Programmation C (Remise à niveau)

Vendredi 05 Octobre 2018

Michael FRANÇOIS

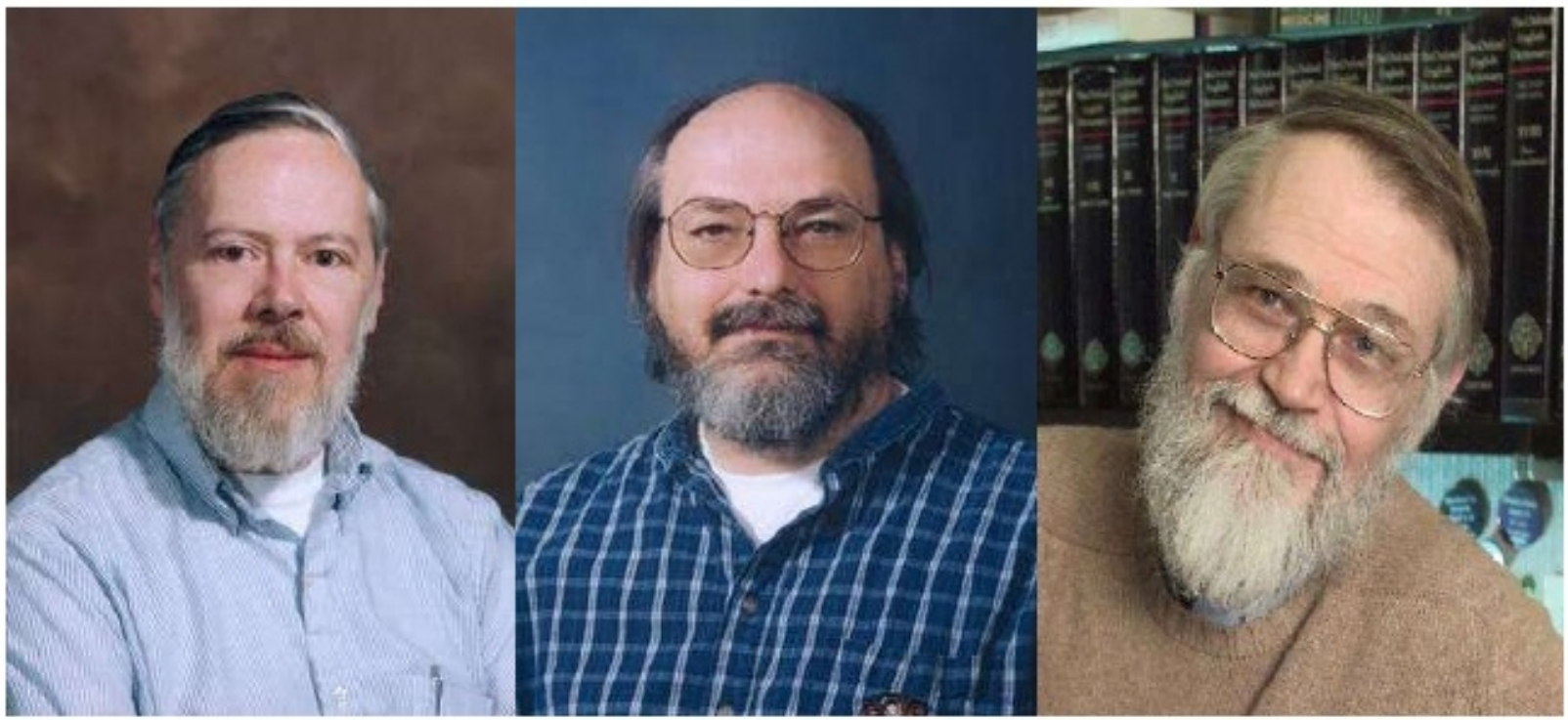
francois@esiea.fr

https://francois.esiea.fr

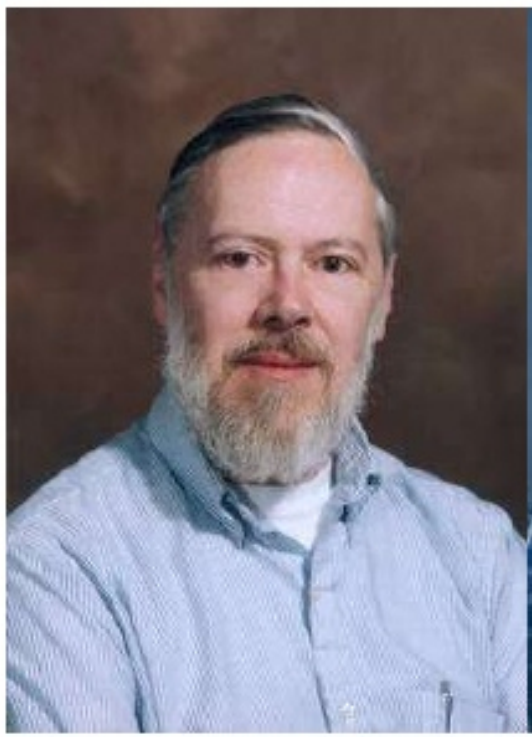
esiea
ECOLE D'INGENIEURS
DU MONDE NUMERIQUE

Historique du langage C

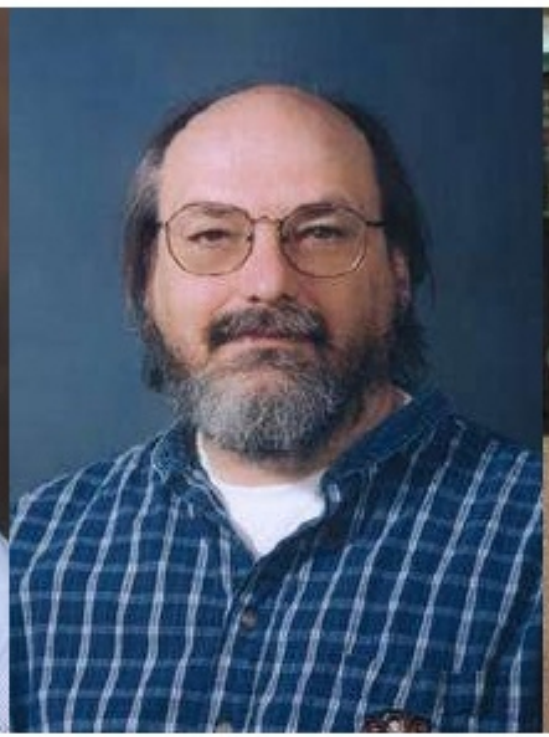
Historique du langage C



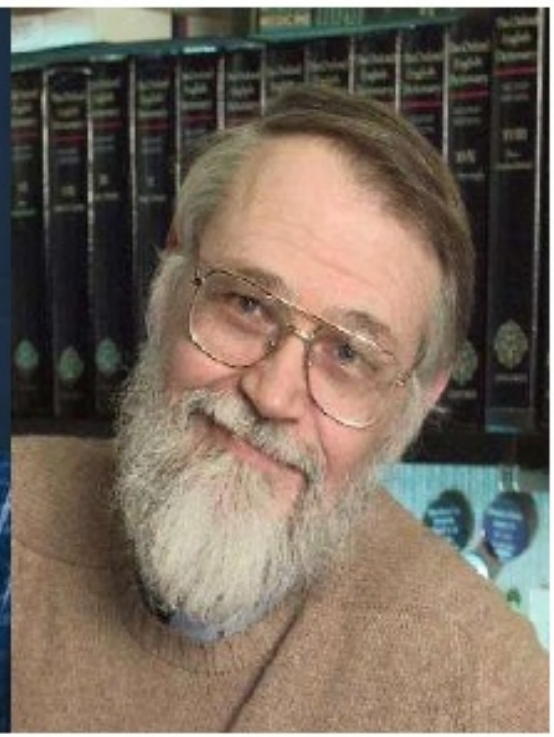
Historique du langage C



Dennis Ritchie



Ken Thompson



Brian Kernighan

Historique du langage C

• Le langage C a été créé en **1972** par Denis Ritchie, dans le but d'écrire un système d'exploitation (UNIX).

• Destiné au départ pour un usage interne des laboratoires Bell, le langage C a été complètement décrit pour la 1ère fois en 1978 dans le livre "*The C programming language*" de B. Kernighan et D. Ritchie.

- Grâce à sa puissance, le langage C devint rapidement très populaire.

• Le succès international de ce langage, a conduit à sa normalisation :

- ANSI (American National Standard Institute) en 1989 $\Rightarrow$ C ANSI
- ISO (International Standardization Organisation) en 1990 $\Rightarrow$ C ISO
- CEN (Comité Européen de Normalisation) en 1993
- AFNOR (Association Française de NORmalisation) en 1994

NB : ces normes sont similaires, et on parle cependant de la norme "ANSI" ou le "C ANSI".

• Le langage C est souvent associé au système UNIX, car ce système est écrit en C ainsi qu'une bonne partie des logiciels qui tournent sous UNIX.

• De nombreux principes fondamentaux du C sont issus du langage `BCPL (Basic Combined Programming Language)`, créé par Martin Richards en 1966.

• Le langage C n'est pas lié à une structure matérielle ou à une machine quelconque :

- il permet donc d'écrire aisément des programmes fonctionnant sur n'importe quelle machine acceptant le langage C.

• C est considéré comme un langage **bas niveau** car, il permet l'accès à des données que manipulent les ordinateurs (bits, octets, adresses, etc.).

Les phases de compilation en C

Les phases de compilation en C

- Un programme C est décrit par un fichier texte appelé fichier source :
  - fichier non exécutable par le microprocesseur, il faut alors le traduire en langage machine (opération effectuée par un compilateur).
- La compilation est la traduction dans le langage de l'ordinateur d'un programme écrit en langage C.

La compilation passe par différentes phases, produisant ou non des fichiers intermédiaires :

1. **Préprocessing :** un préprocesseur réalise plusieurs opérations de substitution sur le code C :

   - suppression des commentaires, inclusion des fichiers .h (#include)
   - traitement des directives de compilation (#define, etc.)

2. **Compilation en langage assembleur :** traduction du fichier source en code assembleur (`.s`), c'est à dire en une suite d'instructions qui sont chacune associées à une fonctionnalité du microprocesseur (addition, comparaison, etc.).
3. **Assemblage :** le code assembleur est transformé en fichier binaire (`.o`), directement compréhensible par le processeur.
4. **Édition des liens :** va réunir le fichier objet et les fonctions contenues dans les bibliothèques, pour produire le fichier exécutable (`.out`) final.

Compilation en C

Exemple : fichier source $\Longrightarrow$ `programme.c`
Compilation en deux commandes (sur le terminal) :

```
gcc -c programme.c   #création du fichier objet programme.o
gcc -o EXEC programme.o      #création de l'exécutable EXEC
./EXEC                              #pour lancer l'exécutable
```

Ou en une seule commande :

```
gcc -o EXEC programme.c      #Création de l'exécutable EXEC
```

```
gcc programme.c                #Création de l'exécutable a.out
```

NB : quelques options de compilation :

- `-Wall` $\Rightarrow$ active tous les warnings possibles
- `-w` $\Rightarrow$ supprime tous les warnings
- `-pedantic` $\Rightarrow$ affiche les warnings requis par la norme ANSI du langage
- `-On` $\Rightarrow$ active les optimisations (n allant de 0 à 3, ou ≪s≫)
- etc.

Généralités

Quelques bibliothèques standards

- `<stdio.h>` : fournit les capacités centrales d'entrée/sortie du langage C, comme la fonction `printf` ou `scanf`.
- `<stdlib.h>` : pour exécuter diverses opérations comme la conversion, la génération de nombres pseudo-aléatoires, l'allocation de mémoire, etc.
- `<math.h>` : pour calculer des fonctions mathématiques courantes.
- `<time.h>` : pour convertir entre différents formats de date et d'heure.
- `<string.h>` : pour manipuler les chaînes de caractères.
- `<complex.h>` : pour manipuler les nombres complexes.

Exemple : `#include <stdio.h>`
permet d'utiliser les fonctions définies dans la librairie standard dentrée/sortie "stdio".

Jeu de caractères source

• Le *jeu de caractères source* désigne l'ensemble des caractères qu'on peut utiliser pour écrire un programme source. La liste est la suivante :

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

plus l'espace, et les caractères dits de "contrôle" car non imprimables :

- la tabulation horizontale
- la tabulation verticale
- le saut de page
- "indication" de fin de ligne

Le code **ASCII**

• **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), est une norme de codage de caractères.

• Le codage **ASCII** a permis l'échange de textes en anglais à un niveau mondial.

• Le codage **ASCII** définit seulement 128 caractères numérotés de 0 (*i.e.* 0000000) à 127 (*i.e.* 1111111) :

- sachant que chaque caractère est quand même stocké en machine sur 1 octet, dont le 8ème bit (de poids fort) est mis à 0.

• L'absence des caractères pour les langues étrangères à l'anglais rend ce standard insuffisant pour les textes étrangers.

ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	**space**	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Le code **ASCII** étendu

• Le besoin d'avoir davantage de caractères s'est fait vite ressentir. Ainsi, 128 caractères supplémentaires ont été rajoutés pour un total de 256 caractères (codage sur 8 bits).

• Cependant, même avec ces caractères supplémentaires de nombreuses langues comportent des symboles impossible à résumer en 256 caractères :

- exemple la norme **ISO 8859-1** (Latin-1 ou Europe occidentale) qui est une variante de l'**ASCII** et qui est utilisée par de nombreux logiciels pour les langues (allemand, catalan, basque, flamand, etc.)

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Remarque : les diverses extensions du code **ASCII**, dont les **ISO 8859**, présentent l'inconvénient d'être incompatibles entre elles, et le plus souvent d'être destinées à un jeu de caractères pour une langue spécifique. Dans ce cas, comment coder dans un même document des textes rédigés avec des alphabets aussi divers : latin, cyrillique, grec, arabe, etc. ?

Table des caractères Unicode

• L'**Unicode** désigne un système de codage utilisé pour l'échange de contenus à l'échelle internationale (dans différentes langues), Ce système a vu le jour dans les années 90.

• Développé par le **Consortium Unicode**, une organisation privée sans but lucratif.

• Ce système contient plus de 137000 caractères.

• Plusieurs codages des caractères Unicode existent :

- **UTF-32** : code chaque caractère sur 32 bits (*i.e.* 4 octets)
- **UTF-16** : code chaque caractère sur 16 ou 32 bits
- **UTF-8** : code chaque caractère sur 8, 16, 24 ou 32 bits

Premier programme

- Un programme est produit à partir d'un fichier source dont un compilateur se sert pour produire un fichier exécutable :
  - fichier source : texte
  - fichier de sortie : binaire

Exemple de premier programme : Hello world !

```
#include <stdio.h>

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

Remarques :
- Instructions délimitées par un ";"
- Parenthèses et accolades (notion de paramètre et de bloc)
- Directive include (pas de ";" à la fin !)
- Définition de la fonction principale via le mot clé `main`
- Retour du programme : `return`
- Utilisation de `printf` (fait partie de *stdio*)

Les variables

C'est à la fois :

- un espace dans la mémoire où de l'information est stockée
- un identifiant (label) dans le code source pour manipuler cette donnée

• Déclaration et initialisation

```
type nom_de_la_variable; //déclaration
nom_de_la_variable = valeur; //affectation ou initialisation
type nom_de_la_variable = valeur; //déclaration + initialisation
```

Exemple :

```
int a;
int b = 0;
char d, ma_variable;
a = 65;
ma_variable = 'r';
```

Identifiants de variables

Règle sur les identifiants (labels) des variables :

- commencent par une lettre
- des caractères ASCII portables (pas de é, à, etc.)
- pas d'espace !
- le " _ " est le bienvenu
- surtout un identifiant parlant (pour une meilleure compréhension du code)

```
-------------------------------------------------
int temperature = 45;
int vitesse_de_l_objet = 0;
char note, alphabet, lettre;
alphabet = 'r';
-------------------------------------------------
```

Les mots clés du langage C

- Le langage C est un langage à mots-clés, ce qui signifie qu'un certain nombre de mots sont réservés pour le langage lui-même.
- Un mot clé ne peut donc pas être employé comme identifiant.
- Liste des mots clés :

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

Nb : si le compilateur produit un message d'erreur syntaxique incompréhensible il est recommandé d'avoir le réflexe de consulter la liste des mots clés pour vérifier que l'on a pas pris comme identificateur un mot-clé.

Les séparateurs et les espaces blancs

• Le compilateur s'appuie sur les espaces blancs pour séparer les mots du langage des variables, sauf lorsqu'un séparateur (, ; { etc.) indique la délimitation. Ainsi :

```
-------------------------------------------------
intx,y; // Impossible à distinguer
int x,y; // ok
int x,y,z; // ok
int x, y, z; // ok et plus visible
-------------------------------------------------
```

Le format libre

- Le langage C autorise une mise en page parfaitement libre :
  - une instruction peut s'étendre sur un nombre quelconque de lignes
  - une même ligne peut comporter autant d'instructions que voulu

Exemples :

```
int x
 ,
   y; // OK
int x
      y; // KO
const char * message = "remise à niveau,
en programmation C"; /* Pas ok, car les fins de ligne ne
                        sont pas autorisées dans les
                        constantes chaîne */
```

Nb : il faut faire attention à ne pas aboutir à des programmes peu lisibles.

Les commentaires

• Comme tout langage évolué, le langage C autorise la présence de commentaires dans les programmes source.

• Il s'agit de textes explicatifs pour une meilleure compréhension du programme. Ils n'ont aucune incidence sur la compilation.

```
-------------------------------------------------
// Commentaire sur une ligne

/* Un autre commentaire sur une ligne */

/*
Commentaires
sur
plusieurs lignes
*/
float valeur; /* valeur à calculer */
-------------------------------------------------
```

Les types de données

- L'information sur la machine est une séquence de 0 et de 1 (information binaire).
- Le type indique :
  - comment traduire la représentation binaire ;
  - la place mémoire que va occuper la valeur ;
  - le type d'encodage utilisé.
- Trois types élémentaires :
  - Entier : `int`
  - Réel : `float, double`
  - Caractère : `char`

Les types de données (tableau récapitulatif) :

Type	Octets	Valeurs
char	1	[-128, +127]
unsigned char	1	[0, 255]
short int	2	[-32 768, +32 767]
int	4	[-2 147 483 648, +2 147 483 647]
long int	4	[-2 147 483 648, +2 147 483 647]
long long int	8	[9 223 372 036 854 775 808, +9 223 372 036 854 775 807]
unsigned short int	2	[0, 65 535]
unsigned int	4	[0, 4 294 967 295]
unsigned long int	4	[0, 4 294 967 295]
unsigned long long int	8	[0, +18 446 744 073 709 551 615]
float (IEEE-754)	4	$\pm 3.4 \cdot 10^{\pm} 38$ ($\sim$ 7 chiffres de sensi.)
double (IEEE-754)	8	$\pm 1.7 \cdot 10^{\pm} 308$ ($\sim$ 15 chiffres de sensi.)

Les opérateurs

- Opérateurs définis :
  - arithmétiques : $+ - * /$ %(modulo)
  - relationnels : $> >= < <= == !=$
  - logiques booléens : && || !
  - affectation : $=$
  - affectation composée : $+=$ -= $/= *=$ %=
  - incrémentation : $++ --$
  - gestion des priorités : ()

Cas des opérateurs logiques OU (||) et ET (&&)

- En C, le OU et ET logique s'évaluent ainsi :
  - expr1 && expr2

    -expr1 est évaluée, si expr1 est fausse, alors retourner FAUX
    -dans le cas où expr1 est vraie alors expr2 est évaluée, et si expr2 est fausse, alors retourner FAUX, sinon retourner VRAI
  - expr1 || expr2

    -si expr1 est VRAI, alors retourner VRAI
    -si expr2 est VRAI, alors retourner VRAI
    Retourner FAUX

NB : attention à ne pas confondre && avec l'opérateur de manipulation de bits (&), ni || avec l'opérateur de manipulation de bits (|).

Présentation générale de `printf`

Présentation générale de printf

• La fonction printf, est une fonction de la bibliothèque qui affiche sur la sortie standard du texte, des valeurs de variables, etc.

• printf ne passe jamais à la ligne automatiquement. Il faut utiliser le symbole '\n' pour ajouter un caractère de fin de ligne dans l'argument de printf.

• Syntaxe : printf("<format>",<Expr1>,<Expr2>, ...);
"<format>" : format de représentation
<Expr1>,... : valeurs des variables ou expressions à représenter

```
-------------------------------------------------
int val = 94200;
char lettre = 'g';
printf("Paris est magique \n");
printf("Valeur = %d\n", val);
printf("lettre = %c\n", lettre);
-------------------------------------------------
```

Exemple :
La lettre qui suit les "%" dans le format correspond à un type de variable.

Type	Lettre
int	%d
long	%ld
float/double	%f / %lf
char	%c
string (char*)	%s
pointeur (void*)	%p
entier hexadécimal	%x

Le gabarit d'affichage

• Chaque code format peut comporter une indication dite de gabarit qui précise un nombre minimal de caractères à afficher. Si cela n'est pas indiqué, `printf` utilise un gabarit par défaut.

• **Le gabarit par défaut** consiste à utiliser exactement le nombre d'emplacements nécessaires pour afficher l'information concernée.

```
printf("%d", x)     /* entier avec gabarit par défaut */
x = 30                 30
x = 2                  2
x = -7352              -7352

printf("%f", y)     /* notation décimale avec gabarit par défaut
                       (6 chiffres après le point) */
y = 2.655              2.655000
y = 32.65578339        32.655783
y = 0.000013265        0.000013
```

Le paramètre de gabarit

• On peut agir sur le gabarit d'affichage en lui imposant une valeur minimale (pas de valeur max. sauf pour les chaînes).

• On peut définir un gabarit minimal en utilisant le paramètre dit de gabarit, placé après le caractère % et avant le caractère de conversion.

Exemples :

```
printf("%4d", x)    /* entier avec 4 caractères minimum */
```

x = 30	__30
x = 2	___2
x = -7352	-7352

```
printf("%7.3f", y)    /* notation décimale avec gabarit min. 7
                         (3 chiffres après le point) */
```

y = 2.655	__2.655
y = 32.65578339	_32.656
y = 0.000013265	__0.000

Gabarit ou précision variable

• Il est possible d'utiliser des paramètres dont la valeur peut varier d'un appel à l'autre, en utilisant à leur place le caractère *.

Exemple :

```
printf("%7.*f", n, y)    /* n indique la précision */
```

n=1 y = 2.655 ____2.7 (avec arrondi)

n=4 y = 2.655 _2.6550

NB : la valeur de *n* n'est pas affichée, car elle est utilisée pour la valeur du paramètre.

Les caractères

• Un caractère (char) est codé sur un domaine numérique de 256. (signé, de -128 à 127, non signé de 0 à 255).

• Il y a donc peu de caractères disponibles, ce qui explique pourquoi les caractères nationaux peuvent ne pas être représentés suivant le système où l'on compile.

Exemple :

```
char c = 'é';
printf("%c\n", c);
```

$\Longrightarrow$ conduira à une erreur de ce type :

error: stray '\342' in program
char c = 'é';

Les caractères échappés

• Certains caractères non imprimables peuvent être représentés par une convention à l'aide de l'anti-slash :

- \n : saut de ligne
- \t : tabulation
- \' : apostrophe
- \" : guillemet

Exemple :

```
printf("Paris est magique, \t \"c\'est vrai ? \" \n"); ==>
Paris est magique,   "c'est vrai ? "
```

NB : utiliser l'anti-slash pour préfixer le caractère s'appelle utiliser une séquence d'échappement. L'anti-slash peut-être imprimé en échappant l'anti-slash (\\). Attention à ne pas confondre le slash '/' qui n'a pas besoin d'être échappé.

Présentation générale de `scanf`

Présentation générale de `scanf`

- La fonction `scanf` lit une suite de caractères sur l'entrée standard.
- Comme dans le cas de `printf`, le format destiné à `scanf` contiendra des codes de format, à raison d'un code par information à lire.
- Syntaxe : `scanf("<format>", <&donnee_1>, <&donnee_2>, ...);`

NB : attention, cette syntaxe s'applique uniquement aux variables qui stockent des entiers, des réels ou un caractère.

- Les différents formats :
  - %d entier décimal
  - %f réel simple précision
  - %lf réel double précision
  - %c caractère (1 seul)
  - %s chaîne de caractères

Exemple d'utilisation de `scanf` :

```
-------------------------CODE-------------------------
int a;
double d;
char c;
printf("Donner les valeurs a, d et c : \n");
scanf ("%d %lf %c", &a, &d, &c);
printf("les valeurs saisies sont :\n");
printf("a = %d, d = %.2lf, c = %c\n", a, d, c);
------------------------------------------------------



=========================RÉSULTAT=====================
Donner les valeurs a, d et c :
94 4.6267 X
les valeurs saisies sont :
a = 94, d = 4.63, c = X
======================================================
```

La valeur de retour de scanf

- La valeur de retour de `scanf` permet de savoir si la lecture des informations s'est bien déroulée.
- Elle indique le nombre de valeurs convenablement lues et affectées.

Exemple d'utilisation de la valeur de retour d'un `scanf` :

```
-------------------------CODE-------------------------
int compte, a, b;
printf("Donner a et b :\n");
compte = scanf("%d %d", &a, &b);
printf("compte = %d\n", compte);
------------------------------------------------------


=========================RÉSULTAT=======================
Donner a et b :
94 200 #valeurs acceptables
compte = 2

Donner a et b :
94 az #ici la deuxième valeur n'est pas acceptée
compte = 1
======================================================
```

Limitation du gabarit

• Contrairement à `printf`, la notion de gabarit par défaut n'existe pas pour `scanf`.

• Il n'est pas possible d'imposer un nombre minimal de caractères ; en revanche on peut imposer un nombre maximal en mentionnant un gabarit à la suite du caractère % et avant le code de conversion.

Exemple de limitation du gabarit dans un `scanf` :

```
-------------------------CODE-------------------------
int a, b;
printf("Donner a et b :\n");
scanf("%3d %2d", &a, &b);
printf("a = %d, b = %d\n", a, b);
------------------------------------------------------


=========================RÉSULTAT=======================
Donner a et b :
94   2334
a = 94, b = 23

Donner a et b :
653677828229
a = 653, b = 67
========================================================
```

Le rôle des conversions numériques

Le rôle des conversions numériques

• On appelle conversion numérique, la conversion d'un type de base en un autre type de base. Une telle conversion peut être :

- implicite,
- explicite (cast)

• Les conversions implicites sont intègres (*i.e.* justes), c'est-à-dire qu'elles préservent la valeur initiale. Elles constituent un cas particulier de la conversion explicite.

• Les conversions explicites ou forcées, ne sont plus nécessairement intègres :

- Exemple de conversion flottant vers entier.

Exemple de conversion implicite :

```
printf("%d\n", 'A'+2); ==>
67
```

Remarque 1 : 'A' est de type char et 2 de type int. Dans ce cas, 'A' est tout d'abord converti en int (ici 65) avant que l'expression ne soit évaluée.

Exemple de conversion explicite :

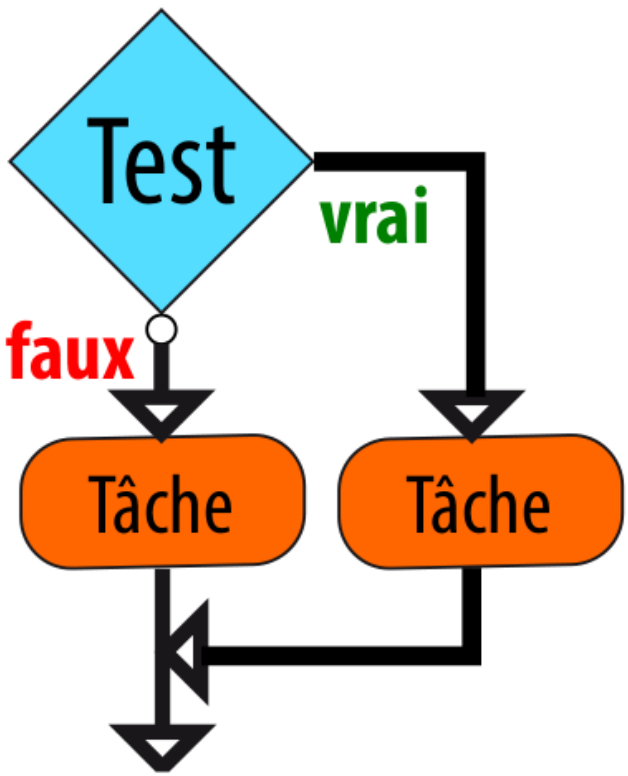
```
int res;
float a, b;
a=2.3; b=3.6;
res = (int)a + (int)b;
printf("res = %d\n", res); ==>
5
```

Remarque 2 : ici, on a converti explicitement des flottants en entiers avant l'opération. Lorsqu'on affecte un flottant à un entier, seule la partie entière, si elle peut être représentée, est retenue.

Structures de contrôle (if, while, for, switch, ...)

Structures conditionnelles

La forme la plus simple de structure conditionnelle est d'exécuter quelque chose dans le cas où une condition (*i.e.* Test) est vérifiée :



- Structure de contrôle `if`
- Syntaxe d'utilisation en `C` :

```
if (condition)
{
// Instructions si condition vraie
}
else // Optionnel
{
// Instructions si condition fausse
}
```

Exemple d'utilisation de `if` :

```
---------------------------------CODE---------------------------------
int a, b;
printf("Donner a et b :\n");
scanf("%d %d", &a, &b);
if (a > b)
{
   printf("a est plus grand que b !\n");
}
else if (a < b)
{
   printf("a est plus petit que b !\n");
}
else
{
   printf("a et b sont egaux !\n");
}
----------------------------------------------------------------------

================================RÉSULTAT================================
Donner a et b :
4 3
a est plus grand que b !
======================================================================
```

Opérateurs conditionnels "? :"

• Le langage C possède une paire d'opérateurs "? :" qui peut être utilisée comme une alternative à **if-else** et qui a l'avantage de pouvoir être intégrée dans une expression.

• Syntaxe d'utilisation :
`<expr1> ? <expr2> : <expr3>`

• Interprétation :
Si `<expr1>` est vraie alors la valeur de `<expr2>` est utilisée, sinon c'est celle de `<expr3>` qui est utilisée.

Exemple d'utilisation des opérateurs conditionnels "? :" :

```
-------------------------CODE-------------------------
int MAX, A, B;

printf("Saisir A et B : \n");

scanf("%d %d", &A, &B);
MAX = (A > B) ? A : B;

printf("MAX = %d\n", MAX);
------------------------------------------------------



=========================RÉSULTAT=======================
Saisir A et B :
94200 75005
MAX = 94200
======================================================
```

Autre exemple :

----------CODE----------

```
unsigned int P;
printf("Saisir votre nombre de pièces : \n");
scanf("%u", &P);
printf("Vous avez %u pièce%c \n", P, (P==1) ? ' ' : 's');
return 0;
```

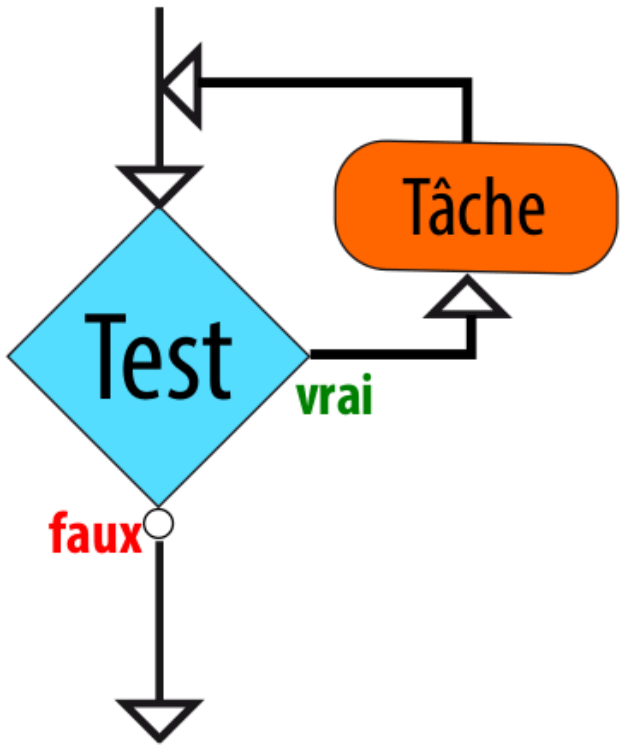
==========RÉSULTAT==========

```
Saisir votre nombre de pièces :
1
Vous avez 1 pièce

Saisir votre nombre de pièces :
94
Vous avez 94 pièces
```

==========

Itérations (`while` "Tant que ... faire")

Permet l'exécution de plusieurs fois une portion de code, généralement jusqu'à ce qu'une condition soit fausse.



- Boucle `While`

- Syntaxe en `C` :

```
while (condition)
{
// Instructions
}
```

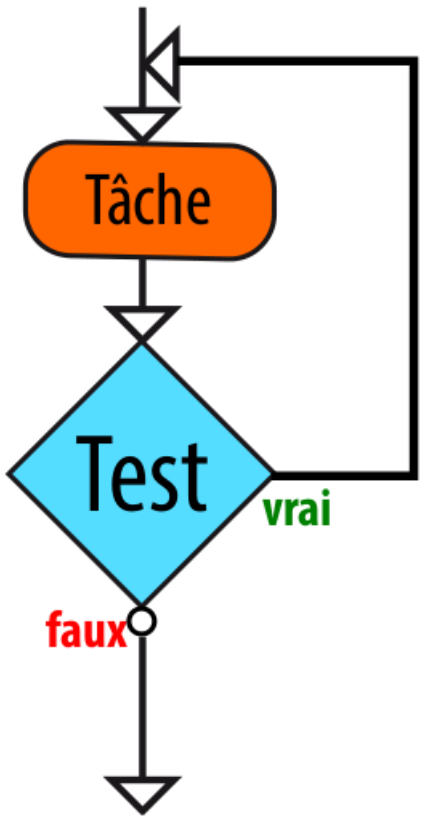
NB : le plus grand danger que présentent les boucles `while`, est que leur condition de sortie de boucle ne soit jamais fausse. Dans un tel cas, on ne sort jamais de la boucle (boucle infinie).

Exemple d'utilisation de `while` :

```
-------------------------CODE--------------------------
int a, b;
printf("Donner a et b :\n");
scanf("%d %d", &a, &b);

while (a > b)
{
  printf("Donner a et b :\n");
  scanf("%d %d", &a, &b);
}
-------------------------------------------------------
=========================RÉSULTAT======================
Donner a et b :
2 1
Donner a et b :
30 7
Donner a et b :
3 4
=======================================================
```

Itérations (`do ... while` "faire ... tant que")

Cela dit, contrairement à while, avec do ... while, la condition est évaluée à la fin de la boucle ; cela signifie que les instructions correspondant au corps de la structure de contrôle seront toujours exécutées au moins **une fois**, même si la condition est toujours fausse.



- Boucle `do ... While`

- Syntaxe en `C`

```
do
{
// Instruction à exécuter tant que la condition est vraie.
}
while ( condition );
```

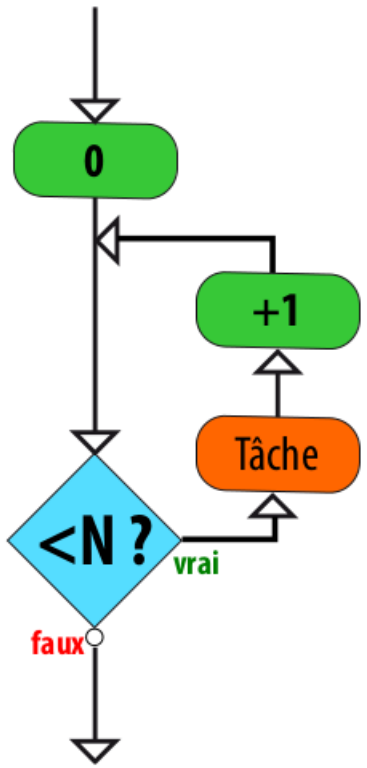
Exemple d'utilisation de `do ... while` :

```
----------------------------CODE-----------------------------
int a, b;

do
{
  printf("Donner a et b :\n");
  scanf("%d %d", &a, &b);
}while (a > b);
-------------------------------------------------------------

============================RÉSULTAT=========================
Donner a et b :
67 3
Donner a et b :
3 4
=============================================================
```

Remarque : les deux boucles vues précédemment sont des boucles de type événementiel. Si on ne sait pas à l'avance combien de fois on doit répéter une action, on utilise une boucle événementielle.

Itérations (`for`)

Cette boucle est généralement utilisée lorsque l'on veut répéter un nombre de fois connu une action.



- Boucle `for`

- Syntaxe en C

```
for (init.  ; condition d'arrêt ; instruction de boucle)
{
// Instructions
}
```

Exemple d'utilisation de `for` :

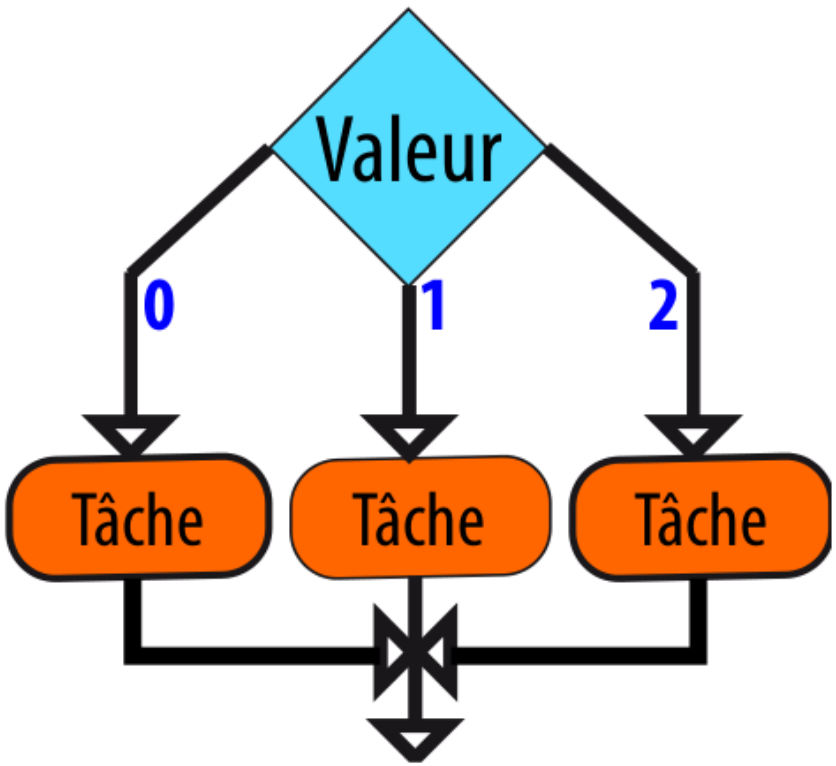
```
-------------------------CODE--------------------------
int i;
for (i=0; i<10; i++)
{printf("i = %d\n", i);}
-------------------------------------------------------


=========================RÉSULTAT======================
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
=======================================================
```

Boucles d'apparence infinie

```
for ( ; ; ; )
{
// Instructions contenant à un moment
// un break
}

while (1)
{
// Instructions contenant à un moment
// un break
}
```

Le switch

Cette structure est particulière dans le sens où elle ne permet que de comparer une variable à plusieurs valeurs.



- Structure `switch`

- Syntaxe en C :

```
switch(nom_de_la_variable)
{
case valeur_1:
Instructions à exécuter dans le cas où la variable vaut valeur_1
break; // sort directement de la boucle
case valeur_2:
Instructions à exécuter dans le cas où la variable vaut valeur_2
break;
default:
Instructions à exécuter dans le cas où la variable vaut
une valeur autre que valeur_1 et valeur_2
break;
}
```

NB : une structure `switch` peut avoir autant de case que vous le souhaitez. Le cas 'default' est optionnel.

Exemple d'utilisation de switch :

```
--------------------------------CODE--------------------------------
char c;
scanf("%c", &c);
switch(c)
{
    case 'a':
            printf("Paris est magique !\n");
        break;
    case 'b':
            printf("ESIEA\n");
        break;
    case 'c':
            printf("Remise à niveau prog C\n");
        break;
}
--------------------------------------------------------------------

================================RÉSULTAT================================
a
Paris est magique !

b
ESIEA
====================================================================
```

Programmation structurée

- Utiliser au mieux les structures de contrôle :
  - Sélection : if ou case
  - Itération : for ou while (2 façons)
  - Instructions de sortie exit ou return
- Minimiser l'imbrication de ces structures :
  - Pas plus de 3 niveaux d'imbrications
  - Sinon, repenser votre code !
  - ... ou utiliser des fonctions !

Exercice : Qu'affiche ce programme ?

```c
int main()
{
   int a, b, c;
   a=94; b=5; c=53;

   a=(b>5)?a-4:a+4;
   b=(a<100)?b/2:2*b;
   c=((b%2)==1)?c+b:c-b;

   printf("a=%d, b=%d, c=%d \n", a--, b++, --c);
   printf("a=%d, b=%d, c=%d \n", a, b, c);

   return 0;
}
```

Résultats :

```
a=98, b=2, c=50
a=97, b=3, c=50
```

Remarques :

```
printf("a=%d, b=%d, c=%d \n", a--, b++, --c);
```

1. `a--` : a sera utilisée, puis après affecter `a-1` à a. Ce que l'on appelle une **post-décrémentation**.
2. `b++` : b sera utilisée, puis après affecter `b+1` à b. Ce que l'on appelle une **post-incrémentation**.
3. `--c` : affecter d'abord `c-1` à c, puis après utiliser c. Ce que l'on appelle une **pré-décrémentation**.
4. Il existe aussi une **pré-incrémentation**, par exemple `++c`.

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var1, var2;
  var1 = 2; var2 = 1;

  while((var1 % var2) == 0)
  {
    printf("%d %d\n", var1, var2);
    var1 *= 2;
    var2++;
  }

  printf("%d %d\n", var1, var2);

  return 0;
}
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var1, var2;
  var1 = 2; var2 = 1;

  while((var1 % var2) == 0)
  {
    printf("%d %d\n", var1, var2);
    var1 *= 2;
    var2++;
  }

  printf("%d %d\n", var1, var2);

  return 0;
}
```

Résultats :

```
2 1
4 2
8 3
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var1, var2;
  var1 = 2; var2 = 1;

  do
  {
    printf("%d %d\n", var1, var2);
    var1 *= 2;
    var2++;
  }while((var1 % var2) == 0);

  printf("%d %d\n", var1, var2);
  return 0;
}
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var1, var2;
  var1 = 2; var2 = 1;

  do
  {
    printf("%d %d\n", var1, var2);
    var1 *= 2;
    var2++;
  }while((var1 % var2) == 0);

  printf("%d %d\n", var1, var2);
  return 0;
}
```

Résultats :

```
2 1
4 2
8 3
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var3, var4;
  var3 = 8; var4 = 3;

  while((var3 % var4) == 0)
  {
    printf("%d %d \n", var3, var4);
    var3 *= 2;
    var4++;
  }

  printf("%d %d \n", var3, var4);

  return 0;
}
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var3, var4;
  var3 = 8; var4 = 3;

  while((var3 % var4) == 0)
  {
    printf("%d %d \n", var3, var4);
    var3 *= 2;
    var4++;
  }

  printf("%d %d \n", var3, var4);

  return 0;
}
```

Résultats :

```
8 3
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var3, var4;
  var3 = 8; var4 = 3;

  do
  {
    printf("%d %d \n", var3, var4);
    var3 *= 2;
    var4++;
  } while((var3 % var4) == 0);

  printf("%d %d \n", var3, var4);

  return 0;
}
```

Résultats :

```
8 3
16 4
32 5
```

Remarque :
On constate dans ce cas, à cause du `do ...while` on a pu rentrer la première fois dans la boucle et cela a permis au test d'être vrai grâce à la mise à jour de `var3` et `var4`.

Exercice : Qu'affiche ce programme ?

```c
int main()
{
   int var5;

   for( var5 = 0 ; var5 < 10 ;var5++)
   {
      printf("%d ", var5);
   }

   printf("\n%d \n", var5);

   return 0;
}
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
  int var5;

  for( var5 = 0 ; var5 < 10 ;var5++)
  {
    printf("%d ", var5);
  }

  printf("\n%d \n", var5);

  return 0;
}
```

Résultats :

```
0 1 2 3 4 5 6 7 8 9
10
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
   int var6;

   for( var6 = 10 ; var6 > 0 ; var6--)
   {
      printf("%d ", var6);
   }

   printf("\n%d \n", var6);

   return 0;
}
```

Exercice : Qu'affiche ce programme ?

```c
int main()
{
   int var6;

   for( var6 = 10 ; var6 > 0 ; var6--)
   {
      printf("%d ", var6);
   }

   printf("\n%d \n", var6);

   return 0;
}
```

Résultats :

```
10 9 8 7 6 5 4 3 2 1
0
```

Les fonctions

Les fonctions

- Définition : bloc d'instructions nommé.
- Déclaration :

```
type nom_de_la_fonction(type0 arg0, type1 arg1, ...);
```

- Définition :

```
type nom_de_la_fonction(type0 arg0, type1 arg1, ...)
{
//Corps de la fonction
}
```

Règles d'utilisation

• L'ordre, le type et le nombre d'argument doivent être respectés lors de l'appel de la fonction.

• Une fonction ne doit pas être déclarée dans le main. On peut la déclarer soit :

- au dessus du main
- en dessous du main, mais dans ce cas déclarer d'abord le prototype de la fonction

• Ainsi, l'appel d'une fonction doit être stipulé après sa déclaration ou celle de son prototype.

• Dans le cas où la fonction ne renvoie rien, alors elle est de type `void`.

Retour de la fonction

- Une fonction retourne un résultat :
  - grâce à l'instruction `return`
  - si rien n'est renvoyé alors le type doit être `void`
- Le type de la fonction correspond au type de la valeur retournée.
- La valeur retournée peut être ≪capturée≫ avec une affectation (=).
- Pour appeler la fonction, il suffit de donner son nom, les paramètres (constantes, variables, etc.) et éventuellement en stockant la valeur retour.

Exemple :

```c
#include<stdio.h>

unsigned int factorielle(unsigned int n); /* déclaration */


void main() /* Fonction principale */
{
 unsigned int res;
 res = factorielle(12);
 printf("Resultat : %u\n", res);
}

unsigned int factorielle(unsigned int n) /* définition */
{
 unsigned int res=1;
 int i;
 if(n==0) return (1);
 for(i=2; i<=n; i++)
 {
   res *= i;
 }
 return (res);
}
```

```
Resultat : 479001600
```

Propriétés

Tout paramètre est passé par copie ou par valeur :

- Recopie en mémoire des paramètres temporaires.
- Toute modification des paramètres dans le corps de la fonction ne modifie pas les paramètres dans la fonction appelante.
- Toute variable déclarée dans un bloc (et donc dans une fonction) est détruite à la fin de celui-ci.
- Avant d'utiliser une fonction il faut en déclarer le prototype avant (dans l'en-tête du programme).

Passage d'argument par valeur

```c
#include<stdio.h>

int multiplie(int n, int facteur); /* déclaration */

void main() /* Fonction principale */
{
 int n, facteur;
 printf("Rentrer un premier chiffre : ");
 scanf("%d",&n);

 printf("Rentrer un deuxième chiffre : ");
 scanf("%d",&facteur);

 printf("%d x %d = %d\n", n, facteur, multiplie(n, facteur));
}


int multiplie(int n, int facteur) /* définition */
{
 return n*facteur;
}
```

```
Rentrer un premier chiffre : 4
Rentrer un deuxième chiffre : 5
4 x 5 = 20
```

Passage d'argument par adresse

- Au lieu de recopier la valeur de la variable, on passe en argument l'adresse mémoire de l'endroit où se trouve la variable.
- Il nous sera donc possible de modifier son contenu.
- Cette technique permet de modifier plusieurs arguments à la fois.
- Permet également de passer des tableaux en argument.
- Un exemple sera donné plus tard pendant l'étude des pointeurs.

Les variables globales

Les variables globales

• On peut définir en C, des variables globales qui sont en théorie accessibles à toutes les fonctions, que ces dernières soient ou non définies dans le même fichier source.

• Il existe aussi un mécanisme permettant d'interdire l'usage d'une variable globale en dehors du fichier source où elle a été définie.

Exemple d'utilisation de variables globales :

```c
#include <stdio.h>

int A, B; /* variables globales */

float fct(float x)
{
 return (A*x + B);
}

int main(int argc, char *argv[])
{
 float x, y;
 x = 2.5;
 A = 2; B = 3;
 y = fct(x);
 printf("%.2f\n", y);
return 0;
}
```

```
$ ./EXEC
8.00
```

NB : les variables *A* et *B* ont été déclarées en dehors de toute fonction, elles sont donc connues de toutes les fonctions dont la définition apparaît dans le même fichier source.

Utilisation de variables globales définies dans un autre fichier source

- Dans ce contexte, il est nécessaire dans l'un des deux fichiers de prévenir le compilateur que l'allocation des emplacements est prise en compte ailleurs.
- On utilise le mot clé `extern` dans l'une des deux déclarations globales.

Exemple :

Fichier 1 => code1.c

```c
#include <stdio.h>
int A, B; /* variables globales */
float fct(float x);

int main(int argc, char *argv[]){
 float x, y;
 x = 2.5;
 A = 2; B = 3;
 y = fct(x);
 printf("%.2f\n", y);
 return 0;
}
```

Fichier 2 => code2.c

```c
#include <stdio.h>
extern int A, B; /* A et B sont globales, mais leur emplacement
                    est réservé dans un autre fichier source */
float fct(float x)
{return (A*x + B);}
```

```
$ gcc -c code1.c
$ gcc -c code2.c
$ gcc -o EXEC code1.o code2.o
$ ./EXEC
8.00
```

Variable globale cachée dans un fichier source

• Il est cependant possible de "cacher" une variable dans un fichier source, c'est-à-dire de la rendre inaccessible à un autre fichier source :

- on utilise dans ce cas le mot clé `static`

```
-----------------------------------------------------------
static int A;

fct()
{
  ...
}

main()
{
  ...
}
-----------------------------------------------------------
```

NB : Sans `static`, *A* serait une variable globale "ordinaire". Avec `static`, il devient impossible de faire référence à *A* depuis un autre fichier source, même en utilisant le mot clé `extern`.

Les opérateurs de manipulation de bits

Présentation

- En langage C, il existe des opérateurs permettant d'agir directement sur les bits d'une valeur :
  - ces opérateurs ne fonctionnent que sur les types **entiers**.
- Ils permettent de réaliser des opérations logiques bit-à-bit :
  - ET
  - OU-inclusif
  - OU-exclusif
  - Complément
  - Décalages de bits
- ATTENTION à ne pas confondre avec les opérateurs logiques && et ||.

Les opérateurs bit-à-bit

- Voici la table de vérité des opérateurs bit-à-bit **binaires**;

Bit (opérande 1)	0	0	1	1
Bit (opérande 2)	0	1	0	1
& (ET)	0	0	0	1
| (OU-inclusif)	0	1	1	1
^(OU-exclusif "XOR")	0	1	1	0

- L'opérateur unaire $\sim$ de complémentarité est également du type bit-à-bit. Il inverse simplement les bits de son unique opérande.

Bit (opérande)	0	1
$\sim$ (Complément à un)	1	0

NB : le " ^" appelé également XOR, est très souvent utilisé en cryptographie car, il est réversible.

Exemple :

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
  unsigned short int M, N;
  M=18000;/*01000110 01010000 = 0x4650*/
  N=10563;/*00101001 01000011 = 0x2943*/
  printf("%hu\n%hu\n%hu\n%hu\n", M&N,M|N,M^N,~M);
  return 0;
}
```

```
64
28499
28435
47535
```

Valeur	Binaire	Hexadécimal
64	00000000 01000000	0x0040
28499	01101111 01010011	0x6F53
28435	01101111 00010011	0x6F13
47535	10111001 10101111	0xB9AF

Les opérateurs de décalage

• Ils permettent de réaliser des décalages à "droite" ou à "gauche" sur les bits du premier opérande.

• $X << n$: décale les bits de X de n positions vers la gauche (X n'est pas modifié).

• $X >> n$: décale les bits de X de n positions vers la droite (X n'est pas modifié).

• Pour le décalage à gauche (resp. droite) les bits de poids fort (resp. faible) sont perdus.

Exemple :

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
  unsigned short int M, N, P;
  M=18000;/*01000110 01010000 = 0x4650*/
  N=M<<4;
  P=M>>8;
  printf("M=%hu\nN=%hu\nP=%hu\n", M,N,P);
  return 0;
}
```

```
M=18000
N=25856
P=70
```

Valeur	Binaire	Hexadécimal
18000	01000110 01010000	0x4650
25856	01100101 00000000	0x6500
70	00000000 01000110	0x0046

Décalage circulaire à gauche

• Le décalage circulaire consiste à ré-injecter les bits sortant en tant que bits entrant.

• Pour décaler A de taille $SIZE$ de b bits vers la gauche :

- on copie d'abord A dans une variable TMP ;
- on décale TMP de $SIZE - b$ vers la droite (pour que les b bits forts deviennent faibles) ;
- on décale A de b bits vers la gauche (pour que les $SIZE - b$ bits faibles deviennent forts) ;
- on effectue ensuite un OU-inclusif ($|$) entre les résultats des décalages de A et TMP (pour greffer chacune des parties).

• **NB :** pour le décalage circulaire à droite, il suffit de d'effectuer les opérations dans l'autre sens.

Exemple :

```c
#include<stdio.h>

unsigned char DEC_CIR_GAUCHE (unsigned char A, int b)
{
 b = b % 8;
 return ((A>>(8-b))|(A<<b));
}

int main(int argc, char ** argv)
{
 /*240 = 1111 0000*/
 printf("%d\n", DEC_CIR_GAUCHE (240, 4));
 printf("%d\n", DEC_CIR_GAUCHE (240, 7));
 printf("%d\n", DEC_CIR_GAUCHE (240, 8));
 return 0;
}
```

```
15
120
240
```

Comment modifier à la source la valeur d'un ou de plusieurs bits de positions données

• On utilise ce que l'on appelle un masque binaire, à savoir un entier non-signé dans lequel les bits ayant la même position que les bits à modifier ont la valeur 1 et les autres 0.

• Ainsi, combiner l'entier concerné avec le masque via l'opérateur "|", permet de forcer à un les bits ciblés.

• Combiner l'entier concerné avec le complément à un du masque via l'opérateur "&", permet de forcer à zéro les bits ciblés.

Exemple :

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
  unsigned short int M, masque;
  M = 18000;    /*01000110 01010000 = 0x4650*/
  masque = 255;/* 00000000 11111111 = 0x00FF*/
  M = M | masque; /*forcer à un les 8 bits de poids faible*/
  printf("M | masque = %hu\n", M);
  M = 18000; /*Réinitialisation de la valeur de M*/
  M = M & ~masque;/*forcer à 0 les 8 bits de poids fort*/
  printf("M & ~masque = %hu\n", M);

  return 0;
}
```

```
M | masque = 18175
M & ~masque = 17920
```

Valeur	Binaire	Hexadécimal
18175	01000110 11111111	0x46FF
17920	01000110 00000000	0x4600

Comment connaître la valeur d'un ou de plusieurs bits de positions données

• Supposons qu'on veut extraire le bit correspondant à une position p.
On peut procéder de deux façons :

1. Utiliser un masque égal à 1 en le combinant via l'opérateur "&" avec l'élément décalé à droite de p positions.
2. Utiliser un masque où seulement le bit situé à la position p est égal à 1, puis on le combine à l'élément via l'opérateur "&".

NB : la 1ère est plus simple, car il suffit après de faire un test à "1" ou "0" pour connaître le bit en question.

Exemple :

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
  unsigned short int M, masque, p;
  M = 18000;   /*01000110 01010000 = 0x4650*/
  masque = 1;
  printf ("Donnez la position du bit à extraire : ");
  scanf("%hu", &p);
  while (p != 16)
  {
    if (((M>>p) & masque) == 1){printf("bit = 1\n");}
    else{printf("bit = 0\n");}
    printf ("Donnez la position du bit à extraire : ");
    scanf("%hu", &p);
  }

  return 0;
}
```

```
Donnez la position du bit à extraire : 4
bit = 1
Donnez la position du bit à extraire : 15
bit = 0
Donnez la position du bit à extraire : 9
bit = 1
Donnez la position du bit à extraire : 0
bit = 0
Donnez la position du bit à extraire : 16
```

Exemple : (affichage de tous les bits)

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
  int i;
  unsigned short int M, NB_BITS;
  NB_BITS = sizeof(unsigned short int) * 8;
  printf("Donnez l'élément M : ");scanf("%hu", &M);

  while(M != 0)
  {
   for (i=NB_BITS-1; i>=0; i--){printf("%c", ((M>>i)&1)?'1':'0');}
   printf("\nDonnez l'élément M : ");scanf("%hu", &M);
  }
  return 0;
}
```

```
Donnez l'élément M : 1
0000000000000001
Donnez l'élément M : 255
0000000011111111
Donnez l'élément M : 65535
1111111111111111
Donnez l'élément M : 18000
0100011001010000
Donnez l'élément M : 0
```

L'instruction goto et les étiquettes

Présentation

• Toute instruction exécutable peut être **précédée** d'une étiquette (*i.e.* identificateur) suivie de deux-points.

• Exemples d'étiquettes :

```
-------------------------------------------------
test :   if (...) /*test est une étiquette pour if*/
         {
           exec : etat = 1;
         }
boucle : for (...) /*boucle est une étiquette pour for*/
         {...}
-------------------------------------------------
```

• Ces étiquettes ne peuvent être utilisées que par l'instruction `goto`.

• L'instruction `goto` permet d'atteindre puis exécuter directement l'instruction portant l'étiquette spécifiée.

NB : l'instruction portant l'étiquette et le `goto` doivent figurer dans le corps de la même fonction.

goto à l'intérieur d'un même bloc

• On peut utiliser l'instruction `goto` dans un même bloc d'instructions. Par exemple une boucle `for` ou `while`.

• Cette utilisation est beaucoup moins fréquente, car il suffit d'utiliser une instruction `if` appropriée pour gérer les actions.

Exemple : (ce goto n'est pas trop utile car un if-else fera mieux l'affaire)

```c
#include <stdio.h>

int main (int argc, char ** argv)
{
  int X;
  do
  {
    printf("Saisir la valeur de X (>=0) : ");
    scanf("%d", &X);
    if (X<0)
    {
      printf("X doit être >= 0\n");
      goto suite;
    }
    printf("%d mod 25 = %d\n",X, X % 25);
    suite : ;
  }while (X);
  return 0;
}
```

```
Saisir la valeur de X (>=0) : 26
26 mod 25 = 1
Saisir la valeur de X (>=0) : -56
X doit être >= 0
Saisir la valeur de X (>=0) : 56482
56482 mod 25 = 7
Saisir la valeur de X (>=0) : -4
X doit être >= 0
Saisir la valeur de X (>=0) : 0
0 mod 25 = 0
```

goto pour traiter une circonstance exceptionnelle (erreur)

• Pour gérer une circonstance très particulière (erreur), qui peut compromettre le bon déroulement de la suite du programme, on peut utiliser `goto`.

NB : en général `break` n'est pas toujours la meilleure solution dans ces genres de situations.

Exemple :

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

int main (int argc, char ** argv)
{
  int i;float X;
  X = atof(argv[1]);
  for (i=0; i<1000000000; i++)
  {
    X=3.9999*X*(1-X);
    if (X == 0.500)
    {
      printf("Processus Interrompu \n");
      goto erreur;
    }
    /*Suite du programme pour manipuler X*/
  }
  return EXIT_SUCCESS;

  erreur : printf("Valeur non gérée par le programme !!!\n");
  exit(-1);
}
```

```
$ ./EXEC 0.637
$
```

goto passant de l'extérieur vers l'intérieur d'un bloc

- Ce cas est beaucoup plus dangereux pour le bon fonctionnement du programme car, on peut rentrer directement dans un bloc avec des informations manquantes par exemple des variables non encore initialisées ou même des variables ayant d'autres valeurs différentes de celles attendues.
- Cette façon de faire est fortement déconseillée.

Exemple :

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

int main (int argc, char ** argv)
{
  int i, X, res;
  FILE * fic = fopen("données.txt", "r");
  goto suite;
  i=0;
  while (i<10)
  {
            fscanf(fic, "%d", &X);
    suite : res = (i * X + 3421) % 256;
            printf("%d ", res);
            i++;
  }
  printf("\n");
  fclose(fic);
  return 0;
}
```

```
93
 108 133 158 183 208 233 2 27 52 77 102 127 152 177 202 227 252 21 46 71
246 15 40 65 90 115 140 165 190 215 240 9 34 59 84 109 134 159 1^C
```

NB : le programme affiche soit la valeur 93 et il s'arrête ou rentre dans une grande boucle.

Tableaux
Chaînes de caractères
Pointeurs

Les tableaux

- Un tableau permet de stocker plusieurs valeurs de même type dans des cases contiguës de la mémoire.
- Déclaration avec [entier] : réserve l'espace mémoire.
- Accès en temps constant lecture/écriture (Attention aux erreurs d'accès).

Tableaux statiques à 1 dimension

• Définition : ensemble de variable de même type, de même nom caractérisées par un index.

• Déclaration :

```
type nom_tableau[dimension];
```

Exemples de déclaration de tableaux :

```
char buffer[25];
int tableau[100];
unsigned int TAB[27];
```

- Après déclaration, on peut initialiser le tableau avec des valeurs du même type, par exemple :
  - `int tableau[256] = {12, 142, 12, 2, 48};`
  - tableau[129] = 7856;
  - tableau[255] = 6;
- Pour accéder aux éléments du tableau, il suffit de taper :
  - `nom_tableau[indice]`

Remarques :

- le premier élément du tableau commence à l'indice `'0'`
- le dernier élément se trouve à l'indice `'dimension-1'`
- au départ, les valeurs ne sont pas initialisées
- les débordements sur le tableau ne sont pas vérifiés (**segmentation fault**)

Exemple :

```c
#include<stdio.h>

void Affichage_tab(int tab[], int taille)
/* fonction qui affiche les éléments d'un tableau */
{
 int i;
 for (i=0; i<taille; i++)
 {
   printf("%d ", tab[i]);
 }
 printf("\n");
}

void main() /* Fonction principale */
{
 int tableau[10] = {199, 12, 2, 248};
 Affichage_tab(tableau, 10);
}
```

```
199 12 2 248 0 0 0 0 0 0
```

Tableaux statiques à 2 dimension et plus

- Définition : il s'agit d'un tableau de tableaux.
- Déclaration :

```
type nom_tableau[dim1][dim2]···[dim3];
```

- Permet de manipuler par exemple des matrices.

Exemple :

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

void Affichage_tab_2D(int nb_l, int nb_c, int tab[nb_l][nb_c])
/* fonction qui affiche les éléments d'un tableau 2 dimensions */
{
 int i, j;
 for (i=0; i<nb_l; i++)
 {
   for (j=0; j<nb_c; j++)
   {
     printf("%d ", tab[i][j]);
   }
   printf("\n");
 }
 printf("\n");
}

void main() /* Fonction principale */
{
 int i, j, nb_l, nb_c, R;
 nb_l=10; nb_c = 8;
 int matrice[nb_l][nb_c];
 srand (time(NULL));

 for (i=0; i<nb_l; i++)
 {
   for (j=0; j<nb_c; j++)
   {
     R = rand() % 10; //Récupération d'un élément aléatoire entre 0 et 9
     matrice[i][j] = R;
   }
 }
   Affichage_tab_2D(nb_l, nb_c, matrice);
}
```

Le résultat de l'exécution :

```
6 1 0 6 4 5 4 5
2 1 8 2 8 7 6 6
0 1 4 4 1 9 1 0
0 7 7 6 9 1 7 7
3 9 6 7 4 0 4 9
3 3 3 3 0 0 9 2
3 6 7 4 5 0 4 8
0 3 4 1 4 1 1 9
0 7 6 7 9 3 6 4
6 9 7 8 1 8 2 4
```

Tableaux de caractères

• Jusqu'à présent, on a utilisé des tableaux de caractères pour manipuler des ensembles de caractères.

• Par exemple pour déclarer et initialiser un tableau composé de 7 caractères ('B', 'o', 'n', 'j', 'o', 'u', 'r') :

```
char word[7] = {'B','o','n','j','o','u','r'};
```

• Une limite opérationnelle s'impose quand le nombre de caractères à manipuler explose.

Quelques problèmes :

- Si on ne connaît pas la taille du mot (ou de la phrase) que l'on souhaite mémoriser, quelle taille de tableau choisir ?
- Si le mot (ou la phrase) est d'une taille inférieure, comment gérer les cases non utilisées ?

```
char word[10] = {'B','o','n','j','o','u','r'};
```

Que valent les cases d'indices 7 à 9 ?

Quelques problèmes :

- Si on ne connaît pas la taille du mot (ou de la phrase) que l'on souhaite mémoriser, quelle taille de tableau choisir ?
- Si le mot (ou la phrase) est d'une taille inférieure, comment gérer les cases non utilisées ?

```
char word[10] = {'B','o','n','j','o','u','r'};
```

Que valent les cases d'indices 7 à 9 ?
Les cases de 7 à 9 contiennent le caractère `NUL` symbolisé par '\0' associé également à la valeur 0.

Comment s'en sortir ?

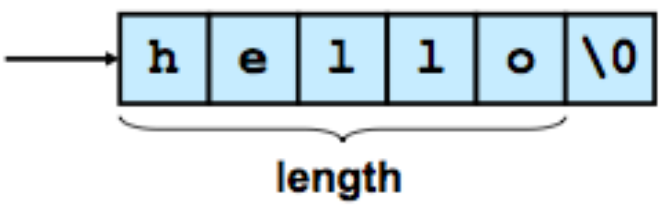
- prendre une taille (maximum) de tableau qui puisse convenir dans tous les cas ;
- ou mettre une balise spéciale de fin de partie utile dans le tableau : toutes les cases d'indice inférieur à celle contenant la balise contiennent la partie utile et toutes les autres contiennent de l'information non maîtrisée (état de la mémoire à l'instant du programme).

Chaînes de caractère

• ATTENTION : en C, il n'existe pas de variable de type chaîne.

• Par contre il existe une convention de représentation des chaînes qui consiste à placer un caractère de code NUL (\0) à la fin d'une succession d'octets représentant chacun des caractères de la chaîne.

• Ainsi, une chaîne de n caractères occupe en mémoire un emplacement de $n + 1$ octets. En mémoire, la chaîne "hello" est représentée ainsi :



- Une chaîne de caractères est tout simplement un tableau de caractères contenant une balise de fin. On peut donc utiliser l'opérateur [] pour accéder à une case en lecture ou écriture.
- La déclaration d'une chaîne de caractères suit le modèle suivant :

```
char word[10];
```

NB : `word` est un tableau de `char` dans lequel on peut stocker une chaîne d'au plus 9 caractères.

- Il est possible de déclarer et d'initialiser en même une chaîne de caractères en terminant par le caractère '\0'

```
char word[10] = {'B','O','N','J','O','U','R','\0'};
```

- En pratique, on utilise plutôt l'opérateur "" qui sépare la chaîne en caractères, fait la conversion caractère à caractère via la table ASCII et ajoute '\0' à la fin.

```
char word[10] = "BONJOUR";
```

Affichage d'une chaîne de caractère

Pour afficher une chaîne de caractère, on peut effectuer une boucle sur le tableau correspondant et afficher élément par élément la chaîne jusqu'au dernier caractère.

```c
#include <stdio.h>


int main()
{
  int i=0; char word[10] = "BONJOUR";

  while (word[i] != '\0')
  {
    printf("%c", word[i]);
    i++;
  }
  printf("\n");

  return 0;
}
```

```
BONJOUR
```

NB : cette façon marche mais n'est pas optimale puisqu'on peut effectuer cela en évitant la boucle.

Sachant que la chaîne se termine par '\0', il existe un moyen de l'afficher en une instruction en utilisant la fonction `printf` avec le descripteur `%s`, comme dans cet exemple :

```c
#include <stdio.h>


int main()
{
  char word[10] = "BONJOUR";

  printf("%s\n", word);

  return 0;
}
```

```
BONJOUR
```

NB : on a le même affichage en évitant les boucles et aussi les possibilités de faire des débordements de zone mémoire.

Comment saisir une chaîne de caractère depuis l'entrée standard ?

Il existe plusieurs fonctions qu'on peut utiliser pour récupérer une chaîne de caractères depuis l'entrée standard.

scanf()

• Pour saisir une chaîne de caractères, on peut utiliser la fonction `scanf` avec le descripteur `%s` et sans mettre `&` devant le nom du tableau :

```
scanf("%s", word);
```

• `scanf` remplira le tableau `word` lettre à lettre et mettra '\0' à la fin SAUF si la fonction rencontre un espace ou entrée.

Exemple : (Quand la chaîne saisie ne contient pas d'espace)

```c
#include <stdio.h>


int main()
{
  char word[20];

  printf("Saisir une chaîne de caractères : ");
  scanf("%s", word);
  printf("Chaîne saisie : %s\n", word);


  return 0;
}
```

Après exécution on obtient :

```
Saisir une chaîne de caractères : ESIEA
Chaîne saisie : ESIEA
```

Exemple : (Quand la chaîne saisie contient un espace)

```c
#include <stdio.h>


int main()
{
  char word[20];

  printf("Saisir une chaîne de caractères : ");
  scanf("%s", word);
  printf("Chaîne saisie : %s\n", word);


  return 0;
}
```

Après exécution on obtient :

```
Saisir une chaîne de caractères : Paris est magique !
Chaîne saisie : Paris
```

NB : la récupération des caractères s'est arrêtée juste avant l'espace.

Il est possible de spécialiser la fonction `scanf` en précisant entre [] la liste des caractères autorisés (donc tout autre caractère est interdit et provoque la fin du `scanf`)

Exemple : (on autorise que les caractères numériques)

```c
#include <stdio.h>


int main()
{
  char word[20];

  printf("Saisir une chaîne de caractères : ");
  scanf("%[0123456789]", word);
  printf("Chaîne saisie : %s\n", word);

  return 0;
}
```

Après exécution on obtient :

```
Saisir une chaîne de caractères : 123456789A
Chaîne saisie : 123456789
```

```
Saisir une chaîne de caractères : 12345F1234
Chaîne saisie : 12345
```

Il est possible également d'indiquer uniquement la liste des caractères qui sont interdits, en utilisant le symbole ^ . Cela permettrait de récupérer toute la chaîne saisie.

Exemple : (on autorise tous les caractères sauf **entrée**)

```c
#include <stdio.h>


int main()
{
  char word[20];

  printf("Saisir une chaîne de caractères : ");
  scanf("%[^\n]", word);
  printf("Chaîne saisie : %s\n", word);


  return 0;
}
```

Après exécution on obtient :

```
Saisir une chaîne de caractères : Paris est magique !
Chaîne saisie : Paris est magique !
```

NB : on voit ici qu'on a pu récupérer toute notre chaîne saisie y compris les espaces.

Il est également possible de limiter le nombre de caractères maximum à saisir. Cela permettrait de ne pas déborder sur l'espace mémoire réservé, car l'utilisateur peut saisir n'importe quoi en entrée.

Exemple : (on limite le nombre de caractères à saisir)

```c
#include <stdio.h>


int main()
{
  char word[20];

  printf("Saisir une chaîne de caractères : ");
  scanf("%19[^\n]", word);
  printf("Chaîne saisie : %s\n", word);


  return 0;
}
```

Après exécution on obtient :

```
Saisir une chaîne de caractères : Ici c'est Paris !!!!!!!!!!!!!
Chaîne saisie : Ici c'est Paris !!!
```

NB : on voit que seulement 19 caractères on été récupérés, comme ça à la 20 ème position il y aura le '\0' de fin de chaîne.

fgets()

• Cette fonction permet de lire une suite de caractères à partir d'un flux quelconque (l'entrée standard ou un fichier).

• Le nombre maximal de caractères à lire est de n-1 (où *n* désigne la taille du tableau), car le dernier caractère est réservé au zéro de fin de chaîne.

• Concernant l'adresse de retour, `fgets` fournit l'adresse de la chaîne lue, lorsque la lecture s'est bien déroulée. Elle renvoie le pointeur NULL en cas d'erreur.

Exemple : (récupération d'une chaîne avec `fgets`)

```c
#include <stdio.h>


int main()
{
  char word[20];

  printf("Saisir une chaîne de caractères : ");
  fgets(word, 20, stdin); /*19 caractères à lire au maximum*/
  printf("Chaîne saisie : %s\n", word);


  return 0;
}
```

Après exécution on obtient :

```
Saisir une chaîne de caractères : La MCN en force !!!!!!!
Chaîne saisie : La MCN en force !!!
```

NB : les caractères excédentaires restent disponibles dans le tampon pour une prochaine lecture.

Exemple : (récupération d'une chaîne avec son excédent)

```c
#include <stdio.h>


int main()
{
  char word[23];
  char excedent[10];

  printf("Saisir une chaîne de caractères : ");
  fgets(word, 23, stdin); /*22 caractères à lire au maximum*/
  printf("Chaîne saisie : %s\n", word);
  scanf("%9[^\n]", excedent); /*Pour récupérer le reste de la chaîne*/
  printf("Reste de la chaîne : %s\n", excedent);

  return 0;
}
```

Après exécution on obtient :

```
Saisir une chaîne de caractères : J'adore le langage C !YOUPI ...
Chaîne saisie : J'adore le langage C !
Reste de la chaîne : YOUPI ...
```

NB : on voit qu'on n'a pas eu à faire une deuxième saisie, car le `scanf` à récupéré directement les caractères restants lors de la première saisie.

La bibliothèque `string.h` contient de nombreuses fonctions utiles pour manipuler les chaînes comme :

- `strlen` qui renvoie le nombre de caractères utiles d'une chaîne de caractères.

- `strcat` qui concatène deux chaînes de caractères (*i.e.* met une chaîne de caractères à la suite d'une autre pour en faire une unique grande).

- `strcpy` qui copie une chaîne dans une autre.

- `strcmp` qui compare une chaîne de caractères avec une autre.

- etc.

Les pointeurs

• C permet de manipuler des adresses d'objets ou de fonctions, par le biais de ce que l'on nomme les pointeurs.

• Un pointeur est une variable dont la valeur est une adresse (ou qui a pour objet de mémoriser une adresse).

• Ainsi, on peut définir des variables de type pointeur, censées contenir des adresses.

• Il existe une corrélation très étroite entre la notion de tableau et celle de pointeur :

- un nom de tableau peut être assimilable à un pointeur sur son premier élément.

• Déclaration :

```
type * identificateur;
```

• Manipulation :

```
&_ :  adresse de_
*_ :  valeur à l'adresse _ ou valeur pointée par _
```

Déclaration et affectation

• Considérons la déclaration suivante :
`int *adr ;` /* on peut également écrire : `int * adr ;` */

• Cette déclaration précise que `adr` est une variable de type pointeur sur des objets de type `int`.

• Pour assigner une valeur à `adr` et donc la faire pointer sur un entier précis, on peut affecter directement à `adr` l'adresse d'un objet déjà existant :

```
int x = 10;
int tab[15];

adr = &x;

ou encore

adr = &tab[6];
```

• On peut également utiliser une allocation dynamique pour créer un emplacement mémoire. Par exemple :

```
adr = malloc (sizeof (int));
```

• Une autre démarche consiste à affecter une valeur d'une variable pointeur à une autre variable pointeur :

```
-------------------------------------------------
int *adr1, *adr2, *adr3;
...
adr1 = adr2; /* adr1 et adr2 pointent désormais sur
                le même objet */
-------------------------------------------------
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); =>
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94;
printf("%d\n", *adr); =>
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); =>
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5;
printf("%d\n", *adr); =>
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5; //équivalent à *adr = *adr + 5
printf("%d\n", *adr); => affiche 99
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5; //équivalent à *adr = *adr + 5
printf("%d\n", *adr); => affiche 99
printf("%p %p\n", adr, &x); =>
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5; //équivalent à *adr = *adr + 5
printf("%d\n", *adr); => affiche 99
printf("%p %p\n", adr, &x); => affiche 0x7ffe5c9b4434 0x7ffe5c9b4434
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5; //équivalent à *adr = *adr + 5
printf("%d\n", *adr); => affiche 99
printf("%p %p\n", adr, &x); => affiche 0x7ffe5c9b4434 0x7ffe5c9b4434
printf("%p\n", &adr); =>
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5; //équivalent à *adr = *adr + 5
printf("%d\n", *adr); => affiche 99
printf("%p %p\n", adr, &x); => affiche 0x7ffe5c9b4434 0x7ffe5c9b4434
printf("%p\n", &adr); => affiche 0x7ffe5c9b4438
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5; //équivalent à *adr = *adr + 5
printf("%d\n", *adr); => affiche 99
printf("%p %p\n", adr, &x); => affiche 0x7ffe5c9b4434 0x7ffe5c9b4434
printf("%p\n", &adr); => affiche 0x7ffe5c9b4438
printf("%d %d\n", *(*(&adr)), *(&x)); =>
```

Manipulation

Le pointeur sert bien sur à la manipulation d'adresse, mais il permet également d'effectuer des opérations classiques sur l'objet pointé.

Exemple :

```
int * adr;
int x = 14;
adr = &x;
printf("%d\n", *adr); => affiche 14
*adr = 94; //équivalent à x = 94;
printf("%d\n", *adr); => affiche 94
printf("%d\n", x); => affiche 94
*adr += 5; //équivalent à *adr = *adr + 5
printf("%d\n", *adr); => affiche 99
printf("%p %p\n", adr, &x); => affiche 0x7ffe5c9b4434 0x7ffe5c9b4434
printf("%p\n", &adr); => affiche 0x7ffe5c9b4438
printf("%d %d\n", *(*(&adr)), *(&x)); =>affiche 99 99
```

Le pointeur null

• Il existe un symbole noté `NULL`, dont la valeur représente conventionnellement un pointeur ne pointant sur rien, c'est-à-dire associé à aucune adresse.

• Cette valeur peut être affectée à un pointeur de tout type. Par exemple :

```
-------------------------------------------------
int * adr;
adr = NULL; /* par précaution */
if (adr != NULL)
*adr= .../* ici on est sûr que adr a reçu une valeur */
-------------------------------------------------
```

• **NB :** cette valeur `NULL` ne doit pas être confondue avec une valeur de pointeur indéfinie. On peut tester l'égalité (==) ou la différence (!=) de n'importe quel pointeur avec `NULL`.

Lien entre pointeurs et tableaux

- Principe :
  - Nom du tableau = adresse du premier élément
  - tableau[i] $\Longleftrightarrow$ *(tableau+i)

- Par exemple avec `int TAB[15]`

`TAB[i]` est en fait interprété comme suit :

  - `TAB` est d'abord converti en un pointeur sur `TAB[0]`
  - on ajoute ensuite *i* au résultat pour considérer l'objet pointé
  - `TAB[i]` est donc bien équivalent à `*(TAB+i)`

Exemple :

```c
#include<stdio.h>

void main() /* Fonction principale */
{
int TAB[5] = {51, 22, 33, 66, 97};
int *pt;

printf("TAB[3] = %d\n", TAB[3]);
printf("*(TAB+3) = %d\n", *(TAB+3));

pt = TAB;
pt = TAB+2;
printf("pt = %d\n", *pt);
pt = pt+1;
printf("pt = %d\n", *pt);
}
```

```
TAB[3] = 66
*(TAB+3) = 66
pt = 33
pt = 66
```

Pointeurs & Fonctions

- Rappel : en C le passage des arguments se fait par valeur.
- Et si la valeur est une adresse ? Cela devient un passage d'argument par référence (*i.e.* par adresse).

Exemple 1 :

```c
#include<stdio.h>

void mafonction(int * val)
{
 int temp = 94;
 val = &temp;
 printf("(pendant appel) val = %d\n", *val);
}


void main() /* Fonction principale */
{
 int A = 200;
 printf("(avant appel) A = %d\n", A);
 mafonction(&A);
 printf("(apres) A = %d\n", A);
}
```

```
(avant appel) A = 200
(pendant appel) val = 94
(apres) A = 200
```

Exemple 2 :

```c
#include <stdio.h>

void mafonction(int * val)
{
 int temp = 94;
 //val = &temp;
 *val = temp;
 printf("(pendant appel) val = %d\n", *val);
}

int main(int argc, char *argv[]) /* Fonction principale */
{
 int A = 200;
 printf("(Avant appel) A = %d\n", A);
 mafonction(&A);
 printf("(apres) A = %d\n", A);
 return 0;
}
```

```
(Avant appel) A = 200
(pendant appel) val = 94
(apres) A = 94
```

NB : on remarque ici quand on revient dans la fonction `main`, *A* conserve la valeur 94 affectée dans `mafonction`. Ceci met en avant l'utilité des pointeurs en C.

Bibliographie

• C. DELANNOY, Le guide complet du langage C, éditions EYROLLES, Octobre 2014.

• C. DELANNOY, Langage C, éditions EYROLLES, 4ème tirage 2005.

• D. Defour, "Programmation en C", Univ. de Perpignan Via Domitia.

• M. François, "Sécurité des Applications (Bonnes pratiques du langage C)", STI 5A-EO, INSA Centre Val de Loire, Oct. 2013.

• J. F. Lalande, "Programmation C", INSA Centre val de Loire, 13 Nov. 2012.

• B. W. Kernighan, D. M. Ritchie, "Le langage C (Norme ANSI)", DUNOD, 2ème édition, 2004.