Lycée TOUCHARD WASHINGTON

Développement logiciel

Programmation en Langage C

OBJECTIFS

À l'issue de cette séquence, vous devrez être capable de :

-

Niveau de maîtrise attendu pour le BTS **C**ybersécurité, **I**nformatique, **EL**ectronique option Informatique et Réseaux :

S4. Développement logiciel		**IR**
S4.1. Principe de base	Organisation des fichiers dans un projet logiciel ; La chaîne de développement (préprocesseur, compilateur, éditeur de lien, chargeur, etc.)	3
S4.2 Algorithmique	Structures fondamentales : enchaînements, alternatives, itérations, etc. Représentation graphique (organigrammes)	4
	Manipulations de texte (chaînes de caractères)	3
	Algorithmes de tri/de recherche	3
	Modèle canonique de gestion d'E/S : ouvrir, lire, écrire, fermer	3
	Bibliothèque standard (ANSI C)	3

CIEL Version 1.2

Sommaire

1. Organisation des fichiers dans un projet logiciel

1.1. Préambule

Les langages de programmation sont répartis en général en deux familles, les **langages interprétés** comme **Python** et les **langages compilés** comme le **langage C**. Dans le premier cas, les instructions fournies par le programmeur sont interprétées, ou traduites dans un langage propre au processeur, au fur et à mesure du déroulement du programme. Dans le second cas, l'ensemble des instructions est traduit une bonne fois pour toutes dans le langage du processeur par un compilateur et exécuté ensuite. Un langage interprété est donc réputé plus lent à l'exécution qu'un langage compilé du fait de la traduction avant l'exécution de chaque instruction. En revanche, un langage compilé fournit un code spécifique pour chaque environnement (processeur, système d'exploitation). Il est donc nécessaire de recompiler le programme pour le porter sur un autre système. Si le système dispose du bon interpréteur, un langage interprété est portable directement sous un autre système.

Certains langages comme **Java**, utilise un compromis entre les deux techniques. Le code source est traduit en pseudo-code une première fois et ensuite interprété par une machine virtuelle lors de l'exécution. Le langage est ainsi directement portable si l'autre système dispose de ladite machine virtuelle.

1.2. Le langage C

Dans ce document, il sera uniquement question du langage C et des mécanismes qui lui sont associés. Sa syntaxe a été reprise dans de nombreux autres langages et est facilement transposable.

Développé principalement par **Brian KERNIGHAN** et **Dennis RITCHIE**, le langage C a été conçu pour de multiples utilisations. Son origine remonte au début des années 70. Il a été utilisé pour l'écriture du système d'exploitation UNIX sur le DEC PDP-11. Cependant, ce langage n'est pas lié à une structure matérielle ou à une machine particulière. Il permet d'écrire facilement des programmes portables sur différents types d'ordinateur.

Une application C possède trois unités d'organisation, le **fichier**, la **fonction** et le **bloc**. Le fichier contient l'ensemble des fonctions et des données de l'application. Une fonction est une entité compilable réalisant un traitement particulier. Un bloc contient une suite d'instructions. Un programme C peut être constitué de multiples fichiers. Le nom d'un fichier source doit refléter la fonction d'usage réalisée par le fichier.

Un ou plusieurs fichiers avec l'extension **.c** contiennent l'implémentation des différentes fonctions, soit l'ensemble des instructions constituant le programme. Tandis que les fichiers d'entête, avec l'extension **.h,** contiennent la définition ou prototype des différentes fonctions précisant leur utilisation.

1.3. La chaîne de développement

Le développement de programme en langage C se fait généralement avec l'utilisation d'un EDI pour **E**nvironnement de **D**éveloppement **I**ntégré comme Netbeans, Qt Creator… Sous une même entité, ces environnements regroupent les différentes fonctionnalités d'écrites ci-après.

L'**éditeur de texte** permet la saisie du code source dans les différents fichiers.

Le **préprocesseur** est chargé ensuite d'interpréter les directives de compilation. Ces directives sont précédées du caractère #, elles permettent l'inclusion de fichier d'entête, la définition de constante et la compilation conditionnelle de certains blocs principalement.

Directive du préprocesseur

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

#define NB_ELEMENTS 15
```

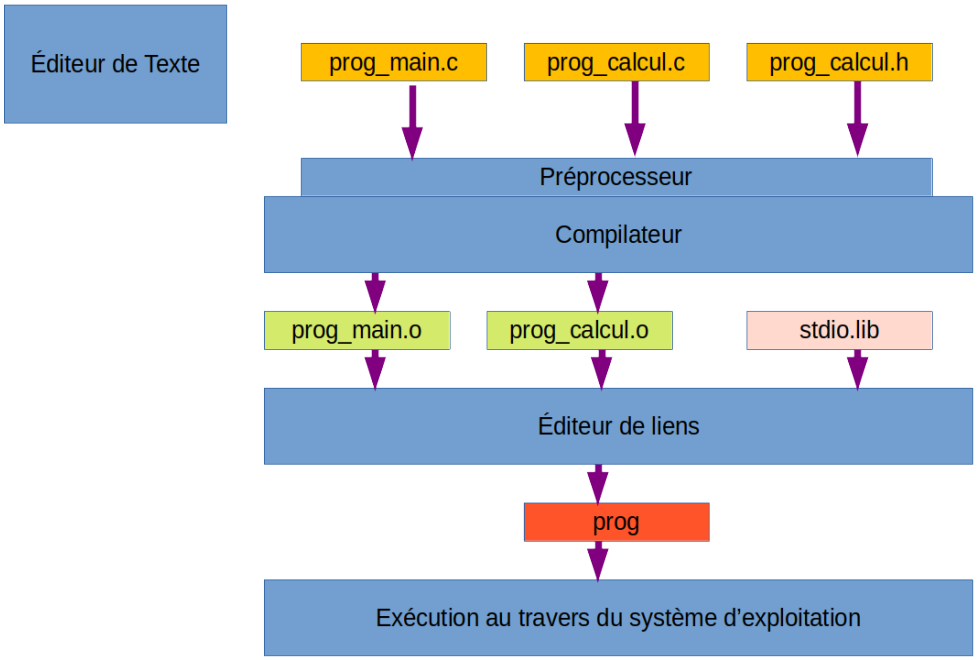
Les signes **<** et **>** désignent un fichier fourni par l'environnement de développement ici la bibliothèque standard stdio

Les signes " encadrent le nom d'un fichier se trouvant dans le répertoire de développement, un fichier d'entête que nous avons écrit par exemple.

La directive **#define** remplace dans l'ensemble du fichier les caractères NB_ELEMENTS par la valeur 15.

Le **compilateur** prend la suite pour traduire chaque fichier source écrit en langage C en fichier objet propre au processeur sur lequel le programme va être exécuté. Cette opération permet d'obtenir des fichiers **.o** ou **.obj** suivant les environnements Linux ou Windows. Ils ne sont pas encore exécutables directement.

L'étape suivante consiste à réaliser l'**édition de lien**. Cette opération regroupe l'ensemble des fichiers objets obtenus lors de l'étape précédente et les bibliothèques liées au langage pour obtenir un fichier unique exécutable. Sous Linux, il ne possède généralement pas d'extension ou **.exe** sous Windows. Il désigne également comme le point d'entrée du programme, la fonction **main()**, du programme ainsi réalisé.



Il est également possible de réaliser un programme exécutable en utilisant uniquement la ligne de commande et éventuellement un fichier **makefile** contenant les différentes préconisations de compilation et d'édition de liens. Les étapes restent identiques.

2. Le langage C

2.1. Les variables, les constantes et l'affectation

Dans un programme, une variable est un élément dont la valeur peut évoluer au fil du temps. Le langage C est un langage fortement typé ce qui signifie que chaque variable doit être déclarée pour être utilisée.

Pour sa déclaration, le nom de la variable est précédé de son type. Celui-ci ne change pas pour toute la durée de vie de la variable. Le nom des variables est formé d'une suite de lettres [a..z], [A..Z], de chiffres [0..9] ou du signe soulignement. Tous les autres caractères sont proscrits, seuls les 32 premiers caractères sont utilisés pour désigner le nom de la variable. Le nom d'une variable ne peut pas également commencer par un chiffre.

Par convention, les noms de variable commencent par une minuscule et forment un mot significatif pour une meilleure lisibilité du programme. Si le nom de la variable se compose de plusieurs mots, ils sont séparés par une majuscule.

Déclaration de variables simple :

```
int compteur ;
char carLu ;
float unReelSimplePrecision ;
double unReelDoublePrecision ;
```

Les variables non initialisées possèdent une valeur indéterminée. Il est donc important de procéder au plus tôt à leur initialisation soit par une affectation, soit par la lecture de leur valeur au clavier, ou dans un fichier...

Déclaration d'une variable simple suivie de son initialisation :

```
int compteur = 0 ;
```

Pour accroître la maintenabilité des programmes, il est fortement recommandé d'utiliser des constantes en place des valeurs numériques. La définition des constantes est réalisée avec la directive du préprocesseur **#define**.

Par convention, la définition d'une constante utilise uniquement des lettres majuscules avec éventuellement le caractère de soulignement pour séparer deux mots.

Attention, la valeur de la constante n'est jamais suivie d'un point-virgule, car le préprocesseur réalise un remplacement du nom par la valeur dans tout le programme avant la compilation.

Définition de constantes

```
#define NB_ELEMENTS 15
#define TAUX 12.8
```

L'affectation d'une variable consiste à lui attribuer une valeur au cours de l'exécution du programme à l'aide de l'opérateur d'affectation, le signe égal « **=** ».

Définition de constantes

```
int compteur = 0 ;
compteur = compteur + 1 ; // équivalent algorithmique de compteur ← compteur + 1
```

La variable à gauche du signe **=** reçoit la valeur ou l'expression à droite du signe.

2.2. Les opérateurs

2.2.1. Les opérateurs arithmétiques :

Les opérateurs arithmétiques s'appliquent sur tous les types numériques entier ou réel, quelle que soit leur taille à l'exception de l'opérateur modulo qui n'est pas utilisable avec les réels.

Opérateur	Rôle	Exemples
+	Addition	2 + 4 vaut 6
-	Soustraction	4 – 2 vaut 2
*****	Multiplication	2 * 4 vaut 8
/	Division, si usage avec des réels	9.0 / 2.0 vaut 4.5
	Division entière, si usage avec des entiers	9 / 2 vaut 4
%	Modulo soit le reste de la division entière	11 % 5 vaut 1 4 % 2 vaut 0

La priorité s'applique de gauche à droite. La multiplication, la division et le modulo sont prioritaires sur l'addition et la soustraction. Pour des raisons de simplicité, il convient d'utiliser des parenthèses pour lever toute ambiguïté.

Ces opérations s'entendent sur des opérandes de même type, fournissant un résultat également de même type. Si ce n'est pas le cas, un changement de type implicite est mis en œuvre par le compilateur afin que le calcul soit fait dans le type dominant. La hiérarchie des types est : ***char < short < int < long < float < double***

Le langage C propose également une écriture simplifiée pour les opérations du type ***val1 = val1 + val2***, l'opération peut s'écrire ***val1 += val2***. Il en est de même pour la soustraction, la multiplication et la division.

L'opérateur **modulo** permet également de simplifié l'écriture lorsque l'on souhaite une remise à zéro d'une valeur après une incrémentation par exemple ainsi : ***val1 = (val1+1) % 10***, val1 prendra successivement les valeurs 0 à 9 puis recommencera à partir de 0.

Les opérateurs **++** ou **--** permettent respectivement l'incrémentation ou la décrémentation d'une variable, exemple ***val1++*** est équivalent à ***val1 = val1 + 1***. Attention cependant, ces opérateurs peuvent se placer avant ou après la variable, on parle alors de **post** ou **pré** incrémentation (décrémentation).

Exemple : **val1 = val2++** équivaut à **val1 = val2** et **val2 = val2 + 1** (l'incrémentation est faite après l'affectation)

val1 = ++val2 équivaut à **val2 = val2 + 1** et **val1 = val2** (l'incrémentation est faite avant l'affectation)

2.2.2. Les opérateurs binaires

Ces opérateurs permettent des manipulations des variables au niveau du bit. Les opérandes sont généralement des entiers, des caractères signés ou non.

Pour les exemples :

val1 = 0xAA soit 1010 1010 val2 = 0x6E soit 0110 1110

Opérateur	Rôle	Exemple	Résultat : val3		Exemple d'utilisation
|	Opérateur OU	val3 = val1 | val2	0xEE	1110 1110	Mise à 1 de bits
&	Opérateur ET	val3 = val1 & val2	0x2A	0010 1010	Mise à 0 de bits
^	Opérateur OU exclusif	val3 = val1 ^ val2	0xC4	1100 0100	Calcul de sommes de contrôle
~	Complément à 1	val3 = ~val1	0x55	0101 0101	Inverse les 1 et les 0
>>	Décalage à droite	val3 = val1 >> 4	0x0A	0000 1010	1 décalage → divise par 2
<<	Décalage à gauche	val3 = val2 << 1	0xDC	1101 1100	1 décalage → multiplie par 2

Pour les opérations de décalage, des 0 sont ajoutés à gauche ou à droite suivant le cas, les bits décalés hors du mot sont perdus.

2.2.3. L'opérateur sizeof

Cet opérateur fournit la taille en octets du type de la variable ou du type passé en paramètre. La taille de la variable est liée au système et au processeur sur lequel le programme est implémenté. Ainsi un **int** ne possédera pas la même taille sur un système 32 bits ou sur un système 64 bits.

Utilisation : sizeof (***type ou nom de variable***)

L'opérateur **sizeof** n'est pas limité aux types simples. Son utilisation est courante pour les tableaux, les structures, les pointeurs... Il permet de réaliser une programmation plus portable.

Utilisation de sizeof

```
int compteur = 0 ;
sizeof(int) // vaut 2 ou 4 suivant le compilateur équivalent à sizeof(compteur)
```

2.2.4. L'opérateur &

Cet opérateur est l'opérateur d'adresse, il fournit l'adresse du premier octet occupé par une variable en mémoire. Il est très utile pour le passage de paramètre par adresse d'une variable afin d'obtenir un paramètre de sortie.

Important

&var représente l'adresse de la variable **var**

Utilisation de l'opérateur &

```
int compteur = 0 ;
int *ptrCompteur = &compteur ;

// la variable ptrCompteur reçoit l'adresse de la variable compteur.
// la variable ptrCompteur est un pointeur sur un entier, d'ou * lors de la déclaration.
// Cette notion est reprise plus tard dans ce document.
```

2.3. Les structures de contrôle

2.3.1. Évaluation d'un prédicat

Toute structure de contrôle est soumise à l'évaluation d'un prédicat. Le résultat de cette évaluation conditionne le déroulement du programme, il ne peut être que vrai (TRUE) ou faux (FALSE), lorsqu'il y a plus de deux termes à comparer, le prédicat devient un prédicat composé à l'aide des connecteurs **ET** et **OU**. Parfois, il est plus simple d'exprimer la condition inverse, le connecteur **NON** permet d'inverser cette condition.

Contrairement à d'autres langages, le C ne possède pas de type booléen. Au besoin, ce type peut-être redéfini comme le montre l'exemple ci-après.

Exemple définition du type booléen

```
#define BOOL unsigned int
#define TRUE 1
#define FALSE 0
```

Comparaisons	Algorithmique	Langage C
Égale	=	==
Différent	≠	!=
Inférieur	<	<
Inférieur ou égale	≤	<=
Supérieur	>	>
Supérieur ou égale	≥	>=

Connecteurs	Langage C
ET	&&
OU	||
NON	!

Il est parfois nécessaire d'utiliser plusieurs niveaux de parenthèse pour respecter la priorité entre les différents connecteurs.

Les conditions utilisées au travers des différentes structures de contrôles, sont toujours exprimées entre parenthèses en langage C. Elles encadrent les prédicats. De la même manière, l'appel d'une fonction peut être utilisé à la place d'un prédicat, dès l'instant où cette fonction possède un paramètre de retour.

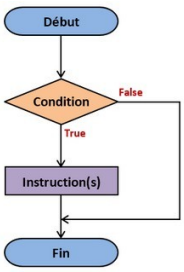
Pour des raisons d'efficacité, le langage C arrête l'évaluation d'un prédicat composé lorsque la condition est évaluable. Ainsi pour le prédicat composé ***(a* OU *b)*** si ***a*** est vrai, ***b*** ne sera pas évalué, la condition est vraie. De même, pour le prédicat ***(a* ET *b)*** si ***a*** est faux, alors ***b*** ne sera pas évalué, la condition restera fausse. Ceci peut avoir des conséquences si un prédicat est remplacé par un appel de fonction, qui ne sera donc pas appelé ou si une auto-incrémentation doit-être effectuée lors de l'évaluation du second prédicat.

Remarque

En langage C, toute valeur égale à 0 est considérée comme fausse, toute valeur différente de 0 est considérée comme vraie.
Le test d'égalité ou d'inégalité entre deux réels est à proscrire, le résultat est incertain. Il dépend des arrondis effectués lors du codage en mémoire.

2.3.2. Alternative simple : si … alors …

Ce schéma conditionnel permet de prendre une décision en réponse à l'évaluation d'un prédicat. Si l'évaluation du prédicat est vraie alors le traitement soumis à la condition est réalisé. Tout prédicat en langage C est exprimé entre parenthèses.



Algorithmique	Langage C

Si condition
Alors Instruction(s)
FinSi
	if (condition)
	{
	// instructions exécutées si la condition est vraie
	}

Exemple d'alternative simple — **SiAlors1.c**

```
#include <stdio.h>

int main()
{
    char carLu ;
    printf("Appuyer sur une touche puis Entrée :");
    carLu = getchar();
    if(carLu >= '0' && carLu <= '9')
    {
        printf("Vous avez saisi un chiffre \n");
    }
    printf("Fin du programme\n");
    return 0;
}
```

Pour illustrer l'utilisation d'un appel de fonction en place d'un prédicat, ce même programme peut s'écrire d'une autre manière en appelant la fonction *int* ***isdigit****(int character);* de la librairie *ctype.h.* Cette fonction permet de tester si le caractère passé en paramètre représente un nombre décimal.

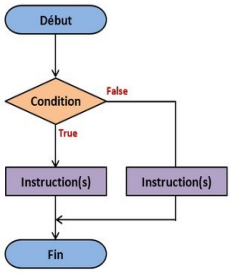
Exemple d'alternative simple — **SiAlors2.c**

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

int main()
{
    char carLu ;
    printf("Appuyer sur une touche puis Entrée :");
    carLu = getchar();
    if(isdigit(carLu))
    {
        printf("Vous avez saisi un chiffre \n");
    }
    printf("Fin du programme\n");

    return 0;
}
```

La valeur de retour est interprétée en tant que valeur booléenne. Une valeur positive non nulle signifie qu'il s'agit bien d'un chiffre décimal, la valeur `0` indique le contraire.

2.3.3. Alternative composée : si … alors … sinon …

Ce schéma conditionnel permet de réaliser un traitement particulier lorsque l'évaluation de la condition est vraie et un autre lorsqu'elle est fausse.



Algorithmique	Langage C

Si condition
Alors Instruction(s)
Sinon Autre(s)
FinSi
	if (condition)
	{
	// instructions exécutées si la condition est vraie
	}
	else
	{
	// instructions exécutées si la condition est fausse
	}

Exemple d'alternative composée — **SiAlorsSinon.c**

```c
#include <stdio.h>

int main()
{
    char carLu ;
    printf("Appuyer sur une touche puis Entrée :");
    carLu = getchar();
    if(carLu >= '0' && carLu <= '9')
    {
        printf("Vous avez saisi un chiffre \n");
    }
    else
    {
        printf("Vous n'avez pas saisi un chiffre \n");
    }
    printf("Fin du programme\n");
    return 0;
}
```

Exercices

a) Écrire un programme en langage C permettant de déterminer si une touche est un caractère majuscule, minuscule, un chiffre ou autre chose…

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

b) Après avoir observé la table des codes ASCII ci-dessus, expliquez comment fait le programme pour savoir si une lettre est comprise entre deux valeurs.

2.3.4. Imbrication des alternatives : Si … alors … sinon … si ….

Il est tout à fait possible d'imbriquer des structures conditionnelles en respectant parfaitement l'encastrement des blocs. Du point de vue algorithmique, la fin du bloc est marquée par ***FinSi*** en langage C, le bloc se termine par l'accolade fermante **}**.

ImbricationAlternatives.c

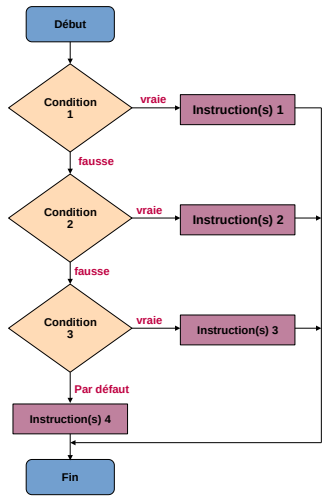
```
Si (val1 > 0 ou val2 > val3) et
                (val4 > val1 ou val4 > 5)
   Alors
      val1 ← 0
      val4  ← val2 + val3
   Sinon
      val3   ← val1 - val2
      Si val3 > 0
         Alors val4  ←  - val4érial
      FinSi
      val2 ←  0
FinSi
```

```
if((val1 > 0 || val2 > val3) && (val4 > val1 || val4 > 5))
{
    val1 = 0;
    val4 = val2 + val3;
}
else
{
    val3 = val1 - val2;
    if (val3 > 0)
    {
        val4 = -val4;
    }
    val2 = 0;
}
```

Les accolades ne sont pas toujours indispensables pour matérialiser un bloc lorsque celui-ci ne comporte qu'une seule instruction, mais elles facilitent la lecture et lèvent toute ambiguïté lors de l'imbrication de structures de contrôle.

2.3.5. Choix multiple : cas … parmi …

Avec cette structure de contrôle, seule la comparaison d'une variable **entière** avec différentes constantes est possible. Pour toute comparaison faisant appel à un autre opérateur ou un autre type, il est nécessaire de revenir aux instructions d'alternative classique présentées précédemment.



Algorithmique

Ici VAL1, VAL2, VAL3 représentent des constantes.

Cas **nomDeVariable** parmi
VAL1 : Instruction(s) 1
VAL2 : Instruction(s) 2
VAL3 : Instruction(s) 3
...
Par défaut : Instruction(s) 4
FinDesCas

Langage C

```
switch (nomDeVariable)
{
  case VAL1:  // instruction(s) 1
        break;
  case VAL2:  // instruction(s) 2
        break;
  case VAL3:  // instruction(s) 3
        break;
  default:    // instruction(s) 4
}
```

Contrairement à la notion introduite en algorithme à travers le **cas … parmi …** et dans d'autres langages comme le langage Pascal, le **switch(…) case…** du langage C est inclusif. C'est-à-dire que lorsqu'une condition est vérifiée toutes les suivantes le sont aussi. Sauf, si on précise l'arrêt du traitement par l'instruction **break**.

Le cas par défaut permet de réaliser un traitement spécifique si aucun des cas n'est vérifié.

Remarques

Les caractères sont considérés comme des entiers et peuvent être donc utilisés dans le switch(…) case …
Ils sont simplement mis entre apostrophes ' '.
Le cas par défaut est facultatif. Cependant, certains compilateurs signalent son absence, pour que tous les cas, même ceux d'erreurs, soient bien pris en compte. Cela permet d'accroître la qualité du code.

1er exemple de choix multiple — calculatrice.c

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int val1 = 6;
    int val2 = 2;
    char operateur;
    printf("Choisissez une opération parmi : +, -, *, / : ");
    operateur = getchar();
    switch (operateur)
    {
        case '+': printf("\n%d + %d = %d\n",val1,val2,val1+val2);
            break;
        case '-': printf("\n%d - %d = %d\n",val1,val2,val1-val2);
            break;
        case '*': printf("\n%d * %d = %d\n",val1,val2,val1*val2);
            break;
        case '/': printf("\n%d / %d = %d\n",val1,val2,val1/val2);
            break;
        default: printf("\nJe ne connais pas cette opération\n");
    }

    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Editer  Affichage  Rechercher  Terminal  Aide
Choisissez une opération parmi : +, -, *, / : +

6 + 2 = 8
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
Terminal
Fichier  Editer  Affichage  Rechercher  Terminal  Aide
Choisissez une opération parmi : +, -, *, / : -

6 - 2 = 4
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
Terminal
Fichier  Editer  Affichage  Rechercher  Terminal  Aide
Choisissez une opération parmi : +, -, *, / : *

6 * 2 = 12
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
Terminal
Fichier  Editer  Affichage  Rechercher  Terminal  Aide
Choisissez une opération parmi : +, -, *, / : /

6 / 2 = 3
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
Terminal
Fichier  Editer  Affichage  Rechercher  Terminal  Aide
Choisissez une opération parmi : +, -, *, / : #

Je ne connais pas cette opération
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Cet exemple illustre le mode **exclusif** de l'instruction ***switch(…) case …*** avec l'utilisation de l'instruction ***break***. Afin d'accroître les performances du langage, le fait de ne pas utiliser systématiquement cette instruction d'arrêt permet de réunir des traitements communs. Ce deuxième exemple illustre ce mode de fonctionnement **inclusif**.

2e exemple de choix multiple — traitementCommun.c

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

int main()
{
    char lettre;
    printf("Choisissez un chiffre : ");
    lettre = getchar() - '0'; // pour transformer le code ASCII en chiffre
    switch (lettre)
    {
        case 1:
        case 2:
        case 3: printf("\nCas de 1 à 3\n");
                break;
        case 4:
        case 5:
        case 6: printf("\nCas de 4 à 6\n");
                break;
        case 7:
        case 8:
        case 9: printf("\nCas de 7 à 9\n");
                break;
        case 0: printf("\nCas 0\n");
            break;

        default: printf("\nCe cas n'est pas répertorié\n");
    }

    return EXIT_SUCCESS;
}
```

Si l'utilisateur saisit indifféremment le chiffre 1, 2 ou 3, le système affiche « Cas de 1 à 3 ». Le fonctionnement est similaire pour les autres cas.

Exercices d'application

a) Réalisez un programme permettant de déterminer si une année est bissextile ou pas. L'année sera contenue dans une variable initialisée au début du programme. Vérifiez pour les valeurs proposées dans l'illustration suivante :

b) Une entreprise rémunère ses salariés à l'heure. Chaque semaine, elle calcule leur rémunération suivant un taux horaire auquel, elle applique un coefficient donné par le tableau suivant :

Horaire	Coefficient
Les 35 premières heures	1
De la 36e à la 39e heure incluse	1,2
De la 40e à la 45e heure incluse	1,5
Au-delà de la 45e heure	1,8

Réalisez un programme permettant de calculer et d'afficher le salaire de la semaine en fonction du volume d'heures réalisé et du taux horaire.

Le taux horaire sera fixé à **11,52 €** et sera défini en tant que **constante** dans le programme

La variable **volumeHeures**, dont vous choisirez le type, sera déclarée et initialisée au début du programme.

2.3.6. Les itérations

Ces structures de contrôle permettent, en fonction d'une condition de maintien, la répétition d'une suite d'instructions, constituant le corps de la boucle, un certain nombre de fois. Il est généralement nécessaire de s'assurer que dans le corps de la boucle, il y a moyen de faire évoluer de la condition de maintien sous peine de rester indéfiniment bloqué dans la boucle.

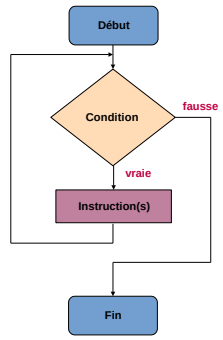
Deux situations peuvent se présentent :

- On ne connaît pas à l'avance le nombre de répétitions, c'est le cas lorsque l'on attend l'arrivée d'un événement ou l'action d'un utilisateur. C'est une **boucle indéfinie.**
- On connaît le nombre de répétitions avant de faire la boucle, on parle alors de **boucle définie**. C'est le cas pour répéter une suite d'actions n fois.

Pour la boucle indéfinie, deux cas peuvent se présenter, pour le premier, la condition de maintien est évaluée au début de la boucle, pour le second, elle est évaluée à la fin.

Boucle indéfinie avec la condition de maintien en début de boucle : Tant que …

Pour exécuter la suite d'instruction réalisant le corps de la boucle, la condition de maintien doit être vérifiée avant de commencer la boucle.



Algorithmique	Langage C

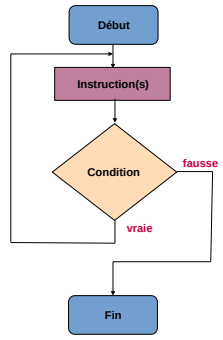
Tant que prédicat
Instruction(s)
FinTantQue
	while (/* Prédicat */)
	{
	// instruction(s) du corps de la boucle
	}

Exemple : Tant que … pgcd.c

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

int main()
{
    int var1 = 45;
    int var2 = 33;
    while (var1 * var2 != 0)
    {
        if(var1 > var2)
            var1 -= var2;
        else
            var2 -= var1;
    }
    if(var1 != 0)
        printf("Le plus grand commun diviseur est %d\n",var1);
    else
        printf("Le plus grand commun diviseur est %d\n",var2);
    return EXIT_SUCCESS;
}
```

Boucle indéfinie avec la condition de maintien en fin de boucle : Faire … Tant que …

Cette boucle impose de faire au moins une fois les instructions avant de vérifier la condition de maintien pour éventuellement recommencer.



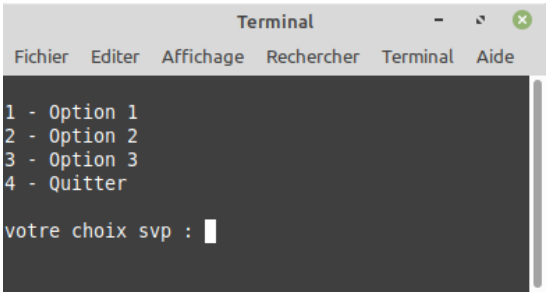
Algorithmique	Langage C

Faire
Instruction(s)
Tant que prédicat
	do
	{
	// instruction(s) du corps de la boucle
	}
	while (/* Prédicat */) ;

Attention : L'instruction ***do … while(…)*** se termine toujours par un **point-virgule**.

Exemple : Faire … Tant que **menu.c**

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

int main()
{
    char choix = 0;

    do
    {
        if(choix != '\n') // pour ne pas afficher 2 fois le menu à cause de l'appui sur la touche Entrée
            printf("\n"
                   "1 - Option 1\n"
                   "2 - Option 2\n"
                   "3 - Option 3\n"
                   "4 - Quitter\n"
                   "\nvotre choix svp : ");
        choix = getchar();
    } while (choix < '1' || choix > '4');

    printf("Vous avez choisi l'option : %c\n",choix);
    return EXIT_SUCCESS;
}
```



L'utilisateur est sollicité tant qu'il n'a pas appuyé sur une touche comprise entre 1 et 4. La boucle doit au moins être exécutée une fois pour afficher le menu et attendre le choix de l'utilisateur.

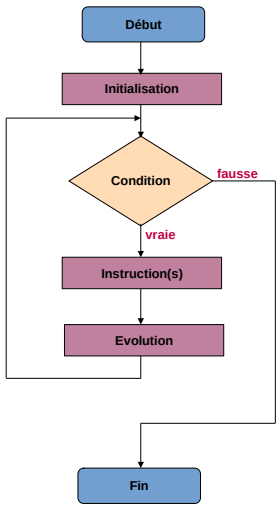
Exercices

a) En vous inspirant de l'exemple ci-dessus, modifiez le programme calculatrice.c pour que l'utilisateur puisse enchaîner les opérations tant qu'il n'appuie pas sur une autre touche que celles correspondant aux opérateurs arithmétiques de base.

b) Modifier à nouveau le programme calculatrice.c en ajoutant une variable booléenne nommée sortie. Elle est initialisée à FAUX lors de sa déclaration puis, passe à VRAI lorsque l'utilisateur n'appuie pas sur un des opérateurs arithmétiques traités dans votre programme.

Boucle définie : Pour …

Afin de comptabiliser le nombre d'itérations, un indice est initialisé à une valeur de départ, puis est évalué afin de vérifier si au moins une première fois, les instructions à répéter doivent être exécutées. À la suite de la dernière instruction constituant le corps de cette boucle, l'indice évolue pour passer à la valeur suivante. La condition de maintien est alors de nouveau évaluée pour savoir si un autre tour de boucle doit être effectué, ainsi de suite à chaque passage.



Algorithmique

Pour indice allant de 0 à NB_VAL pas de 1
    Instruction(s)
FinPour

Langage C

```
int indice;

for(indice = 0 ; indice <= NB_VAL ; indice++)
{
    // Instruction(s)
}
```

Indice est la variable permettant de comptabiliser le nombre d'itérations, elle est initialisée à 0 et évolue jusqu'à la valeur NB_VAL de 1 en 1. La boucle sera donc effectuée NB_VAL+1 fois. La valeur initiale, le pas d'évolution ainsi que la valeur finale sont fonction du traitement à effectuer. L'évolution peut être positive ou négative, les bornes sont choisies en conséquence.

Écriture classique du Pour...

```
int indice;

for(indice = 0 ; indice <= NB_VAL ; indice++)
{
    // Instruction(s)
}
```

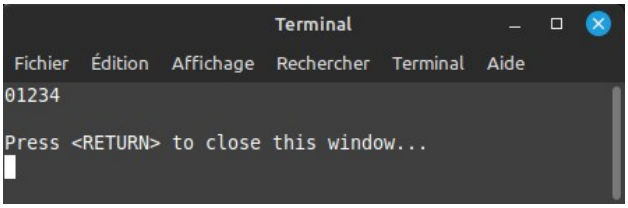
Écriture équivalente avec un Tant que...

```
int indice;
indice = 0;
while (indice <= NB_VAL)
{
   // Instruction(s)
   indice++;
}
```

La condition de maintien est évaluée avant d'effectuer le premier tour de boucle à la manière d'un tant que…

Exemple : Pour … **menu.c**

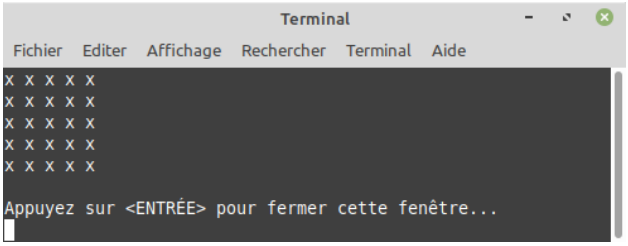
```
#include <stdio.h>
#include <stdlib.h>
#define NB_VAL 5

int main()
{
    int indice;

    for(indice = 0 ; indice < NB_VAL ; indice++)
    {
        printf("%d",indice);
    }
    printf("\n");
    return EXIT_SUCCESS;
}
```



Comme pour les autres structures de contrôle, il est tout à fait possible de les imbriquer entre elles.

Exemple : Pour … **menu.c**

```c
#include <stdio.h>
#include <stdlib.h>
#define NB_VAL 5

int main()
{
    int ligne;
    int colonne;
    for(ligne = 0 ; ligne < NB_VAL ; ligne++)
    {
        for (colonne = 0;colonne < NB_VAL ; colonne++)
        {
            printf("x ");
        }
        printf("\n");
    }
    printf("\n");
    return EXIT_SUCCESS;
}
```



Exercices d'application

a) En vous inspirant du programme précédent, réalisez un programme qui réalise un triangle isocèle, comme le montre la figure suivante :

x x x x x
x x x x
x x x
x x
x

b) Réaliser un programme permettant d'afficher la figure suivante représentant un sablier en fonction d'une constante TAILLE, avec TAILLE impaire.

Pour l'exemple, ici TAILLE vaut 5.

1	2	3	4	5
0	6	7	8	0
0	0	9	0	0
0	10	11	12	0
13	14	15	16	17

Pour résoudre ce problème, on propose la méthode suivante :

Pour chaque ligne, déterminer le nombre de zéro et le nombre de chiffres à écrire, puis écrire des zéros, écrire des chiffres, écrire des zéros.

On remarque que jusqu'à la moitié de la figure, le nombre de zéro augmente d'une unité chaque fois et que le nombre de chiffres diminue de 2. Pour la seconde moitié de la figure, c'est le contraire.

b) Construite pour l'exposition universelle de 1889, la tour Eiffel possède une hauteur de 320.755m (antenne comprise). À partir d'une feuille de papier d'une épaisseur de 0.076474 millimètre, et d'une surface suffisante, combien de fois faudrait-il plier la feuille en deux pour atteindre la hauteur de la tour ?

Réalisez le programme qui permet de résoudre ce problème et affiche le résultat.

Remarques : Il n'y a pas d'espace entre les pliures,
Les données numériques sont définies dans des constantes

Rappel : 1 m équivaut à 100 cm,
1 cm équivaut à 10 mm

2.4. Les fonctions

2.4.1. Introduction

Les fonctions sont un des piliers du langage C. En effet, la syntaxe du langage étant restreinte aux structures de contrôle et aux différentes opérations de base étudiées dans les précédents chapitres, les fonctions permettent d'interagir avec l'utilisateur, de réaliser des opérations mathématiques hormis les opérateurs de base, de manipuler des chaînes de caractères…

Pour une meilleure appréhension des problèmes, le programmeur est lui aussi amené à découper ses projets en différentes fonctions pour réduire la complexité de son travail. Une fonction doit généralement se limiter à la résolution d'une problématique. Chaque fonction constitue une entité compilable réalisant un certain traitement.

C'est également une manière de réduire le code en évitant de nombreuses répétitions d'une même suite d'instructions.

Pour plus de flexibilité, les fonctions peuvent recevoir des paramètres et retourner un résultat à la manière d'une fonction mathématique. L'algorithme ainsi paramétré peut s'adapter à diverses situations.

2.4.2. Utilisation des fonctions standards du langage C

Les fonctions standards du langage C sont implémentées sous la forme de collections normalisées formant différentes bibliothèques. Le programmeur n'a pas à se soucier de la manière dont elles sont réalisées. Il doit juste savoir comment les utiliser, c'est le rôle **du prototype de la fonction**.

Le prototype indique le type du paramètre de retour, s'il y en a un, le nom de la fonction et entre parenthèses les éventuels paramètres que la fonction reçoit.

Exemple de prototype avec un paramètre de retour

```
// Prototype de la fonction tolower, elle convertit si nécessaire une lettre majuscule en minuscule
int tolower ( int c );
        // Paramètre d'entrée C un entier, la lettre que l'on souhaite convertir en minuscules
        // Paramètre de retour de type int, la lettre éventuellement convertit en minuscule
```

Le type de retour dépend du rôle de la fonction, en langage C seul un élément peut être retourné (entier, caractère, réel…). Si toutefois la fonction ne retourne pas de valeur, son type de retour est **void** pour indiquer qu'elle ne retourne rien.

Exemple de prototype sans paramètre de retour

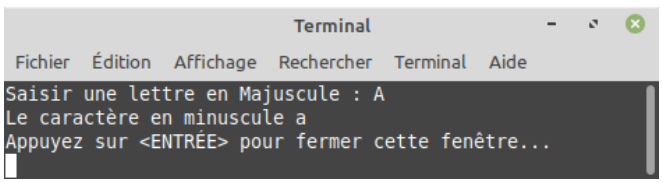
```
// Prototype de la fonction Allumer, Allume une led en (x,y) de la couleur choisie sur la carte sensHat
void Allumer(int x, int y, unit_16 couleur);
        // Paramètre d'entrée : x, y et couleur deux entiers pour la position sur la matrice et la
                                couleur définie par une valeur sur 16 bits non signés.
        // Paramètre de retour : il n'y en a pas la fonction ne retourne rien.
```

Bien souvent, les prototypes de fonctions sont regroupés dans un fichier d'entêtes, exemple : <stdio.h>, <math.h>

Pour appeler une fonction dans un programme, il est nécessaire de nommer la fonction et de lui passer les éventuels paramètres attendus et si nécessaire d'affecter le paramètre de retour de la fonction à une variable.

Exemple d'appel de fonction

```
#include <stdio.h>    // inclusion du fichier stdio.h pour getchar et printf
#include <stdlib.h>   // inclusion du fichier stdlib.h pour la constante EXIT_SUCCESS
#include <ctype.h>    // inclusion du fichier ctype.h pour tolower
int main( )
{
    char carLu;
    printf("Saisir une lettre en majuscule : ");
    carLu = getchar();
    carLu = tolower(carLu);
    printf("Le caractère en minuscule %c\n",carLu);

    return EXIT_SUCCESS;
}
```



2.4.3. Création de nouvelles fonctions

Lorsque le programmeur souhaite réaliser une nouvelle fonction, en plus du prototype, il est nécessaire qu'il réalise la **définition** de la fonction. Cette définition regroupe l'ensemble des instructions pour faire le traitement attendu.

Exemple d'appel de fonction

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

int EstUnChiffre(const char _car);   // Prototype de la fonction

int main( )
{
    char carLu;
    int resultat;
    printf("Appuyer sur une touche (puis entrée) : ");
    carLu = getchar();
    resultat = EstUnChiffre(carLu);          //  Appel de la fonction
    if(resultat == 1)
        printf("%c est un chiffre\n",carLu);
    else
        printf("%c n'est pas un chiffre\n",carLu);
    return EXIT_SUCCESS;
}

int EstUnChiffre(const char _car)    // Définition de la fonction
{
    int retour = 0;  // Par défaut on considère que c'est FAUX
    if( _car >= '0' && _car <= '9')
        retour = 1;  // C'est VRAI le caractère est bien un chiffre
    return retour;
}
```

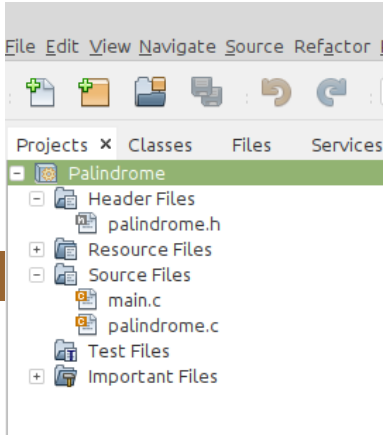
Par souci de qualité de code, la première fonction rencontrée doit être la fonction **main**. Afin de respecter la règle du langage C qui veut que tout élément doive être connu avant son utilisation, les prototypes des fonctions sont placés avant la fonction **main**, les définitions sont placées après.

Par convention, le nom de la fonction est composé d'un verbe, le plus souvent à l'infinitif, suivi d'un complément. La première lettre est une majuscule. La convention de nommage veut que les mots soient joints, une majuscule les distingue. Exemple : **EstUnChiffre**.

Le mot clé **const** indique que le paramètre d'entrée n'est pas modifiable dans la fonction. Dans l'exemple précédent, la variable **_car** n'est jamais affectée. Par convention, le nom des paramètres est précédé du caractère de soulignement pour les distinguer des variables locales.

La portée des variables est limitée à la fonction ou elles sont déclarées, même si elle porte le même nom. Ainsi dans l'exemple, la variable **retour** n'est connue que dans la fonction **EstUnChiffre**. De même, **carLu**, déclarée dans la fonction **main**, n'est pas accessible dans la fonction, d'où le passage de paramètre. Sa valeur est recopiée dans la variable **_car** pour être utilisée dans la fonction avant l'appel.

Il est également possible de regrouper **les prototypes** ou **déclarations** des fonctions dans un fichier d'entêtes « **.h** » et leur définition dans un autre fichier « **.c** » pour rendre plus lisible le programme. Ce dernier fichier doit alors être ajouté au projet pour être compilé.



Généralement, un fichier « **.h** » et son pendant « **.c** » regroupent une série de fonctions traitant d'une même problématique. Dans l'exemple ci-contre toutes les fonctions liées à la reconnaissance d'un palindrome [1] sont regroupées ensemble.

Inclusion d'un fichier .h d'une bibliothèque crée par nos soins

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

#include "palindrome.h"
```

Cette fois-ci, l'inclusion se fait avec le nom du fichier encadré par le caractère **"**.

[1] Mot ou expression pouvant se lire de gauche à droite ou de droite à gauche, aux espaces, accents et majuscules près.

Exemple de programme[2] utilisant des fonctions dans un fichier séparé. Le premier fichier contient la définition des différentes fonctions pour la recherche de palindrome.

Bibliothèque palindrome **palindrome.c**

```c
#include <string.h>
#include <ctype.h>
#include "palindrome.h" // Pour pouvoir utiliser les constantes


/**
 * @brief SupprimerEspace
 * @param _chaine, chaîne de caractères dont on souhaite supprimer les espaces
 * @return nombre d'espaces supprimé
 */
int SupprimerEspace(char _chaine[])
{
    int lecture = 0;
    int ecriture = 0;
    while(_chaine[lecture] != '\0')
    {
        if(_chaine[lecture] != ' ')
        {
            _chaine[ecriture++] = _chaine[lecture];
        }
        lecture++;
    }
    _chaine[ecriture] = '\0';
    return lecture - ecriture;
}

/**
 * @brief ConvertirMajusculeEnMinuscule
 * @param _chaine, chaîne de caractères dont on souhaite convertir les majuscules en minuscules
 */
void ConvertirMajusculeEnMinuscule(char _chaine[])
{
    int indice = 0;
    while(_chaine[indice] != '\0')
    {
        _chaine[indice] = tolower(_chaine[indice]);
        indice++;
    }
}

/**
 * @brief VerifierPalindrome
 * @param _chaine, chaîne de caractère dont on souhaite savoir si c'est un palindrome.
 *                 La chaîne d'origine n'est pas détruite.
 * @return VRAI si c'est un palindrome, FAUX sinon
 */
int VerifierPalindrome(const char _chaine[])
{
    char palindrome[TAILLE_MAX] ;
    int retour = VRAI;
    int debut = 0;
    int fin ;
    strcpy(palindrome,_chaine);
    SupprimerEspace(palindrome);
    ConvertirMajusculeEnMinuscule(palindrome);
    fin = strlen(palindrome) -1 ;
    while(palindrome[debut] == palindrome[fin] && debut <= fin)
    {
        debut++;
        fin--;
    }
    if(palindrome[debut] != palindrome[fin])
        retour = FAUX;
    return retour;
}
```

Dans la fonction **VerifierPalindrome**, il n'est pas nécessaire ici, de savoir combien d'espace ont été supprimés c'est pourquoi on ne récupère pas le paramètre de retour.

[2] Ce programme utilise les tableaux de caractères, si nécessaire, se reporter au chapitre sur les tableaux et les chaînes de caractères dans un premier temps.

Le fichier palindrome.h contient les prototypes des fonctions et la définition des constantes

Bibliothèque palindrome **palindrome.h**

```
#ifndef PALINDROME_H  // Pour éviter d'inclure plusieurs fois le fichier .h (inclusion circulaire)
#define PALINDROME_H
        #define TAILLE_MAX 50
        #define VRAI 1
        #define FAUX 0
        int SupprimerEspace(char _chaine[]);
        void ConvertirMajusculeEnMinuscule(char _chaine[]);
        int VerifierPalindrome(const char _chaine[]);
#endif // PALINDROME_H
```

Le programme principal fait appel à la fonction **VerifierPalindrome** et affiche le résultat en conséquence.

Programme principal **main.c**

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

#include "palindrome.h"


int main()
{
    char unePhrase[TAILLE_MAX];
    int dernierCar;
    printf("Saisir une phrase : ");

    fgets(unePhrase,TAILLE_MAX -1, stdin);
    dernierCar = strlen(unePhrase) - 1;
    if(unePhrase[dernierCar] == '\n')       // Suppression du '\n' éventuel introduit par fgets
        unePhrase[dernierCar] = '\0';


    if(VerifierPalindrome(unePhrase))
        printf("La phrase \"%s\" est un palindrome\n",unePhrase);
    else
        printf("La phrase %s n'est pas un palindrome\n",unePhrase);
    return 0;
}
```

2.4.4. Les paramètres d'une fonction

Les fonctions utilisées en langage C ne peuvent recevoir que des paramètres en entrée, et disposer que d'un seul paramètre de retour.

Passage de paramètres par valeur

C'est le cas courant, lorsque la fonction utilise des valeurs fournies par le programme appelant pour le traitement qui lui est demandé. L'exemple de la fonction **EstUnChiffre**, vue précédemment, en est une illustration, la fonction reçoit un caractère, il est recopié dans **_car** lors de l'appel et utilisé dans la fonction.

Exemple de passage de paramètres par valeur

```
int EstUnChiffre(const char _car)
{
    int retour = 0;  // Par défaut on considère que c'est FAUX
    if( _car >= '0' && _car <= '9')
        retour = 1;  // C'est VRAI, le caractère est bien un chiffre
    return retour;
}
```

Cela fonctionne bien si le programme appelant n'attend pas en retour la ou les valeurs modifiées.

Passage de paramètres par adresse

Si plusieurs données doivent être renvoyées au programme appelant, les fonctions en langage C ne disposant que d'un seul paramètre de retour, il est nécessaire de fournir les adresses des variables destinées à recevoir les valeurs afin que la fonction puisse faire les mises à jour.

Exemple non fonctionnel — **echange_1.c**

```c
#include <stdio.h>
#include <stdlib.h>
void Echanger(int _min, int _max);
int main()
{
    int nb1;
    int nb2;
    printf("Saisir deux nombre : ");
    scanf("%d %d",&nb1,&nb2);
    Echanger(nb1,nb2);
    printf("%d est inférieur ou égale à %d\n", nb1,nb2);
    return EXIT_SUCCESS;
}
void Echanger(int _min, int _max)
{
    int aux;
    if(_max < _min)
    {
        aux = _min;
        _min = _max;
        _max = aux;
    }
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Saisir deux nombres : 5 2
5 est inférieur ou égale à 2
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

L'échange est bien réalisé dans la fonction, si le deuxième nombre est inférieur au premier, l'échange a lieu, mais les valeurs ne sont pas mises à jour dans le programme principal d'où le résultat. Le programme doit être réécrit de la manière suivante :

Exemple fonctionnel — **echange_2.c**

```c
#include <stdio.h>
#include <stdlib.h>
void Echanger(int *_min, int *_max);
int main()
{
    int nb1;
    int nb2;
    printf("Saisir deux nombres : ");
    scanf("%d %d",&nb1,&nb2);
    Echanger(&nb1,&nb2);
    printf("%d est inférieur ou égale à %d\n", nb1,nb2);
    return EXIT_SUCCESS;
}
void Echanger(int *_min, int *_max)
{
    int aux;
    if(*_max < *_min)
    {
        aux = *_min;
        *_min = *_max;
        *_max = aux;
    }
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Saisir deux nombres : 5 2
2 est inférieur ou égale à 5
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Cette fois-ci, le programme fonctionne correctement, l'échange s'est produit dans la fonction et les variables du programme principal sont à jour.

Les variables entre parenthèses `(int *_min ou int *_max)`, dans l'entête de la fonction indiquent que la fonction attend les adresses de deux entiers. Le passage doit se faire en fournissant des adresses et non pas des valeurs.

Lors de l'appel de la fonction **Echanger**, les paramètres utilisent l'opérateur **&** qui fournit l'adresse d'une variable, `Echanger(&nb1,&nb2);` C'est la même chose pour la fonction **scanf**, Elle a besoin également de mettre à jour les variables **nb1** et **nb2** du programme principal.

Dans la fonction `*_max` et `*_min` désignent l'élément contenu aux adresses `_max` et `_min`. Elle travaille directement sur les variables du programme appelant dont elles connaît les adresses en mémoire.

Cas particulier des tableaux

Le nom d'un tableau désignant déjà l'adresse de la première case du tableau, il n'est donc pas nécessaire d'utiliser l'opérateur **&** lors de l'appel de la fonction.

On remarque également en regardant le fichier d'entêtes de la librairie **string.h** ou le manuel d'une de ces fonctions que la notation utilisée est **char *nomDeVariable** au lieu **char nomDeVariable[ ]**. Les deux notations sont équivalentes, toutes deux désignant l'adresse d'un caractère.

Déclaration de la fonction strlen calcul la longueur d'une chaîne contenant des caractères ASCII

```
# include  string h
size_t strlen(const char *s);
```

Le mot clé **const** indique en revanche que la chaîne de caractères désignée par **s** ne peut en aucun cas être modifiée. Cette chaîne est considérée comme un paramètre d'entrée à la fonction.

En revanche dans l'exemple suivant la chaîne **dest** est en sortie et l'autre **src** en entrée.

Déclaration de la fonction strcpy copie la chaîne src dans la chaîne dst

```
# include  string h

char *strcpy(char *dest, const char *src);
```

On remarque l'importance du mot clé **const** ici. La chaîne destination est mise à jour par la fonction alors que la source reste inchangée.

Autre exemple avec l'utilisation de scanf dans deux cas de figure

```
int main
{
    char mot[50];
    int  nombre;

    printf("Saisir un mot :");
    scanf("%s",mot);
    printf("saisir un nombre entier : ");
    scanf("%d",&nombre);

    return EXIT_SUCCESS;
}
```

Dans le premier cas, **scanf** met à jour le tableau **mot** donc il n'y a **pas de &** devant mot lors de l'appel

Dans le second cas, **scanf** met à jour l'entier **nombre**, il est nécessaire d'utiliser l'opérateur **&**.

3. Les Bibliothèques standard du langage C

Pour les systèmes UNIX, elle fait partie du système d'exploitation, et est installée sous forme de paquets avec le compilateur.

Les fichiers d'entêtes décrivant la manière dont les fonctions doivent être utilisées regroupent les prototypes. Ils se trouvent par fonctionnalité dans le répertoire **/usr/include**. Les routines, en langage objet propre au processeur sur lesquelles elles sont exécutées, sont dans le répertoire **/usr/lib** sous le nom de libc.a ou libc.so.

Installation des paquets pour le développement en langage C

```
Dans une console avec les droits administrateur :

apt install build-essential gcc
```

3.1. stdio - Bibliothèque standard de fonctions d'entrées-sorties

La bibliothèque **stdio** fournit une interface simple pour les entrées-sorties. Ces entrées-sorties s'apparentent à des tubes permettant le transfert de données sous la forme de texte. Le terme de **flux de données logiques** est souvent utilisé en programmation pour désigner cette fonctionnalité qui permet de masquer les caractéristiques physiques des entrées-sorties. Généralement, les flux sont associés à des tampons, les données sont transférées lorsque le tampon est plein ou sur un ordre spécifique.

Ainsi, sous Linux, il sera facile de rediriger un flux vers un fichier, une liaison série, une imprimante…

3.1.1. Les flux standards

Lors de l'exécution d'un programme écrit en langage C, trois voies de communication avec l'utilisateur sont naturellement créées et disponibles pour interagir.

- **sdtin** : Représente l'entrée standard de l'application, c'est le flux naturellement associé au clavier.
- **stdout** : Représente la sortie standard de l'application, le flux associé à l'écran
- **sdterr** : Représente la sortie d'erreur de l'application, redirigée également vers l'écran de l'ordinateur

3.1.2. Affichage avec la fonction *printf*

Cette fonction permet d'effectuer l'écriture de texte selon un certain format sur la console. Elle permet de transmettre une simple chaîne de caractères ou plus sophistiquée, une chaîne formatée faisant l'objet d'une ou plusieurs concaténations sur la sortie standard **stdout**.

Prototype de printf : écriture de données formatées **stdio.h**

```
int printf(const char *format, …) ;
documentation : man printf ou http://manpagesfr.free.fr/man/man3/printf.3.html
```

Le paramètre de retour indique le nombre de caractères réellement affichés, un nombre négatif indique une erreur.

Utilisation typique dans un programme ***printf("<format>",<Expr1>,<Expr2>, ...) ;***

<format> représente une chaîne caractères contenant du texte, des séquences d'échappement ou des spécificateurs de format. La chaîne contient autant de spécificateurs commençant toujours par le symbole **%** que d'expressions **<expr1>**, **<expr2>…** Les spécificateurs se terminent par un ou deux caractères indiquant le format d'impression et permettent la conversion de divers types, entier, réel, caractère en texte.

Le caractère d'échappement \ permet d'écrire les symboles utilisés dans le formatage de la chaîne ainsi :

\\	Permet d'afficher \ (antislash)
\n	Permet de faire un passage à la ligne suivante
\"	Permet d'afficher " (double quote)

\t	Permet de faire une tabulation horizontale
\r	Permet de faire un retour en début de ligne
\'	Permet d'afficher ' (simple quote)

Les principaux spécificateurs pour la concaténation des différentes parties de la chaîne et le transtypage des données en texte sont regroupés dans le tableau suivant :

Spécificateur	Information à afficher	Type	Exemples
%c	Caractère	char	À b ? +
%d %x %X	Affichage d'un entier en décimal Affichage d'un entier en hexadécimal en minuscules Affichage d'un entier en hexadécimal en majuscules	int ou char	125 -4578 4fe A15F
%ld %lx %lX	Affichage d'un entier long en décimal Affichage d'un entier long en hexadécimal en minuscules Affichage d'un entier long en hexadécimal en majuscules	long int	-6553589 456789123 afcb235 B4E35FF
%u %u %lu	Affichage d'un entier non signé en décimal Affichage d'un caractère non signé en décimal (octet) Affichage d'un entier long non signé	unsigned int unsigned char unsigned long	65535 254 456987231
%f %e	Affichage d'un réel simple précision	float	virgule flottante : 3.145 exposant : 1.54580e-6
%lf %le	Affichage d'un réel double précision	double	virgule flottante : 3.145 exposant : 1.54380e-6
%s	Affichage d'une chaîne de caractères	char *	Bienvenue !
%%	Affiche le caractère %		

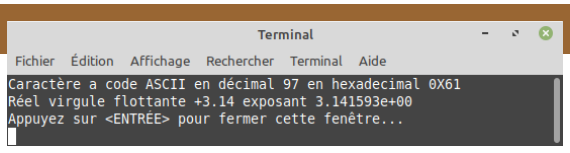
À ces spécificateurs peuvent être ajouté une largeur, une précision suivant la syntaxe :
`%[drapeaux][largeur][.précision]spécificateur`

drapeaux	Description
-	Justifier à gauche dans la largeur de champ donnée, la justification à droite est la valeur par défaut.
+	Oblige à préciser le signe plus ou moins (+ ou -) même pour les nombres positifs. Par défaut, seuls les nombres négatifs sont précédés du - signe .
#	Pour l'affichage hexadécimal, fait précéder la valeur de 0x ou 0X pour affirmer la base 16
0	Rempli à gauche avec des 0 au lieu d'espace sur une largeur donnée

largeur	Description
Nombre entier	Indique le nombre minimum de caractères affichés. Si l'affichage est plus court, l'affichage est complété par des espaces, s'il est trop long, il n'est pas tronqué.

.précision	Description
.Nombre entier	Pour les réels, indique le nombre de chiffres après la virgule Pour les entiers, spécifie le nombre minimum de chiffres à afficher éventuellement complété par des 0 si l'entier est trop court. L'entier n'est pas tronqué s'il est trop long.

Exemple printf **affichage.c**




```c
#include <stdio.h>
#include <math.h>
int main(int argc, char** argv)
{
    printf("Caractère %c code ASCII en décimal %d en hexadécimal %#X\n", 'a','a','a' );
    printf("Réel virgule flottante %+.2f exposant %e\n", M_PI, M_PI);
    return EXIT_SUCCESS;
}
```

3.1.3. Lecture du clavier avec la fonction *scanf*

La fonction *scanf* a pour rôle la lecture de l'entrée standard **stdin** associée le plus souvent au clavier selon un certain format. Les informations lues dans ce tampon seront converties en entier, réel ou caractères suivant le format attendu. Son prototype s'apparente à celui de la fonction *printf* à la différence près que les paramètres désignant les variables sont des paramètres de sortie pour la fonction *scanf* et qu'ils étaient des paramètres d'entrée pour la fonction *printf*.

Prototype de scanf : écriture de données formatées **stdio.h**

```
int scanf(const char *format, ...);
documentation : man scanf ou http://manpagesfr.free.fr/man/man3/scanf.3.html
```

Le paramètre de retour indique le nombre de caractères réellement lus.

Utilisation typique dans un programme ***retour = scanf("<format>",<arg1>,<arg2>, ...) ;***

<format> est construit avec les mêmes spécificateurs que ceux utilisés avec la fonction *printf*.

<arg1>, <arg2>, …. représentent des paramètres de sortie pour la fonction *scanf*. En langage C ce sont donc les adresses des variables recevant les valeurs lues au clavier.

Exemple scanf 1 **lecture1.c**

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

int main()
{
    char var1;
    int var2;
    float var3;
    char chaine1[20];
    printf("Veuillez saisir un caractère : ");
    scanf("%c",&var1);
    printf("Veuillez saisir une valeur entière : ");
    scanf("%d",&var2);
    printf("Veuillez saisir une valeur réelle : ");
    scanf("%f",&var3);
    printf("Veuillez saisir une chaîne de caractères : ");
    scanf("%s",chaine1);
    printf("var1 : %c - var2 : %d - var3 : %f - La chaîne de caractères : %s\n",var1,var2,var3,chaine1);
    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Veuillez saisir un caractère : a
Veuillez saisir une valeur entière : 12
Veuillez saisir une valeur réel : 5.8
Veuillez saisir une chaine de caractères : azerty

var1 : a - var2 : 12 - var3 : 5.800000 - La chaine de caractères : azerty

Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Remarque : Pour les variables simples, caractère, entier, réel, et leurs dérivés, l'opérateur **&** est utilisé pour obtenir l'adresse de la variable. Ce qui n'est pas nécessaire avec un tableau de caractères comme ***chaine1*** dans l'exemple qui représente déjà l'adresse de la première case du tableau.

Il est possible de lire plusieurs valeurs avec la fonction *scanf* et d'imposer un format spécifique :

Exemple scanf 2 **lecture2.c**

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

int main()

{
    int jour;
    int mois;
    int annee;
    printf("Veuillez saisir une date au format jj/mm/aaaa : ");
    scanf("%d/%d/%d",&jour,&mois,&annee);
    printf("Nous sommes le : %d-%d-%d\n",jour,mois,annee);
    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Veuillez saisir une date au format jj/mm/aaaa : 20/10/2021

Nous sommes le : 20-10-2021

Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

La lecture d'un caractère après la lecture d'une autre variable peut parfois poser problème. En effet, la lecture se faisant dans le tampon d'entrée et non pas directement au niveau des touches du clavier, le tampon n'est pas

forcément vide. Il contient encore un retour chariot par exemple qui correspond à l'appui sur la touche entrée pour valider la précédente saisie. L'appel de la fonction *fflush(stdin)* supprime tous les caractères présents dans le tampon d'entrée.

Exemple scanf 3 **lecture3.c**

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


int main()
{
    char var1;
    int var2;
    printf("Veuillez saisir une valeur entière : ");
    scanf("%d",&var2);
    printf("Veuillez saisir un caractère : ");
    scanf("%c",&var1);

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

    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Veuillez saisir une valeur entière : 12
Veuillez saisir un caractère :
var1 :
 - var2 : 12

Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

La saisie du caractère ne s'est pas effectuée, le programme est directement passé à l'affichage… La solution consiste donc d'utiliser la fonction fflush(stdin), mais ce n'est pas forcement suffisant d'où la solution proposée ci-après.

Exemple scanf 4 **lecture4.c**

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

int main()
{
    char var1;
    int var2;
    printf("Veuillez saisir une valeur entière : ");
    scanf("%d",&var2);
    fflush(stdin);
    printf("Veuillez saisir un caractère : ");

    while(scanf("%c",&var1) != 1 || var1 == '\n' )
        fflush(stdin);


    printf("var1 : %c - var2 : %d \n",var1,var2);
    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Veuillez saisir une valeur entière : 12
Veuillez saisir un caractère : f
var1 : f - var2 : 12
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

La saisie du caractère est réalisée tant qu'un caractère n'a pas été lu et tant que c'est le caractère **'\n'**.

Exemple scanf 5 **lecture5.c**

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

int main()
{
    int var;
    char var1;
    printf("Veuillez saisir une valeur entière : ");
    fflush(stdin);
    while(scanf("%d",&var) != 1)
    {
        while(scanf("%c",&var1) != 1 || var1 == '\n' );
        printf("Valeur incorrecte : Veuillez saisir une valeur entière : ");
    }
    printf("var : %d \n",var);
    return EXIT_SUCCESS;
}
```

Dans cet exemple, on s'assure que la saisie est bien un entier, sinon on redemande la valeur.
Il est nécessaire de gérer '\n' qui est resté dans le tampon.

4. Les structures de données

4.1. Les tableaux

Un tableau est une structure de données en mémoire consécutive contenant un nombre défini d'éléments de même type. Chaque élément est accessible à partir du nom du tableau et d'un indice désignant sa position dans ce tableau. L'indice du tableau commence à 0. Les tableaux ont pour avantage de permettre un traitement d'un ensemble de données en utilisant des itérations, boucles définies pour un traitement sur l'ensemble du tableau, boucles indéfinies pour la recherche d'élément.

Lors de sa déclaration, comme une variable simple, un tableau doit être déclaré. En plus du type des données qu'il contient, il faut préciser sa taille entre crochets **[ ]**.

indice	valeurs
0	10
1	23
2	5
3	4
4	20

Exemple de déclaration avec une initialisation

```c
#include <stdio.h>
#include <stdlib.h>
#define NB_ELEMENTS 5
int main()
{
    int valeurs[NB_ELEMENTS] = {10,23,5,4,20} ;
    printf("La valeur contenue dans le tableau à la 3ème case est : ");
    printf("%d\n",valeurs[2]);

    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
La valeur contenue dans le tableau à la 3ème case est : 5
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

L'accès à chaque élément peut se faire en indiquant le numéro de la case entre crochets comme le montre l'exemple précédent. Attention, la numérotation commence à 0, ou par un indice qui évolue dans une boucle comme le montre l'exemple suivant. Attention de ne pas dépasser les bornes du tableau au risque d'écraser les valeurs des variables adjacentes.

Initialisation d'un tableau à l'aide d'une boucle — **tableau.c**

```c
#include <stdio.h>
#include <stdlib.h>
#define NB_NOTES 24
int main()
{
    double notes[NB_NOTES];
    int indice;
    for(indice=0 ; indice < NB_NOTES ; indice++)
    {
        notes[indice] = 0;
    }
    return EXIT_SUCCESS;
}
```

Les tableaux peuvent être déclarés et initialisés par la suite dans le programme.Dans ce cas, le tableau contient des valeurs indéterminées. Au besoin, il est nécessaire de les initialiser avec la valeur 0 par exemple.

L'indice de la boucle évolue bien ici entre 0 et 23, les bornes du tableau.

La copie d'un tableau vers un autre ne peut pas se faire en utilisant l'instruction **tab1 = tab2**. En effet, les variables **tab1** et **tab2** représentent les adresses respectives de la première case de chaque tableau. Il est nécessaire d'utiliser une boucle pour recopier individuellement chaque case.

Initialisation d'un tableau à l'aide d'une boucle — **tableau.c**

```c
#include <stdio.h>
#include <stdlib.h>
#define NB_ELEMENTS 10
int main()
{
    int tab1[] = {10,4,8,9,3,54,76,23,6,15};
    int tab2[NB_ELEMENTS];
    int indice;
    for(indice = 0 ; indice < NB_ELEMENTS ; indice++)
        tab2[indice] = tab1[indice] ;

    // memcpy(tab2, tab1, NB_ELEMENTS);
    return EXIT_SUCCESS;
}
```

L'instruction :

*void ****memcpy****(void *dest, const void *src, size_t n);* de la librairie **string.h** réalise le même traitement.

Le premier paramètre représente le tableau destination, le second le tableau source enfin le troisième paramètre indique le nombre d'éléments à copier.

En commentaire l'appel de la fonction pour le cas présent en remplacement de la boucle.

4.2. Les chaînes de caractères

Une chaîne de caractères est un tableau de caractères à une dimension avec une particularité, en plus des lettres qui composent la chaîne, un caractère spécial réalise un marqueur de fin. Ce marqueur correspond au code **ASCII 0**, le caractère nul également noté '\0'. Une chaîne de caractères peut contenir des mots, des phrases, elle permet tout type de traitement sur des données textuelles.

Manipulation de chaînes **chaîne.c**

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // Définition de 3 chaînes de caractères
    char nom[20];
    char prenom[20];
    char ligne[80];

    printf( "Entrez votre nom et votre prénom : ") ;
    scanf("%s%s", nom, prenom); // Saisie de 2 chaînes au clavier (espace séparateur)

    // fabrication d'une nouvelle chaîne en mettant bout à bout prénom et nom
    sprintf( ligne,"%s %s", prenom, nom ) ;
    printf("\nBonjour, %s !\n", ligne ); // Affichage
    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Recherche  Terminal  Aide
Entrez votre nom et votre prenom : Cruchet Philippe

Bonjour, Philippe Cruchet !
Press <RETURN> to close this window...
```

Remarque

Les fonctions manipulant les chaînes de caractères ajoutent automatiquement le caractère '\0' à la fin du tableau. En revanche, si les caractères sont mis un par un dans un tableau de caractères, il est nécessaire d'ajouter le marqueur de fin de chaîne manuellement.

4.2.1. Déclaration et initialisation d'une chaîne de caractères

Lorsque la chaîne n'est pas amenée à être modifiée pendant l'exécution du programme la déclaration peut être réalisée de la manière suivante :

indice	***texte***
0	's'
1	'a'
2	'l'
3	'u'
4	't'
5	'\0'

Exemple de déclaration avec une initialisation

```c
char texte[] = "salut";

char texte2[] = { 's','a','l','u','t', 0 }; // pas judicieux
```

Pour la déclaration de ***texte***, le compilateur ajoute le code ASCII 0 à la fin du tableau et le dimensionne automatiquement pour recevoir 6 caractères, les 5 lettres du mot et le caractère '\0'.

La déclaration de **texte2** est équivalente à la première, mais le caractère '\0' n'est pas mis automatiquement dans le tableau.

Attention

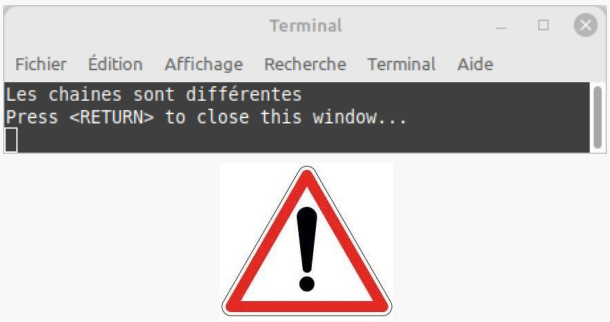
Il est fortement déconseillé de modifier une chaîne de caractères déclarée et initialisée de la sorte lors de l'exécution du programme sous peine d'avoir un débordement de tableau.

4.2.2. Fonctions de traitements de chaînes de caractères

Pour effectuer la comparaison de deux chaînes de caractères, il est absolument nécessaire d'utiliser les fonctions de la librairie string.h. En effet, le test d'égalité entre deux chaînes par exemple ne fait que comparer les adresses des tableaux de caractères et non pas le contenu des deux tableaux.

Comparaison de deux chaînes de caractère **compare1.c**

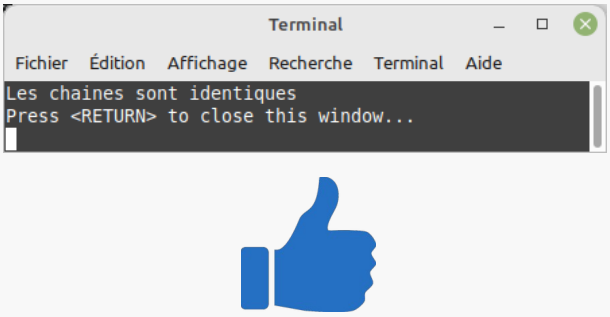
```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char texte1[] = "salut";
    char texte2[] = "salut";

    if(texte1 == texte2)
        printf("Les chaînes sont identiques \n");
    else
        printf("Les chaînes sont différentes \n");

    return EXIT_SUCCESS;
}
```



Il est nécessaire d'utiliser la fonction strcmp.

Comparaison de deux chaînes de caractère **compare2.c**

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char texte1[] = "salut";
    char texte2[] = "salut";

    if(strcmp(texte1,texte2) == 0)
        printf("Les chaînes sont identiques \n");
    else
        printf("Les chaînes sont différentes \n");

    return EXIT_SUCCESS;
}
```



Les principales fonctions de la librairie sont présentées dans le tableau ci-dessous

Nom	Description
strlen	Cette fonction permet de calculer la longueur de la chaîne de caractères.
strcpy	Cette fonction permet de copier une chaîne de caractères.
strcmp	Cette fonction permet de comparer deux chaînes de caractères et de savoir si la première est inférieure, égale ou supérieure à la seconde.
strchr	Cette fonction recherche la première occurrence d'un caractère dans une chaîne de caractères.
strstr	Cette fonction recherche la première occurrence d'une sous-chaîne dans une chaîne de caractères principale.
strcat	Cette fonction ajoute une chaîne de caractères à la suite d'une autre chaîne.

4.3. Les structures

Les chapitres précédents ont montré que le tableau permet de regrouper sous une même entité un ensemble de données de même type accessible grâce à un indice. Une structure va permettre de regrouper un ensemble d'informations de nature différente pour un élément donné et ainsi structurer les données d'un programme. Une structure est matérialisée par une variable contenant plusieurs champs désignés par un nom, chaque nom représente une variable de type éventuellement différent.

Exemple de structure

```
struct personne
{
    char nom[50];
    char prénom[50];
    int score;
    char niveau;
};
```

Le mot clé ***struct*** déclare une structure définie par les accolades, ***personne*** est une étiquette non obligatoire. Les variables déclarées entre accolades ***nom***, ***prenom***, ***score*** et ***niveau*** sont les membres de la structure, ils en représentent les différents champs. Généralement, la définition des structures est réalisée dans un fichier d'entête **.h**.

Déclaration d'une variable de type struct personne

```
struct personne joueur ;
```

La variable ***joueur*** représente une variable du type structure personne. Il faut pour cela que la structure ***personne*** soit définie au préalable. Les variables de type structure sont déclarées en fonction de leur utilisation dans le fichier source **.c**.

Autre manière de faire

```
struct
{
    char nom[50];
    char prenom[50];
    int score;
    char niveau;
} joueur ;
```

Dans l'exemple précédent, le nom personne a été volontairement omis. Il y a une variable ***joueur*** qui contient les quatre champs, ***nom*** ,***prenom***, ***score*** et ***niveau***. Dans ce cas de figure, le contenu de la structure doit être répété pour chaque nouvelle variable de ce type.

Pour simplifier l'écriture, le langage C permet de définir de nouveaux types. Par convention, le nom du nouveau est précédé de la lettre T suivie d'un souligné, le nom est écrit en majuscules.

Définition de type **structure.h**

```
// Dans le fichier .h
typedef struct
{
    char nom[50];
    char prénom[50];
    int score;
    char niveau;
} T_PERSONNE;

// Dans le code fichier.c déclaration d'une variable
T_PERSONNE joueur;
```

Le mot clé **typedef** permet de définir un nouveau type, qui peut être utilisé simplement dans le code par la suite.

4.3.1. Utilisation des structures dans un programme

Chaque champ d'une structure est accessible individuellement, ce sont des variables comme les autres, ainsi pour les chaînes de caractères il est nécessaire d'utiliser la fonction **strcpy** pour l'affectation ou une boucle pour recopier un tableau.

Utilisation de la structure **structure.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "structure.h"

int main()
{
    T_PERSONNE joueur1;

    char unPrenom[]={"Robert"};

    printf("Entrez le nom du joueur : ");
    scanf("%s",joueur1.nom);
    strcpy(joueur1.prenom,unPrenom);
    joueur1.score = 0;
    printf("Entrez la lettre correspondant au niveau du joueur : ");

    do // Pour ne pas prendre le caractère '\n' du dernier scanf
    {
        scanf("%c",&joueur1.niveau);
    }while(joueur1.niveau == '\n');

    printf("\n%s %s\n",joueur1.prenom,joueur1.nom);
    printf("Score : %d au niveau : %c\n\n",joueur1.score,joueur1.niveau);

    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Entrez le nom du joueur : Bidochon
Entrez la lettre correspondant au niveau du joueur : A

Robert Bidochon
Score : 0 au niveau : A

Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

La structure est une variable, l'accès à ses champs se fait en utilisant un point entre la variable et le nom du champ.

4.3.2. Passage d'une structure en paramètre à une fonction

Contenu de la taille mémoire d'une structure, elle est composée de plusieurs variables éventuellement de tableaux, il est judicieux de passer une structure par adresse. En effet dans ce cas seul un pointeur est transmis. Si cette structure est juste utilisée en lecture, le mot clé const peut précéder la déclaration du paramètre.

Passage en paramètre d'une structure **structure2.c**

```c
#include <stdio.h>
#include "structure.h"
void CreerNouveauJoueur(T_PERSONNE *_joueur)
{
    printf("Entrez le nom du joueur : ");
    scanf("%s",_joueur->nom);

    printf("Entrez le prénom du joueur : ");
    scanf("%s",_joueur->prenom);

    _joueur->score= 0;
    _joueur->niveau = 'A';
}
void AfficherScore(const T_PERSONNE *_joueur)
{
    printf("\n%s %s Score %d au niveau %c\n\n",_joueur->nom,_joueur->prenom,
           _joueur->score, _joueur->niveau);
}
```

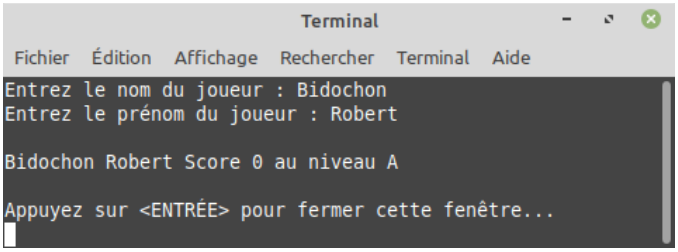
Cette fois-ci, l'accès aux champs de la structure ce fait par les signes – et > « **->** » et non pas avec un point comme il était d'usage dans l'exemple précédent.

Pour l'appel des fonctions dans le programme principal, il est nécessaire de fournir l'adresse de la variable déclarée dans le programme principal.

L'exemple suivant montre l'appel des fonctions :

Appel des fonctions en passant l'adresse de la variable **structure3.c**

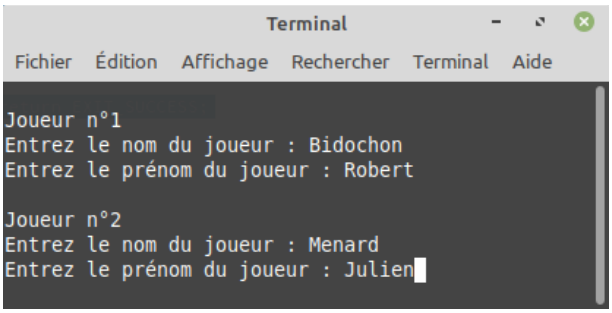
```c
#include <stdio.h>
#include <stdlib.h>

#include "structure.h"

int main()
{
    T_PERSONNE joueur1;

    CreerNouveauJoueur(&joueur1);
    AfficherScore(&joueur1);

    return EXIT_SUCCESS;
}
```



Il est également tout à fait possible de déclarer un tableau de structures si nécessaire.

Appel des fonctions en passant l'adresse de la variable **structure4.c**

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

#include "structure.h"

#define NB_JOUEURS 10

int main()
{
    T_PERSONNE lesJoueurs[NB_JOUEURS];
    int indice;
    for (indice = 0; indice < NB_JOUEURS; indice++ )
    {
        printf("\nJoueur n°%d\n",indice+1);
        CreerNouveauJoueur(&lesJoueurs[indice]);
    }
    printf("\nNon du premier joueur : %s\n ",lesJoueurs[0].nom);
    return EXIT_SUCCESS;
}
```



Attention, chaque case du tableau contient une structure, il est donc nécessaire de fournir l'adresse de cette structure lors de l'appel de la fonction en utilisant l'opérateur **&** devant la variable passée en paramètre. Pour une utilisation de la variable directement, comme pour afficher le nom du premier joueur, l'utilisation du point est de nouveau en vigueur.

4.4. Les fichiers

4.4.1. Introduction

De nombreuses applications doivent conserver les informations traitées bien au-delà de la durée d'exécution du programme de traitement. Ces informations doivent être conservées en dehors de la mémoire centrale de l'ordinateur, sur un support permanent.

Par ailleurs, le volume de certaines données interdit de les représenter ensemble dans l'espace disponible en mémoire centrale. Ces données doivent être fractionnées en unités plus petites pour être traitées.

Dans un premier temps, seules les fonctions de haut niveau de la librairie stdio sont abordées.

4.4.2. Définition

Un fichier est une collection d'informations, structuré en unités d'accès appelées **enregistrement**. Ils sont tous de même type, en général composé de plusieurs champs.

4.4.3. Association d'un fichier à un programme

Un fichier est identifié de manière unique sur le support externe à la mémoire centrale par un **nom physique**, c'est le nom du fichier sur le disque. Chaque programme utilisant ce fichier doit associer le **nom physique** à une **entité logique** pour pouvoir l'utiliser, c'est le **descripteur de fichier**.

Cette association est effectuée par l'action d'**ouverture** du fichier. Cette action assure également au programme le contrôle du fichier (un fichier peut être utilisé en même temps par plusieurs programmes de traitement, le partage doit être contrôlé pour éviter les incidents). Enfin, l'ouverture du fichier place le dispositif de lecture/écriture en position initiale sur le support de fichier. Un fichier peut être ouvert suivant différents modes.

Prototype de fopen **Ouverture d'un fichier**

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode);
```

FILE est une structure gérée par le système pour la manipulation, le programmeur a juste à déclarer un pointeur sur cette structure.

La fonction fopen() ouvre le fichier dont le nom est contenu dans la chaîne pointée par le premier paramètre **path** et lui associe un flux.

Le paramètre **mode** pointe vers une chaîne commençant par l'une des séquences suivantes : "r" , "r+", "w", "w+", "a" ou "a+". Le fichier est dans ce cas un fichier texte. Les caractères contenus dans le fichier sont lisibles avec un éditeur de texte. Par défaut, le système manipule des **fichiers texte**.

r	Ouverture d'un fichier texte en lecture. Le pointeur de flux est placé au début du fichier.
r+	Ouverture d'un fichier texte en lecture et écriture. Le pointeur de flux est placé au début du fichier.
w	Ouverture d'un fichier texte en écriture, si le fichier existait il est écrasé. Le pointeur de flux est placé au début du fichier.
w+	Ouverture d'un fichier texte en lecture et écriture. Le fichier est créé s'il n'existait pas. S'il existait déjà, sa longueur est remise à 0. Le pointeur de flux est placé au début du fichier.
a	Ouverture d'un fichier texte en ajout (écriture à la fin du fichier). Le fichier est créé s'il n'existait pas. Le pointeur de flux est placé à la fin du fichier
a+	Ouverture d'un fichier texte en ajout (écriture à la fin du fichier). Le fichier est créé s'il n'existait pas. Le pointeur de flux est placé à la fin du fichier.

La chaîne représentant le **mode** peut être suivie de la lettre **b** pour désigné l'ouverture d'un **fichier binaire**, l'information est représentée dans ce cas comme dans la mémoire de l'ordinateur, sous forme d'entiers, de réels… Ils ne sont pas directement lisibles par un éditeur de texte.

Ouverture d'un fichier **fichier.c**

```c
#include <stdio.h>
int main()
{
    FILE *pFichTexte ;
    FILE *pFichBinaire ;

    pFichTexte = fopen("NonDuFichier","r");
    // Ouvre un fichier texte en lecture. Le pointeur de flux est placé au début du fichier.
    if( pFichTexte == NULL)
        exit(errno); // Quitte le programme avec le code d'erreur correspondant


    pFichBinaire = fopen("NonDuFichier","wb");
    // Ouvre un fichier binaire en écriture, si le fichier existait il est écrasé.
    // Le pointeur de flux est placé au début du fichier.
    if( pFichTexte == NULL)
        exit(errno);
    // Suite du programme
}
```

Il est nécessaire de toujours tester l'ouverture d'un fichier, pour ne pas poursuivre de manière erronée la suite du programme, une erreur peut se produire, si en lecture le fichier n'existe pas, si en écriture le support est plein… La solution proposée ici est radicale, des manières moins violentes peuvent être proposées à l'utilisateur comme choisir un nouveau nom de fichier, un nouveau support...

À la fin du traitement sur un fichier, le programme indique qu'il n'en a plus besoin en effectuant sa **fermeture**. Ceci permet de rendre le contrôle du fichier pour les autres programmes, de terminer le transfert des informations dans le cas d'une écriture et éventuellement d'ajouter la marque de fin de fichier.

Prototype de fclose — **fermeture d'un fichier**

```c
#include <stdio.h>
int fclose(FILE *fp);
```

La fonction vide le flux pointé par **fp**, en écrivant toute donnée de sortie en tampon et ferme le descripteur de fichier sous-jacent. Si la fonction réussit intégralement, elle renvoie 0, sinon elle renvoie **EOF** et errno contient le code d'erreur. Dans tous les cas, tout autre accès ultérieur au flux, y compris un autre appel de **fclose()**, conduit à un comportement indéfini. Elle renvoie **EBADF**. Le descripteur de fichier sous-jacent **fp** est invalide.

Fermeture d'un fichier — **fichier.c**

```c
#include <stdio.h>
int main()
{
    FILE *pFichTexte ;
    FILE *pFichBinaire ;

    // Fin du programme
    fclose(pFichTexte);
    fclose(pFichBinaire);

    return EXIT_SUCCESS;
}
```

Le test de la valeur de retour est facultatif, mais permet de s'informer pour savoir si l'opération s'est bien déroulée.

4.4.4. Traitement d'un fichier ouvert en écriture

Deux situations sont à distinguer en fonction du type de fichier. Pour un fichier texte, il existe plusieurs manières d'écrire dans le fichier soit caractère par caractère, soit par ligne ou suivant un format spécifique :

Par caractère	**int fputc(int c, FILE *stream);** Écrit le caractère **c**, transformé en unsigned char, dans le flux **stream**. La fonction renvoie le caractère écrit en tant qu'un caractère non signé (converti en int), ou **EOF** en cas d'erreur.
Par chaîne	**int fputs(const char *s, FILE *stream);** Écrit la chaîne de caractères ***s*** dans le flux **stream**, sans écrire l'octet nul « \0 » final. La fonction renvoie un nombre positif, si elle réussit et **EOF** si elle échoue.
Suivant un format spécifique	**int fprintf(FILE *stream, const char *format, ...);** La fonction, tout comme printf() le fait sur la sortie standard, écrit des sorties sous forme de texte en accord avec le **format** sur le flux stream **indiqué**.

Pour les fichiers binaires, l'écriture peut se faire octet par octet, mot par mot ou bloc par bloc, une seule fonction permet de réaliser cette opération. Il est nécessaire de préciser la taille du ou des éléments à écrire.

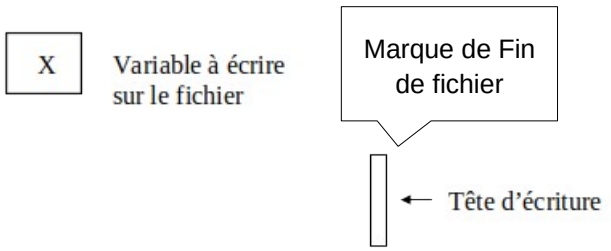
Lecture d'un fichier binaire	**size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);** Écrit **nmemb** éléments de données, chacun d'eux représentant **size** octet de long, dans le flux pointé par **stream**, après les avoir récupérés depuis l'emplacement pointé par ptr. **void *** représentant ici un pointeur générique s'adaptant à tous types de données. En cas de réussite, la fonction renvoie le nombre d'éléments écrits. Ce nombre n'est égal au nombre d'octets transférés que si **size** est 1. Si une erreur se produit, le nombre renvoyé est plus petit que nmemb et peut même être nul.

C'est la technique est utilisée pour écrire une structure dans un fichier.

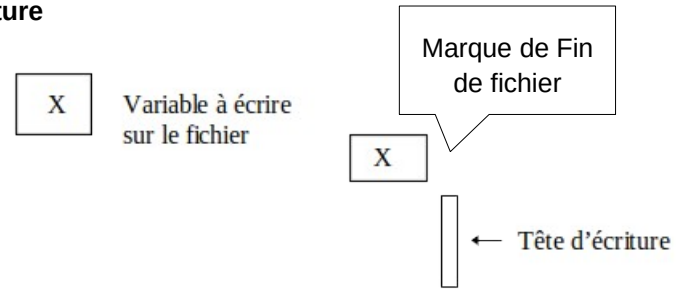
Principe de l'écriture

Traiter un fichier en écriture, c'est avant tout créer ce fichier. En effet, avant toute opération d'écriture le fichier doit être ouvert en mode ECRITURE ce qui a pour conséquence de placer la « tête d'écriture » en position initiale et d'écraser le précédent contenu. L'écriture entraîne le déplacement automatique du dispositif d'écriture après chaque ajout.

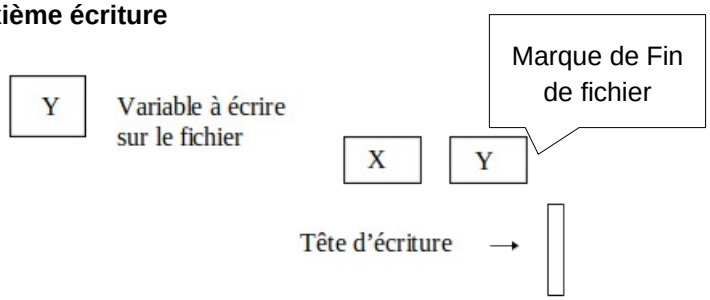
- **Immédiatement après l'ouverture du fichier**



- **Après une écriture**



- **Après une deuxième écriture**



Les exemples suivants utilisent les structures et les fonctions du chapitre précédent.

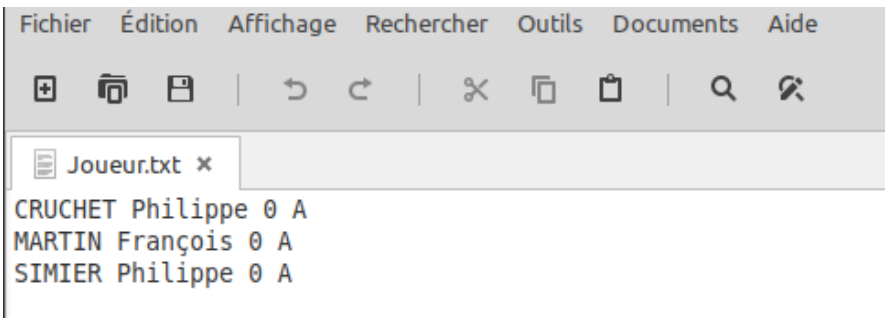
Exemple d'écriture d'un fichier texte **fichier2.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "structure.h"
#define NB_JOUEURS 3
int main()
{
    FILE *pFich ;
    T_PERSONNE lesJoueurs[NB_JOUEURS];

    int indice;
    size_t nbEcrits;
    for (indice = 0; indice < NB_JOUEURS; indice++)
    {
        printf("\nJoueur n°%d\n",indice+1);
        CreerNouveauJoueur(&lesJoueurs[indice]);
    }

    pFich = fopen("Joueur.txt","w");
    for (indice = 0; indice < NB_JOUEURS; indice++ )
    {
       nbEcrits =fprintf(pFich,"%s %s %d %c\n", lesJoueurs[indice].nom, lesJoueurs[indice].prenom,
                         lesJoueurs[indice].score,lesJoueurs[indice].niveau);
        if(nbEcrits != 4)
            exit(errno);       //  L'écriture n'a pas pu se faire correctement, on quitte le programme
    }
    fclose(pFich);

    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide

Joueur n°1
Entrez le nom du joueur : CRUCHET
Entrez le prénom du joueur : Philippe

Joueur n°2
Entrez le nom du joueur : MARTIN
Entrez le prénom du joueur : François

Joueur n°3
Entrez le nom du joueur : SIMIER
Entrez le prénom du joueur : Philippe
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Si l'écriture n'est pas réussie, le fichier risque d'être corrompu, le programme s'arrête. D'autres solutions moins radicales peuvent également être prévues en informant l'utilisateur du problème survenu par exemple.

Voici le résultat obtenu dans le fichier Joueur.txt, ce fichier se trouve dans le même répertoire que l'exécutable.



Voici le même exemple en utilisant un fichier binaire

Exemple d'écriture d'un fichier binaire **fichier3.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "structure.h"
#define NB_JOUEURS 3
int main()
{
    FILE *pFich ;
    T_PERSONNE lesJoueurs[NB_JOUEURS];

    int indice;
    size_t nbEnreg;
    for (indice = 0; indice < NB_JOUEURS; indice++)
    {
        printf("\nJoueur n°%d\n",indice+1);
        CreerNouveauJoueur(&lesJoueurs[indice]);
    }
    pFich = fopen("Joueur.bin","wb");
    nbOctets =fwrite(lesJoueurs, sizeof (T_PERSONNE), NB_JOUEURS,pFich);
    if(nbEnreg != NB_JOUEURS)
           exit(errno);       //  L'écriture n'a pas pu se faire correctement, on quitte le programme
    fclose(pFich);
    return EXIT_SUCCESS;
}
```

```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide

Joueur n°1
Entrez le nom du joueur : CRUCHET
Entrez le prénom du joueur : Philippe

Joueur n°2
Entrez le nom du joueur : MARTIN
Entrez le prénom du joueur : François

Joueur n°3
Entrez le nom du joueur : SIMIER
Entrez le prénom du joueur : Philippe
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Dans ce cas, le fichier **Joueur.bin** ne peut pas être visualisé avec un éditeur de texte. L'écriture a pu se faire en une seule ligne en précisant le nombre de structures et leur taille à écrire dans le fichier.

Remarque

L'écriture n'est pas directement réalisée sur le support disque, clé USB… mais dans un tampon qui est vidé lorsqu'il est plein ou lors de l'exécution de l'instruction **fclose()**. C'est aussi pour cela qu'il est important de **démonter** une clé USB pour la libérer et non pas la retirer brutalement afin que les opérations d'écriture aient bien le temps de se faire complètement.

4.4.5. Traitement d'un fichier ouvert en ajout

Le fichier doit être ouvert en AJOUT, ce qui correspond à l'ouverture de celui-ci en écriture cependant la tête d'écriture se place juste derrière le dernier élément du fichier. Le contenu précédent n'est pas détruit, les nouvelles données sont enregistrées à la suite en fin de fichier.

4.4.6. Traitement d'un fichier ouvert en lecture

Par analogie, il existe également plusieurs manières de lire un fichier texte soit caractère par caractère, par ligne ou suivant un format spécifique pour un fichier texte :

Par caractère	**int fgetc(FILE *stream);** Lit un caractère depuis le flux stream et le renvoie sous forme d'un unsigned char, transformé en int, ou **EOF** en cas d'erreur ou de fin de fichier.
Par chaîne	**char *fgets(char *s, int size, FILE *stream);** Lit au plus **size - 1** caractères depuis stream et les places dans le tampon pointé par **s**. La lecture s'arrête après **EOF** ou un retour chariot. Si un retour chariot est lu, il est placé dans le tampon. Un octet nul (« **\0** ») final est placé à la fin de la ligne.
Suivant un format spécifique	**int fscanf(FILE *stream, const char *format, ...);** Lit ses entrées depuis le flux pointé par stream, à la manière d'un scanf depuis le clavier. Dans ce cas les données sont organisées et toujours dans le même ordre. Elles sont écrites sous forme de texte lisible avec un éditeur de texte.

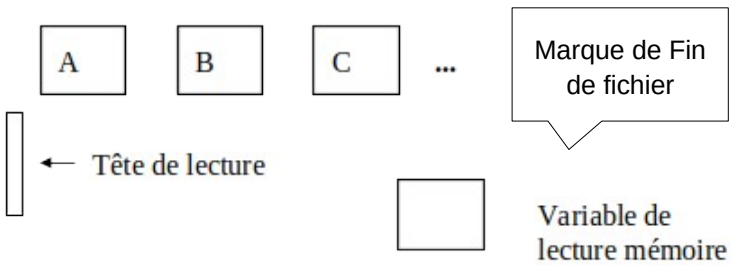
Pour les fichiers binaires, la lecture peut se faire également octet par octet, mot par mot ou bloc par bloc, une seule fonction permet de réaliser cette lecture. Il est nécessaire de préciser la taille de l'élément qui doit être lu et la variable le recevant dans la mémoire de l'ordinateur doit être correctement dimensionnée.

Lecture d'un fichier binaire	**size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);** Lit **nmemb** éléments de données, chacun d'eux représentant **size** octets de long, depuis le flux pointé par **stream**, et les stocke à l'emplacement pointé par **ptr**. **void *** représentant ici un pointeur générique s'adaptant à tous types de données.

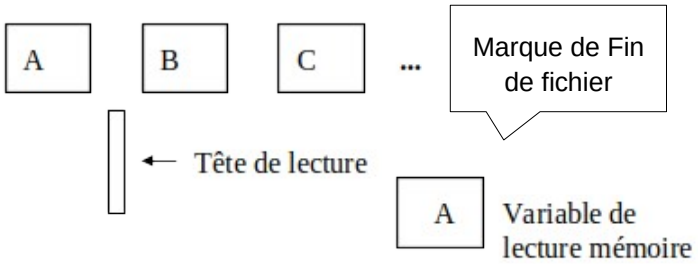
Principe de la lecture

Une « tête de lecture » est associée au fichier ouvert en mode LECTURE et pour toute opération de lecture, il y a un déplacement relatif automatique de la tête de lecture par rapport au fichier.

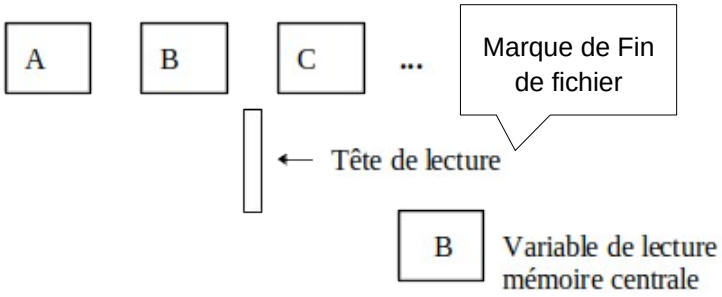
- **Immédiatement après l'ouverture du fichier**



- **Après une lecture**



- **Après une deuxième lecture**



Important

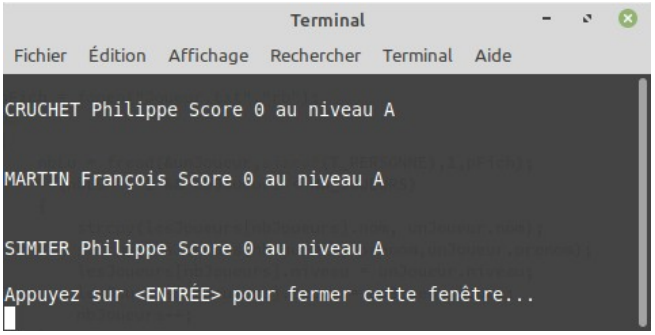
Chaque opération de lecture demande une vérification avant traitement de la donnée. En effet, la lecture n'a pas pu être réalisée si la tête de lecture est à la fin du fichier ou si le nombre d'octets demandé n'a pas été lu complètement.

Exemple de lecture d'un fichier texte **fichier4.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "structure.h"
#define NB_JOUEURS 3
int main()
{
    FILE *pFich ;
    T_PERSONNE lesJoueurs[NB_JOUEURS];
    T_PERSONNE unJoueur;
    int nbJoueurs=0;
    int indice;
    size_t nbLu;

    pFich = fopen("Joueur.txt","r");
    for (indice = 0; indice < NB_JOUEURS && !feof(pFich); indice++ )
    {
        nbLu =fscanf(pFich,"%s %s %d %c\n",unJoueur.nom, unJoueur.prenom, &unJoueur.score,
                    &unJoueur.niveau);
        if(nbLu == 4) // Il y a 4 valeurs à lire à la fois
        {
            strcpy(lesJoueurs[indice].nom, unJoueur.nom);
            strcpy(lesJoueurs[indice].prénom,unJoueur.prénom);
            lesJoueurs[indice].niveau = unJoueur.niveau;
            lesJoueurs[indice].score = unJoueur.score;
        }
    }
    fclose(pFich);
    for (indice = 0; indice < NB_JOUEURS; indice++ )
    {
        AfficherScore(&lesJoueurs[indice]);
    }

    return EXIT_SUCCESS;
}
```

Le fichier est lu ligne par ligne avec un fscanf pour récupérer chaque champ de la structure.

La fonction feof() vérifie si la fin de fichier est atteinte.



Exemple de lecture d'un fichier binaire **fichier5.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "structure.h"
#define NB_JOUEURS 3
int main()
{
    FILE *pFich ;
    T_PERSONNE lesJoueurs[NB_JOUEURS];
    T_PERSONNE unJoueur;
    int nbJoueurs=0;
    int indice;
    size_t nbLu;
    pFich = fopen("Joueur.bin","rb");
    do
    {
        nbLu = fread(&unJoueur,sizeof(T_PERSONNE),1,pFich);
        if(nbLu == 1 && nbJoueurs < NB_JOUEURS)
        {
            strcpy(lesJoueurs[nbJoueurs].nom, unJoueur.nom);
            strcpy(lesJoueurs[nbJoueurs].prénom,unJoueur.prénom);
            lesJoueurs[nbJoueurs].niveau = unJoueur.niveau;
            lesJoueurs[nbJoueurs].score = unJoueur.score;
            nbJoueurs++;
        }
    } while(!feof(pFich));
    fclose(pFich);
    for (indice = 0; indice < NB_JOUEURS; indice++ )
    {
        AfficherScore(&lesJoueurs[indice]);
    }
    return EXIT_SUCCESS;
}
```

Ici, on ne sait pas combien il y a de structures contenues dans le fichier. On les lit jusqu'à ce que la fin de fichier soit atteinte en vérifiant que le tableau de structures n'a pas dépassé sa capacité.

4.4.7. Combinaison des différents modes

Les modes d'ouverture en écriture, en lecture ou même en ajout peuvent être combinés entre eux pour accéder en lecture et écriture au fichier. Les enregistrements sont accessibles directement par une fonction de déplacement spécifique.

Déplacement direct dans le fichier	**int fseek(FILE *stream, long offset, int whence);** La fonction définit l'indicateur de position du flux pointé par **stream**. La nouvelle position, mesurée en octets, est obtenue en additionnant **offset** octets au point de départ indiqué par **whence**. Si **whence** vaut **SEEK_SET**, **SEEK_CUR**, ou **SEEK_END**, le point de départ correspond respectivement au début du fichier, à la position actuelle, ou à la fin du fichier. Un appel réussi à fseek() efface l'indicateur de fin de fichier du flux. Si l'opération est réussie totalement, la fonction renvoie 0, sinon, elles renvoient -1 et la variable globale errno contient le code d'erreur. : • EBADF Si le flux stream n'est pas un flux positionnable. • EINVAL Si l'argument whence n'était ni SEEK_SET, ni SEEK_END, ni SEEK_CUR. Autrement, le décalage du fichier aurait été négatif.
Retour au début du fichier	**void rewind(FILE *stream);** La fonction place l'indicateur de position du flux pointé par stream au début du fichier. C'est l'équivalent de : ***fseek(stream, 0L, SEEK_SET) ;***