



Le Langage C

Cours de **Nils Beaussé et Bilel BENZIANE**
nils.beausse@isen-ouest.yncrea.fr
bilel.benziane@isen-ouest.yncrea.fr

D'après les notes de cours et le
cours de Benoit Lardeux, Jean-
Benoît Pierrot, Pierre-Jean
Bouvet et Leandro Montero

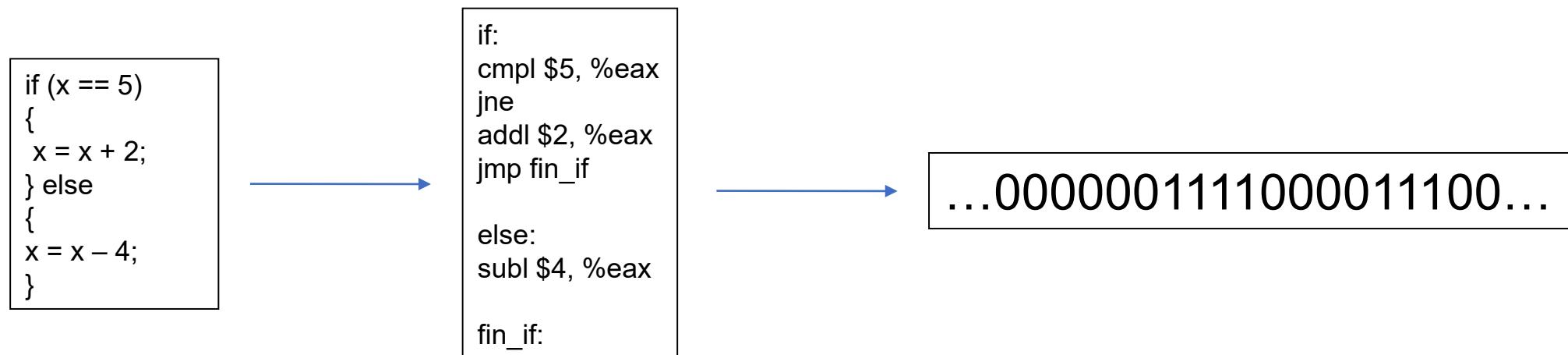
Introduction

Historique du langage C

- Apparu au cours de l'année 1972 dans les Bell Labs
 - Développé par D. Ritchie et K. Thompson
 - Inspiré des langages BCPL et B
 - Lié à la conception du système d'exploitation UNIX
 - UNIX est écrit à 95% en Langage C
- Normalisé en 1989 par l'ANSI
 - ANSI C89
- Approuvé par l'ISO en 1990
 - ISO C
- Évolution en 1999 puis itérativement (Dernière version en 2018).
 - ISO C99 -> ISO C18 (2018)

Langage compilé ?

- Le C est un langage **compilé** de **bas niveau** : **Qu'est-ce que ça veut dire ?**
→ Une instruction est traduite directement en code machine par le compilateur (gcc/clang etc.)



Chaque instruction assembleur à un sens précis et simple

Langage compilé ?

- Par exemple :

```
if:  
cmpl $5, %eax  
jne  
addl $2, %eax  
jmp fin_if  
  
else:  
subl $4, %eax  
  
fin_if:
```

CMPL



...0000001111000011100...



Un code binaire qui est une
succession d'instruction
élémentaire



Bits	Value
0-5	31
6-8	BF
9	/
10	L
11-15	RA
16-20	RB
21-30	32
31	/

Langage compilé ?

- Par exemple :

Chacune est comprise par un sous-circuit du processeur et effectue une opération directement au niveau électronique

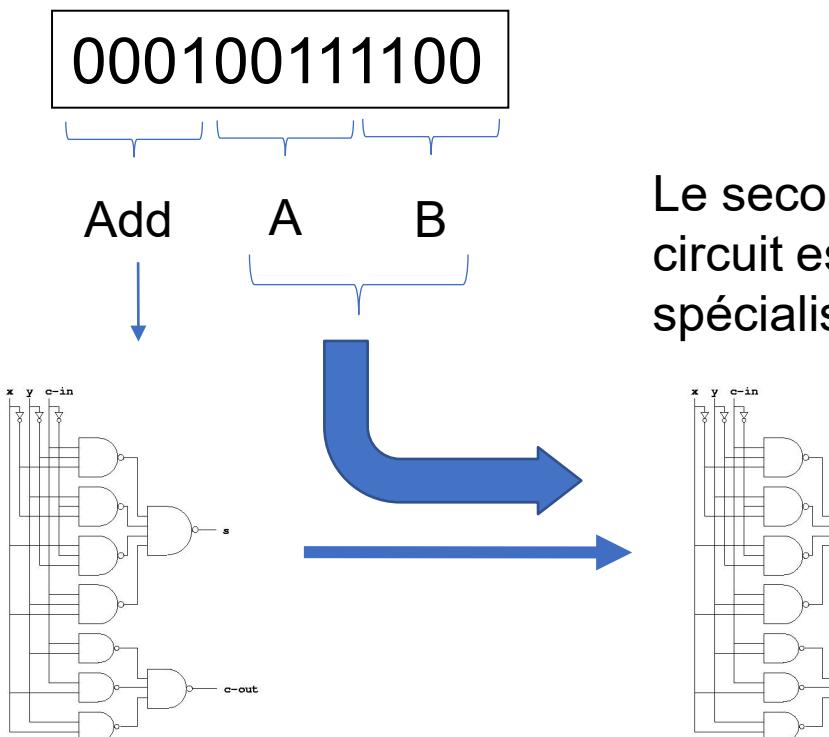


Bits	Value
0-5	31
6-8	BF
9	/
10	L
11-15	RA
16-20	RB
21-30	32
31	/

Langage compilé ?

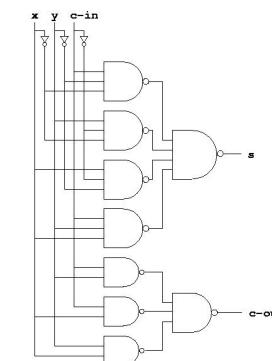
- Grossièrement :

Un circuit comprend la première instruction et redirige la suite à un autre circuit (ici un circuit d'addition par exemple)



Le second circuit est spécialisé

Résultat



En conclusion

- La programmation dans un **langage compilé** permet d'interagir directement avec l'électronique du processeur (CPU).
 - Ce point permet aux langages compilés d'être très « proche » du matériel :
 - On parle de langage « **bas niveau** ».
 - On parle aussi de programme « **natif** » pour les programmes qui en résultent.
 - De fait : Programmes extrêmement rapides par rapport à une autre approches.
 - En opposition aux langages dit « interprétés » : Java / Python etc.
 - On a créé le C à l'origine pour éviter d'écrire directement en langage machine, **ce qui est très dur pour un humain.**

Résumé :

Résumé :

- C <=> Basé sur un **standard ouvert**
 - Compilateurs et bibliothèques disponibles sur la plupart des architectures

Résumé :

- C <=> Basé sur un **standard ouvert**
 - Compilateurs et bibliothèques disponibles sur la plupart des architectures
- Langage destiné au départ à **simplifier les opérations de programmation et de compilation** (à la place d'un programme en binaire ou en assembleur)

Résumé :

- C <=> Basé sur un **standard ouvert**
 - Compilateurs et bibliothèques disponibles sur la plupart des architectures
- Langage destiné au départ à **simplifier les opérations de programmation et de compilation** (à la place d'un programme en binaire ou en assembleur)
 - Des opérations très proches des éléments matériels

Résumé :

- C <=> Basé sur un **standard ouvert**
 - Compilateurs et bibliothèques disponibles sur la plupart des architectures
- Langage destiné au départ à **simplifier les opérations de programmation et de compilation** (à la place d'un programme en binaire ou en assembleur)
 - Des opérations très proches des éléments matériels
 - Chaque instruction du langage est conçue pour être compilée en un nombre d'instructions machine assez prévisible en termes d'occupation mémoire et de charge de calcul.

Résumé :

- C <=> Basé sur un **standard ouvert**
 - Compilateurs et bibliothèques disponibles sur la plupart des architectures
- Langage destiné au départ à **simplifier les opérations de programmation et de compilation** (à la place d'un programme en binaire ou en assembleur)
 - Des opérations très proches des éléments matériels
 - Chaque instruction du langage est conçue pour être compilée en un nombre d'instructions machine assez prévisible en termes d'occupation mémoire et de charge de calcul.
- Son usage s'est largement répandu
 - Unix est écrit à 95% en langage C
 - A influencé de nombreux langages : C++, java, PHP, Python
 - L'un des langages les plus utilisés aujourd'hui et le langage compilé le plus utilisé (jeux vidéo, IA etc.) avec son évolution (le C++).

Intérêts pour vous

- Met en œuvre un **nombre restreint de concepts**
 - Langage facile à apprendre
- Langage **efficace**
 - Rapidité d'exécution,
 - Faible encombrement des exécutables
- Il permet l'écriture de nombreux logiciels tels que
 - Les noyaux de **système d'exploitation**
 - **Systèmes embarqués**
 - **Tout logiciel où l'on désire de la performance (Jeux, Bibliothèque IA, etc.)**

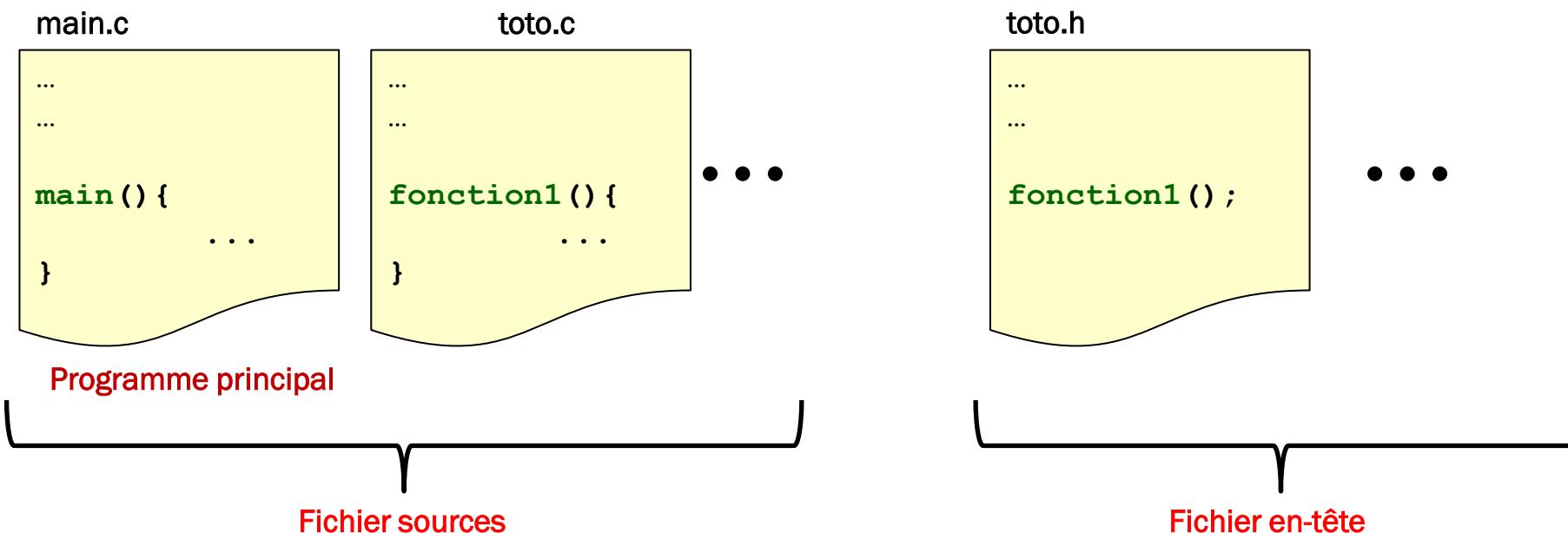
Inconvénients

- **Peu de vérifications** lors de la **compilation** et **aucune vérification** pendant **l'exécution** (la réussite de la compilation n'assure pas que le programme fonctionne)
 - Exemple : Une allocation mémoire qui passe une compilation mais qui est trop petite pour le nombre de données allouées
- Pas de support direct aux **concept**s informatiques modernes
 - Programmation orientée objet
 - Programmes **portables**. Le comportement exact des exécutables dépend de l'ordinateur et du système cible. Il faut souvent **recompiler pour obtenir des versions compatibles avec tous les systèmes et matériels**.

Généralités

Organisation des fichiers de votre programmation

- Il existe 2 types de fichiers en C, ce sont de simples fichiers textes :
 - Les fichiers sources (*.c)
 - Contient les programmes et sous-programmes utilisés par votre algorithme
 - Contient la description des étapes successives à réaliser
 - Les fichiers en-tête ou header(*.h)
 - Rappel comment sont définis vos programmes (comment sont définis les entrées et les sorties)



Programme principal

- Il s'agit du seul fichier qui contient la fonction main
 - C'est le point de démarrage de l'exécutable

test.c

```
int main()
{
    // reste du programme //
    return 0;
}
```

Programme principal

- Il s'agit du seul fichier qui contient la fonction main
 - C'est le point de démarrage de l'exécutable

test.c

```
int main() ← Le type de la fonction suivi de son nom et de parenthèses*
{
    // reste du programme //
    return 0;
}
```

Programme principal

- Il s'agit du seul fichier qui contient la fonction main
 - C'est le point de démarrage de l'exécutable

test.c

```
int main()
{
    // reste du programme //
    return 0;
}
```

Le type de la fonction suivi de son nom et de parenthèses*

L'accolade ouvrante qui marque le début de la fonction, et donc ici le début du programme

Programme principal

- Il s'agit du seul fichier qui contient la fonction main
 - C'est le point de démarrage de l'exécutable

test.c

```
int main()
{
    // reste du programme //
    return 0;
}
```

Le type de la fonction suivi de son nom et de parenthèses*

L'accolade ouvrante qui marque le début de la fonction, et donc ici le début du programme

La fin de la fonction

Programme principal

- Il s'agit du seul fichier qui contient la fonction main
 - C'est le point de démarrage de l'exécutable

test.c

```
int main()
{
    // reste du programme //
    return 0;
}
```

Le type de la fonction suivi de son nom et de parenthèses*

L'accolade ouvrante qui marque le début de la fonction, et donc ici le début du programme

La fin de la fonction

accolade fermante

Programme principal

- Il s'agit du seul fichier qui contient la fonction main
 - C'est le point de démarrage de l'exécutable

test.c

```
int main()
{
    // reste du programme //
    return 0;
}
```

Le type de la fonction suivi de son nom et de parenthèses*

L'accolade ouvrante qui marque le début de la fonction, et donc ici le début du programme

La fin de la fonction

accolade fermante

* Pour le moment nous ne savons pas forcément ce qu'est un type ou ce que signifie exactement le return 0, **le point important à retenir pour le moment est que le programme existe dans cette fonction main qui commence par son nom, une accolade ouvrante, puis le contenu, et enfin un return 0 et une accolade fermante.**

Compilation simple

- Le but : produire à partir des fichiers textes contenant votre code C le code en langage machine qui pourra être exécuté par votre ordinateur.
- Il y a une étape de traduction, c'est un autre programme qui se charge de traduire, il s'agit du compilateur.
- On parle donc de compilation.

Compilation simple

- Comment produire un exécutable à partir des fichiers textes (compilation) ?

```
>> ls ← Liste les fichiers présents dans le répertoire  
test_pi.c  
>> gcc test_pi.c ← Compile le fichier test_pi.c  
>> ls  
a.out test_pi.c ← a.out est l'exécutable (nom par défaut car on n'en a pas précisé à gcc)
```

- Comment lancer l'exécutable ?

```
>> a.out : command not found !  
>> ./a.out ← Execute a.out  
PI vaut 3.14
```

Résultat du programme

- Il faut s'assurer que l'exécutable possède bien les droits d'exécution (voir cours de Linux) !

Compilation simple

- Nommer un exécutable
 - Si on compile plusieurs programmes, le a.out sera celui du dernier programme compilé,
 - Le nom a.out n'est pas forcément explicite...
 - **L'option -o** de gcc permet de **nommer l'exécutable**. Le o signifiant output (sortie).

```
>> ls ← Liste les fichiers présents dans le répertoire  
test_pi.c  
>> gcc test_pi.c -o test_pi.x ← Compile le fichier test1.c  
>> ls  
test_pi.c      test_pi.x
```

Compilation simple

Note sur l'extension :

```
>> ls  
test_pi.c  
>> gcc test_pi.c  
>> ls  
a.out test_pi.c
```

Remarquez que le nom de l'exécutable est « **a.out** », d'extension « **.out** ».

Ceci pourrait vous paraître bizarre car sous Windows les exécutables sont tous d'extension « **.exe** »

En fait, sous Linux, **il n'y a pas d'extension type**, tous les fichiers peuvent être des exécutables, pour peu qu'il y ait les droits dessus. Souvent, on crée des exécutables **sans extensions**, tout simplement, mais rien ne vous interdit de mettre .exe, .out ou tout ce que vous voudrez.

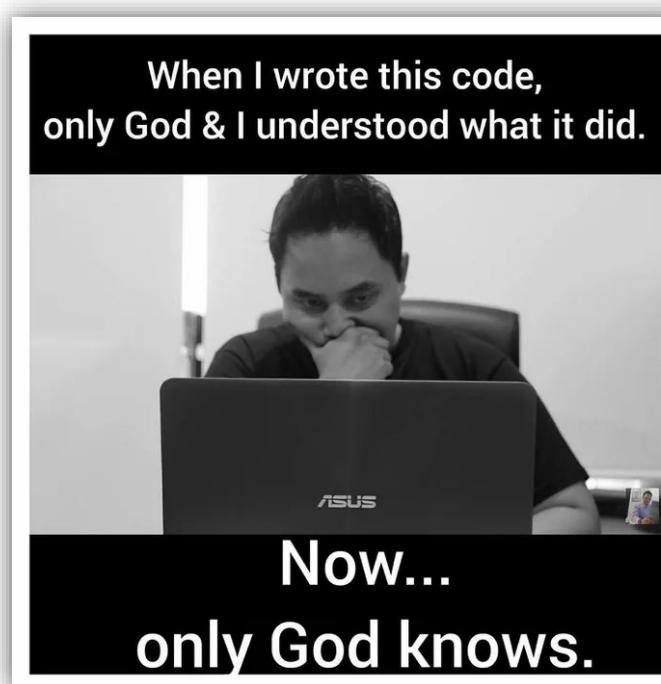
Syntaxe de base

Les commentaires

- Les commentaires débutent par /* et se terminent par */

```
/* This is a comment */
```

- **Leur but est de transmettrez une information utile à vos collègues et, surtout, à votre vous du futur !**



Les commentaires

- On peut commenter :
 - Le fichier lui-même.
 - Des fonctions
 - Des groupes d'instructions dans une fonction (quand c'est pertinent)
 - Des déclaration et instruction unique (quand c'est pertinent)
- Dans des projets internationaux la langue anglaise est conseillée, néanmoins, dans un projet strictement interne à un groupe fermé, il peut être intéressant de privilégier sa langue natale pour aider à la compréhension.
 - **L'objectif n°1 reste de transmettre efficacement des informations.**

Les commentaires (2)

- Premier niveau :
 - Raison d'être du fichier, nom de l'auteur, copyright, date de création, ...

```
/*! *****
File: test_algo.c

Author: Nilsou & Bilel

Version: v1.2 18/12/21 (NB)      : correct bugs
          v1.1 18/12/21 (BB)      : update parameters
          v1.0 16/12/22 (NB)      : initial version

Brief: test bench for the algorithm

Compilator : gcc version 8.3.0

Copyright(c) 2023, Nilsou et Bilel ISEN NANTES
Licence CeCILL compatible GNU-GPL.
******/
```

Les commentaires (3)

- Au niveau des fonctions
 - Les paramètres et la raison d'être de la procédure

```
/*! -----  
Function: carre  
  
Brief: Perform square value of a real number  
  
Interface:  
    Input  
        x : input value (float)  
    Output  
        NA  
    Input/Output  
        NA  
  
Return value : square value of x (float)  
-----*/
```

Les commentaires (4)

- Au niveau des déclarations et expressions
 - Expliquer la déclaration d'une variable (à quoi sert-elle)
 - Commenter une instruction spécifique
 - Exprimer ce que réalise une fraction significative d'une procédure

```
float x,y,z; // input value
float x2,y2,z2; // square values
float var; // variance

/*
-----*
Compute the variance by performing
the average of square values
-----*/
x2 = x*x; // compute square of x
y2 = y*y; // compute square of y
z2 = z*z; // compute square of z

var = ( x2 + y2 +z2 )/3; // compute average of square values
```

Les types de données

- Le C est un langage typé : en effet, la mémoire est constituée de 0 et de 1 comme nous n'avons vu.
 - Mais il existe pas mal de manière de coder un nombre en binaire !
Je peux décider que « 2 » s'écrit :

0010

Mais je pourrais tout à fait décider que c'est dans l'autre sens :

0100

Les types de données

- Le problème existe aussi pour la taille des nombres :
 - Je peux avoir envie de stocker :

2,0

sans plus de précision ...

- Ou alors :

2,000000000000

Pour avoir une bonne précision !

On comprend bien, sans rentrer dans les détails, que le second va prendre plus de place en mémoire que le premier !

Les types de données

- Même problème pour tout ce qui n'est PAS un simple chiffre. Le codage des caractères, par exemple, est purement arbitraire, je peux décider que

$a \Leftrightarrow 001001$ ou que $a \Leftrightarrow 000101$

si j'ai envie !

Les types de données

- Comment savoir quel codage est employé, quelle taille je veux utiliser, et comment ne pas les confondre entre eux ?
- Les **types** nous permettent de dire au programme comment coder ce que l'on souhaite, et ils permettent de faire des vérifications d'erreurs **pour ne pas mélanger les choux et les carottes !**



Quelques types de données utiles :

- Le type entier : **int** -> contient un nombre entier.
 - Attributs de précision : **short** ou **long**
 - Attribut de représentation : **unsigned** (négatif autorisé ou non)

Quelques types de données utiles :

- Le type entier : **int** -> contient un nombre entier.
 - Attributs de précision : **short** ou **long**
 - Attribut de représentation : **unsigned** (négatif autorisé ou non)
- Le type **flottant** (nombre à virgule)
 - Il en existe 3 types correspondant à 3 précisions possibles
 - **float**
 - **double**
 - **long double**

Quelques types de données utiles :

- Le type entier : **int** -> contient un nombre entier.
 - Attributs de précision : **short** ou **long**
 - Attribut de représentation : **unsigned** (négatif autorisé ou non)
- Le type **flottant** (nombre à virgule)
 - Il en existe 3 types correspondant à 3 précisions possibles
 - **float**
 - **double**
 - **long double**
- Le type caractère : **char**
 - Doit pouvoir contenir le code de n'importe quel **caractère**
 - Le code **ASCII** est plus majoritairement utilisé

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL]

Quelques types de données utiles :

Type de donnée	Signification	Taille (octets)	Plage
char	Caractère	1	-128 à 127 (avec ce type, 255 caractères différent dispo)
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32768 à 32767
unsigned short int	Entier court non signé	2	0 à 65535
int	Entier	2/4	cf short int et long int
unsigned int	Entier non signé	2/4	cf unsigned short et long int
long int	Entier long	4	-2147483648 à 2147486647
unsigned long int	Entier long non signé	4	0 à 4294967295
float	Flottant (réel)	4	$\pm 3.4 \times 10^{-38}$ à 3.4×10^{38}
double	Flottant double	8	$\pm 1.7 \times 10^{-308}$ à 1.7×10^{308}
long double	Flottant double long	10	$\pm 3.4 \times 10^{-4932}$ à 3.4×10^{4932}

Déclaration d'une variable

Pseudo-langage

variable

a,b,c : Entier

c : Caractère

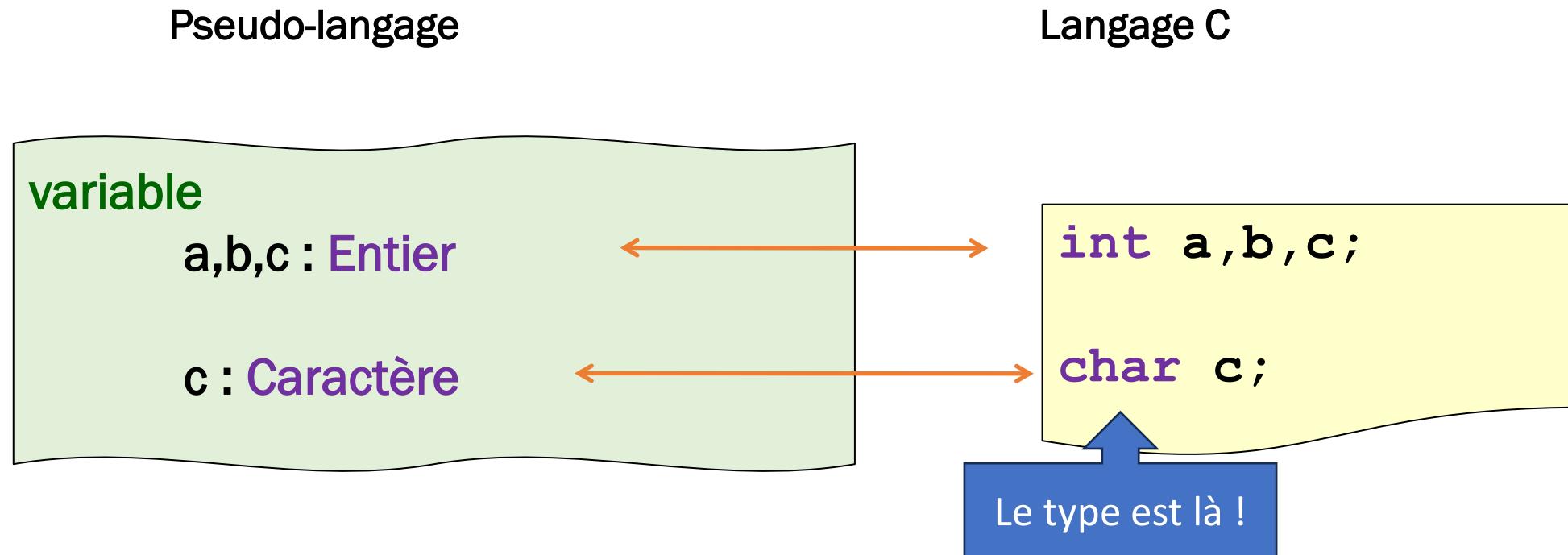
Langage C

```
int a,b,c;
```

```
char c;
```

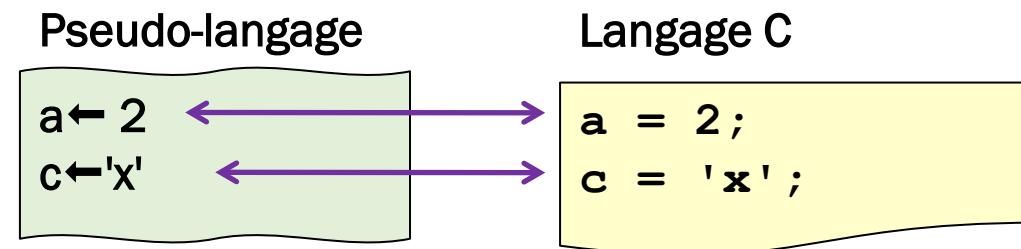


Déclaration d'une variable

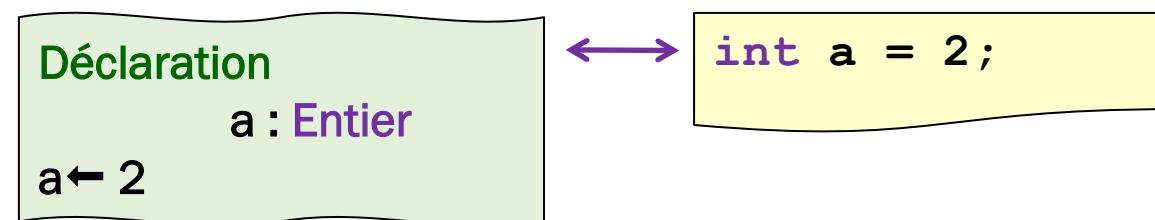


Affectation d'une variable

- L'affectation (changement de valeur d'une variable) se fait avec l'opérateur « = »

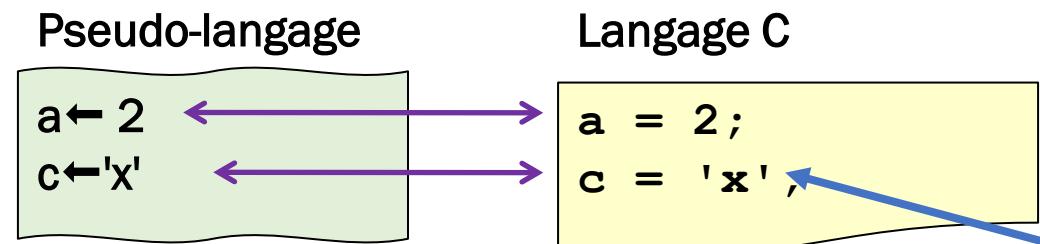


- L'affectation peut être englobée dans la déclaration :

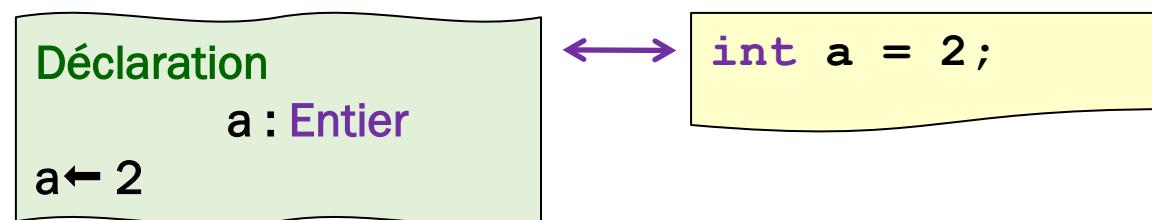


Affectation d'une variable

- L'affectation (changement de valeur d'une variable) se fait avec l'opérateur « = »



- L'affectation peut être englobée dans la déclaration :



Attention à bien penser aux guillemets simples autour du caractère !

Affectation d'une variable

- Grace à vos connaissances acquises récemment sur le fonctionnement des ordinateurs, **devinez ce que contient a si je ne fais que déclarer la variable sans l'affecter ?**

```
int a ;
```

Affectation d'une variable

- Grace à vos connaissances acquises récemment sur le fonctionnement des ordinateurs, **devinez ce que contient a si je ne fais que déclarer la variable sans l'affecter ?**

```
int a ;
```

- Réponse : **Le contenu est aléatoire** ! En effet, **on réserve** un espace mémoire de votre RAM **mais on ne le modifie pas**. En faisant cela il reste dans cette mémoire ce qu'il y avait auparavant (reste d'anciens programmes etc.)

Opérateurs usuels

- Addition

```
c = a + b;
```

1 : Evaluation de a et b

2 : Addition de a et b

3 : Affectation du résultat dans c

- Soustraction

```
c = a - b;
```

- Multiplication

```
c = a * b;
```

- Division

```
c = a / b;
```

- Modulo

```
c = a % b;
```

Opérateurs usuels

- Addition

 $c = a + b;$

1 : Evaluation de a et b

2 : Addition de a et b

3 : Affectation du résultat dans c

- Soustraction

 $c = a - b;$

- Multiplication

 $c = a * b;$

- Division

 $c = a / b;$

L'opérateur '/' désigne à la fois la division **entière (euclidienne)** et la division **flottante (division à virgule)**. Si a et b sont entiers alors division entière sinon division flottante

- Modulo

 $c = a \% b;$ 

Opérateurs de comparaison

- Strictement supérieur

```
res = ( a > b );
```

- Supérieur ou égal

```
res = ( a >= b );
```

- Strictement inférieur

```
res = ( a < b );
```

- Inférieur ou égal

```
res = ( a <= b );
```

- Égal

```
res = ( a == b );
```

- Différent

```
res = ( a != b );
```

Opérateurs de comparaison

- Strictement supérieur

```
res = ( a > b );
```

- Supérieur ou égal

```
res = ( a >= b );
```

- Strictement inférieur

```
res = ( a < b );
```

- Inférieur ou égal

```
res = ( a <= b );
```

- Égal

```
res = ( a == b );
```

- Différent

```
res = ( a != b );
```

1 : Evaluation de a et b

2 : Comparaison de a et b, **1** si vrai , **0** sinon

3 : Affectation du résultat dans res

Opérateurs de comparaison

- Strictement supérieur
- Supérieur ou égal
- Strictement inférieur
- Inférieur ou égal
- Égal
- Différent

```
res = ( a > b );
```

```
res = ( a >= b );
```

```
res = ( a < b );
```

```
res = ( a <= b );
```

```
res = ( a == b );
```

```
res = ( a != b );
```

1 : Evaluation de a et b

2 : Comparaison de a et b, 1 si vrai , 0 sinon

3 : Affectation du résultat dans res

Ne pas confondre l'opérateur de comparaison '==' et l'opérateur d'affectation '='



Opérateurs booléens

Syntaxe	Description
&&	Et logique
 	OU logique
!	Négation
==	Égalité (1 si égal, 0 sinon)

E = ((a < b) && (c < d)) ;

F = ((a < b) || (c < d)) ;

G = !(a < b) ;

H = ((a < b)==(c < d)) ;

Opérateurs booléens

Syntaxe	Description
&&	Et logique
 	OU logique
!	Négation
==	Égalité (1 si égal, 0 sinon)

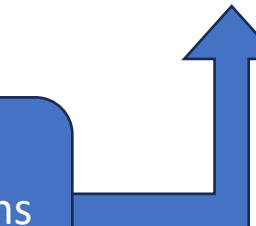
```
E =(( a < b) &&(c < d)) ;
```

```
F =(( a < b) ||(c < d)) ;
```

```
G =(! ( a < b)) ;
```

```
H = (( a < b)==(c < d)) ;
```

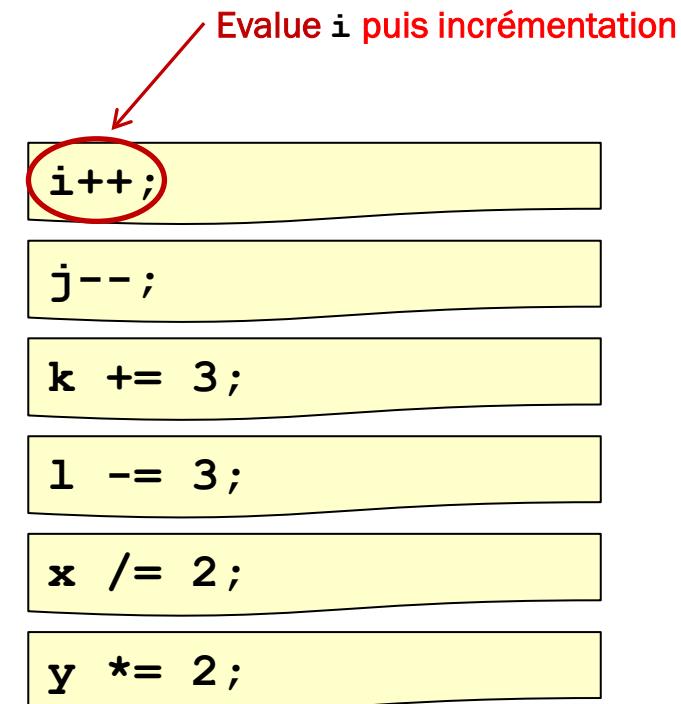
Êtes-vous capable d'expliquer ces combinaisons d'opérateurs ? Qu'y a-t-il dans les entiers E, F, G, H après ces lignes ?



Opérateurs combinant plusieurs opérations

Syntaxe	Description
<code>++</code>	Incrémantation de 1
<code>--</code>	Décrémentation de 1
<code>+=</code>	Addition puis affectation
<code>-=</code>	Soustraction puis affectation
<code>/=</code>	Division puis affectation
<code>*=</code>	Multiplication puis affectation

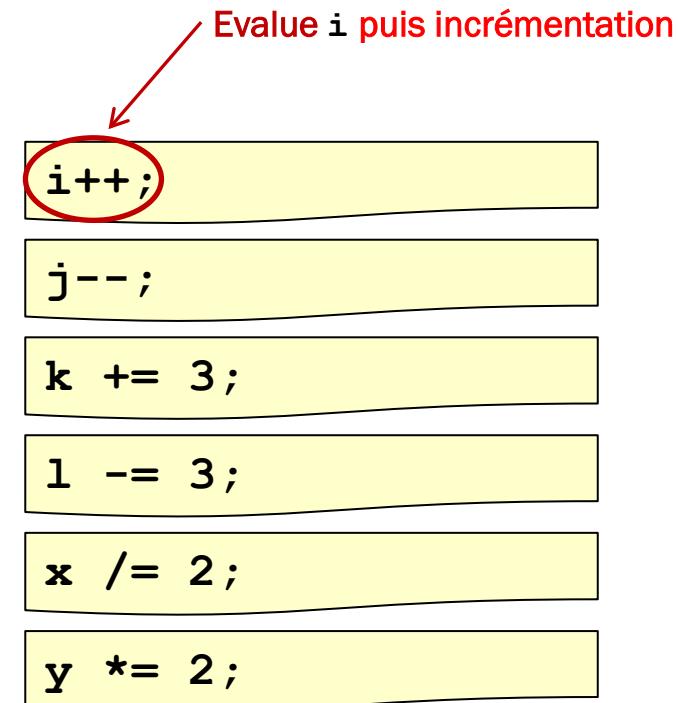
Value i puis incrémentation



```
i++;
j--;
k += 3;
l -= 3;
x /= 2;
y *= 2;
```

Opérateurs combinant plusieurs opérations

Syntaxe	Description
<code>++</code>	Incrémantation de 1
<code>--</code>	Décrémentation de 1
<code>+=</code>	Addition puis affectation
<code>-=</code>	Soustraction puis affectation
<code>/=</code>	Division puis affectation
<code>*=</code>	Multiplication puis affectation



```
i++;  
j--;  
k += 3;  
l -= 3;  
x /= 2;  
y *= 2;
```

Êtes-vous capable d'expliquer ce qu'y a-t-il dans les entiers i, j, k, l, x et y après ces lignes ?
(On suppose qu'ils sont égaux à 0 au départ)

Opérateurs sur les bits

C'est pour information : vous aurez en effet peu à employer ceci pour le moment.

Syntaxe	Description
&	ET
	OU
^	OU EXCLUSIF
~	NON
<<	Décalage à gauche des bits
>>	Décalage à droite des bits

`c = a & b;`

`c = a | b;`

`c = a ^ b;`

`b = ~a;`

`b = a << 5;`

`b = a >> 5;`

Illustration
de
l'enseignant
au tableau !

Ecriture formatée (la fonction afficher du C)

```
#include <stdio.h>
int main(){
    printf("Bonjour \n");
    return 0;
}
```

Caractère spécial	Signification
\n	Saut à la ligne
\t	tabulation
\\"	Antislash
%%	Pourcentage



Ecriture formatée (la fonction afficher du C)

```
#include <stdio.h>
int main(){
    printf("Bonjour \n");
    return 0;
}
```

```
#include <stdio.h>
int main(){
    printf("x=%d",2);
    return 0;
}
```

Caractère spécial	Signification
\n	Saut à la ligne
\t	tabulation
\\"	Antislash
%%	Pourcentage

Caractère spécial	Format
%d	Décimal
%f	Flottant
%e	Scientifique
%c	Caractère
%x	Hexadécimal
%s	Chaine de caractères
%g	Meilleur de %e et %f



Ecriture formatée (la fonction afficher du C)

```
#include <stdio.h>
int main(){
    printf("Bonjour \n");
    return 0;
}
```

```
#include <stdio.h>
int main(){
    printf("x=%d",2);
    return 0;
}
```

Caractère spécial	Signification
\n	Saut à la ligne
\t	tabulation
\\\	Antislash
%%	Pourcentage

Attention, un *même contenu* de variable a *plusieurs représentations différentes*

Caractère spécial	Format
%d	Décimal
%f	Flottant
%e	Scientifique
%c	Caractère
%x	Hexadécimal
%s	Chaine de caractères
%g	Meilleur de %e et %f



Lecture formatée (simple)

```
#include <stdio.h>
int main(){
    int x;

    printf("Donnez votre taille :");
    scanf("%d", &x);
    printf("Vous mesurez %d cms\n", x);
    return 0;
}
```



Lecture formatée (simple)

```
#include <stdio.h>
int main(){
    int x;

    printf("Donnez votre taille :");
    scanf("%d", &x);
    printf("Vous mesurez %d cms\n", x);
    return 0;
}
```



Attention, il est préférable de ne lire **qu'une seule variable** à la fois

Lecture formatée (simple)

```
#include <stdio.h>
int main(){
    int x;

    printf("Donnez votre taille :");
    scanf("%d", &x);
    printf("Vous mesurez %d cms\n", x);
    return 0;
}
```



Attention, il est préférable de ne lire **qu'une seule variable** à la fois

Attention à **ne pas oublier le &** sinon le programme **plantera quand vous le lancerez** (nous verrons plus tard ce que signifie ce symbole, pour le moment nous l'appliquerons « bêtement » dans le scanf)

Lecture formatée (simple)

Cette ligne est nécessaire pour que printf et scanf fonctionnent, on vous explique plus tard pourquoi

```
#include <stdio.h>
int main(){
    int x;

    printf("Donnez votre taille :");
    scanf("%d", &x);
    printf("Vous mesurez %d cms\n", x);
    return 0;
}
```



Attention, il est préférable de ne lire **qu'une seule variable** à la fois

Attention à **ne pas oublier le &** sinon le programme **plantera quand vous le lancerez** (nous verrons plus tard ce que signifie ce symbole, pour le moment nous l'appliquerons « bêtement » dans le scanf)

Exemple de nommage....

- Pour les variables

Renseigne sur le type

```
int maVariable_i
float maVariable_f
double maVariable_d
char maVariable_c
```

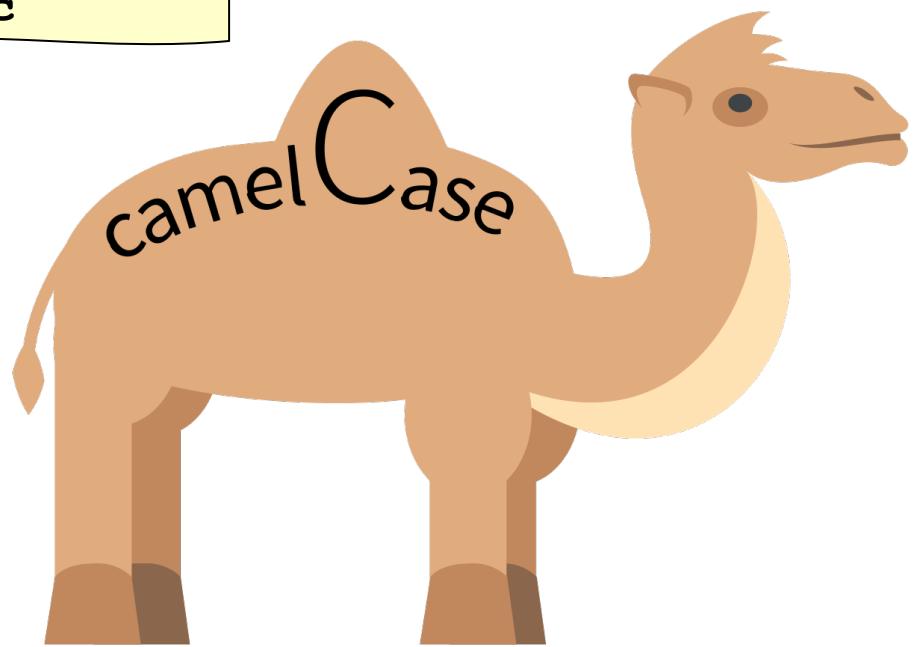
Exemple de nommage....

- Pour les variables

Renseigne sur le type

```
int maVariable_i
float maVariable_f
double maVariable_d
char maVariable_c
```

Première lettre du second mot en majuscule :
On parle de « camelCase » quand on fait ceci.



Exemple de nommage....

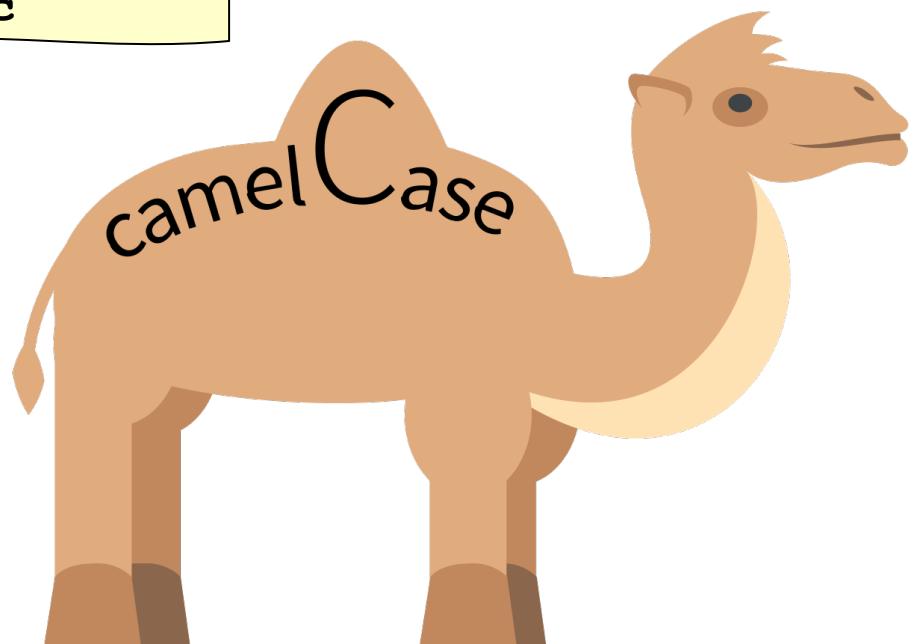
- Pour les variables

```
int maVariable_i  
float maVariable_f  
double maVariable_d  
char maVariable_c
```

Renseigne sur le type

Première lettre du second mot en majuscule :
On parle de « camelCase » quand on fait ceci.

Il existe évidemment de nombreux autres types de convention de nommage, le tout est d'être cohérent dans un même programme ou projet.



Mini TP d'introduction

Exo 1

Écrire un programme en langage C qui fait saisir à l'utilisateur 2 entiers et qui calcule puis affiche la somme des 2 valeurs entrées, puis, incrémenté de 2 le résultat (avec un seul opérateur) avant de l'afficher de nouveau.

Exo 2

Écrire un programme en langage C qui fait saisir à l'utilisateur 2 entiers et qui affiche si l'entier 1 est supérieur ou égal à l'entier 2 (interdiction d'utiliser des « if », on utilisera que ce que nous avons vu jusqu'ici)

Correction

Ecrire un programme en langage C qui fait saisir à l'utilisateur 2 entiers et qui calcule puis affiche la somme des 2 valeurs entrées puis incrémenté de 2 le résultat (avec un seul opérateur) avant de l'afficher de nouveau.

```
#include <stdio.h>

int main(){
    int a,b,s;
    printf("Entrez 1ere valeur :");
    scanf("%d",&a);
    printf("Entrez 2eme valeur :");
    scanf("%d",&b);
    s = a+b;
    printf("la somme vaut %d",s);
    s += 2;
    printf("la somme incrémentée vaut %d",s);
    return 0;
}
```

Correction

Ecrire un programme en langage C qui fait saisir à l'utilisateur 2 entiers et qui affiche si l'entier 1 est supérieur ou égal à l'entier 2 (interdiction d'utiliser des « if », on utilisera que ce que nous avons vu jusqu'ici)

```
#include <stdio.h>

int main(){
    int a,b,s;
    printf("Entrez 1ere valeur :");
    scanf("%d",&a);
    printf("Entrez 2eme valeur :");
    scanf("%d",&b);
    s = (a>=b);
    printf("L'entier 1 est-il supérieur ou égal au 2 ? 0 si faux, 1 si vrai : %d",s);
    return 0;
}
```

Exercice 2

Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 8;
    int b = 14;
    int c,d;
    a++;
    b-=2;
    c = a|b;
    d = (a<<2) & (b>>1);
    printf("c=%x et d=%d",c,d);

    return 0;
}
```

Exercice 2

Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 8;
    int b = 14;
    int c,d;      ←
    a++;
    b-=2;
    c = a|b;
    d = (a<<2) & (b>>1);
    printf("c=%x et d=%d",c,d);

    return 0;
}
```

a 08 : 0000 1000

b 14 : 0000 1110

c X : xxxx xxxx

d X : xxxx xxxx

Exercice 2

Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 8;
    int b = 14;
    int c,d;
    a++;
    b-=2;          ←
    c = a|b;
    d = (a<<2) & (b>>1);
    printf("c=%x et d=%d",c,d);

    return 0;
}
```

- a 08 : 0000 1000 09 : 0000 1001
- b 14 : 0000 1110 12 : 0000 1100
- c X : xxxx xxxx
- d X : xxxx xxxx

Exercice 2

Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 8;
    int b = 14;
    int c,d;
    a++;
    b-=2;
    c = a|b;      ←
    d = (a<<2) & (b>>1);
    printf("c=%x et d=%d",c,d);

    return 0;
}
```

- | | | |
|---|----------------|----------------|
| a | 08 : 0000 1000 | 09 : 0000 1001 |
| b | 14 : 0000 1110 | 12 : 0000 1100 |
| c | X : xxxx xxxx | 13 : 0000 1101 |
| d | X : xxxx xxxx | |

Exercice 2

Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 8;
    int b = 14;
    int c,d;
    a++;
    b-=2;
    c = a|b;
    d = (a<<2) & (b>>1); ←
    printf("c=%x et d=%d",c,d);

    return 0;
}
```

- a 08 : 0000 1000 09 : 0000 1001 36 : 0010 0100
- b 14 : 0000 1110 12 : 0000 1100 06 : 0000 0110
- c X : xxxx xxxx 13 : 0000 1101
- d X : xxxx xxxx

Exercice 2

Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 8;
    int b = 14;
    int c,d;
    a++;
    b-=2;
    c = a|b;
    d = (a<<2) & (b>>1); ←
    printf("c=%x et d=%d",c,d);

    return 0;
}
```

- | | | | |
|---|----------------|----------------|----------------|
| a | 08 : 0000 1000 | 09 : 0000 1001 | 36 : 0010 0100 |
| b | 14 : 0000 1110 | 12 : 0000 1100 | 06 : 0000 0110 |
| c | X : xxxx xxxx | 13 : 0000 1101 | |
| d | X : xxxx xxxx | | 04 : 0000 0100 |

Exercice 2

Qu'affiche le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int a = 8;
    int b = 14;
    int c,d;
    a++;
    b-=2;
    c = a|b;
    d = (a<<2) & (b>>1); ←
    printf("c=%x et d=%d",c,d);
    return 0;
}
```

a 08 : 0000 1000

09 : 0000 1001

36 : 0010 0100

b 14 : 0000 1110

12 : 0000 1100

06 : 0000 0110

c X : xxxx xxxx

13 : 0000 1101

04 : 0000 0100

d X : xxxx xxxx

Affichage:

c=d et d=4

(c = 13, affiché en mode %x
donc en hexadecimal soit 0xD)

Structures de contrôle

Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors  
    ...  
sinon  
    ...  
finsi
```



Langage C

```
if( a == b ) {  
    ...  
} else {  
    ...  
}
```

Si ... Sinon ...

- Le « **if** » teste si l'intérieur vaut 1 (vrai) ou 0 (faux).
 - Ainsi, comme `(a == b)` renvoie 1 si c'est égal et 0 si c'est faux on peut l'utiliser directement !
 - Mais on peut très bien écrire

```
a = 1;  
if(a) {  
    /* ce code s'executera car 1 <=> vrai */  
}
```

- Ou :

```
a = (b>c);  
if(a) {  
    /* ce code s'executera car la valeur du test b>c (1  
     <=> vrai ou 0 <=> faux), est stocké dans a/*  
}
```

Si ... Sinon ...

- Le « **if** » teste si l'intérieur vaut 1 (vrai) ou 0 (faux).
 - En pratique on écrit bien souvent les tests directement dedans, par simplicité :

```
if(a == 23){  
}
```

```
if((a && b) == 1){  
}
```

```
if(a <= b){  
}
```

```
if(a > b){  
}
```

etc...

Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors  
    ...  
sinon  
    ...  
finsi
```



Langage C

```
if( a == b ) {  
    ...  
} else {  
    ...  
}
```

Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors  
    ...  
sinon  
    ...  
finsi
```



Langage C

```
if( a == b){  
    ...  
}else{  
    ...  
}
```

Exemple de code

```
int i=2;  
if( i == 2){  
    printf("Branche 1");  
}else{  
    printf("Branche 2");  
}  
printf("Fin");
```

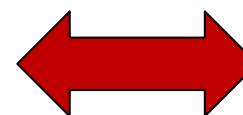
Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors  
    ...  
sinon  
    ...  
finsi
```

Langage C

```
if( a == b){  
    ...  
}else{  
    ...  
}
```



Exemple de code

```
int i=2;  
if( i == 2){  
    printf("Branche 1");  
}else{  
    printf("Branche 2");  
}  
printf("Fin");
```

Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors  
    ...  
sinon  
    ...  
finsi
```



Langage C

```
if( a == b){  
    ...  
}else{  
    ...  
}
```

Exemple de code

```
int i=2;  
if( i == 2){  
    printf("Branche 1");  
}else{  
    printf("Branche 2");  
}  
printf("Fin");
```

Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors
    ...
sinon
    ...
finsi
```



Langage C

```
if( a == b) {
    ...
}else{
    ...
}
```

Exemple de code

```
int i=2;
if( i == 2){
    printf("Branche 1");
}else{
    printf("Branche 2");
}
printf("Fin");
```



Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors
    ...
sinon
    ...
finsi
```



Langage C

```
if( a == b) {
    ...
}else{
    ...
}
```

Exemple de code

```
int i=2;
if( i == 2){
    printf("Branche 1");
}else{
    printf("Branche 2");
}
printf("Fin");
```



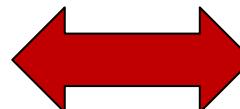
Exécution

```
>> ./a.out
Branche 1
Fin
>>
```

Si ... Sinon ...

Pseudo-langage

```
si (a = b) alors  
    ...  
sinon  
    ...  
finsi
```



Langage C

```
if( a == b ) {  
    ...  
} else {  
    ...  
}
```

Si ... Sinon ...

- Notez que le « else » n'est absolument pas obligatoire !

```
if( a == b){  
    ...  
}else{  
    ...  
}
```

```
if( a == b){  
    ...  
}
```

Cela dépend de la logique de votre code.

Si ... Sinon ...

- À l'inverse, notez qu'il est tout à fait possible de rajouter des étapes intermédiaires pour tester des cas particuliers :

```
if( a > 15){  
    /* cas si a = 15 */  
} else if (a == 5)  
    /* cas particulier si a = 5 */  
else{  
    /* Tout les autres cas */  
}
```

- On peut en mettre autant qu'on le souhaite (et le else final n'est toujours pas obligatoire)

Choix multiple

- Si je veux décrire plein de cas particulier, avec ce qu'on a vu avant je pourrais écrire :

Choix multiple

- Si je veux décrire plein de cas particulier, avec ce qu'on a vu avant je pourrais écrire :

```
if(a == 1){  
    /* cas si a = 1 */  
} else if (a == 2) {  
    /* cas particulier si a = 2 */  
} else if (a == 3) {  
    /* cas particulier si a = 3 */  
} else if (a == 4) {  
    /* cas particulier si a = 4 */  
} else if (a == 5) {  
    /* cas particulier si a = 5 */  
} else{  
    /* Tout les autres cas */  
}
```

Choix multiple

- Si je veux décrire plein de cas particulier, avec ce qu'on a vu avant je pourrais écrire :

```
if(a == 1){  
    /* cas si a = 1 */  
} else if (a == 2) {  
    /* cas particulier si a = 2 */  
} else if (a == 3) {  
    /* cas particulier si a = 3 */  
} else if (a == 4) {  
    /* cas particulier si a = 4 */  
} else if (a == 5) {  
    /* cas particulier si a = 5 */  
} else{  
    /* Tout les autres cas */  
}
```

C'est possible mais c'est moche !

Choix multiple

- Si je veux décrire plein de cas particulier, avec ce qu'on a vu avant je pourrais écrire :

```
if(a == 1){  
    /* cas si a = 1 */  
} else if (a == 2) {  
    /* cas particulier si a = 2 */  
} else if (a == 3) {  
    /* cas particulier si a = 3 */  
} else if (a == 4) {  
    /* cas particulier si a = 4 */  
} else if (a == 5) {  
    /* cas particulier si a = 5 */  
} else{  
    /* Tout les autres cas */  
}
```

C'est possible mais c'est moche !

Et pour diverses raisons d'optimisations etc. on recommande de ne pas faire ça !

Choix multiple

- Si je veux décrire plein de cas particulier, avec ce qu'on a vu avant je pourrais écrire :

```
if(a == 1){  
    /* cas si a = 1 */  
} else if (a == 2) {  
    /* cas particulier si a = 2 */  
} else if (a == 3) {  
    /* cas particulier si a = 3 */  
} else if (a == 4) {  
    /* cas particulier si a = 4 */  
} else if (a == 5) {  
    /* cas particulier si a = 5 */  
} else{  
    /* Tout les autres cas */  
}
```

C'est possible mais c'est moche !

Et pour diverses raisons d'optimisations etc. on recommande de ne pas faire ça !

Que faire alors ?

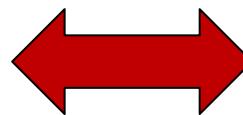
Choix multiple : le switch case !

Pseudo-langage

```
Selon  
(a=1)    faire  
...  
        fait  
faire  
...  
        fait  
Sinon faire  
...  
        fait  
Finselon
```

Langage C

```
switch (a) {  
    case 1 :  
        ...  
        break;  
    case 2 :  
        ...  
        break;  
    default :  
        ...  
        break;  
}
```



Choix multiple : le switch case !

Pseudo-langage

```
Selon  
(a=1)    faire  
...  
        fait  
faire  
...  
        fait  
Sinon faire  
...  
        fait  
Finselon
```

Langage C

```
switch (a) {  
    case 1 :  
        ...  
        break;  
    case 2 :  
        ...  
        break;  
    default :  
        ...  
        break;  
}
```



*L'instruction **break** permet d'interrompre la structure de contrôle en cours et de remonter à la structure de contrôle englobant*

Choix multiple : le switch case !

Langage C

```
switch(a) {  
    case 1 :  
        ...  
        break;  
    case 2 :  
        ...  
        break;  
    default :  
        ...  
        break;  
}
```

Choix multiple : le switch case !

Langage C

```
switch (a) {  
    case 1 :  
        ...  
        break;  
    case 2 :  
        ...  
        break;  
    default :  
        ...  
        break;  
}
```

Attention à ne pas oublier
le « break » sinon on
exécute le premier qui est
juste puis on exécute tous
les autres cas en dessous
(!!)

Choix multiple : le switch case !

Langage C

```
char a='a'  
switch(a) {  
    case 'a' :  
        ...  
        break;  
    case 'w' :  
        ...  
        break;  
    default :  
        ...  
        break;  
}
```

Note : on peut comparer a avec n'importe quoi dans les « case » tant que c'est cohérent avec le type de a, ici par exemple avec des lettres.

Mini TP d'introduction n°2

Exo 1

Écrire un programme en langage C qui fait saisir à l'utilisateur 2 entiers et qui calculera le résultat de la division euclidienne !

Le programme affiche une erreur si le dénominateur est égal à 0 (car on ne divise pas par 0 !)

Sinon le programme affiche le résultat !

Exo 2

Écrire un super système de correction de copie en langage C qui demande à l'utilisateur la première lettre de son prénom puis fait saisir à l'utilisateur le caractère en question.

Il teste ensuite ce caractère par rapport à 5 cas particuliers (les lettres a, b, c, d, e : pensez à utiliser un switch) et attribue un score à l'utilisateur (vous mettrez des scores différents et au hasard pour ces lettres et un score par défaut).

Enfin, il affiche le score.

Correction

Écrire un programme en langage C qui fait saisir à l'utilisateur 2 entiers et qui calculera le résultat de la division euclidienne !

Le programme affiche une erreur si le dénominateur est égal à 0 (car on ne divise pas par 0 !)

Sinon le programme affiche le résultat !

```
#include <stdio.h>

int main(){
    int a,b,s;
    printf("Entrez le numérateur:");
    scanf("%d",&a);
    printf(" Entrez le dénominateur:");
    scanf("%d",&b);
    if(b==0){
        printf("Erreur ! \n");
    }else
    {
        s = a/b;
        printf("la division euclidienne vaut %d\n",s);
    }
    return 0;
}
```

Écrire un programme en langage C qui demande à l'utilisateur la première lettre de son prénom puis fait saisir à l'utilisateur un caractère. Il teste ensuite ce caractère par rapport à 5 cas particuliers (les lettres a, b, c, d, e : pensez à utiliser un switch) et attribue un score à l'utilisateur (vous mettrez des scores différents pour ces lettres).

Enfin, il affiche le score.

```
#include <stdio.h>

int main(){
    int score;
    char lettre;
    printf("Entrez la première lettre de votre prénom :");
    scanf("%c",&lettre);
    switch(lettre){
        case 'a':
            score = 5;
            break;
        case 'b':
            score = 15;
            break;
        case 'c':
            score = 2;
            break;
        case 'd':
            score = 23;
            break;
        case 'e':
            score = 50;
            break;
        default :
            score = 0;
            break;
    }
    printf("le score est %d \n", score)
    return 0;
}
```

Les boucles

Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```



Langage C

```
while( a == b ) {  
    ...  
}
```

Répétition (1) : while

Pseudo-langage

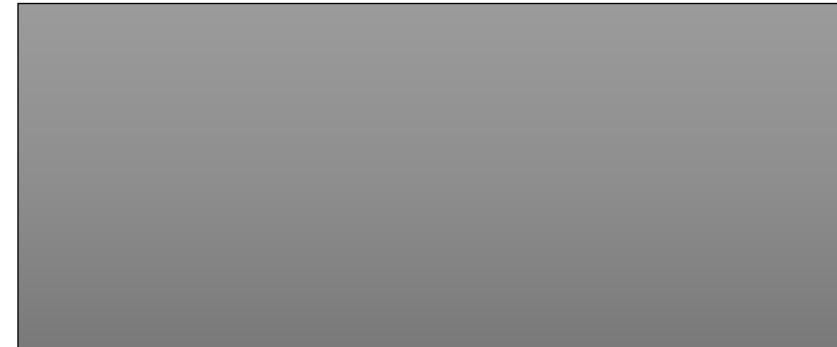
```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```



Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out
```



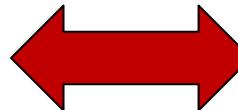
Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out
```

Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```

```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out
```

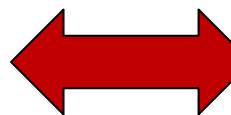
Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out
```



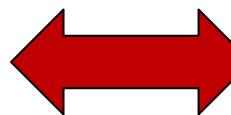
Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out  
2
```

Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```

```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out
```

```
2
```

Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```



```
>> ./a.out  
2
```

Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```

```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out
```

```
2
```

Répétition (1) : while

Pseudo-langage

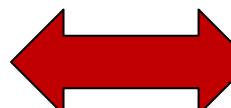
```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```

```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out  
2  
1
```



Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out  
2  
1
```



Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```



```
>> ./a.out  
2  
1
```

Répétition (1) : while

Pseudo-langage

```
Tant que (a=b) faire  
    ...  
Fin tant que
```

Langage C

```
while( a == b ) {  
    ...  
}
```



```
int i=2;  
while( i > 0){  
    printf("%d\n", i);  
    i--;  
}  
printf("Fin");
```

```
>> ./a.out  
2  
1  
Fin  
>>
```



Répétition (2) : do ... while

Pseudo-langage

Répéter

...

Jusqu'à (a=b)



Langage C

do {

...

}while(a == b);

Dans cette variante le test est à la fin et test uniquement si on Recommence la boucle, de fait, la boucle est faite au moins une fois !

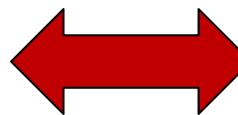
Répétition (2) : do ... while

Pseudo-langage

Répéter

...

Jusqu'à (a=b)



Langage C

```
do {  
    ...  
} while(a == b);
```

```
int i=2;  
do {  
    printf("%d\n", i);  
    i--;  
} while(i > 0);  
printf("Fin");
```

```
>> ./a.out  
2  
1  
Fin  
>>
```

Dans cette variante le test est à la fin et test uniquement si on Recommence la boucle, de fait, la boucle est faite au moins une fois !

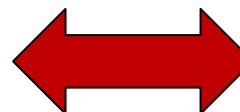
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
```

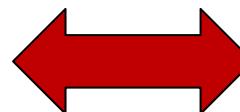
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
```

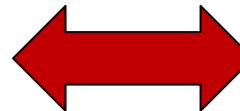
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
```



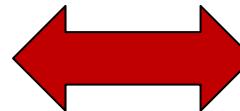
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    → printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
0
```

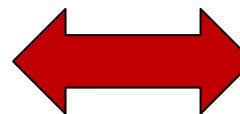
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
0
```



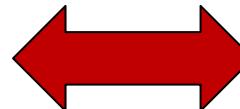
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    → printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
0
1
```

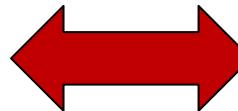
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
0
1
```



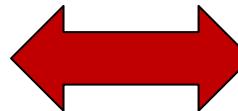
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    → printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
0
1
2
```

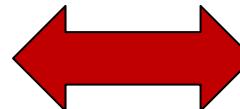
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire  
    ...  
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {  
    ...  
}
```



```
int i=0;  
for(i=0;i<=2;i++){  
    printf("%d\n", i);  
}  
printf("Fin");
```

```
>> ./a.out  
0  
1  
2
```

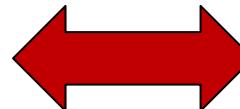
Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```



```
int i=0;
for(i=0;i<=2;i++) {
    printf("%d\n", i);
}
printf("Fin");
```



```
>> ./a.out
0
1
2
Fin
```

Répétition (3) : for

Pseudo-langage

```
pour (i ← 0 à N par pas de 1) faire
    ...
finpour
```

Langage C

```
for (i=0 ; i<=N ; i++) {
    ...
}
```

```
int i=0;
for(i=0;i<=2;i++) {
    printf("%d\n", i);
}
printf("Fin");
```

```
>> ./a.out
0
1
2
Fin
>>
```

Mini TP d'introduction n°3

Exo 1

Exo 1 : Par définition, la factorielle de n , c'est-à-dire $n!$, vau :

$1 * 2 * 3 * 4 * 5 * 6 * \dots * n$.

*On remarque que cela équivaut à dire que $n! = (n-1)! * n$*

Fort de cette information, écrivez un programme qui demande à l'utilisateur le nombre n et calcule avec une boucle la factorielle, avant de l'afficher. (vous ferez l'exo avec une boucle while, et une boucle for), attention au cas de $0! = 1$.

Exo 2

*On souhaite afficher les résultats d'une table de multiplications de taille $N*M$ ->*

L'utilisateur doit pouvoir donner M et N , ensuite on affichera le résultat de chaque case.

Pour cela on utilisera un système à double boucle.

0	1	2	3	4	...
1	1	2	3	4	..
2	2	4	6	8	..
...

Correction (juste pour le for)

Exo 1 : Par définition, la factorielle de n, c'est-à-dire $n!$, vaut :
 $1*2*3*4*5*6*...*n.$
*On remarque que cela équivaut à dire que $n! = (n-1)! * n$*
Fort de cette information, écrivez un programme qui demande à l'utilisateur le nombre n et calcule avec une boucle la factorielle, avant de l'afficher. (vous ferez l'exo avec une boucle while, une boucle do while et une boucle for), attention au cas de $0! = 1$

```
#include <stdio.h>

int main(){
    int n,i,resultat;

    printf("Factoriel de ? :\n");
    scanf("%d",&n);

    if(n==0)
    {
        resultat = 1
    }else
    {
        resultat = 1;
        for(i=1;i<=n;i++)
        {
            resultat=resultat*i;
        }
    }

    printf("le resultat est %d \n", resultat)
    return 0;
}
```

Correction (juste pour le for 2^{ème} version)

*Exo 1 : Par définition, la factorielle de n, c'est-à-dire $n!$, vau : $1*2*3*4*5*6*....*n$. On remarque que cela équivaut à dire que $n! = (n-1)! * n$*

Fort de cette information, écrivez un programme qui demande à l'utilisateur le nombre n et calcule avec une boucle la factorielle, avant de l'afficher. (vous ferez l'exo avec une boucle while, une boucle do while et une boucle for), attention au cas de $0! = 1$

```
#include <stdio.h>

int main(){
    int n,i,resultat;

    printf("Factoriel de ? :\n");
    scanf("%d",&n);

    for(i=0;i<=n;i++)
    {
        if(i==0)
        {
            resultat = 1
        }else
        {
            resultat=resultat*i;
        }
    }

    printf("le resultat est %d \n", resultat)
    return 0;
}
```

Correction exo 2

On souhaite afficher les résultats d'une table de multiplications de taille $N \times M$. L'utilisateur doit pouvoir donner M et N , ensuite on affichera le résultat de chaque case de ce tableau.

Pour cela on utilisera un système à double boucle.

0	1	2	3	4	...
1	1	2	3	4	..
2	2	4	6	8	..
...

```
#include <stdio.h>

int main(){
    int n,m,i,j;

    printf(" N = ? :\n");
    scanf("%d",&n);
    printf(" M = ? :\n");
    scanf("%d",&m);

    for(i=1;i<=n;i++)
    {
        for(j=1;j<=m;j++)
        {
            resultat = i*j;
            printf("case %d par %d = %d\n",i,j,resultat)
        }
    }
    return 0;
}
```

Correction exo 2

On souhaite afficher les résultats d'une table de multiplications de taille $N \times M$. L'utilisateur doit pouvoir donner M et N , ensuite on affichera le résultat de chaque case de ce tableau.

Pour cela on utilisera un système à double boucle.

0	1	2	3	4	...
1	1	2	3	4	..
2	2	4	6	8	..
...

```
#include <stdio.h>

int main(){
    int n,m,i,j;

    printf(" N = ? :\n");
    scanf("%d",&n);
    printf(" M = ? :\n");
    scanf("%d",&m);

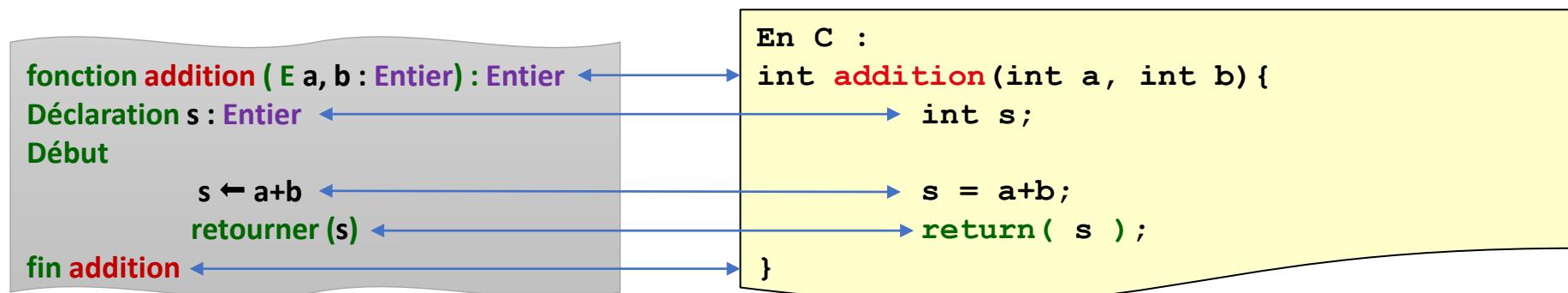
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=m;j++)
        {
            resultat = i*j;
            printf("case %d par %d = %d\n",i,j,resultat)
        }
    }
    return 0;
}
```

On aurait pu aussi mettre $i*j$ directement dans le `printf` sans passer par « `resultat` »

Les fonctions

Programme et Fonctions

- En C le main est la fonction principale, mais il est tout à fait possible d'écrire des fonctions secondaires qu'on pourra ensuite utiliser dans le main ou même dans d'autres fonctions.
- Les fonctions sont des programmes admettant des paramètres formels et retournant un seul résultat :
 - comme les fonctions mathématiques $y=f(x,y,\dots)$)
- La valeur de retour est spécifiée par l'instruction **return**



Particularités du langage C

- Une fonction ne renvoie pas forcément une donnée en sortie
 - Le mot clef **void** est utilisé pour exprimer l'idée « **d'aucune valeur** »
 - *Par exemple : une fonction qui ne ferait qu'un affichage avec des printf, n'a pas besoin de retourner quelque chose au programme principal, elle sera souvent de type void.*
- Il n'existe qu'un seul passage de paramètre en entrée d'une fonction :
le **passage par copie**
- Ce passage permet de copier-coller la valeur qu'on lui donne dans la fonction pour pouvoir travailler avec.

Créer une fonction en langage C

Méthode 1

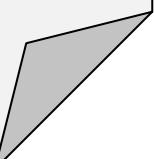
Méthode 2

Créer une fonction en langage C

Méthode 1

```
main.c
```

Méthode 2



Créer une fonction en langage C

Méthode 1

main.c

```
int maFonction(int a,int b);
```

Méthode 2



Créer une fonction en langage C

Méthode 1

main.c

```
int maFonction(int a,int b);

void main(){
    int c;
    ...
    c =  maFonction ( a, b);
    ...
}
```

Méthode 2

Créer une fonction en langage C

Méthode 1

main.c

```
int maFonction(int a,int b);  
  
void main(){  
    int c;  
    ...  
    c =  maFonction ( a, b);  
    ...  
}  
  
int maFonction(int a,int b){  
    ...  
}
```

Méthode 2

Créer une fonction en langage C

Méthode 1

main.c

```
int maFonction(int a,int b);
```

```
void main(){
    int c;
    ...
    c =  maFonction ( a, b);
    ...
}
```

```
int maFonction(int a,int b){
    ...
}
```

Méthode 2

func.h

```
int maFonction(int a,int b);
```

Créer une fonction en langage C

Méthode 1

main.c

```
int maFonction(int a,int b);
```

```
void main(){
    int c;
    ...
    c =  maFonction ( a, b);
    ...
}
```

```
int maFonction(int a,int b){
    ...
}
```

Méthode 2

func.h

```
int maFonction(int a,int b);
```

func.c

```
#include "func.h"
int maFonction(int a,int b){
    ...
}
```

Créer une fonction en langage C

Méthode 1

main.c

```
int maFonction(int a,int b);
```

```
void main(){
    int c;
    ...
    c =  maFonction ( a, b);
    ...
}
```

```
int maFonction(int a,int b){
    ...
}
```

Méthode 2

func.h

```
int maFonction(int a,int b);
```

func.c

```
#include "func.h"
int maFonction(int a,int b){
    ...
}
```

main.c

```
#include "func.h"
void main(){
    int c;
    ...
    c =  maFonction ( a, b);
}
```

Syntaxe des fonctions

Définition

```
int monAddition( int a, int b ) {  
    int res;  
    res = a + b;  
    return( res );  
}
```

Paramètres d'entrée

Paramètres de retour

Utilisation

```
void main(){  
    int c;  
    ...  
    c = monAddition( 1, 2)  
    ...  
}
```

Portée des variables

- Variables globales
 - Déclarées en dehors d'une fonction, accessible partout ! (on évite en général de les utiliser pour des raisons d'optimisation et sécurité)

```
int MA_VARIABLE;  
void fonction () {  
    ...  
}
```

- Variables locales
 - Déclarées dans une fonction : accessible uniquement dans la fonction !

```
void fonction () {  
    int maVariable;  
}
```

Le passage par copie ?

- Le fait que le passage est par copie engendre des limitations

```
int mafonction( int a, int b ){
    a = 3+a;
    b = 5+b;
    res = a+b
    return( res );
}
```

Utilisation

```
void main(){
    int a = 1;
    int b = 2;
    ...
    c = mafonction( 1, 2 )
    ...
}
```

Le passage par copie ?

- Le fait que le passage est par copie engendre des limitations

Ici, a et b sont copié dans ma fonction dans de nouvelles variables nommées elles aussi a et b mais qui n'existe que localement !

De fait, en sortant de « mafonction » les a et b du main n'ont pas été modifié !

```
int mafonction( int a, int b ){
    a = 3+a;
    b = 5+b;
    res = a+b
    return( res );
}
```

Utilisation

```
void main(){
    int a = 1;
    int b = 2;
    ...
    c = mafonction( 1, 2 )
    ...
}
```

Autres limitations

- Les fonctions permettent de sortir la valeur d'une seule variable
 - L'instruction **return (a ,b)** n'existe pas!!!!
- Si l'on désire plusieurs valeurs de variable en sortie de la fonction/procédure, on ne peut pas le faire en l'état actuel de nos connaissances. Nous verrons plus tard comment le faire !
 - **On évite d'utiliser des variables globales ! C'est considéré comme du très mauvais code.**

Résumé :

test_carre.c

```
#include <stdio.h>

float carre( float x );

main(){ /* debut de la fonction main */

    float pi2;
    pi2 = carre( 3.14 );
    printf("PI^2 vaut %f", pi2);

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */

    float x2;
    x2 = x*x;
    return( x2);

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x );

main(){ /* debut de la fonction main */

    float pi2;
    pi2 = carre( 3.14 );
    printf("PI^2 vaut %f", pi2);

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */

    float x2;
    x2 = x*x;
    return( x2);

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main(){ /* debut de la fonction main */

    float pi2;
    pi2 = carre( 3.14 );
    printf("PI^2 vaut %f", pi2);

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */

    float x2;
    x2 = x*x;
    return( x2 );

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main(){ /* debut de la fonction main */
    float pi2; ← Déclaration de la variable locale pi2
    pi2 = carre( 3.14 );
    printf("PI^2 vaut %f", pi2);

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */

    float x2;
    x2 = x*x;
    return( x2 );

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main(){ /* debut de la fonction main */

    float pi2; ← Déclaration de la variable locale pi2
    pi2 = carre( 3.14 )← Utilisation du sous-programme carré avec comme argument
                           d'entrée PI. Affectation du résultat à la variable pi2
    printf("PI^2 vaut %f", pi2);

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */

    float x2;
    x2 = x*x;
    return( x2 );

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main(){ /* debut de la fonction main */

    float pi2; ← Déclaration de la variable locale pi2
    pi2 = carre( 3.14 )← Utilisation du sous-programme carré avec comme argument
                           d'entrée PI. Affectation du résultat à la variable pi2
    printf("PI^2 vaut %f", pi2); ← Affichage de la variable pi2 à
                                   l'écran

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */

    float x2;
    x2 = x*x;
    return( x2 );

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main(){ /* debut de la fonction main */

    float pi2; ← Déclaration de la variable locale pi2
    pi2 = carre( 3.14 )← Utilisation du sous-programme carré avec comme argument
                           d'entrée PI. Affectation du résultat à la variable pi2
    printf("PI^2 vaut %f", pi2); ← Affichage de la variable pi2 à
                                   l'écran

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */ ← Définition de la
                           fonction carre

    float x2;
    x2 = x*x;
    return( x2 );

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main() { /* debut de la fonction main */

    float pi2; ← Déclaration de la variable locale pi2
    pi2 = carre( 3.14 ); ← Utilisation du sous-programme carré avec comme argument
                           d'entrée PI. Affectation du résultat à la variable pi2
    printf("PI^2 vaut %f", pi2); ← Affichage de la variable pi2 à
                                   l'écran

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */ ← Définition de la
                      fonction carre

    float x2; ← Déclaration d'une variable x2 de type float
    x2 = x*x;
    return( x2);

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main() { /* debut de la fonction main */

    float pi2; ← Déclaration de la variable locale pi2
    pi2 = carre( 3.14 ); ← Utilisation du sous-programme carré avec comme argument
                           d'entrée PI. Affectation du résultat à la variable pi2
    printf("PI^2 vaut %f", pi2); ← Affichage de la variable pi2 à
                                   l'écran

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */ ← Définition de la
                      fonction carre

    float x2; ← Déclaration d'une variable x2 de type float
    x2 = x*x; ← On affecte a x2 le résultat de l'opération x*x
    return( x2 );

} /* fin de la fonction carré */
```

Résumé :

test_carre.c

```
#include <stdio.h> ← Header de la bibliothèque standard (in/out) pour la fonction printf

float carre( float x ); ← Déclaration d'une fonction carre prenant comme argument d'entrée
                           un variable de type float et renvoie en sortie une variable de type float

main() { /* debut de la fonction main */

    float pi2; ← Déclaration de la variable locale pi2
    pi2 = carre( 3.14 ); ← Utilisation du sous-programme carré avec comme argument
                           d'entrée PI. Affectation du résultat à la variable pi2
    printf("PI^2 vaut %f", pi2); ← Affichage de la variable pi2 à
                                   l'écran

} /* fin de la fonction main */

float carre( float x){ /* début de la fonction carre */ ← Définition de la
                      fonction carre

    float x2; ← Déclaration d'une variable x2 de type float
    x2 = x*x; ← On affecte a x2 le résultat de l'opération x*x
    return( x2); ← On renvoie en sortie de fonction la valeur de la
                   variable x2

} /* fin de la fonction carré */
```

Notes :

- Vous aurez maintenant remarqué que le « main » est une fonction comme les autres !
 - C'est une fonction qui renvoi au système un chiffre :
 - **0 Si tout s'est passé comme prévu.**
 - **Un code d'erreur sinon (-1 par exemple).**
- De fait il est courant d'écrire que la fonction main est de type int et qu'elle retourne à la toute fin 0.

```
int main()
{
    // reste du programme //
    return 0;
}
```

Mini TP :

Exercice 1

*Écrire une fonction en langage C nommée **cube** qui prend en entrée un nombre à virgule et qui calcul sa valeur cubique. Tester votre fonction dans un programme principale avec la valeur 3.*

Exercice 2

*Ecrire une fonction en langage C nommée **max** qui renvoie la valeur max de 2 entiers. Tester votre fonction dans un programme principale qui affiche le résultat avec les valeurs 2 et 3*

Correction : exercice 2

*Écrire une fonction en langage C nommée **cube** qui prend en entrée un nombre à virgule et qui calcul sa valeur cubique. Tester votre fonction dans un programme principale avec la valeur 3.*

```
#include <stdio.h>

double cube(double a) {
    int res;
    res = a * a * a;
    return res;
}
int main(){
    double a=3,2;
    double resultat;
    resultat = cube(a);
    printf("a^3=%d",a);
    return 0;
}
```

Correction exercice 2

*Écrire une fonction en langage C nommée **max** qui renvoie la valeur max 2 entiers. Tester votre fonction dans un programme principale avec les valeurs 2 et 3.*

```
#include <stdio.h>

int max(int a, int b){
    int res;
    if(a>b) {
        res=a;
    }else{
        res=b;
    }
    return res
}
int main(){
    int a=2,b=3;
    int mx;
    mx = max(a,b);
    printf("max=%d",mx);
    return 0;
}
```

Gestion des sources et des autres fichiers

Inclusion de sources

- Comme il est souvent préférable de ne pas réinventer la roue, de nombreuses fonctions existent qui sont déjà écrites, elles sont mises à disposition sous forme de Librairie (Ou **Bibliothèque**)
- Ce sont des ensembles de fonctions prêtes à l'usage !



Inclusion de sources

- Les bibliothèques sont définies par des fonctions dans des fichiers binaires (fichier objet : c'est comme un fichier exécutable, mais sans fonction main)
- Les fonctions utilisables sont décrites dans les .h de la bibliothèque.

toto_et_tata.o

Code binaire de :
`int toto()
int tata()`
0001110101010101 ...
1001010101010010 ...

toto_et_tata.h

Fichier texte écrivant simplement :
`int toto()
int tata()`
Permettant de savoir que ces deux fonctions existent.

Inclusion de sources

- Il faut donc inclure ces deux choses :
 - Pour copier le .h au début de vos fichiers, deux solutions :
 - Trouver le .h et copier-coller le contenu du texte à la bourrin (pas recommandé)
 - **Ou utiliser la directive include !**

```
#include "nom_de_fichier"
```

```
#include <nom_de_fichier>
```

Inclusion de sources

- Il faut donc inclure ces deux choses :
 - Pour copier le .h au début de vos fichiers, deux solutions :
 - Trouver le .h et copier-coller le contenu du texte à la bourrin (pas recommandé)
 - **Ou utiliser la directive include !**

```
#include "nom_de_fichier"
```

```
#include <nom_de_fichier>
```

- Dans le système UNIX :
 - **<nom_de_fichier>** représente un fichier système
 - Le fichier sera recherché dans les répertoires système connus du compilateur
 - **"nom_de_fichier"** représente un fichier utilisateur
 - Le fichier sera recherché dans le répertoire courant

Inclusion de sources

- Il faut donc inclure ces deux choses :
 - Pour inclure le fichier objet, il faut le préciser à gcc avec l'option « -l »
 - Exemple avec la librairie mathématique, qui s'abrége en « m » pour math :

```
gcc -o monprogramme.exe -lm monprogramme.c
```

- Il existe aussi les options :
 - –L pour lui préciser le chemin où il doit chercher si celui-ci n'est pas évident
 - –I pour lui préciser le chemin où il doit chercher les .h si celui-ci n'est pas évident.
- Inutile de préciser le chemin pour les outils standards déjà fournit comme la bibliothèque mathématique, gcc sait où trouver celles-ci sur le système (en général).

Quelques bibliothèques standards...

Fichier	Contenu
<code><stdio.h></code>	Fonctions puts(), gets(), printf(), scanf(),...
<code><limits.h></code>	Indique les limites des types
<code><string.h></code>	Traitement des chaînes, copie, concaténation, recherche de sous-chaînes
<code><math.h></code>	Fonctions mathématiques
<code><stdlib.h></code>	Allocation dynamique de mémoire (malloc), conversion nombre vers ascii,...
<code><time.h></code>	Fonctions liées à l'heure et la génération des nombres aléatoires

Complément :

- En plus du `#include`, il existe d'autres directive commençant par `#`, la plus utile pour vous pour le moment étant sans doute le :

#define

- Les équivalences sont remplacées par leur valeur par le pré-processeur

#define identificateur reste-de-la-ligne

- Exemple :

#define PI 3.14

Remplace tout mot « PI »
par 3,14

Les mots définis par équivalence sont généralement mis en majuscule car ils sont globaux et constants

Attention : ce ne sont pas du tout des variables ! C'est juste un remplacement de mot à l'échelle du fichier => place en mémoire = 0





Commençons le TP1 !