

# Techniques de Programmation

## *Les bases du C*

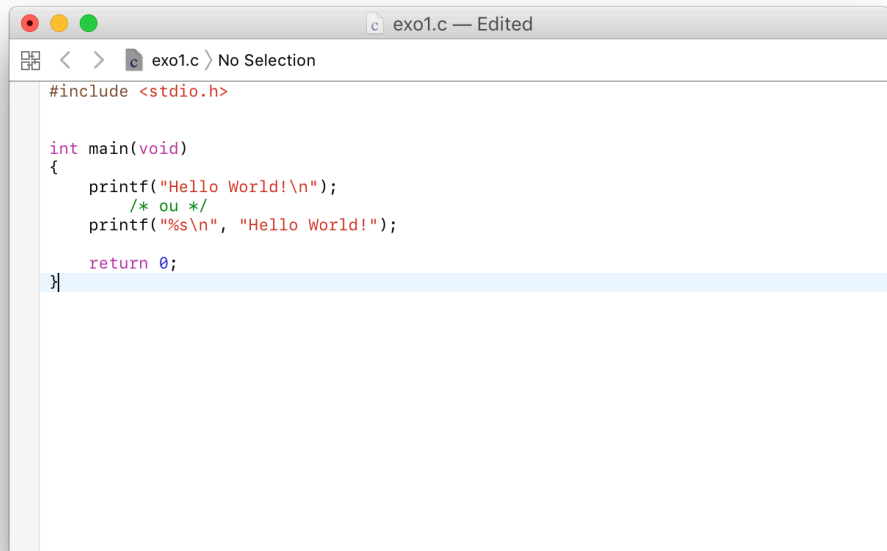
Nicolas Belloir

[nicolas.belloir@univ-pau.fr](mailto:nicolas.belloir@univ-pau.fr)

# Introduction

# Langages de hauts-niveaux

- Le C est un langage de **haut niveau**
- Les langages de hauts-niveaux doivent être **traduits** vers le langage machine afin d'être exécutés



```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    /* ou */
    printf("%s\n", "Hello World!");

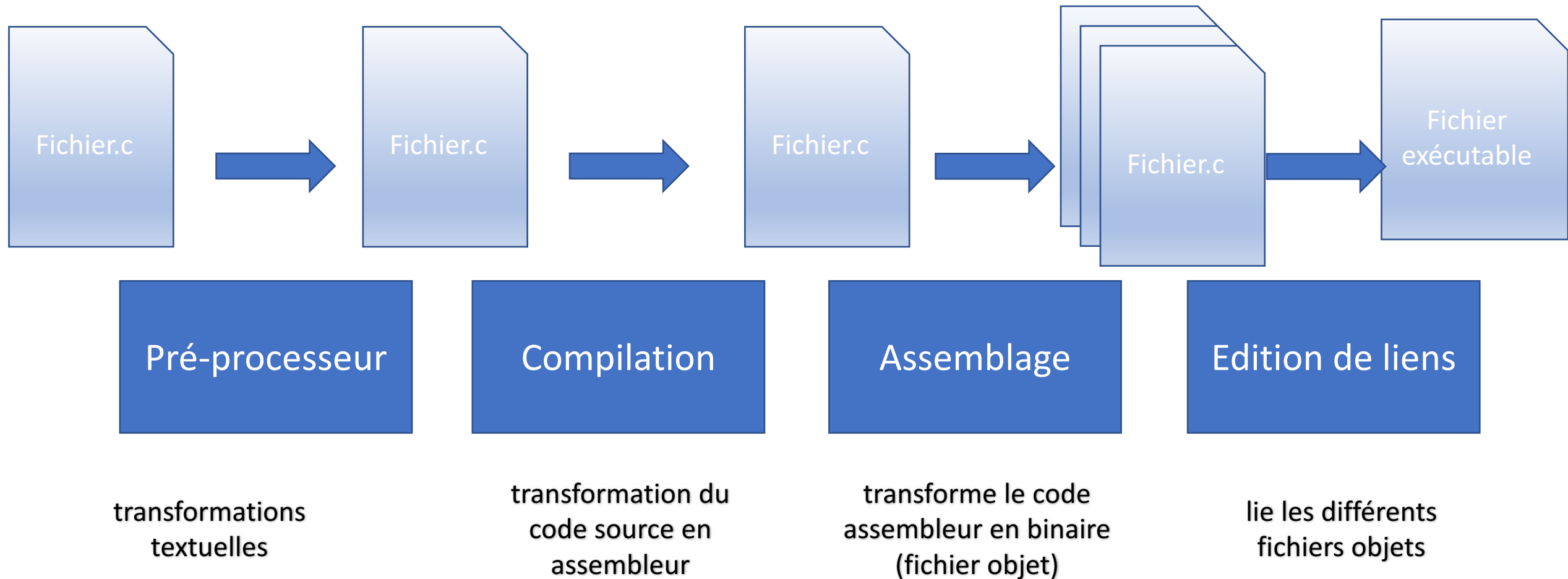
    return 0;
}
```



# Langages de hauts-niveaux

- Les langages **interprétés**
  - Traduction instruction par instruction.
  - Le programme est traduit a chaque exécution.
  - Java, Scheme, Shell Unix . . .
- Les langages **compilés**
  - Traduction des programmes dans leur ensemble.
  - Le programme est traduit en une seule fois pour générer un binaire exécutable.
  - C, C++, Pascal . . .
  - Efficacité du programme, propriété intellectuelle...

# Le langage C, un langage compilé



# Structure d'un programme C

# Le langage C, un langage structuré

- Basé sur **3 niveaux d'écriture**
  - **Fichier**
    - Contient un ensemble de fonctions
    - Une fonction peut s'appeler **main**
      - Point d'entrée du programme
  - **Fonction**
    - Contient un seul bloc et peut avoir
      - Des paramètres
      - Une valeur de retour
  - **Bloc**
    - Contient un **ensemble d'instructions**
    - Délimitées par **{ }**
- **Chaque composante possède ses propres variables**

Fichier toto.c

```
int var_globale;

void fa () {
    ...
}

int fb () {
    ...
}

int main () {
    ...
}
```

Détail fonction fb

```
int fb (int x) {
    ...
    return x+1;
}
```

Bloc lambda

```
{
    int A;
    A = A + 1;
}
```

# Exemple – Le fichier Bonjour.c

```
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```



# Exemple – Le fichier TVA.c

```
#include <stdio.h>                                //preprocesseur
#define TVA 19.6

float calcule_TVA(float prixHT) {                  // fonction
    return ((prixHT * TVA) / 100);
}

int main () {                                      // point d entrée du programme
    float HT;
    scanf ("%f", &HT);                            // saisie prix HT
    HT=HT + calcule_TVA (HT);                      //calcul du prix TTC
    printf ("prix TTC = %f\n", HT);                // affichage
    printf("\n");                                  //Retour à la ligne
    return (0);                                    // Point de sortie du programme
}
```

# Les identificateurs

- Suite de caractères permettant de reconnaître une entité du programme (Variables, Fonctions, ...)
- Est compose d'un ou plusieurs caractères.
  - Le premier parmi [A..Z] [a..z] [\_]
  - Les suivants parmi [A..Z] [a..z] [\_] [0..9]
- Exemples :
  - `nombre_max`, `NbMax`, `_nombMax`, `_FC012B`, ...

# Mots clés réservés

- Le langage C possède un vocabulaire dont les mots ne peuvent pas être pris comme des identificateurs d'un programme.

auto	break	case	char	continue	default
do	double	else	enum	entry	extern
float	for	int	long	register	return
short	sizeof	static	struct	switch	typedef
union	unsigned	void	while		

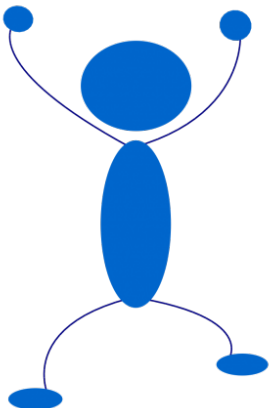
# Variables, types et declaration

# Variable

- Association d'un **identificateur** a une **zone mémoire** contenant la **valeur** de cette variable
- possède une classe d'enregistrement, un type
- doit être déclarée et être allouée avant son utilisation

*Une sorte de boîte dans laquelle on stocke un objet  
d'un type spécifique!*

*« On ne met pas un poisson dans une caisse à outils ni un  
tournevis dans le frigo »*



# Types de bases

char	(1)
short	(2)
int	(4)
long int	(8)
long long int	(8)

entiers signés

float	(4)
double	(8)
long double	(16)

Flottants

unsigned char	
unsigned int	
unsigned short	
unsigned long int	
unsigned long long int	

entiers positifs

char*	(8)
int*	(8)
...	
void*	(8)

pointeurs

# Types de bases

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	$3.4 \cdot 10^{-38}$ à $3.4 \cdot 10^{38}$
double	Flottant double	8	$1.7 \cdot 10^{-308}$ à $1.7 \cdot 10^{308}$
long double	Flottant double long	10	$3.4 \cdot 10^{-4932}$ à $3.4 \cdot 10^{4932}$

# Exemple de code

```
#include <stdio.h>
#define TAILLE 20
int somme;

void main() { /* Fonction principale */ /* Début du bloc de la fonction main */
    int a;
    int b; /* Variables du bloc */
    printf("Entrez deux entiers séparés par un blanc\n");
    scanf("%d %d",&a,&b);
    /* Affichage d'un message, et saisie au clavier de deux entiers */
    if (a > = TAILLE) { /*Début du bloc */
        int c; /*Variable du bloc */
        c = a + TAILLE; /*Calcul de la valeur de la variable c */
        a = 2 * c; /*Calcul de la valeur de la variable a */
    } /*Fin du bloc */
    somme = a + b; /*calcul de la somme */
    printf("La somme des deux entiers est : % d \n", somme); /*Affiche la valeur
de la variable somme */
} /*Fin du bloc de la fonction main */
```



# Instructions, operateurs, et constantes

# Instruction

- Définition

- **action élémentaire** du langage qui se termine obligatoirement par le caractère ' ;'.
- Les instructions sont exécutées **séquentiellement** de la première à la dernière sauf dans le cas d'instructions de contrôle.

- Exemple

```
;
A=1;
printf("Entrez deux entiers séparés par un blanc\n");
```

# Opérateurs arithmétiques

-	- <exp>
* / %	<exp1> OP <exp2>
- +	<exp1> OP <exp2>

```
a = b - 12 ;
```

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

```
b = b * ( 12 + c ) ;
```

# Opérateurs logiques et relationnels

!	! <exp>
%%	<exp1> OP <exp2>
== != < > >= <=	<exp1> OP <exp2>

```
a = !b;
```

```
/* a reçoit non b */
```

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

```
/* a reçoit le résultat de la  
comparaison de l'égalité de b avec c */
```

```
a = b && c ;
```

```
/* a reçoit le résultat de la  
l'opération logique b ET c */
```

# Opérateurs d'affectation

```
=          <LVALUE> = <exp>  
OP =      < LVALUE> OP= <exp2>
```

```
a = b;          /* a reçoit b */  
a += b;         /* a reçoit la somme de a avec b */  
a *= b;         /* a reçoit le produit de a par b */
```

ERREUR CLASSIQUE : Le programmeur veut affecter b à a.

```
a = b;          /* OK */  
a == b;         /* KO */
```

# Opérateurs d'incrémentation

++	++<LVALUE> ou <LVALUE>++
--	--< LVALUE> ou < LVALUE>--

```
i=1;
```

```
a = i++;
```

```
b = ++i;
```

```
/* a reçoit i puis i est incrémenté de 1*/
```

```
/* i est incrémenté de 1 puis b reçoit i */
```

# Opérateurs d'indirection

*	*<exp>
&	&< LVALUE>

```
int a;  
int *p1 = &a;
```

/\* le pointeur p1 reçoit l'adresse de la variable a\*/

```
int b;  
b = *a;
```

/\* b reçoit la valeur de la variable pointée par p1 \*/

# Opérateurs de calcul de taille

```
sizeof  
sizeof <exp>  
sizeof (<nom type>)
```

```
printf („%d“, sizeof(char)); /*donne la taille en octet d'un type  
                               char */
```



# Les constantes

Decimales	10	-32768L
Octales	03677	0177
Hexadecimales	0xFF	0xA1L
Reelles	3.14159	-1,234e18
Caracteres	'A'	'\n'
Chaînes	"Bonjour"	"Oui"

## ATTENTION!

Les caractères sont entre simples cotes  
Les chaînes de caractères sont entre double cotes

# Les constantes nommées

Il vaut mieux éviter de manipuler des constante valuées directement dans le code!  
Préférer les **constantes nommées**!

```
#define NOTE_MAX 20
#define NOTE_MIN 0

int main() {
    int note;
    scanf (« %d », &note);
    if ((note < NOTE_MIN) || (note > NOTE_MAX)
        printf (« note saisie incorrecte. La note doit
            être en %d et %d \n, NOTE_MIN, NOTE_MAX);
    return 0;
}
```

# Instructions de contrôle

# Les conditionnelles

```
if (a<b)
    printf (« a est plus petit que b\n »);
else
    printf (« b est plus petit que (ou égal à) b\n »);
```

```
if (a<b){
    printf (« a est plus petit que b\n »);
    a = b++;
}
else{
    printf (« b est plus petit que (ou égal à) b\n »);
    b = a++;
}
```

# Les conditionnelles

Attention, erreur classique!!!!

```
/* le programmeur veut comparer a et b */  
/* il écrit : */                if (a=b) ...  
/* il aurait dû écrire : */    if (a==b) ...
```

# Les itératives : le *while*

- L'instruction est répétée **tant que le résultat de l'expression est VRAI**.
- S'utilise lorsque le **nombre d'itération n'est pas déterminé avant le départ** de la première itération.
- La condition est évaluée avant le départ de l'itération.

```
while ( a != 10 ) {  
    sum = sum + a ;  
    a = a + 1 ;  
}
```

si a vaut 0 il y aura 10 itérations.  
si a > 10 il y aura un nombre indéterminé d'itérations.

# Les itératives : le *for*

- L'instruction est répétée tant que le résultat de l'expression condition est VRAI.
- S'utilise lorsque le nombre d'itération à exécuter est exactement défini.

```
sum = 0;
for( i = 0; i < 10; i++ ) {
    sum = sum + i ;
}
```

# Equivalence entre la boucle for et while

```
i = 0;
while( i < 0) {
    sum = sum + i ;
    i++;
}
```

```
sum = 0;
for( i = 0; i < 10; i++ ) {
    sum = sum + i ;
}
```



# Remarque

Dans toutes les formes d'itérations, il faut veiller à **faire évoluer la condition** pendant l'itération.

```
/* Exemples à ne pas suivre ! Itérations infinies */

i = 4 ;
while ( i < 10 )
    sum = sum + i ;

/* ... */

for ( i = 0 ; i < 10 ; a++)
    sum = i + sum;
```

# Les itératives : le *faire ... tant que*

- L'instruction est répétée tant que le résultat de l'expression est VRAI.
- Dans ce type d'itération, la condition n'est vérifiée qu'après l'itération, donc l'instruction est au moins exécutée une fois.

```
do{  
    sum = sum + i ;  
    i = i + 1 ;  
} while (i <10);
```

# Les itératives : la sortie de boucle

- Il est possible de **sortir** d'une itération de manière **non algorithmique**.
- Utilisation de l'instruction **break**.
- L'exécution se poursuit à la sortie immédiate du bloc.

```
i =0;
while ( i < 20) {
    sum = sum + i ;
    if (i==14) /* Arrêt de la boucle lorsque i vaudra 14 */
        break ;
    i = i + 1 ;
}
printf (" somme = %d \n", somme);
```

# Les itératives : le retour en début de boucle

- Il est possible de **revenir en début de boucle** d'une itération de manière **non algorithmique**.
- Utilisation de l'instruction ***continue***.

```
i =0;
while ( i <20) {
    if (i == 12){ /* elimine le cas i==12 de l'itération */
        i++;
        continue;
    }
    sum += 1 ;
    i++;
}
```

# Autre instruction : le retour de fonction

- L'instruction **return** est utilisée pour terminer l'exécution d'une fonction avec la possibilité de retourner une valeur au contexte qui a appelé cette fonction.

```
int main() {  
    int x = 2;  
    int y = carre(x);  
    printf ("y = %d \n", y);  
    return 0;  
}
```

```
int carre(int a) {  
    return (a*a);  
}
```

# Autre instruction : le choix multiple

- L'instruction **switch** permet de mettre en place une structure d'exécution qui permet des choix multiples parmi des cas de même type et faisant intervenir uniquement des valeurs constantes entières (*char, short, int*).

```
switch (val) {  
    case 1 :  
        ...  
        break;  
    case 2 :  
        ...  
        break;  
    case 3 :  
        ...  
        break;  
    default:  
        ...  
}
```

# Autre instruction : le choix multiple

```
int a;
printf ("Saisissez une valeur entière entre 1 et 3\n");
scanf ("%d", &a);
switch (a){
    case 1 :
        printf ("valeur saisie : 1\n");
        break;
    case 2 :
        printf ("valeur saisie : 2\n");
        break;
    case 3 :
        printf ("valeur saisie : 3\n");
        break;
    default:
        printf ("valeur saisie : autre que 1, 2 ou 3\n");
}
```

# Autre instruction : le choix multiple

```
switch(a) {  
    case 1 : b++;  
    case 2 : b++;  
             break ;  
    case 3 : b-- ;  
    default : b-- ;  
}  
/* resultat final */
```

<i>a==1</i>	<i>a==2</i>	<i>a==3</i>	<i>autre</i>
+1			
+1	+1		
fin	fin		
		-1	
		-1	-1
		fin	fin
b+2	b+1	b-2	b-1



# Autre instruction : le choix imbriqué

```
if ( a<b )  
    printf("a plus petit que b");  
else if ( a>b )  
    printf("a plus grand que b");  
else  
    printf("a égal à b");
```

# Autre instruction : l'opérateur conditionnel : ?

- Format : `<condition> ? <expression1> : <expression2>`
- Le résultat de l'évaluation est conditionne par la condition :
  - si `<condition> == VRAI` résultat est `<expression1>`
  - si `<condition> == FAUX` résultat est `<expression2>`

```
int plusgrand = ( a > b ) ? a : b ;
```

```
/* ... */
```

```
while ( a != b )  
    a > b ? a-- : a++;
```

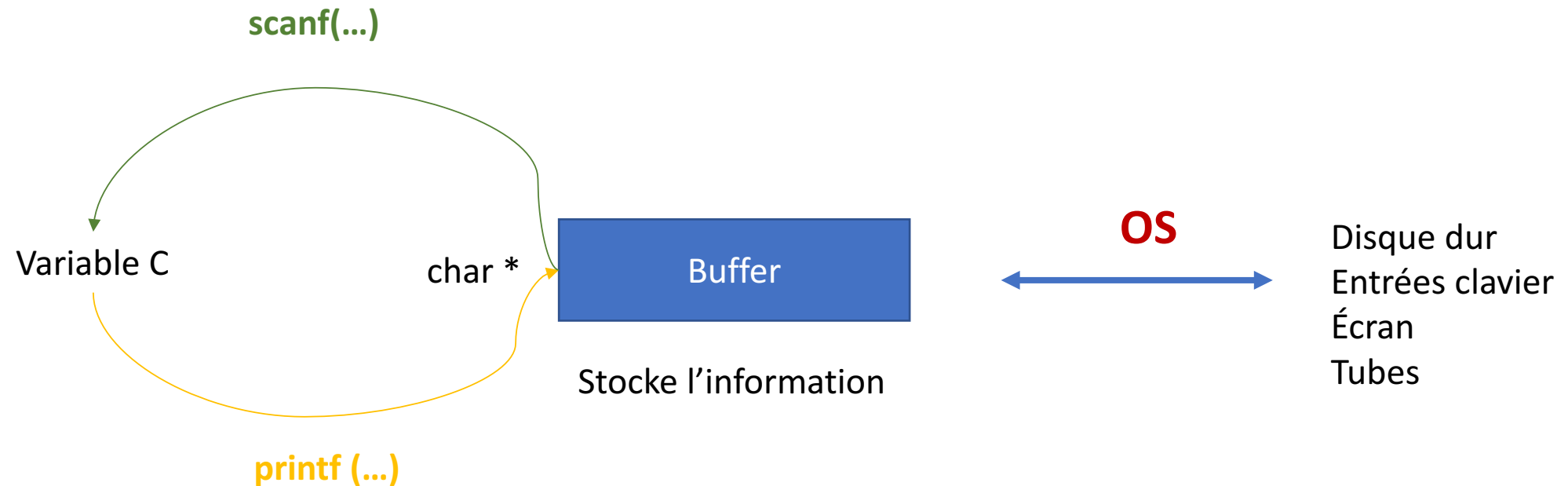
# Les entrées sorties formatées

# Les entrées sorties

- Sont de plusieurs sortes
  - Affichage, lecture clavier, fichiers...
- Ecran
  - Information utilisateur
  - Debug d'un programme
  - Communication avec l'utilisateur
- Fichier
  - Suivi execution (fichier log)
  - Sauvegarde information (disque=mémoire non volatile)
  - Communication entre programme (disque=mémoire partagée)

# Les entrées sorties

- En C, l'écriture fichier et écran sont similaires



# Rappel sur les caractères : *char*

- Un caractère (char)
  - nombre entre 0 et 256
- 1 octet => 1 emplacement mémoire => 2 caractères hexa

- Exemple

- `char c = 'M';` ← 4D

- `int value = (int) c;` ← 77

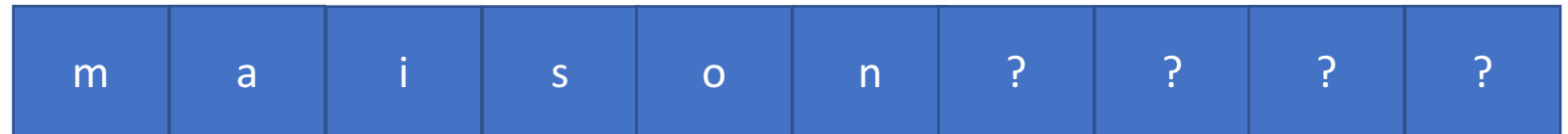


Cast : fonction de conversion

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	\$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

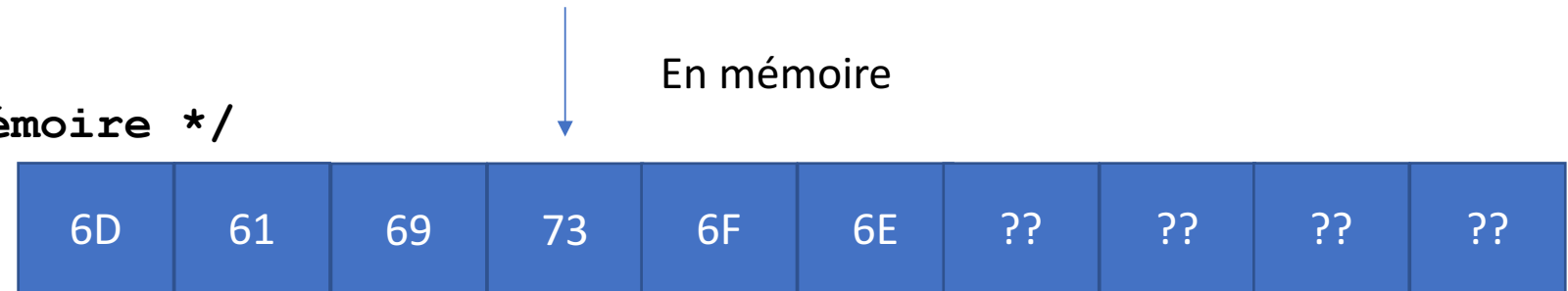
# Les chaînes de caractères

- Suites de caractères stockées dans un tableau de caractères
- Définies par *const char \**, *char []*



```
char nom [10];  
/*Réserve 10 emplacements mémoire */
```

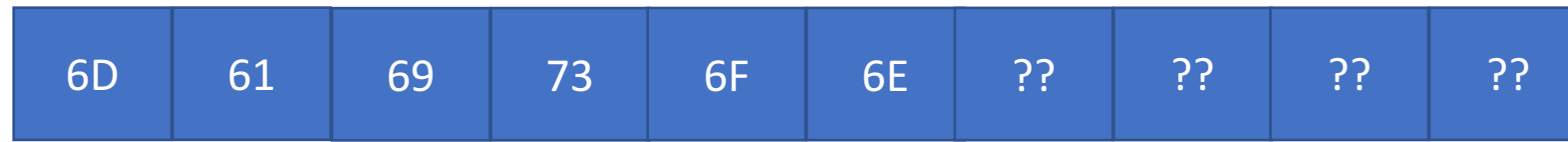
```
nom[0] = 'm' ;  
nom[1] = 'a' ;  
nom[2] = 'i' ;  
nom[3] = 's' ;  
nom[4] = 'o' ;  
nom[5] = 'n' ;
```



# Les chaînes de caractères



En mémoire



**nom** correspond à la première case de la chaîne

**nom** = pointeur sur la première case

## Comment savoir où s'arrêter?

```
#include <stdio.h>

int main() {
    char nom [10];

    nom[0] = 'm';
    nom[1] = 'a';
    nom[2] = 'i';
    nom[3] = 's';
    nom[4] = 'o';
    nom[5] = 'n';

    printf("Chaîne : %s",
nom);

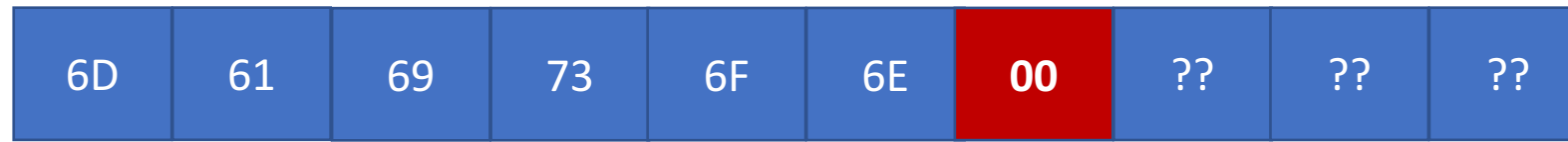
    return 0;
}
```



# Les chaînes de caractères



En mémoire



Fin de chaine = principe de sentinelle de fin

affiche : maison

(comportement déterministe)

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

```
int main() {  
    char nom [10];
```

```
    nom[0] = 'm';
```

```
    nom[1] = 'a';
```

```
    nom[2] = 'i';
```

```
    nom[3] = 's';
```

```
    nom[4] = 'o';
```

```
    nom[5] = 'n';
```

```
    nom[6] = '\0';
```

```
    printf("Chaine : %s",  
    nom);
```

```
    return 0;
```

```
}
```

# Les chaînes de caractères

Fin de chaine = `'\0'`

Toujours prévoir une case pour placer `'\0'`

```
int calculTaille(char mot []){  
    char *pointeur_debut = mot;  
    char *pointeur_fin = pointeur_debut;  
  
    while(*pointeur_fin != '\0')  
        ++pointeur_fin;  
  
    return (int)(pointeur_fin - pointeur_debut);  
}
```

Si pas de `'\0'`, comportement indéterministe

# Les chaînes de caractères : bonnes pratiques

fonctions de manipulation de chaine: `#include <string.h>`

`str<...>`

`strcpy`  
`strcmp`  
`strcat`  
...

Jusqu'à `'\0'`

`strncpy`  
`strncmp`  
`strncat`  
...

Jusqu'à `'\0'` avec  
un maximum de **n**  
caractères

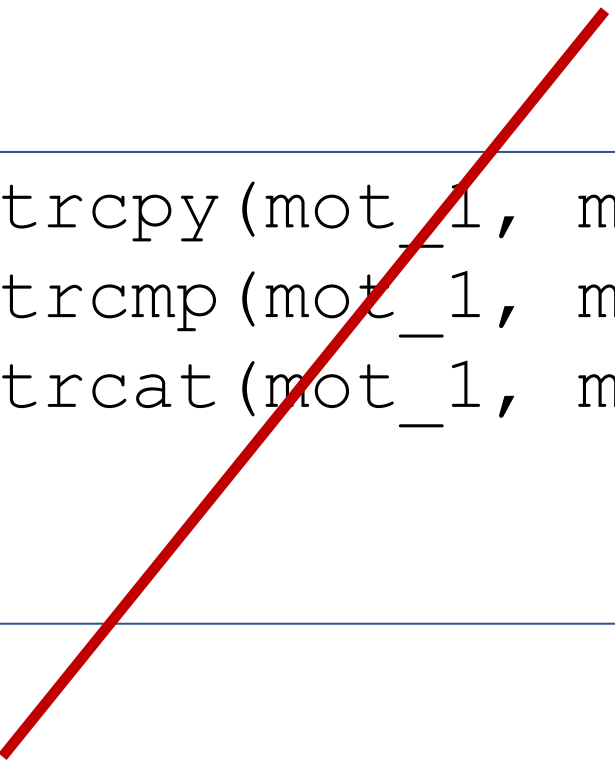
→ copie  
→ compare  
→ concatène  
...

Préférer les fonction à taille limitées  
(plus sûres, debug plus aisé)

# Les chaînes de caractères : bonnes pratiques

Toujours utiliser les fonctions à tailles limitées:

**str**n<...>



```
strcpy(mot_1, mot2);  
strcmp(mot_1, mot2);  
strcat(mot_1, mot2);
```

...

```
strncpy(mot_1, mot2, nb_max);  
strncmp(mot_1, mot2, nb_max);  
strncat(mot_1, mot2, nb_max)
```

...

Plus sécurisé  
Débug plus facile

# Les chaînes de caractères : bonnes pratiques

Ne pas utiliser d'opérateurs sur des chaîne de caractères !!!

```
#include <stdio.h>

int main(){
    char mot_1[] = "abricot";
    char *mot_2 = "peche";
    mot_1 = mot_2;
    printf(" %s\n ", mot_2);
    char mot_3[] = "abricot";
    if(mot_1==mot_3)
        printf("meme mot");
    char *mot_4 = "hello";
    char *mot_5 = "world";
    mot_4 += *mot_5;

    return 0;
}
```

Attention ne donne aucun warning  
à la compilation!

# Les chaînes de caractères : bonnes pratiques

Ne pas utiliser d'opérateurs sur des chaîne de caractères !!!

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

```
int main(){
```

```
    char mot_1[] = "abricot";
```

```
    char *mot_2 = "peche";
```

```
    mot_1 = mot_2;
```

```
    printf(" %s\n ", mot_2);
```

```
    char mot_3[] = "abricot";
```

```
    if(mot_1==mot_3)
```

```
        printf("meme mot");
```

```
    char *mot_4 = "hello";
```

```
    char *mot_5 = "world";
```

```
    mot_4 += *mot_5;
```

```
    return 0;
```

```
}
```

OK

Mauvaise habitude

Affectation de pointeurs !!

Ce n'est pas une copie de chaîne!

Test d'égalité d'adresses de pointeurs!

Ne compare pas les chaînes

Mauvaise habitude

Mauvaise habitude

ajoute (int)(mot\_5[0])=119 à l'adresse de mot\_4

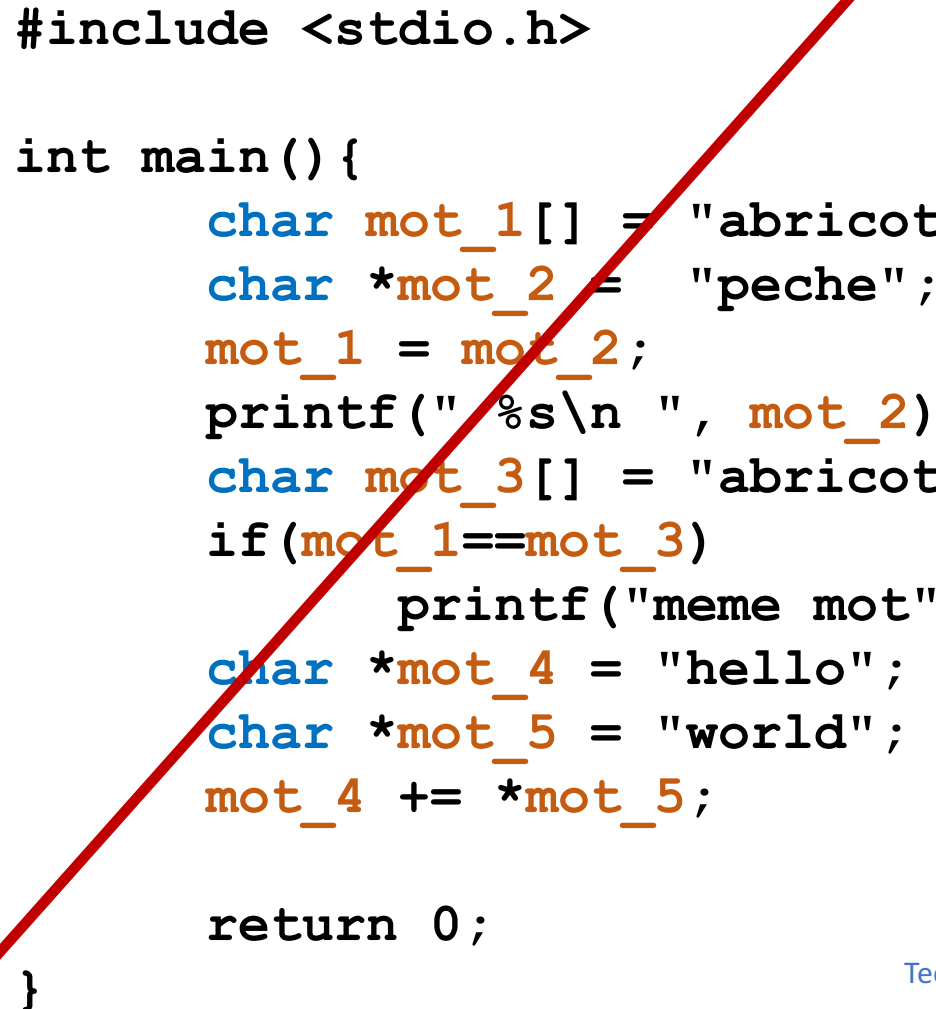
Ne réalise absolument la concaténation d'une chaîne!!

# Les chaînes de caractères : bonnes pratiques

```
#include <stdio.h>

int main(){
    char mot_1[] = "abricot";
    char *mot_2 = "peche";
    mot_1 = mot_2;
    printf("%s\n ", mot_2);
    char mot_3[] = "abricot";
    if(mot_1==mot_3)
        printf("meme mot");
    char *mot_4 = "hello";
    char *mot_5 = "world";
    mot_4 += *mot_5;

    return 0;
}
```



```
#include <stdio.h>
#include <string.h>
#define N 10
int main(){
    char mot_1[N] = "abricot";
    char mot_2[N] = "peche";
    strncpy(mot_1,mot_2,N);

    char mot_3[N] = "abricot";
    if(strncmp(mot_1,mot_3,N)==0)
        printf("meme mot");

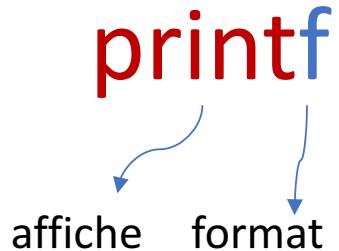
    char mot_4[N] = "hello";
    char mot_5[N] = "world";
    strcat(mot_4, mot_5, N);

    return 0;
}
```

OK

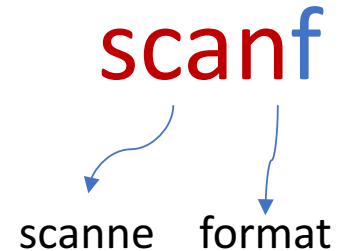
# Les fonctions *printf* et *scanf*

**printf**



affiche    format

**scanf**



scanne    format

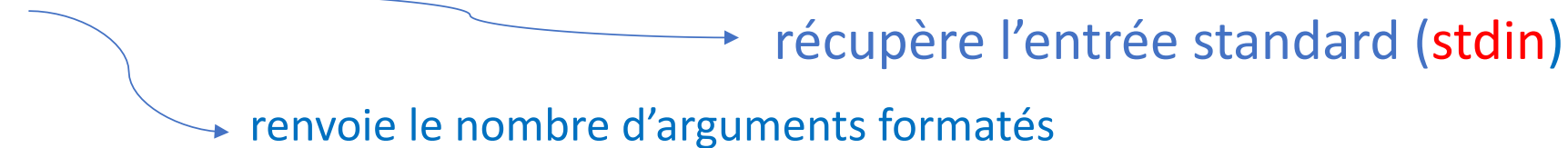
```
int a = printf(<format>, variables ...)
```



affiche sur la sortie standard (**stdout**)

renvoie le nombre de caractères formatés

```
int a = scanf(<format>, variables ...)
```



récupère l'entrée standard (**stdin**)

renvoie le nombre d'arguments formatés



# Les fonctions *printf* et *scanf*

```
int a = 12;
printf("%d", a);      //affiche: 12

int b = 0xA AFF4E3D;
printf("%x", b);      //affiche: aaff4e3d

char c = 'e';
printf("%c", c);      //affiche: e

char mot [] = "bonjour à tous";
printf("%s", a);      //affiche: bonjour à tous

float x = 1.25;
printf("%f", x);      //affiche: 1.250000
printf("%1.3f", x);   //affiche: 1.250
```

# Les fonctions *printf* et *scanf*

```
int a;  
scanf("%d", &a);           //lit un entier au clavier  
  
char c;  
scanf("%c", &c);           //lit un caractère au clavier  
  
char mot [10];  
scanf("%s", mot);          //lit un chaine de caractères  
                             au clavier. Pas de & !
```