

# LU1IN002 éléments de programmation 2

## Cours 1

Cours : Jean-Lou Desbarbieux, François Bouchet et  
Mathilde Carpentier  
et toute l'équipe de l'ue  
LU1IN002 SU 2022/2023

# Présentation du module

- ▶ Module niveau 1 de **9 ECTS**
- ▶ Objectifs principaux :
  - ▶ Consolidation de notions de programmation impérative (alternatives, boucles, variables, fonctions)
  - ▶ Représentation de la mémoire, gestion explicite de la mémoire
  - ▶ Spécificités du langage C
  - ▶ Structure de données autoréférentielles : listes, piles, files
  - ▶ Bonnes habitudes de programmation (tests, structuration)
  - ▶ Notions d'algorithmique

# Calendrier

- ▶ 11 semaines de cours (1 par semaine), 11 semaines de TD (1 par semaine), 11 semaines des TME (2 par semaine)
- ▶ Début des CM de LU2IN002 : 16/1/2023. Fin le 14/4/2023
- ▶ Début des TD de LU2IN002 : 23/1/2023. Fin le 17/4/2023
- ▶ Début des TME de LU2IN002 : 30/01/2023. Fin le mardi 9/5/2023 (rattrapage du lundi de Pâques)

**Poly de TD** à aller chercher à l'association étudiante ALIAS en salle 14-15-506. Horaires : 12h45-13h45 & 18h-19h.

# Évaluation

Modalité de contrôle des connaissances :

- ▶ 5% pour les quizz de cours sur Moodle
- ▶ 5% pour les TP rendus sur Moodle
- ▶ 15% TME solitaire 1, 45min, la semaine du TME4
- ▶ 25% TME solitaire 2, 1h30, la dernière semaine de TP
- ▶ 50% pour l'examen final

## Déroulé de l'UE (à titre indicatif)

1. Noyau impératif des langages : de Python à C
2. Principes de fonctionnement des ordinateurs
3. Tableaux, pointeurs et allocation
4. Algorithmes avec les tableaux
5. Arithmétique de pointeurs et chaînes de caractères
6. Enregistrement (structures) et pointeurs
7. Structure de données linéaires (liste, files d'attente) (4 semaines)

# Bibliographie

- ▶ Apprenez à programmer en C , Mathieu Nebra, Collection OpenClassrooms, Eyrolles, 2015
- ▶ Programmer en langage C : Cours et exercices corrigés , Claude Delannoy, Collection Noire, Eyrolles, 2016
- ▶ Le langage C - Norme ANSI , Brian W. Kernighan et Dennis M. Ritchie, Collection Sciences Sup, Dunod, 2014
- ▶ ...

# Outils informatiques utilisés

Environnement : Linux.

- ▶ éditeurs : gedit/emacs/vi/gvim
- ▶ compilateur : gcc
- ▶ débogueur : gdb, ddd

# Le Langage C : historique

- ▶ Le langage C a été inventé en 1972 par Dennis Ritchie et Ken Thompson (AT&T Bell Laboratories) pour réécrire Unix et développer des programmes sous Unix.
- ▶ En 1978, Brian Kernighan et Dennis Ritchie publient la définition classique du C dans le livre The C Programming language .
- ▶ C est une norme ANSI (ANSI-C) depuis 1989 et un standard ISO depuis 1990, standard étendu en 1999 (C99), 2011 (C11) et en 2018 (C18).



## Caractéristiques du langage :

- ▶ impératif
- ▶ bas-niveau
- ▶ typé
- ▶ compilé

# Comparaison de différents langages


	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy efficiency across programming languages : how do energy, time, and memory relate ?. In Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (pp. 256-267).

# Premier programme



The image shows a screenshot of a code editor window. The title bar at the top reads 'Hello\_World.c' and 'Free Mode'. The editor contains the following C code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Premier programme */
5
6  int main(){
7
8      printf("Hello World\n");
9
10     return 0;
11 }
12
```

The code is color-coded: preprocessor directives are red, comments are purple, the function signature is blue, and the printf format string is pink. Line numbers 1 through 12 are visible on the left margin. The status bar at the bottom indicates the file is saved on 21/01/2021 at 17:38:36, with 119 lines out of 16 / 12, and is at 100% zoom.

# Premier programme



```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Premier programme */
5
6  int main(){
7
8      printf("Hello World\n");
9
10     return 0;
11 }
12
```

src — -bash — 59x15

```
mathilde@abizarre-2:src/$gcc -Wall -o hello Hello_World.c
mathilde@abizarre-2:src/$./hello
Hello World
mathilde@abizarre-2:src/$
```

— > DEMO Hello\_World.c


# Fonctions, programmes, exécution

Au 1er semestre : environnement intégré mrpython

Au 2nd semestre : éditeur de texte

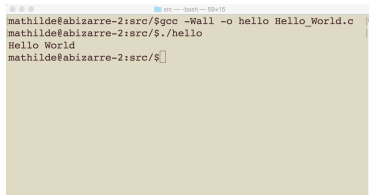
+ commande de compilation

+ commande d'exécution du programme



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* Premier programme */
5
6 int main(){
7     printf("Hello World\n");
8
9     return 0;
10 }
```

(a) Fichier texte contenant le code C



```
mathilde@abizarre-2:src/$gcc -Wall -o hello Hello_World.c
mathilde@abizarre-2:src/$./hello
Hello World
mathilde@abizarre-2:src/$
```

(b) Compilation et execution dans le terminal

Figure – Exemple de programme C, de sa compilation et de son execution

# Les fonctions

Les éléments à définir sont :

1. le type de la valeur de retour (**void** si rien n'est retourné)
2. le nom de la fonction
3. les arguments éventuels, avec leur type
4. le corps de la fonction (entre **accolades**)

```
1 int main() {  
2     printf("Hello World\n");  
3     return 0;  
4 }
```

# Types que nous utiliserons

Type	Signification	Taille (o)	Plage de valeurs	Exemple
char	Caractère	1	-128 à 127	'a'
int	Entier	4	-2 147 483 648 à 2 147 483 647	25
float	Simple précision	4	+/- 1.175494e-38 à 3.402823e+38	3.14

## ⚠ Attention

'a' est différent de "a" en langage C

car ' et " définissent deux types différents :

- ▶ 'a' est **un caractère** (**char**)
- ▶ "a" est une **chaîne de caractères** (comme "Hello World \n").

⚠ Attention Pas de type booléen. On utilise le type entier avec 0 comme valeur Faux et tout le reste comme Vrai.

# Tous les types simples

Uniquement pour information. Tous ne sont pas à connaître, seuls les précédents seront utilisés

Type	Signification	Taille (o)	Plage de valeurs
char	Caractère	1	-128 à 127
unsigned char	Caractère	1	0 à 255
short int	Entier court	2	-32768 à 32767
uns. short int	Entier court non s.	2	0 à 65535
int	Entier	2 (16 b) 4	-32768 à 32767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (16 b) 4 (32 et 64 b)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4 8 (64 b)	-2 147 483 648 à 2 147 483 647 -9 223 372 036 854 775 080 à 9 223 372 036 854 775 807
uns. long int	Entier long non s.	4	0 à 4 294 967 295
float	Simple précision	4	+/- 1.175494e-38 à 3.402823e+38
double	Double précision	8	+/- 2.225074e-308 à 1.797693e+308
long double	Double préc. long	12	+/- 3.362103e-4932 à 1.189731e+4932



# Les fonctions

## Attention

Il faut toujours une (et une seule) fonction **main** par programme.

```
1 int main(){  
2     printf("Hello World\n");  
3     return 0;  
4 }
```

# Les fonctions

```
1 void Hello(){
2     printf(" Hello World\n");
3 }
4
5 int main(){
6     Hello(); //Appel de la fonction
7     return 0;
8 }
```

Pour être testée, la fonction Hello est appelée dans la fonction main. La fonction main est le point de départ du programme.

# Les fonctions : ordre

 **Attention** à l'ordre de définition des fonctions

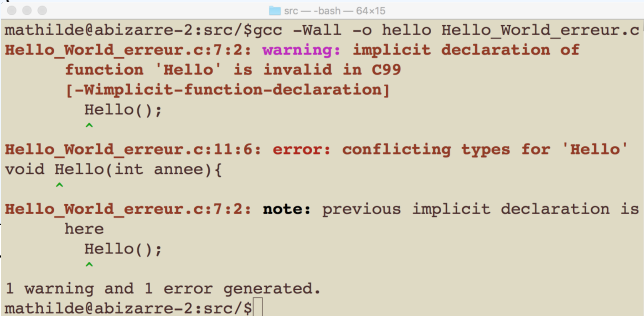
```
1 int main() {  
2     Hello();  
3     return 0;  
4 }  
5  
6 void Hello() {  
7     printf("Hello World\n");  
8 }
```

Ce code provoquera une erreur de compilation.

# Les fonctions : ordre

⚠ **Attention** à l'ordre de définition des fonctions

```
1 int main() {
2     Hello(
3     return
4 }
5
6 void Hello
7     printf
8 }
```



The screenshot shows a terminal window titled 'src -- -bash -- 64x15'. The user runs the command `mathilde@abizarre-2:src/$gcc -Wall -o hello Hello_World_erreur.c`. The output shows a warning and an error. The warning is: `Hello_World_erreur.c:7:2: warning: implicit declaration of function 'Hello' is invalid in C99 [-Wimplicit-function-declaration]`. The error is: `Hello_World_erreur.c:11:6: error: conflicting types for 'Hello'`. The note is: `Hello_World_erreur.c:7:2: note: previous implicit declaration is here`. The terminal also shows the output of `1 warning and 1 error generated.` and the prompt `mathilde@abizarre-2:src/$`.

```
mathilde@abizarre-2:src/$gcc -Wall -o hello Hello_World_erreur.c
Hello_World_erreur.c:7:2: warning: implicit declaration of
      function 'Hello' is invalid in C99
      [-Wimplicit-function-declaration]
      Hello();
      ^
Hello_World_erreur.c:11:6: error: conflicting types for 'Hello'
void Hello(int annee){
      ^
Hello_World_erreur.c:7:2: note: previous implicit declaration is
      here
      Hello();
      ^
1 warning and 1 error generated.
mathilde@abizarre-2:src/$
```

Ce code pr

# Les fonctions : arguments

```
1 void Hello(int annee){  
2     printf(" Hello World %d\n" , annee);  
3 }  
4  
5 int main(){  
6     Hello(2021); //Appel de la fonction  
7     return 0;  
8 }
```

# Les fonctions :prototypes

**void** Hello(**int** annee);

et

**int** main();

sont les **prototypes** des fonctions définis ci-dessous.

```
1 void Hello(int annee){  
2     printf(" Hello World %d\n" , annee);  
3 }  
4  
5 int main(){  
6     Hello(2021); //Appel de la fonction  
7     return 0;  
8 }
```

# La fonction `printf`

Exemples :

```
1 printf("un entier un entier %d et un float %f\n",  
    2021, 3.14);  
2 printf("un caractère %c \n", 'b');
```

Arguments :

- ▶ une chaîne de caractères entre " " (le format) contenant éventuellement %f pour les **float**, %d pour les **int** et %c pour les **char**
- ▶ les valeurs ou variables à afficher

# Variables : déclaration et affectation

En C, il faut obligatoirement **déclarer** les variables.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 float aire_triangle(float a, float b, float c) {
6     float p;
7     p = (a + b + c) / 2;
8     return sqrt(p * (p - a) * (p - b) * (p - c));
9 }
10
11 int main() {
12     float a=0, b=1, c=3, d=3.5;
13     a=aire_triangle(b,c,d);
14     printf("L'aire du triangle est %f\n", a);
15     return 0;
16 }
```

Les variables sont **locales** au bloc où elles sont déclarées.

Déclaration et initialisation peuvent être groupées.



# Opérateurs

Par ordre de priorité :

```
référence: () [] -> .  
unaire: ! ++ + - * & (type) sizeof  
arithmétique: * / %  
arithmétique: + -  
relationnels: < <= > >=  
relationnels: == !=  
logique: &&  
logique: ||  
affectation: =
```

 **Attention**

Pas d'opérateur puissance (\*\*) comme en python.

# Tous les opérateurs

Uniquement pour information. Tous ne sont pas à connaître, seuls les précédents seront utilisés

opérateurs	action
. et -> [] ( )	sélection de champ indixage appel de fonction
++ -- ~ ! + - & * (type) sizeof	pré- et post-incrémentation, pré- et post-décrémentation complément à 1 négation plus et moins unaire référence et indirection cast taille
* / %	multiplication, division modulo
+ -	addition et soustraction
<< et >>	décalage à gauche et à droite
< > <= >=	inférieur et supérieur inférieur ou égal et supérieur ou égal
== !=	égal et différent

opérateurs	action
&	ET bit à bit
^	OU exclusif bit à bit
—	OU inclusif bit à bit
&& —	ET logique OU logique
? :	conditionnel (ternaire)
=	affectation
+= -= *= /= %=	modification et affectation
,	évaluation séquentielle

## De Python à C : exemple de la fonction perimetre

```
1 def perimetre(largeur:int, longer:int) -> int:  
2     """hypothese : longueur >= largeur >= 0  
3     retourne le perimetre du rectangle defini par  
4     sa largeur et sa longueur."""  
5     return 2 * (largeur + longueur)
```

```
1 /* hypothese : longueur >= largeur >= 0  
2  retourne le perimetre du rectangle defini par  
3  sa largeur et sa longueur.*/  
4 int perimetre(int largeur, int longueur) {  
5     return 2 * (largeur + longueur);  
6 }
```

# De Python à C : spécification

## Méthode : exemple pour la fonction `perimetre` :

1. Que doit calculer la fonction ?
  - ▶ périmètre du rectangle défini par sa largeur et sa longueur.
2. Quels sont les paramètres (nombre, nom, type) ?
  - ▶ les **deux** côtés largeur et longueur,
  - ▶ tous les deux de type **int**.
3. Quelles sont les hypothèses sur les paramètres ?
  - ▶ *longueur*  $\geq$  *largeur*  $\geq$  0
4. Quel est le type de la valeur de retour ?
  - ▶ le périmètre du rectangle est de type **int**.

```
1 /* hypothese : longueur >= largeur >= 0
2    retourne le perimetre du rectangle defini par
3    sa largeur et sa longueur.*/
4 int perimetre(int largeur, int longueur) {
5     return 2 * (largeur + longueur);
6 }
```

# De Python à C

Ce qui change

## En-tête des déclarations (signature ou prototype)

```
1 def perimetre(largeur:int, longer:int) -> int:
```

C : typage explicite

```
1 int perimetre(int largeur, int longueur)
```

La signature (prototype) d'une fonction précise :

1. la valeur de retour
2. le nom de la fonction
3. le type des arguments

# De Python à C

## Ce qui change

**Commentaires** en C : `/* ...*/` pour commenter plusieurs lignes ou `//...` pour commenter une seule ligne (commentaires C++)

```
1  """hypothese : longueur >= largeur >= 0
2  retourne le perimetre du rectangle defini par
3  sa largeur et sa longueur."""
```

```
1  /* hypothese : longueur >= largeur >= 0
2  retourne le perimetre du rectangle defini par
3  sa largeur et sa longueur.*/
```

# De Python à C

Ce qui change

**Bloc :**

- ▶ Python : indentation, une instruction par ligne
- ▶ C : accolades, point virgule en fin d'instruction

```
1 def perimetre(largeur:int, longer:int) -> int:  
2     """hypothese : longueur >= largeur >= 0  
3     retourne le perimetre du rectangle defini par  
4     sa largeur et sa longueur."""  
5     return 2 * (largeur + longueur)
```

```
1 /* hypothese : longueur >= largeur >= 0  
2  retourne le perimetre du rectangle defini par  
3  sa largeur et sa longueur.*/  
4 int perimetre(int largeur, int longueur) {  
5     return 2 * (largeur + longueur);  
6 }
```

# Les structures de contrôle

- ▶ **if** / **else**
- ▶ **while** et **do ... while**
- ▶ **for**



## Alternative ( if et else)

```
if (expression) {  
    instructions;  
}  
else {  
    instructions;  
}
```

```
1 float valeur_absolue(float x) {  
2     float abs_x ;  
3     if (x >= 0) {  
4         abs_x= x;  
5     } else {  
6         abs_x=-x;  
7     }  
8     return abs_x;  
9 }
```

- ▶ Condition entre parenthèses
- ▶ Blocs délimités par des accolades

# Alternative ( if et else)

## Python

```
1 def valeur_absolue(x: float)
  → float:
2     """ retourne la valeur
    absolue de x.
3     """
4     # abs_x : Number
5     abs_x = 0
6     if x >= 0:
7         abs_x = x
8     else:
9         abs_x = -x
10    return abs_x
```

## C

```
1 float valeur_absolue( float x)
2 {
3     float abs_x ;
4     if (x >= 0) {
5         abs_x= x;
6     } else {
7         abs_x=-x;
8     }
9     return abs_x;
10 }
```

- Condition entre parenthèses
- Blocs délimités par des accolades

## Alternatives imbriquées

```
if (expression) {  
    if (expression) {  
        instructions;  
    } else {  
        instructions;  
    }  
}  
else {  
    if (expression) {  
        instructions;  
    } else {  
        instructions;  
    }  
}
```

# Alternatives imbriquées

```
1  int nb_sol(float a, float b, float c) {  
2      float d = b*b - 4*a*c;  
3      if (d == 0) {  
4          return 1;  
5      } else {  
6          if (d > 0) {  
7              return 2;  
8          } else {  
9              return 0;  
10         }  
11     }  
12 }
```

- ▶ Condition entre parenthèses
- ▶ Blocs délimités par des accolades
- ▶ Pas de `elif` : des `if` et `else` imbriqués

## Boucle **while**

```
while (expression)
{
    instructions;
}
```

```
1 int somme_entiers(int n){
2     int i = 1;
3     int s = 0;
4     while (i <= n){
5         s = s + i;
6         i = i + 1;
7     }
8     return s;
9 }
```

Comme pour l'alternative :

- ▶ parenthèses autour de la condition
- ▶ bloc délimité par des accolades

# Boucle **while**

```
1 def somme_entiers(n: int)  
    -> int:  
2     # i : int  
3     i = 1  
4     # s : int  
5     s = 0  
6     while i <= n:  
7         s = s + i  
8         i = i + 1  
9     return s
```

```
/* hypothèse: n >= 1 [...] */  
2 int somme_entiers(int n){  
3     int i = 1;  
4     int s = 0;  
5     while (i <= n){  
6         s = s + i;  
7         i = i + 1;  
8     }  
9     return s;  
10 }
```

- ▶ parenthèses autour de la condition
- ▶ bloc délimité par des accolades

## Boucles **do ... while**

```
1 int main(){  
2     int i = 1, n=1;  
3     do {  
4         printf(" i:%d\n" , i);  
5         i = i + 1;  
6     } while (i < n);  
7     return 0;  
8 }
```

- ▶ bloc délimité par des accolades
- ▶ condition **à la fin**, et entre parenthèses
- ▶ ne pas oublier le **" ;"** après la condition

Qu'affiche ce programme ?

## Boucle **for**

Forme synthétique de la boucle en C :

```
1  for ( expression1; expression2; expression3 )  
    {  
2      instructions;  
3    }
```

```
1  int i;  
2  for ( i = 1; i <= n; i++ ) {  
3      s = s + i;  
4  }
```

Les 3 éléments qui font tourner la boucle :

initialisation :  $i = 1$

condition :  $(i \leq n)$

incrément :  $i++$  (équivalent à  $i = i+1$ )



# Équivalence **while** et **for** en C

**for**

```
1  for (expression1; expression2;  
2      expression3) {  
3      instructions;  
    }
```

equivaut à :

**while**

```
1  expression1;  
2  while (expression2) {  
3      instructions;  
4      expression3;  
5  }
```

# Boucle **for**

Forme synthétique de la boucle en C :

```
1  int i;  
2  for (i = 1; i <= n; i++) {  
3      s = s + i;  
4  }
```

Les 3 éléments qui font tourner la boucle :

initialisation :  $i = 1$

condition :  $(i \leq n)$

incrément :  $i++$  (équivalent à  $i = i+1$ )

Analogue à **for** i in **range** en Python :

```
1  for i in range(1, n+1):  
2      s = s + i
```

Rq :  $i \leq n$  équivaut à  $i < n+1$

# Équivalence **while** et **for** en C

**for**

```
1  for (expression1; expression2; expression3) {  
2      instructions;  
3  }
```

equivaut à :

**while**

```
1  expression1;  
2  while (expression2) {  
3      instructions;  
4      expression3;  
5  }
```

C'est tout pour aujourd'hui !