Introduction à l'algorithmique

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique

23 septembre 2014

Que signifie le mot « programmer » ?

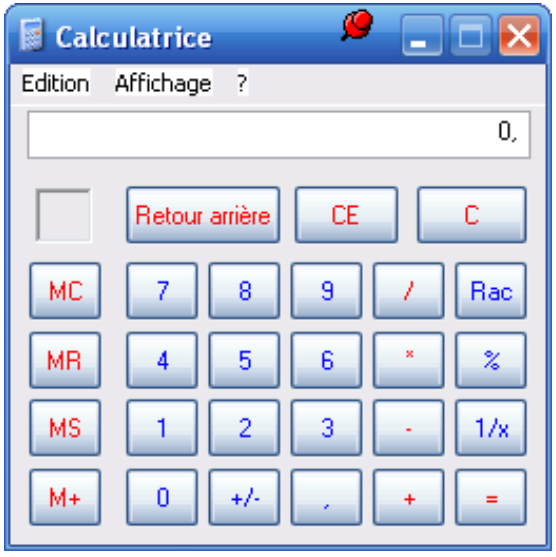
Programmer signifie réaliser des « programmes informatiques ». Les programmes demandent à l'ordinateur d'effectuer des actions.

- la calculatrice est un programme ;
- votre traitement de texte est un programme ;
- votre logiciel de « chat » est un programme ;
- les jeux vidéo sont des programmes.

Que signifie le mot « programmer » ?

Programmer signifie réaliser des « programmes informatiques ». Les programmes demandent à l'ordinateur d'effectuer des actions.

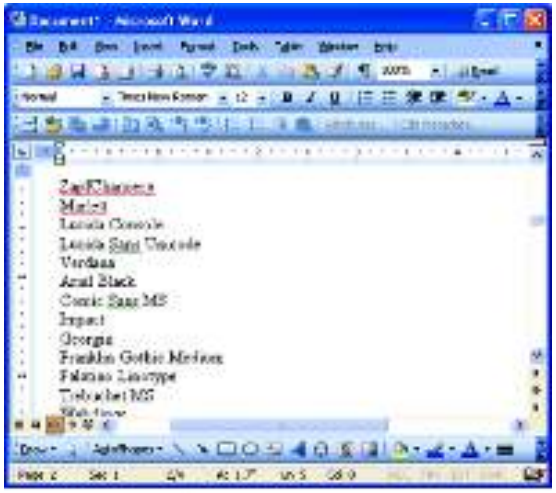
- la calculatrice est un programme ;
- votre traitement de texte est un programme ;
- votre logiciel de « chat » est un programme ;
- les jeux vidéo sont des programmes.



Que signifie le mot « programmer » ?

Programmer signifie réaliser des « programmes informatiques ». Les programmes demandent à l'ordinateur d'effectuer des actions.

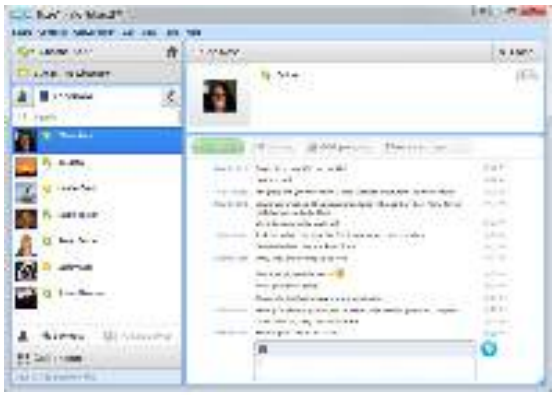
- la calculatrice est un programme ;
- votre traitement de texte est un programme ;
- votre logiciel de « chat » est un programme ;
- les jeux vidéo sont des programmes.



Que signifie le mot « programmer » ?

Programmer signifie réaliser des « programmes informatiques ». Les programmes demandent à l'ordinateur d'effectuer des actions.

- la calculatrice est un programme ;
- votre traitement de texte est un programme ;
- votre logiciel de « chat » est un programme ;
- les jeux vidéo sont des programmes.



Que signifie le mot « programmer » ?

Programmer signifie réaliser des « programmes informatiques ». Les programmes demandent à l'ordinateur d'effectuer des actions.

- la calculatrice est un programme ;
- votre traitement de texte est un programme ;
- votre logiciel de « chat » est un programme ;
- les jeux vidéo sont des programmes.

Programmer oui ! mais avec quel langage ?

- L'ordinateur ne comprend que le langage informatique. Par exemple, l'instruction « Fais le calcul 4 + 7 » se traduit en langage informatique par : **00101101100100110100111110**
- Ce langage informatique est appelé langage **binaire**
- Ce langage binaire est **incompréhensible**.
- L'ordinateur ne parle pas l'anglais ou le français, et encore moins l'arabe.

Problème

Comment parler à l'ordinateur plus simplement qu'en binaire ?

Programmer oui ! mais avec quel langage ?

- L'ordinateur ne comprend que le langage informatique. Par exemple, l'instruction « Fais le calcul 4 + 7 » se traduit en langage informatique par : **00101101100100110100111110**
- Ce langage informatique est appelé langage **binaire**
- Ce langage binaire est **incompréhensible**.
- L'ordinateur ne parle pas l'anglais ou le français, et encore moins l'arabe.

Problème

Comment parler à l'ordinateur plus simplement qu'en binaire ?

Programmer oui ! mais avec quel langage ?

- L'ordinateur ne comprend que le langage informatique. Par exemple, l'instruction « Fais le calcul 4 + 7 » se traduit en langage informatique par : **00101101100100110100111110**
- Ce langage informatique est appelé langage **binaire**
- Ce langage binaire est **incompréhensible**.
- L'ordinateur ne parle pas l'anglais ou le français, et encore moins l'arabe.

Problème

Comment parler à l'ordinateur plus simplement qu'en binaire ?

Programmer oui ! mais avec quel langage ?

- L'ordinateur ne comprend que le langage informatique. Par exemple, l'instruction « Fais le calcul 4 + 7 » se traduit en langage informatique par : **00101101100100110100111110**
- Ce langage informatique est appelé langage **binaire**
- Ce langage binaire est **incompréhensible**.
- L'ordinateur ne parle pas l'anglais ou le français, et encore moins l'arabe.

Problème

Comment parler à l'ordinateur plus simplement qu'en binaire ?

Programmer oui ! mais avec quel langage ?

- L'ordinateur ne comprend que le langage informatique. Par exemple, l'instruction « Fais le calcul 4 + 7 » se traduit en langage informatique par : **00101101100100110100111110**
- Ce langage informatique est appelé langage **binaire**
- Ce langage binaire est **incompréhensible**.
- L'ordinateur ne parle pas l'anglais ou le français, et encore moins l'arabe.

Problème

Comment parler à l'ordinateur plus simplement qu'en binaire ?

Les langages de programmation

- Inventer de nouveaux langages qui seraient ensuite traduits en binaire pour l'ordinateur.
- Le plus dur à faire, c'est de réaliser le programme qui fait la « traduction ».
- Ce programme a déjà été écrit par des informaticiens et nous n'aurons pas à le refaire.
- On va s'en servir pour écrire des phrases comme : « Fais le calcul 3 + 5 » qui seront traduites par le programme de « traduction » en quelque chose comme : « 00101101100100110100111110 ».

Les langages de programmation

- Inventer de nouveaux langages qui seraient ensuite traduits en binaire pour l'ordinateur.
- Le plus dur à faire, c'est de réaliser le programme qui fait la « traduction ».
- Ce programme a déjà été écrit par des informaticiens et nous n'aurons pas à le refaire.
- On va s'en servir pour écrire des phrases comme : « Fais le calcul 3 + 5 » qui seront traduites par le programme de « traduction » en quelque chose comme : « 00101101100100110100111110 ».

Les langages de programmation

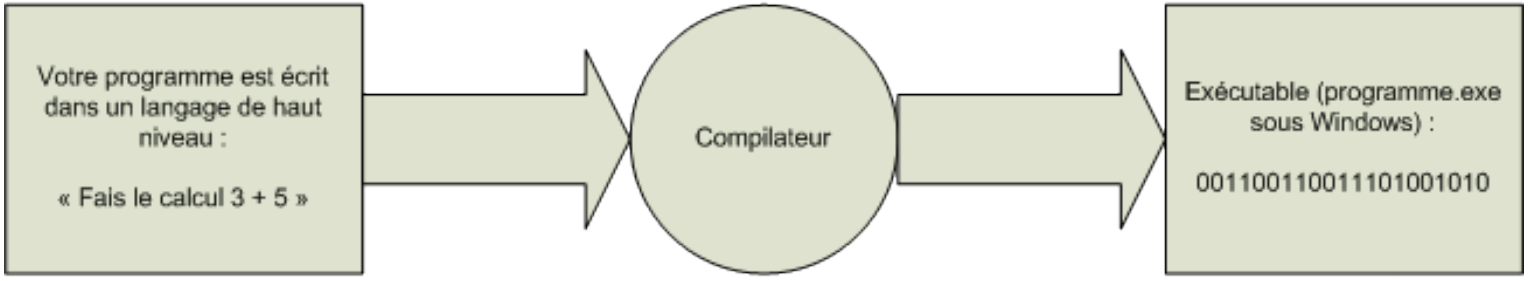
- Inventer de nouveaux langages qui seraient ensuite traduits en binaire pour l'ordinateur.
- Le plus dur à faire, c'est de réaliser le programme qui fait la « traduction ».
- Ce programme a déjà été écrit par des informaticiens et nous n'aurons pas à le refaire.
- On va s'en servir pour écrire des phrases comme : « Fais le calcul 3 + 5 » qui seront traduites par le programme de « traduction » en quelque chose comme : « 00101101100100110100111110 ».

Les langages de programmation

- Inventer de nouveaux langages qui seraient ensuite traduits en binaire pour l'ordinateur.
- Le plus dur à faire, c'est de réaliser le programme qui fait la « traduction ».
- Ce programme a déjà été écrit par des informaticiens et nous n'aurons pas à le refaire.
- On va s'en servir pour écrire des phrases comme : « Fais le calcul 3 + 5 » qui seront traduites par le programme de « traduction » en quelque chose comme : « 00101101100100110100111110 ».

Les langages de programmation

- Inventer de nouveaux langages qui seraient ensuite traduits en binaire pour l'ordinateur.
- Le plus dur à faire, c'est de réaliser le programme qui fait la « traduction ».
- Ce programme a déjà été écrit par des informaticiens et nous n'aurons pas à le refaire.
- On va s'en servir pour écrire des phrases comme : « Fais le calcul 3 + 5 » qui seront traduites par le programme de « traduction » en quelque chose comme : « 00101101100100110100111110 ».



Pourquoi programmer en C ?

- Il existe de nombreux langages de plus ou moins haut niveau en informatique tels que le C, le C++, Java, Visual Basic, Delphi, etc.
- le C est un langage très populaire.
- Il permet de vous donner de solides connaissances sur la programmation et le fonctionnement de votre ordinateur.
- Il vous permet d'être ensuite largement capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.
- Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.
- Il est très fréquent qu'il soit enseigné lors d'études supérieures en informatique.

Pourquoi programmer en C ?

- Il existe de nombreux langages de plus ou moins haut niveau en informatique tels que le C, le C++, Java, Visual Basic, Delphi, etc.
- le C est un langage très populaire.
- Il permet de vous donner de solides connaissances sur la programmation et le fonctionnement de votre ordinateur.
- Il vous permet d'être ensuite largement capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.
- Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.
- Il est très fréquent qu'il soit enseigné lors d'études supérieures en informatique.

Pourquoi programmer en C ?

- Il existe de nombreux langages de plus ou moins haut niveau en informatique tels que le C, le C++, Java, Visual Basic, Delphi, etc.
- le C est un langage très populaire.
- Il permet de vous donner de solides connaissances sur la programmation et le fonctionnement de votre ordinateur.
- Il vous permet d'être ensuite largement capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.
- Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.
- Il est très fréquent qu'il soit enseigné lors d'études supérieures en informatique.

Pourquoi programmer en C ?

- Il existe de nombreux langages de plus ou moins haut niveau en informatique tels que le C, le C++, Java, Visual Basic, Delphi, etc.
- le C est un langage très populaire.
- Il permet de vous donner de solides connaissances sur la programmation et le fonctionnement de votre ordinateur.
- Il vous permet d'être ensuite largement capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.
- Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.
- Il est très fréquent qu'il soit enseigné lors d'études supérieures en informatique.

Pourquoi programmer en C ?

- Il existe de nombreux langages de plus ou moins haut niveau en informatique tels que le C, le C++, Java, Visual Basic, Delphi, etc.
- le C est un langage très populaire.
- Il permet de vous donner de solides connaissances sur la programmation et le fonctionnement de votre ordinateur.
- Il vous permet d'être ensuite largement capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.
- Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.
- Il est très fréquent qu'il soit enseigné lors d'études supérieures en informatique.

Pourquoi programmer en C ?

- Il existe de nombreux langages de plus ou moins haut niveau en informatique tels que le C, le C++, Java, Visual Basic, Delphi, etc.
- le C est un langage très populaire.
- Il permet de vous donner de solides connaissances sur la programmation et le fonctionnement de votre ordinateur.
- Il vous permet d'être ensuite largement capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.
- Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.
- Il est très fréquent qu'il soit enseigné lors d'études supérieures en informatique.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? NON
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? NON
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

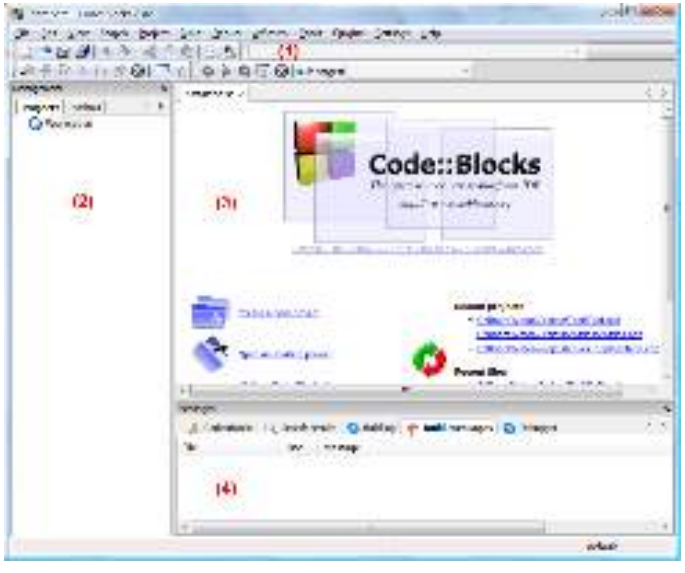
- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

Programmer : est-ce difficile ?

- faut-il être un surdoué ou un génie pour pouvoir commencer la programmation ? **NON**
- faut-il être un super-mathématicien pour pouvoir commencer la programmation ? **NON**
- Il suffit juste de savoir comment un ordinateur fonctionne.
- Mais notez qu'un programmeur a aussi certaines qualités comme :
  - **la patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
  - **le sens de la logique** : pas besoin d'être forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir.
  - **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. Ce n'est pas ça qui fera marcher votre programme.

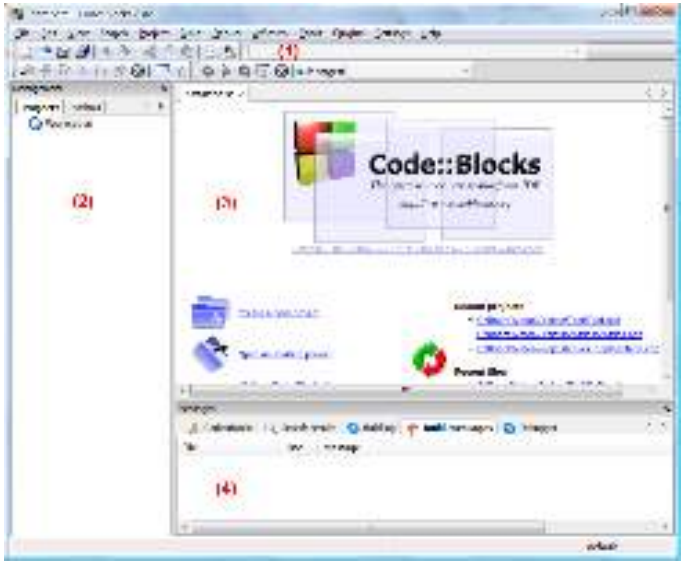
De quoi a-t-on besoin pour programmer en C ?

- Les programmeurs ont besoin de trois outils : un éditeur de texte, un compilateur et un débogueur.
- Il est possible d'installer ces outils séparément, mais il est courant aujourd'hui d'avoir un package trois-en-un que l'on appelle **IDE**, l'environnement de développement.
- Code : :Blocks, Visual C++ et Xcode comptent parmi les IDE les plus célèbres.



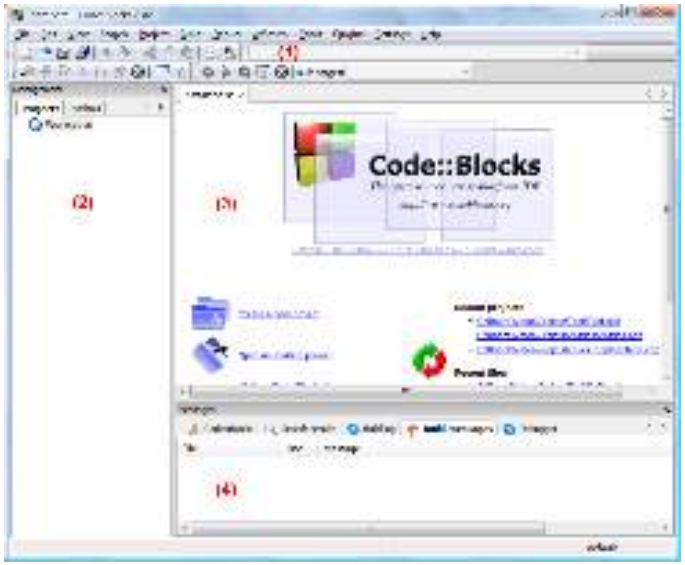
De quoi a-t-on besoin pour programmer en C ?

- Les programmeurs ont besoin de trois outils : un éditeur de texte, un compilateur et un débogueur.
- Il est possible d'installer ces outils séparément, mais il est courant aujourd'hui d'avoir un package trois-en-un que l'on appelle **IDE**, l'environnement de développement.
- Code : :Blocks, Visual C++ et Xcode comptent parmi les IDE les plus célèbres.



De quoi a-t-on besoin pour programmer en C ?

- Les programmeurs ont besoin de trois outils : un éditeur de texte, un compilateur et un débogueur.
- Il est possible d'installer ces outils séparément, mais il est courant aujourd'hui d'avoir un package trois-en-un que l'on appelle **IDE**, l'environnement de développement.
- Code : :Blocks, Visual C++ et Xcode comptent parmi les IDE les plus célèbres.



Mon premier programme en C

Exemple de programme en C

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

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

Les commentaires

Exemple :

```
/* Ci-dessous, ce sont des directives de preprocesseur. Ces
    lignes permettent d'ajouter des fichiers au projet,
fichiers que l'on appelle bibliotheques. Grace a ces
    bibliotheques, on disposera de fonctions toutes pretes
    pour afficher
par exemple un message a l'ecran. */

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

/* Ci-dessous, vous avez la fonction principale du programme
    , appelee main. C'est par cette fonction que tous les
    programmes commencent. Ici, ma fonction se contente d'
    afficher Bonjour a l'ecran. */

int main()
{
  printf("Bonjour"); // Cette instruction affiche Bonjour a
      l'ecran
  return 0;    // Le programme renvoie le nombre 0 puis s'
      arrete
}
```

Initiation à l'algorithmique
Les chaînes de caractères

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

Le type Char

- Le type char permet de stocker des nombres allant de -128 à 127.
- Un caractère est une variable de type Char qui prend 1 octet.
- Comme la mémoire ne peut stocker que des nombres, on a inventé une table qui fait la conversion entre les nombres et les lettres. Ainsi par exemple que le nombre 65 équivaut à la lettre A.
- En C, on peut travailler sur un caractère à partir de son numéro dans la table ASCII, il suffit d'écrire cette lettre entre apostrophes, comme ceci : 'A'. 'A' sera donc remplacé par la valeur correspondante : 65.

Exemple

```c
int main()
{
    char lettre = 'A';
    printf("%d\n", lettre);
    return 0;
}
```

Le type Char

- Le type char permet de stocker des nombres allant de -128 à 127.
- Un caractère est une variable de type Char qui prend 1 octet.
- Comme la mémoire ne peut stocker que des nombres, on a inventé une table qui fait la conversion entre les nombres et les lettres. Ainsi par exemple que le nombre 65 équivaut à la lettre A.
- En C, on peut travailler sur un caractère à partir de son numéro dans la table ASCII, il suffit d'écrire cette lettre entre apostrophes, comme ceci : 'A'. 'A' sera donc remplacé par la valeur correspondante : 65.

Exemple

```
int main()
{
    char lettre = 'A';
    printf("%d\n", lettre);
    return 0;
}
```

Le type Char

- Le type char permet de stocker des nombres allant de -128 à 127.
- Un caractère est une variable de type Char qui prend 1 octet.
- Comme la mémoire ne peut stocker que des nombres, on a inventé une table qui fait la conversion entre les nombres et les lettres. Ainsi par exemple que le nombre 65 équivaut à la lettre A.
- En C, on peut travailler sur un caractère à partir de son numéro dans la table ASCII, il suffit d'écrire cette lettre entre apostrophes, comme ceci : 'A'. 'A' sera donc remplacé par la valeur correspondante : 65.

Exemple

```c
int main()
{
    char lettre = 'A';
    printf("%d\n", lettre);
    return 0;
}
```

Le type Char

- Le type char permet de stocker des nombres allant de -128 à 127.
- Un caractère est une variable de type Char qui prend 1 octet.
- Comme la mémoire ne peut stocker que des nombres, on a inventé une table qui fait la conversion entre les nombres et les lettres. Ainsi par exemple que le nombre 65 équivaut à la lettre A.
- En C, on peut travailler sur un caractère à partir de son numéro dans la table ASCII, il suffit d'écrire cette lettre entre apostrophes, comme ceci : 'A'. 'A' sera donc remplacé par la valeur correspondante : 65.

Exemple

```
int main()
{
    char lettre = 'A';
    printf("%d\n", lettre);
    return 0;
}
```

Le type Char

- Le type char permet de stocker des nombres allant de -128 à 127.
- Un caractère est une variable de type Char qui prend 1 octet.
- Comme la mémoire ne peut stocker que des nombres, on a inventé une table qui fait la conversion entre les nombres et les lettres. Ainsi par exemple que le nombre 65 équivaut à la lettre A.
- En C, on peut travailler sur un caractère à partir de son numéro dans la table ASCII, il suffit d'écrire cette lettre entre apostrophes, comme ceci : 'A'. 'A' sera donc remplacé par la valeur correspondante : 65.

Exemple

```
int main()
{
    char lettre = 'A';
    printf("%d\n", lettre);
    return 0;
}
```

Table ASCI

- La table ASCII est un tableau de 256 caractères, numérotés de 0 à 255.
- La plupart des caractères « de base » sont codés entre les nombres 0 et 127.

Le code ASCII

American Standard Code for Information Interchange

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de línea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift out)
15	0Fh	SI	(shift in)
16	10h	DLE	(data link escape)
17	11h	DC1	(dispositivo control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(dispositivo control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	7Fh	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	≡
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Table ASCI

- La table ASCII est un tableau de 256 caractères, numérotés de 0 à 255.
- La plupart des caractères « de base » sont codés entre les nombres 0 et 127.

Le code ASCII

American Standard Code for Information Interchange

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de línea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(dispositivo control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(dispositivo control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	7Fh	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	≡
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Lire et afficher un caractère

- On peut demander à l'utilisateur d'entrer une lettre en utilisant le %c dans un scanf (c comme caractère).
- Pour afficher un caractère on doit également utiliser le symbole %c :

Exemple

```c
int main()
{
    char lettre = 0;

    scanf("%c", &lettre);
    printf("%c\n", lettre);

    return 0;
}
```

Lire et afficher un caractère

- On peut demander à l'utilisateur d'entrer une lettre en utilisant le %c dans un scanf (c comme caractère).
- Pour afficher un caractère on doit également utiliser le symbole %c :

Exemple

```c
int main()
{
    char lettre = 0;

    scanf("%c", &lettre);
    printf("%c\n", lettre);

    return 0;
}
```

Lire et afficher un caractère

- On peut demander à l'utilisateur d'entrer une lettre en utilisant le %c dans un scanf (c comme caractère).
- Pour afficher un caractère on doit également utiliser le symbole %c :

Exemple

```c
int main()
{
    char lettre = 0;

    scanf("%c", &lettre);
    printf("%c\n", lettre);

    return 0;
}
```

Les chaînes de caractères

Une chaîne de caractères n'est rien d'autre qu'un tableau de type char.

Adresse	Valeur
18000	'S'
18001	'a'
18002	'l'
18003	'u'
18004	't'

Le symbole de fin de chaîne de caractères

Une chaîne de caractère doit impérativement contenir un caractère spécial à la fin de la chaîne, appelé « caractère de fin de chaîne ». Ce caractère s'écrit '\0'.

Adresse	Valeur
18000	'S'
18001	'a'
18002	'l'
18003	'u'
18004	't'
18005	'\0'

Le symbole de fin de chaîne de caractères

- Le caractère '\0' permet tout simplement d'indiquer la fin la chaîne.
- Par conséquent, pour stocker le mot « Salut » (qui comprend 5 lettres) en mémoire, il ne faut pas un tableau de 5 char, mais de 6 !
- Chaque fois que vous créez une chaîne de caractères, vous allez donc devoir penser à prévoir de la place pour le caractère de fin de chaîne. Il faut toujours toujours ajouter un bloc de plus dans le tableau pour stocker ce caractère '\0', c'est impératif !
- Oublier le caractère de fin '\0' est une source d'erreurs impitoyable du langage C.

Le symbole de fin de chaîne de caractères

- Le caractère '\0' permet tout simplement d'indiquer la fin la chaîne.
- Par conséquent, pour stocker le mot « Salut » (qui comprend 5 lettres) en mémoire, il ne faut pas un tableau de 5 char, mais de 6 !
- Chaque fois que vous créez une chaîne de caractères, vous allez donc devoir penser à prévoir de la place pour le caractère de fin de chaîne. Il faut toujours toujours ajouter un bloc de plus dans le tableau pour stocker ce caractère '\0', c'est impératif !
- Oublier le caractère de fin '\0' est une source d'erreurs impitoyable du langage C.

Le symbole de fin de chaîne de caractères

- Le caractère '\0' permet tout simplement d'indiquer la fin la chaîne.
- Par conséquent, pour stocker le mot « Salut » (qui comprend 5 lettres) en mémoire, il ne faut pas un tableau de 5 char, mais de 6 !
- Chaque fois que vous créez une chaîne de caractères, vous allez donc devoir penser à prévoir de la place pour le caractère de fin de chaîne. Il faut toujours toujours ajouter un bloc de plus dans le tableau pour stocker ce caractère '\0', c'est impératif !
- Oublier le caractère de fin '\0' est une source d'erreurs impitoyable du langage C.

Le symbole de fin de chaîne de caractères

- Le caractère '\0' permet tout simplement d'indiquer la fin la chaîne.
- Par conséquent, pour stocker le mot « Salut » (qui comprend 5 lettres) en mémoire, il ne faut pas un tableau de 5 char, mais de 6 !
- Chaque fois que vous créez une chaîne de caractères, vous allez donc devoir penser à prévoir de la place pour le caractère de fin de chaîne. Il faut toujours toujours ajouter un bloc de plus dans le tableau pour stocker ce caractère '\0', c'est impératif !
- Oublier le caractère de fin '\0' est une source d'erreurs impitoyable du langage C.

Le symbole de fin de chaîne de caractères

Grâce au caractère '\0' :

- vous n'aurez pas à retenir la taille de votre tableau car il indique que le tableau s'arrête à cet endroit.
- vous pourrez passer votre tableau de char à une fonction sans avoir à ajouter à côté une variable indiquant la taille du tableau.
- cela n'est valable que pour les chaînes de caractères (c'est-à-dire le type char*, qu'on peut aussi écrire char[]).
- pour les autres types de tableaux, vous êtes toujours obligés de retenir la taille du tableau quelque part.

Le symbole de fin de chaîne de caractères

Grâce au caractère '\0' :

- vous n'aurez pas à retenir la taille de votre tableau car il indique que le tableau s'arrête à cet endroit.
- vous pourrez passer votre tableau de char à une fonction sans avoir à ajouter à côté une variable indiquant la taille du tableau.
- cela n'est valable que pour les chaînes de caractères (c'est-à-dire le type char*, qu'on peut aussi écrire char[]).
- pour les autres types de tableaux, vous êtes toujours obligés de retenir la taille du tableau quelque part.

Le symbole de fin de chaîne de caractères

Grâce au caractère '\0' :

- vous n'aurez pas à retenir la taille de votre tableau car il indique que le tableau s'arrête à cet endroit.
- vous pourrez passer votre tableau de char à une fonction sans avoir à ajouter à côté une variable indiquant la taille du tableau.
- cela n'est valable que pour les chaînes de caractères (c'est-à-dire le type char*, qu'on peut aussi écrire char[]).
- pour les autres types de tableaux, vous êtes toujours obligés de retenir la taille du tableau quelque part.

Le symbole de fin de chaîne de caractères

Grâce au caractère '\0' :

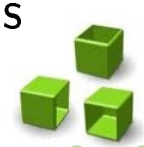
- vous n'aurez pas à retenir la taille de votre tableau car il indique que le tableau s'arrête à cet endroit.
- vous pourrez passer votre tableau de char à une fonction sans avoir à ajouter à côté une variable indiquant la taille du tableau.
- cela n'est valable que pour les chaînes de caractères (c'est-à-dire le type char*, qu'on peut aussi écrire char[]).
- pour les autres types de tableaux, vous êtes toujours obligés de retenir la taille du tableau quelque part.

Création et initialisation de la chaîne

- Si on veut initialiser notre tableau chaine avec le texte « Salut », on peut utiliser la méthode manuelle mais peu efficace :

```c
int main()
{
    char chaine[6];

    chaine[0] = 'S';
    chaine[1] = 'a';
    chaine[2] = 'l';
    chaine[3] = 'u';
    chaine[4] = 't';
    chaine[5] = '\0';
    // Affichage de la chaine grace au %s du printf
    printf("%s", chaine);
    return 0;
}
```

- Pour afficher une chaine de caractère, il faut utiliser le symbole%s (s comme string, qui signifie « chaîne » en anglais) dans la fonction printf.



Création et initialisation de la chaîne

- En tapant entre guillemets la chaîne que vous voulez mettre dans votre tableau, le compilateur C calcule automatiquement la taille nécessaire. C'est-à-dire qu'il compte les lettres et ajoute 1 pour placer le caractère '\0'.
- Il écrit ensuite une à une les lettres du mot « Salut » en mémoire et ajoute le '\0' comme on l'a fait nous-mêmes manuellement.

```c
int main()
{
    char chaine[] = "Salut"; // La taille du tableau chaine
        est automatiquement calculee

    printf("%s", chaine);

    return 0;
}
```

- Il y a toutefois un défaut : ça ne marche que pour l'initialisation ! Vous ne pouvez pas écrire plus loin dans le code : chaine = "Salut" ;

Création et initialisation de la chaîne

- En tapant entre guillemets la chaîne que vous voulez mettre dans votre tableau, le compilateur C calcule automatiquement la taille nécessaire. C'est-à-dire qu'il compte les lettres et ajoute 1 pour placer le caractère '\0'.
- Il écrit ensuite une à une les lettres du mot « Salut » en mémoire et ajoute le '\0' comme on l'a fait nous-mêmes manuellement.

```c
int main()
{
    char chaine[] = "Salut"; // La taille du tableau chaine
        est automatiquement calculee

    printf("%s", chaine);

    return 0;
}
```

- Il y a toutefois un défaut : ça ne marche que pour l'initialisation ! Vous ne pouvez pas écrire plus loin dans le code : chaine = "Salut" ;

Création et initialisation de la chaîne

- En tapant entre guillemets la chaîne que vous voulez mettre dans votre tableau, le compilateur C calcule automatiquement la taille nécessaire. C'est-à-dire qu'il compte les lettres et ajoute 1 pour placer le caractère '\0'.
- Il écrit ensuite une à une les lettres du mot « Salut » en mémoire et ajoute le '\0' comme on l'a fait nous-mêmes manuellement.

```c
int main()
{
    char chaine[] = "Salut"; // La taille du tableau chaine
        est automatiquement calculee

    printf("%s", chaine);

    return 0;
}
```

- Il y a toutefois un défaut : ça ne marche que pour l'initialisation ! Vous ne pouvez pas écrire plus loin dans le code : chaine = "Salut" ;

Création et initialisation de la chaîne

- En tapant entre guillemets la chaîne que vous voulez mettre dans votre tableau, le compilateur C calcule automatiquement la taille nécessaire. C'est-à-dire qu'il compte les lettres et ajoute 1 pour placer le caractère '\0'.
- Il écrit ensuite une à une les lettres du mot « Salut » en mémoire et ajoute le '\0' comme on l'a fait nous-mêmes manuellement.

```c
int main()
{
    char chaine[] = "Salut"; // La taille du tableau chaine
        est automatiquement calculee

    printf("%s", chaine);

    return 0;
}
```

- Il y a toutefois un défaut : ça ne marche que pour l'initialisation ! Vous ne pouvez pas écrire plus loin dans le code : chaine = "Salut" ;

Lire une chaîne de caractères

- On peut enregistrer une chaîne entrée par l'utilisateur via un scanf, en utilisant là encore le symbole %s.
- Seul problème : on ne sait pas combien de caractères l'utilisateur va entrer.
- Il va falloir créer un tableau de char très grand, suffisamment grand !

```
int main()
{
    char prenom[100];

    printf("Comment t'appelles-tu ? ");
    scanf("%s", prenom);
    printf("Salut %s, je suis heureux de te rencontrer !",
        prenom);

    return 0;
}
```

Lire une chaîne de caractères

- On peut enregistrer une chaîne entrée par l'utilisateur via un scanf, en utilisant là encore le symbole %s.
- Seul problème : on ne sait pas combien de caractères l'utilisateur va entrer.
- Il va falloir créer un tableau de char très grand, suffisamment grand !

```c
int main()
{
    char prenom[100];

    printf("Comment t'appelles-tu ? ");
    scanf("%s", prenom);
    printf("Salut %s, je suis heureux de te rencontrer !",
        prenom);

    return 0;
}
```

Lire une chaîne de caractères

- On peut enregistrer une chaîne entrée par l'utilisateur via un scanf, en utilisant là encore le symbole %s.
- Seul problème : on ne sait pas combien de caractères l'utilisateur va entrer.
- Il va falloir créer un tableau de char très grand, suffisamment grand !

```
int main()
{
    char prenom[100];

    printf("Comment t'appelles-tu ? ");
    scanf("%s", prenom);
    printf("Salut %s, je suis heureux de te rencontrer !",
        prenom);

    return 0;
}
```

Fonctions de manipulation des chaînes

- Afin de nous aider un peu à manipuler les chaînes, on nous fournit dans la bibliothèque **string.h** un ensemble de fonctions dédiées aux calculs sur des chaînes.

- Pensez donc à inclure #include <**string.h**> en haut des fichiers .c où vous en avez besoin.

- Si vous ne le faites pas, l'ordinateur ne connaîtra pas ces fonctions car il n'aura pas les prototypes, et la compilation plantera.

Fonctions de manipulation des chaînes

- Afin de nous aider un peu à manipuler les chaînes, on nous fournit dans la bibliothèque **string.h** un ensemble de fonctions dédiées aux calculs sur des chaînes.

- Pensez donc à inclure #include <**string.h**> en haut des fichiers .c où vous en avez besoin.

- Si vous ne le faites pas, l'ordinateur ne connaîtra pas ces fonctions car il n'aura pas les prototypes, et la compilation plantera.

Fonctions de manipulation des chaînes

- Afin de nous aider un peu à manipuler les chaînes, on nous fournit dans la bibliothèque **string.h** un ensemble de fonctions dédiées aux calculs sur des chaînes.

- Pensez donc à inclure #include <**string.h**> en haut des fichiers .c où vous en avez besoin.

- Si vous ne le faites pas, l'ordinateur ne connaîtra pas ces fonctions car il n'aura pas les prototypes, et la compilation plantera.

strlen : calculer la longueur d'une chaîne

strlen est une fonction qui calcule la longueur d'une chaîne de caractères (sans compter le caractère '\0').

Exemple

```
int main()
{
    char chaine[] = "Salut";
    int longueurChaine = 0;

    // On recupere la longueur de la chaine dans
        longueurChaine
    longueurChaine = strlen(chaine);

    // On affiche la longueur de la chaine
    printf("La chaine %s fait %d caracteres de long", chaine
        , longueurChaine);

    return 0;
}
```

strlen : calculer la longueur d'une chaîne

```c
int longueurChaine(const char* chaine);
int main()
{
    char chaine[] = "Salut";
    int longueur = 0;
    longueur = longueurChaine(chaine);
    printf("La chaine %s fait %d caracteres de long", chaine
        , longueur);
    return 0;
}
int longueurChaine(const char* chaine)
{
    int nombreDeCaracteres = 0;
    char caractereActuel = 0;
    do
    {
        caractereActuel = chaine[nombreDeCaracteres];
        nombreDeCaracteres++;
    }while(caractereActuel != '\0');
    nombreDeCaracteres--;
    return nombreDeCaracteres;
}
```

strcpy : copier une chaîne dans une autre

La fonction strcpy (comme « string copy ») permet de copier une chaîne à l'intérieur d'une autre.

```c
int main()
{
    char chaine[] = "Texte", copie[100] = {0};
    strcpy(copie, chaine); // On copie "chaine" dans "copie"
    printf("chaine vaut : %s\n", chaine);
    printf("copie vaut : %s\n", copie);
    return 0;
}
```

Cette fonction prend deux paramètres :

- copie : c'est un pointeur vers un char* (tableau de char). C'est dans ce tableau que la chaîne sera copiée ;
- chaine : c'est un pointeur vers un autre tableau de char. Cette chaîne sera copiée dans copie.

La fonction renvoie un pointeur sur copie, ce qui n'est pas très utile. En général, on ne récupère pas ce que cette fonction renvoie.

strcpy : copier une chaîne dans une autre

La fonction strcpy (comme « string copy ») permet de copier une chaîne à l'intérieur d'une autre.

```c
int main()
{
    char chaine[] = "Texte", copie[100] = {0};
    strcpy(copie, chaine); // On copie "chaine" dans "copie"
    printf("chaine vaut : %s\n", chaine);
    printf("copie vaut : %s\n", copie);
    return 0;
}
```

Cette fonction prend deux paramètres :

- copie : c'est un pointeur vers un char* (tableau de char). C'est dans ce tableau que la chaîne sera copiée ;
- chaine : c'est un pointeur vers un autre tableau de char. Cette chaîne sera copiée dans copie.

La fonction renvoie un pointeur sur copie, ce qui n'est pas très utile. En général, on ne récupère pas ce que cette fonction renvoie.

strcpy : copier une chaîne dans une autre

La fonction strcpy (comme « string copy ») permet de copier une chaîne à l'intérieur d'une autre.

```c
int main()
{
    char chaine[] = "Texte", copie[100] = {0};
    strcpy(copie, chaine); // On copie "chaine" dans "copie"
    printf("chaine vaut : %s\n", chaine);
    printf("copie vaut : %s\n", copie);
    return 0;
}
```

Cette fonction prend deux paramètres :

- copie : c'est un pointeur vers un char* (tableau de char). C'est dans ce tableau que la chaîne sera copiée ;
- chaine : c'est un pointeur vers un autre tableau de char. Cette chaîne sera copiée dans copie.

La fonction renvoie un pointeur sur copie, ce qui n'est pas très utile. En général, on ne récupère pas ce que cette fonction renvoie.

strcpy : copier une chaîne dans une autre

La fonction strcpy (comme « string copy ») permet de copier une chaîne à l'intérieur d'une autre.

```c
int main()
{
    char chaine[] = "Texte", copie[100] = {0};
    strcpy(copie, chaine); // On copie "chaine" dans "copie"
    printf("chaine vaut : %s\n", chaine);
    printf("copie vaut : %s\n", copie);
    return 0;
}
```

Cette fonction prend deux paramètres :

- copie : c'est un pointeur vers un char* (tableau de char). C'est dans ce tableau que la chaîne sera copiée ;
- chaine : c'est un pointeur vers un autre tableau de char. Cette chaîne sera copiée dans copie.

La fonction renvoie un pointeur sur copie, ce qui n'est pas très utile. En général, on ne récupère pas ce que cette fonction renvoie.

strcpy : copier une chaîne dans une autre

La fonction strcpy (comme « string copy ») permet de copier une chaîne à l'intérieur d'une autre.

```c
int main()
{
    char chaine[] = "Texte", copie[100] = {0};
    strcpy(copie, chaine); // On copie "chaine" dans "copie"
    printf("chaine vaut : %s\n", chaine);
    printf("copie vaut : %s\n", copie);
    return 0;
}
```

Cette fonction prend deux paramètres :

- copie : c'est un pointeur vers un char* (tableau de char). C'est dans ce tableau que la chaîne sera copiée ;
- chaine : c'est un pointeur vers un autre tableau de char. Cette chaîne sera copiée dans copie.

La fonction renvoie un pointeur sur copie, ce qui n'est pas très utile. En général, on ne récupère pas ce que cette fonction renvoie.

strcat : concaténer 2 chaînes

Cette fonction ajoute une chaîne à la suite d'une autre. On appelle cela la concaténation.

```c
int main()
{
    char chaine1[100] = "Toto ", chaine2[] = "Loulou";

    strcat(chaine1, chaine2); // On concatene chaine2 dans
        chaine1
    printf("chaine1 vaut : %s\n", chaine1);
    printf("chaine2 vaut toujours : %s\n", chaine2);
    return 0;
}
```

Supposons que l'on ait les variables suivantes :

- chaine1 = "Toto "
- chaine2 = "Loulou"

Si on concatène chaine2 dans chaine1, alors chaine1 vaudra "Toto Loulou". Quant à chaine2, elle n'aura pas changé et vaudra donc toujours "Loulou". Seule chaine1 est modifiée.

strcat : concaténer 2 chaînes

Cette fonction ajoute une chaîne à la suite d'une autre. On appelle cela la concaténation.

```c
int main()
{
    char chaine1[100] = "Toto ", chaine2[] = "Loulou";

    strcat(chaine1, chaine2); // On concatene chaine2 dans
        chaine1
    printf("chaine1 vaut : %s\n", chaine1);
    printf("chaine2 vaut toujours : %s\n", chaine2);
    return 0;
}
```

Supposons que l'on ait les variables suivantes :

- chaine1 = "Toto "
- chaine2 = "Loulou"

Si on concatène chaine2 dans chaine1, alors chaine1 vaudra "Toto Loulou". Quant à chaine2, elle n'aura pas changé et vaudra donc toujours "Loulou". Seule chaine1 est modifiée.

strcat : concaténer 2 chaînes

Cette fonction ajoute une chaîne à la suite d'une autre. On appelle cela la concaténation.

```c
int main()
{
    char chaine1[100] = "Toto ", chaine2[] = "Loulou";

    strcat(chaine1, chaine2); // On concatene chaine2 dans
        chaine1
    printf("chaine1 vaut : %s\n", chaine1);
    printf("chaine2 vaut toujours : %s\n", chaine2);
    return 0;
}
```

Supposons que l'on ait les variables suivantes :

- chaine1 = "Toto "
- chaine2 = "Loulou"

Si on concatène chaine2 dans chaine1, alors chaine1 vaudra "Toto Loulou". Quant à chaine2, elle n'aura pas changé et vaudra donc toujours "Loulou". Seule chaine1 est modifiée.

strcmp : comparer 2 chaînes

La fonction strcmp compare 2 chaînes entre elles

```c
int main()
{
    char chaine1[] = "Texte de test", chaine2[] = "Texte de
        test";
    if (strcmp(chaine1, chaine2) == 0)
    {
        printf("Les chaines sont identiques\n");
    }
    else
    {
        printf("Les chaines sont differentes\n");
    }
    return 0;
}
```

Il est important de récupérer ce que la fonction renvoie. En effet, strcmp renvoie :

- 0 si les chaînes sont identiques ;
- une autre valeur (positive ou négative) si les chaînes sont différentes.

strcmp : comparer 2 chaînes

La fonction strcmp compare 2 chaînes entre elles

```c
int main()
{
    char chaine1[] = "Texte de test", chaine2[] = "Texte de
        test";
    if (strcmp(chaine1, chaine2) == 0)
    {
        printf("Les chaines sont identiques\n");
    }
    else
    {
        printf("Les chaines sont differentes\n");
    }
    return 0;
}
```

Il est important de récupérer ce que la fonction renvoie. En effet, strcmp renvoie :

- 0 si les chaînes sont identiques ;
- une autre valeur (positive ou négative) si les chaînes sont différentes.

Précédence alphabétique des caractères

- La précédence des caractères dans l'alphabet d'une machine est dépendante du code de caractères utilisé.
  ...,0,1,2,...,9,...,A,B,C,...,Z,...,a,b,c,...,z,...
- Les symboles spéciaux (' ,+ ,- ,/ ,{ ,] ,...) et les lettres accentuées (é ,è ,à ,û ,...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules).
- Leur précédence ne correspond à aucune règle d'ordre spécifique
- On peut déduire une relation de précédence 'est inférieur à' sur l'ensemble des caractères. Ainsi, on peut dire que '0' est inférieur à 'Z' et noter '0' $<$ 'Z'.
- Car le code du caractère '0' (ASCII : 48) est inférieur au code du caractère 'Z' (ASCII : 90).

Précédence alphabétique des caractères

- La précédence des caractères dans l'alphabet d'une machine est dépendante du code de caractères utilisé.
  ...,0,1,2,...,9,...,A,B,C,...,Z,...,a,b,c,...,z,...

- Les symboles spéciaux (' ,+ ,- ,/ ,{ ,] ,...) et les lettres accentuées (é ,è ,à ,û ,...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules).

- Leur précédence ne correspond à aucune règle d'ordre spécifique

- On peut déduire une relation de précédence 'est inférieur à' sur l'ensemble des caractères. Ainsi, on peut dire que '0' est inférieur à 'Z' et noter '0' $<$ 'Z'.

- Car le code du caractère '0' (ASCII : 48) est inférieur au code du caractère 'Z' (ASCII : 90).

Précédence alphabétique des caractères

- La précédence des caractères dans l'alphabet d'une machine est dépendante du code de caractères utilisé.
  ...,0,1,2,...,9,...,A,B,C,...,Z,...,a,b,c,...,z,...

- Les symboles spéciaux (' ,+ ,- ,/ ,{ ,] ,...) et les lettres accentuées (é ,è ,à ,û ,...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules).

- Leur précédence ne correspond à aucune règle d'ordre spécifique

- On peut déduire une relation de précédence 'est inférieur à' sur l'ensemble des caractères. Ainsi, on peut dire que '0' est inférieur à 'Z' et noter '0' $<$ 'Z'.

- Car le code du caractère '0' (ASCII : 48) est inférieur au code du caractère 'Z' (ASCII : 90).

Précédence alphabétique des caractères

- La précédence des caractères dans l'alphabet d'une machine est dépendante du code de caractères utilisé.
  ...,0,1,2,...,9,...,A,B,C,...,Z,...,a,b,c,...,z,...

- Les symboles spéciaux (' ,+ ,- ,/ ,{ ,] ,...) et les lettres accentuées (é ,è ,à ,û ,...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules).

- Leur précédence ne correspond à aucune règle d'ordre spécifique

- On peut déduire une relation de précédence 'est inférieur à' sur l'ensemble des caractères. Ainsi, on peut dire que '0' est inférieur à 'Z' et noter '0' $<$ 'Z'.

- Car le code du caractère '0' (ASCII : 48) est inférieur au code du caractère 'Z' (ASCII : 90).

Précédence alphabétique des caractères

- La précédence des caractères dans l'alphabet d'une machine est dépendante du code de caractères utilisé.
  ...,0,1,2,...,9,...,A,B,C,...,Z,...,a,b,c,...,z,...

- Les symboles spéciaux (' ,+ ,- ,/ ,{ ,] ,...) et les lettres accentuées (é ,è ,à ,û ,...) se trouvent répartis autour des trois grands groupes de caractères (chiffres, majuscules, minuscules).

- Leur précédence ne correspond à aucune règle d'ordre spécifique

- On peut déduire une relation de précédence 'est inférieur à' sur l'ensemble des caractères. Ainsi, on peut dire que '0' est inférieur à 'Z' et noter '0' $<$ 'Z'.

- Car le code du caractère '0' (ASCII : 48) est inférieur au code du caractère 'Z' (ASCII : 90).

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Précédence lexicographique des chaînes de caractères

- La **précédence lexicographique** pour les chaînes de caractères suit l'ordre du dictionnaire et est définie de façon récurrente :
  - La chaîne vide "" précède lexicographiquement toutes les autres chaînes.
  - La chaîne A = "$a_1 a_2 a ... a_p$" (p caractères) précède lexicographiquement la chaîne B = "$b_1 b_2 ... b_m$" (m caractères) si l'une des deux conditions suivantes est remplie :
    1. 'a1' < 'b1'
    2. 'a1' = 'b1' et "$a_2 a_3 ... a_p$" précède lexicographiquement "$b_2 b_3 ... b_m$"
- Exemples :
  1. "ABC" précède "BCD" car 'A'<'B'
  2. "ABC" précède "B" car 'A'<'B'
  3. "Abc" précède "abc" car 'A'<'a'
  4. "ab" précède "abcd" car "" précède "cd"
  5. " ab" précède "ab" car ' '<'a'
- le code ASCII de ' ' est 32, et le code ASCII de 'a' est 97

Exemples d'utilisation de la précédence lexicographique

En tenant compte de l'ordre alphabétique des caractères, on peut contrôler le type du caractère (chiffre, majuscule, minuscule).

```
int main()
{
        char C=' ';
        C=getchar();
        if (C>='0' && C<='9') printf("Chiffre\n", C);
        if (C>='A' && C<='Z') printf("Majuscule\n", C);
        if (C>='a' && C<='z') printf("Minuscule\n", C);
// Il est facile, de convertir des lettres majuscules dans
    des minuscules:
        if (C>='A' && C<='Z') C = C-'A'+'a';
//ou vice-versa:
        if (C>='a' && C<='z') C = C-'a'+'A';
        putchar(C);

return 0;
}
```

strchr : rechercher un caractère

La fonction strchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le premier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```c
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strchr(chaine, 'd');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strchr : rechercher un caractère

La fonction strchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le premier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strchr(chaine, 'd');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strchr : rechercher un caractère

La fonction strchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le premier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```c
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strchr(chaine, 'd');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strchr : rechercher un caractère

La fonction strchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le premier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strchr(chaine, 'd');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strchr : rechercher un caractère

La fonction strrchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le dernier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```c
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strrchr(chaine, 'e');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strchr : rechercher un caractère

La fonction strrchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le dernier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strrchr(chaine, 'e');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strchr : rechercher un caractère

La fonction strrchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le dernier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strrchr(chaine, 'e');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strchr : rechercher un caractère

La fonction strrchr recherche un caractère dans une chaîne. Elle renvoie un pointeur vers le dernier caractère qu'elle a trouvé. Elle renvoie NULL sinon.

```
int main()
{
    char chaine[] = "Texte de test", *suiteChaine = NULL;
    suiteChaine = strrchr(chaine, 'e');
    if (suiteChaine != NULL) // Si on a trouve quelque chose
    {
        printf("Voici la fin de la chaine a partir du
            premier d : %s", suiteChaine);
    }
    return 0;
}
```

La fonction prend 2 paramètres :

- chaine : la chaîne dans laquelle la recherche doit être faite ;
- caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

strstr : rechercher une chaîne dans une autre

Cette fonction recherche la première occurrence d'une chaîne dans une autre chaîne.

```
int main()
{
    char *suiteChaine;

    // On cherche la premiere occurrence de "test" dans "
        Texte de test" :
    suiteChaine = strstr("Texte de test", "test");
    if (suiteChaine != NULL)
    {
        printf("Premiere occurrence de test dans Texte de
            test : %s\n", suiteChaine);
    }

    return 0;
}
```

Elle renvoie, comme les autres, un pointeur quand elle a trouvé ce qu'elle cherchait. Elle renvoie NULL si elle n'a rien trouvé.

strstr : rechercher une chaîne dans une autre

Cette fonction recherche la première occurrence d'une chaîne dans une autre chaîne.

```
int main()
{
    char *suiteChaine;

    // On cherche la premiere occurrence de "test" dans "
        Texte de test" :
    suiteChaine = strstr("Texte de test", "test");
    if (suiteChaine != NULL)
    {
        printf("Premiere occurrence de test dans Texte de
            test : %s\n", suiteChaine);
    }

    return 0;
}
```

Elle renvoie, comme les autres, un pointeur quand elle a trouvé ce qu'elle cherchait. Elle renvoie NULL si elle n'a rien trouvé.

sprintf : écrire dans une chaîne

Cette fonction ressemble énormément au printf que vous connaissez mais, au lieu d'écrire à l'écran, sprintf écrit dans une chaîne !

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char chaine[100];
    int age = 15;
    // On ecrit "Tu as 15 ans" dans chaine
    sprintf(chaine, "Tu as %d ans !", age);
    // On affiche chaine pour verifier qu'elle contient bien
        cela :
    printf("%s", chaine);
    return 0;
}
```

Elle renvoie, comme les autres, un pointeur quand elle a trouvé ce qu'elle cherchait. Elle renvoie NULL si elle n'a rien trouvé.

sprintf : écrire dans une chaîne

Cette fonction ressemble énormément au printf que vous connaissez mais, au lieu d'écrire à l'écran, sprintf écrit dans une chaîne !

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char chaine[100];
    int age = 15;
    // On ecrit "Tu as 15 ans" dans chaine
    sprintf(chaine, "Tu as %d ans !", age);
    // On affiche chaine pour verifier qu'elle contient bien
        cela :
    printf("%s", chaine);
    return 0;
}
```

Elle renvoie, comme les autres, un pointeur quand elle a trouvé ce qu'elle cherchait. Elle renvoie NULL si elle n'a rien trouvé.

Quelques fonctions utiles de <stdio.h>

1. **getchar** : qui lit le prochain caractère du fichier d'entrée standard stdin (le clavier).
2. **putchar(c)** : transfère le caractère c vers le fichier de sortie standard stdout (l'écran).
3. **puts (s)** : écrit la chaîne de caractères désignée par s sur stdout (l'écran) et provoque un retour à la ligne.
4. **gets(s)** : lit une ligne de de caractères du clavier et la copie à l'adresse indiquée par s. Le retour à la ligne final est remplacé par le symbole de fin de chaîne '\0'.

```
void main()
{
    char chaine[100];
    char C=' ';
        gets(chaine);
        puts(chaine);
        C=getchar();
        putchar(C);
}
```

Quelques fonctions utiles de <stdio.h>

1. **getchar** : qui lit le prochain caractère du fichier d'entrée standard stdin (le clavier).
2. **putchar(c)** : transfère le caractère c vers le fichier de sortie standard stdout (l'écran).
3. **puts (s)** : écrit la chaîne de caractères désignée par s sur stdout (l'écran) et provoque un retour à la ligne.
4. **gets(s)** : lit une ligne de de caractères du clavier et la copie à l'adresse indiquée par s. Le retour à la ligne final est remplacé par le symbole de fin de chaîne '\0'.

```
void main()
{
    char chaine[100];
    char C=' ';
        gets(chaine);
        puts(chaine);
        C=getchar();
        putchar(C);
}
```

Quelques fonctions utiles de <stdio.h>

1. **getchar** : qui lit le prochain caractère du fichier d'entrée standard stdin (le clavier).
2. **putchar(c)** : transfère le caractère c vers le fichier de sortie standard stdout (l'écran).
3. **puts (s)** : écrit la chaîne de caractères désignée par s sur stdout (l'écran) et provoque un retour à la ligne.
4. **gets(s)** : lit une ligne de de caractères du clavier et la copie à l'adresse indiquée par s. Le retour à la ligne final est remplacé par le symbole de fin de chaîne '\0'.

```
void main()
{
    char chaine[100];
    char C=' ';
        gets(chaine);
        puts(chaine);
        C=getchar();
        putchar(C);
}
```

Quelques fonctions utiles de <stdio.h>

1. **getchar** : qui lit le prochain caractère du fichier d'entrée standard stdin (le clavier).
2. **putchar(c)** : transfère le caractère c vers le fichier de sortie standard stdout (l'écran).
3. **puts (s)** : écrit la chaîne de caractères désignée par s sur stdout (l'écran) et provoque un retour à la ligne.
4. **gets(s)** : lit une ligne de de caractères du clavier et la copie à l'adresse indiquée par s. Le retour à la ligne final est remplacé par le symbole de fin de chaîne '\0'.

```
void main()
{
    char chaine[100];
    char C=' ';
        gets(chaine);
        puts(chaine);
        C=getchar();
        putchar(C);
}
```

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de chaînes de caractères en nombres

- **atoi(s)** :retourne la valeur numérique représentée par <s> comme int
- **atol(s)** :retourne la valeur numérique représentée par <s> comme long
- **atof(s)** : retourne la valeur numérique représentée par <s> comme double.

Règles générales pour la conversion :

- Les espaces au début d'une chaîne sont ignorés
- Il n'y a pas de contrôle du domaine de la cible
- La conversion s'arrête au premier caractère non convertible
- Pour une chaîne non convertible, les fonctions retournent zéro

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <stdlib.h>

Conversion de nombres en chaînes de caractères

- **itoa** (n_int, s, b)
- **ltoa** (n_long, s, b)
- **ultoa** (n_uns_long, s, b)

Chacune de ces trois procédures convertit son premier argument en une chaîne de caractères qui sera ensuite attribuée à <s>. La conversion se fait dans la base <b>.

- n_int est un nombre du type int
- n_long est un nombre du type long
- n_uns_long est un nombre du type unsigned long
- s est une chaîne de caractères longueur maximale de la chaîne : 17 resp. 33 byte
- b est la base pour la conversion (2 ... 36)

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Quelques fonctions de <ctype.h>

Les fonctions de classification suivantes fournissent un résultat du type int différent de zéro, si la condition respective est remplie, sinon zéro.

- **isupper(<c)** si <c> est une majuscule ('A'...'Z')
- **islower(c)** si c est une minuscule ('a'...'z')
- **isdigit(c)** si c est un chiffre décimal ('0'...'9')
- **isalpha(c)** si islower(c) ou isupper(c)
- **isalnum(c)** si isalpha(c) ou isdigit(c)
- **isxdigit(c)** si c est un chiffre hexadécimal ('0'...'9' ou 'A'...'F' ou 'a'...'f')
- **isspace(c)** si c est un signe d'espacement (' ', '\t', '\n', '\r', '\f')

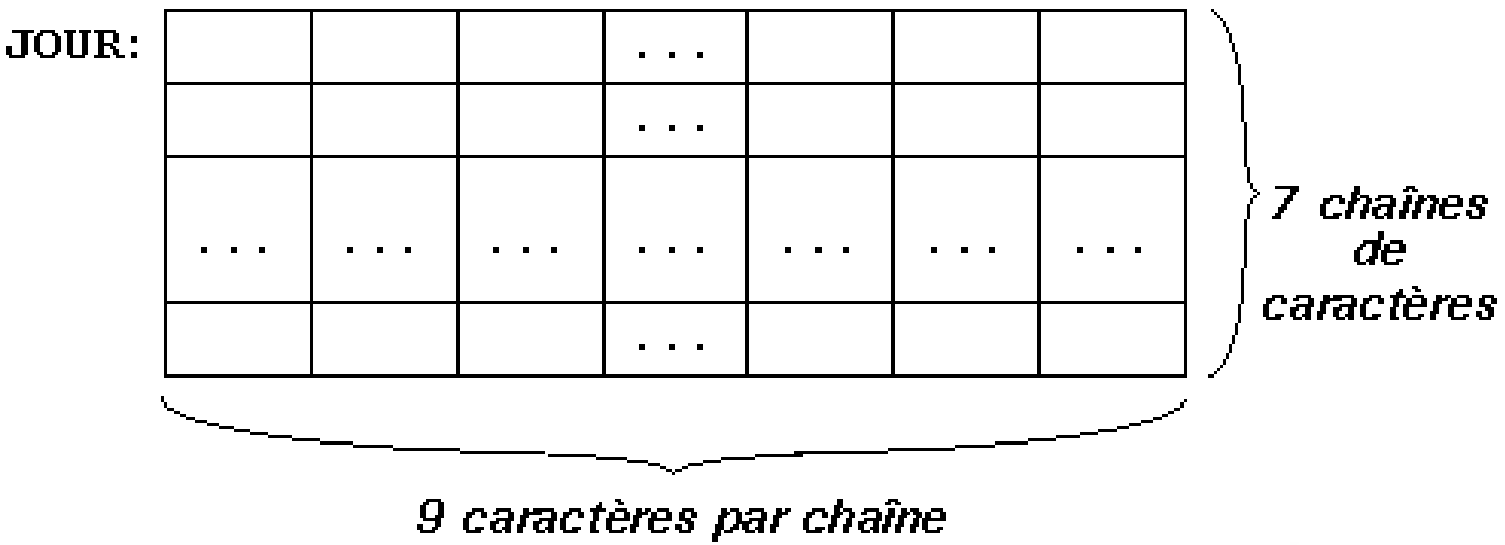
Les fonctions de conversion suivantes fournissent une valeur du type int qui peut être représentée comme caractère ; la valeur originale de c reste inchangée :

- **tolower(c)** retourne c converti en minuscule si c est une majuscule
- **toupper(c)** retourne c converti en majuscule si c est une minuscule

Tableaux de chaînes de caractères

- Un tableau de chaînes de caractères correspond à un tableau à deux dimensions du type char, où chaque ligne contient une chaîne de caractères.

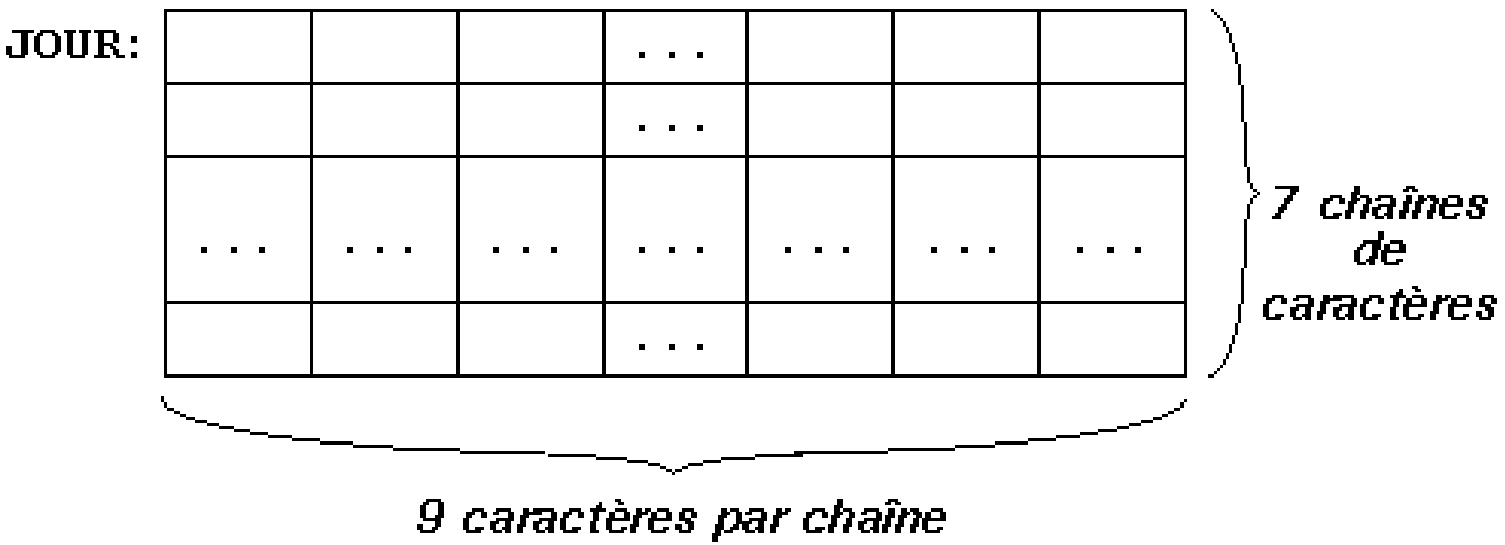
- Déclaration : **char JOUR[7][9] ;** réserve l'espace en mémoire pour 7 mots contenant 9 caractères (dont 8 caractères significatifs).



Tableaux de chaînes de caractères

- Un tableau de chaînes de caractères correspond à un tableau à deux dimensions du type char, où chaque ligne contient une chaîne de caractères.

- Déclaration : **char JOUR[7][9] ;** réserve l'espace en mémoire pour 7 mots contenant 9 caractères (dont 8 caractères significatifs).



Initialisation d'un tableau de chaînes de caractères

- Il est possible d'accéder aux différentes chaînes de caractères d'un tableau, en indiquant simplement la ligne correspondante.

```
char JOUR[7][9]= {"lundi", "mardi", "mercredi",
                  "jeudi", "vendredi",
                  "samedi", "dimanche"};
int I = 2;
printf("Aujourd'hui, c'est %s !\n", JOUR[I]);
```

- **attention :** Des expressions comme JOUR[I] représentent l'adresse du premier élément d'une chaîne de caractères. N'essayez donc pas de 'modifier' une telle adresse par une affectation directe !

- L'attribution d'une chaîne de caractères à une composante d'un tableau de chaînes se fait en général à l'aide de la fonction strcpy. Par exemple : **strcpy(JOUR[6], "Friday");**

Initialisation d'un tableau de chaînes de caractères

- Il est possible d'accéder aux différentes chaînes de caractères d'un tableau, en indiquant simplement la ligne correspondante.

```
char JOUR[7][9]= {"lundi", "mardi", "mercredi",
                  "jeudi", "vendredi",
                  "samedi", "dimanche"};
int I = 2;
printf("Aujourd'hui, c'est %s !\n", JOUR[I]);
```

- **attention :** Des expressions comme JOUR[I] représentent l'adresse du premier élément d'une chaîne de caractères. N'essayez donc pas de 'modifier' une telle adresse par une affectation directe !

- L'attribution d'une chaîne de caractères à une composante d'un tableau de chaînes se fait en général à l'aide de la fonction strcpy. Par exemple : **strcpy(JOUR[6], "Friday") ;**

Initialisation d'un tableau de chaînes de caractères

- Il est possible d'accéder aux différentes chaînes de caractères d'un tableau, en indiquant simplement la ligne correspondante.

```
char JOUR[7][9]= {"lundi", "mardi", "mercredi",
                  "jeudi", "vendredi",
                  "samedi", "dimanche"};
int I = 2;
printf("Aujourd'hui, c'est %s !\n", JOUR[I]);
```

- **attention :** Des expressions comme JOUR[I] représentent l'adresse du premier élément d'une chaîne de caractères. N'essayez donc pas de 'modifier' une telle adresse par une affectation directe !

- L'attribution d'une chaîne de caractères à une composante d'un tableau de chaînes se fait en général à l'aide de la fonction strcpy. Par exemple : **strcpy(JOUR[6], "Friday");**

Accès aux différents éléments d'un tableaux de chaînes de caractères

Lors de la déclaration il est possible d'initialiser toutes les composantes du tableau par des chaînes de caractères constantes :

```
char JOUR[7][9]= {"lundi", "mardi", "mercredi",
                  "jeudi", "vendredi", "samedi",
                  "dimanche"};
```

JOUR:									
	'l'	'u'	'n'	'd'	'i'	'\0'			
	'm'	'a'	'r'	'd'	'i'	'\0'			
	'm'	'e'	'r'	'c'	'r'	'e'	'd'	'i'	'\0'
	...	...	...	...	...	...	...	...	...
	'd'	'i'	'm'	'a'	'n'	'c'	'h'	'e'	'\0'

Initiation à l'algorithmique
Les fichiers

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

Introduction

- Jusqu'à présent, les données utilisées dans nos programmes sont :
  1. incluses dans le programme lui-même, par le programmeur,
  2. entrées à l'exécution par l'utilisateur.
- Mais évidemment, cela ne suffit pas à combler les besoins réels.
  - comment peut-on sauvegarder, dans ce cas-là, les noms et les notes des étudiants, les meilleurs scores des joueurs, les documents de texte qu'on rédige ...
  - nécessité d'un moyen de stockage permanent
- Les fichiers sont là pour combler ce manque. Ils servent à stocker des données de manière permanente, entre deux exécutions d'un programme.

Introduction

- Jusqu'à présent, les données utilisées dans nos programmes sont :
  1. incluses dans le programme lui-même, par le programmeur,
  2. entrées à l'exécution par l'utilisateur.
- Mais évidemment, cela ne suffit pas à combler les besoins réels.
  - comment peut-on sauvegarder, dans ce cas-là, les noms et les notes des étudiants, les meilleurs scores des joueurs, les documents textes qu'on rédige...
  - nécessité d'un moyen de stockage permanent
- Les fichiers sont là pour combler ce manque. Ils servent à stocker des données de manière permanente, entre deux exécutions d'un programme.

Introduction

- Jusqu'à présent, les données utilisées dans nos programmes sont :
  1. incluses dans le programme lui-même, par le programmeur,
  2. entrées à l'exécution par l'utilisateur.
- Mais évidemment, cela ne suffit pas à combler les besoins réels.
  - comment peut-on sauvegarder, dans ce cas-là, les noms et les notes des étudiants, les meilleurs scores des joueurs, les documents textes qu'on rédige...
  - nécessité d'un moyen de stockage permanent
- Les fichiers sont là pour combler ce manque. Ils servent à stocker des données de manière permanente, entre deux exécutions d'un programme.

Introduction

- Jusqu'à présent, les données utilisées dans nos programmes sont :
  1. incluses dans le programme lui-même, par le programmeur,
  2. entrées à l'exécution par l'utilisateur.
- Mais évidemment, cela ne suffit pas à combler les besoins réels.
  - comment peut-on sauvegarder, dans ce cas-là, les noms et les notes des étudiants, les meilleurs scores des joueurs, les documents textes qu'on rédige...
  - nécessité d'un moyen de stockage permanent
- Les fichiers sont là pour combler ce manque. Ils servent à stocker des données de manière permanente, entre deux exécutions d'un programme.

Introduction

- Jusqu'à présent, les données utilisées dans nos programmes sont :
  1. incluses dans le programme lui-même, par le programmeur,
  2. entrées à l'exécution par l'utilisateur.
- Mais évidemment, cela ne suffit pas à combler les besoins réels.
  - comment peut-on sauvegarder, dans ce cas-là, les noms et les notes des étudiants, les meilleurs scores des joueurs, les documents textes qu'on rédige...
  - nécessité d'un moyen de stockage permanent
- Les fichiers sont là pour combler ce manque. Ils servent à stocker des données de manière permanente, entre deux exécutions d'un programme.

Introduction

- Jusqu'à présent, les données utilisées dans nos programmes sont :
  1. incluses dans le programme lui-même, par le programmeur,
  2. entrées à l'exécution par l'utilisateur.
- Mais évidemment, cela ne suffit pas à combler les besoins réels.
  - comment peut-on sauvegarder, dans ce cas-là, les noms et les notes des étudiants, les meilleurs scores des joueurs, les documents textes qu'on rédige...
  - nécessité d'un moyen de stockage permanent
- Les fichiers sont là pour combler ce manque. Ils servent à stocker des données de manière permanente, entre deux exécutions d'un programme.

Introduction

- Jusqu'à présent, les données utilisées dans nos programmes sont :
  1. incluses dans le programme lui-même, par le programmeur,
  2. entrées à l'exécution par l'utilisateur.
- Mais évidemment, cela ne suffit pas à combler les besoins réels.
  - comment peut-on sauvegarder, dans ce cas-là, les noms et les notes des étudiants, les meilleurs scores des joueurs, les documents textes qu'on rédige...
  - nécessité d'un moyen de stockage permanent
- Les fichiers sont là pour combler ce manque. Ils servent à stocker des données de manière permanente, entre deux exécutions d'un programme.

Les fichiers

- Toute donnée en mémoire externe est organisée sous forme de fichier(s).
- Un fichier est :
  - une séquence d'octets,
  - repéré par un nom (dit nom externe), par exemple "montexte.txt" ou "TP2.c",...
  - enregistré sur un support physique non volatile de l'ordinateur : disque, clé USB, carte sd,...
- Un fichier n'est pas détruit à l'arrêt de l'ordinateur.
- La taille d'un fichier n'est pas précisée à sa création

Les fichiers

- Toute donnée en mémoire externe est organisée sous forme de fichier(s).
- Un fichier est :
  - une séquence d'octets,
  - repéré par un nom (dit nom externe), par exemple "montexte.txt" ou "TP2.c",...
  - enregistré sur un support physique non volatile de l'ordinateur : disque, clé USB, carte sd,...
- Un fichier n'est pas détruit à l'arrêt de l'ordinateur.
- La taille d'un fichier n'est pas précisée à sa création

Les fichiers

- Toute donnée en mémoire externe est organisée sous forme de fichier(s).
- Un fichier est :
  - une séquence d'octets,
  - repéré par un nom (dit nom externe), par exemple "montexte.txt" ou "TP2.c",...
  - enregistré sur un support physique non volatile de l'ordinateur : disque, clé USB, carte sd,...
- Un fichier n'est pas détruit à l'arrêt de l'ordinateur.
- La taille d'un fichier n'est pas précisée à sa création

Les fichiers

- Toute donnée en mémoire externe est organisée sous forme de fichier(s).
- Un fichier est :
  - une séquence d'octets,
  - repéré par un nom (dit nom externe), par exemple "montexte.txt" ou "TP2.c",...
  - enregistré sur un support physique non volatile de l'ordinateur : disque, clé USB, carte sd,...
- Un fichier n'est pas détruit à l'arrêt de l'ordinateur.
- La taille d'un fichier n'est pas précisée à sa création

Les fichiers

- Toute donnée en mémoire externe est organisée sous forme de fichier(s).
- Un fichier est :
  - une séquence d'octets,
  - repéré par un nom (dit nom externe), par exemple "montexte.txt" ou "TP2.c",...
  - enregistré sur un support physique non volatile de l'ordinateur : disque, clé USB, carte sd,...
- Un fichier n'est pas détruit à l'arrêt de l'ordinateur.
- La taille d'un fichier n'est pas précisée à sa création

Les fichiers

- Toute donnée en mémoire externe est organisée sous forme de fichier(s).
- Un fichier est :
  - une séquence d'octets,
  - repéré par un nom (dit nom externe), par exemple "montexte.txt" ou "TP2.c",...
  - enregistré sur un support physique non volatile de l'ordinateur : disque, clé USB, carte sd,...
- Un fichier n'est pas détruit à l'arrêt de l'ordinateur.
- La taille d'un fichier n'est pas précisée à sa création

Déclarer un Fichier

Un ficher est déclaré en C comme suit :

FILE* <nom_interne> ;

où nom_interne

- désigne un identificateur
- est une variable "pointeur" associée à un fichier de nom nom_externe
- permet de réaliser toutes les opérations d'un programme C sur ce fichier.

```
void main()
{
        FILE* f = NULL;
        // manipulation du fichier f
        // ...
}
```

Un lien doit toujours être établi entre ce fichier logique (nom_enterne f) et un fichier physique réel (nom externe "montexte.txt") se trouvant sur un support externe.

Déclarer un Fichier

Un ficher est déclaré en C comme suit :

FILE* <nom_interne> ;

où nom_interne

- désigne un identificateur
- est une variable "pointeur" associée à un fichier de nom nom_externe
- permet de réaliser toutes les opérations d'un programme C sur ce fichier.

```
void main()
{
        FILE* f = NULL;
        // manipulation du fichier f
        // ...
}
```

Un lien doit toujours être établi entre ce fichier logique (nom_enterne f) et un fichier physique réel (nom externe "montexte.txt") se trouvant sur un support externe.

Manipuler un fichier

Les principales opérations permettant de manipuler un fichier en C sont les suivantes :

1. **fopen() :** ouvrir un fichier.
2. **fclose() :** fermer un fichier.
3. **fwrite() :** écrire des données dans un fichier.
4. **fread() :** lire des données à partir d'un fichier.
5. **fseek() :** se positionner à un endroit précis du fichier.

☞Attention

Assurez-vous, dans la suite, que vous incluez bien au moins les bibliothèques **stdio.h** et **stdlib.h** en haut de votre fichier .c

Manipuler un fichier

Les principales opérations permettant de manipuler un fichier en C sont les suivantes :

1. **fopen() :** ouvrir un fichier.
2. **fclose() :** fermer un fichier.
3. **fwrite() :** écrire des données dans un fichier.
4. **fread() :** lire des données à partir d'un fichier.
5. **fseek() :** se positionner à un endroit précis du fichier.

☞**Attention**

Assurez-vous, dans la suite, que vous incluez bien au moins les bibliothèques **stdio.h** et **stdlib.h** en haut de votre fichier .c

Manipuler un fichier

Les principales opérations permettant de manipuler un fichier en C sont les suivantes :

1. **fopen() :** ouvrir un fichier.
2. **fclose() :** fermer un fichier.
3. **fwrite() :** écrire des données dans un fichier.
4. **fread() :** lire des données à partir d'un fichier.
5. **fseek() :** se positionner à un endroit précis du fichier.

☞**Attention**

Assurez-vous, dans la suite, que vous incluez bien au moins les bibliothèques **stdio.h** et **stdlib.h** en haut de votre fichier .c

Manipuler un fichier

Les principales opérations permettant de manipuler un fichier en C sont les suivantes :

1. **fopen() :** ouvrir un fichier.
2. **fclose() :** fermer un fichier.
3. **fwrite() :** écrire des données dans un fichier.
4. **fread() :** lire des données à partir d'un fichier.
5. **fseek() :** se positionner à un endroit précis du fichier.

☞Attention

Assurez-vous, dans la suite, que vous incluez bien au moins les bibliothèques **stdio.h** et **stdlib.h** en haut de votre fichier .c

Manipuler un fichier

Les principales opérations permettant de manipuler un fichier en C sont les suivantes :

1. **fopen() :** ouvrir un fichier.
2. **fclose() :** fermer un fichier.
3. **fwrite() :** écrire des données dans un fichier.
4. **fread() :** lire des données à partir d'un fichier.
5. **fseek() :** se positionner à un endroit précis du fichier.

☞**Attention**

Assurez-vous, dans la suite, que vous incluez bien au moins les bibliothèques **stdio.h** et **stdlib.h** en haut de votre fichier .c

Manipuler un fichier

Les principales opérations permettant de manipuler un fichier en C sont les suivantes :

1. **fopen() :** ouvrir un fichier.
2. **fclose() :** fermer un fichier.
3. **fwrite() :** écrire des données dans un fichier.
4. **fread() :** lire des données à partir d'un fichier.
5. **fseek() :** se positionner à un endroit précis du fichier.

☞Attention

Assurez-vous, dans la suite, que vous incluez bien au moins les bibliothèques **stdio.h** et **stdlib.h** en haut de votre fichier .c

Manipuler un fichier

Les principales opérations permettant de manipuler un fichier en C sont les suivantes :

1. **fopen() :** ouvrir un fichier.
2. **fclose() :** fermer un fichier.
3. **fwrite() :** écrire des données dans un fichier.
4. **fread() :** lire des données à partir d'un fichier.
5. **fseek() :** se positionner à un endroit précis du fichier.

☞**Attention**
Assurez-vous, dans la suite, que vous incluez bien au moins les bibliothèques **stdio.h** et **stdlib.h** en haut de votre fichier .c

Ouvrir un fichier : fopen()

L'ouverture d'un fichier se fait à l'aide de la fonction **fopen** de prototype :

```
FILE *fopen(char *nom_externe, char *mode);
```

- **nom_externe** est une chaîne de caractères contenant le nom du fichier à ouvrir.
- **mode** est une chaîne de caractères définissant le type et le mode d'accès du fichier. C'est-à-dire une indication qui mentionne ce que vous voulez faire : seulement écrire dans le fichier, seulement le lire, ou les deux à la fois.
- **fopen** retourne le **nom_interne** du fichier en cas de succès et **NULL** dans le cas contraire.

Ouvrir un fichier : fopen()

L'ouverture d'un fichier se fait à l'aide de la fonction **fopen** de prototype :

```
FILE *fopen(char *nom_externe, char *mode);
```

- **nom_externe** est une chaîne de caractères contenant le nom du fichier à ouvrir.
- **mode** est une chaîne de caractères définissant le type et le mode d'accès du fichier. C'est-à-dire une indication qui mentionne ce que vous voulez faire : seulement écrire dans le fichier, seulement le lire, ou les deux à la fois.
- **fopen** retourne le **nom_interne** du fichier en cas de succès et **NULL** dans le cas contraire.

Ouvrir un fichier : fopen()

L'ouverture d'un fichier se fait à l'aide de la fonction **fopen** de prototype :

```
FILE *fopen(char *nom_externe, char *mode);
```

- **nom_externe** est une chaîne de caractères contenant le nom du fichier à ouvrir.
- **mode** est une chaîne de caractères définissant le type et le mode d'accès du fichier. C'est-à-dire une indication qui mentionne ce que vous voulez faire : seulement écrire dans le fichier, seulement le lire, ou les deux à la fois.
- **fopen** retourne le **nom_interne** du fichier en cas de succès et **NULL** dans le cas contraire.

Ouvrir un fichier : fopen()

L'ouverture d'un fichier se fait à l'aide de la fonction **fopen** de prototype :

```
FILE *fopen(char *nom_externe, char *mode);
```

- **nom_externe** est une chaîne de caractères contenant le nom du fichier à ouvrir.
- **mode** est une chaîne de caractères définissant le type et le mode d'accès du fichier. C'est-à-dire une indication qui mentionne ce que vous voulez faire : seulement écrire dans le fichier, seulement le lire, ou les deux à la fois.
- **fopen** retourne le **nom_interne** du fichier en cas de succès et **NULL** dans le cas contraire.

Ouvrir un fichier : modes d'accès

```
FILE* f1 = NULL, f2 = NULL, f3 = NULL;
f1 = fopen("montexte1.txt", "r");          // lecture seule
f2 = fopen("montexte2.txt", "w");          // ecriture seule
f3 = fopen("montexte3.txt", "a");          // mode d'ajout
```

- **"r" : lecture seule**. Pour lire le contenu du fichier, mais pas y écrire. Le fichier doit avoir été créé au préalable. Le curseur associé au fichier est positionné au début du fichier.
- **"w" : écriture seule**. Pour écrire dans le fichier, mais pas lire son contenu. Si le fichier existe il sera détruit. S'il n'existe pas, il sera créé. Le curseur est positionné au début du fichier.
- **"a" : mode d'ajout**. Pour écrire à la fin du fichier quelle que soit la position courante du curseur. Si le fichier n'existe pas, il sera créé.

Ouvrir un fichier : modes d'accès

```
FILE* f1 = NULL, f2 = NULL, f3 = NULL;
f1 = fopen("montexte1.txt", "r");          // lecture seule
f2 = fopen("montexte2.txt", "w");          // ecriture seule
f3 = fopen("montexte3.txt", "a");          // mode d'ajout
```

- **"r" : lecture seule**. Pour lire le contenu du fichier, mais pas y écrire. Le fichier doit avoir été créé au préalable. Le curseur associé au fichier est positionné au début du fichier.

- **"w" : écriture seule**. Pour écrire dans le fichier, mais pas lire son contenu. Si le fichier existe il sera détruit. S'il n'existe pas, il sera créé. Le curseur est positionné au début du fichier.

- **"a" : mode d'ajout**. Pour écrire à la fin du fichier quelle que soit la position courante du curseur. Si le fichier n'existe pas, il sera créé.

Ouvrir un fichier : modes d'accès

```
FILE* f1 = NULL, f2 = NULL, f3 = NULL;
f1 = fopen("montexte1.txt", "r");          // lecture seule
f2 = fopen("montexte2.txt", "w");          // ecriture seule
f3 = fopen("montexte3.txt", "a");          // mode d'ajout
```

- **"r" : lecture seule**. Pour lire le contenu du fichier, mais pas y écrire. Le fichier doit avoir été créé au préalable. Le curseur associé au fichier est positionné au début du fichier.

- **"w" : écriture seule**. Pour écrire dans le fichier, mais pas lire son contenu. Si le fichier existe il sera détruit. S'il n'existe pas, il sera créé. Le curseur est positionné au début du fichier.

- **"a" : mode d'ajout**. Pour écrire à la fin du fichier quelle que soit la position courante du curseur. Si le fichier n'existe pas, il sera créé.

Ouvrir un fichier : modes d'accès

```
FILE* f1 = NULL, f2 = NULL, f3 = NULL;
f1 = fopen("montexte1.txt", "r+");// lecture et ecriture
f2 = fopen("montexte2.txt", "w+");// lecture et ecriture,
    avec suppression du contenu au prealable
f3 = fopen("montexte3.txt", "a+");// ajout en lecture/
    ecriture a la fin
```

- **"r+" : lecture et écriture**. Identique au mode **"r"** avec possibilité d'écriture
- **"w+" : lecture et écriture, avec suppression du contenu au préalable**. Identique au mode **"w"** avec possibilité de lecture.
  ☞Attention ! Le fichier est d'abord vidé de son contenu.
- **"a+" : ajout en lecture/écriture à la fin**. Identique au mode **"a"** avec possibilité de lecture.

Ouvrir un fichier : modes d'accès

```
FILE* f1 = NULL, f2 = NULL, f3 = NULL;
f1 = fopen("montexte1.txt", "r+");// lecture et ecriture
f2 = fopen("montexte2.txt", "w+");// lecture et ecriture,
    avec suppression du contenu au prealable
f3 = fopen("montexte3.txt", "a+");// ajout en lecture/
    ecriture a la fin
```

- **"r+" : lecture et écriture**. Identique au mode **"r"** avec possibilité d'écriture
- **"w+" : lecture et écriture, avec suppression du contenu au préalable**. Identique au mode **"w"** avec possibilité de lecture.
  ☞Attention ! Le fichier est d'abord vidé de son contenu.
- **"a+" : ajout en lecture/écriture à la fin**. Identique au mode **"a"** avec possibilité de lecture.

Ouvrir un fichier : modes d'accès

```
FILE* f1 = NULL, f2 = NULL, f3 = NULL;
f1 = fopen("montexte1.txt", "r+");// lecture et ecriture
f2 = fopen("montexte2.txt", "w+");// lecture et ecriture,
    avec suppression du contenu au prealable
f3 = fopen("montexte3.txt", "a+");// ajout en lecture/
    ecriture a la fin
```

- **"r+" : lecture et écriture**. Identique au mode **"r"** avec possibilité d'écriture

- **"w+" : lecture et écriture, avec suppression du contenu au préalable**. Identique au mode **"w"** avec possibilité de lecture.
☞Attention ! Le fichier est d'abord vidé de son contenu.

- **"a+" : ajout en lecture/écriture à la fin**. Identique au mode **"a"** avec possibilité de lecture.

Ouvrir un fichier : exemple d'une bonne pratique

Ouvrir un fichier revient à créer un lien entre le **nom_interne** (f) et le **nom_externe** ("montexte.txt") :

```
void main()
{
    FILE* f = NULL;
    f = fopen("montexte.txt", "r+");
    if (f != NULL)
    {
        // On peut lire et ecrire dans le fichier
    }
    else
    {
      // On affiche un message d'erreur si on veut
      printf("Impossible d'ouvrir le fichier montexte.txt");
    }
}
```

- Si l'ouverture a fonctionné (si le pointeur est différent de NULL), alors on peut s'amuser à lire et écrire dans le fichier.
- Si le pointeur vaut NULL, c'est que l'ouverture du fichier a échoué. On ne peut donc pas continuer (afficher un message d'erreur).

Ouvrir un fichier : exemple d'une bonne pratique

Ouvrir un fichier revient à créer un lien entre le **nom_interne** (f) et le **nom_externe** ("montexte.txt") :

```
void main()
{
    FILE* f = NULL;
    f = fopen("montexte.txt", "r+");
    if (f != NULL)
    {
        // On peut lire et ecrire dans le fichier
    }
    else
    {
      // On affiche un message d'erreur si on veut
      printf("Impossible d'ouvrir le fichier montexte.txt");
    }
}
```

- Si l'ouverture a fonctionné (si le pointeur est différent de NULL), alors on peut s'amuser à lire et écrire dans le fichier.
- Si le pointeur vaut NULL, c'est que l'ouverture du fichier a échoué. On ne peut donc pas continuer (afficher un message d'erreur).

Ouvrir un fichier : chemins des fichiers

```
FILE* f1, f2, f3, f4;
f1 = fopen("montexte1.txt", "r");          // meme dossier
f2 = fopen("TP2/monTP2.c", "w");           // chemin relatif
f3 = fopen("C:\\Mes_TP\\TP3\\source.h", "a");//chemin absolu
f4 = fopen("/home/Toto/Loulou/TD.doc", "a"); // sous linux
```

- **f1** doit être situé dans le même dossier que l'exécutable (.exe)
- **f2** est situé dans un sous-dossier appelé "TP2". c'est ce qu'on on appelle **chemin relatif**. Peu importe l'endroit où est installé votre programme cela fonctionnera toujours.
- Il est aussi possible d'ouvrir un autre fichier n'importe où ailleurs sur le disque dur en utilisant ce qu'on appelle **chemin absolu** (par exemple le fichier **f3**).
- Le défaut des chemins absolus, c'est qu'ils ne fonctionnent que sur un OS précis. Sous Linux par exemple, on aurait dû écrire un chemin à-la-linux, comme pour le fichier **f4**.

Ouvrir un fichier : chemins des fichiers

```
FILE* f1, f2, f3, f4;
f1 = fopen("montexte1.txt", "r");          // meme dossier
f2 = fopen("TP2/monTP2.c", "w");           // chemin relatif
f3 = fopen("C:\\Mes_TP\\TP3\\source.h", "a");//chemin absolu
f4 = fopen("/home/Toto/Loulou/TD.doc", "a"); // sous linux
```

- **f1** doit être situé dans le même dossier que l'exécutable (.exe)
- **f2** est situé dans un sous-dossier appelé "TP2". c'est ce qu'on on appelle **chemin relatif**. Peu importe l'endroit où est installé votre programme cela fonctionnera toujours.
- Il est aussi possible d'ouvrir un autre fichier n'importe où ailleurs sur le disque dur en utilisant ce qu'on appelle **chemin absolu** (par exemple le fichier **f3**).
- Le défaut des chemins absolus, c'est qu'ils ne fonctionnent que sur un OS précis. Sous Linux par exemple, on aurait dû écrire un chemin à-la-linux, comme pour le fichier **f4**.

Ouvrir un fichier : chemins des fichiers

```
FILE* f1, f2, f3, f4;
f1 = fopen("montexte1.txt", "r");          // meme dossier
f2 = fopen("TP2/monTP2.c", "w");           // chemin relatif
f3 = fopen("C:\\Mes_TP\\TP3\\source.h", "a");//chemin absolu
f4 = fopen("/home/Toto/Loulou/TD.doc", "a"); // sous linux
```

- **f1** doit être situé dans le même dossier que l'exécutable (.exe)
- **f2** est situé dans un sous-dossier appelé "TP2". c'est ce qu'on on appelle **chemin relatif**. Peu importe l'endroit où est installé votre programme cela fonctionnera toujours.
- Il est aussi possible d'ouvrir un autre fichier n'importe où ailleurs sur le disque dur en utilisant ce qu'on appelle **chemin absolu** (par exemple le fichier **f3**).
- Le défaut des chemins absolus, c'est qu'ils ne fonctionnent que sur un OS précis. Sous Linux par exemple, on aurait dû écrire un chemin à-la-linux, comme pour le fichier **f4**.

Ouvrir un fichier : chemins des fichiers

```
FILE* f1, f2, f3, f4;
f1 = fopen("montexte1.txt", "r");          // meme dossier
f2 = fopen("TP2/monTP2.c", "w");           // chemin relatif
f3 = fopen("C:\\Mes_TP\\TP3\\source.h", "a");//chemin absolu
f4 = fopen("/home/Toto/Loulou/TD.doc", "a"); // sous linux
```

- **f1** doit être situé dans le même dossier que l'exécutable (.exe)
- **f2** est situé dans un sous-dossier appelé "TP2". c'est ce qu'on on appelle **chemin relatif**. Peu importe l'endroit où est installé votre programme cela fonctionnera toujours.
- Il est aussi possible d'ouvrir un autre fichier n'importe où ailleurs sur le disque dur en utilisant ce qu'on appelle **chemin absolu** (par exemple le fichier **f3**).
- Le défaut des chemins absolus, c'est qu'ils ne fonctionnent que sur un OS précis. Sous Linux par exemple, on aurait dû écrire un chemin à-la-linux, comme pour le fichier **f4**.

Fermer un fichier : fclose()

La fermeture d'un fichier se fait par la fonction **fclose** de prototype :

int **fclose**(FILE* <nom_interne>)

```
void main()
{
    FILE* f = NULL;
    f = fopen("montexte.txt", "r+");
    if (f != NULL)
    {
        // On peut lire et ecrire dans le fichier
        //...
        fclose(f); // On ferme le fichier qui a ete ouvert
    }
    else
      printf("Impossible d'ouvrir le fichier montexte.txt");
}
```

- **fclose** retourne zéro en cas de succès
- ☞**Attention !** Il faut toujours penser à fermer son fichier une fois que l'on a fini de travailler avec. Cela permet de libérer de la mémoire.

Fermer un fichier : fclose()

La fermeture d'un fichier se fait par la fonction **fclose** de prototype :

int **fclose**(FILE* <nom_interne>)

```
void main()
{
    FILE* f = NULL;
    f = fopen("montexte.txt", "r+");
    if (f != NULL)
    {
        // On peut lire et ecrire dans le fichier
        //...
        fclose(f); // On ferme le fichier qui a ete ouvert
    }
    else
      printf("Impossible d'ouvrir le fichier montexte.txt");
}
```

- **fclose** retourne zéro en cas de succès
- ☞**Attention !** Il faut toujours penser à fermer son fichier une fois que l'on a fini de travailler avec. Cela permet de libérer de la mémoire.

Écrire dans un fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fputc :** écrit un caractère dans le fichier (**un seul** caractère à la fois) ;
2. **fputs :** écrit une chaîne dans le fichier ;
3. **fprintf :** écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.
4. **fwrite :** écrit dans le fichier un certain nombre éléments pointés de taille précise.

Écrire dans un fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fputc :** écrit un caractère dans le fichier (**un seul** caractère à la fois) ;
2. **fputs :** écrit une chaîne dans le fichier ;
3. **fprintf :** écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.
4. **fwrite :** écrit dans le fichier un certain nombre éléments pointés de taille précise.

Écrire dans un fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fputc :** écrit un caractère dans le fichier (**un seul** caractère à la fois) ;
2. **fputs :** écrit une chaîne dans le fichier ;
3. **fprintf :** écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.
4. **fwrite :** écrit dans le fichier un certain nombre éléments pointés de taille précise.

Écrire dans un fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fputc :** écrit un caractère dans le fichier (**un seul** caractère à la fois) ;
2. **fputs :** écrit une chaîne dans le fichier ;
3. **fprintf :** écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.
4. **fwrite :** écrit dans le fichier un certain nombre éléments pointés de taille précise.

Écrire dans un fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fputc :** écrit un caractère dans le fichier (**un seul** caractère à la fois) ;
2. **fputs :** écrit une chaîne dans le fichier ;
3. **fprintf :** écrit une chaîne « formatée » dans le fichier, fonctionnement quasi-identique à printf.
4. **fwrite :** écrit dans le fichier un certain nombre éléments pointés de taille précise.

Écrire dans un fichier : fputc()

Cette fonction écrit un caractère à la fois dans le fichier :

int **fputc**(int caractere, FILE* pointeurSurFichier);

```
FILE* f = NULL;
f = fopen("montexte.txt", "r+");
if (f != NULL)
{
    fputc('A', f); // Ecriture du caractere 'A'
    fclose(f);
}
else
  printf("Impossible d'ouvrir le fichier montexte.txt");
```

Elle prend deux paramètres :

- Le caractère à écrire (de type int).
- Le pointeur sur le fichier dans lequel écrire.

La fonction **fputc** retourne un int, c'est un code d'erreur. Il vaut **EOF** si l'écriture a échoué, sinon il a une autre valeur.

Écrire dans un fichier : fputc()

Cette fonction écrit un caractère à la fois dans le fichier :

int **fputc**(int caractere, FILE* pointeurSurFichier);

```
FILE* f = NULL;
f = fopen("montexte.txt", "r+");
if (f != NULL)
{
    fputc('A', f); // Ecriture du caractere 'A'
    fclose(f);
}
else
  printf("Impossible d'ouvrir le fichier montexte.txt");
```

Elle prend deux paramètres :

- Le caractère à écrire (de type int).
- Le pointeur sur le fichier dans lequel écrire.

La fonction **fputc** retourne un int, c'est un code d'erreur. Il vaut **EOF** si l'écriture a échoué, sinon il a une autre valeur.

Écrire dans un fichier : fputc()

Cette fonction écrit un caractère à la fois dans le fichier :

int **fputc**(int caractere, FILE* pointeurSurFichier);

```
FILE* f = NULL;
f = fopen("montexte.txt", "r+");
if (f != NULL)
{
    fputc('A', f); // Ecriture du caractere 'A'
    fclose(f);
}
else
  printf("Impossible d'ouvrir le fichier montexte.txt");
```

Elle prend deux paramètres :

- Le caractère à écrire (de type int).
- Le pointeur sur le fichier dans lequel écrire.

La fonction **fputc** retourne un int, c'est un code d'erreur. Il vaut **EOF** si l'écriture a échoué, sinon il a une autre valeur.

Écrire dans un fichier : fputs()

Cette fonction écrit une chaîne de caractères dans le fichier :

char* **fputs**(const char* chaine, FILE* pointeurSurFichier) ;

```
FILE* f = NULL;
f = fopen("montexte.txt", "r+");
if (f != NULL)
{
  fputs("Ecriture de Toto\n et Loulou dans le fichier",f);
  fclose(f);
}
else
  printf("Impossible d'ouvrir le fichier montexte.txt");
```

Elle prend deux paramètres :

- **chaine :** la chaîne à écrire.
- **pointeurSurFichier :** comme pour fputc, il s'agit de votre pointeur de type FILE* sur le fichier que vous avez ouvert.

La fonction **fputs** renvoie **EOF** s'il y a eu une erreur.

Écrire dans un fichier : fputs()

Cette fonction écrit une chaîne de caractères dans le fichier :

char* **fputs**(const char* chaine, FILE* pointeurSurFichier);

```
FILE* f = NULL;
f = fopen("montexte.txt", "r+");
if (f != NULL)
{
  fputs("Ecriture de Toto\n et Loulou dans le fichier",f);
  fclose(f);
}
else
  printf("Impossible d'ouvrir le fichier montexte.txt");
```

Elle prend deux paramètres :

- **chaine :** la chaîne à écrire.
- **pointeurSurFichier :** comme pour fputc, il s'agit de votre pointeur de type FILE* sur le fichier que vous avez ouvert.

La fonction **fputs** renvoie **EOF** s'il y a eu une erreur.

Écrire dans un fichier : fputs()

Cette fonction écrit une chaîne de caractères dans le fichier :

char* **fputs**(const char* chaine, FILE* pointeurSurFichier);

```
FILE* f = NULL;
f = fopen("montexte.txt", "r+");
if (f != NULL)
{
  fputs("Ecriture de Toto\n et Loulou dans le fichier",f);
  fclose(f);
}
else
  printf("Impossible d'ouvrir le fichier montexte.txt");
```

Elle prend deux paramètres :

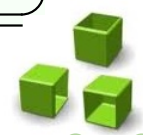
- **chaine :** la chaîne à écrire.
- **pointeurSurFichier :** comme pour fputc, il s'agit de votre pointeur de type FILE* sur le fichier que vous avez ouvert.

La fonction **fputs** renvoie **EOF** s'il y a eu une erreur.

Écrire dans un fichier : fprintf()

- Elle s'utilise de la même manière que **printf**.
- Il faut juste indiquer un pointeur de **FILE** en premier paramètre.

```
FILE* f = NULL;
int age = 0;
f = fopen("montexte.txt", "r+");
if (f != NULL)
{
     // On demande l'age
     printf("Quel age avez-vous ? ");
     scanf("%d", &age);

     // On l'ecrit dans le fichier
     fprintf(f, "Vous avez %d ans", age);
     fclose(f);
}
else
  printf("Impossible d'ouvrir le fichier montexte.txt");
```



Écrire dans un fichier : fwrite()

Cette fonction écrit une chaîne de caractères dans le fichier :

int **fwrite**(void *p, int taille, int nombre, FILE* nom_interne) ;

```
typedef struct
    { char nom[20];
      int age;
    } Etudiant;
FILE* f = NULL;
Etudiant e1, etu_tab[3];
f = fopen("montexte.txt", "r+");
if (f != NULL)
{ fwrite(&e1, sizeof(Etudiant), 1, f);
      fwrite(etu_tab, sizeof(Etudiant), 3, f);
      fclose(f);
}
```

- **fwrite** écrit dans le fichier "**nombre**" éléments pointés par "**p**", chacun de "**taille**" octets ;
- **fwrite** retourne le nombre d'éléments effectivement écrits ;
- l'ecriture se fait à partir de la position courante du curseur, et déplace celui-ci du nombre d'éléments écrits.

Écrire dans un fichier : fwrite()

Cette fonction écrit une chaîne de caractères dans le fichier :

int **fwrite**(void *p, int taille, int nombre, FILE* nom_interne) ;

```
typedef struct
    { char nom[20];
      int age;
    } Etudiant;
FILE* f = NULL;
Etudiant e1, etu_tab[3];
f = fopen("montexte.txt", "r+");
if (f != NULL)
{ fwrite(&e1, sizeof(Etudiant), 1, f);
      fwrite(etu_tab, sizeof(Etudiant), 3, f);
      fclose(f);
}
```

- **fwrite** écrit dans le fichier "**nombre**" éléments pointés par "**p**", chacun de "**taille**" octets ;
- **fwrite** retourne le nombre d'éléments effectivement écrits ;
- l'ecriture se fait à partir de la position courante du curseur, et déplace celui-ci du nombre d'éléments écrits.

Écrire dans un fichier : fwrite()

Cette fonction écrit une chaîne de caractères dans le fichier :

int **fwrite**(void *p, int taille, int nombre, FILE* nom_interne) ;

```
typedef struct
    { char nom[20];
      int age;
    } Etudiant;
FILE* f = NULL;
Etudiant e1, etu_tab[3];
f = fopen("montexte.txt", "r+");
if (f != NULL)
{ fwrite(&e1, sizeof(Etudiant), 1, f);
      fwrite(etu_tab, sizeof(Etudiant), 3, f);
      fclose(f);
}
```

- **fwrite** écrit dans le fichier "**nombre**" éléments pointés par "**p**", chacun de "**taille**" octets ;
- **fwrite** retourne le nombre d'éléments effectivement écrits ;
- l'ecriture se fait à partir de la position courante du curseur, et déplace celui-ci du nombre d'éléments écrits.

Lire depuis un fichier

Il existe plusieurs fonctions permettant de lire des données à partir d'un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fgetc :** lit un caractère depuis le fichier (**un seul** caractère à la fois) ;
2. **fgets :** lit une chaîne de caractère depuis le fichier ;
3. **fscanf :** lit une chaîne de caractères « formatée » depuis le fichier, fonctionnement quasi-identique à scanf.
4. **fread :** lit depuis le fichier un certain nombre éléments de taille définie, et range les éléments lus en mémoire à une adresse précise.

Lire depuis un fichier

Il existe plusieurs fonctions permettant de lire des données à partir d'un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fgetc :** lit un caractère depuis le fichier (**un seul** caractère à la fois) ;
2. **fgets :** lit une chaîne de caractère depuis le fichier ;
3. **fscanf :** lit une chaîne de caractères « formatée » depuis le fichier, fonctionnement quasi-identique à scanf.
4. **fread :** lit depuis le fichier un certain nombre éléments de taille définie, et range les éléments lus en mémoire à une adresse précise.

Lire depuis un fichier

Il existe plusieurs fonctions permettant de lire des données à partir d'un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fgetc :** lit un caractère depuis le fichier (**un seul** caractère à la fois) ;
2. **fgets :** lit une chaîne de caractère depuis le fichier ;
3. **fscanf :** lit une chaîne de caractères « formatée » depuis le fichier, fonctionnement quasi-identique à scanf.
4. **fread :** lit depuis le fichier un certain nombre éléments de taille définie, et range les éléments lus en mémoire à une adresse précise.

Lire depuis un fichier

Il existe plusieurs fonctions permettant de lire des données à partir d'un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fgetc :** lit un caractère depuis le fichier (**un seul** caractère à la fois) ;
2. **fgets :** lit une chaîne de caractère depuis le fichier ;
3. **fscanf :** lit une chaîne de caractères « formatée » depuis le fichier, fonctionnement quasi-identique à scanf.
4. **fread :** lit depuis le fichier un certain nombre éléments de taille définie, et range les éléments lus en mémoire à une adresse précise.

Lire depuis un fichier

Il existe plusieurs fonctions permettant de lire des données à partir d'un fichier. Il faut choisir celle qui est la plus adaptée à notre cas :

1. **fgetc :** lit un caractère depuis le fichier (**un seul** caractère à la fois) ;
2. **fgets :** lit une chaîne de caractère depuis le fichier ;
3. **fscanf :** lit une chaîne de caractères « formatée » depuis le fichier, fonctionnement quasi-identique à scanf.
4. **fread :** lit depuis le fichier un certain nombre éléments de taille définie, et range les éléments lus en mémoire à une adresse précise.

Écrire dans un fichier : fgetc()

Cette fonction lit un caractère à la fois depuis le fichier :

int **fgetc**(FILE* pointeurSurFichier) ;

```
FILE* f = NULL;
int c = 0;
f = fopen("montexte.txt", "r");
if (f != NULL){
  do{
      c = fgetc(f); // On lit le caractere
      printf("%c", c); // On l'affiche
    } while (c != EOF); // On continue tant que fgetc n'
        a pas retourne EOF (fin de fichier)
}
```

- Cette fonction retourne un int : c'est le caractère qui a été lu.
- Si la fonction n'a pas pu lire de caractère, elle retourne **EOF**.

fgetc avance le curseur d'un caractère à chaque fois que vous en lisez un. Si vous appelez fgetc une seconde fois, la fonction lira donc le second caractère, puis le troisième et ainsi de suite. Vous pouvez donc faire une boucle pour lire les caractères un par un dans le fichier.

Écrire dans un fichier : fgetc()

Cette fonction lit un caractère à la fois depuis le fichier :

int **fgetc**(FILE* pointeurSurFichier) ;

```
FILE* f = NULL;
int c = 0;
f = fopen("montexte.txt", "r");
if (f != NULL){
  do{
      c = fgetc(f); // On lit le caractere
      printf("%c", c); // On l'affiche
    } while (c != EOF); // On continue tant que fgetc n'
      a pas retourne EOF (fin de fichier)
}
```

- Cette fonction retourne un int : c'est le caractère qui a été lu.
- Si la fonction n'a pas pu lire de caractère, elle retourne **EOF**.

fgetc avance le curseur d'un caractère à chaque fois que vous en lisez un. Si vous appelez fgetc une seconde fois, la fonction lira donc le second caractère, puis le troisième et ainsi de suite. Vous pouvez donc faire une boucle pour lire les caractères un par un dans le fichier.

Lire depuis un fichier : fgets()

Cette fonction lit une chaîne depuis le fichier.

char* **fgets**(char* chaine, int nbr_Caracteres, FILE* ptr_Fichier) ;

```
#define TAILLE_MAX 1000
int main()
{ FILE* f = NULL;
  char chaine[TAILLE_MAX] = "";
  f = fopen("montexte.txt", "r");
  if (f != NULL)
        {fgets(chaine, TAILLE_MAX, f); // On lit maximum
            TAILLE_MAX caracteres du fichier, on stocke le
            tout dans "chaine"
    printf("%s", chaine); // On affiche la chaine
    fclose(f);
        }
}
```

- **nbr_Caracteres :** c'est le nombre de caractères à lire. En effet, La fonction fgets s'arrête de lire la ligne si elle contient plus de nbr_Caracteres caractères.
- La fonction lit au maximum une ligne (elle s'arrête au premier \n qu'elle rencontre).

Lire depuis un fichier : fgets()

Cette fonction lit une chaîne depuis le fichier.

char* **fgets**(char* chaine, int nbr_Caracteres, FILE* ptr_Fichier) ;

```
#define TAILLE_MAX 1000
int main()
{ FILE* f = NULL;
  char chaine[TAILLE_MAX] = "";
  f = fopen("montexte.txt", "r");
  if (f != NULL)
        {fgets(chaine, TAILLE_MAX, f); // On lit maximum
            TAILLE_MAX caracteres du fichier, on stocke le
            tout dans "chaine"
    printf("%s", chaine); // On affiche la chaine
    fclose(f);
        }
}
```

- **nbr_Caracteres :** c'est le nombre de caractères à lire. En effet, La fonction fgets s'arrête de lire la ligne si elle contient plus de nbr_Caracteres caractères.
- La fonction lit au maximum une ligne (elle s'arrête au premier \n qu'elle rencontre).

Lire depuis un fichier : fgets()

Cette fonction lit une chaîne depuis le fichier.

char* **fgets**(char* chaine, int nbr_Caracteres, FILE* ptr_Fichier) ;

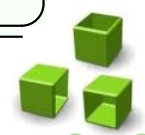
```
#define TAILLE_MAX 1000
int main()
{ FILE* f = NULL;
  char chaine[TAILLE_MAX] = "";
  f = fopen("montexte.txt", "r");
  if (f != NULL)
        {fgets(chaine, TAILLE_MAX, f); // On lit maximum
            TAILLE_MAX caracteres du fichier, on stocke le
            tout dans "chaine"
    printf("%s", chaine); // On affiche la chaine
    fclose(f);
        }
}
```

- **nbr_Caracteres :** c'est le nombre de caractères à lire. En effet, La fonction fgets s'arrête de lire la ligne si elle contient plus de nbr_Caracteres caractères.
- La fonction lit au maximum une ligne (elle s'arrête au premier \n qu'elle rencontre).

Lire depuis un fichier : fgets()

Si vous voulez lire plusieurs lignes, il faudra faire une boucle.

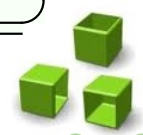
```
#define TAILLE_MAX 1000
int main()
{ FILE* f = NULL;
  char chaine[TAILLE_MAX] = "";
  f = fopen("montexte.txt", "r");
  if (f != NULL)
  {
        while (fgets(chaine, TAILLE_MAX, fichier) != NULL)
            // On lit le fichier tant qu'on ne recoit pas d'
            erreur (NULL)
        {
            printf("%s", chaine); // On affiche la chaene qu
                'on vient de lire
        }
    fclose(f);
  }
}
```



Lire depuis un fichier : fscanf()

- La fonction **fscanf** s'utilise de la même manière que **scanf**.
- Elle lit depuis un fichier qui doit avoir été écrit en respectant un format particulier.

```
int main()
{
    FILE* f = NULL;
    Etudiant e = {"Toto", 19};

    int age = 0;
    f = fopen("montexte.txt", "r");
    if (f != NULL)
    {
        fscanf(fichier, "%s %d", &e.nom, &e.age);
        printf("Nom : %d, Age : %d ans", e.nom, e.age);
        fclose(f);
    }
}
```



Lire depuis un fichier : fread()

Cette fonction permet un certain nombre de données depuis un fichier :

int **fread**(void *p, int taille, int nombre, FILE* nom_interne) ;

```
FILE* f = NULL;
Etudiant e, etu_tab[3];
f = fopen("montexte.txt", "r");
if (f != NULL)
{
    fread(&e, sizeof(Etudiant), 1, f);
    fread(etu_tab, sizeof(Etudiant), 3, f);
    fclose(f);
}
```

- **fread** lit dans le fichier "**nombre**" éléments, chacun de "**taille**" octets, et range les éléments lus en mémoire à l'adresse "**p**".
- **fread** retourne le nombre d'éléments effectivement lus.
- la lecture se fait à partir de la position courante du curseur, et déplace le curseur du nombre d'éléments lus.

Lire depuis un fichier : fread()

Cette fonction permet un certain nombre de données depuis un fichier :

int **fread**(void *p, int taille, int nombre, FILE* nom_interne) ;

```
FILE* f = NULL;
Etudiant e, etu_tab[3];
f = fopen("montexte.txt", "r");
if (f != NULL)
{
    fread(&e, sizeof(Etudiant), 1, f);
    fread(etu_tab, sizeof(Etudiant), 3, f);
    fclose(f);
}
```

- **fread** lit dans le fichier "**nombre**" éléments, chacun de "**taille**" octets, et range les éléments lus en mémoire à l'adresse "**p**".
- **fread** retourne le nombre d'éléments effectivement lus.
- la lecture se fait à partir de la position courante du curseur, et déplace le curseur du nombre d'éléments lus.

Lire depuis un fichier : fread()

Cette fonction permet un certain nombre de données depuis un fichier :

int **fread**(void *p, int taille, int nombre, FILE* nom_interne) ;

```
FILE* f = NULL;
Etudiant e, etu_tab[3];
f = fopen("montexte.txt", "r");
if (f != NULL)
{
    fread(&e, sizeof(Etudiant), 1, f);
    fread(etu_tab, sizeof(Etudiant), 3, f);
    fclose(f);
}
```

- **fread** lit dans le fichier "**nombre**" éléments, chacun de "**taille**" octets, et range les éléments lus en mémoire à l'adresse "**p**".
- **fread** retourne le nombre d'éléments effectivement lus.
- la lecture se fait à partir de la position courante du curseur, et déplace le curseur du nombre d'éléments lus.

Lire depuis un fichier : fread()

Cette fonction permet un certain nombre de données depuis un fichier :

int **fread**(void *p, int taille, int nombre, FILE* nom_interne) ;

```
FILE* f = NULL;
Etudiant e, etu_tab[3];
f = fopen("montexte.txt", "r");
if (f != NULL)
{
    fread(&e, sizeof(Etudiant), 1, f);
    fread(etu_tab, sizeof(Etudiant), 3, f);
    fclose(f);
}
```

- **fread** lit dans le fichier "**nombre**" éléments, chacun de "**taille**" octets, et range les éléments lus en mémoire à l'adresse "**p**".
- **fread** retourne le nombre d'éléments effectivement lus.
- la lecture se fait à partir de la position courante du curseur, et déplace le curseur du nombre d'éléments lus.

Se positionner dans un fichier

Il existe trois fonctions à connaître :

```
long ftell(FILE* ptrFichier);

int fseek(FILE* ptrFichier, long deplacement, int origine);

void rewind(FILE* ptrFichier);
```

- **ftell()** : renvoie la position actuelle du curseur sous la forme d'un long.
- **fseek()** : positionne le curseur à un endroit précis.
- **rewind()** : remet le curseur au début du fichier (c'est équivalent à demander à la fonction fseek de positionner le curseur au début).

Se positionner dans un fichier

Il existe trois fonctions à connaître :

```
long ftell(FILE* ptrFichier);

int fseek(FILE* ptrFichier, long deplacement, int origine);

void rewind(FILE* ptrFichier);
```

- **ftell() :** renvoie la position actuelle du curseur sous la forme d'un long.
- **fseek() :** positionne le curseur à un endroit précis.
- **rewind() :** remet le curseur au début du fichier (c'est équivalent à demander à la fonction fseek de positionner le curseur au début).

Se positionner dans un fichier

Il existe trois fonctions à connaître :

```
long ftell(FILE* ptrFichier);

int fseek(FILE* ptrFichier, long deplacement, int origine);

void rewind(FILE* ptrFichier);
```

- **ftell() :** renvoie la position actuelle du curseur sous la forme d'un long.
- **fseek() :** positionne le curseur à un endroit précis.
- **rewind() :** remet le curseur au début du fichier (c'est équivalent à demander à la fonction fseek de positionner le curseur au début).

Se positionner dans un fichier

Il existe trois fonctions à connaître :

```
long ftell(FILE* ptrFichier);

int fseek(FILE* ptrFichier, long deplacement, int origine);

void rewind(FILE* ptrFichier);
```

- **ftell() :** renvoie la position actuelle du curseur sous la forme d'un long.
- **fseek() :** positionne le curseur à un endroit précis.
- **rewind() :** remet le curseur au début du fichier (c'est équivalent à demander à la fonction fseek de positionner le curseur au début).

Se positionner dans un fichier : fseek()

La fonction **fseek** permet de déplacer le curseur d'un certain nombre de caractères (indiqué par **deplacement**) à partir de la position indiquée par **origine**.

```
int fseek(FILE* ptrFichier, long deplacement, int origine);
```

- Le nombre **deplacement** peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant au nombre **origine**, il prend l'une des trois constantes (généralement des **#define**) listées ci-dessous :
  1. **SEEK_SET** : indique le début du fichier ;
  2. **SEEK_CUR** : indique la position actuelle du curseur ;
  3. **SEEK_END** : indique la fin du fichier.

Se positionner dans un fichier : fseek()

La fonction **fseek** permet de déplacer le curseur d'un certain nombre de caractères (indiqué par **deplacement**) à partir de la position indiquée par **origine**.

```
int fseek(FILE* ptrFichier, long deplacement, int origine);
```

- Le nombre **deplacement** peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant au nombre **origine**, il prend l'une des trois constantes (généralement des **#define**) listées ci-dessous :
  1. **SEEK_SET** : indique le début du fichier ;
  2. **SEEK_CUR** : indique la position actuelle du curseur ;
  3. **SEEK_END** : indique la fin du fichier.

Se positionner dans un fichier : fseek()

La fonction **fseek** permet de déplacer le curseur d'un certain nombre de caractères (indiqué par **deplacement**) à partir de la position indiquée par **origine**.

```
int fseek(FILE* ptrFichier, long deplacement, int origine);
```

- Le nombre **deplacement** peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant au nombre **origine**, il prend l'une des trois constantes (généralement des **#define**) listées ci-dessous :
  1. **SEEK_SET** : indique le début du fichier ;
  2. **SEEK_CUR** : indique la position actuelle du curseur ;
  3. **SEEK_END** : indique la fin du fichier.

Se positionner dans un fichier : fseek()

La fonction **fseek** permet de déplacer le curseur d'un certain nombre de caractères (indiqué par **deplacement**) à partir de la position indiquée par **origine**.

```
int fseek(FILE* ptrFichier, long deplacement, int origine);
```

- Le nombre **deplacement** peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant au nombre **origine**, il prend l'une des trois constantes (généralement des **#define**) listées ci-dessous :
  1. **SEEK_SET** : indique le début du fichier ;
  2. **SEEK_CUR** : indique la position actuelle du curseur ;
  3. **SEEK_END** : indique la fin du fichier.

Se positionner dans un fichier : fseek()

La fonction **fseek** permet de déplacer le curseur d'un certain nombre de caractères (indiqué par **deplacement**) à partir de la position indiquée par **origine**.

```
int fseek(FILE* ptrFichier, long deplacement, int origine);
```

- Le nombre **deplacement** peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant au nombre **origine**, il prend l'une des trois constantes (généralement des **#define**) listées ci-dessous :
  1. **SEEK_SET** : indique le début du fichier ;
  2. **SEEK_CUR** : indique la position actuelle du curseur ;
  3. **SEEK_END** : indique la fin du fichier.

Se positionner dans un fichier : fseek()

La fonction **fseek** permet de déplacer le curseur d'un certain nombre de caractères (indiqué par **deplacement**) à partir de la position indiquée par **origine**.

```
int fseek(FILE* ptrFichier, long deplacement, int origine);
```

- Le nombre **deplacement** peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
- Quant au nombre **origine**, il prend l'une des trois constantes (généralement des **#define**) listées ci-dessous :
  1. **SEEK_SET** : indique le début du fichier ;
  2. **SEEK_CUR** : indique la position actuelle du curseur ;
  3. **SEEK_END** : indique la fin du fichier.

Se positionner dans un fichier : exemples

```
    FILE* f = NULL;
Etudiant e, etu_tab[3];
f = fopen("toto.txt", "r");
if (f != NULL)
{
    fread(&e, sizeof(Etudiant), 1, f);
    fread(etu_tab, sizeof(Etudiant), 3, f);
    fseek(f, 0, SEEK_SET);
    fseek(f, sizeof(Etudiant), SEEK_CUR);
    fseek(f, -2*sizeof(Etudiant), SEEK_END);
    printf("%d   %d\n", ftell(f), sizeof(Etudiant));
    rewind(f);
    printf("%d\n", ftell(f));
    fclose(f);
}
```

Renommer et supprimer un fichier

```
                        // prototypes :
int rename(const char* ancienNom, const char* nouveauNom);

int remove(const char* fichierASupprimer);

                        // utilisation
void main()
{
    rename("test.txt", "test1.txt");
    remove("test1.txt");
}
```

- **rename :** permet de renommer un fichier ;
- **remove :** permet de supprimer un fichier sans demander de confirmation .

Ces fonctions ne nécessitent pas de pointeur de fichier, il suffira simplement d'indiquer le nom du fichier à renommer ou à supprimer.

Renommer et supprimer un fichier

```
                        // prototypes :
int rename(const char* ancienNom, const char* nouveauNom);

int remove(const char* fichierASupprimer);

                        // utilisation
void main()
{
    rename("test.txt", "test1.txt");
    remove("test1.txt");
}
```

- **rename :** permet de renommer un fichier ;
- **remove :** permet de supprimer un fichier sans demander de confirmation .

Ces fonctions ne nécessitent pas de pointeur de fichier, il suffira simplement d'indiquer le nom du fichier à renommer ou à supprimer.

Renommer et supprimer un fichier

```
                        // prototypes :
int rename(const char* ancienNom, const char* nouveauNom);

int remove(const char* fichierASupprimer);

                        // utilisation
void main()
{
    rename("test.txt", "test1.txt");
    remove("test1.txt");
}
```

- **rename :** permet de renommer un fichier ;
- **remove :** permet de supprimer un fichier sans demander de confirmation .

Ces fonctions ne nécessitent pas de pointeur de fichier, il suffira simplement d'indiquer le nom du fichier à renommer ou à supprimer.

Renommer et supprimer un fichier

```
                        // prototypes :
int rename(const char* ancienNom, const char* nouveauNom);

int remove(const char* fichierASupprimer);

                        // utilisation
void main()
{
    rename("test.txt", "test1.txt");
    remove("test1.txt");
}
```

- **rename :** permet de renommer un fichier ;
- **remove :** permet de supprimer un fichier sans demander de confirmation .

Ces fonctions ne nécessitent pas de pointeur de fichier, il suffira simplement d'indiquer le nom du fichier à renommer ou à supprimer.

Initiation à l'algorithmique
Les pointeurs

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

Introduction

Exercice

Écrire une fonction à laquelle on envoie une durée exprimée en minutes. Celle-ci renverrait le nombre d'heures et minutes correspondantes :

- si on envoie 45, la fonction renvoie 0 heure et 45 minutes ;
- si on envoie 60, la fonction renvoie 1 heure et 0 minutes ;
- si on envoie 90, la fonction renvoie 1 heure et 30 minutes.

Problèmes :

- En effet, on ne peut renvoyer qu'une valeur par fonction !
- On peut utiliser des variables globales mais cette pratique est fortement déconseillée.

Introduction

Exercice

Écrire une fonction à laquelle on envoie une durée exprimée en minutes. Celle-ci renverrait le nombre d'heures et minutes correspondantes :

- si on envoie 45, la fonction renvoie 0 heure et 45 minutes ;
- si on envoie 60, la fonction renvoie 1 heure et 0 minutes ;
- si on envoie 90, la fonction renvoie 1 heure et 30 minutes.

Problèmes :

- En effet, on ne peut renvoyer qu'une valeur par fonction !
- On peut utiliser des variables globales mais cette pratique est fortement déconseillée.

Introduction

Exercice

Écrire une fonction à laquelle on envoie une durée exprimée en minutes. Celle-ci renverrait le nombre d'heures et minutes correspondantes :

- si on envoie 45, la fonction renvoie 0 heure et 45 minutes ;
- si on envoie 60, la fonction renvoie 1 heure et 0 minutes ;
- si on envoie 90, la fonction renvoie 1 heure et 30 minutes.

Problèmes :

- En effet, on ne peut renvoyer qu'une valeur par fonction !
- On peut utiliser des variables globales mais cette pratique est fortement déconseillée.

Introduction

Exercice

Écrire une fonction à laquelle on envoie une durée exprimée en minutes. Celle-ci renverrait le nombre d'heures et minutes correspondantes :

- si on envoie 45, la fonction renvoie 0 heure et 45 minutes ;
- si on envoie 60, la fonction renvoie 1 heure et 0 minutes ;
- si on envoie 90, la fonction renvoie 1 heure et 30 minutes.

Problèmes :

- En effet, on ne peut renvoyer qu'une valeur par fonction !
- On peut utiliser des variables globales mais cette pratique est fortement déconseillée.

Introduction

Exercice

Écrire une fonction à laquelle on envoie une durée exprimée en minutes. Celle-ci renverrait le nombre d'heures et minutes correspondantes :

- si on envoie 45, la fonction renvoie 0 heure et 45 minutes ;
- si on envoie 60, la fonction renvoie 1 heure et 0 minutes ;
- si on envoie 90, la fonction renvoie 1 heure et 30 minutes.

Problèmes :

- En effet, on ne peut renvoyer qu'une valeur par fonction !
- On peut utiliser des variables globales mais cette pratique est fortement déconseillée.

Introduction

Exercice

Écrire une fonction à laquelle on envoie une durée exprimée en minutes. Celle-ci renverrait le nombre d'heures et minutes correspondantes :

- si on envoie 45, la fonction renvoie 0 heure et 45 minutes ;
- si on envoie 60, la fonction renvoie 1 heure et 0 minutes ;
- si on envoie 90, la fonction renvoie 1 heure et 30 minutes.

Problèmes :

- En effet, on ne peut renvoyer qu'une valeur par fonction !
- On peut utiliser des variables globales mais cette pratique est fortement déconseillée.

Solution : notion de pointeur

On doit donc apprendre à se servir de la notion de pointeur...

Adresse	Valeur
0	145
1	3.8028322
2	0.827551
3	3901930
...	...
3 448 765 900 126 (et des poussières)	940.5118

Adresse vs. Valeur

Quand vous créez une variable age de type int par exemple, en tapant :

```
int age = 10;
```

1. votre programme demande au système d'exploitation (Windows, par exemple) la permission d'utiliser un peu de mémoire.
2. Le système d'exploitation répond en indiquant à quelle adresse en mémoire il vous laisse le droit d'inscrire votre nombre.
3. La valeur 10 est inscrite quelque part en mémoire, disons par exemple à l'adresse 4655.
4. Ensuite, le compilateur remplace le mot age dans votre programme par l'adresse 4655 à l'exécution
5. Donc, l'ordinateur se rendra toujours à l'adresse 4655 pour récupérer la valeur de la variable « age ».

Adresse vs. Valeur

Quand vous créez une variable age de type int par exemple, en tapant :

```
int age = 10;
```

1. votre programme demande au système d'exploitation (Windows, par exemple) la permission d'utiliser un peu de mémoire.
2. Le système d'exploitation répond en indiquant à quelle adresse en mémoire il vous laisse le droit d'inscrire votre nombre.
3. La valeur 10 est inscrite quelque part en mémoire, disons par exemple à l'adresse 4655.
4. Ensuite, le compilateur remplace le mot age dans votre programme par l'adresse 4655 à l'exécution
5. Donc, l'ordinateur se rendra toujours à l'adresse 4655 pour récupérer la valeur de la variable « age ».

Adresse vs. Valeur

Quand vous créez une variable age de type int par exemple, en tapant :

```
int age = 10;
```

1. votre programme demande au système d'exploitation (Windows, par exemple) la permission d'utiliser un peu de mémoire.
2. Le système d'exploitation répond en indiquant à quelle adresse en mémoire il vous laisse le droit d'inscrire votre nombre.
3. La valeur 10 est inscrite quelque part en mémoire, disons par exemple à l'adresse 4655.
4. Ensuite, le compilateur remplace le mot age dans votre programme par l'adresse 4655 à l'exécution
5. Donc, l'ordinateur se rendra toujours à l'adresse 4655 pour récupérer la valeur de la variable « age ».

Adresse vs. Valeur

Quand vous créez une variable age de type int par exemple, en tapant :

```
int age = 10;
```

1. votre programme demande au système d'exploitation (Windows, par exemple) la permission d'utiliser un peu de mémoire.
2. Le système d'exploitation répond en indiquant à quelle adresse en mémoire il vous laisse le droit d'inscrire votre nombre.
3. La valeur 10 est inscrite quelque part en mémoire, disons par exemple à l'adresse 4655.
4. Ensuite, le compilateur remplace le mot age dans votre programme par l'adresse 4655 à l'exécution
5. Donc, l'ordinateur se rendra toujours à l'adresse 4655 pour récupérer la valeur de la variable « age ».

Adresse vs. Valeur

Quand vous créez une variable age de type int par exemple, en tapant :

```
int age = 10;
```

1. votre programme demande au système d'exploitation (Windows, par exemple) la permission d'utiliser un peu de mémoire.
2. Le système d'exploitation répond en indiquant à quelle adresse en mémoire il vous laisse le droit d'inscrire votre nombre.
3. La valeur 10 est inscrite quelque part en mémoire, disons par exemple à l'adresse 4655.
4. Ensuite, le compilateur remplace le mot age dans votre programme par l'adresse 4655 à l'exécution
5. Donc, l'ordinateur se rendra toujours à l'adresse 4655 pour récupérer la valeur de la variable « age ».

Adresse vs. valeur

Comment récupérer l'adresse d'une variable ?

Exemple :

```c
int age = 10;
printf("La variable age vaut : %d", age);
printf("L'adresse de la variable age est : %p", &age);
```

```
La variable age vaut : 10
L'adresse de la variable age est : 0028FF1C
```

Donc à retenir :

1. age : désigne la valeur de la variable ;
2. & age : désigne l'adresse de la variable.

Adresse vs. valeur

Comment récupérer l'adresse d'une variable ?

Exemple :

```c
int age = 10;
printf("La variable age vaut : %d", age);
printf("L'adresse de la variable age est : %p", &age);
```

```
La variable age vaut : 10
L'adresse de la variable age est : 0028FF1C
```

Donc à retenir :

1. age : désigne la valeur de la variable ;
2. & age : désigne l'adresse de la variable.

Adresse vs. valeur

Comment récupérer l'adresse d'une variable ?

Exemple :

```c
int age = 10;
printf("La variable age vaut : %d", age);
printf("L'adresse de la variable age est : %p", &age);
```

```
La variable age vaut : 10
L'adresse de la variable age est : 0028FF1C
```

Donc à retenir :

1. age : désigne la valeur de la variable ;
2. & age : désigne l'adresse de la variable.

C'est quoi un pointeur ?

- Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres.
- Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses
- Ce sont justement ce qu'on appelle des pointeurs.

Déclaration d'un pointeur :

```
int *monPointeur;
```

- Pour créer une variable de type pointeur, on doit rajouter le symbole * devant le nom de la variable.
- Notez qu'on peut aussi écrire int* monPointeur ;

C'est quoi un pointeur ?

- Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres.
- Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses
- Ce sont justement ce qu'on appelle des pointeurs.

Déclaration d'un pointeur :

```
int *monPointeur;
```

- Pour créer une variable de type pointeur, on doit rajouter le symbole * devant le nom de la variable.
- Notez qu'on peut aussi écrire int* monPointeur ;

C'est quoi un pointeur ?

- Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres.
- Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses
- Ce sont justement ce qu'on appelle des pointeurs.

Déclaration d'un pointeur :

```
int *monPointeur;
```

- Pour créer une variable de type pointeur, on doit rajouter le symbole * devant le nom de la variable.
- Notez qu'on peut aussi écrire int* monPointeur ;

C'est quoi un pointeur ?

- Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres.
- Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses
- Ce sont justement ce qu'on appelle des pointeurs.

Déclaration d'un pointeur :

```
int *monPointeur;
```

- Pour créer une variable de type pointeur, on doit rajouter le symbole * devant le nom de la variable.
- Notez qu'on peut aussi écrire int* monPointeur ;

C'est quoi un pointeur ?

- Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres.
- Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses
- Ce sont justement ce qu'on appelle des pointeurs.

Déclaration d'un pointeur :

```
int *monPointeur;
```

- Pour créer une variable de type pointeur, on doit rajouter le symbole * devant le nom de la variable.
- Notez qu'on peut aussi écrire int* monPointeur ;

C'est quoi un pointeur ?

- Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres.
- Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses
- Ce sont justement ce qu'on appelle des pointeurs.

Déclaration d'un pointeur :

```
int *monPointeur;
```

- Pour créer une variable de type pointeur, on doit rajouter le symbole * devant le nom de la variable.
- Notez qu'on peut aussi écrire int* monPointeur ;

Initialiser et affecter une adresse à un pointeur

- Pour initialiser un pointeur, c'est-à-dire lui donner une valeur par défaut, on n'utilise généralement pas le nombre 0 mais le mot-clé NULL (veillez à l'écrire en majuscules)

Exemple

```c
int *monPointeur = NULL;

int age = 10;
int *pointeurSurAge = &age;
```

- La première ligne réserve une case en mémoire pour contenir une adresse mais que cette case ne contient aucune adresse pour le moment.
- La seconde ligne signifie : « Créer une variable de type int dont la valeur vaut 10 ».
- La dernière ligne signifie : « Créer une variable de type pointeur dont la valeur vaut l'**adresse** de la variable age ».

Initialiser et affecter une adresse à un pointeur

- Pour initialiser un pointeur, c'est-à-dire lui donner une valeur par défaut, on n'utilise généralement pas le nombre 0 mais le mot-clé NULL (veillez à l'écrire en majuscules)

Exemple

```
int *monPointeur = NULL;

int age = 10;
int *pointeurSurAge = &age;
```

- La première ligne réserve une case en mémoire pour contenir une adresse mais que cette case ne contient aucune adresse pour le moment.
- La seconde ligne signifie : « Créer une variable de type int dont la valeur vaut 10 ».
- La dernière ligne signifie : « Créer une variable de type pointeur dont la valeur vaut l'**adresse** de la variable age ».

Initialiser et affecter une adresse à un pointeur

- Pour initialiser un pointeur, c'est-à-dire lui donner une valeur par défaut, on n'utilise généralement pas le nombre 0 mais le mot-clé NULL (veillez à l'écrire en majuscules)

Exemple

```
int *monPointeur = NULL;

int age = 10;
int *pointeurSurAge = &age;
```

- La première ligne réserve une case en mémoire pour contenir une adresse mais que cette case ne contient aucune adresse pour le moment.
- La seconde ligne signifie : « Créer une variable de type int dont la valeur vaut 10 ».
- La dernière ligne signifie : « Créer une variable de type pointeur dont la valeur vaut l'**adresse** de la variable age ».

Initialiser et affecter une adresse à un pointeur

- Pour initialiser un pointeur, c'est-à-dire lui donner une valeur par défaut, on n'utilise généralement pas le nombre 0 mais le mot-clé NULL (veillez à l'écrire en majuscules)

Exemple

```
int *monPointeur = NULL;

int age = 10;
int *pointeurSurAge = &age;
```

- La première ligne réserve une case en mémoire pour contenir une adresse mais que cette case ne contient aucune adresse pour le moment.
- La seconde ligne signifie : « Créer une variable de type int dont la valeur vaut 10 ».
- La dernière ligne signifie : « Créer une variable de type pointeur dont la valeur vaut l'**adresse** de la variable age ».

Initialiser et affecter une adresse à un pointeur

- Pour initialiser un pointeur, c'est-à-dire lui donner une valeur par défaut, on n'utilise généralement pas le nombre 0 mais le mot-clé NULL (veillez à l'écrire en majuscules)

Exemple

```
int *monPointeur = NULL;

int age = 10;
int *pointeurSurAge = &age;
```

- La première ligne réserve une case en mémoire pour contenir une adresse mais que cette case ne contient aucune adresse pour le moment.
- La seconde ligne signifie : « Créer une variable de type int dont la valeur vaut 10 ».
- La dernière ligne signifie : « Créer une variable de type pointeur dont la valeur vaut l'**adresse** de la variable age ».

Les pointeurs ont-ils un type ?

- Il n'y a pas de type « pointeur » comme il y a un type int et un type double. On n'écrit donc pas pointeur pointeurSurAge ;
- On utilise le symbole *, mais on continue à indiquer quel est le type de la variable dont le pointeur va contenir l'adresse.

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;
```

- Comme le pointeur pointeurSurAge va contenir l'adresse de la variable age (qui est de type int), alors le pointeur doit être de type int*.
- Si la variable age avait été de type double, alors on aurait dû écrire double *monPointeur.

Les pointeurs ont-ils un type ?

- Il n'y a pas de type « pointeur » comme il y a un type int et un type double. On n'écrit donc pas pointeur pointeurSurAge ;
- On utilise le symbole *, mais on continue à indiquer quel est le type de la variable dont le pointeur va contenir l'adresse.

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;
```

- Comme le pointeur pointeurSurAge va contenir l'adresse de la variable age (qui est de type int), alors le pointeur doit être de type int*.
- Si la variable age avait été de type double, alors on aurait dû écrire double *monPointeur.

Les pointeurs ont-ils un type ?

- Il n'y a pas de type « pointeur » comme il y a un type int et un type double. On n'écrit donc pas pointeur pointeurSurAge ;
- On utilise le symbole *, mais on continue à indiquer quel est le type de la variable dont le pointeur va contenir l'adresse.

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;
```

- Comme le pointeur pointeurSurAge va contenir l'adresse de la variable age (qui est de type int), alors le pointeur doit être de type int*.
- Si la variable age avait été de type double, alors on aurait dû écrire double *monPointeur.

Les pointeurs ont-ils un type ?

- Il n'y a pas de type « pointeur » comme il y a un type int et un type double. On n'écrit donc pas pointeur pointeurSurAge ;
- On utilise le symbole *, mais on continue à indiquer quel est le type de la variable dont le pointeur va contenir l'adresse.

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;
```

- Comme le pointeur pointeurSurAge va contenir l'adresse de la variable age (qui est de type int), alors le pointeur doit être de type int*.
- Si la variable age avait été de type double, alors on aurait dû écrire double *monPointeur.

Les pointeurs ont-ils un type ?

- Il n'y a pas de type « pointeur » comme il y a un type int et un type double. On n'écrit donc pas pointeur pointeurSurAge ;
- On utilise le symbole *, mais on continue à indiquer quel est le type de la variable dont le pointeur va contenir l'adresse.

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;
```

- Comme le pointeur pointeurSurAge va contenir l'adresse de la variable age (qui est de type int), alors le pointeur doit être de type int*.
- Si la variable age avait été de type double, alors on aurait dû écrire double *monPointeur.

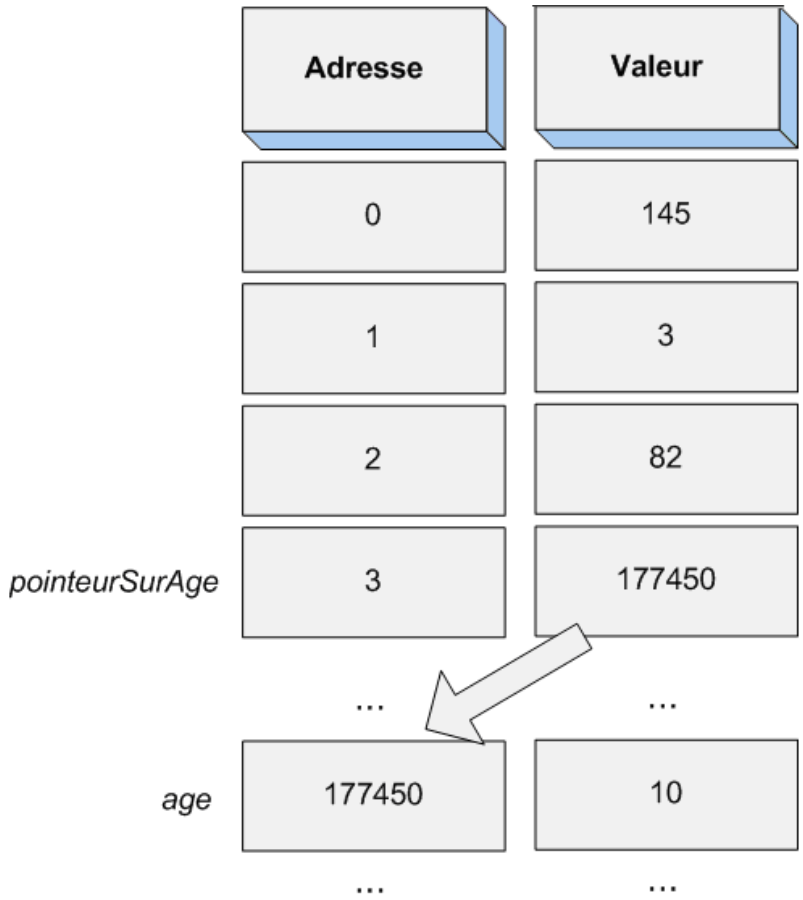
Schéma explicatif

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;
```

Vocabulaire :

On dit que le pointeur **pointeurSurAge** pointe sur la variable **age**.



La valeur d'un pointeur

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;

printf("%d", pointeurSurAge);
printf("%d", *pointeurSurAge);
printf("%d", &pointeurSurAge);
```

- Le premier appel de printf affiche la valeur de pointeurSurAge, et sa valeur c'est l'adresse de la variable age (177450).
- Le second appel de printf affiche la valeur de la variable se trouvant à l'adresse indiquée dans pointeurSurAge.
- Il faut donc placer le symbole * devant le nom du pointeur pour récupérer la valeur de la variable se trouvant à l'adresse indiquée dans un pointeur.
- ler dernier appel de printf affiche l'adresse à laquelle se trouve le pointeur (ici, c'est 3).

La valeur d'un pointeur

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;

printf("%d", pointeurSurAge);
printf("%d", *pointeurSurAge);
printf("%d", &pointeurSurAge);
```

- Le premier appel de printf affiche la valeur de pointeurSurAge, et sa valeur c'est l'adresse de la variable age (177450).
- Le second appel de printf affiche la valeur de la variable se trouvant à l'adresse indiquée dans pointeurSurAge.
- Il faut donc placer le symbole * devant le nom du pointeur pour récupérer la valeur de la variable se trouvant à l'adresse indiquée dans un pointeur.
- ler dernier appel de printf affiche l'adresse à laquelle se trouve le pointeur (ici, c'est 3).

La valeur d'un pointeur

Exemple :

```c
int age = 10;
int *pointeurSurAge = &age;

printf("%d", pointeurSurAge);
printf("%d", *pointeurSurAge);
printf("%d", &pointeurSurAge);
```

- Le premier appel de printf affiche la valeur de pointeurSurAge, et sa valeur c'est l'adresse de la variable age (177450).
- Le second appel de printf affiche la valeur de la variable se trouvant à l'adresse indiquée dans pointeurSurAge.
- Il faut donc placer le symbole * devant le nom du pointeur pour récupérer la valeur de la variable se trouvant à l'adresse indiquée dans un pointeur.
- ler dernier appel de printf affiche l'adresse à laquelle se trouve le pointeur (ici, c'est 3).

La valeur d'un pointeur

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;

printf("%d", pointeurSurAge);
printf("%d", *pointeurSurAge);
printf("%d", &pointeurSurAge);
```

- Le premier appel de printf affiche la valeur de pointeurSurAge, et sa valeur c'est l'adresse de la variable age (177450).
- Le second appel de printf affiche la valeur de la variable se trouvant à l'adresse indiquée dans pointeurSurAge.
- Il faut donc placer le symbole * devant le nom du pointeur pour récupérer la valeur de la variable se trouvant à l'adresse indiquée dans un pointeur.
- ler dernier appel de printf affiche l'adresse à laquelle se trouve le pointeur (ici, c'est 3).

La valeur d'un pointeur

Exemple :

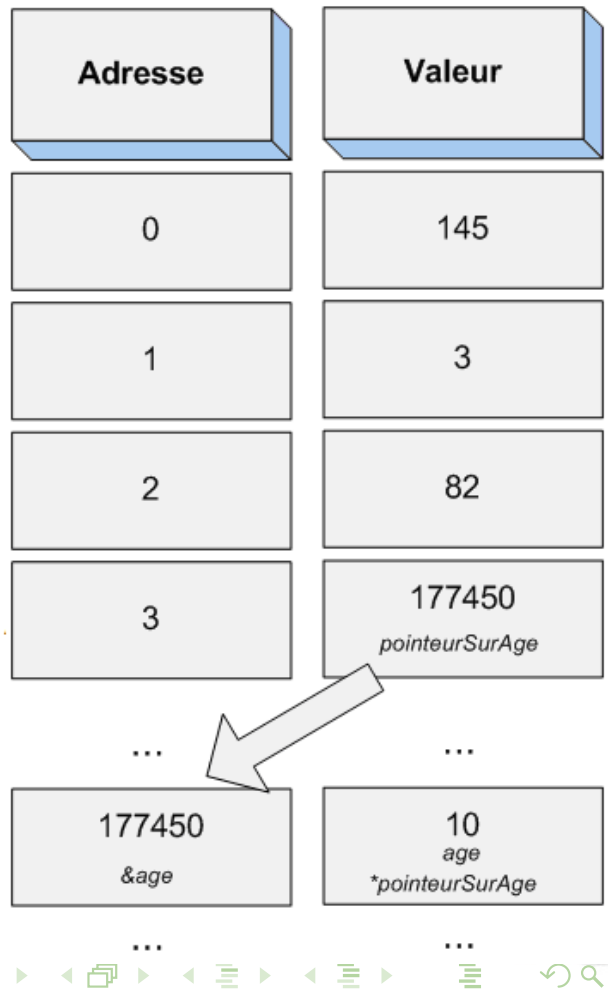
```
int age = 10;
int *pointeurSurAge = &age;

printf("%d", pointeurSurAge);
printf("%d", *pointeurSurAge);
printf("%d", &pointeurSurAge);
```

- Le premier appel de printf affiche la valeur de pointeurSurAge, et sa valeur c'est l'adresse de la variable age (177450).
- Le second appel de printf affiche la valeur de la variable se trouvant à l'adresse indiquée dans pointeurSurAge.
- Il faut donc placer le symbole * devant le nom du pointeur pour récupérer la valeur de la variable se trouvant à l'adresse indiquée dans un pointeur.
- ler dernier appel de printf affiche l'adresse à laquelle se trouve le pointeur (ici, c'est 3).

À retenir

Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

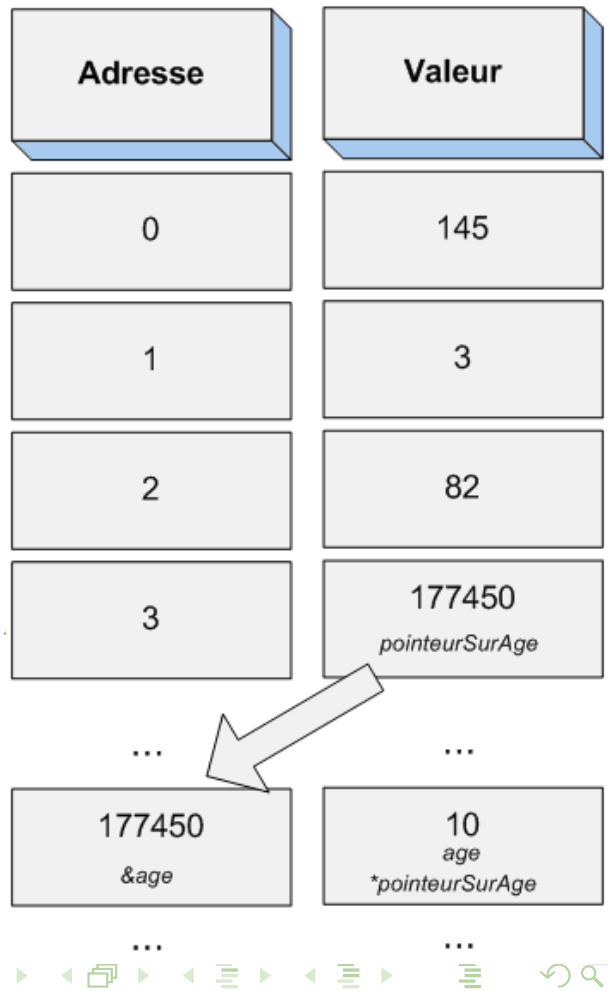
- Le nom d'une variable reste toujours lié à la même adresse.
  - **age** signifie : « Je veux la valeur de la variable age »,
  - **& age** signifie : « Je veux l'adresse à laquelle se trouve la variable age » ;
- Un pointeur est une variable qui peut **pointer** sur différentes adresses.
  - pointeurSurAge signifie : « Je veux la valeur de pointeurSurAge » (cette valeur étant une adresse),
  - *pointeurSurAge signifie : « Je veux la valeur de la variable pointée à l'adresse contenue dans pointeurSurAge »



À retenir

Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

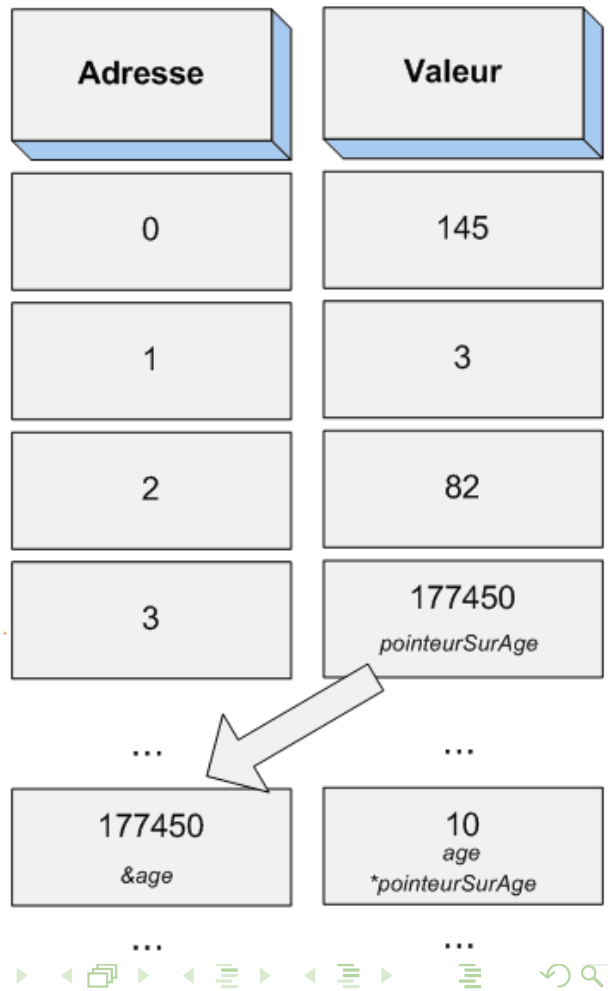
- Le nom d'une variable reste toujours lié à la même adresse.
  - **age** signifie : « Je veux la valeur de la variable age »,
  - **& age** signifie : « Je veux l'adresse à laquelle se trouve la variable age » ;
- Un pointeur est une variable qui peut **pointer** sur différentes adresses.
  - pointeurSurAge signifie : « Je veux la valeur de pointeurSurAge » (cette valeur étant une adresse),
  - *pointeurSurAge signifie : « Je veux la valeur de la variable qui se trouve à l'adresse contenue dans pointeurSurAge ».



À retenir

Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

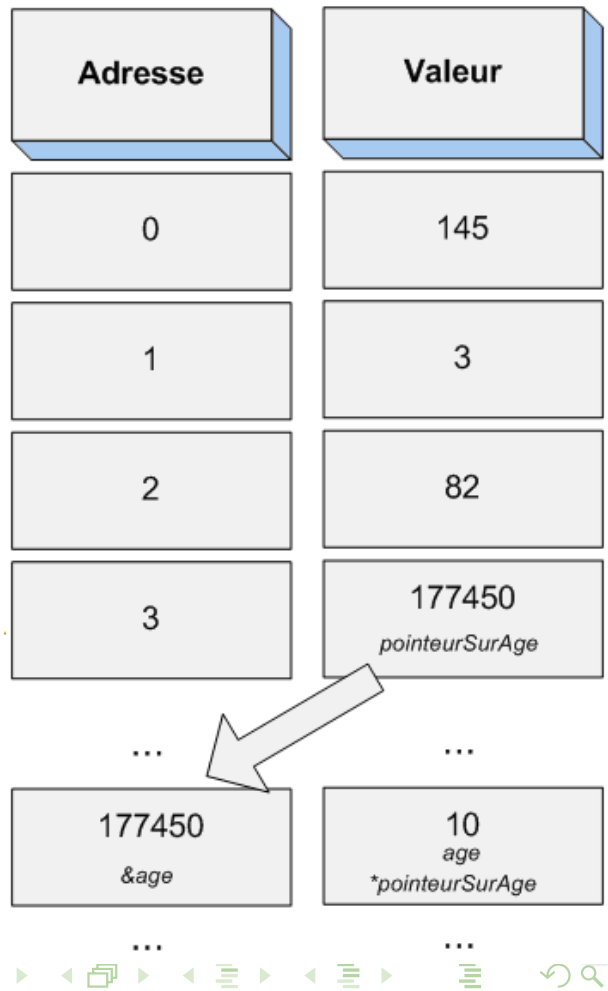
- Le nom d'une variable reste toujours lié à la même adresse.
  - **age** signifie : « Je veux la valeur de la variable age »,
  - **& age** signifie : « Je veux l'adresse à laquelle se trouve la variable age » ;
- Un pointeur est une variable qui peut **pointer** sur différentes adresses.
  - pointeurSurAge signifie : « Je veux la valeur de pointeurSurAge » (cette valeur étant une adresse),
  - *pointeurSurAge signifie : « Je veux la valeur de la variable qui se trouve à l'adresse contenue dans pointeurSurAge ».



À retenir

Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

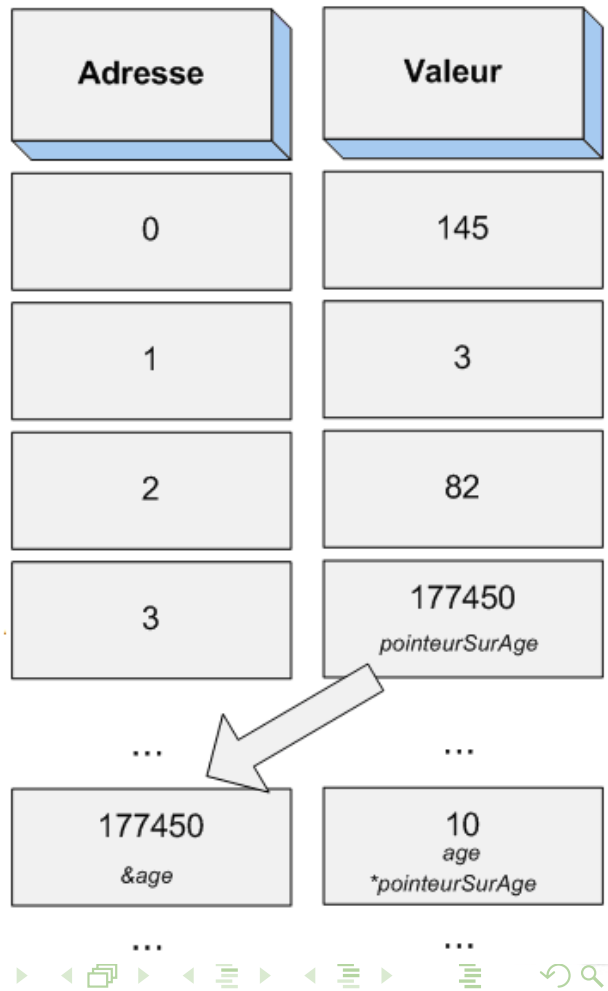
- Le nom d'une variable reste toujours lié à la même adresse.
  - **age** signifie : « Je veux la valeur de la variable age »,
  - **& age** signifie : « Je veux l'adresse à laquelle se trouve la variable age » ;
- Un pointeur est une variable qui peut **pointer** sur différentes adresses.
  - **pointeurSurAge** signifie : « Je veux la valeur de pointeurSurAge » (cette valeur étant une adresse),
  - ***pointeurSurAge** signifie : « Je veux la valeur de la variable qui se trouve à l'adresse contenue dans pointeurSurAge ».



À retenir

Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

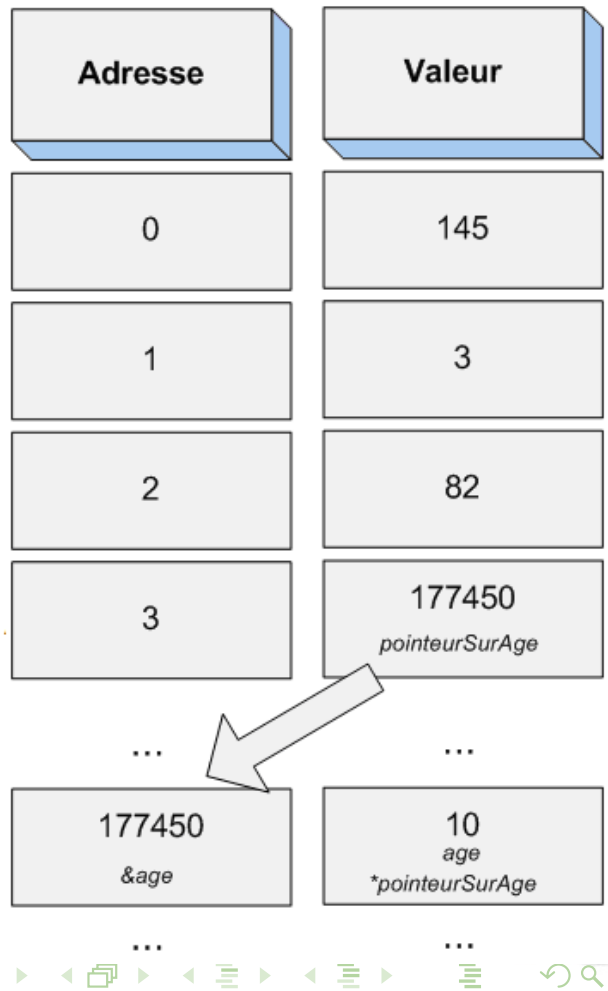
- Le nom d'une variable reste toujours lié à la même adresse.
  - **age** signifie : « Je veux la valeur de la variable age »,
  - **& age** signifie : « Je veux l'adresse à laquelle se trouve la variable age » ;
- Un pointeur est une variable qui peut **pointer** sur différentes adresses.
  - **pointeurSurAge** signifie : « Je veux la valeur de pointeurSurAge » (cette valeur étant une adresse),
  - ***pointeurSurAge** signifie : « Je veux la valeur de la variable qui se trouve à l'adresse contenue dans pointeurSurAge ».



À retenir

Les pointeurs et les noms de variables ont le même rôle : Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence :

- Le nom d'une variable reste toujours lié à la même adresse.
  - **age** signifie : « Je veux la valeur de la variable age »,
  - **& age** signifie : « Je veux l'adresse à laquelle se trouve la variable age » ;
- Un pointeur est une variable qui peut **pointer** sur différentes adresses.
  - **pointeurSurAge** signifie : « Je veux la valeur de pointeurSurAge » (cette valeur étant une adresse),
  - ***pointeurSurAge** signifie : « Je veux la valeur de la variable qui se trouve à l'adresse contenue dans pointeurSurAge ».



Déclaration vs. Utilisation d'un pointeur

Attention :

Ne pas confondre les différentes significations de l'étoile !

1. Lorsque on **déclare** un pointeur, l'étoile sert juste à indiquer qu'on veut créer un pointeur : int *pointeurSurAge ;.
2. En revanche, lorsqu'ensuite on **utilise** le pointeur en écrivant printf("%d", *pointeurSurAge) ;, cela ne signifie pas « on veut créer un pointeur » mais : « on veut la valeur de la variable sur laquelle pointe le pointeurSurAge ».

Exemple :

```
int age = 10;
int *pointeurSurAge = &age;      //declaration d'un pointeur

*pointeurSurAge += 1;            //utilisation du pointeur
printf("%d", *pointeurSurAge);   //utilisation du pointeur
```

Passage par adresse des paramètres d'une fonction

Le gros intérêt des pointeurs (mais ce n'est pas le seul) est qu'on peut les envoyer à des fonctions pour qu'elles modifient directement une variable en mémoire, et non une copie comme on l'a déjà vu auparavant.

Exemple :

```
void triplePointeur(int *pointeurSurNombre);
int main()
{
    int nombre = 5;

    triplePointeur(&nombre); // On envoie l'adresse de
        nombre a la fonction
    printf("%d", nombre); // On affiche la variable nombre.

    return 0;
}
void triplePointeur(int *pointeurSurNombre)
{
    *pointeurSurNombre *= 3;
}
```

Passage par adresse des paramètres d'une fonctions

En exécutant le programme du slide précédent, voici ce qu'il se passe dans l'ordre, en partant du début du main :

1. une variable **nombre** est créée dans le main. On lui affecte la valeur 5.
2. on appelle la fonction **triplePointeur**. On lui envoie en paramètre l'adresse de notre variable nombre ;
3. la fonction **triplePointeur** reçoit cette adresse dans **pointeurSurNombre**. À l'intérieur de la fonction **triplePointeur**, on a donc un pointeur **pointeurSurNombre** qui contient l'adresse de la variable **nombre** ;
4. maintenant qu'on a un pointeur sur **nombre**, on peut modifier directement la variable **nombre** en mémoire ! Il suffit d'utiliser ***pointeurSurNombre** pour désigner la variable **nombre** ! Pour l'exemple, on multiplie simplement la variable **nombre** par 3 ;
5. de retour dans la fonction main, notre **nombre** vaut maintenant 15 car la fonction **triplePointeur** a modifié directement la valeur de **nombre**.

Passage par adresse des paramètres d'une fonctions

En exécutant le programme du slide précédent, voici ce qu'il se passe dans l'ordre, en partant du début du main :

1. une variable **nombre** est créée dans le main. On lui affecte la valeur 5.
2. on appelle la fonction **triplePointeur**. On lui envoie en paramètre l'adresse de notre variable nombre ;
3. la fonction **triplePointeur** reçoit cette adresse dans **pointeurSurNombre**. À l'intérieur de la fonction **triplePointeur**, on a donc un pointeur **pointeurSurNombre** qui contient l'adresse de la variable **nombre** ;
4. maintenant qu'on a un pointeur sur **nombre**, on peut modifier directement la variable **nombre** en mémoire ! Il suffit d'utiliser ***pointeurSurNombre** pour désigner la variable **nombre** ! Pour l'exemple, on multiplie simplement la variable **nombre** par 3 ;
5. de retour dans la fonction main, notre **nombre** vaut maintenant 15 car la fonction **triplePointeur** a modifié directement la valeur de **nombre**.

Passage par adresse des paramètres d'une fonctions

En exécutant le programme du slide précédent, voici ce qu'il se passe dans l'ordre, en partant du début du main :

1. une variable **nombre** est créée dans le main. On lui affecte la valeur 5.
2. on appelle la fonction **triplePointeur**. On lui envoie en paramètre l'adresse de notre variable nombre ;
3. la fonction **triplePointeur** reçoit cette adresse dans **pointeurSurNombre**. À l'intérieur de la fonction **triplePointeur**, on a donc un pointeur **pointeurSurNombre** qui contient l'adresse de la variable **nombre** ;
4. maintenant qu'on a un pointeur sur **nombre**, on peut modifier directement la variable **nombre** en mémoire ! Il suffit d'utiliser ***pointeurSurNombre** pour désigner la variable **nombre** ! Pour l'exemple, on multiplie simplement la variable **nombre** par 3 ;
5. de retour dans la fonction main, notre **nombre** vaut maintenant 15 car la fonction **triplePointeur** a modifié directement la valeur de **nombre**.

Passage par adresse des paramètres d'une fonctions

En exécutant le programme du slide précédent, voici ce qu'il se passe dans l'ordre, en partant du début du main :

1. une variable **nombre** est créée dans le main. On lui affecte la valeur 5.
2. on appelle la fonction **triplePointeur**. On lui envoie en paramètre l'adresse de notre variable nombre ;
3. la fonction **triplePointeur** reçoit cette adresse dans **pointeurSurNombre**. À l'intérieur de la fonction **triplePointeur**, on a donc un pointeur **pointeurSurNombre** qui contient l'adresse de la variable **nombre** ;
4. maintenant qu'on a un pointeur sur **nombre**, on peut modifier directement la variable **nombre** en mémoire ! Il suffit d'utiliser ***pointeurSurNombre** pour désigner la variable **nombre** ! Pour l'exemple, on multiplie simplement la variable **nombre** par 3 ;
5. de retour dans la fonction main, notre **nombre** vaut maintenant 15 car la fonction **triplePointeur** a modifié directement la valeur de **nombre**.

Passage par adresse des paramètres d'une fonctions

En exécutant le programme du slide précédent, voici ce qu'il se passe dans l'ordre, en partant du début du main :

1. une variable **nombre** est créée dans le main. On lui affecte la valeur 5.
2. on appelle la fonction **triplePointeur**. On lui envoie en paramètre l'adresse de notre variable nombre ;
3. la fonction **triplePointeur** reçoit cette adresse dans **pointeurSurNombre**. À l'intérieur de la fonction **triplePointeur**, on a donc un pointeur **pointeurSurNombre** qui contient l'adresse de la variable **nombre** ;
4. maintenant qu'on a un pointeur sur **nombre**, on peut modifier directement la variable **nombre** en mémoire ! Il suffit d'utiliser ***pointeurSurNombre** pour désigner la variable **nombre** ! Pour l'exemple, on multiplie simplement la variable **nombre** par 3 ;
5. de retour dans la fonction main, notre **nombre** vaut maintenant 15 car la fonction **triplePointeur** a modifié directement la valeur de **nombre**.

Passage par adresse des paramètres d'une fonctions

En exécutant le programme du slide précédent, voici ce qu'il se passe dans l'ordre, en partant du début du main :

1. une variable **nombre** est créée dans le main. On lui affecte la valeur 5.
2. on appelle la fonction **triplePointeur**. On lui envoie en paramètre l'adresse de notre variable nombre ;
3. la fonction **triplePointeur** reçoit cette adresse dans **pointeurSurNombre**. À l'intérieur de la fonction **triplePointeur**, on a donc un pointeur **pointeurSurNombre** qui contient l'adresse de la variable **nombre** ;
4. maintenant qu'on a un pointeur sur **nombre**, on peut modifier directement la variable **nombre** en mémoire ! Il suffit d'utiliser ***pointeurSurNombre** pour désigner la variable **nombre** ! Pour l'exemple, on multiplie simplement la variable **nombre** par 3 ;
5. de retour dans la fonction main, notre **nombre** vaut maintenant 15 car la fonction **triplePointeur** a modifié directement la valeur de **nombre**.

Et si on revenait à notre exercice de départ ?

Solution

```
void decoupeMinutes(int, int* pointeurHeures, int*
    pointeurMinutes);
int main()
{
    int duree=0;
    int heures = 0, minutes =0 ;
    printf("Donnez le nombre de minutes : \n"); scanf("%d",
        &duree);
    // On envoie l'adresse de heures et minutes
    decoupeMinutes(duree, &heures, &minutes);
    // Cette fois, les valeurs ont ete modifiees !
    printf("%d heures et %d minutes", heures, minutes);
    return 0;
}
void decoupeMinutes(int fduree, int* pointeurHeures, int*
    pointeurMinutes)
{
    *pointeurHeures = fduree / 60;
    *pointeurMinutes = fduree % 60;
}
```

En résumé

- Chaque variable est stockée à une adresse précise en mémoire.
- Les pointeurs sont semblables aux variables. Au lieu de stocker un nombre ils stockent l'adresse à laquelle se trouve une variable en mémoire.
- Si on place un symbole **&** devant un nom de variable, on obtient son adresse au lieu de sa valeur (ex. : & age).
- Si on place un symbole * devant un nom de pointeur, on obtient la valeur de la variable stockée à l'adresse indiquée par le pointeur.
- Les pointeurs constituent une notion essentielle du langage C, mais néanmoins un peu complexe au début. Il faut prendre le temps de bien comprendre comment ils fonctionnent car beaucoup d'autres notions sont basées dessus.

En résumé

- Chaque variable est stockée à une adresse précise en mémoire.
- Les pointeurs sont semblables aux variables. Au lieu de stocker un nombre ils stockent l'adresse à laquelle se trouve une variable en mémoire.
- Si on place un symbole **&** devant un nom de variable, on obtient son adresse au lieu de sa valeur (ex. : & age).
- Si on place un symbole * devant un nom de pointeur, on obtient la valeur de la variable stockée à l'adresse indiquée par le pointeur.
- Les pointeurs constituent une notion essentielle du langage C, mais néanmoins un peu complexe au début. Il faut prendre le temps de bien comprendre comment ils fonctionnent car beaucoup d'autres notions sont basées dessus.

En résumé

- Chaque variable est stockée à une adresse précise en mémoire.
- Les pointeurs sont semblables aux variables. Au lieu de stocker un nombre ils stockent l'adresse à laquelle se trouve une variable en mémoire.
- Si on place un symbole **&** devant un nom de variable, on obtient son adresse au lieu de sa valeur (ex. : & age).
- Si on place un symbole * devant un nom de pointeur, on obtient la valeur de la variable stockée à l'adresse indiquée par le pointeur.
- Les pointeurs constituent une notion essentielle du langage C, mais néanmoins un peu complexe au début. Il faut prendre le temps de bien comprendre comment ils fonctionnent car beaucoup d'autres notions sont basées dessus.

En résumé

- Chaque variable est stockée à une adresse précise en mémoire.
- Les pointeurs sont semblables aux variables. Au lieu de stocker un nombre ils stockent l'adresse à laquelle se trouve une variable en mémoire.
- Si on place un symbole **&** devant un nom de variable, on obtient son adresse au lieu de sa valeur (ex. : & age).
- Si on place un symbole ***** devant un nom de pointeur, on obtient la valeur de la variable stockée à l'adresse indiquée par le pointeur.
- Les pointeurs constituent une notion essentielle du langage C, mais néanmoins un peu complexe au début. Il faut prendre le temps de bien comprendre comment ils fonctionnent car beaucoup d'autres notions sont basées dessus.

En résumé

- Chaque variable est stockée à une adresse précise en mémoire.
- Les pointeurs sont semblables aux variables. Au lieu de stocker un nombre ils stockent l'adresse à laquelle se trouve une variable en mémoire.
- Si on place un symbole **&** devant un nom de variable, on obtient son adresse au lieu de sa valeur (ex. : & age).
- Si on place un symbole ***** devant un nom de pointeur, on obtient la valeur de la variable stockée à l'adresse indiquée par le pointeur.
- Les pointeurs constituent une notion essentielle du langage C, mais néanmoins un peu complexe au début. Il faut prendre le temps de bien comprendre comment ils fonctionnent car beaucoup d'autres notions sont basées dessus.

Initiation à l'algorithmique

Les structures

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Introduction

Exercice :

On souhaite écrire un programme qui permet de gérer un ensemble d'étudiants. Chaque étudiant est décrit par son nom, son age et sa moyenne, écrire les fonctions suivantes :

1. Saisir les informations concernant un étudiant.
2. Afficher les informations concernant un étudiant.
3. Saisir un tableau d'étudiants.
4. afficher un tableau d'étudiants.
5. calculer la moyenne générale des étudiants.
6. Ajouter un attribut pour le numéro d'étudiant, et un autre pour son état (nouveau, répétitif ou endetté) et pour plus de précision, remplacer l'âge par la date de naissance.
7. Enfin séparer les étudiants en trois tableaux ; tab_nouveaux, tab_repetitifs et tab_endettes.

Les structures (enregistrements)

- Une structure (ou enregistrement) est une structure de données consistant en un nombre fixé de composants, appelés champs.
- Á la différence du tableau, ces composants ne sont pas obligatoirement du même type, et ne sont pas indexés.
- La définition du type enregistrement précise pour chaque composant un identificateur de champ, dont la portée est limitée à l'enregistrement, et le type de ce champ .

```
struct Nom_Structure
{
    int champ1;
    char champ2;
    double champ3;
    char[20] champ4;
};
```

☞Attention

Le point-virgule après l'accolade fermante est obligatoire.

Les structures (enregistrements)

- Une structure (ou enregistrement) est une structure de données consistant en un nombre fixé de composants, appelés champs.
- Á la différence du tableau, ces composants ne sont pas obligatoirement du même type, et ne sont pas indexés.
- La définition du type enregistrement précise pour chaque composant un identificateur de champ, dont la portée est limitée à l'enregistrement, et le type de ce champ .

```
struct Nom_Structure
{
    int champ1;
    char champ2;
    double champ3;
    char[20] champ4;
};
```

☞Attention

Le point-virgule après l'accolade fermante est obligatoire.

Les structures (enregistrements)

- Une structure (ou enregistrement) est une structure de données consistant en un nombre fixé de composants, appelés champs.
- Á la différence du tableau, ces composants ne sont pas obligatoirement du même type, et ne sont pas indexés.
- La définition du type enregistrement précise pour chaque composant un identificateur de champ, dont la portée est limitée à l'enregistrement, et le type de ce champ .

```
struct Nom_Structure
{
     int champ1;
     char champ2;
     double champ3;
     char[20] champ4;
};
```

☞Attention

Le point-virgule après l'accolade fermante est obligatoire.

Les structures (enregistrements)

- Une structure (ou enregistrement) est une structure de données consistant en un nombre fixé de composants, appelés champs.
- Á la différence du tableau, ces composants ne sont pas obligatoirement du même type, et ne sont pas indexés.
- La définition du type enregistrement précise pour chaque composant un identificateur de champ, dont la portée est limitée à l'enregistrement, et le type de ce champ .

```
struct Nom_Structure
{
     int champ1;
     char champ2;
     double champ3;
     char[20] champ4;
};
```

☞Attention

Le point-virgule après l'accolade fermante est obligatoire.

Les types structurés (enregistrements)

- Une fois qu'on a défini un type structuré, on peut déclarer des variables enregistrements exactement de la même façon que l'on déclare des variables d'un type primitif.

```
struct Etudiant {
    char nom[30];
    int age;
    float moyenne;
};

void main(){
    struct Etudiant e1, e2;
}
```

☞Faut-il obligatoirement écrire le mot-clé **struct** lors de la définition de la variable ?

Les types structurés (enregistrements)

- Une fois qu'on a défini un type structuré, on peut déclarer des variables enregistrements exactement de la même façon que l'on déclare des variables d'un type primitif.

```
struct Etudiant {
    char nom[30];
    int age;
    float moyenne;
};

void main(){
    struct Etudiant e1, e2;
}
```

☞Faut-il obligatoirement écrire le mot-clé **struct** lors de la définition de la variable ?

Le typedef

- L'instruction appelée **typedef** permet de créer de nouveaux noms de types (autrement dit, elle sert à créer un alias de structure) :

```
typedef float reel ;
typedef struct Etudiant Etudiant ;

struct Etudiant {
    char nom[30];
    int age;
    reel moyenne;
};

void main(){
        reel r;
    Etudiant e1, e2;
}
```

- **typedef** : permet de créer un alias de structure ;
- **struct Etudiant** : c'est le nom de la structure pour laquelle on veut créer un alias (c'est-à-dire un « équivalent ») ;
- **Etudiant** : c'est le nom de l'équivalent.

Le typedef

- L'instruction appelée **typedef** permet de créer de nouveaux noms de types (autrement dit, elle sert à créer un alias de structure) :

```
typedef float reel ;
typedef struct Etudiant Etudiant ;

struct Etudiant {
    char nom[30];
    int age;
    reel moyenne;
};

void main(){
        reel r;
    Etudiant e1, e2;
}
```

- **typedef** : permet de créer un alias de structure ;
- **struct Etudiant** : c'est le nom de la structure pour laquelle on veut créer un alias (c'est-à-dire un « équivalent ») ;
- **Etudiant** : c'est le nom de l'équivalent.

Le typedef

- L'instruction appelée **typedef** permet de créer de nouveaux noms de types (autrement dit, elle sert à créer un alias de structure) :

```
typedef float reel ;
typedef struct Etudiant Etudiant ;

struct Etudiant {
    char nom[30];
    int age;
    reel moyenne;
};

void main(){
        reel r;
    Etudiant e1, e2;
}
```

- **typedef** : permet de créer un alias de structure ;
- **struct Etudiant** : c'est le nom de la structure pour laquelle on veut créer un alias (c'est-à-dire un « équivalent ») ;
- **Etudiant** : c'est le nom de l'équivalent.

Le typedef

- L'instruction appelée **typedef** permet de créer de nouveaux noms de types (autrement dit, elle sert à créer un alias de structure) :

```
typedef float reel ;
typedef struct Etudiant Etudiant ;

struct Etudiant {
    char nom[30];
    int age;
    reel moyenne;
};

void main(){
        reel r;
    Etudiant e1, e2;
}
```

- **typedef** : permet de créer un alias de structure ;
- **struct Etudiant** : c'est le nom de la structure pour laquelle on veut créer un alias (c'est-à-dire un « équivalent ») ;
- **Etudiant** : c'est le nom de l'équivalent.

Manipulation des champs d'une structure

- La manipulation d'une structure se fait au travers de ses champs.
- Les champs d'une structure sont accessibles à travers leur nom, grâce à l'opérateur '.'

```
void main()
{
    Etudiant e1, e2;

    puts("Donnez votre nom : ");
    scanf("%s", &e1.nom);
    puts("Donnez votre age : ");
    scanf("%d", &e1.age);
    puts("Donnez votre moyenne : ");
    scanf("%f", &e1.moyenne);
    e2=e1;
}
```

- Comme pour les tableaux, il n'est pas possible de manipuler une structure globalement, sauf pour affecter une structure à un autre de même type (Par exemple : e2=e1 ;).
- Par exemple, pour afficher une structure il faut afficher tous ses champs un par un.

Manipulation des champs d'une structure

- La manipulation d'une structure se fait au travers de ses champs.
- Les champs d'une structure sont accessibles à travers leur nom, grâce à l'opérateur '.'

```
void main()
{
    Etudiant e1, e2;

    puts("Donnez votre nom : ");
    scanf("%s", &e1.nom);
    puts("Donnez votre age : ");
    scanf("%d", &e1.age);
    puts("Donnez votre moyenne : ");
    scanf("%f", &e1.moyenne);
    e2=e1;
}
```

- Comme pour les tableaux, il n'est pas possible de manipuler une structure globalement, sauf pour affecter une structure à un autre de même type (Par exemple : e2=e1 ;).
- Par exemple, pour afficher une structure il faut afficher tous ses champs un par un.

Manipulation des champs d'une structure

- La manipulation d'une structure se fait au travers de ses champs.
- Les champs d'une structure sont accessibles à travers leur nom, grâce à l'opérateur '.'

```
void main()
{
    Etudiant e1, e2;

    puts("Donnez votre nom : ");
    scanf("%s", &e1.nom);
    puts("Donnez votre age : ");
    scanf("%d", &e1.age);
    puts("Donnez votre moyenne : ");
    scanf("%f", &e1.moyenne);
    e2=e1;
}
```

- Comme pour les tableaux, il n'est pas possible de manipuler une structure globalement, sauf pour affecter une structure à un autre de même type (Par exemple : e2=e1 ;).
- Par exemple, pour afficher une structure il faut afficher tous ses champs un par un.

Initialiser une structure

- l'initialisation d'une structure ressemble un peu ressembler à celle d'un tableau.

- En effet, on peut initialiser une structure au moment de sa déclaration :

```
void main(){
    Etudiant e = {"Toto", 19, 12.65}  ;
}
```

- Sinon, on pour créer une fonction **initialiserEtudiant** qui se charge de faire l'initialisation d'une variable de type Etudiant.

- Mais pour pouvoir faire cela il faut envoyer un pointeur de la variable structure à la fonction initialiserEtudiant.

Initialiser une structure

- l'initialisation d'une structure ressemble un peu ressembler à celle d'un tableau.
- En effet, on peut initialiser une structure au moment de sa déclaration :

```
void main(){
    Etudiant e = {"Toto", 19, 12.65}  ;
}
```

- Sinon, on pour créer une fonction **initialiserEtudiant** qui se charge de faire l'initialisation d'une variable de type Etudiant.
- Mais pour pouvoir faire cela il faut envoyer un pointeur de la variable structure à la fonction initialiserEtudiant.

Initialiser une structure

- l'initialisation d'une structure ressemble un peu ressembler à celle d'un tableau.

- En effet, on peut initialiser une structure au moment de sa déclaration :

```
void main(){
    Etudiant e = {"Toto", 19, 12.65}   ;
}
```

- Sinon, on pour créer une fonction **initialiserEtudiant** qui se charge de faire l'initialisation d'une variable de type Etudiant.

- Mais pour pouvoir faire cela il faut envoyer un pointeur de la variable structure à la fonction initialiserEtudiant.

Pointeur de structure

- Un pointeur de structure se crée de la même manière qu'un pointeur de int, de double ou de n'importe quelle autre type de base

- En effet, on peut faire à la déclaration de la variable :

```
void initialiserEtudiant(Etudiant *e){
    (*e).nom = "";
    (*e).age = 0;
    (*e).moyenne = 0;
}
void main(){
    Etudiant e1, *e2=NULL;
    initialiserEtudiant(&e1);
    initialiserEtudiant(e2);
}
```

- Il faut placer des parenthèses autour de *e, car *e.age et (*e).age sont deux écritures différentes.

Pointeur de structure

- Un pointeur de structure se crée de la même manière qu'un pointeur de int, de double ou de n'importe quelle autre type de base

- En effet, on peut faire à la déclaration de la variable :

```
void initialiserEtudiant(Etudiant *e){
    (*e).nom = "";
    (*e).age = 0;
    (*e).moyenne = 0;
}
void main(){
    Etudiant e1, *e2=NULL;
    initialiserEtudiant(&e1);
    initialiserEtudiant(e2);
}
```

- Il faut placer des parenthèses autour de *e, car *e.age et (*e).age sont deux écritures différentes.

Pointeur de structure : une autre notation

- Vu qu'en programmation on manipulera très souvent des pointeurs de structures, le langage C nous offre un raccourci très pratique et très utilisé.
- Ce raccourci consiste à former une flèche avec un tiret suivi d'un chevron > (par exemple (*e).age devient e->age).

```
void initialiserEtudiant(Etudiant *e){
    strcpy((e->nom,"");
    e->age = 0;
    e->moyenne = 0;
}
void main(){
    Etudiant e1, *e2=&e1;
    e1.moyenne = 10; // une variable: on utilise le "point"
    e2->moyenne = 12; //un pointeur: on utilise la fleche
}
```

- Il ne faut surtout pas confondre la flèche avec le « point ».
- En effet, la flèche est réservée aux pointeurs, le « point » est réservé aux variables.

Pointeur de structure : une autre notation

- Vu qu'en programmation on manipulera très souvent des pointeurs de structures, le langage C nous offre un raccourci très pratique et très utilisé.
- Ce raccourci consiste à former une flèche avec un tiret suivi d'un chevron > (par exemple (*e).age devient e->age).

```
void initialiserEtudiant(Etudiant *e){
    strcpy((e->nom,"");
    e->age = 0;
    e->moyenne = 0;
}
void main(){
    Etudiant e1, *e2=&e1;
    e1.moyenne = 10; // une variable: on utilise le "point"
    e2->moyenne = 12; //un pointeur: on utilise la fleche
}
```

- Il ne faut surtout pas confondre la flèche avec le « point ».
- En effet, la flèche est réservée aux pointeurs, le « point » est réservé aux variables.

Pointeur de structure : une autre notation

- Vu qu'en programmation on manipulera très souvent des pointeurs de structures, le langage C nous offre un raccourci très pratique et très utilisé.
- Ce raccourci consiste à former une flèche avec un tiret suivi d'un chevron > (par exemple (*e).age devient e->age).

```
void initialiserEtudiant(Etudiant *e){
    strcpy((e->nom,"");
    e->age = 0;
    e->moyenne = 0;
}
void main(){
    Etudiant e1, *e2=&e1;
    e1.moyenne = 10; // une variable: on utilise le "point"
    e2->moyenne = 12; //un pointeur: on utilise la fleche
}
```

- Il ne faut surtout pas confondre la flèche avec le « point ».
- En effet, la flèche est réservée aux pointeurs, le « point » est réservé aux variables.

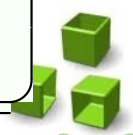
Les structures imbriquées

- Un champ dans une structure peut lui-même être **structure**.

```
typedef struct Date Date ;
struct Date {
    int jour;
    int mois;
    int annee;
};

typedef struct Etudiant Etudiant ;
struct Etudiant {
    char nom[30];
    int age;
    float moyenne;
    Date date_naissance ;
};
```

Retour à l'exercice : Saisir une variable de type Etudiant

```
void saisie_Date(Date *d){
    puts("Date de naissance");
    puts("Donnez le jour : ");
    scanf("%d", &d->jour);
    puts("Donnez le mois : ");
    scanf("%d", &d->mois);
    puts("Donnez l'annee : ");
    scanf("%d", &d->annee);

}
void saisie(Etudiant *e){
    puts("Donnez le nom : ");
    scanf("%s", &e->nom);
    puts("Donnez l'age : ");
    scanf("%d", &e->age);
    puts("Donnez la moyenne : ");
    scanf("%f", &e->moyenne);

    saisie_Date(&e->date_naissance);
}
```



Retour à l'exercice : Afficher une variable de type Etudiant

```
void dateToString(Date d, char s[12]){
   sprintf(s, "%d/%d/%d", d.jour, d.mois, d.annee);
}

void afficher(Etudiant e){
    char s[12];
    dateToString(e.date_naissance, s);
    printf("[ Nom : %s, Age : %d , Moyenne : %2.2f, Date de
        naissance : %s, Etat : %s ] \n", e.nom, e.age, e.
        moyenne, s , etat_ToString(e));

}
```

Retour à l'exercice : Afficher une variable de type Etudiant (une autre version)

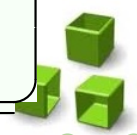
```
char* date_ToString(Date d){
    char *s;
    s = malloc (sizeof (*s) * 20);
    sprintf(s, "%d/%d/%d", d.jour, d.mois, d.annee);
    return s;
}

void afficher(Etudiant e){
    printf("[ Nom : %s, Age : %d , Moyenne : %2.2f, Date de
        naissance : %s ] \n", e.nom, e.age, e.moyenne,
        date_ToString(e.date_naissance));

}
```

Retour à l'exercice : Saisir et afficher plusieurs étudiants

```
void saisie_tab(Etudiant tab[100], int N){
    int i=0;
    for (i=0; i<N; i++)
    {
        printf("--- Etudiant N: %d ---\n", i+1);
        saisie(&tab[i]);
    }
}

void afficher_tab(Etudiant tab[100], int N){
    int i=0;
    for (i=0; i<N; i++)
    {
        afficher(tab[i]);
    }
}
```

Retour à l'exercice : Calculer la moyenne de plusieurs étudiants

```
float moyenne_Etudiants(Etudiant tab[], int N)
{
    int i=0;
    float somme=0;
    float moyenne = 0;
    for (i=0; i<N; i++)
    {
        somme = somme + tab[i].moyenne;
    }
    moyenne = somme/N;
    return moyenne;
}

void main(){
    Etudiant tab_Etudiants[100];
    saisie_tab(tab_Etudiants, 3);
    afficher_tab(tab_Etudiants, 3);
    printf("la moyenne des etudiants est : %lf",
        moyenne_Etudiants(tab_Etudiants, 3));
}
```



Types énumérés

- Au lieu de donner des valeurs conventionnelles à des données symboliques, on peut utiliser les types énumérés.

```
typedef enum Jours Jour;
typedef enum Feux Feux;
typedef enum Etat_Etudiant Etat_Etudiant;

enum Jours {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI,
    VENDREDI, SAMEDI};
enum Feux {ROUGE, ORANGE, VERT};
enum Etat_Etudiant {Nouveau, Repetitif, Endette};
```

- Un type énuméré est un type dont les variables associées n'auront qu'un nombre très limité de valeurs (au maximum 256 différentes possibles).
- La définition d'un type énuméré consiste à déclarer une liste de valeurs possibles associées à un type.

Types énumérés

- Au lieu de donner des valeurs conventionnelles à des données symboliques, on peut utiliser les types énumérés.

```
typedef enum Jours Jour;
typedef enum Feux Feux;
typedef enum Etat_Etudiant Etat_Etudiant;

enum Jours {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI,
    VENDREDI, SAMEDI};
enum Feux {ROUGE, ORANGE, VERT};
enum Etat_Etudiant {Nouveau, Repetitif, Endette};
```

- Un type énuméré est un type dont les variables associées n'auront qu'un nombre très limité de valeurs (au maximum 256 différentes possibles).
- La définition d'un type énuméré consiste à déclarer une liste de valeurs possibles associées à un type.

Types énumérés

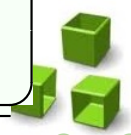
- Au lieu de donner des valeurs conventionnelles à des données symboliques, on peut utiliser les types énumérés.

```
typedef enum Jours Jour;
typedef enum Feux Feux;
typedef enum Etat_Etudiant Etat_Etudiant;

enum Jours {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI,
    VENDREDI, SAMEDI};
enum Feux {ROUGE, ORANGE, VERT};
enum Etat_Etudiant {Nouveau, Repetitif, Endette};
```

- Un type énuméré est un type dont les variables associées n'auront qu'un nombre très limité de valeurs (au maximum 256 différentes possibles).
- La définition d'un type énuméré consiste à déclarer une liste de valeurs possibles associées à un type.

Types énumérés : exemple d'utilisation

```
typedef enum Etat_Etudiant Etat_Etudiant;
enum Etat_Etudiant  {NOUVEAU, REPETITIF, ENDETTE};

typedef struct Etudiant Etudiant ;
struct Etudiant {
    char nom[30];
    int age;
    float moyenne;
    Date date_naissance ;
    Etat_Etudiant etat ;
};
void separer_tabs(Etudiant tab[100], int N){
    int i=0, e=0, n=0, r=0;
    for (i=0; i<N; i++)
    {
        switch(tab[i].etat){
            case 0 : tab_nouveaux[n] = tab[i]; n++; break;
            case 1 : tab_repetitifs[r] = tab[i]; r++; break;
            case 2 : tab_endettes[e] = tab[i]; e++; break;
        }
    }
}
```



Résumé

- Une structure est un type de variable personnalisé qu'on peut créer et utiliser dans des programmes. C'est au programmeur de le définir, contrairement aux types de base tels que **int** et **double**,...
- Une structure est composée de « sous-variables » qui sont en général des variables de type de base comme int et double, mais aussi des tableaux ou d'autres structures.
- On accède à un des composants de la structure en séparant le nom de la variable et la composante d'un point : e.nom
- Si on manipule un pointeur de structure et qu'on veut accéder à une des composantes, on utilise une flèche à la place du point : ptrE->nom.
- Une énumération est un type de variable personnalisé qui peut seulement prendre une des valeurs prédéfinies : rouge, orange ou vert par exemple.

Résumé

- Une structure est un type de variable personnalisé qu'on peut créer et utiliser dans des programmes. C'est au programmeur de le définir, contrairement aux types de base tels que **int** et **double**,...
- Une structure est composée de « sous-variables » qui sont en général des variables de type de base comme int et double, mais aussi des tableaux ou d'autres structures.
- On accède à un des composants de la structure en séparant le nom de la variable et la composante d'un point : e.nom
- Si on manipule un pointeur de structure et qu'on veut accéder à une des composantes, on utilise une flèche à la place du point : ptrE->nom.
- Une énumération est un type de variable personnalisé qui peut seulement prendre une des valeurs prédéfinies : rouge, orange ou vert par exemple.

Résumé

- Une structure est un type de variable personnalisé qu'on peut créer et utiliser dans des programmes. C'est au programmeur de le définir, contrairement aux types de base tels que **int** et **double**,...
- Une structure est composée de « sous-variables » qui sont en général des variables de type de base comme int et double, mais aussi des tableaux ou d'autres structures.
- On accède à un des composants de la structure en séparant le nom de la variable et la composante d'un point : e.nom
- Si on manipule un pointeur de structure et qu'on veut accéder à une des composantes, on utilise une flèche à la place du point : ptrE->nom.
- Une énumération est un type de variable personnalisé qui peut seulement prendre une des valeurs prédéfinies : rouge, orange ou vert par exemple.

Résumé

- Une structure est un type de variable personnalisé qu'on peut créer et utiliser dans des programmes. C'est au programmeur de le définir, contrairement aux types de base tels que **int** et **double**,...
- Une structure est composée de « sous-variables » qui sont en général des variables de type de base comme int et double, mais aussi des tableaux ou d'autres structures.
- On accède à un des composants de la structure en séparant le nom de la variable et la composante d'un point : e.nom
- Si on manipule un pointeur de structure et qu'on veut accéder à une des composantes, on utilise une flèche à la place du point : ptrE->nom.
- Une énumération est un type de variable personnalisé qui peut seulement prendre une des valeurs prédéfinies : rouge, orange ou vert par exemple.

Résumé

- Une structure est un type de variable personnalisé qu'on peut créer et utiliser dans des programmes. C'est au programmeur de le définir, contrairement aux types de base tels que **int** et **double**,...
- Une structure est composée de « sous-variables » qui sont en général des variables de type de base comme int et double, mais aussi des tableaux ou d'autres structures.
- On accède à un des composants de la structure en séparant le nom de la variable et la composante d'un point : e.nom
- Si on manipule un pointeur de structure et qu'on veut accéder à une des composantes, on utilise une flèche à la place du point : ptrE->nom.
- Une énumération est un type de variable personnalisé qui peut seulement prendre une des valeurs prédéfinies : rouge, orange ou vert par exemple.

Initiation à l'algorithmique
Les tableaux

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

Introduction 1/3

Exercice 1

Écrire un programme permettant de **stocker** les 10 notes des examens du S1 d'un étudiant en MI puis les afficher avec sa moyenne.

Introduction 1/3

Exercice 1

Écrire un programme permettant de **stocker** les 10 notes des examens du S1 d'un étudiant en MI puis les afficher avec sa moyenne.

Solution

```
void main()
{
int note=0, somme= 0, moyenne=0, i=0;
for (i=0 ; i<10 ; i++)
{       printf ("donnez la note numero %d : ", i+1) ;
        scanf ("%d", &note) ;
        somme+=M;
}
moyenne = somme/10;
printf("La moyenne est : %d", moyenne);
}
```

Introduction 2/3

Exercice 2

Écrire un programme qui permet de stocker les moyennes de tous les 450 étudiants inscrit en MI puis calculer la moyenne générale.

Introduction 2/3

Exercice 2

Écrire un programme qui permet de stocker les moyennes de tous les 450 étudiants inscrit en MI puis calculer la moyenne générale.

Solution :

```
void main()
{
int M=0, somme= 0, moyenne=0, i=0;
for (i=0 ; i<450 ; i++)
{       printf ("donnez la moyenne numero %d : ", i+1) ;
        scanf ("%d", M) ;
        somme+=M;
}
moyenne = somme/450;
printf("La moyenne est : %d", moyenne);
}
```

Introduction 3/3

Exercice 3

Écrire un programme qui permet de stocker les moyennes de tous les 450 étudiants inscrit en MI puis de déterminer combien d'entre elles sont supérieures à la moyenne de la classe.

Introduction 3/3

Exercice 3

Écrire un programme qui permet de stocker les moyennes de tous les 450 étudiants inscrit en MI puis de déterminer combien d'entre elles sont supérieures à la moyenne de la classe.

Solution :

```
void main()
{
int M1, M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12,M13,.....
...
MoyenneGenerale = (M1+M2+M3+M4+M5+M6+M7+M8+M9+M10+M11+M12+
    M13+...)/450 ;
printf("La moyenne est : %d", Moyenne);

Euuuuuuuuuuuuhhhhh !!!!
...
}
```

Introduction 3/3

Exercice 3

Écrire un programme qui permet de stocker les moyennes de tous les 450 étudiants inscrit en MI puis de déterminer combien d'entre elles sont supérieures à la moyenne de la classe.

Solution :

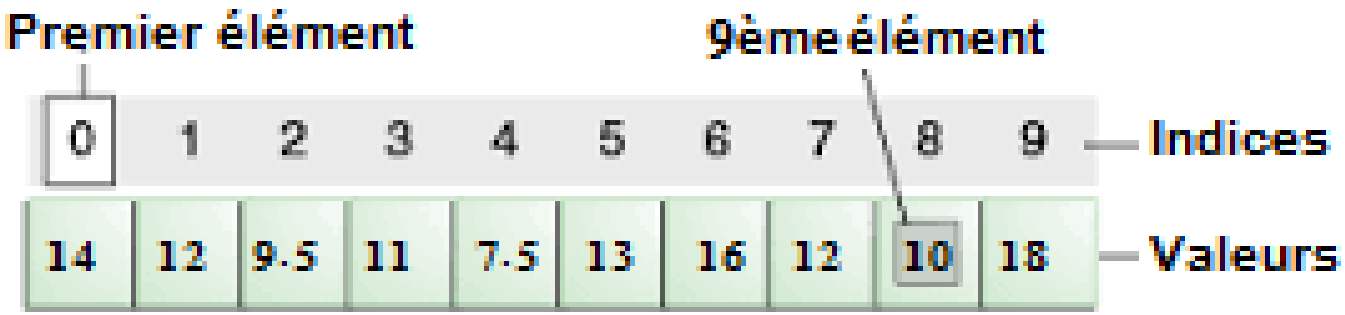
```
void main()
{
int M1, M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12,M13,.....
...
MoyenneGenerale = (M1+M2+M3+M4+M5+M6+M7+M8+M9+M10+M11+M12+
    M13+...)/450 ;
printf("La moyenne est : %d", Moyenne);

Euuuuuuuuuuuuhhhhh !!!!
...
}
```

N'y a t-il pas un moyen plus simple et plus élégant pour écrire ça ?

Bien sûr que si : notion de tableaux

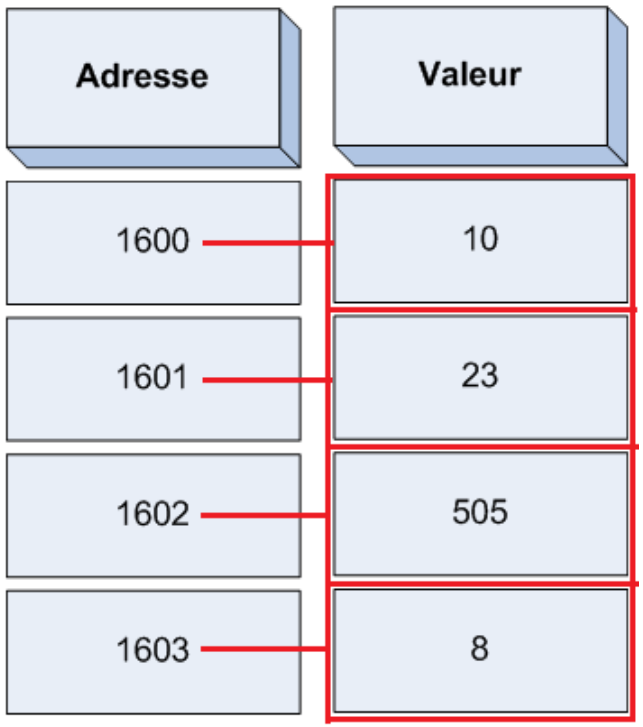
Définition

- Un ensemble de valeurs portant le même nom de variable et repérées par un nombre, s'appelle un tableau, ou encore une variable indicée
- Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle l'indice
- Chaque élément du tableau est désigné le nom du tableau, suivi de l'indice de l'élément, entre parenthèses



Le tableau en mémoire

- Un schéma d'illustration d'un tableau de 4 cases en mémoire qui commence à l'adresse 1600.
- Lorsqu'un tableau est créé, il prend un espace contigu en mémoire : les cases sont les unes à la suite des autres.
- Toutes les cases d'un tableau sont du même type. Ainsi, un tableau de int contiendra uniquement des int, et pas autre chose.



Déclarer et initialiser un tableau

Exemple

```
int notes[10];
...
notes[0] = 10;
notes[1] = 8;
notes[2] = 12;
notes[3] = 17;
...
```

1. Il suffit donc de rajouter entre crochets le nombre de cases que vous voulez mettre dans votre tableau. pas de limite.
2. Pour accéder à chaque élément du tableau, il faut écrire le nom du tableau suivi de l'indice de l'élément concerné entre crochets.

Attention

Un tableau commence à l'indice numéro 0 ! Notre tableau **notes** de 10 **int** a donc les indices 0, 1, 2,... et 9. Il n'y a pas d'indice 10 dans un tableau de 10 cases ! C'est une source d'erreurs très courantes pour les débutants.

Déclarer et initialiser un tableau

Exemple

```
int notes[10];
...
notes[0] = 10;
notes[1] = 8;
notes[2] = 12;
notes[3] = 17;
...
```

1. Il suffit donc de rajouter entre crochets le nombre de cases que vous voulez mettre dans votre tableau. pas de limite.
2. Pour accéder à chaque élément du tableau, il faut écrire le nom du tableau suivi de l'indice de l'élément concerné entre crochets.

Attention

Un tableau commence à l'indice numéro 0 ! Notre tableau **notes** de 10 **int** a donc les indices 0, 1, 2,... et 9. Il n'y a pas d'indice 10 dans un tableau de 10 cases ! C'est une source d'erreurs très courantes pour les débutants.

Le nom d'un tableau est un pointeur

Exemple :

```
int notes[10];
printf("%d", notes);
printf("%d", notes[0]);
printf("%d", *notes);
```

- Au début, on affiche l'adresse où se trouve notes : 1600
- En revanche, si on indique l'indice d'une case du tableau notes entre crochets, on obtient sa valeur : 10. De même pour les autres indices.
- Le nom du tableau **notes** est un pointeur vers la première case du tableau notes, on peut donc utiliser le symbole * pour connaître la première valeur : *notes : 10
- Il est aussi possible d'obtenir la valeur de la seconde case avec *(notes + 1) (adresse de tableau + 1). notes[1] et *(notes + 1) sont donc équivalents.
- En clair, notes[0] est la valeur qui se trouve à l'adresse notes + 0 (1600). notes[1] est la valeur se trouvant à l'adresse notes + 1 (1601).

Le nom d'un tableau est un pointeur

Exemple :

```
int notes[10];
printf("%d", notes);
printf("%d", notes[0]);
printf("%d", *notes);
```

- Au début, on affiche l'adresse où se trouve notes : 1600
- En revanche, si on indique l'indice d'une case du tableau notes entre crochets, on obtient sa valeur : 10. De même pour les autres indices.
- Le nom du tableau **notes** est un pointeur vers la première case du tableau notes, on peut donc utiliser le symbole * pour connaître la première valeur : *notes : 10
- Il est aussi possible d'obtenir la valeur de la seconde case avec *(notes + 1) (adresse de tableau + 1). notes[1] et *(notes + 1) sont donc équivalents.
- En clair, notes[0] est la valeur qui se trouve à l'adresse notes + 0 (1600). notes[1] est la valeur se trouvant à l'adresse notes + 1 (1601).

Le nom d'un tableau est un pointeur

Exemple :

```
int notes[10];
printf("%d", notes);
printf("%d", notes[0]);
printf("%d", *notes);
```

- Au début, on affiche l'adresse où se trouve notes : 1600
- En revanche, si on indique l'indice d'une case du tableau notes entre crochets, on obtient sa valeur : 10. De même pour les autres indices.
- Le nom du tableau **notes** est un pointeur vers la première case du tableau notes, on peut donc utiliser le symbole * pour connaître la première valeur : *notes : 10
- Il est aussi possible d'obtenir la valeur de la seconde case avec *(notes + 1) (adresse de tableau + 1). notes[1] et *(notes + 1) sont donc équivalents.
- En clair, notes[0] est la valeur qui se trouve à l'adresse notes + 0 (1600). notes[1] est la valeur se trouvant à l'adresse notes + 1 (1601).

Le nom d'un tableau est un pointeur

Exemple :

```
int notes[10];
printf("%d", notes);
printf("%d", notes[0]);
printf("%d", *notes);
```

- Au début, on affiche l'adresse où se trouve notes : 1600
- En revanche, si on indique l'indice d'une case du tableau notes entre crochets, on obtient sa valeur : 10. De même pour les autres indices.
- Le nom du tableau **notes** est un pointeur vers la première case du tableau notes, on peut donc utiliser le symbole * pour connaître la première valeur : *notes : 10
- Il est aussi possible d'obtenir la valeur de la seconde case avec *(notes + 1) (adresse de tableau + 1). notes[1] et *(notes + 1) sont donc équivalents.
- En clair, notes[0] est la valeur qui se trouve à l'adresse notes + 0 (1600). notes[1] est la valeur se trouvant à l'adresse notes + 1 (1601).

Le nom d'un tableau est un pointeur

Exemple :

```
int notes[10];
printf("%d", notes);
printf("%d", notes[0]);
printf("%d", *notes);
```

- Au début, on affiche l'adresse où se trouve notes : 1600
- En revanche, si on indique l'indice d'une case du tableau notes entre crochets, on obtient sa valeur : 10. De même pour les autres indices.
- Le nom du tableau **notes** est un pointeur vers la première case du tableau notes, on peut donc utiliser le symbole ***** pour connaître la première valeur : *notes : 10
- Il est aussi possible d'obtenir la valeur de la seconde case avec *(notes + 1) (adresse de tableau + 1). notes[1] et *(notes + 1) sont donc équivalents.
- En clair, notes[0] est la valeur qui se trouve à l'adresse notes + 0 (1600). notes[1] est la valeur se trouvant à l'adresse notes + 1 (1601).

Les tableaux à taille dynamique !

Exemple :

```
int nombreMatieres = 10;
int notes[nombrMatieres];
```

- Une version récente, appelée le C99, autorise la création de tableaux à taille dynamique, c'est-à-dire de tableaux dont la taille est définie par une variable.
- Or cela n'est pas forcément reconnu par tous les compilateurs, certains planteront sur la seconde ligne. Nous considèrerons donc que faire cela est interdit.
- Mais pour faire cela, nous utiliserons une autre technique (plus sûre et qui marche partout) appelée l'allocation dynamique. Nous verrons cela bien plus loin dans ce cours.

Les tableaux à taille dynamique !

Exemple :

```
int nombreMatieres = 10;
int notes[nombrMatieres];
```

- Une version récente, appelée le C99, autorise la création de tableaux à taille dynamique, c'est-à-dire de tableaux dont la taille est définie par une variable.
- Or cela n'est pas forcément reconnu par tous les compilateurs, certains planteront sur la seconde ligne. Nous considèrerons donc que faire cela est interdit.
- Mais pour faire cela, nous utiliserons une autre technique (plus sûre et qui marche partout) appelée l'allocation dynamique. Nous verrons cela bien plus loin dans ce cours.

Les tableaux à taille dynamique !

Exemple :

```
int nombreMatieres = 10;
int notes[nombrMatieres];
```

- Une version récente, appelée le C99, autorise la création de tableaux à taille dynamique, c'est-à-dire de tableaux dont la taille est définie par une variable.
- Or cela n'est pas forcément reconnu par tous les compilateurs, certains planteront sur la seconde ligne. Nous considèrerons donc que faire cela est interdit.
- Mais pour faire cela, nous utiliserons une autre technique (plus sûre et qui marche partout) appelée l'allocation dynamique. Nous verrons cela bien plus loin dans ce cours.

Parcourir un tableau

- Supposons qu'on veuille maintenant afficher les valeurs de chaque case du tableau.
- On aurait pu faire autant de printf qu'il y a de cases. Mais ce serait répétitif et lourd, et imaginez un peu la taille de notre code si on devait afficher le contenu de chaque case du tableau une à une !
- Le mieux est de se servir d'une boucle qui est très pratique pour parcourir un tableau :

Exemple

```
void main()
{
    double notes[10], i = 0;
    notes[0] = 10;
    notes[1] = 8.5;
    notes[2] = 16;
    ...
    for (i = 0 ; i < 10 ; i++)
        printf("%d\n", notes[i]);
}
```

Parcourir un tableau

- Supposons qu'on veuille maintenant afficher les valeurs de chaque case du tableau.
- On aurait pu faire autant de printf qu'il y a de cases. Mais ce serait répétitif et lourd, et imaginez un peu la taille de notre code si on devait afficher le contenu de chaque case du tableau une à une !
- Le mieux est de se servir d'une boucle qui est très pratique pour parcourir un tableau :

Exemple

```
void main()
{
    double notes[10], i = 0;
    notes[0] = 10;
    notes[1] = 8.5;
    notes[2] = 16;
    ...
    for (i = 0 ; i < 10 ; i++)
        printf("%d\n", notes[i]);
}
```

Parcourir un tableau

- Supposons qu'on veuille maintenant afficher les valeurs de chaque case du tableau.
- On aurait pu faire autant de printf qu'il y a de cases. Mais ce serait répétitif et lourd, et imaginez un peu la taille de notre code si on devait afficher le contenu de chaque case du tableau une à une !
- Le mieux est de se servir d'une boucle qui est très pratique pour parcourir un tableau :

Exemple

```
void main()
{
    double notes[10], i = 0;
    notes[0] = 10;
    notes[1] = 8.5;
    notes[2] = 16;
    ...
    for (i = 0 ; i < 10 ; i++)
        printf("%d\n", notes[i]);
}
```

Parcourir un tableau

```c
void main()
{
    double notes[10], i = 0;
    notes[0] = 10;
    notes[1] = 8.5;
    notes[2] = 16;
    ...
    for (i = 0 ; i < 10 ; i++)
        printf("%d\n", notes[i]);
}
```

- La boucle parcourt le tableau à l'aide d'une variable appelée i (c'est le nom le plus souvent utilisé pour parcourir un tableau !).
- Notez qu'on peut mettre une variable entre crochets pour « parcourir » le tableau, c'est-à-dire accéder ses valeurs. En effet, la variable était interdite uniquement lors de la création du tableau.
- Attention à ne pas tenter d'afficher la valeur de notes[10] !Sinon vous aurez soit n'importe quoi, soit une belle erreur, l'OS coupant votre programme car il aura tenté d'accéder à une adresse ne lui appartenant pas.

Parcourir un tableau

```c
void main()
{
    double notes[10], i = 0;
    notes[0] = 10;
    notes[1] = 8.5;
    notes[2] = 16;
    ...
    for (i = 0 ; i < 10 ; i++)
        printf("%d\n", notes[i]);
}
```

- La boucle parcourt le tableau à l'aide d'une variable appelée i (c'est le nom le plus souvent utilisé pour parcourir un tableau !).
- Notez qu'on peut mettre une variable entre crochets pour « parcourir » le tableau, c'est-à-dire accéder ses valeurs. En effet, la variable était interdite uniquement lors de la création du tableau.
- Attention à ne pas tenter d'afficher la valeur de notes[10] !Sinon vous aurez soit n'importe quoi, soit une belle erreur, l'OS coupant votre programme car il aura tenté d'accéder à une adresse ne lui appartenant pas.

Parcourir un tableau

```c
void main()
{
    double notes[10], i = 0;
    notes[0] = 10;
    notes[1] = 8.5;
    notes[2] = 16;
    ...
    for (i = 0 ; i < 10 ; i++)
        printf("%d\n", notes[i]);
}
```

- La boucle parcourt le tableau à l'aide d'une variable appelée i (c'est le nom le plus souvent utilisé pour parcourir un tableau !).
- Notez qu'on peut mettre une variable entre crochets pour « parcourir » le tableau, c'est-à-dire accéder ses valeurs. En effet, la variable était interdite uniquement lors de la création du tableau.
- Attention à ne pas tenter d'afficher la valeur de notes[10] !Sinon vous aurez soit n'importe quoi, soit une belle erreur, l'OS coupant votre programme car il aura tenté d'accéder à une adresse ne lui appartenant pas.

Initialiser un tableau

- Maintenant que l'on sait parcourir un tableau, nous sommes capables d'initialiser toutes ses valeurs à 0 en faisant une boucle !

Exemple :

```
void main()
{
    int tableau[4], i = 0;

    // Initialisation du tableau
    for (i = 0 ; i < 4 ; i++)
    {
        tableau[i] = 0;
    }

    // Affichage de ses valeurs pour verifier
    for (i = 0 ; i < 4 ; i++)
    {
        printf("%d\n", tableau[i]);
    }
}
```

Initialiser un tableau

- Il faut savoir qu'il existe une autre façon d'initialiser un tableau un peu plus automatisée en C.
- Elle consiste à placer les valeurs une à une entre accolades, séparées par des virgules.

Exemple :

```
void main()
{
    int tableau[4] = {0, 0, 0, 0}, i = 0;

    for (i = 0 ; i < 4 ; i++)
    {
        printf("%d\n", tableau[i]);
    }
}
```

D'autres exemples d'initialisation de tableaux

- on peut également définir les valeurs des premières cases du tableau, toutes celles que vous n'aurez pas renseignées seront automatiquement mises à 0.

Exemple :

```
void main()
{
    int tab1[4] = {0, 0, 0, 0}; // 0, 0, 0, 0
    int tab2[6] = {10, 23};     // 10, 23, 0, 0, 0, 0
    int tab3[4] = {0};          // 0, 0, 0, 0
    int tab4[5] = {1};          // 1, 0, 0, 0, 0, 0,
}
```

Attention

Dans le tableau tab4, on n'initialise pas toutes les cases à 1 : seule la première case sera à 1, toutes les autres seront à 0.

D'autres exemples d'initialisation de tableaux

- on peut également définir les valeurs des premières cases du tableau, toutes celles que vous n'aurez pas renseignées seront automatiquement mises à 0.

Exemple :

```
void main()
{
    int tab1[4] = {0, 0, 0, 0}; // 0, 0, 0, 0
    int tab2[6] = {10, 23};     // 10, 23, 0, 0, 0, 0
    int tab3[4] = {0};          // 0, 0, 0, 0
    int tab4[5] = {1};          // 1, 0, 0, 0, 0, 0,
}
```

Attention

Dans le tableau tab4, on n'initialise pas toutes les cases à 1 : seule la première case sera à 1, toutes les autres seront à 0.

Et si on revenait à notre exercice de départ ?

Solution

```
#include <stdio.h>
main()
{
int i, som, nbm ;
double moy ;
int t[450] ;
for (i=0 ; i<450 ; i++)
{printf ("donnez la note de l'etudiant numero %d : ", i+1) ;
 scanf ("%d", &t[i]) ;
}
for (i=0, som=0 ; i<450 ; i++) som += t[i] ;
moy = som / 450 ;
printf ("\n\n moyenne de la promo : %f\n", moy) ;
for (i=0, nbm=0 ; i<450 ; i++ )
          if (t[i] > moy) nbm++ ;
printf ("%d etudiants  ont plus de cette moyenne", nbm) ;
}
```

Passage de tableaux à une fonction

Exemple :

```
void saisir(int tab[], int tailleTab)
{
    int i;
    for (i = 0 ; i < tailleTab ; i++)
        scanf("%d", &tab[i]);
}
void afficher(int tab[], int tailleTab)
{
    int i;
    for (i = 0 ; i < tailleTab ; i++)
        printf("%d\n", tab[i]);
}
void main()
{
    int tableau[4] = {0};
    saisir(tableau, 4);
    afficher(tableau, 4);
}
```

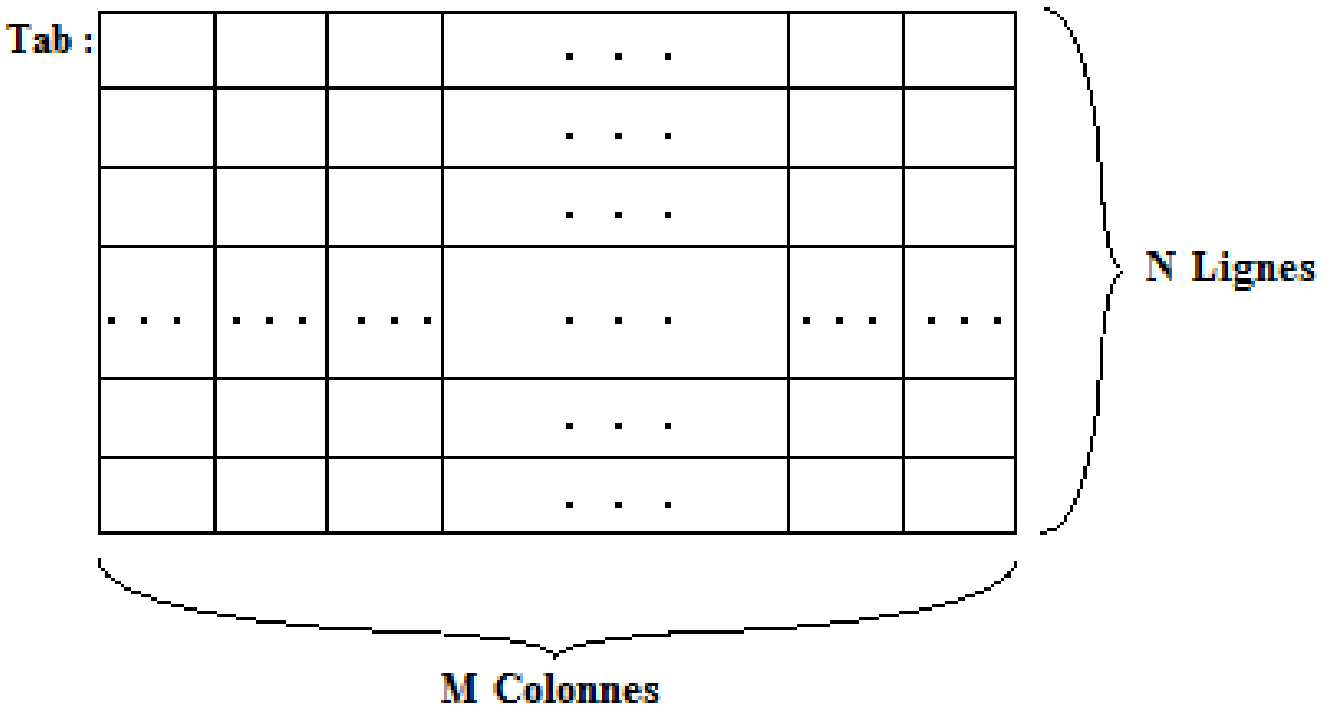
Passage de tableaux à une fonction : pointeur

Exemple :

```
void saisir(int *tableau, int tailleTableau)
{
    int i;
    for (i = 0 ; i < tailleTableau ; i++)
        scanf("%d", &tableau[i]);
}
void afficher(int *tableau, int tailleTableau)
{
    int i;
    for (i = 0 ; i < tailleTableau ; i++)
        printf("%d\n", tableau[i]);
}
void main()
{
    int tab[4] = {0};
    saisir(tab, 4);
    afficher(tab, 4);
}
```

Tableaux à deux dimensions

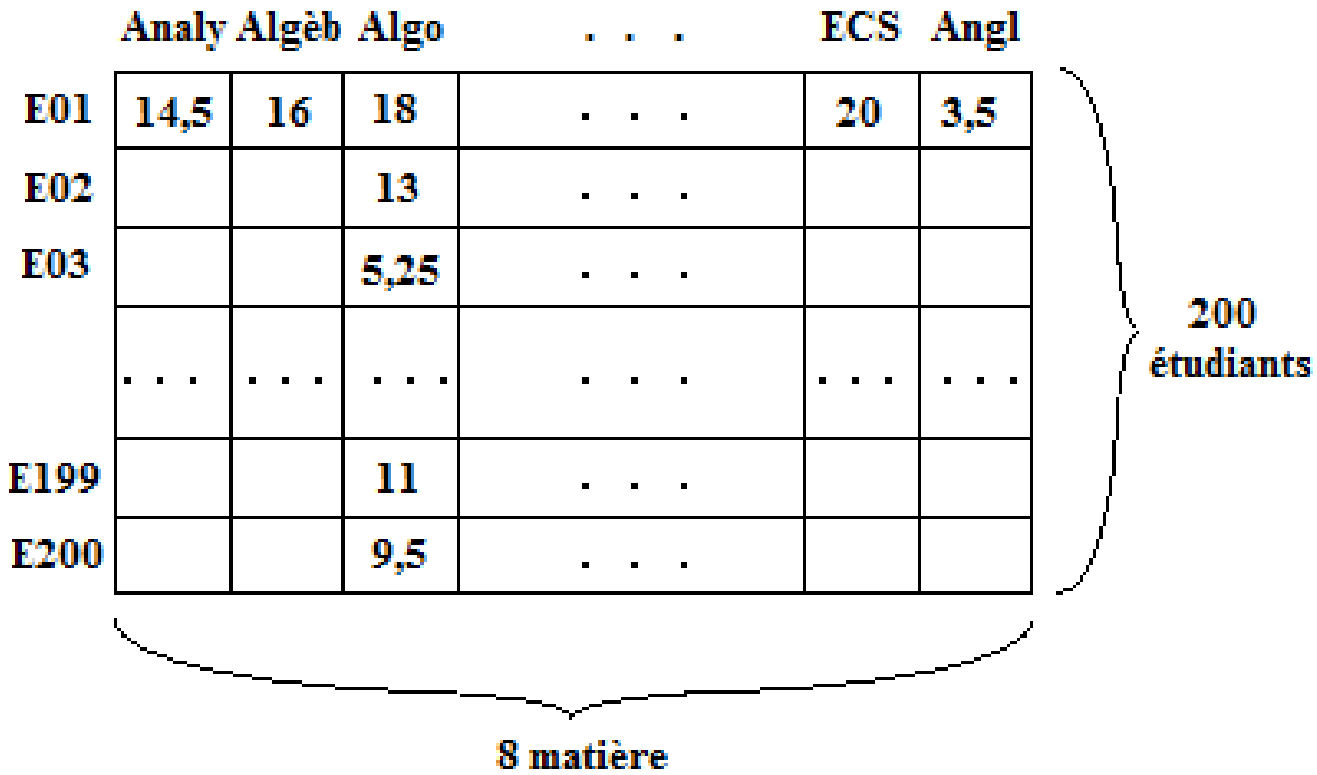
- Un tableau à deux dimensions est à interpréter comme un tableau de dimension N dont chaque élément est un tableau de dimension M.



- On appelle N le nombre de lignes et M le nombre de colonnes du tableau Tab. N et M sont alors les deux dimensions du tableau.
- Un tableau à deux dimensions contient donc N*M éléments.

Tableaux à deux dimensions

- Un tableau à deux dimensions est à interpréter comme un tableau de dimension N dont chaque élément est un tableau de dimension M.



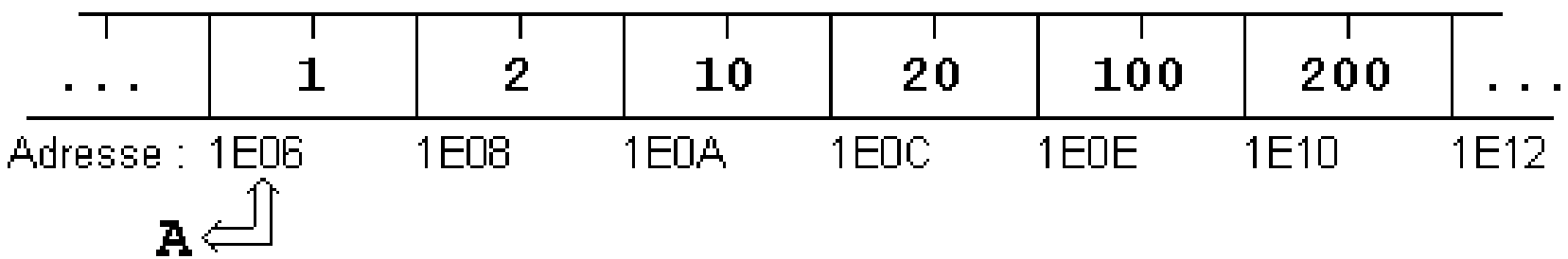
- On appelle N le nombre de lignes et M le nombre de colonnes du tableau Tab. N et M sont alors les deux dimensions du tableau.
- Un tableau à deux dimensions contient donc N*M éléments.

Déclaration et mémorisation des tableaux à deux dimensions

- Comme pour les tableaux à une dimension, le nom d'un tableau est le représentant de l'adresse du premier élément du tableau (c.-à-d. l'adresse de la première ligne du tableau).

```
int A[3][2] =    {{1,    2  },
                  {10,   20 },
                  {100,  200}};
```

- Les éléments d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire.



Initialisation des tableaux à deux dimensions

- Lors de la déclaration d'un tableau, on peut initialiser les éléments du tableau, en indiquant la liste des valeurs respectives entre accolades.
- À l'intérieur de la liste, les éléments de chaque ligne du tableau sont encore une fois comprises entre accolades.
- Pour améliorer la lisibilité des programmes, on peut indiquer les composantes dans plusieurs lignes.

Exemples :

```
int  A[3][10] ={{ 0,10,20,30,40,50,60,70,80,90},
                {10,11,12,13,14,15,16,17,18,19},
                { 1,12,23,34,45,56,67,78,89,90}};

double  B[3][2] = {{-1.05,    -1.10  },
                   {86e-5,     87e-5 },
                   {-12.5E4, -12.3E4}};
```

- Lors de l'initialisation, les valeurs sont affectées ligne par ligne en passant de gauche à droite.
- Nous ne devons pas nécessairement indiquer toutes les valeurs : Les valeurs manquantes seront initialisées par zéro.

Accès aux éléments d'un tableau à deux dimensions

Considérons un tableau A de dimensions N et M.

```
int A[][10] = {{ 0,10,20,30,40,50,60,70,80,90},
               {10,11,12,13,14,15,16,17,18,19},
               { 1,12}};
```

- Les indices du tableau varient de 0 à N-1, respectivement de 0 à M-1.
- L'élément de la $i^{ième}$ ligne et $j^{ième}$ colonne est noté : A[i-1][j-1]

Tableaux à deux dimensions

```c
#include <stdio.h>
#include <stdlib.h>
void afficherTableau(int tableau[2][2]);
int main(void)
{
    int tableau[2][2] = {{10, 20} , {15, 35}};
    afficherTableau(tableau);
        return 0;
}
void afficherTableau(int tableau[2][2])
{
  int i = 0;
  int j = 0;
  for (i = 0; i < 2; i++) {
    for(j = 0; j < 2; j++) {
      printf("Tableau[%d][%d] = %d\n", i, j, tableau[i][j]);
    }
  }
}
```

Initiation à l'algorithmique

Structures répétitives « Les Boucles »

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

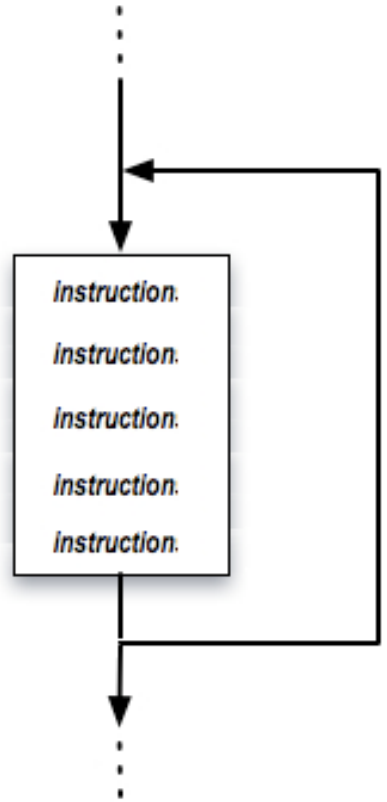
12 novembre 2014

Qu'est qu'une boucle ?

Une boucle est une structure de contrôle qui permet de répéter les mêmes instructions plusieurs fois.

On distingue types de boucles courantes en C :

1. **while**
2. **do... while**
3. **for**



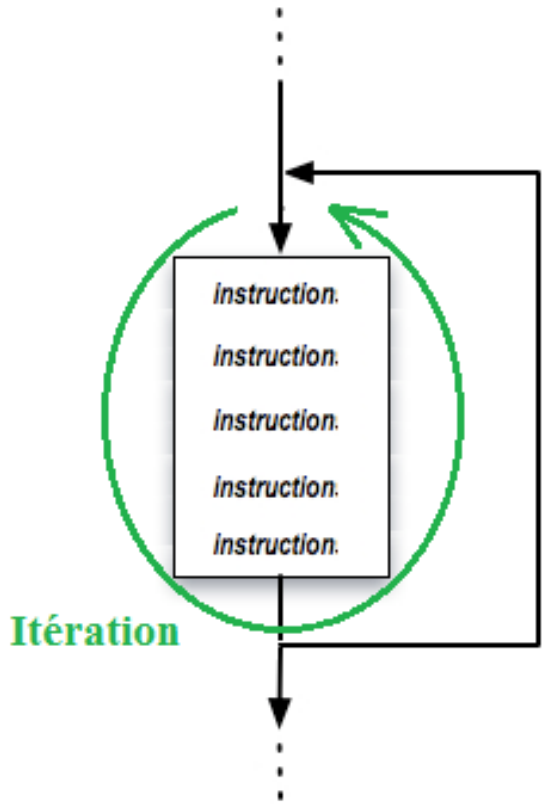
Le passage dans une boucle est appelé **itération**

Qu'est qu'une boucle ?

Une boucle est une structure de contrôle qui permet de répéter les mêmes instructions plusieurs fois.

On distingue types de boucles courantes en C :

1. **while**
2. **do... while**
3. **for**

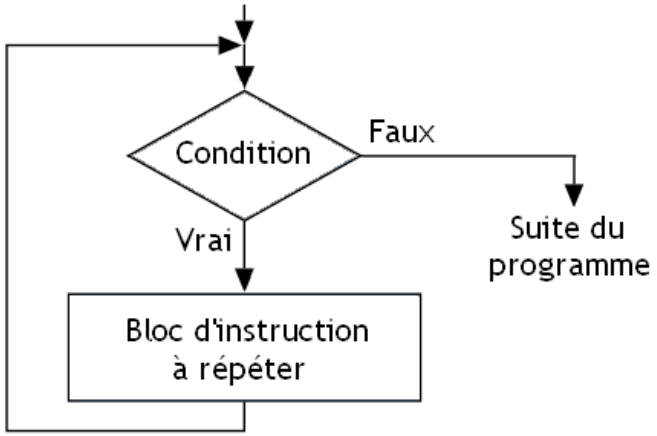


Le passage dans une boucle est appelé **itération**

La boucle « While »

Syntaxe :

```
while ( Condition )
{
    // Bloc d'instructions
}
```

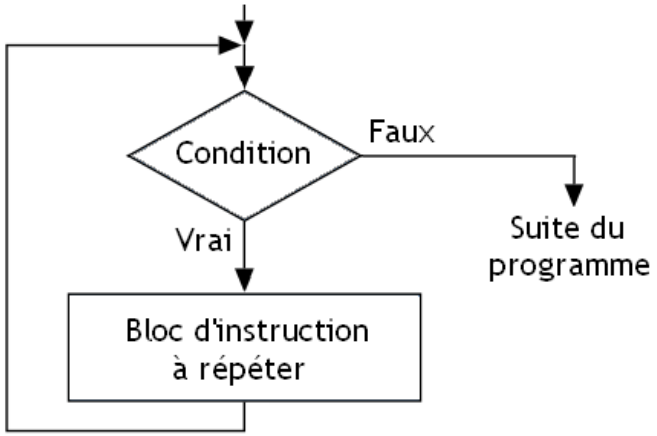


- la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute le bloc d'instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution,...
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui se trouve juste après l'accolade fermante « } ».

La boucle « While »

Syntaxe :

```
while ( Condition )
{
    // Bloc d'instructions
}
```

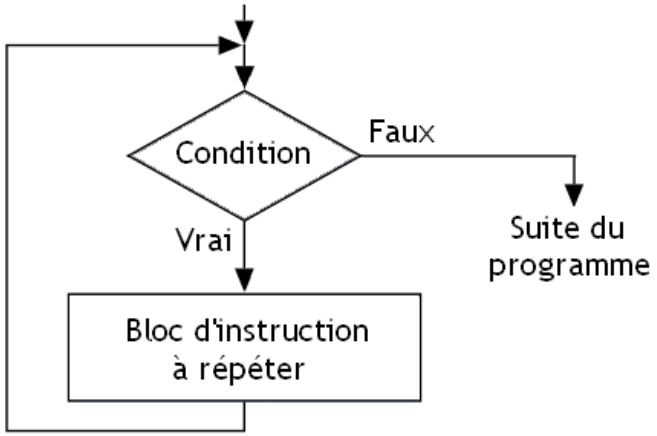


- la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute le bloc d'instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution,...
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui se trouve juste après l'accolade fermante « } ».

La boucle « While »

Syntaxe :

```
while ( Condition )
{
    // Bloc d'instructions
}
```



- la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute le bloc d'instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution,...
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui se trouve juste après l'accolade fermante « } ».

Exemple de boucle « While »

On souhaite écrire un programme qui permet de contrôler la saisie d'un entier positif.

Exemple :

```
int entierPositif = 0;

while (entierPositif <=0)
{
    printf("Tapez un entier positif ! ");
    scanf("%d", &entierPositif);
}
```

Répéter un certain nombre de fois

- On va pour cela créer une variable compteur qui vaudra 0 au début du programme
- Et que l'on va incrémenter à chaque itération.

Exemple :

```
int compteur = 0;

while (compteur < 10)
{
    printf("La variable compteur vaut %d \n", compteur);
    compteur++; // equivalent a compteur = compteur+1;
}
```

Une **incrémentation** consiste à ajouter 1 à la variable en faisant **var++ ;**.

Attention aux boucles infinies

- Le nombre d'itérations dans une boucle **while** n'est pas connu à l'avance. Il dépend de l'évaluation de la condition.
- Lorsque vous créez une boucle, assurez-vous toujours qu'elle peut s'arrêter à un moment ! Si la condition est toujours vraie, votre programme ne s'arrêtera jamais !

Exemple :

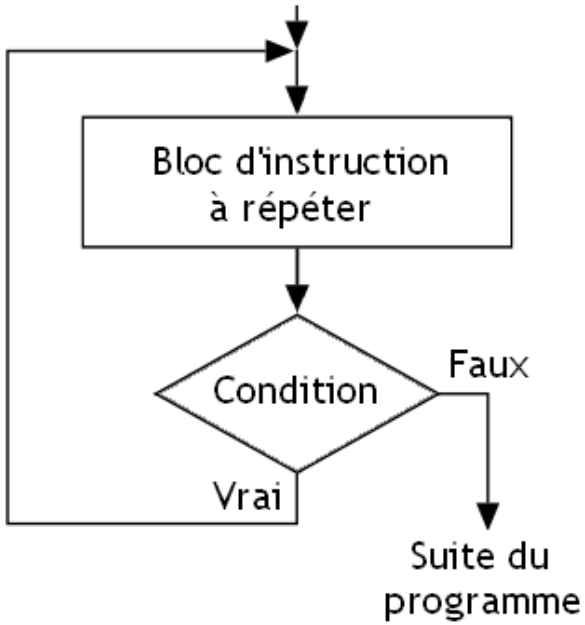
```
int compteur = 0;

while (compteur >= 0)
{
    printf("La variable compteur vaut %d \n", compteur);
    compteur++; // equivalent a compteur = compteur+1;
}
```

- Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment

La boucle « do...while »

Syntaxe :

```
do
{
    // Bloc d'instructions
} while ( Condition );
```



- La boucle **do...while** est très similaire à while
- La seule chose qui change par rapport à while, c'est la position de la condition. Au lieu d'être au début de la boucle, la condition est à la fin.
- Cette boucle s'exécutera donc toujours au moins une fois

Exemple de boucle « do...while »

Exemple :

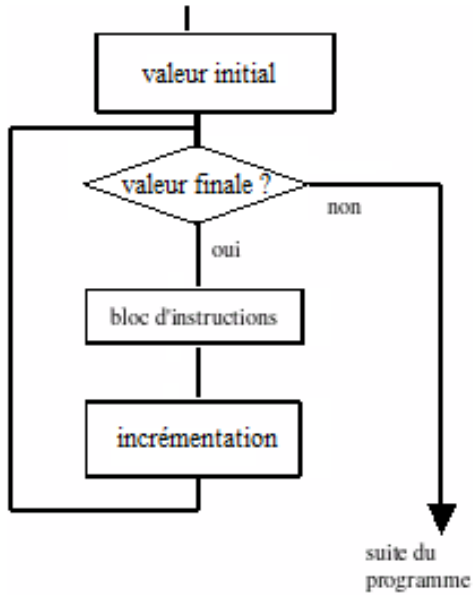
```
int compteur = 0;
int somme = 0;
do
{
    somme =+ compteur // equivalent a somme = somme+compteur
    compteur++;
}while (compteur <= 10);
```

Dans la boucle **« do...while »**, n'oubliez pas de mettre un point-virgule à la fin.

La boucle « For »

Syntaxe :

```
for (initialisation ; condition ; pas )
{
    // Bloc d'instructions
}
```



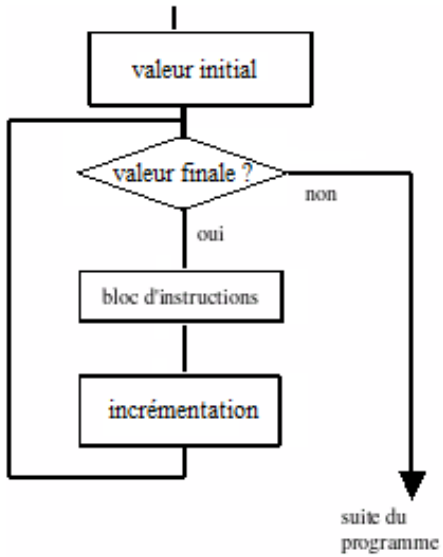
Il y a trois instructions condensées, chacune séparée par un point-virgule.

- La première est l'**initialisation** : cette première instruction est utilisée pour préparer notre variable compteur.
- La seconde est la **condition** : comme pour la boucle while, c'est la condition qui dit si la boucle doit être répétée ou non. Tant que la condition est vraie, la boucle **for** continue.
- Enfin, il y a l'**incrémentation** : cette dernière instruction est exécutée à la fin de chaque tour de boucle pour mettre à jour la variable compteur. par exemple).

Principe de la boucle « For »

Syntaxe :

```
int cpt;
for (cpt=initial; cpt<=finale; cpt=cpt+pas)
{
    printf("cpt vaut %d !\n", cpt);
}
```



1. La valeur **initiale** est affectée à la variable **cpt** ;
2. On compare la valeur du **cpt** et la valeur **finale** :
3. Si la **condition** est **fausse**, on sort de la boucle et on continue avec l'instruction qui suit l'accolade fermante.
4. Si la **condition** est **vrai** alors :
   - 4.1 les instructions de la boucle seront exécutées
   - 4.2 Ensuite, la valeur de **cpt** est **incrémentée** de la valeur du **pas** (sinon 1 par défaut).
   - 4.3 On recommence l'étape 2 : La comparaison entre cpt et finale est de nouveau effectuée, et ainsi de suite...

Exemple de la boucle « for »

Exemple :

```
int compteur;

for (compteur = 0 ; compteur < 10 ; compteur++)
{
    printf("La variable compteur vaut %d !\n", compteur);
}
```

La plus part du temps on fera une **incrémentation**, mais on peut aussi faire une **décrémentation (variable–)** ou encore n'importe quelle autre opération (**variable += 2**; pour avancer de 2 en 2 par exemple).

Attention !

Il est fortement déconseillé de modifier la valeur du compteur (et/ou la valeur de finale) à l'intérieur de la boucle. En effet, une telle action :

- perturbe le nombre d'itérations prévu par la boucle Pour
- présente le risque d'aboutir à une boucle infinie

Exemples :

```c
int compteur;

for (compteur = 0 ; compteur < 10 ; compteur++)
{
    printf("La variable compteur vaut %d !\n", compteur);
    compteur = compteur+2;
}

for (compteur = 9 ; compteur > 0 ; compteur--)
{
    printf("La variable compteur vaut %d !\n", compteur);
    compteur = compteur+2;
}
```

La boucle « For »Vs. La boucle « While »

La boucle Pour est un cas particulier de la boucle While (cas où le nombre d'itérations est connu et fixé) . Tout ce qu'on peut écrire avec For peut être remplacé par une boucle While (la réciproque n'est pas forcément vraie).

Exemple :

```
int compteur;
for (compteur = 0 ; compteur < 10 ; compteur++)
{
    printf("La variable compteur vaut %d !\n", compteur);
}
// est equivalent a :
int compteur = 0;
while (compteur < 10)
{
    printf("La variable compteur vaut %d !\n", compteur);
    compteur++;
}
```

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Les boucles imbriqués

Le bloc d'instructions d'une boucle peut contenir lui même une autre boucle. C'est ce qu'on appelle des boucles imbriquées

Exemple :

```
int i;
int j=1;

for (i = 1 ; i <= 3 ; i++)
{
   j=1
   while (j <= 4)
   {
    printf("i=%d et j=%d!\n", i, j);
    j++;
   }
}
```

Historique d'exécution

Inst.	i	j	Affichage
1	1	1	i=1 et j=1
2	1	2	i=1 et j=2
3	1	3	i=1 et j=3
4	1	4	i=1 et j=4
5	2	1	i=2 et j=1
6	2	2	i=2 et j=2
7	2	3	i=2 et j=3
8	2	4	i=2 et j=4
9	3	1	i=3 et j=1
10	3	2	i=3 et j=2
11	3	3	i=3 et j=3
12	3	4	i=3 et j=4

Quelle boucle puisse-je utiliser pour mon programme ?

Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser :

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez while ou for.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez do - while.
- Si le nombre d'exécutions du bloc d'instructions est connu à l'avance alors utilisez for.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez while.

Le choix entre for et while n'est souvent qu'une question de préférence ou d'habitudes.

Quelle boucle puisse-je utiliser pour mon programme ?

Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser :

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez while ou for.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez do - while.
- Si le nombre d'exécutions du bloc d'instructions est connu à l'avance alors utilisez for.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez while.

Le choix entre for et while n'est souvent qu'une question de préférence ou d'habitudes.

Quelle boucle puisse-je utiliser pour mon programme ?

Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser :

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez while ou for.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez do - while.
- Si le nombre d'exécutions du bloc d'instructions est connu à l'avance alors utilisez for.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez while.

Le choix entre for et while n'est souvent qu'une question de préférence ou d'habitudes.

Quelle boucle puisse-je utiliser pour mon programme ?

Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser :

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez while ou for.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez do - while.
- Si le nombre d'exécutions du bloc d'instructions est connu à l'avance alors utilisez for.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez while.

Le choix entre for et while n'est souvent qu'une question de préférence ou d'habitudes.

Quelle boucle puisse-je utiliser pour mon programme ?

Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser :

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez while ou for.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez do - while.
- Si le nombre d'exécutions du bloc d'instructions est connu à l'avance alors utilisez for.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez while.

Le choix entre for et while n'est souvent qu'une question de préférence ou d'habitudes.

Quelle boucle puisse-je utiliser pour mon programme ?

Utilisez la structure qui reflète le mieux l'idée du programme que vous voulez réaliser :

- Si le bloc d'instructions ne doit pas être exécuté si la condition est fausse, alors utilisez while ou for.
- Si le bloc d'instructions doit être exécuté au moins une fois, alors utilisez do - while.
- Si le nombre d'exécutions du bloc d'instructions est connu à l'avance alors utilisez for.
- Si le bloc d'instructions doit être exécuté aussi longtemps qu'une condition extérieure est vraie alors utilisez while.

Le choix entre for et while n'est souvent qu'une question de préférence ou d'habitudes.

Exemples pour conclure ...

Exemple avec while :

```
int N;          /* nombre de donnees */
int NOMB;       /* nombre courant    */
int I;          /* compteur */
long SOM;       /* la somme   des nombres entres */
double PROD; /* le produit des nombres entres */

printf("Nombre de donnees : ");
scanf("%d", &N);
SOM=0; PROD=1; I=1;
while(I<=N)
    {
     printf("%d. nombre : ", I);
     scanf("%d", &NOMB);
     SOM  += NOMB;
     PROD *= NOMB;
     I++;
    }

printf("La somme   des %d nombres est %ld \n", N, SOM);
printf("Le produit des %d nombres est %.0f\n", N, PROD);
```

Un exemple pour conclure ...

Exemple avec do...while :

```
int N;          /* nombre de donnees */
int NOMB;       /* nombre courant    */
int I;          /* compteur */
long SOM;       /* la somme   des nombres entres */
double PROD; /* le produit des nombres entres */

printf("Nombre de donnees : ");
scanf("%d", &N);

SOM=0;
PROD=1;
I=1;
do
   {
    printf("%d. nombre : ", I);
    scanf("%d", &NOMB);
    SOM  += NOMB;
    PROD *= NOMB;
    I++;
   } while(I<=N);
```

Un exemple pour conclure ...

Exemple avec for :

```
int N;          /* nombre de donnees */
int NOMB;       /* nombre courant    */
int I;          /* compteur */
long SOM;       /* la somme   des nombres entres */
double PROD;    /* le produit des nombres entres */


printf("Nombre de donnees : ");
scanf("%d", &N);


for (SOM=0, PROD=1, I=1 ; I<=N ; I++)
    {
     printf("%d. nombre : ", I);
     scanf("%d", &NOMB);
     SOM  += NOMB;
     PROD *= NOMB;
    }
```

Initiation à l'algorithmique
Les fonctions

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commençe par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

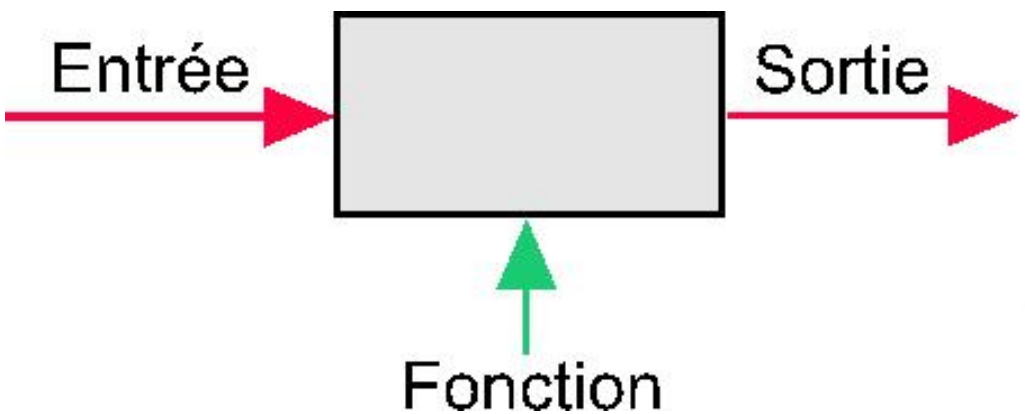
- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Solution : Notion de fonction

- On doit donc apprendre à nous organiser.
- On doit découper nos programmes en petits bouts.
- Chaque « petit bout de programme » sera ce qu'on appelle une **fonction**.

Fonction

Une fonction exécute des actions et renvoie un résultat. C'est un morceau de code qui sert à faire quelque chose de précis.

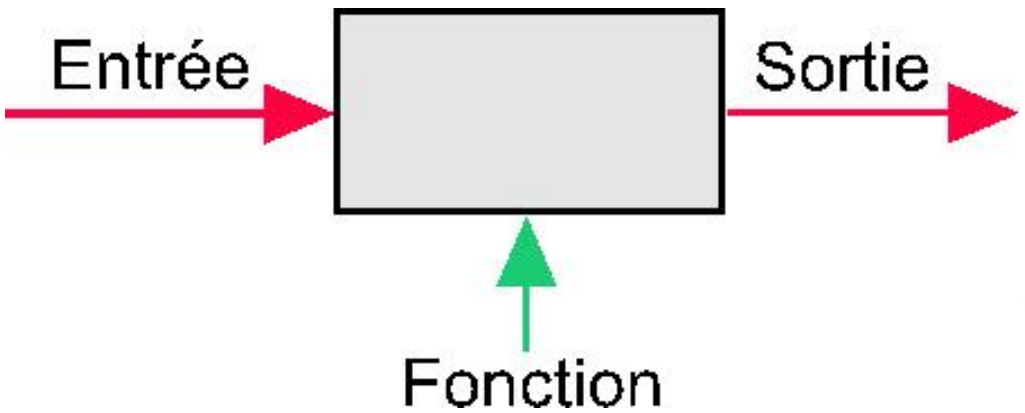


Solution : Notion de fonction

- On doit donc apprendre à nous organiser.
- On doit découper nos programmes en petits bouts.
- Chaque « petit bout de programme » sera ce qu'on appelle une **fonction**.

Fonction

Une fonction exécute des actions et renvoie un résultat. C'est un morceau de code qui sert à faire quelque chose de précis.



Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent
2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes
3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)
4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent
2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes
3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)
4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent

2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes

3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)

4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent

2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes

3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)

4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent

2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes

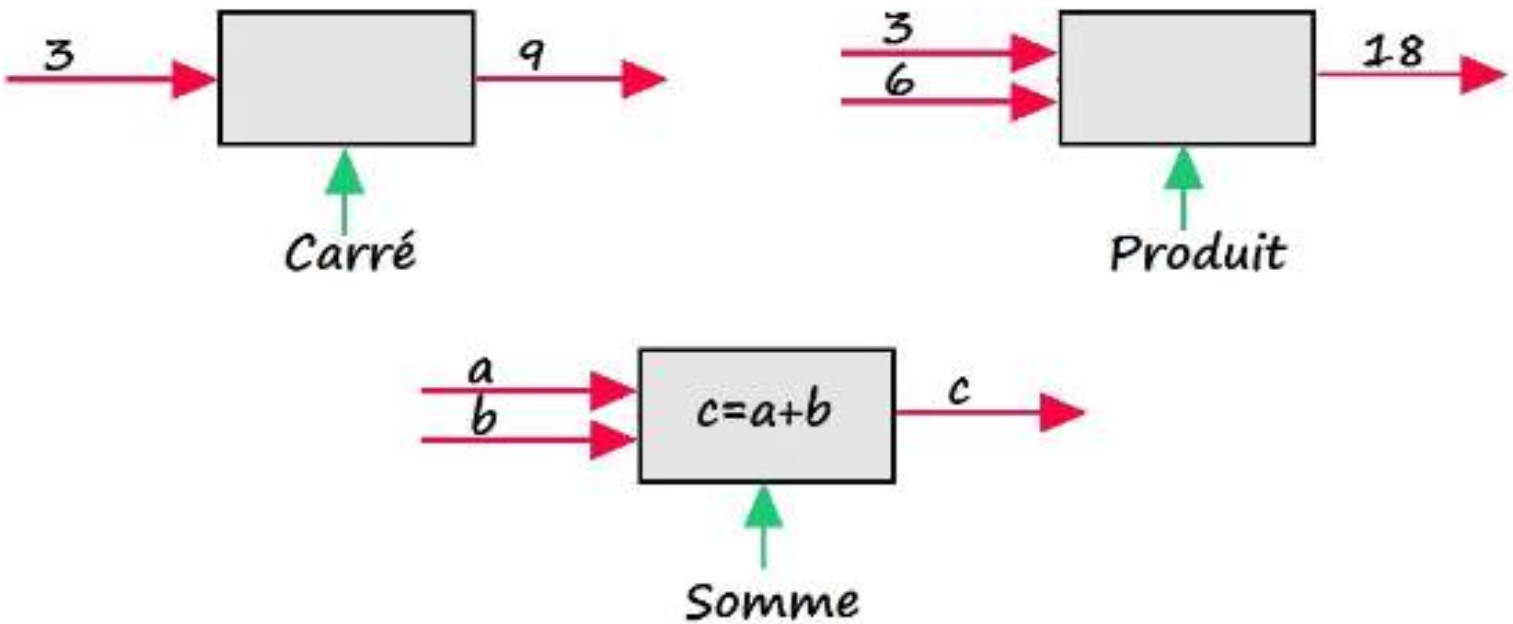
3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)

4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Principe

Une fonction est définie par trois éléments :

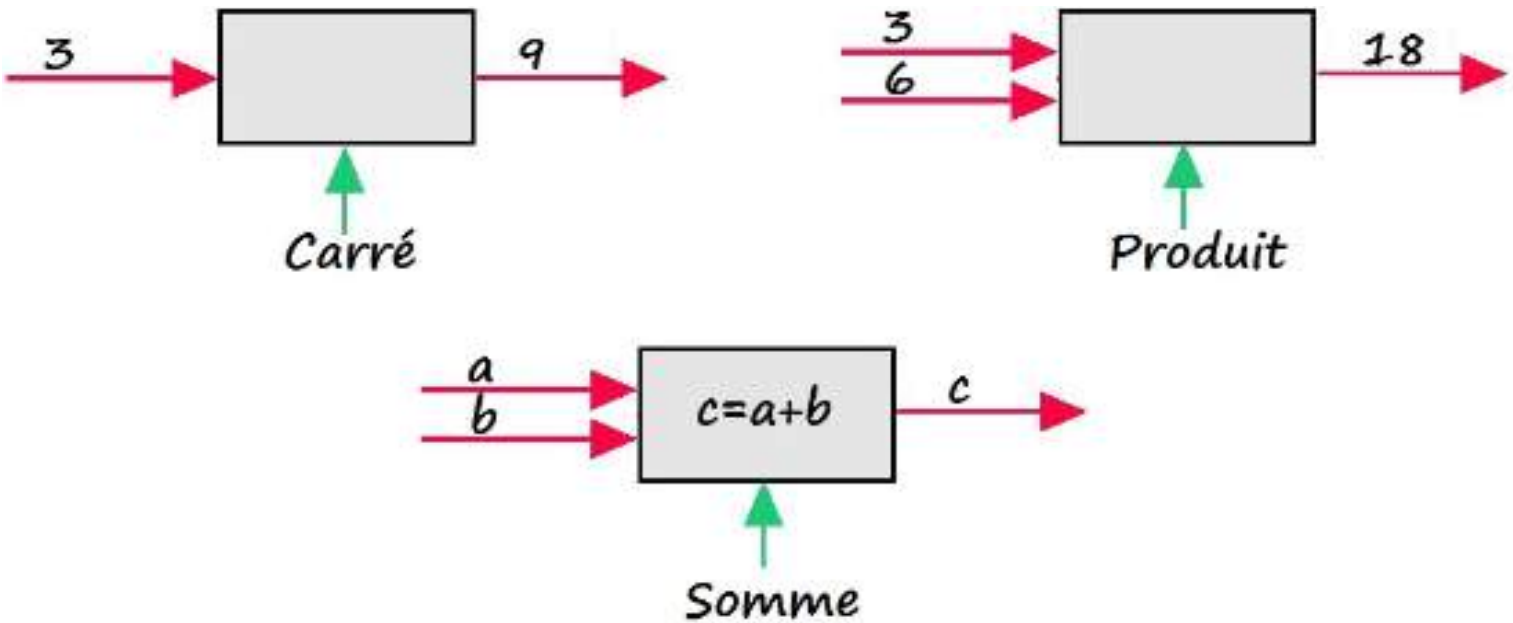
1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Principe

Une fonction est définie par trois éléments :

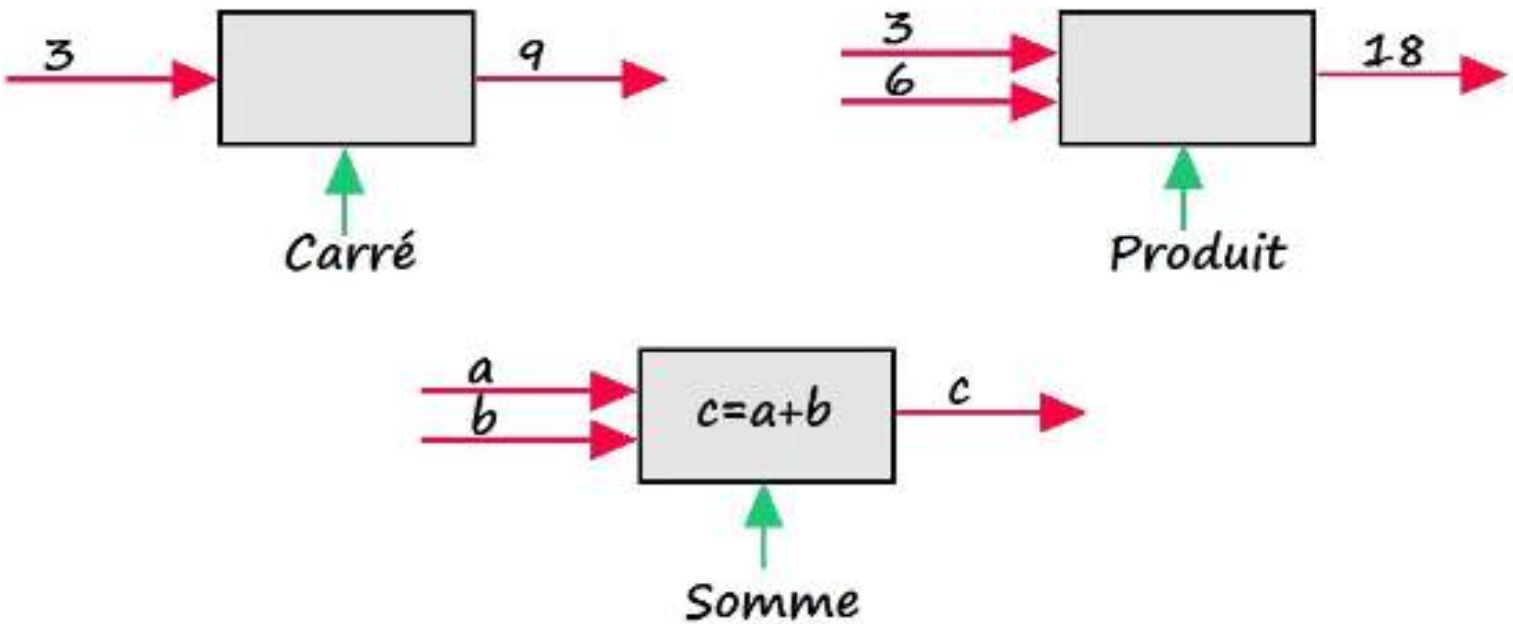
1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Principe

Une fonction est définie par trois éléments :

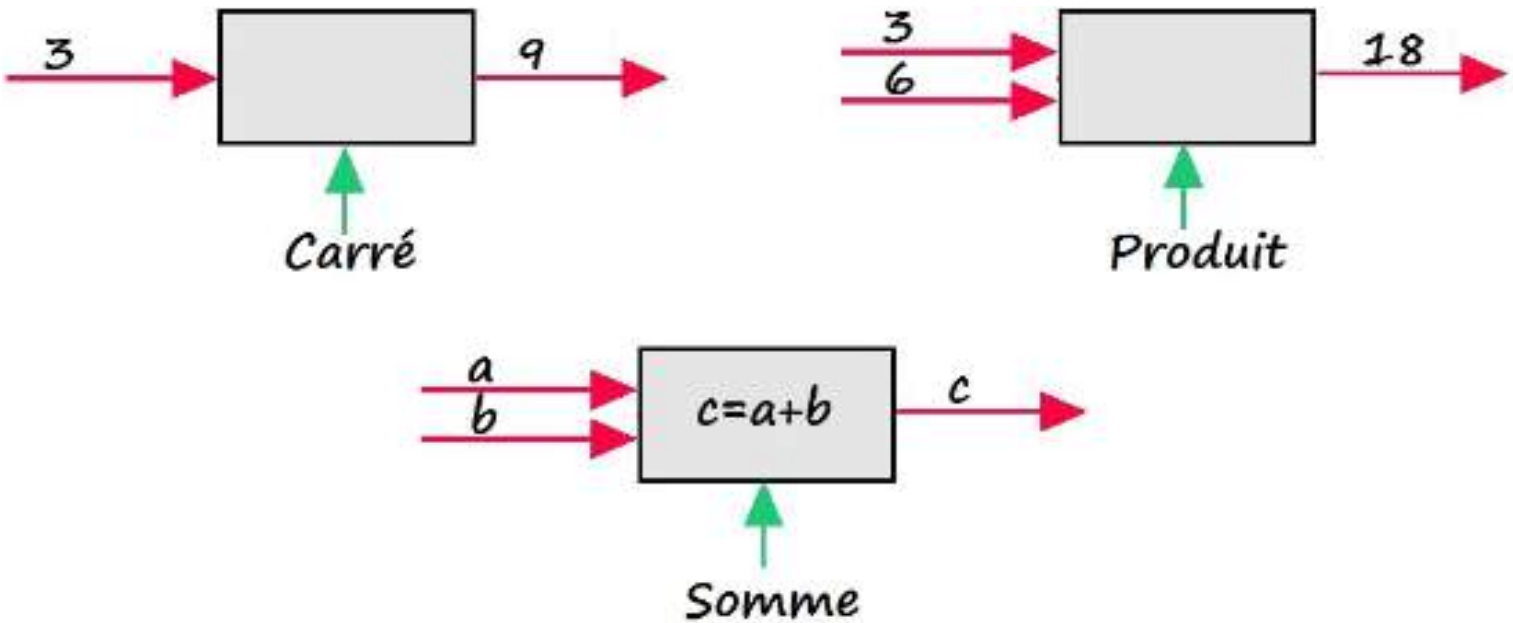
1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Principe

Une fonction est définie par trois éléments :

1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
  {
   <declarations locales>
   <instructions>
  }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
  {
   <declarations locales>
   <instructions>
  }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

L'instruction return

- L'instruction **return** permet de préciser quel est le résultat que la fonction doit retourner (renvoyer)
- On peut mentionner n'importe quelle expression après un **return**.

Exemple :

```
float polynome (float x, int b, int c)
 {
 float resultat;
 resultat = x * x + b * x + c
 return (resultat) ;

 // est equivalent a

 float polynome (float x, int b, int c)
 {
 return (x * x + b * x + c) ;
 }
```

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

Utilisation d'une fonction

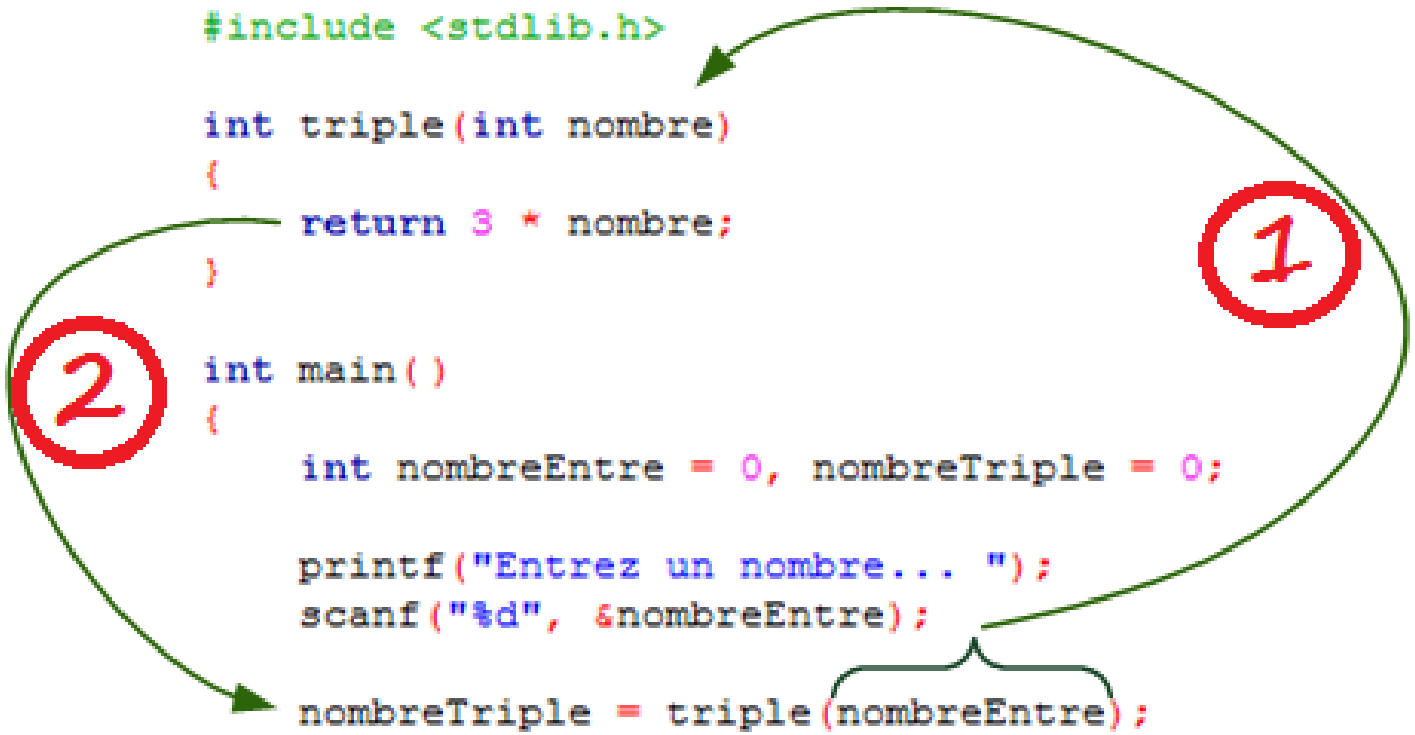
Il suffit de taper le nom de la fonction suivi des paramètres entre parenthèses.

Exemple :

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

int triple(int nombre) // 6
{
    return 3 * nombre; // 7
}
int main() // 1
{
  int nombreEntre = 0, nombreTriple = 0; // 2
  printf("Entrez un nombre... "); // 3
  scanf("%d", &nombreEntre); // 4

  nombreTriple = triple(nombreEntre); // 5

  printf("Le triple de ce nombre est %d\n",nombreTriple);//8
  return 0; // 9
}
```

Appel de fonction

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

int triple(int nombre)
{
    return 3 * nombre;
}

int main()
{
    int nombreEntre = 0, nombreTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);

    nombreTriple = triple(nombreEntre);
    printf("Le triple de ce nombre est %d\n", nombreTriple);

    return 0;
}
```



On n'est pas obligé de stocker le résultat d'une fonction

Exemple :

```c
int triple(int nombre)
{
    return 3 * nombre;
}
int main()
{
    int nombreEntre = 0;
    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);
    // Le resultat de la fonction est directement envoye au
        printf et n'est pas stocke dans une variable
    printf("Le triple de ce nombre est %d\n", triple(
        nombreEntre));
    return 0;
}
```

La fonction **main** appelle la fonction **printf**, qui elle-même appelle la fonction **triple**. C'est une imbrication de fonctions.

Paramètre formels Vs. Paramètre effectifs

```c
int triple(int nombre)
{
    return 3 * nombre;
}

int main()
{
        ...
        printf("Le triple est %d\n", triple(nombreEntre));
        ...
}
```

1. Les noms des arguments figurant dans l'en-tête de la fonction se nomment des **« paramètres formels »**. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.
2. Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des **« paramètres effectifs »**. on peut utiliser n'importe quelle expression comme argument effectif.

Paramètre formels Vs. Paramètre effectifs

```c
int triple(int nombre)
{
    return 3 * nombre;
}

int main()
{
        ...
        printf("Le triple est %d\n", triple(nombreEntre));
        ...
}
```

1. Les noms des arguments figurant dans l'en-tête de la fonction se nomment des **« paramètres formels »**. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.
2. Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des **« paramètres effectifs »**. on peut utiliser n'importe quelle expression comme argument effectif.

Passage de paramètres par valeur

Exemple :

```c
#include <stdio.h>

void fonction(int nombre)
{
    ++nombre;
    printf("Variable nombre dans la fonction : %d\n", nombre
        );
}
int main(void)
{
    int nombre = 5;
    fonction(nombre);
    printf("Variable nombre dans le main : %d\n", nombre);
    return 0;
}
```

Passage de paramètres par valeur

Exemple :

```c
#include <stdio.h>

void fonction(int nombre)
{
    ++nombre;
    printf("Variable nombre dans la fonction : %d\n", nombre
        );
}
int main(void)
{
    int nombre = 5;
    fonction(nombre);
    printf("Variable nombre dans le main : %d\n", nombre);
    return 0;
}
```

Variable nombre dans la fonction : 6

Passage de paramètres par valeur

Exemple :

```c
#include <stdio.h>

void fonction(int nombre)
{
    ++nombre;
    printf("Variable nombre dans la fonction : %d\n", nombre
        );
}
int main(void)
{
    int nombre = 5;
    fonction(nombre);
    printf("Variable nombre dans le main : %d\n", nombre);
    return 0;
}
```

Variable nombre dans la fonction : 6
Variable nombre dans le main : 5

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

Exemple :

```c
#include <stdio.h>
int carre(int nombre);

int main(void)
{
    int nombre, nombre_au_carre;
    puts("Entrez un nombre :");
    scanf("%d", &nombre);
    nombre_au_carre = carre(nombre);
    printf("Voici le carre de %d : %d\n", nombre,
        nombre_au_carre);
    return 0;
}

int carre(int nombre)
{
    nombre *= nombre;
    return nombre;
}
```

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.
- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.
- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.

- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).

- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.

- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.
- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.
- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.

- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).

- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.

- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.