortest representation: %E or %F
- a Hexadecimal floating point, lowercase
- A Hexadecimal floating point, uppercase
- c Character
- s String of characters
- p Pointer address
- % A % followed by another % character will write a single % to the stream

Caractères spéciaux :

- `\n` → retour à la ligne
- `\t` → Tabulation (en général 8 caractères)
- `\b` → retour en arrière (backspace)
- `\c` → passage à la ligne
- `\f` → retour au début de la ligne
- `\a` → Alert (bip)
- `\v` → Tabulation verticale

printf : Exemple

- Exemple précédent:

```
int i=0 ; char c='a' ;  
printf(''i vaut %d c vaut %c\n'',i,c) ;
```

- i est un entier donc on utilise %d (ou %i)
- c est un caractère donc on utilise %c
- Les variables sont ajoutées à la fin par ordre d'apparition dans la chaîne de caractère qui doit être affichée.

scanf

- `scanf` permet de lire les valeurs saisies par l'utilisateur.
- Le format est presque le même que `printf`

- Exemple :

```
int entier ; float nflottant ;  
printf('Saisissez un entier : ' ) ;  
scanf('%i', &entier) ;  
scanf('%f', &nflottant) ;
```

Scanf

- Dans la mesure où scanf s'attend à rencontrer un certain type de donnée des erreurs peuvent arriver si l'utilisateur commet une erreur
- Exemple d'erreur :
on exécute :
`scanf("%d", &a) ; //a est int`
- Et l'utilisateur tape « abc »