

# Programmation C

## Première année

Dr Nelson **SAHO**

Institut de Formation et de Recherche en Informatique

2 février 2025



## Contenu

**1** Les bases de la programmation en C

**2** Les instructions de contrôle

**3** Les types composés



## Bref historique sur le langage C

- Le langage C a été inventé au cours de l'année 1972. Il était développé en même temps que UNIX par Dennis Ritchie et Ken Thompson. Ken Thompson avait développé un prédecesseur de C, le langage B, qui est lui-même inspiré de BCPL.
- Par la suite, Brian Kernighan aida à populariser le langage C. Il procéda aussi à quelques modifications de dernière minute.
- En 1978, Kernighan fut le principal auteur du livre The C Programming Language décrivant le langage enfin stabilisé. Ritchie s'était occupé des appendices et des exemples avec Unix. On appelle aussi ce livre « le K&R », et l'on parle de C traditionnel ou de C K&R.



## La compilation

- Le langage C est un langage compilé (par opposition aux langages interprétés).
  - Un programme C est décrit par un fichier texte appelé *fichier source*. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine.
  - Cette traduction est effectuée par un programme appelé *compilateur*.



## La compilation : Les phases de la compilation

- **Le traitement par le pré-processeur** : le fichier source est analysé par le pré-processeur qui effectue des transformations purement textuelles (remplacement des chaînes de caractères, inclusion d'autres fichiers source, etc.).
  
- **La compilation** : Le fichier généré par le pré-processeur est traduit en assembleur, c'est-à-dire une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.



## La compilation : Les phases de la compilation

- **L'assemblage** : Cette opération transforme le code assembleur en un fichier binaire. Généralement la compilation et l'assemblage se font dans la foulée. Le fichier produit est appelé *fichier objet*.
- **L'édition des liens** : Un programme est souvent séparé en plusieurs fichiers source. L'édition des liens produit alors un fichier dit *exécutable*



## Les phases de la compilation : Application

- Les différents types de fichier utilisés lors de la compilation sont distingués par leur suffixe : source (.c), pré-processeur (.i), assembleur (.s) et les objets (.o). Les .a sont les objets des librairies pré-compilés.
- Le compilateur de C sous UNIX est le *gcc*.
- *gcc [options] fichier.c [-l librairies]*
- Par défaut le fichier exécutable est *a.out* modifiable avec l'option *-o*.



## Les composants élémentaires du C

## Les composants élémentaires du C

- Les identificateurs
- Les mots-clefs
- Les constantes
- Les chaînes de caractères
- Les opérateurs
- Les signes de ponctuation
- Les commentaires mais sont enlevés par le pré-processeur



## Les composants élémentaires du C

## Les composants élémentaires du C : Les identificateurs

- Le rôle d'un identificateur est de donner un nom à une entité du programme. C'est-à-dire il peut désigner :
  - un nom de variable ou de fonction
  - un type défini par *typedef struct*, *union* ou *enum*
  - une étiquette
- Un identificateur est une suite de caractère parmi :
  - les lettres (minuscules ou majuscules mais non accentuées)
  - les chiffres
  - le blanc souligné (\_)



## Les composants élémentaires du C

## Les composants élémentaires du C : Les identificateurs

- Le premier caractère d'un identificateur ne peut pas être un chiffre.
- **Exercice :** Quels sont les identificateurs valides parmi ceux les suivants : 1i; var1; nom2t; i :j
- Il est déconseillé de commencer un nom de variable par \_ car employé pour définir les variables globales de l'environnement C



## Les composants élémentaires du C : Les mots-clefs

- Les spécificateurs de stockage : *auto register static extern typedef*
- Les spécificateurs de type : *char double enum float int long short signed struct union unsigned void*
- Les qualificateurs de type : *const volatile*
- Les instructions de contrôle : *break case continue default do else for goto if switch while*
- Autres : *return sizeof*



## Les composants élémentaires du C : Les commentaires

- Un commentaire multiligne débute par /\* et se termine par \*/ et une seule ligne est mise en commentaire avec //
- On ne peut pas imbriquer des commentaires. Quand on met en commentaire un morceau de programme, il faut veiller à ce que celui-ci ne contienne pas de commentaire
- Aussi, la lecture d'un programme devra-t-elle être facile en faisant une bonne indentation.



## Structure d'un programme C

- Une expression est une suite de composants syntaxiquement correcte, par exemple  $x = 0$  ou bien  
 $(i \geq 0) \&\& (i < 0) \&\& (p[i]! = 0)$
- Une instruction est une expression suivie d'un point-virgule. Il signifie en quelque sorte évaluer l'expression.
- Plusieurs instructions peuvent être rassemblées par des accolades { et } pour former un bloc. Par exemple :

```
if (x != 0) {  
    z = y / x;  
    t = y % x;  
}
```



## Structure d'un programme C (suite)

- Une instruction composée d'un identificateur de type et d'une liste d'identificateur séparés par une virgule est une déclaration.

```
int a;  
int b = 1, c;  
double x = 2.38e4;  
char message[80];
```

- En C toute variable doit faire l'objet d'une déclaration avant son utilisation



## Structure d'un programme C (suite)

- Un programme C se présente sous la forme

- [directives au pré-processeur]
- [déclarations de variables externes]
- [fonctions secondaires]

```
main(){  
    déclarations de variables internes  
    instructions  
}
```



## Structure d'un programme C (suite)

- La fonction principale main peut avoir des paramètres formels
- Les fonctions secondaires peuvent être placées avant ou après la fonction principale. Elles sont de la forme :

```
type ma_fonction (arguments){  
    déclarations de variables internes  
    instructions  
}
```



## Structure d'un programme C (suite)

- Cette fonction retournera un objet dont le type sera *type* à l'aide de l'instruction *return objet;*. Les arguments de la fonction obéissent à une syntaxe voisine de celle des déclarations : *type objet* et séparés par des virgules. Voici un exemple :

```
int produit (int a, int b) {  
    int resultat;  
    resultat = a * b;  
    return (resultat);  
}
```



## Les types prédéfinis

- C est un langage *typé* implique que toutes variable, constante ou fonction est d'un type précis. Le type d'un objet définit la façon dont il est présenté en mémoire.
- Les types de base en C sont les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants :
  - *char*
  - *int*
  - *float double*
  - *short long unsigned*



## Les types prédéfinis : Le type caractère

- *char* est le mot-clé pour le désigner.
- un *char* peut contenir n'importe quel élément du jeu de caractères de la machine utilisée.
- Représenté généralement sur un octet.



## Les types prédefinis : Le type caractère (suite)

- Une des particularité du type *char* en C est qu'il peut être assimilé à un entier.

```
main() {  
    char c = 'A';  
    printf("%c", c + 1);  
}
```

- Le type *char* est signé ou non : *unsigned char* ou *signed char*



## Les types prédefinis

## Le type caractère : Caractères imprimables du code ASCII

	déc.	oct.	hex.		déc.	oct.	hex.		déc.	oct.	hex.
	32	40	20	@	64	100	40	'	96	140	60
!	33	41	21	A	65	101	41	a	97	141	61
"	34	42	22	B	66	102	42	b	98	142	62
#	35	43	23	C	67	103	43	c	99	143	63
\$	36	44	24	D	68	104	44	d	100	144	64
%	37	45	25	E	69	105	45	e	101	145	65
&	38	46	26	F	70	106	46	f	102	146	66
'	39	47	27	G	71	107	47	g	103	147	67
(	40	50	28	H	72	110	48	h	104	150	68
)	41	51	29	I	73	111	49	i	105	151	69
*	42	52	2a	J	74	112	4a	j	106	152	6a
+	43	53	2b	K	75	113	4b	k	107	153	6b
,	44	54	2c	L	76	114	4c	l	108	154	6c
-	45	55	2d	M	77	115	4d	m	109	155	6d
.	46	56	2e	N	78	116	4e	n	110	156	6e



## Les types prédefinis

## Le type caractère : Caractères imprimables du code ASCII

/	47	57	2f	O	79	117	4f	o	111	157	6f
0	48	60	30	P	80	120	50	p	112	160	70
1	49	61	31	Q	81	121	51	q	113	161	71
2	50	62	32	R	82	122	52	r	114	162	72
3	51	63	33	S	83	123	53	s	115	163	73
4	52	64	34	T	84	124	54	t	116	164	74
5	53	65	35	U	85	125	55	u	117	165	75
6	54	66	36	V	86	126	56	v	118	166	76
7	55	67	37	W	87	127	57	w	119	167	77
8	56	70	38	X	88	130	58	x	120	170	78
9	57	71	39	Y	89	131	59	y	121	171	79
:	58	72	3a	Z	90	132	5a	z	122	172	7a
;	59	73	3b	[	91	133	5b	{	123	173	7b
<	60	74	3c	\	92	134	5c		124	174	7c
=	61	75	3d	]	93	135	5d	}	125	175	7d
>	62	76	3e	~	94	136	5e	~	126	176	7e
?	63	77	3f	-	95	137	5f	DEL	127	177	7f



## Les types prédefinis : Les types entiers

- *int* est le mot-clef pour le désigner.
- Représenté généralement sur un mot mémoire (32 bits).

	1	2	
char	8 bits	8 bits	caractère
short	16 bits	16 bits	entier court
int	32 bits	32 bits	entier
long	64 bits	32 bits	entier long
long long	N/A	64 bits	entier long (non ANSI)

1 = DEC Alpha et 2 = PC Intel (Linux)



## Les types prédefinis : Les types entiers (suite)

### ■ Les intervalles de représentation

- signed char  $[-2^7; 2^8 - 1]$
- unsigned char  $[0; 2^8 - 1]$
- short int  $[-2^{15}; 2^{15} - 1]$
- unsigned short int  $[0; 2^{16} - 1]$
- int  $[-2^{31}; 2^{31} - 1]$
- unsigned int  $[0; 2^{32} - 1]$
- long int (DEC alpha)  $[-2^{63}; 2^{63} - 1]$
- unsigned long int (DEC alpha)  $[0; 2^{64} - 1]$



## Les types prédefinis : Les types entiers (suite)

- Plus généralement, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard *limits.h*
- Le mot-clef *sizeof* a pour syntaxe *sizeof(expression)* où expression est un type ou un objet. Le résultat est un entier égal au nombre d'octets nécessaires pour stocker le type ou l'objet.



## Les types prédefinis : Les types entiers (suite)

- Par exemple :

```
unsigned short x;  
int taille = sizeof(unsigned short);  
taille = sizeof(x);
```

- Pour obtenir des programmes portables, on s'efforcera de ne jamais présumer de la taille d'un objet de type entier. On utilisera toujours une des constantes de *limits.h* ou le résultat obtenu en appliquant l'opérateur *sizeof*.



## Les types prédefinis : Les types flottants

- Les types *float*, *double* et *long double* servent à représenter les nombres à virgule flottante.
- On les écrit sous la forme *signe 0, mantisse B<sup>exposant</sup>*.

	1	2	
float	32 bits	32 bits	flottant
double	64 bits	64 bits	flottant double précision
long double	64 bits	128 bits	flottant quadruple précision

1 = DEC Alpha et 2 = PC Intel (Linux)



## Les constantes

- Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont elle est écrite.
- On a 4 types de constantes : entier, flottant, caractère, énumération.
- Elles peuvent servir pour initialiser une variable.



## Les constantes entières

- Elles sont représentées de trois manières :
  - **décimale** : par exemple 0 et 289 sont des constantes décimales
  - **octale** : c'est la représentation en base 8. Elles commencent par un 0. Par exemple les représentations octales des entiers 0 et 255 sont respectivement 00 et 0377
  - **hexadécimale** : c'est la représentation en base 16. Elles commencent par un 0x ou 0X. Par exemple les représentations hexadécimales des entiers 14 et 255 sont respectivement 0xe et 0xff



## Les constantes entières (suite)

- On peut spécifier explicitement le format d'une constante entière en la suffixant par `u` ou `U` pour indiquer qu'elle est non signée, ou en la suffixant par `l` ou `L` pour indiquer qu'elle est de type `long`. Par exemple :

constante	type
1234	int
02322	int /* octal */
0x4d2	int /* hexadécimal */
123456789L	long
1234U	unsigned int
123456789UL	unsigned long int



## Les constantes réelles

- Elles sont représentées par la notation classique par mantisse et exposant. L'exposant est introduit par la lettre e ou E.
- Par défaut les constantes réelles sont représentées sous la format du type *double*. Néanmoins les suffixes f ou F et l ou L pourront être utilisés.

constante	type
12.34	double
$12.3e^{-4}$	double
$12.34F$	float
$12.34L$	long double



## Les constantes caractères

- Pour définir un caractère imprimable, il suffit de le mettre entre apostrophes ('q', '\$').
- Les seuls caractères qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe qui sont respectivement désignés par \\ et \'. Les points d'interrogation et les guillemets aussi par \ ? et \ "
- Les caractères non imprimables peuvent être désignés par \code-octal où code-octal est le code en octal du caractère ou \xcode-hexa où code-hexa est le code en hexadécimal.



## Les constantes caractères (suite)

- Par exemple \33 et \x1b désignent le caractère *escape*.
- Toutefois les caractères non-imprimables les plus fréquents disposent aussi d'une notation simple :

\n	nouvelle ligne	\r	retour chariot
\t	tabulation horizontale	\f	saut de page
\v	tabulation verticale	\a	signal d'alerte
\b	retour arrière		



## Les constantes chaînes de caractères

- Une chaîne de caractères est une suite de caractères entourés par des guillemets. Par exemple "*Ceci est une chaîne de caractère*".
- Une chaîne de caractères peut contenir des caractères non-imprimables, désignés par les représentations vues précédemment. Par exemple "*ligne 1 \n ligne 2*".
- A l'intérieur d'une chaîne de caractères, le caractère " doit être désigné par \".
- Le caractère \ suivi d'un passage à la ligne est ignoré. Cela permet de tenir de longue chaîne de caractères sur plusieurs lignes.



## Les opérateurs : l'affectation

- En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe `=`. Sa syntaxe est la suivante :

*variable = expression*

- Le terme de gauche de l'affectation peut être une variable, un élément de tableau mais pas une constante.
- Elle évalue expression et affecte sa valeur à variable. L'affectation effectue une conversion de type implicite.



## Les opérateurs : l'affectation (suite)

- Quelle est la valeur de x après la compilation du programme ci-dessous ?

```
main() {  
    int i, j = 2;  
    float x = 2.5;  
    i = j + x;  
    x = x + i;  
    printf("\n %f \n", x);  
}
```



## Les opérateurs : les opérateurs arithmétiques

- Les opérateurs arithmétiques classiques sont :

- l'opérateur unaire - (changement de signe)
- + addition
- - soustraction
- \* multiplication
- / division
- % modulo (reste de la division)

- Ces opérateurs agissent aussi bien sur les entiers que sur les flottants.



## Les opérateurs : les opérateurs arithmétiques (suite)

- **Attention :** Si les deux opérandes de l'opérateur / sont entiers cela produira une division entière. Par contre, il délivrera une valeur flottante dès que l'un des opérateurs est un flottant.

```
float x;  
x = 3 / 2; /*Affecte à x la valeur 1*/  
x = 3 /2.; /*Affecte à x la valeur 1.5
```



## Les opérateurs : les opérateurs arithmétiques (suite)

- L'opérateur % ne s'applique qu'à des opérandes de type entier.
- En C, il n'y a pas d'opérateur l'élévation à la puissance. Il faut utiliser la fonction *pow(x,y)* de la librairie *math.h* pour calculer  $x^y$ .



## Les opérateurs : les opérateurs de comparaison

- Les opérateurs de comparaison sont :

- > strictement supérieur
- >= supérieur ou égal
- < strictement inférieur
- <= inférieur ou égal
- == égal
- != différent

- Leur syntaxe est : *expression1 opérateur expression2*



## Les opérateurs : les opérateurs de comparaison (suite)

- Les deux expressions sont évaluées puis comparées. La valeur rendue est de type *int*. Il n'y a pas de type *booléen* en C; elle vaut 1 si la valeur est vraie, et 0 sinon.
- **Attention** : Ne pas confondre l'opérateur de test d'égalité `==` avec l'opérateur d'affectation `=`.



## Les opérateurs : les opérateurs de comparaison (suite)

- Exercice : Qu'avons-nous à l'écran après exécution du programme ci-dessous ?

```
main () {  
    int a = 0;  
    int b = 1;  
    if (a == b)  
        printf("\n a et b sont égaux \n");  
    else  
        printf("\n a et b sont différents \n");  
}
```



## Les opérateurs : les opérateurs logiques booléens

- Les opérateurs logiques booléens sont :

- && et logique
- || ou logique
- ! négation logique

- Leur syntaxe est :

*expression1 opérateur1 expression2 opérateur2...expressionN*



## Les opérateurs : les opérateurs logiques booléens (suite)

- Comme pour les opérateurs de comparaison, la valeur rendue est de type *int*. Elle vaut 1 si la valeur est vraie, et 0 sinon.  
L'évaluation se fait de la gauche vers la droite.
- Exemple

```
int i;  
int p[10];  
if ((i >= 0) && (i <= 9) && !(p[i] == 0))  
/* La dernière clause ne sera pas évaluée  
si i n'est pas compris entre 0 et 9 */  
.....
```



## Les opérateurs : les opérateurs d'affectation composée

- Les opérateurs d'affectation composée sont :  
 $+ =$ ,  $- =$ ,  $* =$ ,  $/ =$ ,  $\% =$
- Pour tout opérateur  $op$ , l'expression  
 $expression1 \ op \ = \ expression2$  est équivalente à :  
 $expression1 = expression1 \ op \ expression2$
- Toutefois, avec l'affectation composée,  $expression1$  n'est évaluée qu'une seule fois.



## Les opérateurs : les opérations d'incrémentation et de décrémentation

- Les opérateurs d'incrémentation `++` et de décrémentation `--` s'utilise aussi bien en suffixe (`i ++`) qu'en préfixe (`++ i`). Dans les deux cas, la variable `i` sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de `i` alors que dans la notation préfixe se sera la nouvelle valeur.

```
int a = 3, b, c;  
b = ++a; /* a et b valent 4 */  
c = b++; /* c vaut 4 et b vaut 5 */
```



## Les opérateurs : l'opérateur virgule

- Une expression peut être constituée d'une suite d'expressions séparées par des virgules :

*expression<sub>1</sub>, expression<sub>2</sub>, ..., expression<sub>N</sub>*

Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de l'expression de droite. Par exemple :

```
main() {  
    int a, b;  
    b = (( a = 3),(a + 2));  
    printf("\n b = %d \n", b); /* Imprime b = 5 */  
}
```



## Les opérateurs : l'opérateur virgule (suite)

- La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule.
- **Exercice :** Quel est l'affichage de l'écran à l'exécution du programme ci-après :

```
main() {  
    int a = 3;  
    printf("\n %d %d \n", ++a, a);  
}
```



## Les opérateurs : l'opérateur conditionnel ternaire

- L'opérateur conditionnel ? est un opérateur ternaire. Sa syntaxe est la suivante :

*condition ? expression1 : expression2*

Cette expression est égale à *expression1* si *condition* est satisfaite, et à *expression2* sinon. Par exemple

`x >= 0 ? x : -x;`

correspond à la valeur absolue de *x*. De même l'instruction

`m = ((a > b) ? a : b);`

affecte à *m* le maximum de *a* et *b*.



## Les opérateurs : l'opérateur de conversion de type

- L'opérateur de conversion de type, appelé *cast*, permet de modifier explicitement le type d'un objet. On écrit :

*(type) objet*

. Par exemple :

```
main () {  
    int i = 3, j = 2;  
    printf("%f \n", (float)i/j);  
}
```

Retourne la valeur 1.5



## Les opérateurs : l'opérateur adresse

- L'opérateur d'adresse & appliqué à une variable retourne l'adresse mémoire de cette variable.
- La syntaxe est la suivante :

*&objet*



## Règles de priorité des opérateurs

- Le tableau ci-après classe les opérateurs par ordre de priorité décroissante.
- Les opérateurs placés sur la même ligne ont la même priorité.
- Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d'évaluation est définie par la flèche de la seconde colonne du tableau.
- Toutefois, il est préférable de mettre des parenthèses en cas de doute.



## Règles de priorité des opérateurs : Le tableau

Opérateurs	
( ) [ ] - > .	→
! ~ ++ -(unaire) (type) *(indirection) &(adresse) sizeof	←
* / %	→
+ -(binaire)	→
<< >>	→



## Règles de priorité des opérateurs : Le tableau (suite)

Opérateurs	
<   <=   >   >=	→
==   !=	→
&(et bit-à-bit)	→
^	→
	→



## Règles de priorité des opérateurs : Le tableau (suite)

Opérateurs	
&&	→
	→
? :	←
= + = - = * = / = % = & = ^ =   = <<= >>=	←
,	→



Merci pour votre attention.  
Commentaires ? Questions ?





## Les instructions de branchements conditionnelles

## Branchement conditionnel if else

- La forme la plus générale est celle-ci :

```
if (expression-1)
    instruction-1
else if (expression-2)
    instruction-2
...
else if (expression-n)
    instruction-n
else
    instruction-m
```



## Les instructions de branchements conditionnelles

## Branchement conditionnel if else (suite)

- La forme la plus simple est :

```
if (expression)
    instruction
```

- Chaque instruction peut être un bloc d'instruction. Les blocs d'instructions restent entre { et }.



## Les instructions de branchements conditionnelles

## Branchement multiple switch

- Sa forme la plus générale est celle-ci :

```
switch (expression-1) {  
    case constante-1:  
        Liste d'instructions 1  
        break;  
  
    ...  
    case constante-n:  
        Liste d'instructions n  
        break;  
    default:  
        Liste d'instructions n+1  
        break;  
}
```



## Les instructions de branchements conditionnelles

## Branchement multiple switch (suite)

- Si la valeur de *expression* est égale à l'une des constantes, la *liste d'instructions* correspondante est exécutée. Sinon la *liste d'instructions n+1* correspondant à *default* est exécutée.  
L'instruction *default* est facultative.



## La boucle *while*

- La syntaxe de la boucle *while* est la suivante :

```
while (expression)
    instruction
```

- Tant que *expression* est vérifiée (i.e non nulle), *instruction* est exécutée. Si *expression* est nulle au départ, *instruction* ne sera jamais exécutée.
- *instruction* peut être un bloc d'instruction.



## La boucle *while* (suite)

- Le programme suivant imprime les entiers de 1 à 9.

```
int i = 1;
while (i < 10) {
    printf("\n i = %d", i);
    i++;
}
```



## La boucle *do — while*

- Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle *do — while*. Sa syntaxe est :

```
do  
    instruction  
    while (expression);
```

- Ici, *instruction* sera exécutée tant que *expression* est non nulle. Cela signifie que *instruction* est exécutée au moins une fois.



## La boucle *do — while* (suite)

- Par exemple pour saisir au clavier un entier entre 1 et 10

```
int a;
do {
    printf("\n Entrer un entier entre 1 et 10 : ");
    scanf("%d", &a);
} while ((a <= 0) || (a > 10));
```



## La boucle *for*

- La syntaxe de la boucle *for* est la suivante :

```
for (expression1; expression2; expression3)  
    instruction
```

- Une version équivalente plus intuitive est :

```
expression1;  
while (expression2) {  
    instruction  
    expression3;  
}
```



## La boucle *for* (suite)

- Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

```
for (i = 0; i < 10; i++)  
    printf("\n i = %d", i);
```

- A la fin de la boucle, *i* vaudra 10.
- Les trois expressions utilisées dans la boucle *for* peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois.



## La boucle *for* (suite)

- **Exercice :** Ecrire un programme C qui calcule la factorielle d'un entier.
- **Solution :**

```
int main() {  
    int n, i, fact;  
    for (i = 1, fact = 1; i <= n; i++)  
        fact *= i;  
    printf("%d ! = %d \n", n, fact);  
    return 0;  
}
```



## La boucle *for* (suite)

- On peut également insérer l'instruction `fact *= i;` dans la boucle *for*. Ce qui donne :

```
int n, i, fact;  
for (i = 1, fact = 1; i <= n; fact *= i, i++);  
printf("%d != %d \n", n, fact);
```

- On évitera toutefois ce type d'acrobacies qui n'apportent rien et rendent le programme difficilement lisible.



## Les instructions de branchements non conditionnels

## Branchement non conditionnels break

- On a vu le rôle de l'instruction break au sein de l'instruction de branchement multiple switch. Il peut être employé à l'intérieur de n'importe quelle boucle.
- Elle interrompt le déroulement de la boucle et passe à la première instruction qui suit la boucle.
- En cas de boucles imbriquées, *break* fait sortir de la boucle la plus interne.



## Les instructions de branchements non conditionnels

## Branchement non conditionnels break (suite)

```
main () {  
    int i;  
    for (i = 0; i < 5; i++) {  
        printf("i = %d \n", i);  
        if (i == 3)  
            break;  
    }  
    printf("Valeur de i a la sortie de la  
boucle = %d \n", i);  
}
```



## Les instructions de branchements non conditionnels

## Branchement non conditionnels break (suite)

- Ce programme imprime à l'écran :

```
i = 0
```

```
i = 1
```

```
i = 2
```

```
i = 3
```

Valeur de i à la sortie de la boucle = 3



## Les instructions de branchements non conditionnels

## Branchement non conditionnels continue

- L'instruction *continue* permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle.

```
main () {  
    int i;  
    for (i = 0; i < 5; i++) {  
        if (i == 3)  
            continue;  
        printf("i = %d \n", i);  
    }  
    printf("Valeur de i a la sortie de la  
boucle = %d \n", i);  
}
```



## Les instructions de branchements non conditionnels

## Branchement non conditionnels continue (suite)

- Ce programme imprime à l'écran :

```
i = 0
```

```
i = 1
```

```
i = 2
```

```
i = 4
```

Valeur de i à la sortie de la boucle = 5



## Les fonctions d'entrées sorties classiques

- Il s'agit des fonctions de la librairie standard *stdio.h* utilisées avec les unitées classiques d'entrées sorties qui sont respectivement le clavier et l'écran.
- L'appel d'une librairie par la directive au pré-processeur se fait de la manière suivante :

```
#include <nom-librairie>
```
- La fonction *printf* permet d'écrire à l'écran et la fonction *scanf* permet de faire de lecture. Avant de les utiliser appeler en début la librairie *stdio.h*.

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



## La fonction d'écriture *printf*

- La fonction *printf* est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est :

```
printf("chaîne de contrôle", expression1,  
       expression2, ..., expressionN);
```

- La *chaîne de contrôle* contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications ont pour but d'annoncer le format des données à visualiser.



## La fonction d'écriture *printf* (suite)

- Elles sont introduites par le caractère %, suivi d'un caractère désignant le format d'impression.
- En plus du caractère donnant le type de données, on peut éventuellement préciser certains paramètres du format d'impression, qui sont spécifiés entre le % et le caractère de conversion dans l'ordre suivant :
  - largeur minimale du champ d'impression : %10d spécifie que 10 caractères seront réservés pour imprimer l'entier. La donnée sera cadrée à droite du champ. Précédé le chiffre du signe - pour le cadrer à gauche(%-10d).



## La fonction d'écriture *printf* (suite)

- précision : %.12f signifie qu'un flottant sera imprimé avec 12 chiffres après la virgule. %10.2f signifie que l'on réserve 12 caractères (incluant le caractère .) pour imprimer le flottant et que 2 d'entre eux seront destinés aux chiffres après la virgule. Par défaut c'est 6 chiffre après la virgule. %30.4s signifie que l'on réserve un champ de 30 caractères pour imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés ( suivis de 26 blancs).



## Format d'impression de la fonction *printf*

format	conversion en	écriture
%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non ignée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée



## Les fonctions d'entrées sorties classiques

Format d'impression de la fonction *printf* (suite)

%f	double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %f et %e
%c	unsigned char	caractère
%s	char*	chaîne de caractères



## Les fonctions d'entrées sorties classiques

Exemple d'utilisation de *printf*

```
main() {  
    int i = 23674;  
    int j = -23674;  
    long int k = (11 << 32);  
    double x = 1e-8 + 1000;  
    char c = 'A';  
    char *chaine = "chaine de caracteres";  
  
    printf("impression de i: \n");  
    printf("%d \t %u \t %o \t %x",i,i,i,i);  
    printf("\nimpression de j: \n");  
    printf("%d \t %u \t %o \t %x",j,j,j,j);  
}
```



## Les fonctions d'entrées sorties classiques

Exemple d'utilisation de *printf* (suite)

```
printf("\nImpression de k: \n");
printf("%d \t %o \t %x",k,k,k);
printf("\n%ld \t %lu \t %lo \t %lx",k,k,k,k);
printf("\nImpression de x: \n");
printf("%f \t %e \t %g",x,x,x);
printf("\n%.2f \t %.2e",x,x);
printf("\n%.20f \t %.20e",x,x);
printf("\nImpression de c: \n");
printf("%c \t %d",c,c);
printf("\nImpression de chaine: \n");
printf("%s \t %.10s",chaine,chaine);
}
```



## Les fonctions d'entrées sorties classiques

Exemple d'utilisation de *printf* (suite)

Ce programme imprime à l'écran

impression de i :

23674 23674 56172 5c7a

nimpression de j :

-23674 4294943622 37777721606 fffffa386

impression de k :

0 0 0

4294967296 4294967296 40000000000 1000000000



## Les fonctions d'entrées sorties classiques

Exemple d'utilisation de *printf* (suite)

Ce programme imprime à l'écran

impression de x :

1000.000000 1.000000e+03 1000

1000.00 1.00e+03

1000.0000001000000000000000 1.0000000100000000000e+03

impression de c :

A 65

impression de chaine :

chaine de caracteres chaine de



## La fonction de lecture ou de saisie *scanf*

- La fonction *scanf* permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonction.

```
scanf("chaîne de contrôle", argument1,  
      argument2, ..., argumentN);
```

- La *chaîne de contrôle* indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (pas de \n).



## La fonction de lecture ou de saisie *scanf*

- Comme pour *printf* les spécifications introduites par le caractère %, suivi d'un caractère désignant le format d'impression. Les formats valides pour la fonction *scanf* diffèrent légèrement de ceux de la fonction *printf*.
- Les données à entrer doivent être séparées par des blancs ou des <*RETURN*> sauf s'il s'agit de caractères. On peut toujours fixer le nombre de caractères de la donnée à lire. %3s pour une chaîne de 3 caractères, %10d pour un entier qui s'étend sur 10 chiffres signe inclus.



## Les fonctions d'entrées sorties classiques

Format d'impression de la fonction *scanf* (suite)

Format	Type d'objet pointé en	Représentation de la saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale



## Les fonctions d'entrées sorties classiques

Format d'impression de la fonction *scanf* (suite)

Format	Type d'objet pointé en	Représentation de la saisie
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle



## Les fonctions d'entrées sorties classiques

Format d'impression de la fonction *scanf* (suite)

Format	Type d'objet pointé en	Représentation de la saisie
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères



## Les fonctions d'entrées sorties classiques

Format d'impression de la fonction *scanf* (suite)

Format	Type d'objet pointé en	Représentation de la saisie
<code>%[characters]</code>	<code>char*</code>	Un donnée de type chaîne de caractères, constituée que de caractères parmis ceux spécifiés.
<code>%[^characters]</code>	<code>char*</code>	Un donnée de type chaîne de caractères, constituée de tous caractères sauf les caractères spécifiés.



## Les fonctions d'entrées sorties classiques

Exemple d'utilisation de *scanf*

```
#include<stdio.h>
main () {
    int i;
    printf("Entrer un entier sous forme hexadécimale i = ")
    scanf("%x",&i);
    printf("i = %d\n",i);
}
```

Si on entre au clavier la valeur 1a, le programme affichera i = 26.



## Impression et lecture de caractères

- Les fonctions *getchar* et *putchar* permettent respectivement de lire et d'imprimer des caractères.
- La fonction *getchar* retourne un int correspondant au caractère lu. Pour mettre le caractère lu dans une variable *caractere*, on écrit :  

```
caractere = getchar();
```

- Lorsqu'elle détecte la fin de fichier, elle retourne l'entier *EOF* (*End Of File*), valeur définie dans la librairie *stdio.h*. Elle vaut en général  $-1$ .



## Impression et lecture de caractères (suite)

- La fonction *putchar* écrit *caractere* sur la sortie standard :

```
putchar(caractere);
```

- Elle retourne un int correspondant à l'entier lu ou à la constante *EOF* en cas d'erreur.
- Par exemple, le programme suivant lit un fichier et le recopie caractère par caractère à l'écran.

```
#include <stdio.h>
main () {
    char c
    while ((c = getchar()) != EOF)
        putchar(c);
}
```



## Impression et lecture de caractères (suite)

- Pour l'exécuter, il suffit d'utiliser l'opérateur de redirection d'Unix : *programme-executable < nom-fichier*
- L'expression ( $c = getchar()$ ) a pour valeur la valeur de l'expression *getchar()* qui est de type *int*. Le test ( $c = getchar() \neq EOF$ ) compare donc bien deux objets de type *int* (signés).



## Les fonctions d'entrées sorties classiques

## Impression et lecture de caractères (suite)

- Ce n'est pas le cas dans le programme suivant :

```
#include<stdio.h>
main () {
    char c;
    do {
        c = getchar();
        if (c != EOF)
            putchar(c);
    } while ( c != EOF);
}
```



## Impression et lecture de caractères (suite)

- Ici, le test `c != EOF` compare un objet de type `char` et la constante `EOF` qui vaut `-1`. Si le type `char` est non signé par défaut, cette condition est donc toujours vérifiée. Si le type `char` est signé, alors le caractère de code 255 (`\u00ff`) sera converti en entier `-1`. La rencontre du caractère `\u00ff` sera donc interprétée comme la fin du fichier. Il est donc recommandé de déclarer une variable de type `int` et non `char`, variable destiné à recevoir un caractère lu par `getchar` afin de permettre la détection de fin de fichier.



## Les fonctions d'entrées sorties classiques

## Les conventions d'écriture d'un programme C

- On n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère.
- Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne.



## Les conventions d'écriture d'un programme C

- On laisse un blanc
  - entre les mots-clefs *if*, *while*, *do*, *switch* et la parenthèse ouvrante qui suit ;
  - après une virgule ;
  - de part et d'autre d'un opérateur binaire.
- On ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composé.
- Les instructions doivent être indentées afin que toutes les instructions d'un même bloc soient alignées.



Merci pour votre attention.  
Commentaires ? Questions ?





## Les types composés

- A partir des types prédéfinis en C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés *types composés*.
- Ils permettent de représenter un ensemble de données de manière organisée.



## Les tableaux

- Un tableau est un ensemble d'éléments de même type, stockés en mémoire à des adresses contigües.
- La déclaration d'un tableau à une dimension se fait de la façon suivante :

```
type nom-du-tableau[nombre-elements];
```

où *nombre-elements* est une expression constante entière positive. Par exemple, la déclaration `int tab[10];` indique que *tab* est un tableau de 10 éléments de type *int*. Cette déclaration alloue donc en mémoire pour l'objet *tab* un espace de  $10 * 4$  octets consécutifs.



## Les tableaux (suite)

- Pour plus de clarté, il est recommandé de donner un nom à la constante *nombre-elements* par une directive au pré-processeur, par exemple :

```
#define nombre-elements 10
```

- On accède à un élément du tableau en lui appliquant l'opérateur `[]`. Les éléments du tableau sont toujours numérotés de 0 à *nombre-elements* - 1.



## Les tableaux (suite)

- Le programme suivant imprime les éléments du tableau *tab* :

```
#define N 10
main () {
    int tab[N];
    int i;
    ...
    for(i = 0; i < N; i++)
        printf("tab[%d] = %d\n", i, tab[i]);
}
```



## Les tableaux (suite)

- Un tableau correspond en fait à un pointeur constant vers le premier élément du tableau. Aucune opération globale n'est autorisée. Un tableau ne peut pas figurer à gauche d'un opérateur d'affectation. On ne peut pas écrire `tab1 = tab2;`

```
#define N 10
main () {
    int tab1[N], tab2[N];
    int i;
    ...
    for(i = 0; i < N; i++)
        tab1[i] = tab2[i];
}
```



## Les tableaux (suite)

- On peut initialiser un tableau lors de sa déclaration par une liste de constantes de la façon suivante :

```
type nom-du-tableau[N] = {constante1,  
constante2, ..., constanteN};
```

Par exemple, on peut écrire

```
#define N 4  
int tab[N] = {1, 2, 5, 10};  
main () {  
    int i;  
    for(i = 0; i < N; i++)  
        printf("tab[%d] = %d\n", i, tab[i]);  
}
```



## Les tableaux (suite)

- Si le nombre d'éléments dans la liste d'initialisation est inférieur à la dimension du tableau, seuls les premiers éléments seront initialisés. Les autres éléments seront mis à 0 si la tableau est une variable globale ou une variable locale de classe de mémorisation *static*.
- Un tableau de caractère aussi peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères littérale. Notons que le compilateur complète toute chaîne de caractères avec le caractère *nul* (\0). Il faut donc que le tableau ait au moins un éléments de plus que le nombre de caractères de la chaîne littérale.



## Les tableaux (suite)

```
#define N 8
char tab[N] = "exemple";
main () {
    int i;
    for(i = 0; i < N; i++)
        printf("tab[%d] = %c\n", i, tab[i]);
}
```

- Lors d'une initialisation, il est également possible de ne pas spécifier le nombre d'éléments du tableau. Par défaut, il correspondra au nombre de constantes de la liste d'initialisation.



## Les tableaux (suite)

- Ainsi, le programme suivant imprime le nombre de caractères du tableau *tab*, ici 8.

```
char tab[] = "exemple";
main () {
    int i;
    printf("Nombre de caracteres du tabelau
          = %d\n", sizeof(tab)/sizeof(char));
}
```



## Les tableaux (suite)

- De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple un tableau à deux dimensions :

```
type nom-du-tableau[lignes] [colonnes];
```

- En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression *tab[i][j]*. Pour initialiser un tableau à plusieurs dimensions à la compilation, on utilise une liste dont chaque élément est une liste de constantes.



## Les tableaux (suite)

```
#define M 2
#define N 3
int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};
main () {
    int i, j;
    for(i = 0; i < M; i++)
        for(j = 0; j < N; j++)
            printf("tab[%d] [%d] = %d\n", i, j,
                   tab[i][j]);
}
```



## Les structures

- Une structure est une suite d'objets de types différents. Les différents éléments d'une structure n'occupent pas nécessairement des zones contiguës en mémoire. Chaque élément de la structure, appelé *membre* ou *champ*, est désigné par un identificateur.
- On distingue la déclaration d'un *modèle de structure* de celle d'un objet de type structure correspondant à un modèle donné.



## Les structures

- La déclaration d'un modèle de structure dont l'identificateur est *modele* suit la syntaxe suivante :

```
struct modele {  
    type1 champ1;  
    type2 champ2;  
    ...  
    typeN champN;  
};
```



## Les structures (suite)

- Pour déclarer un objet de type structure correspondant au modèle précédent, on utilise la syntaxe :

```
struct modele objet;
```

ou bien, si le modèle n'a pas été déclaré au préalable :

```
struct modele {  
    type1 champ1;  
    ...  
    typeN champN;  
} objet;
```



## Les structures (suite)

- On accède aux différents champs d'une structure grâce à l'opérateur *membre de structure*, noté “.”. Le  $i^{\text{ème}}$  champ de *objet* est donc désigné par l'expression

`objet.champ $i$`

- On peut effectuer sur le  $i^{\text{ème}}$  champ de la structure toutes les opérations valides sur des données de *type $i$* . Par exemple, le programme suivant définit la structure *complexe*, composé de deux champs de type double; il calcule la norme d'un nombre complexe.



## Les structures (suite)

```
#include <math.h>
#include <stdio.h>
struct complexe {
    double reelle;
    double imaginaire;
};

main () {
    struct complexe z;
    double norme;
    ...
}
```



## Les structures (suite)

```
norme = sqrt(z.reelle * z.reelle +
z.imaginaire * z.imaginaire);
printf("Norme de (%f + %fi) = %f\n",
(z.reelle, z.imaginaire, norme);
}
```

- Les règles d'initialisation d'un objet d'une structure lors de sa déclaration sont les mêmes que pour les tableaux. On écrit par exemple :

```
struct complexe z = {2., 2.};
```



## Les structures (suite)

- En ANSI C, on peut appliquer l'opérateur d'affectation aux structures (à la différence des tableaux). Dans le contexte précédent, on peut écrire :

```
main () {  
    struct complexe z1, z2;  
    ...  
    z2 = z1;  
}
```



## Les champs de bits

- Il est possible en C de spécifier la longueur des champs d'une structure au bit près si ce champ est de type entier (*int* ou *unsigned int*). Cela se fait en précisant le nombre de bits du champ avant le ; qui suit sa déclaration. Par exemple la structure suivante :

```
struct registre {  
    unsigned int actif : 1;  
    unsigned int valeur : 31;  
}
```



Les champs de bits



## Les champs de bits (suite)

- Cette structure possède deux champs : *actif*, codé sur un seul bit, et *valeur*, codé sur 31 bits. Tout objet de type *registre* est donc codé sur 32 bits.
- Le champ *actif* de la structure ne peut prendre que deux valeurs : 0 ou 1. Ainsi, l'instruction *r.actif += 2;* ne modifie pas la valeur du champ.
- La taille d'un champ de bits doit être inférieur au nombre de bits d'un entier. Un champ de bits n'a pas d'adresse ; on ne peut donc lui appliquer l'opérateur &.



## Les unions

- Une *union* désigne un ensemble de variables de types différents susceptibles d'occuper alternativement une même zone mémoire. Une *union* permet donc de définir un objet comme pouvant être de type au choix parmi un ensemble fini de types. Si les membres d'une *union* sont de longueurs différentes, la place réservée en mémoire pour la représenter correspond à la taille du membre le plus grand.
- Les déclarations et les opérations sur les objets de type *union* sont les mêmes que celles sur les objets de type *struct*.



## Les unions (suite)

```
union jour {  
    char lettre;  
    int numero;  
}  
  
main () {  
    union jour hier, demain;  
    hier.lettre = 'J';  
    printf("hier = %c\n",hier.lettre);  
    hier.numero = 4;  
    demain.numero = (hier.numero + 2) % 7;  
    printf("demain = %d\n",demain.numero);  
}
```



## Les unions (suite)

- Dans l'exemple précédent, la variable *hier* de type *union jour* peut être soit un entier, soit un caractère.
- Les unions peuvent être utiles lorsqu'on veut voir un objet sous des types différents .
- On peut les utiliser en combinaison avec des structures.



## Les unions (suite)

```
struct coordonnees {  
    unsigned int x;  
    unsigned int y;  
};  
union point {  
    struct coordonnees coord;  
    unsigned long mot;  
};  
  
main () {  
    union point p1, p2, p3;
```



Les unions



## Les unions (suite)

```
p1.coord.x = 0xf;  
p1.coord.y = 0x1;  
p2.coord.x = 0x8;  
p2.coord.y = 0x8;  
p3.mot = p1.mot ^ p2.mot;  
printf("p3.coord.x = %x \t p3.coord.y = %x\n",  
p3.coord.x, p3.coord.y);  
}
```



## Les énumérations

- Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre. Un objet de type énumérations est défini par le mot clé *enum* et un identificateur de modèle, suivi de la liste des valeurs que peut prendre cet objet.

```
enum modele {constante1, constante2,  
..., constanteN};
```

- En réalité les objets de type *enum* sont représentés comme des *int*. Les valeurs possibles *constante1*, *constante2*, ..., *constanteN* sont codés par des entiers de 0 à  $N - 1$ .



## Les énumérations (suite)

- Par exemple, le type *enum boolean* défini dans le programme suivant associe 0 à la valeur *faux* et l'entier 1 à la valeur *vrai*.

```
main () {  
    enum boolean {faux, vrai};  
    enum boolean b;  
    b = vrai;  
    printf("b = %d\n", b);  
}
```



## Définition de types composés avec *typedef*

- Pour alléger l'écriture des programmes, on peut affecter un nouvel identificateur à un type composé à l'aide de *typedef* :

```
typedef type synonyme;
```

- Par exemple en reprenant notre structure élaborée précédemment *complexe* on peut écrire plus simplement ceci :



## Définition de types composés avec *typedef* (suite)

```
struct complexe {  
    double reelle;  
    double imaginaire;  
};  
typedef struct complexe complexe;  
  
main () {  
    complexe z;  
    ...  
}
```



Merci pour votre attention.  
Commentaires ? Questions ?





Merci pour votre attention.  
Commentaires ? Questions ?

