

# Langage "C" Raisonné à l'usage des Programmeurs

*Version du 2 mai 2025*

<i>Version</i>	<i>Auteur</i>	<i>Commentaires</i>
<i>21 octobre 2017</i>	<i>Emile Geahchan</i>	<i>Version Initiale</i>
<i>20 février 2018</i>	<i>Emile Geahchan</i>	<i>La Gestion des Fichiers</i>
<i>10 octobre 2019</i>	<i>Emile Geahchan</i>	<i>La Gestion de la Mémoire</i>
<i>14 mars 2020</i>	<i>Emile Geahchan</i>	<i>+ Exercices</i>
<i>2 janvier 2022</i>	<i>Emile Geahchan</i>	<i>Transtypage</i>
<i>11 avril 2025</i>	<i>Emile Geahchan</i>	<i>Compléments exercices</i>

*Tous droits réservés.*

*Ce document est un support de cours à l'usage exclusif des auditeurs du Cnam dans le cadre de leur formation.  
Tout autre usage est interdit sans l'autorisation écrite du Cnam.*

## SOMMAIRE

<b>INTRODUCTION .....</b>	<b>4</b>
<b>HELLO WORLD .....</b>	<b>6</b>
<b>I- ALGORITHMIQUE .....</b>	<b>8</b>
I.1- LES INSTRUCTIONS .....	8
I.2- LES VARIABLES .....	9
I.3- LES EXPRESSIONS MATHÉMATIQUES .....	11
I.4- LES STRUCTURES DE DÉCISION .....	14
I.5- LES BOUCLES ITÉRATIVES .....	15
• EXERCICES .....	16
<b>II- LE DIALOGUE OPÉRATEUR .....</b>	<b>17</b>
<b>III- VARIABLES SIMPLES .....</b>	<b>19</b>
III.1- LES VARIABLES NUMÉRIQUES .....	19
III.2- LES CARACTÈRES .....	20
III.3- LES BOOLEENS .....	21
• EXERCICES .....	21
<b>IV- TABLEAUX .....</b>	<b>22</b>
• EXERCICES .....	23
<b>V- CHAINES DE CARACTÈRES .....</b>	<b>24</b>
• EXERCICES .....	28
<b>VI- VARIABLES COMPOSÉES .....</b>	<b>29</b>
VI.1- LES STRUCTURES .....	29
VI.2- LES UNIONS .....	30
VI.3- LES ENUMÉRATIONS .....	30
• EXERCICES .....	31
<b>VII- LES POINTEURS .....</b>	<b>32</b>
• EXERCICES .....	35
<b>VIII- LES FONCTIONS .....</b>	<b>36</b>
VIII.1- DÉFINITION DE FONCTION .....	36
VIII.2- BIBLIOTHÈQUES DE FONCTIONS .....	39
• EXERCICES .....	39
<b>MINI-PROJET .....</b>	<b>40</b>
<b>IX- LA PORTEE DES VARIABLES .....</b>	<b>43</b>
<b>X- LA REDEFINITION DES TYPES .....</b>	<b>44</b>
• EXERCICE .....	44

<b>XI- LE PREPROCESSEUR .....</b>	<b>45</b>
<b>XII- STRUCTURE D'UN PROGRAMME "C" .....</b>	<b>48</b>
• EXERCICES.....	49
<b>XIII- LA GESTION DE LA MEMOIRE .....</b>	<b>50</b>
XIII.1- ESPACE MEMOIRE DU PROGRAMME.....	50
XIII.2- ALLOCATION DYNAMIQUE DE MEMOIRE.....	52
XIII.3- RECOPIE DE ZONES MEMOIRE .....	52
• EXERCICES.....	53
<b>XIV- LE TRAITEMENT DES ERREURS .....</b>	<b>54</b>
XIV.1- LES FONCTIONS DES BIBLIOTHEQUES C ET SYSTEME.....	54
XIV.2- LES FONCTIONS DE L'APPLICATION .....	55
XIV.3- LES ERREURS SYSTEME .....	56
CAS DES ERREURS SYSTEME SOURNOISES.....	58
• EXERCICE .....	60
<b>XV- LA GESTION DES FICHIERS.....</b>	<b>61</b>
XV.1- OUVERTURE / CREATION DE FICHIER .....	61
XV.2- FLUX BINAIRE .....	62
XV.3- LE POSITIONNEMENT.....	62
XV.4- SUPPRESSION DE FICHIER.....	63
XV.5- FLUX TEXTES FORMATTES .....	63
• EXERCICES.....	64
<b>XVI- LA PROGRAMMATION MULTITHREAD .....</b>	<b>65</b>
• EXERCICE .....	66
<b>ANNEXE A : CORRIGES DES EXERCICES .....</b>	<b>68</b>
I- ALGORITHMIQUE .....	68
III- VARIABLES SIMPLES.....	69
IV- TABLEAUX.....	70
V- CHAINES DE CARACTERES .....	73
VI- VARIABLES COMPOSEES.....	74
VII- LES POINTEURS.....	75
VIII- LES FONCTIONS .....	77
X- LA REDEFINITION DES TYPES .....	78
XII- STRUCTURE D'UN PROGRAMME "C".....	78
XIII- LA GESTION DE LA MEMOIRE .....	79
XIV- LE TRAITEMENT DES ERREURS .....	79
XV- LA GESTION DES FICHIERS .....	79
XVI- LA PROGRAMMATION MULTITHREADS .....	83
<b>ANNEXE B : INSTALLATION DU COMPILATEUR C.....</b>	<b>85</b>
INSTALLATION SOUS LINUX.....	85
INSTALLATION SOUS WINDOWS .....	85
INSTALLATION SOUS MAC OS.....	85
<b>ANNEXE C : INTERPRETATION DES DECLARATIONS.....</b>	<b>86</b>
<b>ANNEXE D : LES NORMES.....</b>	<b>88</b>
<b>BIBLIOTHEQUES STANDARDS DU C-89 .....</b>	<b>89</b>
<b>INDEX DES 32 MOTS-CLEFS DU C-89.....</b>	<b>90</b>

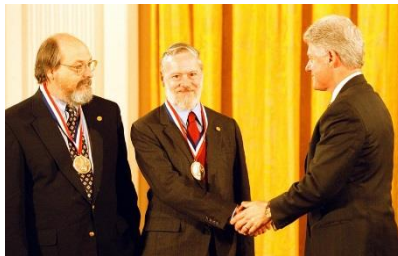
## INTRODUCTION

**Le langage C** est un langage évolué compilé qui possède la particularité de rester proche du langage machine, grâce à une gestion transparente de la mémoire et à la possibilité d'insérer directement de l'assembleur dans le code.

Ces spécificités font du **langage C** le langage de référence pour le développement des systèmes d'exploitation, des couches réseau et des grands logiciels : Linux, Windows, Oracle, Microsoft Office, Chrome, Python etc.

**Le langage C** a été développé par Denis Ritchie qui a procédé à un élargissement du langage B<sup>1</sup>, il a été conçu pour le portage du système d'exploitation Unix. Un portage pilote a été effectué sur un des tous premiers mini-ordinateurs "non-IBM" le PDP11<sup>2</sup>.

**Le Système Unix** a été conçu par Ken Thompson en 1969, dans le cadre d'un projet des Bell Laboratories<sup>3</sup>.



*Remise de la "National Medal of Technology and Innovation"  
par le président Clinton à Ken Thompson et Denis Ritchie en 1998  
(source [nationalmedals.org](http://nationalmedals.org))*

### L'objectif de cet ouvrage

Le présent ouvrage est une présentation raisonnée du langage C, traitant de la programmation mais aussi de sa mise en œuvre organique sous-jacente.

L'objectif étant de permettre aux programmeurs de développer des programmes structurés et de mieux maîtriser la chaîne de production "C".

---

<sup>1</sup> Le langage B développé par Ken Thompson est une version allégée du BCPL (Basic CPL) lequel est lui-même une version allégée du CPL. L'ambitieux CPL (Combined Programming Language) conçu par les universités de Cambridge et de Londres réunies est toujours au stade de prototype.

<sup>2</sup> Programmable Data Processor 11 de DEC (Digital Equipment Corporation), un des premiers mini-ordinateurs.

<sup>3</sup> En hommage à Graham Bell inventeur du téléphone, un des laboratoires du groupe AT&T (American Telegraph & Telephone)

## Prérequis

Cet ouvrage s'adresse à des auditeurs maîtrisant l'algorithmique informatique :

- Variables
- Opérateurs
- Structures de décision
- Structures de variables
- Boucles itératives
- Fonctions

Ainsi que les concepts de base des systèmes d'exploitation :

- La notion de Processus
- La gestion de la mémoire
- Le rôle de la pile

## Environnement de Développement

Vous aurez besoin comme outils de développement d'un compilateur C et d'un simple éditeur de texte. Utilisez votre éditeur de texte préféré.

Nous travaillerons en mode texte dans la console (terminal ou invite de commandes), la présentation graphique ne sera pas traitée pour la simple raison que le compilateur "C" ne fournit pas de bibliothèque graphique standard.

Les exemples de cet ouvrage sont basés sur le compilateur libre Gnu 4.8.4, implémenté sur plateforme Ubuntu 14.04-LTS. Nous faisons confiance aux programmeurs Unix-like<sup>4</sup> et Windows pour adapter ces exemples à leur propre plateforme.

En particulier c'est le compilateur "MinGW" (Minimum GNU for Windows) qui correspond au portage du compilateur GNU sur les plateformes Windows.

➤ Installation du Compilateur C, cf. Annexe A.

## Le Langage C++

🚦 Le langage C++ est un langage orienté objet qui est supposé être un sur-ensemble du langage C, cependant il existe quelques différences structurelles entre ces 2 langages, nous les signalerons en utilisant le symbole ci-contre.

---

<sup>4</sup> Unix, Linux et Mac OS

## HELLO WORLD

Depuis la parution de l'ouvrage "The C Programming Language" de Richie et Kernighan, il est d'usage dans un ouvrage de programmation de présenter un programme élémentaire qui affiche le message "Hello World". C'est symbolique, on donne la vie au programme et en échange le programme salue le monde qu'il découvre.

### Fichier source "hello.c"

```
#include <stdio.h>
int main () {           // fonction principale main()
    printf("Hello World\n"); // affichage de "Hello World"
}
```

- Le nom d'un fichier source C doit comporter l'extension ".c" (pour le compilateur GNU).
- `stdio.h` est un fichier de type "include" fourni par le compilateur, il nous permet d'accéder à la bibliothèque "stdio" (standard Input/Output) laquelle contient la fonction d'affichage `printf()`.
- La fonction `main()` est obligatoire dans tout programme C, elle correspond à la première fonction qui sera lancée par le système d'exploitation.
- Le retour de type `int` de la fonction `main()` correspond au compte-rendu d'exécution du programme lequel par défaut sera à 0 ce qui signifie "exécution correcte".
- Le '\n' correspond à un caractère saut de ligne qui sera inséré à la fin du message.
- Les 2 slash "//" ouvrent un commentaire

### Génération de l'exécutable sous Unix-like

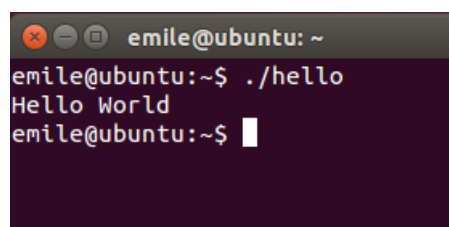
```
$ gcc hello.c -o hello
```

- `gcc` = appel au compilateur Gnu "C"
- l'option "`-o hello`" permet de nommer l'exécutable `hello`,

### Lancement du programme sous Unix-like

- Il faudra préciser le répertoire courant car il ne se trouve sans doute pas dans le PATH :

```
$ ./hello
```



Et si le compilateur n'a pas attribué automatiquement le droit d'exécution au fichier `hello`, ce sera à vous de le faire :

```
$ chmod +x hello
```

## Génération de l'exécutable sous Windows

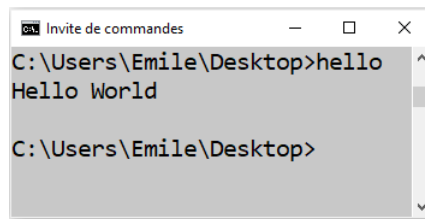
```
$ gcc hello.c -o hello.exe
```

- `gcc` = appel au compilateur Gnu "C"
- l'option "`-o hello.exe`" permet de nommer l'exécutable\* `hello.exe`,

\* Sur une plateforme Windows il faudra donner à l'exécutable l'extension ".exe" :

## Lancement du programme sous Windows

```
> hello
```



## I- ALGORITHMIQUE

- En langage "C" la casse des lettres est significative (majuscules/minuscules).

### I.1- Les Instructions

Les instructions se terminent par un point-virgule ";" :

```
a = 5;
```

Le caractère anti-slash "\" en fin de ligne permet de prolonger une ligne d'instruction :

```
texte = "bla bla bla bla bla bla bla bla \
bla bla bla bla bla";
```

#### Les blocs d'instructions

Certaines instructions C impliquent l'utilisation d'un bloc d'instructions lequel correspond à une séquence d'instructions, délimitée par des accolades :

```
{
    instruction_1;
    instruction_2;
    ...
    instruction_n;
}
```

*\* Le ";" à la fin de la dernière instruction du bloc est facultatif.*

*\*\* Si le bloc d'instructions ne contient qu'une seule instruction les accolades sont facultatives, dans ce cas il faudra conserver le ";" final.*

- ❖ Le langage C permet l'imbrication des blocs d'instructions.

#### Les commentaires

Les commentaires commencent par "//" et se terminent à la fin de la ligne.

```
// ceci est un commentaire
```

Ou sont encadrés par "/\*" et "\*/"

```
/* ceci est un commentaire */
```

Ce second cas autorise les commentaires multilignes :

```
/* ceci est un
commentaire
multiligne */
```

## 1.2- Les Variables

Une variable est identifiée par son nom et elle possède un type.

Une variable doit être déclarée avant toute utilisation et son type doit être précisé lors de cette déclaration ; une fois déclaré le type d'une variable ne pourra plus être modifié.

```
int nombre;           //Déclaration d'une variable de nom "nombre"
                       //et de type entier
long adresse;         //Déclaration d'une variable de nom "adresse"
                       //et de type entier long

float prix;           //Déclaration d'une variable de nom "prix"
                       //et de type flottant (nombre avec décimales)

char c;               //Déclaration d'une variable de nom "c"
                       //et de type caractère
```

- ❖ Le langage C manipule 4 catégories de variables :
  - Les variables simples : Nombres, Booléens et Caractères,
  - les variables composées : Structures, Unions et Enumérations,
  - les tableaux qui sont des collections de variables de même type,
  - les pointeurs qui contiennent des adresses mémoire de variables.

🚦 Le langage C++ propose une nouvelle catégorie de variable : les objets.

### Affectation

C'est l'opérateur "=" qui permet d'affecter une valeur à une variable :

```
int nombre;
nombre = 5;
int nombre = 5;           // Affectation lors de la déclaration
```

On peut aussi recopier une variable dans une autre variable de type compatible,

```
int nombre1 = 10;
int nombre2 = nombre1;    // -> nombre2 = 10
```

- ❖ Cette recopie de variables ne fonctionnera pas avec les tableaux (cf. chapitre "Les Pointeurs").
- Nous verrons par la suite que l'on peut également affecter à une variable une expression mathématique ou le résultat d'une fonction.

### Nommage

Le nom d'une variable peut comporter des lettres (non accentuées), des chiffres et le caractère underscore "\_" ; le nom ne doit cependant pas commencer par un chiffre ni comporter de blancs.

Le nom d'une variable ne peut pas être un mot-clef du langage (cf. liste en annexe).

## Les constantes

Le qualificatif `const` fige une variable en constante c'est à dire qu'une fois sa valeur définie elle ne pourra plus être modifiée.

```
const int nombre = 5;
const char lettre = 'a';
```

## Les Pointeurs

Une variable est stockée en mémoire c'est l'opérateur "&" qui permet d'obtenir l'adresse mémoire<sup>5</sup> d'une variable.

Un pointeur est une variable dont la valeur est l'adresse mémoire d'une autre variable :

```
int nombre;           // Déclaration de l'entier nombre
int *p_nombre;        // Déclaration du pointeur p_nombre
                      // qui pointe sur un entier
p_nombre = &nombre;   // on affecte à p_nombre l'adresse
                      // de nombre, on dit aussi que
                      // p_nombre pointe sur nombre
```

- Le symbole "\*" dans une déclaration de variable signifie qu'il s'agit d'un pointeur.
- Les pointeurs sont typés et leur type doit correspondre à celui de la variable sur laquelle ils pointent.

On peut accéder à la valeur d'une variable à partir d'un pointeur qui pointe sur cette variable, cela s'appelle l'indirection :

```
int nombre = 12;
int *p_nombre = &nombre;
int a = *p_nombre;           // équivalent à "int a = nombre"
printf("%d\n", a)             // -> 12
*p_nombre = 0;                // ici "nombre" est remis à 0
printf("%d\n", nombre)        // -> 0
```

- Le symbole "\*" lors de l'accès à une variable représente l'indirection.
- ❖ Nous étudierons les pointeurs plus en détail au chapitre correspondant.

---

<sup>5</sup> Il s'agit là d'une adresse mémoire dite linéaire relative à l'implémentation du programme en mémoire.

### I.3- Les Expressions Mathématiques

#### Opérateurs arithmétiques

addition	soustraction	division	multiplication	incrémentation	décrémentation	modulo
+	-	/	*	++	--	%

- Si les 2 opérandes sont entiers (**int**, **long** ...), la division sera entière, sinon si au moins un des opérandes est un **float** la division sera réelle.
- L'incrémentation ajoute 1 à la variable.
- Le comportement des incréments++ (et décréments--) sera précisé au chapitre "Expressions Mathématiques"
- Existents également les raccourcis "+=" ; "-=" ; "\*=" ; "/=" par exemple :  
`a = a + b;` est équivalent à `a += b;`

#### Opérateurs de comparaison

égalité	inégalité	plus grand	plus petit	plus grand ou égal	plus petit ou égal
==	!=	>	<	>=	<=

Ces opérateurs réservés aux variables numériques, s'étendent aussi aux caractères en considérant leur codage numérique sur 1 octet (ASCII).

#### Opérateurs logiques

et	ou	Non
&&		!

Ces opérateurs manipulent des variables booléennes (cf. chapitre "Les Variables Simples") ou des expressions booléennes (cf. plus bas).

#### Opérateur booléens

et	ou	ou exclusif	complément	décalage bit à droite	décalage bit à gauche
&		^	~	>>	<<

Existents également les incréments `"&="` ; `"|="` ; `"^="` ; `"~="` ; `">>="` ; `"<<="`.

Ces opérateurs fonctionnent bit à bit sur toute la longueur des opérandes, par exemple :

```
unsigned short x = 0xFF00;
unsigned short y = 0x00FF;
printf ("%x\n",x|y);    // -> 0xFFFF
```

#### Opérateur conditionnel ternaire

`<Expression logique> ? <Résultat si True> : <Résultat si False>`

Par exemple :

```
x = x > 0 ? x : -x    // équivalent valeur absolue de x
```

## Expressions numériques

Tout assemblage cohérent de variables numériques et d'opérateurs arithmétiques constitue une expression numérique (dont la valeur est de type numérique), par exemple :

```
(( (26200/45) + montant)*25) + 100
```

*\* On aurait pu utiliser les règles de précedence des opérateurs pour limiter le nombre de parenthèses, dans ce document nous préférons vous présenter la notation mathématique.*

- ❖ Une expression numérique peut être affectée à une variable numérique :

```
nouveau_montant = (( (26200/45) + montant)*25) + 100
```

## Expressions logiques

Une expression logique comporte des opérateurs de comparaison et ou des opérateurs logiques et ne peut avoir qu'une seule des 2 valeurs "**Vrai**" ou "**Faux**", par exemple :

```
(montant >= 1000) && (nbr_articles > 2)
```

*\* Si l'expression est fausse l'expression aura pour valeur 0, autrement l'expression aura une valeur entière différente de 0 (cf. "Variables booléennes").*

- ❖ En fait toute expression numérique ou mixte numérique/logique peut être considérée comme une expression logique, la valeur "0" représentera "**Faux**" et toute valeur différente de "0" représentera "**Vrai**".
- ❖ Une expression logique peut donc être affectée à une variable numérique :

```
remise = ((montant >= 1000) && (nbr_articles > 2))
```

*\* **remise** aura pour valeur 0 (**Faux**) ou différent de 0 (**Vrai**)*

## L'évaluation des opérandes

Les opérandes sont évalués avant d'effectuer les opérations correspondantes et en particulier :

- L'incrémentation **a++** (ou la décrémentation **a--**) est effectuée juste après l'évaluation de l'opérande dans laquelle se trouve la variable **a**.
- L'incrémentation **++a** (ou la décrémentation **--a**) est effectuée juste avant l'évaluation de l'opérande dans laquelle se trouve la variable **a**.
- Dans le cas de l'opérateur **&&** le second opérande ne sera évalué que si le premier est **Vrai** (sinon l'expression vaudra forcément **Faux**).
- Dans le cas de l'opérateur **||** le second opérande ne sera évalué que si le premier est **Faux** (sinon l'expression vaudra forcément **Vrai**).

## Le Transtypage

Le transtypage s'applique aux variables numériques et par extension aux caractères lesquels sont aussi considérés comme des entiers sur un seul octet.

- Si on soumet à un opérateur des opérandes numériques de types différents, les valeurs des opérandes seront converties automatiquement dans un type commun avant l'opération, ces transtypes convertissent des types plus "étroits" en des types plus "larges", par exemple des `int` en `float`, de façon à ne pas perdre en précision.

```
int a = 2;
float b = 10.5;
float x = b/a; // a sera converti en float avant le calcul
printf("%f",x); // -> 5.25
```

- Lors d'une affectation d'un résultat d'expression numérique, la valeur à droite du signe d'affectation est convertie dans le type à gauche de ce signe. Dans ce cas, il pourrait y avoir perte de précision si le type de gauche est plus "étroit" que celui de droite.

```
int a = 2;
float b = 10.5;
int x = b/a; //a sera converti en float avant le calcul
// puis le résultat converti en entier
printf("%i",x); // -> 5

float y = x; // on revient en float, la précision est perdue
printf("%f",y); // -> 5
```

## L'opérateur "Cast"

- Le programmeur peut forcer les transtypes décrits ci-dessus à l'aide de l'opérateur dit "cast", par exemple ci-dessous le programmeur souhaite une division flottante, il l'impose avec le "cast" en `(float)` de l'un des 2 opérandes, l'autre opérande sera alors automatiquement élevé en `float` et la division sera une division `float` :

```
int a=6;
int b=4;
float c;
c = a/b;
printf ("%f",c); //-> 1.
c =(float)a/b; // ici on caste* "a" en float
printf ("%f",c); //-> 1.5
```

*\* Pour "caster" une variable on la préfixe par le type souhaité entre parenthèses.*

- Le "cast" pourra également être utilisé lors de l'allocation d'une zone mémoire pour typer l'utilisation de cette zone (cf. "pointeur `void`" au chapitre "Les Pointeurs").

## ***I.4- Les Structures de Décision***

### **I.4.1- Les Débranchement Conditionnels**

```
if (Expression logique)
    {bloc d'instructions}
```

#### *Débranchement conditionnel avec alternative*

```
if (Expression logique)
    {bloc d'instructions 1}
else
    {bloc d'instructions 2}
```

par exemple :

```
if (age < 12)
    prix_ticket = 0;
else
    prix_ticket = 8.50;
```

#### *Débranchements conditionnels en cascade*

```
if (Expression logique 1)
    {bloc d'instructions 1}
else if (Expression logique 2)
    {bloc d'instructions 2}
...
else if (Expression logique n)
    {bloc d'instructions n}
else
    {bloc d'instructions alternatif}
```

#### *Commutateur de débranchement conditionnel*

```
switch(variable) {
case valeur_1 :
    séquence d'instructions 1
    break;
case valeur_2 :
    séquence d'instructions 2
    break;
...
...
...
case valeur_n :
    séquence d'instructions n
    break;
default :
    séquence d'instructions alternative
}
```

### Précisions :

- l'instruction **switch** est suivie d'un bloc d'instructions avec une structure particulière, ce bloc comprend des séquences d'instructions.
- si **variable** a pour valeur **valeur\_1** alors c'est la séquence 1 qui sera exécutée.
- si aucun des **case** ne correspond à la valeur de **variable** alors c'est la séquence d'instructions **default** qui sera exécutée.
- un **case** exécuté va enchaîner le **case** suivant, si on ne souhaite pas cet enchaînement il faut utiliser une instruction **break** de sortie du **switch**.

#### **I.4.2- Le Débranchement Inconditionnel**

L'instruction de débranchement inconditionnel "**goto**" permet d'aller de n'importe où à n'importe où<sup>6</sup> à l'intérieur d'une même fonction.

```
goto suite
...
...
suite:           // étiquette de branchement
```

Les débranchements inconditionnels sont bannis des langages de programmation qui se disent "structurés", lesquels les considèrent comme dangereux et nuisant à la lisibilité du code.

Cependant utilisés à bon escient les débranchements inconditionnels peuvent permettre un code plus concis et plus lisible, notamment dans les cas de traitement d'erreurs en cascade ou de sortie de plusieurs boucles itératives imbriquées (cf. **break** ci-dessous).

#### **I.5- Les Boucles Itératives**

##### **Boucle "Pour" :**

```
for (debut, fin, incr)
    {bloc d'instructions}
```

avec :

- **debut** = instruction d'initialisation, par exemple **i = 0**
- **fin** = condition de maintien dans la boucle, par exemple **i < 10**
- **incr** = instruction d'incrément, par exemple **i++**

```
for (i = 0; i < 10; i++){...}; // 10 passages dans la boucle
```

**debut, fin et incr** peuvent être des listes (séparées par des " , " ) :

```
for (i=0, j=10; i < 10; i++, j+=2){...};
```

---

<sup>6</sup> Pratiquement, il y a quelques restrictions

### Boucle "Tant que" :

```
while condition
    {bloc d'instructions}

do while condition
    {bloc d'instructions}
```

Dans le second cas la condition n'est évaluée qu'à la fin du bloc d'instructions.

### Interrompre une itération :

L'instruction **break** permet de quitter la boucle itérative.

L'instruction **continue** permet de passer directement à l'itération suivante (sans aller jusqu'à la fin du bloc d'instructions).

## ■ Exercices

Quelques exercices élémentaires pour mettre en action les instructions de base de l'algorithmique :

- Les variables
- Les opérateurs
- Les structures de décision
- Les boucles itératives

#### 1) Les 4 opérations

Déclarez 2 variables entières **a** et **b**, initialisez les, puis additionnez les, soustrayez les, multipliez les et enfin divisez les entre eux.

Les résultats seront affectés à une variable **c**, laquelle sera affichée précédée par le nom des opérations : **addition**, **soustraction**, **multiplication**, **division**.

#### 2) Test Conditionnel

Déclarez 2 variables entières **a** et **b**, initialisez les puis comparez les entre elles et affichez le résultat en langage naturel : "**a** plus grand que **b**" etc.

#### 3) Lectures / Ecritures mémoire

Expliquez dans les 2 exercices précédents où interviennent les lectures / écritures en mémoire.

#### 4) Boucle Itérative

Développez une boucle itérative pour calculer les 10 premières puissances de 2 et affichez les résultats.

## II- Le Dialogue Opérateur

Ce sont les fonctions `scanf()` et `printf()` qui vont nous permettre de lire et écrire des caractères dans la console (terminal en mode texte).

### La fonction `printf()`

Nous avons déjà abordé la fonction `printf()` lors de l'exercice "Hello World". C'est une fonction qui permet d'afficher des messages dans la console (qui est le fichier standard de sortie `stdout`) :

```
#include <stdio.h>
int printf(const char *format, ...);
```

- ❖ `printf()` Retourne le nombre de caractères affichés, sinon en cas de faute retourne une valeur négative.

Le premier argument de `printf` est une chaîne de caractères comportant du texte et le format de chacune des variables qui suit.

Un élément de format est de la forme `%[w][.p]<type>`

Voici les principaux types :

Type	Signification
<code>c</code>	Caractère
<code>d</code> ou <code>i</code>	Entier décimal signé
<code>ld</code> ou <code>li</code>	Entier décimal long signé
<code>u</code>	Entier décimal non signé
<code>lu</code>	Entier décimal long non signé
<code>x</code>	Entier hexadécimal non signé
<code>lx</code>	Entier hexadécimal long non signé
<code>f</code>	Nombre décimal flottant signé, séparateur le point
<code>e</code>	Notation scientifique (signe, mantisse/exposant) utilisant le "e" minuscule
<code>s</code>	Chaîne de caractères

Et voici les attributs optionnels de largeur et de précision :

Attribut	Action
largeur <code>w</code>	Représente le nombre minimum de caractères à afficher, s'il n'est pas atteint l'affichage sera préfixé par des blancs, mais si l'affichage dépasse <code>w</code> il ne sera pas tronqué.
précision <code>.p</code>	<ul style="list-style-type: none"><li>- Pour les nombres entiers permet de préciser le nombre minimum de chiffres à afficher, s'il n'est pas atteint le nombre sera préfixé par des <code>0</code>, mais si le nombre dépasse <code>w</code> il ne sera pas tronqué.</li><li>- Pour les nombre flottants permet de préciser le nombre de décimales à afficher (par défaut 6).</li></ul>

Exemples :

```
printf("Bonjour\n");           // Texte fixe pas de variables
                                // -> Bonjour (saut de ligne).

char nom[100] = "Emile";
int age = 50;
printf("Bonjour %s vous avez %d ans\n", nom, age);
                                // -> Bonjour Emile vous
                                //      avez 50 ans (saut de ligne).

char nom[100] = "Emile";
float prix = 25.50;
printf("Bonjour %s vous nous devez %.2f Euros\n", nom, prix);
                                // -> Bonjour Emile vous nous devez
                                //      25.50 Euros (saut de ligne).
```

Précisions :

- **Bonjour** et **Emile** sont des chaînes de caractères (cf. chapitre "Les chaînes de caractères")
- **char nom[100] = "Emile";** est une déclaration/affectation de chaîne de caractère à la variable **nom** (cf. chapitre "Les chaînes de caractères")

### La fonction scanf()

**scanf ()** va nous permettre de lire un message saisi au clavier (qui est le fichier standard d'entrée **stdin**) et d'en ranger les éléments dans des variables :

```
#include <stdio.h>
int scanf(const char *format, ...);
```

- ❖ Retourne le nombre de variables correctement interprétées et s'arrête à la première faute.

La saisie au clavier se termine par la touche entrée, par exemple pour saisir le nom et l'âge de l'utilisateur :

```
int age;
char nom[100];
printf ("Veuillez saisir votre nom : ");
scanf ("%s", nom);           // *
printf ("Veuillez saisir votre âge : ");
scanf ("%i", &age);          // *
printf ("Bonjour %s vous avez %i ans\n", nom, age);
```

Précisions :

- **scanf ()** attend en argument des adresses de variables d'où l'utilisation de l'opérateur **&** devant les noms de variables à l'exception des tableaux et des chaînes de caractères (cf. chapitre "Les Pointeurs").
- Dans le cas de chaînes de caractères **scanf ()** rajoute automatiquement le séparateur final **\0** à la chaîne de caractères saisie (cf. chapitre "Les chaînes de caractères").

Limitations :

La fonction **scanf ()** ne récupère pas de façon fine les erreurs de saisies : lettres au lieu de chiffre par exemple, et en format **"%s"** elle s'arrête au premier caractère blanc rencontré etc.. (cf. chapitre "Les chaînes de caractères")

### III- Variables Simples

Les variables simples sont les variables numériques, les caractères et les booléens.

#### III.1- Les Variables Numériques

Type	Définition	Dimension
<code>char</code>	Caractère*	1 octet
<code>signed char</code>	Octet signé	1 octet
<code>unsigned char</code>	Octet non signé	1 octet
<code>short</code>	Entier court signé	2 octets
<code>unsigned short</code>	Entier court non signé	2 octets
<code>int</code>	Entier signé	2 octets minimum
<code>unsigned</code>	Entier non signé	2 octets minimum
<code>long</code>	Entier long signé	4 octets minimum
<code>unsigned long</code>	Entier long non signé	4 octets minimum
<code>float</code>	Nombre flottant	4 octets
<code>double</code>	Nombre flottant double précision	8 octets
<code>long double</code>	Nombre flottant quadruple précision	16 octets

*\* Octet signé ou non signé selon les compilateurs, cité à titre historique, à utiliser uniquement pour le type caractère*

#### Entiers

- Les entiers se notent sans délimiteurs :

```
int nombre = 245;
```

Les nombres octaux\* se notent préfixés par "0" et les nombres hexadécimaux se notent préfixés par "0x" :

```
int nombre = 010;      // 8 décimal
int nombre = 0x10;     // 16 décimal
```

*\* attention à ne pas mettre des "0" en tête de vos nombres décimaux*

- On peut préciser les types des entiers par des suffixes, cela peut être utile pour la précision des calculs (cf. chapitre "Transtypage") :

```
1234U      // traiter comme un entier non signé
123456789L // traiter comme un entier long
123456789UL // traiter comme un entier long non signé
```

- Les dimensions des nombres entiers dépendent des plateformes et des compilateurs, le fichier include `limit.h` donne les valeurs extrêmes des différents types d'entiers.

L'opérateur `sizeof` permet de connaître la dimension en octets d'un type :

```
int x = sizeof (int);
int x = sizeof (char);
```

## Nombres Flottants

- Les nombres flottant comme les nombres entiers se notent sans délimiteurs :

```
float montant = 17.56;  
float montant = 1.5754e7;
```

Le "."<sup>7</sup> est le séparateur décimal et le symbole "e" représente la puissance de 10 :

On peut préciser les types des flottants par des suffixes, cela peut être utile pour la précision des calculs (cf. "Transtypage" ci-dessous) :

```
1234F      // traiter comme un flottant  
1.5754e7D  // traiter comme un double  
1.5754e7L  // traiter comme un long double
```

- Les dimensions des nombres flottants dépendent des plateformes et des compilateurs. Les dimensions mantisse / exposant sont données dans le fichier include `float.h`.

## Le séparateur décimal

Le séparateur décimal est le point anglo-saxon. Pour saisir ou afficher des nombres avec une virgule décimale, il va falloir manipuler les nombres en tant que chaînes de caractères, cf. chapitre "Les Chaînes de Caractères".

## III.2- Les Caractères

En langage C on distingue les caractères des chaînes de caractères.

Les caractères correspondent au type `char`, un `char` peut être aussi vu comme un octet numérique (cf. le chapitre précédent). Le codage des caractères est le codage ASCII (7 bits, 128 caractères), lequel ne comprend pas de lettres accentuées.

Nous verrons par la suite que les chaînes de caractères sont des tableaux de caractères et qu'elles utilisent la table de codage Latin-1 (8 bits, 256 caractères) laquelle comporte les lettres accentuées.

## Types étendus

----- à compléter types étendus de caractères `wchar_t` ... -----

## Notation

Les simples quotes sont les délimiteurs de caractères (attention les délimiteurs de chaînes de caractère seront des doubles quotes) :

```
char lettre = 'G';
```

---

<sup>7</sup> Notation anglo-saxonne

Pour accéder aux caractères non imprimables ou interprétables il faut utiliser le caractère d'échappement "`\`" ou le code ASCII :

Mnémonique	Caractère	Code ASCII (hexa)
<code>\0</code>	Terminaison chaîne de caractères*	0x00
<code>\r</code>	Retour chariot	0x0D
<code>\n</code>	Saut de ligne	0x0A
<code>\\</code>	Antislash	0x5C
<code>\'</code>	Quote	0x27
<code>\"</code>	Double quotes	0x22

\* octet final d'une chaîne de caractères (cf. chapitre correspondant).

### III.3- Les Booléens

Il n'existe pas de type booléen, il faut utiliser une variable numérique entière (`int`, `long` ... ) : La valeur "`0`" représente "**faux**" et toute autre valeur représente "**vrai**"

✚ Le langage C++ introduit le type `bool` qui peut prendre les 2 valeurs `true` et `false`, cependant l'utilisation d'une variable numérique reste valable.

#### ■ Exercices

1) Soit les déclarations suivantes :

```
int x = 2 ; int y = 4 ; float z = 4.55;
```

Donner le type et la valeur de chacune des expressions suivantes :

- a) `x * y`
- b) `x + z`
- c) `x < y`
- d) `x && y`
- e) `y == 4`

2) Ecrire un programme permettant de calculer les intérêts composés au bout de 10 ans d'une somme de 10.000 € à un taux de 7%.

3) Ecrire un programme pour calculer le nombre d'années de remboursement d'une somme de 100.000 € empruntée à un taux de 5% avec des remboursements annuels de 10.000€.

## IV- Tableaux

Un tableau est une variable qui est une collection de variables de même type, chaque élément (variable) étant identifiée par un index (numéro d'ordre) qui va de "0" à "**n-1**" ("**n**" étant la dimension du tableau).

Lors de la déclaration d'un tableau on fixe sa dimension (il ne pourra plus s'agrandir lors de l'exécution) :

```
int tab[100];      // Déclaration d'un tableau de 100 entiers
tab[0] = 150;      // affectation du premier élément du tableau
                  // 0 correspond à l'index

int x;
x = tab[0]; // accès au premier élément du tableau
            // x est égal à 150
```

- ❖ Dans les exemples ci-dessus les tableaux comportent des variables simples, on peut également constituer des tableaux de pointeurs ou de variables composées, ou des tableaux de tableaux comme décrit ci-dessous.

Un tableau à 2 dimensions est considéré comme un tableau de tableaux :

```
int tab[100][5];   //Déclaration d'une tableau de 100 éléments
                  //Contenant chacun un tableau de 5 entiers
tab[2][0] = 364;   //affectation de l'élément d'index 2,0
int x = tab[2][0]; //accès à l'élément d'index 2,0
                  // x est égal à 364
```

- ❖ En "C" on peut créer des tableaux à "**n**" dimensions : Tableaux de tableaux de tableau ... La limite du nombre d'imbrications dépend du compilateur.

### Initialisation

On peut initialiser un tableau lors de sa déclaration :

```
int tab[5] = { 10, 36, 25, 14, 31 }; // ";" final obligatoire
```

Autrement l'initialisation et la mise à jour d'un tableau s'effectuent élément par élément :

```
int tab[5];
int i;
for (i = 0; i < 5 ; i++) tab[i] = 0;
```

### Occupation mémoire

Les variables d'un tableau sont rangées consécutivement en mémoire (dans l'ordre de l'index) de façon contiguë sans aucun octet de vide entre les éléments :

```
int tab [100];
sizeof (int);      // -> 4
sizeof(tab);       // -> 400
```

## ■ Exercices

### 1) Tableau d'entiers

Déclarez un tableau de 20 entiers, initialisez le à 0, puis affichez la valeur des 20 éléments du tableau.

### 2) Tableau d'entiers à 2 dimensions

Idem exercice précédent mais cette fois ci il s'agit d'un tableau à 2 dimensions (tableau de tableaux) de 20 lignes et 10 colonnes.

### 3) Ecrire un programme qui va :

- Saisir 10 nombres entiers et les ranger au fur et à mesure dans une table d'entiers.
- Calculer la moyenne des 10 nombres ainsi saisis.
- Afficher la moyenne

#### Remarque :

*Nous ne traitons pas les erreurs de saisie, si l'utilisateur saisi des caractères au lieu de chiffres par exemple le comportement du programme serait imprévisible.*

*Nous traiterons la gestion des erreurs plus loin dans ce document.*

### 4) Ecrire un programme permettant d'ordonner les éléments d'un tableau de 10 entiers (les ranger dans l'ordre croissant de leurs valeurs).

- *On utilisera l'algorithme du "tri à bulles" : On parcourt une première fois la table par couples "élément/élément suivant" et à chaque fois qu'un élément est supérieur à son élément suivant on inverse ces 2 éléments, ensuite on parcourt "n" fois la table en inversant les couples d'éléments à chaque fois que nécessaire jusqu'à ce qu'il n'y ait plus d'inversions à effectuer auquel cas la table sera ordonnée dans l'ordre croissant.*

### 5) Ecrire un programme pour résoudre le problème des prisonniers du pirate :

- *Le pirate met ses **nbr** prisonniers en cercle et tue un premier prisonnier pris au hasard, puis il saute **3** prisonniers et tue le **quatrième** et ainsi de suite (**3** vivants sautés le **quatrième** tué), le **dernier** prisonnier aura la vie sauve.*

*Soit **0** le numéro du premier prisonnier tué, quel sera le numéro du prisonnier qui aura la vie sauve.*

Pour répondre à cet exercice nous vous suggérons de construire un tableau de chainage à 2 dimensions dans lequel pour chacun des **n** prisonniers nous allons indiquer son précédent encore vivant et son suivant encore vivant.

## V- CHAINES DE CARACTERES

Le type chaîne de caractères n'existe pas en langage C, les chaînes de caractères sont des tableaux de caractères avec un octet final à "0".

```
char texte[8];           // Chaîne "Bonjour"
texte[0] = 'B';
texte[1] = 'o';
texte[2] = 'n';
texte[3] = 'j';
texte[4] = 'o';
texte[5] = 'u';
texte[6] = 'r';
texte[7] = '\0';         // 0 final
```

*\* Le zéro final permet de délimiter la chaîne.*

✚ Le langage C++ propose un objet chaîne de caractère **string**, l'utilisation de tableaux de caractères restant valable.

### Initialisation

L'initialisation et la mise à jour d'une chaîne s'effectue élément par élément comme un tableau. On peut aussi initialiser directement une chaîne par affectation lors de sa déclaration, dans ce cas le zéro final sera rajouté automatiquement :

```
char texte[8] = "Bonjour";
```

*\* les doubles quotes sont les délimiteurs des chaînes de caractères*

### Lettres accentuées

Contrairement aux caractères **char** (codage ASCII), il est possible d'utiliser des lettres accentuées dans les chaînes de caractères (codage UTF-8\*), mais attention dans ce cas la longueur de la chaîne risque d'être plus grande que le nombre de caractères de la chaîne.

```
char c = 'é';           //refusé par le compilateur
char texte[20] = "été"   //accepté par le compilateur,
                        //mais la chaîne finale en UTF-8
                        //sera : C3 A9 74 C3 A9 00
                        //et aura pour longueur 5
```

*\* Cela n'est pas garanti sur tous les compilateurs.*

Cohérence : le codage UTF-8 ne contient aucun octet à 0, cela pour ne pas interférer avec la règle du 0 terminal des chaînes de caractères "C".

## La copie des chaines de caractères

C'est la fonction `strcpy()` qui permet la copie d'une chaine de caractères, attention à bien dimensionner la chaine destination :

```
#include <string.h>
char *strcpy (char *dest, const char *src);
```

- ❖ Retourne un pointeur sur la chaine destination (double emploi avec `chaine_destination`).

La variante `strncpy()` permet de limiter le nombre de caractères de la copie, attention dans ce cas il n'y aura peut-être pas de `"\0"` final à la fin des caractères recopiés.

```
char *strncpy(char *dest, const char *src, size_t nombre);
```

## La concaténation de chaines de caractères

C'est la fonction `strcat()` qui permet la concaténation de 2 chaines de caractères.

```
#include <string.h>
char *strcat(char *dest, const char *src);
```

La seconde chaine `src` est concaténée à la première `dest`, attention de dimensionner correctement le tableau `dest`.

- ❖ Retourne un pointeur sur la chaine concaténée (double emploi avec `dest`).

La variante `strncat()` permet de limiter le nombre de caractères concaténés de la seconde chaine :

```
char *strncat(char *dest, const char *src, size_t n);
```

Attention dans ce cas il n'y aura peut-être pas de `"\0"` final à la fin des caractères recopiés.

## La longueur des chaines de caractères

C'est la fonction `strlen()` qui permet de déterminer la longueur d'une chaine de caractères, le `"\0"` final n'est pas compris dans la longueur retournée :

```
#include <string.h>
size_t strlen(const char *str);
```

- ❖ Retourne la longueur de la chaine

## La comparaison de chaînes de caractères

C'est la fonction `strcmp()` qui permet de comparer 2 chaînes de caractères.

Il s'agit d'une comparaison lexicographique (dans l'ordre des codes des caractères, attention aux lettres accentuées) :

```
#include <string.h>
int strcmp(const char *str1, const char *str2);
```

- ❖ Retourne :
  - < 0 : str1 < (précède) str2
  - = 0 : str1 est strictement égale à str2
  - > 0 : str1 > (succède à) str2

La variante `strncmp()` limite la longueur de la comparaison.

```
int strncmp(const char *str1, const char *str2, size_t n);
```

## Recherche d'un caractère

C'est la fonction `strchr()` qui permet de rechercher un caractère dans une chaîne de caractères.

```
#include <string.h>
char *strchr(const char *str, int c) ;
```

- ❖ Retourne un pointeur sur la première position du caractère, `NULL` sinon.
  - \* *`string.h` introduit le symbole `NULL` qui est le pointeur nul (pointe sur l'adresse 0).*

La variante `strrchr()` effectue la recherche en commençant par la droite (right) de la chaîne.

## Recherche d'une sous-chaîne

C'est la fonction `strstr()` qui permet de rechercher une sous-chaîne dans une chaîne de caractères. La sous-chaîne est recherchée sans le caractère `"\0"` final :

```
#include <string.h>
char *strstr(const char *str, const char *substr);
```

- ❖ Retourne un pointeur sur la première position du caractère, `NULL` sinon
  - \* *`string.h` introduit le symbole `NULL` qui est le pointeur nul (pointe sur l'adresse 0).*

## La fonction sprintf()

La fonction `sprintf()` est identique à la fonction `printf()` (cf. chapitre "Le Dialogue Opérateur") sauf qu'elle ne va pas afficher le message dans la console mais l'affecter à une variable chaîne de caractères :

```
#include <stdio.h>
int fprintf(char *str, const char *format, ...);
```

Par exemple :

```
char nom[100] = "Emile";
int age = 34;
char message[200];
sprintf(message, "Bonjour %s vous avez %d ans", nom, age);
printf("%s\n", message);
```

## La fonction sscanf()

La fonction `sscanf()` est identique à la fonction `scanf()` (cf. chapitre "Le Dialogue Opérateur") sauf qu'elle ne récupère pas son message d'entrée à partir du clavier (fichier standard d'entrée) mais à partir d'une chaîne de caractères :

```
#include <stdio.h>
int sscanf(const char *str, const char *format, ...);
```

Par exemple :

```
int age;
char nom[100];
char message[100] = "Emile 34";
sscanf(message, "%s %d", nom, &age);
printf("Bonjour %s vous avez %i ans\n", nom, age);
```

## Les limitations de scanf() et sscanf()

### La saisie des caractères :

Les espaces sont des séparateurs, le format `"%s"` arrête la lecture au premier espace rencontré.

En saisie au clavier `scanf()` Pour contourner ce problème il faudrait utiliser la fonction `fgets()` qui lit dans le tableau de caractères `buffer` une ligne complète de caractères (jusqu'au Retour Chariot de l'utilisateur).

```
#include <stdio.h>
fgets(buffer, max_carac, stdin);
```

*\* `max_carac` limite le nombre de caractères lus pour des raisons de sécurité (non débordement de `buffer`)*

## La saisie des nombres :

Les fonctions `scanf()` et `sscanf()` ne récupèrent pas les erreurs de saisies (lettres au lieu de chiffres par exemple).

## *La manipulation des chaînes de caractères n'est pas le point fort du langage C*

Nous avons donc développé une fonction `lire_float()` pour la lecture rigoureuse des nombres flottants (cf. chapitre "Mini-Projet").

Vous pourrez facilement transformer cette fonction en une fonction `lire_int()` pour saisir un nombre entier.

- En C++, les fonctions `getline()` de lecture de ligne à la console et `istringstream()` de conversion de chaînes de caractères sont plus sécurisées

## Le séparateur décimal

Pour remplacer un caractère par un autre dans une chaîne de caractère, par exemple le point décimal en virgule (ou vice-versa) il va falloir parcourir la chaîne caractère par caractère :

```
while (*chaine) {
    if (*chaine == '.') {
        *chaine = ',';
    }
    chaine++;
}
```

## ■ Exercices

- La table ASCII :



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BELL	BS	HT	LF	VT	NP	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYNC	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	US

Ecrire un programme qui affiche les 7 lignes de caractères imprimables de la table ASCII (caractères de 20 hexa à 7F hexa).

- Créez les chaînes `h "Hello"` et `w "World"` puis :

- Concaténez `h` + un blanc + `w` dans une nouvelle chaîne `hw`
- Recherchez la position de `"World"` dans la chaîne `hw` et supprimez `"World"` de la chaîne `hw`
- Supprimez le blanc final de la chaîne `hw`
- Comparez les chaînes `hw` et `h`

## VI- VARIABLES COMPOSEES

### VI.1- Les Structures

Une structure est une collection de variables de même type ou de types différents, identifiées par un nom contextuel. Lors de la définition d'une structure on définit tous ses éléments, la structure ne pourra plus évoluer lors de l'exécution :

```
// Définition d'une structure "new_struct"
struct new_struct {                //contenant
    int nombre;                   //un entier nombre
    int tab[100];                 //et un tableau d'entiers tab
};                                //ici le ";" est obligatoire

// Déclaration d'une structure "ma_struct" de type "new_struct"
struct new_struct ma_struct;
```

On peut aussi déclarer une structure lors de la définition de cette structure :

```
struct new_struct {
    int nombre;
    int tab[100];
} ma_struct;
```

*\* C'est pourquoi le ";" est obligatoire après la définition d'une structure, que l'on déclare ou pas une structure lors de cette définition*

- ❖ En C on peut imbriquer une structure dans une autre structure
- ✚ Le langage C++ permet d'inclure des fonctions dans une structure, cela permettra de construire des objets.

#### Accès aux éléments

C'est l'opérateur "." qui symbolise l'appartenance de l'élément à une structure

```
ma_struct.nombre = 150; // affectation de l'élément nombre
```

#### Occupation mémoire

Les variables d'une structure sont rangées consécutivement en mémoire (dans l'ordre de définition de la structure), cependant les éléments seront alignés sur des mots mémoire par le compilateur, ce qui fait que l'image mémoire d'une même structure pourra changer d'une plateforme à l'autre en fonction de la longueur du mot mémoire.

```
struct new_structure {
    char initiale;
    int tab[100];
} s ;
sizeof(s); // -> 404,
           // le char a été complété par 3 octets vides
           // pour un alignement sur 32 bits
```

## VI.2- Les Unions

L'union correspond à une superposition de variables de même type ou de types différents en mémoire, on ne peut donc conserver qu'une seule de ces variables à la fois et la taille de l'union correspond à la taille du plus grand des types.

Les `union` sont généralement utilisées pour créer des fonctions polymorphes c'est à dire des fonctions qui réagissent différemment selon le type de leurs arguments d'entrée.

✚ Les fonctions polymorphes existent nativement en C++ mais pas en C.

```
#include <stdio.h>
union mon_union {
    int i;
    long l;
    char c;
};
sizeof (long);           // -> 8
sizeof (union mon_union); // -> 8
```

C'est l'opérateur `"."` qui symbolise l'appartenance de l'élément à une union.

```
mon_union.i = 5;
mon_union.l = 0; // "l" va écraser "i" qui est perdu.
```

## VI.3- Les Enumérations

Une variable de type `enum` est une variable dont on fournit la liste complète des valeurs (numériques) qui peuvent lui être attribuées :

```
enum jour {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI,
           DIMANCHE};
```

La déclaration ci-dessus définit à la fois la variable énumérable `jour` et les constantes `LUNDI ... DIMANCHE` lesquelles auront par défaut les valeurs respectives de `0` à `6` :

```
enum jour j = MARDI;
printf("jour %i \n", j); // -> 1
```

\* on peut définir la valeur des constantes numériques :

```
enum force {FAIBLE = 10, MOYENNE = 50, FORTE = 100};
```

❖ Les types `enum` sont généralement utilisés pour rendre plus lisibles les séquences de code comportant des variables à domaine de validité limité :

```
for (j = LUNDI; j <= DIMANCHE; j++) { ... };
```

## ■ Exercices

- Définissez un type de structure **dinosaure** comportant les éléments suivants :

**identifiant** : entier  
**nom** : chaîne de 50 caractères maximum  
**longueur** : entier, longueur indiquée en mètres  
**carnivore** : booléen, indique s'il est carnivore ou non  
**os** : entier, indique le nombre d'os du squelette  
**decouverte** : entier, année de la découverte

- Créez une structure de type **dinosaure** et de nom **dino1** correspondant à l'animal suivant : Identifiant 65473, stégosaurus, longueur 16m, non carnivore, 324 os, découvert en 1925. Affichez le contenu de la structure pour vérification.
- Créez un tableau de 10 structures de type **dinosaure**, recopiez dino1 dans le premier élément du tableau, pour les 9 autres on alimentera uniquement les identifiants : 1001, 1002 ... 1009. Affichez les 10 identifiants pour vérification.

## VII- LES POINTEURS

### Rappels

Une variable est stockée en mémoire, c'est l'opérateur "&" qui permet d'obtenir l'adresse mémoire<sup>8</sup> d'une variable :

```
int nombre
long adresse;
adresse = &nombre;
```

Un pointeur est une variable dont la valeur est l'adresse mémoire d'une autre variable.

```
int nombre = 12;
int *ptn;           // déclaration pointeur de type entier
ptn = &nombre;      // ptn pointe maintenant sur nombre
a = *ptn;           // * est l'opérateur d'indirection du
                    // pointeur pour accéder à la valeur
                    // de la variable sur laquelle il pointe
printf("%i", a);    // -> 12
```

*\* Les 2 syntaxes `int* ptn` et `int *ptn` sont équivalentes.*

Remarque : A priori une adresse mémoire n'a pas de type mais les pointeurs C sont typés, car cela va nous permettre de disposer d'une "l'arithmétique des pointeurs" (cf. plus loin).

Attention : Un pointeur déclaré mais non initialisé (sans variable pointée) aura de grandes chances de provoquer une faute applicative ou une erreur système lors de son utilisation.

### Recopie de pointeurs

Rappel : C'est l'opérateur d'affectation "=" qui permet d'affecter la valeur d'un pointeur à un autre pointeur :

```
int *ptn = &nombre;
int *ptn2 = ptn; // nombre possèdera 2 pointeurs
                // ptn et ptn2
```

### Pointeurs et tableaux

Le tableau est une variable particulière dans le sens où un tableau est une collection de variables du même type.

Le nom d'un tableau correspond à un pointeur sur le 1er élément du tableau et de même type que cet élément :

```
int tab[10];
tab[0] = 66;
printf("%i\n", *tab); // -> 66
```

---

<sup>8</sup> Il s'agit là d'une adresse mémoire logique liée à l'implémentation du programme en mémoire.

Le nom du tableau est une constante, si je souhaite un pointeur variable (cf "l'arithmétique des pointeurs plus loin") :

```
int *ptn = tab ; //second pointeur sur le tableau
printf("%i\n", *ptn); // -> 66
```

### Recopie de tableau

On ne peut pas recopier les tableaux en utilisant l'opérateur "=", car cela ne ferait que recopier les pointeurs des tableaux et non pas le contenu des tableaux.

La recopie de tableaux peut s'effectuer élément par élément à l'aide d'une boucle itérative, il est toutefois plus concis d'utiliser la fonction `memcpy()` (cf. chapitre "La Gestion de la Mémoire").

### Recopie de chaînes de caractères

Les chaînes de caractères sont des tableaux, le nom d'une chaîne de caractères correspond donc à un pointeur caractère sur le premier caractère de la chaîne.

```
char str[10] = "Bonjour";
printf("%c\n", *str); // -> "B"
```

C'est pourquoi on ne pas recopier les chaînes de caractères en utilisant l'opérateur d'affectation "=", il faut utiliser les fonctions spécifiques de recopie de chaînes : `strcpy()` ou `strncpy()` (cf. chapitre "Le Traitement des chaînes de caractères").

### Pointeurs et structures

Pour accéder aux éléments d'une structure à partir d'un pointeur sur cette structure il faudrait utiliser des parenthèses car l'opérateur d'appartenance "." est prioritaire par rapport à l'opérateur d'indirection "\*":

```
struct new_structure s;
int *ptn = &s; // pointeur sur la structure
s.nombre = 150;
int x = (*ptn).nombre; // -> 150
```

Cela rend l'écriture inélégante et pour cette raison un nouvel opérateur "->" a été défini pour accéder aux éléments d'une structure à partir d'un pointeur sur cette structure :

```
int x = ptn->nombre; // accès à l'élément nombre
// par pointeur sur structure ->
```

L'opérateur d'adressage quant à lui peut s'appliquer aussi bien à une structure qu'à l'un de ses éléments :

```
int *ptn = &s; // adresse de la structure en mémoire
int x = &s.nombre; // adresse de l'élément "s" en mémoire
```

### Pointeurs et fonctions

Cf. chapitre "les Fonctions".

## L'arithmétique des pointeurs

Le type pointeur est plus riche qu'un type entier long contenant une adresse mémoire, un pointeur est aussi un objet mathématique.

- Lorsqu'un pointeur pointe sur le premier élément d'un tableau, le fait d'incrémenter de 1 le pointeur va le faire pointer sur l'élément suivant, autrement dit le compilateur incrémentera le pointeur de la taille du type considéré.

```
int tab[10]= {2,4,8,16,32,64,128,256,512,1024};
int *ptn = tab;
printf("%i",*ptn);      // -> 2
ptn++;
printf("%i",*ptn);      // -> 4
```

- Vous pouvez donc parcourir un tableau à l'aide d'un pointeur (ce qui est plus performant que d'y accéder au travers de son l'index) :

```
int tab[100];
int *ptn = tab;
int i;
for (i = 0; i < 100; i++) {
    *ptn++ = 0;
}
```

- Vous pouvez également dans un tableau :
  - décrémenter de 1 les pointeurs "--" (passer à l'élément précédent)
  - ajouter ou retrancher un nombre entier aux pointeurs "+= 5", "--=4" (passer aux 5ème élément suivant ou au 4ème élément précédent)
  - soustraire un pointeur d'un autre : vous obtiendrez le nombre d'éléments entre ces 2 pointeurs.

- ✚ Le langage C++ introduit en plus des pointeurs la notion de "Référence" qui est un pointeur non modifiable et dont l'initialisation statique est obligatoire.

## Le Pointeur générique void

Le pointeur **void** est un pointeur générique (il peut pointer sur tout type de variables), pour son indirection il faudra donc obligatoirement utiliser l'opérateur **cast**, pour préciser le type de la variable pointée :

```
int i = 99;
void* ptn = &i;
printf ("%i", *((int*)ptn)) // -> 99
```

Un exemple classique d'utilisation du pointeur **void\*** est la fonction **malloc()** d'allocation mémoire qui renvoie un pointeur **void** (cf. chapitre "Gestion de la Mémoire").

```
int *ptn;
ptn = (int *) malloc (12 * sizeof(int));
```

Ci-dessus le programmeur souhaite utiliser l'espace mémoire pour y ranger 12 entiers, le pointeur **void** retour de la fonction **malloc()** a été "casté" en **int**.

### ■ Exercices

- 1) Créez un tableau de 10 **int** et vérifiez que les éléments soient bien rangés consécutivement en mémoire et sans trous.
- 2) Initialisez à zéro les éléments d'un tableau de 100x100 **int** en utilisant l'arithmétique des pointeurs.
- 3) Récupérez le tableau de 10 structures **dinosaure** créé précédemment :
  - a) En utilisant un pointeur sur le tableau, Mettez à jour la première structure (**dino1**) avec les éléments suivants longueur 18m, 307 os, découvert en 1943.
  - b) Incrémentez de 2 le pointeur et mettez à jour la troisième structure avec les éléments suivants : nom "Tyranosaurus", longueur 15m, carnivore, 456 os, découvert en 1939.

## VIII- LES FONCTIONS

### VIII.1- Définition de Fonction

Le langage C définit de la même façon les fonctions et les sous-programmes. Un sous-programme sera considéré comme une fonction sans paramètre de retour :

```
type_retour ma_fonction (type_1 param_1, ..., type_n param_n){  
    ...  
    // corps de la fonction  
    ...  
}
```

- Le type de chaque paramètre d'entrée doit être défini
- Le type de la fonction elle-même doit être défini c'est le type de la valeur de retour, on utilisera le pseudo-type **void** pour une fonction sans paramètre de retour.
- L'instruction **return** permet de quitter à tout moment une fonction pour revenir au programme appelant.
- Si la fonction retourne une valeur c'est l'instruction **return** qui permet la remontée de cette valeur : **return (valeur);**

❖ Attention en langage C on ne peut pas imbriquer une fonction dans une autre.

Exemples de définition de fonction :

```
int somme (int a, int b){  
    int c;  
    c = a + b;  
    return (c);  
}  
  
void affiche (char *texte){  
    printf("%s\n",texte);  
}
```

- Une fonction peut ne pas posséder de paramètres d'entrée.
- Si la fonction ne retourne pas de valeur (fonction **void**) l'instruction **return** n'est pas obligatoire, le retour se fera automatiquement à la fin du code de la fonction.

✚ En C++ si une fonction ne possède pas de paramètres d'entrée elle doit déclarer en entrée le pseudo-paramètre **void**.

### Appel de fonction

Exemple d'appel d'une fonction avec retour :

```
int a = 2;  
int b = 3;  
int c;  
c = somme (a, b);  
printf("%i\n",c);
```

❖ Comme nous le constatons ci-dessus le retour d'une fonction peut-être directement affecté à une variable de même type.

Exemple d'appel d'une fonction sans retour :

```
char texte[10] = "Bonjour"
affiche (texte);
```

## Déclaration de fonction

Une fonction doit impérativement être définie avant d'être appelée, si cela n'est pas le cas, il faut déclarer le prototype de la fonction avant l'appel.

Le prototype d'une fonction correspond à son type, son nom et l'ordre + les types de ses paramètres (le nom des paramètres est optionnel).

```
type ma_fonction (type1 nom1, ..., type n nom n);
ou
type ma_fonction (type1, ..., type n);
```

exemple :

```
int somme (int, int);
void affiche (char*);
```

## Fonction externe

Un programme source C est composé d'un ou plusieurs fichiers sources (cf. chapitre "Structure d'un Programme C").

Un fichier source pourra accéder à une fonction externe, c'est-à-dire une fonction d'un autre fichier source du programme, à condition de définir son prototype et à condition que cette fonction ne soit pas qualifiée de **static** (auquel cas elle ne serait accessible que de son propre fichier source).

- ❖ Par convention dans un programme multi-sources on écrira les prototypes des fonctions de chaque source (sauf les fonctions **static**) dans un fichier de même nom avec l'extension **".h"**, ainsi en incluant ce fichier dans un autre source ce dernier pourra accéder à ces fonctions externes.

## Le Passage des Arguments

Les variables simples et composées sont passées par recopie, si une fonction modifie un de ces arguments cela n'aura aucun impact sur la variable correspondante de la fonction appelante (pas d'effet de bord).

Les tableaux sont passés par recopie de leur pointeur, donc la fonction pourra utiliser ce pointeur pour modifier le contenu du tableau dans le programme appelant.

Le programmeur peut également utiliser un pointeur pour transmettre une variable simple ou composée, dans ce cas la fonction va pouvoir modifier la valeur de la variable pointée dans le programme appelant.

Dans le cas de l'argument retour la variable en affectation dans le programme appelant sera modifiée qu'il s'agisse d'une valeur directe ou d'un pointeur.

## Le Transtypage

Lors d'un appel d'une fonction, les paramètres numériques seront automatiquement convertis dans les types des paramètres d'entrée de la fonction.

```
int modulo (int a, int b) {
    return (a%b);
}
...
float a = 12;
int b = 5;
printf("%i",modulo(a,b));    //a sera convertit en int
                             //-> 2
```

## Les Pointeurs de Fonction

Les pointeurs peuvent également pointer sur des fonctions. Tout comme les tableaux, les noms des fonctions constituent des pointeurs sur ces fonctions :

```
int func() {int, int}

int (*fp)(int, int); //Déclaration d'un pointeur de fonction
                    //de type int, avec 2 int en paramètres

fp = &func;         //initialisation du pointeur
int a = (*fp)(x, y); //indirection pour appel de la fonction
                    //func()
```

## Les Fonctions Variadiques

Une fonction variadique (variadic function) est une fonction qui admet un nombre variable de paramètres, en fait il s'agit d'une fonction avec une liste de paramètres définis + une liste complémentaire de paramètres dont le nombre et les types ne sont pas définis.

La prototype d'une fonction variadique est le suivant :

```
type fonction_variadique (<liste des paramètres définis>, ... );
```

- la liste des paramètres définis ne doit pas être vide (au moins un paramètre),
- la liste des paramètres non définis est symbolisée par le symbole "..." et doit obligatoirement se trouver après la liste des paramètres définis.

*\* Le compilateur ne peut vérifier ni les types ni le nombre d'arguments de la liste non définie, une erreur sur ces types ou ce nombre peut conduire à des résultats imprévisibles.*

La fonction `printf()` (cf. chapitre "Les chaînes de caractères") est un exemple de fonction variadique dont le prototype est : `printf (char*, ... );`

----- à compléter, création de fonctions variadiques -----

## VIII.2- Bibliothèques de Fonctions

### Fonctions Standard C

Le langage C fournit un grand nombre de fonctions réparties dans des bibliothèques.

Chaque bibliothèque de fonction C possède son fichier include `".h"` qui définit les prototypes de ses fonctions et éventuellement un certain nombre de symboles utilisés par ces fonctions (cf. chapitre "Le Préprocesseur").

Exemple : Rappel de l'en-tête du programme `"hello.c"` :

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

Le fichier `stdio.h` contient le prototype de la fonction `printf()` qui est :

```
int printf(const char *format, ...);
```

*\* Cette fonction est particulière, elle admet un nombre variable d'arguments (cf. "fonctions variadiques" ci-dessous).*

- La liste des Bibliothèques Standard C figure à la fin du document.

### Primitives Système

Le système d'exploitation de la plateforme de travail possède également des bibliothèques de fonctions système "Primitives systèmes", ou API (Application Programming Interface) permettant aux programmes utilisateurs d'accéder aux ressources système : fichiers, périphériques, réseau ...

C'est le cas de la bibliothèque `<sys/syscall.h>` figurant au chapitre "La Programmation Multithread".

Cependant l'utilisation de ces bibliothèques est spécifique de la plateforme en question.

Lorsque des fonctionnalités sont accessibles à la fois par les bibliothèques standards C et par les primitives système, il est préférable d'utiliser les fonctions standards C pour des raisons de portabilité.

### ■ Exercices

- 1) Ecrivez une fonction qui retourne la valeur max des 2 entiers en argument d'entrée.  
+ un programme mettant en action cette fonction
- 2) Reprendre le programme "Intérêts Composés" :
  - Un programme principal permettra de saisir les données : somme, taux et nombre d'années.
  - Une fonction `interet()` calculera les intérêts et les affichera.

## MINI-PROJET

A titre de projet, ci-dessous un programme permettant de calculer l'Indice de Masse Corporelle d'une personne.

Fichier source "IMC" :

```
#include <stdio.h>

/* Programme IMC */

int main () {
    float poids,taille,masse;

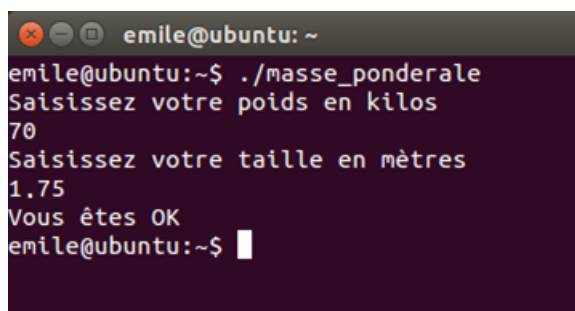
    // Saisie des informations
    printf ("Saisissez votre poids en kilos\n");
    scanf("%f",&poids);
    printf ("Saisissez votre taille en metres\n");
    scanf("%f",&taille);

    // Calcul de la masse ponderale
    masse=poids/(taille*taille);

    // Affichage du resultat
    if (masse < 18.4) printf ("Vous êtes trop maigre\n");
    else if (masse > 25) printf ("Vous êtes trop gros\n");
    else printf ("Vous êtes équilibré\n");

    // Retour
    return (0);
}
```

Résultat :



```
emile@ubuntu: ~
emile@ubuntu:~$ ./masse_ponderale
Saisissez votre poids en kilos
70
Saisissez votre taille en mètres
1.75
Vous êtes OK
emile@ubuntu:~$
```

Rappel :

La fonction `scanf()` de lecture / conversion de chaîne de caractères ne récupère pas de façon fine les erreurs de saisies.

C'est pourquoi nous n'avons pas traité les cas d'erreurs de saisie ci-dessus. Nous allons plutôt développer une fonction spécifique `lire_float()` pour contrôler la saisie des nombres flottants et on en profitera pour gérer la virgule décimale.

```

#include <stdio.h>

/*
 * Lecture d'un nombre flottant
 * en entrée : pointeur sur le nombre flottant retour
 * en sortie :
 *   0 si OK
 *  -1 si saisie incorrecte (car. non numér. ou non virgule)
 *  -2 si nombre trop long (> 100 caractères)
 */

int lire_float(float* nombre) {
    char texte[101];
    int c,i;
    int virgule = 0;

    // Lecture caractère par caractère
    for (i = 0; i <= 100 ; i++) {

        // Contrôle longueur
        if (i == 100) return(-2); // plus de 100 caractères

        // lecture d'un caractère
        c=getchar();
        if (c == '\n') break; // fin de saisie (fin de ligne)

        // controle de la valeur numérique
        // et remplacement de la virgule décimale par un point
        if ((c >= '0') && (c <= '9')) texte[i] = c;
        else if ((c == ',') && (!virgule)) {
            texte[i] = '.';
            virgule = 1 ;
        } else return (-1); // Faute caractère non numérique
    }
    texte[i] = 0; // on ferme la chaine

    // Conversion en flottant et retour
    sscanf(texte,"%f",nombre);
    return(0);
}

```

- Nous avons utilisé la fonction `getchar()` qui lit caractère par caractère, afin de filtrer les erreurs et de présenter une chaîne numérique consistante à la fonction `sscanf()`. (cf. chapitre "Le Traitement des Chaînes de Caractère").

Et voici le programme principal adapté :

```

#include <stdio.h>

int lire_float(float* nombre);

/* Programme IMC */

int main () {
    float poids,taille,masse;

    // Saisie des informations
    printf ("Saisissez votre poids en kilos : ");
    if (lire_float(&poids) != 0) {
        printf("Faute saisie poids\n");
    }
}

```

```

        return(-1);
    }
    printf ("Saisissez votre taille en metres : ");
    if (lire_float(&taille) != 0) {
        printf("Faute saisie taille\n");
        return(-1);
    }

    // Calcul de la masse ponderale
    masse=poids/(taille*taille);

    // Affichage du resultat
    if (masse < 18.4) printf ("Vous êtes trop maigre\n");
    else if (masse > 25) printf ("Vous êtes trop gros\n");
    else printf ("Vous êtes équilibré\n");

    // Retour
    return (0);
}

```

## IX- LA PORTEE DES VARIABLES

Rappel : Toute variable doit être déclarée avant d'être utilisée.

### Variables globales

Dans un fichier source les variables globales sont celles déclarées à l'extérieur des fonctions, elles sont accessibles à toutes les fonctions de ce fichier. Les variables globales sont généralement<sup>9</sup> déclarées en tête du fichier parce que leur visibilité commence à partir de leur déclaration.

### Variables locales

Les variables locales sont celles déclarées à l'intérieur d'une fonction ou d'un bloc d'instruction, elles sont accessibles uniquement à l'intérieur de cette fonction ou de ce bloc.

Les variables locales ont par défaut le qualificatif **auto** (elles sont créées et détruites automatiquement à chaque exécution de la fonction ou du bloc d'instructions), pour rendre permanente (mais non pas globale) une variable locale il faut utiliser le qualificatif **static**.

#### Attention :

- Une variable locale de même nom qu'une variable globale, masquera cette variable globale.
- Dans le cas de blocs d'instruction imbriqués les variables locales d'un bloc sont accessibles dans leur propre bloc et également dans les blocs imbriqués.

### Variables externes

Un programme source C est composé d'un ou plusieurs fichiers sources (cf. chapitre "Structure d'un Programme C").

Un fichier source pourra accéder à une variable externe c'est-à-dire une variable globale d'un autre fichier source à la condition d'utiliser la qualificatif **extern** et à la condition que la variable en question ne soit pas qualifiée de **static**, auquel cas elle ne serait accessible que de son propre fichier source :

```
extern int montant; // montant est une variable globale
                    // déclarée dans un autre fichier source
                    // sans le qualificatif static
```

- ❖ Dans un fichier source une variable globale ne peut pas porter le même nom qu'une variable déclarée en **extern**.

---

<sup>9</sup> Cela était obligatoire dans les anciennes versions du compilateur C.

## X- LA REDEFINITION DES TYPES

L'instruction `typedef` permet de renommer un type de variables, cela est appréciable dans le cas de types complexes et cela peut être utile aussi pour la portabilité des programmes :

```
typedef <ancien nom du type> <nouveau nom du type>
```

- Par exemple pour une structure :

```
typedef struct new_structure {
    int nombre;
    int tab[100];
} newstr;
```

Désormais `newstr` est un alias du type "`struct new_structure`", et pour créer une nouvelle structure de ce type :

```
newstr ma_structure;
```

- Par exemple pour fixer la taille d'un entier à 64 bits pour des raisons de portabilité, on définit un nouveau type `INT` en renommant les types correspondants :

```
typedef int INT;           // plateforme avec int sur 64 bits

typedef long INT; // plateforme avec int sur 32 bits
```

### ■ Exercice

Créez un nouveau type `dino` correspondant à la structure `dinosaure` définie plus haut.

```
typedef struct dinosaure {
    int identifiant;
    char nom[50];
    int longueur;
    int carnivore;
    int os;
    int decouverte;
} dino ;
```

## XI- LE PREPROCESSEUR

Le compilateur C dispose d'un métalangage lequel est interprété dans une pré-phase de compilation que l'on appelle le préprocesseur.

Une directive du préprocesseur doit être préfixée par un caractère `"#"` et se trouver seule sur une ligne.

### Directive `#include`

Nous avons déjà vu cette directive qui permet d'inclure un fichier source dans un autre fichier source, précisons que :

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

Sous cette forme le préprocesseur va chercher le fichier dans les répertoires des bibliothèques C.

```
#include "mon_include.h"
```

Sous cette forme le préprocesseur va chercher le fichier dans le même répertoire que le fichier source.

### Directive `#define`

La directive `#define` permet de définir un symbole :

Exemple : `#define PI 3.14`

A chaque fois que la chaîne `PI` apparaîtra dans une expression numérique, elle sera remplacée par le préprocesseur par la chaîne de caractères `3.14`. Par convention ces symboles sont en majuscules.

Exemple : `#define DEBUG`

Ci-dessus on définit un symbole sans lui affecter de valeur.

On peut également définir un symbole sans valeur lors de la compilation à l'aide de l'option `"D"` :

```
gcc -DDEBUG mon_programme.c
```

*\* Attention si la valeur d'un symbole est définie elle doit être syntaxiquement correcte.*

## Directive #if

Cette directive permet la compilation conditionnelle :

```
#if condition-1
    Code-correspondant
#endif
```

Le code correspondant ne sera compilé (et donc intégré à l'exécutable) que si la condition est Vraie.

La condition est une expression logique.

On peut introduire une alternative avec **#else**.

On peut enchaîner les conditions avec **#elif**.

## Directive #ifdef

Similaire à **#if** sauf que la condition consiste en l'existence d'un symbole :

```
#ifdef DEBUG
    Code-correspondant
#endif
```

Pour que le code correspondant s'exécute il suffit de définir **DEBUG** (même sans valeur) auparavant.

## Directive #ifndef

**#ifndef** est la directive inverse de **#ifdef** elle est valide si un symbole n'existe pas.

Cette directive permet d'éviter les cycles infinis. Supposons qu'un fichier include **x.h** inclue lui-même un fichier **y.h** et que ce dernier inclue à son tour **x.h** ou comme cela serait plus probable que **y.h** inclue **z.h** lequel inclue **x.h**. Nous allons nous retrouver devant un cycle infini et le compilateur va finir par se planter par manque de mémoire.

Pour éviter cela il est conseillé de mettre systématiquement au début des fichiers include la directive **#ifndef** afin d'empêcher les inclusions multiples :

```
Fichier x.h :
#ifndef x
    #define x
    ...
    ...
    ...
#endif
```

Ainsi le fichier include **x.h** ne pourra être inclus qu'une seule fois dans une chaîne de programmes

## La Portabilité

Les directives de compilation conditionnelles sont très utiles pour résoudre les problèmes de portabilité :

```
#define LINUX 1
#define WINDOWS 2
#define MACOS 3
#if (MACHINE == LINUX)
    <code spécifique Linux>
#elif (MACHINE == WINDOWS)
    <code spécifique Windows>
#elif (MACHINE == MACOS)
    <code spécifique Mac/Os>
#endif
```

il suffira alors de toucher à une seule ligne du programme :

```
#define MACHINE WINDOWS
```

Pour compiler la version adaptée à Windows.

## XII- STRUCTURE D'UN PROGRAMME "C"

Un programme C est composé d'un module principal et d'éventuels modules secondaires.

### Les modules

Un module correspond à une unité élémentaire de compilation, il comprend un fichier source (extension `".c"`) accompagné de 0 ou `n` fichiers "include" (extension `".h"`).

### La fonction main

Un des modules du programme doit comporter une fonction de nom `main()`, c'est le module principal. La fonction `main()` est obligatoire c'est elle qui va lancer le programme.

La fonction `main()` pourrait être sans retour (de type `void`), mais sous les systèmes Unix-like elle doit être déclarée `int`, la valeur retour servant de compte rendu d'exécution : un compte rendu à 0 signifie "Exécution correcte", sinon la valeur retour comportera un code d'erreur d'exécution.

La fonction main peut comporter les paramètres (`argc`, `*argv[]`) :

```
int main (argc, *argv[]) { ... }
```

Il s'agit des arguments du programme, arguments qui seront transmis au programme par le système d'exploitation au moment de son lancement :

- `argc` est le nombre de paramètres
- `argv[]` est un tableau de pointeurs qui pointent sur les arguments en entrée :
  - `argv[0]` pointe sur le nom du programme
  - `argv[1]` pointe sur le premier argument
  - ...
  - `argv[argc-1]` pointe sur le dernier argument

### Fichier source

Les fichiers source contiennent le code du programme : Déclarations diverses, variables globales et fonctions.

### Fichiers include

Les fichiers "include" contiennent les prototypes des fonctions externes, ainsi que la définition d'un certain nombre de symboles (cf. chapitre "Le Préprocesseur") relatifs à ces fonctions externes.

Il faut lier chaque fichier "include" au fichier source en utilisant la directive `#include` en tête du fichier source :

```
#include "toto.h"
```

Il serait possible de déclarer les prototypes externes en tête du fichier source lui-même, mais l'usage veut qu'on les déclare dans un fichier include :

- Lorsque l'on souhaite modulariser correctement un programme C, pour chaque fichier source on crée un fichier "include" (portant le même nom hormis l'extension), lequel va contenir les prototypes de ses fonctions internes.

- Chaque fichier source inclura son propre fichier "include" pour qu'il puisse utiliser ses propres fonctions sans se soucier de leur position dans le code.

- Pour accéder aux fonctions d'un autre fichier source il suffira d'inclure le fichier "include" correspondant.

*On se rapproche ainsi de la modularité du modèle objet.*

## ■ Exercices

Reprendre le programme intérêts composés, la version avec la fonction `interet()`.

- Créez un programme avec 2 fichiers source :
  - `principal.c` programme principal faisant appel à la fonction `interet()`.
  - `interet.c` contenant la fonction `interet()`.
- Créez le fichier prototype `interet.h` de la fonction `interet()`, le programme principal devra inclure ce fichier prototype.
- Compilez votre chaine de programmes :

```
gcc principal.c interet.c -o principal
```

### XIII- LA GESTION DE LA MEMOIRE

Une déclaration de variable implique l'allocation de l'espace mémoire correspondant, sauf si la déclaration est précédée du mot clef **extern** (cf. "La portée des variables").

Rappel : Pour obtenir l'adresse mémoire d'une variable il faut utiliser l'opérateur **&**, sauf dans le cas particulier des tableaux :

```
int nombre;  
printf("%lx",&nombre);          // -> 61ff1c
```

```
int tab[100];  
printf("%lx",tab);              // -> 61fd90
```

identique à : `printf("%lx",&tab[0]);`

Rappel : Pour connaître la place mémoire occupée par une variable ou un tableau de variables il faut utiliser l'opérateur **sizeof** appliquée à la variable ou à son type :

```
int a;  
printf("%i",sizeof a);          // -> 4  
printf("%i",sizeof (int));      // -> 4  
  
int tab[100];  
printf("%i",sizeof tab);        // -> 400  
printf("%i",sizeof (int [100])); // -> 400
```

#### XIII.1- Espace Mémoire du Programme

L'exécution d'un programme C correspond à l'exécution d'un processus. Tout processus possède un espace mémoire privé et protégé :

Zone mémoire	Contenu
Environnement	- Variables système - Variables d'environnement - Arguments du processus.
Heap (Tas)	Données globales dynamiques
Stack (Pile) ↑	- Données locales dynamiques - Arguments des fonctions - Contextes d'exécution des fonctions
Bss <sup>10</sup>	Données statiques non initialisées
Data	Données statiques initialisées
Text	- Code source compilé en langage machine - Constantes

- ❖ Si le processus est multithread toutes les zones mémoire du processus seront partagées à l'exception de la pile : Chaque thread aura droit à sa propre pile.

---

<sup>10</sup> Block Started by Symbol)

## Variables globales statiques

"Statique" signifiant que ces données ont été déclarées lors de la compilation. Elles sont stockées dans l'espace de données statiques du processus : Dans la zone **Data** si initialisées, dans la zone **Bss** autrement.

- ❖ Sur les systèmes Unix-like la commande `size` permet de connaître la taille des zones variables statiques d'un exécutable :

```
size a.out

text data bss dec  hex filename
1073 560  12  1641 669 a.out
```

## Variables locales

Elles sont stockées dans la pile **stack** du processus sauf si elles sont déclarées **static** auquel cas elles seront dans la zone **Data** si initialisées et dans la zone **Bss** autrement.

## Variables dynamiques

Les zones de mémoire allouées dynamiquement (lors de l'exécution du programme) sont stockées dans le "Tas".

Le Tas est extensible, théoriquement tant qu'il y a de la mémoire disponible dans le système.

- ✚ Les variables dynamiques C++ (instanciation d'objets) sont stockées dans le tas.

## La pile

Pour s'exécuter une fonction a besoin d'espace mémoire pour stocker son adresse retour, ses arguments et ses variables locales. Pour cela le langage "C" utilise une Pile dite pile d'exécution, les contextes des fonctions sont dynamiquement empilés et dépilés au fur et à mesure des appels et retour (Solution adoptée par la quasi-totalité des langages de programmation).

Rappel : Si le programme est multithread alors il y aura une pile d'exécution par thread.

- ❖ La taille de la pile d'exécution est fixée par le système. Sur les systèmes Unix-like la commande `ulimit -s` permet de connaître la taille de la pile d'exécution :

```
ulimit -s
8192
```

## Les qualificateurs "register" et "volatile"

Une variable qualifiée par **register** sera stockée dans un registre du microprocesseur ou maintenue le plus longtemps possible dans un tel registre afin d'en optimiser les accès.

Au contraire si l'on qualifie une variable de **volatile** on demande au compilateur de respecter la position mémoire de cette variable car elle est susceptible d'être manipulée par des moyens extérieurs au programme (registres d'accès aux périphériques par exemple).

### ***XIII.2- Allocation Dynamique de Mémoire***

- C'est la fonction `malloc()` qui permet d'allouer de l'espace mémoire dynamiquement :

```
#include <stdlib.h>
void *malloc(size_t size);
```

Retourne un pointeur `void` sur l'espace mémoire alloué et retourne `NULL`<sup>11</sup> si plus d'espace mémoire de disponible.

Pour accéder à l'espace mémoire ainsi alloué il faut créer un pointeur du type souhaité et "caster" l'appel à la fonction, comme déjà vu au chapitre "Compléments Pointeurs" :

```
int *ptn;
ptn = (int *) malloc (12 * sizeof(int));
```

Ensuite on peut utiliser le pointeur comme un tableau :

```
for (i = 0; i < 10; i++) ptn[i] = i;
printf("%i\n",ptn[2]); // -> 2
```

- Pour redimensionner l'espace ainsi alloué :

```
ptn = (int *) realloc (ptn, 20 * sizeof(int));
```

Le prototype est :

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- ❖ Retourne un pointeur `void` sur l'espace mémoire alloué et retourne `NULL` si plus d'espace mémoire de disponible.

- Et pour libérer l'espace ainsi alloué : `free (ptn);`

Le prototype est :

```
#include <stdlib.h>
void free (void *ptr);
```

### ***XIII.3- Recopie de Zones Mémoire***

C'est la fonction `memcpy()` qui permet de recopier des zones mémoires :

```
#include <string.h>
void *memcpy(void *str1, const void *str2, size_t n);
```

---

<sup>11</sup> Définit dans `<stdlib.h>`

Par exemple pour recopier deux tableaux de même type et de même dimension :

```
#include <stdio.h>
#include <string.h>
void main () {
    int tab1[100];
    int tab2[100];
    memcpy (ptn2,ptn1,sizeof(tab1));    //recopier tab1 dans tab2
}
```

Et pour sauvegarder une structure dans un tableau d'octets :

```
#include <stdio.h>
#include <string.h>
void main () {

    struct new_structure {
        int nombre;
        int tab[100];
    };
    struct new_structure s1;
    unsigned char taboct[sizeof(s1)];
    ...                                // remplir la structure s1
    void *ptn1 = &s1;
    unsigned char *ptn = taboct;
    memcpy (ptn,ptn1,sizeof(s1));    //recopier s1 dans taboct
}
```

## ■ Exercices

1) Recopie de zones mémoire :

- a) Effectuez une allocation dynamique de 10 nombres entiers
- b) Rangez les 10 premières puissances de 2 dans ces entiers
- c) Effectuez une nouvelle allocation dynamique de 10 nombres entiers
- d) Recopiez la première zone d'allocation sur la seconde
- e) Affichez les valeurs de la seconde zone d'allocation

## XIV- LE TRAITEMENT DES ERREURS

On distingue les erreurs applicatives des erreurs système.

- Les erreurs applicatives sont soit des erreurs produites par l'application elle-même, soit des erreurs provoquées par des appels aux fonctions des bibliothèques C ou aux des bibliothèques Système (API).

Bien entendu l'application doit ou traiter ces erreurs, ou présenter des messages d'erreurs clairs aux utilisateurs.

- Les erreurs système sont provoquées par les divisions par zéro, les instructions illégales, les débordements mémoire, les débordements de buffer (ou tableaux) ou de pointeurs.

Dans les 3 premier cas les erreurs système vont provoquer un arrêt immédiat et brutal du programme par le système d'exploitation avec un message d'erreur en anglais peu convivial.

Le débordement de buffer ou de pointeur est plus sournois : il ne va provoquer d'arrêt du système que si cela impliquait un débordement mémoire, autrement cette erreur va aboutir à un comportement imprévisible du programme sans aucun message d'avertissement.

Bien entendu l'application devrait récupérer ces erreurs – cela n'est pas toujours évident comme nous allons le voir - et arrêter l'application après avoir affiché des messages d'erreurs clairs.

- ✚ Le langage C++ possède un mécanisme standard de gestion des exceptions. Ce mécanisme (`setjmp` et `longjmp`) pourrait être implémenté en langage C mais cela ne sera pas traité dans ce document.

### XIV.1- Les Fonctions des Bibliothèques C et Système

Par convention les fonctions bibliothèque C et les fonctions primitives système doivent retourner une valeur positive ou nulle si l'exécution s'est déroulée correctement et autrement retourner un code d'erreur négatif.

Une minorité de fonctions ne respectent pas cette règle pour des raisons diverses, cela sera précisé dans le manuel d'utilisation de la fonction en question.

En parallèle, les fonctions C et les primitives système vont positionner en cas d'erreur un code d'erreur  $\geq 0$  dans la variable globale `errno` définie dans `<errno.h>`.

\* Attention la variable `errno` n'est pas remise à zéro automatiquement.

Exemple tentative d'ouverture du fichier `toto.txt` :

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    FILE *fp;
    fp = fopen("toto.txt", "r");
    if( fp == NULL ) {
        perror("Faute: "); //-> Faute: No such file or directory
        return (1);
    }
}
```

```

    }
    fclose(fp);
}

```

*\* La variable **errno** n'a pas été initialisée à zéro avant l'appel à **fopen()**, c'est ici inutile car **if(fp == NULL)** nous assure qu'il y a bien eu faute.*

*\*\* La fonction **perror()** va permettre d'afficher un message applicatif + le message d'erreur correspondant au code **errno** :*

#### **XIV.2- Les Fonctions de l'Application**

Pour les fonctions internes à l'application, il est conseillé de mettre en place une politique analogue à celle des bibliothèques C et Système : Retour positif **0** si OK, code d'erreur négatif sinon.

En particulier votre fonction **main()** devrait retourner **0** en cas de réussite et un code de faute négatif sinon. Pour cela vous devez utiliser les fonctions de sortie **return()** (dans le **main()**) ou **exit()** ou **abort()**, autrement le retour serait à **0** par défaut (si le type du **main()** est **int**).

*\* L'instruction **return** (dans le **main()**) et la fonction **exit()** sont pratiquement identiques, dans les 2 cas on libère les ressources de l'application (dont la mémoire) avant de la quitter.*

*La fonction **abort()** est destinée à la mise au point du programme, elle laisse la mémoire et les autres ressources en l'état au moment de quitter.*

- ❖ Vous pouvez utiliser la variable **errno** pour y positionner un code d'erreur.
- ❖ Vous pouvez utiliser les symboles compilateur C "**\_\_FILE\_\_**" et "**\_\_LINE\_\_**" pour préciser le fichier source et le numéro de ligne dans vos messages d'erreur :

```

Printf ("Fichier %s, ligne %i, faute xxx\n", __FILE__, __LINE__);

```

### XII.3- Les Erreurs Système

Ce sont les divisions par zéro, les débordements mémoire, les instructions illégales et les débordement buffer/pointeurs.

Ces erreurs vont provoquer un arrêt immédiat et brutal du programme par le système d'exploitation, à l'exception des erreurs de débordement de buffer/pointeur qui pourront se comporter de façon plus sournoise.

Sur les systèmes Unix-like le système d'exploitation envoie un signal au programme avant l'arrêt, il faut donc intercepter ce signal afin de stopper le programme de façon plus élégante. Voici les signaux système en question :

Signal		Signification
<b>SIGFPE</b>	Signal Floating-Point Exception	Débordement de capacité de calcul
<b>SIGILL</b>	Signal Illegal Instruction	Instruction illégale (généralement code corrompu)
<b>SIGSEGV</b>	Signal Segmentation Violation	Débordement de mémoire
<b>SIGBUS</b>	Bus error	Accès mémoire impossible

C'est l'appel système `signal()` qui permet d'intercepter le signal et de donner la main au handler correspondant : `void(*handler)(int)` .

Il faut donc activer la fonction `signal()` au début du `main()` , pour protéger tout le programme.

#### Cas de la division par zéro

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

void handler (int param){
    printf ("Division par zero\n");
    exit (1);                // code de retour 1
}

int main (){
    int a = 10;
    int b = 0;
    signal (SIGFPE, handler); // Intercepte le signal SIGFPE
                                // + active le handler correspondant
    a = a/b;
    return (0);
}
```

#### Prévention:

On pourrait tester systématiquement la valeur du diviseur avant toute division :

```
if ( b != 0) a = a/b;
else goto faute_div_0;
```

## Cas du débordement mémoire

Un débordement mémoire peut être provoqué par un débordement de tableau y compris les débordements de chaînes de caractères.

Un débordement mémoire peut être provoqué par un pointeur mal initialisé ou mal repositionné.

Un débordement mémoire peut également être provoqué par un débordement de pile.

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

void handler (int param){
    printf ("Débordement mémoire\n");
    exit (1);                // code de retour 1
}

int main ()
{
    int tab[100];
    signal (SIGSEGV, handler); // Intercepte le signal SIGSEGV
                                // et active le handler correspondant
    tab[1000] = 0;             // Débordement index
    return (0);               // code de retour 0
}
```

Mais en fait cette interception ne fonctionne que si le débordement de buffer (index hors du tableau) implique un débordement de mémoire. Pour le tester, vous allez devoir multiplier par **10** l'index du tableau `tab` jusqu'à provoquer l'erreur.

Par précaution il faudrait aussi intercepter le signal **SIGBUS**.

## Cas des instructions illégales

Le problème complexe, en ce qui concerne les instructions illégales, on pourrait intercepter le signal **SIGILL**, mais l'appel à la fonction `signal()` lui-même pourrait être corrompu, le système d'exploitation ou la plateforme matérielle pourraient être corrompus, il faut donc utiliser un handler le plus simple possible.

### Prévention

Vérification de l'intégrité du code avant de lancer le programme (check-sum, empreinte numérique).

## Cas particulier des débordements de pile

Un débordement de la pile d'exécution provoquera un signal de débordement mémoire : **SIGSEGV** ou **SIGBUS**.

La fonction handler du signal utilise la pile noyau et est donc protégée, mais par contre dans la fonction handler il ne faudra utiliser aucune instruction utilisant la pile d'exécution ou alors basculer vers une nouvelle pile d'exécution.

----- à compléter, basculement de pile avec setjmp et longjmp -----

### Prévention

Un débordement de pile peut être provoqué par :

- une allocation de variables locales trop grande,
- un débordement de buffer local,
- un pointeur local non ou mal initialisé.
- une récursivité trop profonde,
- une pile sous-dimensionnée.

### **Cas des erreurs système sournoises**

#### Rappel :

- débordement de buffer global ou local ne provoquant pas de débordement mémoire.
- pointeur mal initialisé ne provoquant pas de débordement mémoire.

#### Prévention :

C'est la prévention la seule façon de contrer ce genre d'erreurs.

Pour les tableaux on peut tester la valeur de l'index avant tout accès, simplement cela risque de se révéler très lourd.

Pour les pointeurs c'est par relecture de code (manuel ou automatisé) que l'on peut vérifier les initialisations et les réinitialisations. En cas d'utilisation de l'arithmétique des pointeurs il faudrait tester la valeur résultante avant tout accès à un tableau et comme précédemment cela risque de se révéler très lourd.

Pour les chaînes de caractères il faut utiliser les fonctions qui limitent le nombre de caractères comme `strncpy()`.

De même pour les saisies de chaînes de caractères il faut utiliser de préférence les fonctions qui limitent le nombre de caractères comme `fgets()`.

- ✚ Le langage C++ possède la notion de "Références", c'est-à-dire de pointeurs sur des variables que l'on doit obligatoirement initialiser lors de leur déclaration et qui ne sont plus modifiables par la suite.

### Alternative :

- Utiliser une fonction `assert()` qui provoque un arrêt du programme si la condition n'est pas respectée. Cela est également lourd mais il est préférable d'avoir cet arrêt plutôt qu'un comportement fantaisiste sans aucun signal d'avertissement.

```
#include <assert.h>
#include <stdio.h>
int main () {
    int tab[100];
    int index;

    ...
    assert ((index >= 0) && (index < 100)); // arrêt du programme
    ...                                     // si condition fausse
}
```

## ■ Exercice

Soit le programme suivant :

`init_pile()` permet de créer une pile de 10 entiers.

`push()` permet d'empiler un entier

`pop()` permet de dépiler un entier

```
#include <stdio.h>
#include <stdlib.h>
int *pile;
int sommet;
int taille;
const int pas = 10;

int init_pile() {
    taille = pas;
    pile = (int *) malloc (pas * sizeof(int));
    if (pile == NULL) return (-1);
    sommet = 0;
    return(0);
}

int push(int a) {
    *(pile + sommet++) = a;
    return(0);
}

int pop(int *a) {
    sommet--;
    *a = *(pile + sommet);
    return(0);
}

int main() {
    int i,k;
    if (init_pile() < 0) return(-1);
    for (i = 0; i < 10; i++) push(i);
    for (i = 0; i < 10; i++) {
        pop(&k);
        printf("%d\n",k);
    }
}
```

Nous constatons que la fonction `init_pile()` gère l'erreur "débordement mémoire" en retournant un code d'erreur négatif. Durcissez à leur tour les fonctions `push()` et `pop()` :

- `push()`
  - doit agrandir automatiquement la pile si nécessaire (par pas de 10 entiers)
  - et en cas d'impossibilité (débordement mémoire) retourner un code d'erreur négatif.
- `pop()` doit retourner un code d'erreur négatif, si la pile est vide.

## XV- LA GESTION DES FICHIERS

Avant de pouvoir accéder à un fichier il faut "ouvrir" ce fichier c'est à dire définir un flot de données "de" ou "vers" ce fichier, voire un flot de données bidirectionnel à la fois "de" et "vers" ce fichier.

Un flot de données c'est un ensemble ordonné de données, il sera représenté en langage C par une variable de type redéfini **FILE** pour cela il faudra inclure le fichier **stdio.h**.

### XV.1- Ouverture / Création de Fichier

#### La fonction fopen ()

Ouverture d'un fichier et obtention d'un "flot de données" pour échanger avec ce fichier .

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

- Le paramètre **pathname** est un pointeur sur le chemin (complet ou relatif) du fichier.
- Le paramètre **mode** est un pointeur sur le mode d'ouverture du fichier :

"r"	ouverture d'un fichier texte en lecture (en écrasement)
"w"	ouverture d'un fichier texte en écriture
"a"	ouverture d'un fichier texte en écriture à la fin (en ajout)
"r+", "w+"	ouverture d'un fichier texte en lecture/écriture
"a+"	ouverture d'un fichier texte en lecture/écriture à la fin

- ❖ En cas de succès retourne une variable de type **FILE** (flot de données sur le fichier).
- ❖ En cas d'échec retourne **NULL** et le code d'erreur dans la variable globale **errno**.  
\* **NULL** est défini dans **stdio.h**

#### Précisions :

- Si le mode contient la lettre **r**, le fichier doit exister, sinon c'est une erreur.
- Si le mode contient la lettre **a** ou **w** et que le fichier n'existe pas, il est créé.

#### La fonction fclose ()

Fermeture d'un fichier et libération du flot de données associé.

```
#include <stdio.h>
int fclose(int stream);
```

- Le paramètre **stream** est le flot de données associé au fichier
- En cas de succès retourne 0
- En cas d'échec **EOF** et le code d'erreur dans la variable globale **errno**.

\* **EOF** est défini dans **stdio.h**

## XV.2- Flux Binaires

### La fonction fread ()

Lecture dans un buffer des **n** blocs de **m** octets suivants d'un fichier.

```
#include <stdio.h>
size_t fread( void * buffer, size_t blocSize,
              size_t blocCount, FILE *stream );
```

- Le paramètre **stream** est l'identifiant du flot de données
  - Le paramètre **buffer** est le buffer de rangement des octets lus
  - Le paramètre **blocSize** est la taille du bloc d'octets de lecture
  - Le paramètre **blocCount** est le nombre de blocs à lire, le nombre d'octets à lire est donc le produit du nombre de blocs par la taille du bloc.
- En cas de succès retourne **blocCount**.
  - Si fin de fichier retourne un nombre de blocs lus égal à **0** ou inférieur à **blocCount**
  - En cas d'échec retourne **-1** et le code d'erreur dans la variable globale **errno**.

### La fonction fwrite ()

Ecriture d'un buffer de **n** blocs de **m** octets.

```
#include <stdio.h>
size_t fwrite( void *buffer, size_t blocSize,
              size_t blocCount, FILE *stream );
```

- Le paramètre **stream** est l'identifiant du flot de données
  - Le paramètre **buffer** est le buffer contenant les octets à écrire
  - Le paramètre **blocSize** est la taille du bloc d'octets en écriture
  - Le paramètre **blocCount** est le nombre de blocs à écrire, le nombre d'octets à écrire est donc le produit du nombre de blocs par la taille du bloc.
- En cas de succès retourne **blocCount**.
  - En cas d'échec retourne un nombre de blocs différent et le code d'erreur dans la variable globale **errno**.

## XV.3- Le Positionnement

Si un fichier est ouvert en lecture ou en écriture "**r**"/"**w**" le curseur est positionné au début de ce fichier (qu'il soit vide ou rempli).

Si un fichier est ouvert en en écriture en précisant la fin du fichier "**a**" le curseur est positionné à la fin de ce fichier.

Le curseur avance automatiquement au fur et à mesure des lectures ou écritures. Pour repositionner le curseur il faut utiliser la fonction **fseek()** :

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence)
```

- Le paramètre `stream` est l'identifiant du flot de données
  - Le paramètre `offset` est le décalage à apporter au curseur
  - Le paramètre `whence` désigne le point de départ du décalage
    - `SEEK_CUR` : position courante dans le fichier
    - `SEEK_SET` : début du fichier
    - `SEEK_END` : fin du fichier
- \* ces symboles sont définis dans `stdio.h`*

- En cas de succès retourne 0
- En cas d'échec retourne une valeur différente de 0 et le code d'erreur dans la variable globale `errno`.

Pour consulter la position du curseur dans le fichier il faut utiliser la fonction `ftell()` :

```
#include <stdio.h>
long ftell(FILE *stream)
```

- Le paramètre `stream` est l'identifiant du flot de données
- En cas de succès retourne 0
- En cas d'échec retourne -1 et le code d'erreur dans la variable globale `errno`.

Pour connaître la taille d'un fichier on peut utiliser des primitives système comme `fstat()` sous Linux, cependant pour des raisons de portabilité il vaut mieux utiliser la séquence standard suivante :

```
long dimension
fseek(FILE *stream, 0L, SEEK_END)
dimension = ftell(FILE *stream);
```

*\* ne pas oublier de repositionner votre curseur ensuite*

#### ***XV.4- Suppression de Fichier***

Pour supprimer un fichier il faut utiliser la fonction `remove()` :

```
#include <stdio.h>
int remove(const char *pathname)
```

- Le paramètre `pathname` est un pointeur sur le chemin (complet ou relatif) du fichier.
- En cas de succès retourne 0
- En cas d'échec retourne -1 et le code d'erreur dans la variable globale `errno`.

#### ***XV.5- Flux Textes Formattés***

*----- à compléter -----*

## ■ Exercices

1) Il s'agit de réaliser un programme permettant de saisir des données une par une et ensuite de les écrire dans un fichier de nom `donnees`.

Une donnée c'est un nombre flottant.

Sur saisie d'un nombre négatif, le programme s'arrête.

- *Vous ne pourrez pas visualiser votre fichier avec un éditeur de texte car votre fichier n'est pas un fichier texte mais un fichier binaire.*
- *Vous pourrez simplement vérifier la taille de votre fichier qui devrait être 4 fois le nombre de données saisies (1 flottant = 34 octets).*
- *Pour relire votre fichier il faudra utiliser le programme de l'exercice suivant.*

2) Il s'agit de réaliser un programme permettant de consulter les données enregistrées précédemment dans le fichier `donnees`.

On consulte les données une par une en saisissant leur numéro d'ordre.

Sur saisie d'un numéro négatif le programme s'arrête et sur saisie d'un numéro trop grand il affiche "Fin de fichier".

3) Ecrire un programme `alimentation.c` permettant de saisir puis de sauvegarder une structure de type `dino` dans un fichier `fiches.csv`

Ensuite écrire un second programme `consultation.c` permettant de relire le fichier `fiches.csv`

Rappel Fichier csv :

Fichier texte au format Excel `csv` : Les enregistrements sont séparés par des sauts de ligne `\n\r` (soit 0d 0a) et les champs dans l'enregistrement par des points-virgules ;

Attention il faut interdire les points-virgules à l'intérieur des champs !

## XVI- LA PROGRAMMATION MULTITHREAD

Un processus peut créer des processus fils ou être partagé en plusieurs processus légers ou threads.

- Les processus possèdent tous des espaces mémoire privés, et peuvent communiquer entre eux par l'intermédiaire de primitives système (dépendantes du système d'exploitation) ou par l'intermédiaire des protocoles TCP/IP.
- Les threads d'un même processus se partagent l'espace mémoire de ce processus (à l'exception des piles qui sont privées) et peuvent donc communiquer entre eux par l'intermédiaire de leurs données qui sont communes (**Data**, **Bss** et **Heap**).
- Choisir une application en mode multiprocessus (multi-programmes C) c'est privilégier la sécurité : Protection maximale.
- Choisir une application en mode multithreads c'est privilégier les performances : Communication inter-thread facilitée et mécanismes de protection mémoire moins coûteux en ressources système.

### Attention :

- En fonctionnement multiprocessus, un thread peut endommager les données d'un autre thread du même processus puisque ces données sont partagées, il va donc falloir gérer la concurrence d'accès aux données.
- Certaines fonctions C ou primitives Systèmes ne sont pas réentrantes, en particulier si elles font appel à des variables globales. Il ne faudra donc utiliser les fonctions déclarées MT-Safe (Multithread-Safe) et non pas celles déclarées MT-Unsafe ; cette information figure normalement dans le manuel Linux **man**.

### Création de Thread

C'est l'appel système **pthread\_create()** qui permet à un processus de créer un thread. Le code du nouveau thread est défini dans une fonction associée.

Lors de la création du premier thread on obtient 2 threads :

- Le processus originel transformé en thread et qui conserve son code.
- Le thread nouvellement créé dont le code figure dans la fonction associée.

Les appels système relatifs aux threads sont délicats à manipuler et en particulier :

- 1) Il faut compiler les programmes avec l'option **-pthread**
- 2) Un thread est identifié par le numéro de son processus **PID** + un numéro **LWP** (light-weight process), en parallèle un thread se voit attribuer un identifiant Posix<sup>12</sup> (cf. Normes).
- 3) L'appel système **gettid()** (obtenir **LWP**) a besoin de **<sys/syscall.h>** lequel fait appel à **<asm.unistd.h>**, il faut donc disposer de **/usr/include/asm.unistd.h** autrement ne pas utiliser l'appel à **gettid()**.

---

<sup>12</sup> Portable Operating System Interface unix

### Exemple de création de thread :

```
#define _GNU_SOURCE          /* or _BSD_SOURCE or _SVID_SOURCE */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>
int pere;

/*****
/* THREADS
*****/
static void *code_thread (void* arg) {
    pid_t tid;
    pthread_t pthr;
    tid = syscall(SYS_gettid);
    pthr = pthread_self();
    printf( "Je suis le thread %u (ID Posix %u) créé par le processus
%d \n", (unsigned int)tid, (unsigned int)pthr, pere);
    while(1) {}
}

/*****
/* PROCESSUS PERE
*****/
int main() {
    int i;
    pthread_t p; // identification Posix du thread
    pere = getpid();

    for (i = 0; i < 3; i++) {
        if (pthread_create(&p, NULL, &code_thread, NULL) != 0)
            printf( "Je suis le processus %d, faute lancement thread \n",
pere );
        else printf( "Je suis le processus %d, j'ai lancé le thread Posix
%u \n", pere, (unsigned int)p );
    }
    while(1) {} // Boucle sans Fin
}
```

### ■ Exercice

Ecrire un programme comportant 2 threads **lecture** et **écriture** :

- a) Le thread **lecture** saisit une ligne de texte au clavier en se présentant (en se nommant)
- b) La ligne lue est sauvegardée dans une variable commune
- c) Le thread **écriture** affiche au fur et à mesure les lignes sauvegardées en se présentant (en se nommant)

Il faudra trouver un moyen de synchroniser les 2 threads afin que l'on sépare bien les phases de lecture et d'écriture.

## **XVII- LA GENERATION DES EXECUTABLES**

----- *à compléter* -----

## **XIII- LES BIBLIOTHEQUES**

----- *à compléter* -----

## **XIX- L'ASSEMBLEUR IN-LINE**

----- *à compléter* -----

## **XX- LA DOCUMENTATION**

----- *à compléter* -----

## **XXI- ELEMENTS DE PORTABILITE**

----- *à Compléter* -----

## **XXII- ELEMENTS DE FIABILITE**

----- *à compléter* -----

## Annexe A : Corrigés des Exercices

### I- ALGORITHMIQUE

#### 1) Les 4 opérations

```
#include <stdio.h>
int main () {

    /* déclaration de variables de type entier */
    int a;
    int b;
    int c;

    /* Initialisation des variables */
    a = 2;
    b = 4;

    /* Addition */
    c = a + b;
    printf("addition : %d\n",c); // le format %d spécifie que
                                // la variable "c" qui suit
                                // est un entier décimal

    /* Soustraction */
    c = b - a;
    printf("soustraction : %d\n",c);

    /* Multiplication */
    c = a * b;
    printf("multiplication : %d\n",c);

    /* Division entière */
    c = b / a;
    printf("division : %d\n",c);
}
```

#### 2) Test Conditionnel

```
#include <stdio.h>
int main () {
    int a = 2;
    int b = 4;

    /* Test conditionnel */
    if (a > b) printf("a plus grand que b\n");
    else if (b > a) printf("b plus grand que a\n");
    else printf("a égal b\n");
}
```

#### 3) Lectures / Ecritures mémoire

Dans les 2 exercices précédents nous comprenons que l'accès aux variables a, b et c correspondent à des lectures / écritures en mémoire (chacune de ces variables correspondant à une zone en mémoire).

#### 4) Boucle Itérative

```
/* Calcul des 10 premières puissances de 2 */
#include <stdio.h>
int main () {
    int puissance = 1;
    int i;
    for (i = 0; i < 10; i++) {
        printf("%d : %d\n",i,puissance);
        puissance = puissance * 2;
    }
}
```

### III- VARIABLES SIMPLES

#### 1) Type et la valeur de chacune des expressions

- a) 8
- b) 6
- c) 1 (ou un entier différent de 0)
- d) 1 (ou un entier différent de 0)
- e) 1 (ou un entier différent de 0)

#### 2) Intérêts composés au bout de 10 ans d'une somme de 10.000 € à un taux de 7%.

```
#include <stdio.h>
int main() {
    int i;

    // Données
    float somme = 10000;
    float taux = 0.07;
    int annees = 10;
    float interet = 0;

    // Calcul des intérêts
    for (i = 1; i<= annees; i++) {
        interet = somme*taux;
        somme += interet;

        // Afficher le résultat
        printf ("%d : intérêt = %.2f\n",i,interet); }}
}
```

#### 3) Nombre d'années de remboursement d'une somme de 100.000 € empruntée à un taux de 5% avec des remboursements annuels de 10.000€.

```
#include <stdio.h>
int main() {
    int annee = 1;

    // Données
    float somme = 100000;
    float taux = 0.05;
    float remboursement = 10000;    // remboursement annuel

    // Calcul année par année
    while (somme > 0) {
```

```

    somme += somme*taux;
    if (somme > remboursement) {
        somme -= remboursement;
        printf ("%d : remboursement = %.2f\n",annee,remboursement);
        annee++;
    }

    /* cas de la dernière année */
    else {
        remboursement = somme;
        somme = 0;
        printf ("%d : remboursement = %.2f\n",annee,remboursement);
    }
}
}

```

#### IV- TABLEAUX

##### 1) Tableau d'entiers

```

#include <stdio.h>
int main (){
    int tab[20];          // tableau de 20 entiers
    int i;

    /* Initialisation du tableau à 0 */
    for (i = 0; i< 20; i++) {
        tab[i] = 0;
    }

    /* Affichage du tableau */
    for (i = 0; i< 20; i++) {
        printf("%d\n",tab[i]);
    }
}

```

##### 2) Tableau d'entiers à 2 dimensions

```

#include <stdio.h>
int main (){
    int tab[20][10];      // Tableau de 20 lignes de 10 entiers
    int i;
    int j;

    /* Initialisation du tableau à 0 */
    for (i = 0; i< 20; i++) {
        for (j = 0; j< 10; j++) {
            tab[i][j] = 0;
        }
    }

    /* Affichage du tableau */
    for (i = 0; i< 20; i++) {
        for (j = 0; j< 10; j++) {
            printf("%d : ",tab[i][j]);
        }
        printf("\n"); // Saut de ligne
    }
}

```

3) Ecrire un programme qui va :

- Saisir 10 nombres entiers et les ranger au fur et à mesure dans une table d'entiers.
- Calculer la moyenne des 10 nombres ainsi saisis.
- Afficher la moyenne

```
#include <stdio.h>
int main () {
    int moyenne;
    int nombre;
    const int nbr = 10;
    int tab [nbr];
    int i;

    // Saisir les nombres
    for (i = 0; i < nbr; i++) {
        printf("Veuillez saisir un nombre entier\n");
        scanf("%d",&nombre);
        tab[i] = nombre;
    }

    // Calcul de la moyenne
    moyenne = 0;
    for (i = 0; i < nbr; i++) {
        moyenne = moyenne + tab[i];
    }
    moyenne = moyenne / nbr;

    // Affichage des résultats
    for (i = 0; i < nbr; i++) {
        printf("table index %d : %d\n",i,tab[i]);
    }
    printf("La moyenne est : %d",moyenne);
}
```

Remarque :

*Nous ne traitons pas les erreurs de saisie, si l'utilisateur saisi des caractères au lieu de chiffres par exemple le comportement du programme serait imprévisible.*

*Nous traiterons la gestion des erreurs plus loin dans ce document.*

4) Tri à Bulles : On parcourt une première fois la table par couples "élément/élément suivant" et à chaque fois qu'un élément est supérieur à son élément suivant on inverse ces 2 éléments, ensuite on parcourt "n" fois la table en inversant les couples d'éléments si nécessaire jusqu'à ce qu'il n'y ait plus d'inversions à effectuer auquel cas la table sera ordonnée dans l'ordre croissant.

```
#include <stdio.h>
int main () {
    const int nbr = 10;
    int tab[10] = {7,9,1,6,8,10,4,3,2,5};
    int poursuite_tri = 1;    // Indicateur tri à poursuivre
    int i;
    int swap;

    // Affichage table avant le tri
    for (i = 0; i < nbr; i++) {
        printf("%d : ",tab[i]);
    }
}
```

```

printf("\n");

// Tri à bulles
while (poursuite_tri) {
    poursuite_tri = 0;

    // Parcours de la table
    for (i=0; i < nbr-1; i++) {

        // Si valeur a permuter
        if (tab[i] > tab[i+1]) {
            swap = tab[i];
            tab[i] = tab[i+1];
            tab[i+1] = swap;
            poursuite_tri = 1;
            for (i = 0; i < nbr; i++) {
                printf("%d : ",tab[i]);
            }
            printf("\n");
        }
    }
}

// Affichage table après le tri
for (i = 0; i < nbr; i++) {
    printf("%d : ",tab[i]);
}
printf("\n");
}

```

5) Ecrire un programme pour résoudre le problème des prisonniers du pirate :

Pour répondre à cet exercice nous allons construire un tableau de chainage à 2 dimensions dans lequel pour chacun des **n** prisonniers nous allons indiquer son précédent encore vivant et son suivant encore vivant.

```

#include <stdio.h>

int main () {
    const int nbr = 20;           // Nombre de prisonniers
    const int nsaut = 3; // Nombre de prisonniers à sauter
    int tab[nbr][2];             // Table des précédents (col 0)
                                // et des suivants (col 1)

    int nbr_t = 0;               // nombre de tués;
    int pr = 0;                  // numéro du prisonnier
    int precedent;               // numéro du précédent
    int suivant;                 // numero du suivant;
    int i;

    /* Initialisations table des suivants et des précédents */
    for(i = 0; i < nbr; i++) {
        tab[i][0] = i - 1; // Précédents
        tab[i][1] = i + 1; // Suivants
    }
    tab[0][0] = nbr-1; // Précédent du premier = dernier
    tab[nbr-1][1] = 0; // Suivant du dernier = premier

    /* Boucle d'Elimination de prisonnnier */
    while(1) {

```

```

    /* Elimination d'un prisonnnier */
    precedent = tab[pr][0];          // précédent
    suivant = tab[pr][1];           // suivant
    tab[precedent][1] = suivant; // suiv. du précéd. = suiv.
    tab[suivant][0] = precedent; // précéd. du suiv. = précéd.
    printf ("%d\n ",pr);

    /* Si nbr tués fini */
    nbr_t++;
    if (nbr_t == nbr) break;

    /* Saut de nsaut prisonniers */
    for(i = 0; i < nsaut+1; i++) {
        pr = tab[pr][1];           // suivant de prisonnier
    }
}
}

```

## V- CHAINES DE CARACTERES

### 2) La table ASCII :

Afficher les 7 lignes de caractères imprimables de la table ASCII (caractères de 20 hexa à 7F hexa).

```

#include <stdio.h>
int main () {
    int i;
    int j;

    /* Affichage numéros de colonnes de 0 à F */
    printf(" | ");
    for (j = 0; j < 16; j++) printf(" | %x",j);
    printf("\n");

    /* Affichage lignes 2 à 7 */
    for (i = 2; i <= 7; i++) {
        printf("%d | ",i);
        for (j = 0; j < 16; j++) {
            printf(" | %c", (i*16)+j);
        }
        printf("\n");
    }
}

```

### 2) Chaines "Hello" et "World" :

```

#include <string.h>
#include <stdio.h>
int main () {
    char h[6] = "Hello";
    char w[6] = "World";
    char hw[12];
    char *ptn;

    /* Concaténation */
    strcpy (hw,h) ;

```

```

    strcat(hw," ");
    strcat(hw,w);
    printf("hw = %s\n",hw);

    /* Sous Chaine World */
    ptn = strstr(hw,"World");
    if (ptn != NULL) printf("World en position %d\n",ptn-hw);
    else printf("non trouvé\n");

    /* Rogner et comparer */
    *ptn--=0; // suppression World et repositionnement2
    *ptn=0;    // suppression du blanc
    if (strcmp(hw,h) == 0) printf("hw et h sont identiques\n");
    else printf ("hw et h sont différents\n");
}

```

## VI- VARIABLES COMPOSEES

- Structure `dinosaure` :

```

#include <stdio.h>
#include <string.h>

/* Structure Dinosaure */
typedef struct dinosaure {
    int identifiant;
    char nom[50];
    int longueur;
    int carnivore;
    int os;
    int decouverte;
} dino;

void main() {
    int nbr;
    dino dinol;
    dino tab_dino[10];
    int i;

    /* Fiche Stégosaurus */
    dinol.identifiant = 65473;
    strcpy(dinol.nom,"stégosaurus");
    dinol.longueur = 16;
    dinol.carnivore = 0;
    dinol.os = 324;
    dinol.decouverte = 1925;
    printf("identifiant : %d\n",dinol.identifiant);
    printf("nom          : %s\n",dinol.nom);
    printf("longueur     : %d\n",dinol.longueur);
    if (dinol.carnivore) printf("carnivore : carnivore\n");
    else printf("carnivore : non carnivore\n");
    printf("identifiant : %d\n",dinol.identifiant);
    printf("os          : %d\n",dinol.os);
    printf("decouverte  : %d\n",dinol.decouverte);

    /* Tableau 10 fiches */
    tab_dino[0] = dinol;
    for (i = 1; i < 10; i++) {
        tab_dino[i].identifiant = 1000 + i;
    }
}

```

```

    }
    for (i = 0; i < 10; i++) {
        printf("identifiant n° %d : %d\n",i,tab_dino[i].identifiant);
    }
}

```

## VII- LES POINTEURS

1) Tableau de 10 int :

```

#include <stdio.h>
int main () {
    int tableau[10];
    int taille;
    int i;

    /* Vérification des tailles */
    taille = sizeof(int);
    printf ("taille de l'entier : %d\n",taille);
    taille = sizeof(tableau);
    printf ("taille du tableau : %d\n",taille);

    /* Vérification des adresses */
    for (i = 0; i < 10; i++) {
        printf("Adresse element %d : %ul\n",i,&tableau[i]);
    }
}

```

2) Initialisez à zéro les éléments d'un tableau :

```

#include <stdio.h>
int main () {
    int tableau[100][100];
    int *ptn = &tableau[0][0];
    int i,j;
    int nb_error;

    /* Initialisation du tableau */
    for (i = 0; i < 100; i++) {
        for (j = 0; j < 100; j++) {
            *ptn++ = 0;
        }
    }

    /* Vérification du tableau */
    nb_error = 0;
    for (i = 0; i < 100; i++) {
        for (j = 0; j < 100; j++) {
            if (tableau[i][j] != 0) nb_error++;
        }
    }
    printf("nombre d'erreurs trouvées : %d\n",nb_error);
}

```

3) Récupérez le tableau de 10 structures **dinosaure** créé précédemment :

```

#include <stdio.h>
#include <string.h>

```

```

/* Structure Dinsaure */
struct dinosaure {
    int identifiant;
    char nom[50];
    int longueur;
    int carnivore;
    int os;
    int decouverte;
};

void main() {
    int nbr;
    struct dinosaure dinol;
    struct dinosaure tab_dino[10];
    struct dinosaure *ptn = &tab_dino[0];
    int i;

    /* Fiche Stégosaurus */
    dinol.identifiant = 65473;
    strcpy(dinol.nom,"stégosaurus");
    dinol.longueur = 16;
    dinol.carnivore = 0;
    dinol.os = 324;
    dinol.decouverte = 1925;

    /* Tableau 10 fiches */
    tab_dino[0] = dinol;
    for (i = 1; i < 10; i++) {
        tab_dino[i].identifiant = 1000 + i;
    }

    /* Modifier première fiche dans tableau */
    /* A l'aide d'un pointeur sur le tableau */
    ptn->longueur = 18;
    ptn->os = 307;
    ptn->decouverte = 1943;
    printf("identifiant : %d\n",tab_dino[0].identifiant);
    printf("nom          : %s\n",tab_dino[0].nom);
    printf("longueur      : %d\n",tab_dino[0].longueur);
    if (tab_dino[0].carnivore) printf("carnivore : carnivore\n");
    else printf("carnivore : non carnivore\n");
    printf("os            : %d\n",tab_dino[0].os);
    printf("decouverte    : %d\n",tab_dino[0].decouverte);

    /* Modifier la troisième fiche dans tableau */
    /* A l'aide d'un pointeur sur le tableau */
    ptn += 2;
    strcpy(ptn->nom,"Tyranosaurus");
    ptn->longueur = 15;
    ptn->carnivore = 1;
    ptn->os = 456;
    ptn->decouverte = 1939;
    printf("identifiant : %d\n",tab_dino[2].identifiant);
    printf("nom          : %s\n",tab_dino[2].nom);
    printf("longueur      : %d\n",tab_dino[2].longueur);
    if (tab_dino[2].carnivore) printf("carnivore : carnivore\n");
    else printf("carnivore : non carnivore\n");
    printf("os            : %d\n",tab_dino[2].os);
    printf("decouverte    : %d\n",tab_dino[2].decouverte);
}

```

*\* Attention : dans ce corrigé on aurait dû créer une fonction affichage de fiche plutôt que de répéter l’affichage 2 fois de suite une fois pour la première fiche et une fois pour la troisième. En fait les fonctions seront introduites au chapitre suivant.*

## VIII- LES FONCTIONS

1) Valeur max des 2 entiers :

```
#include <stdio.h>

int plus_grand (int a, int b) {
    if (a > b) return(a);
    else return (b);
}

int main() {
    int premier, second;
    printf("Veuillez saisir votre premier nombre : ");
    scanf("%d",&premier);
    printf("Veuillez saisir votre second nombre : ");
    scanf("%d",&second);
    printf("La plus grand evaleur : %d\n",plus_grand(premier,second));
}
```

2) Reprendre le programme "Intérêts Composés" :

```
#include <stdio.h>

float interet(float somme, float taux, int annees) {
    float interet;
    int i;
    for (i = 1; i<= annees; i++) {
        interet = somme*taux;
        somme += interet;
    }
    return(somme);
}

int main() {
    int annees;
    float somme, taux;
    printf("Veuillez saisir la somme : ");
    scanf ("%f",&somme);
    printf("Veuillez saisir le taux (0.07 pour 7 pour cent) : ");
    scanf ("%f",&taux);
    printf("Veuillez saisir le nombre d'années : ");
    scanf ("%d",&annees);
    somme = interet(somme,taux,annees);
    printf ("Montant au bout de %d ans : %.2f\n",annees,somme);
}
```

## X- LA REDEFINITION DES TYPES

Créez un nouveau type `dino` correspondant à la structure `dinosaure` définie plus haut.

```
typedef struct dinosaure {
    int identifiant;
    char nom[50];
    int longueur;
    int carnivore;
    int os;
    int decouverte;
} dino ;
```

## XII- STRUCTURE D'UN PROGRAMME "C"

Reprendre le programme intérêts composés, la version avec la fonction `interet()`.

### Programme principal.c

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

int main() {
    int annees;
    float somme, taux;
    printf("Veuillez saisir la somme : ");
    scanf ("%f",&somme);
    printf("Veuillez saisir le taux (0.07 pour 7 pour cent) : ");
    scanf ("%f",&taux);
    printf("Veuillez saisir le nombre d'années : ");
    scanf ("%d",&annees);
    somme = interet(somme,taux,annees);
    printf ("Montant au bout de %d ans : %.2f\n",annees,somme);
}
```

### Fonction interet.c

```
float interet(float somme, float taux, int annees) {
    float interet;
    int i;
    for (i = 1; i<= annees; i++) {
        interet = somme*taux;
        somme += interet;
    }
    return(somme);
}
```

### Include interet.h

```
float interet(float somme, float taux, int annees);
```

### ***XIII- LA GESTION DE LA MEMOIRE***

1) Recopie de zones mémoire :

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

int main() {
    int *ptn;
    int *ptn2;
    int i,k;

    ptn = (int *) malloc (10 * sizeof(int));
    k = 1;
    for (i = 0; i < 10; i++) {
        ptn[i] = k;
        k *= 2;
    }
    ptn2 = (int *) malloc (10 * sizeof(int));
    memcpy (ptn2,ptn,10 * sizeof(int));
    for (i = 0; i < 10; i++) printf("%d ",ptn2[i]);
    printf("\n");
}
```

### ***XIV- LE TRAITEMENT DES ERREURS***

**push()** permet d'empiler un entier

**pop()** permet de dépiler un entier

```
int push(int a) {
    if (sommet >= taille) {
        taille += pas;
        int *new_pile = realloc(pile, pas * sizeof(int));
        if (new_pile == NULL) return(-1);
        pile = new_pile;
    }
    *(pile + sommet++) = a;
    return(0);
}

int pop(int *a) {
    if (sommet-- <= 0) return (-1);
    *a = *(pile + sommet);
    return(0);
}
```

### ***XV- LA GESTION DES FICHIERS***

1) Saisie écriture dans un fichier de données

```
#include <stdio.h>

/* Programme principal */
void main() {
    float data;
    FILE *fplot;
    int taille, nbr;
```

```

/* Ouverture du fichier "donnees" */
flot = fopen ("donnees", "a"); // ouverture du fichier
if (flot == NULL) {
    perror("Faute ouverture : \n");
    return;
}
taille = sizeof (data);

/* Boucle de traitement */
while (1) {

    /* Saisie de la donnée */
    printf("Veuillez saisir votre donnee : ");
    scanf ("%f",&data);
    printf("donnee = %f\n",data);
    if (data < 0) return; // sortie du programme

    /* Enregistrement de la donnee */
    nbr = fwrite(&data, taille, 1, flot); // ecriture dans le fichier
    if (nbr != 1) {
        perror("Faute ecriture : \n");
        return;
    }
}

/* Fermeture du fichier */
fclose(flott);
}

```

## 2) Relecture des données saisies.

On consulte les données une par une en saisissant leur numéro d'ordre.

Sur saisie d'un numéro négatif le programme s'arrête et sur saisie d'un numéro trop grand il affiche "Fin de fichier".

```

#include <stdio.h>

/* Programme principal */
void main() {
    int ret;
    FILE *flot;
    int taille;
    float data;
    int num;

    /* Ouverture du fichier */
    flot = fopen ("donnees", "r"); // ouverture du fichier
    if (flot == NULL) {
        perror("Faute ouverture");
        return;
    }
    taille = sizeof(data);

    /* Boucle de lecture */
    while (1) {

        /* Saisie num donnee */
        printf("Saisissez le num de la donnee : ");

```

```

scanf("%d",&num);
if (num < 0) return;      // Sortie du programme

/* Lecture du fichier */
ret = fseek(flot, taille*(num-1), SEEK_SET);
if (ret != 0) {
    perror("Faute positionnement");
    return;
}
ret = fread(&data, taille, 1, flot); // lecture du fichier
if (ret == -1) {
    perror("Faute lecture");
    return;
}

/* Affichage */
if (ret != 0) printf("%f\n",data);
else printf("Fin de fichier\n");
}

/* Fermeture du fichier */
fclose(flot);
}

```

3) Ecrire un programme **alimentation.c** permettant de saisir puis de sauvegarder une structure de type **dino** dans un fichier **fiches.csv**

Ensuite écrire un second programme **consultation.c** permettant de relire le fichier **fiches.csv**

#### SOURCE ALIMENTATION.C

```

#include<stdio.h>
#include<string.h>
char oui[10];
int main () {
    FILE *descripteur;
    char message[1000];
    int i;

    // Définition
    typedef struct dinosaure {
        int id;
        char nom[51];
        int lng;
        int carn;
        int os;
        int annee;
    } DINO;

    // Déclaration
    DINO dinol;
    DINO *ptn = &dinol;
    int ret;

    // Saisie des Données
    printf("Identifiant : ");
    scanf("%d",&dinol.id);
    printf("Nom : ");
    scanf("%s",&dinol.nom);

```

```

printf("Longueur : ");
scanf("%d",&dino1.lng);
printf("Carnivore ? (oui/non) : ");
scanf("%s",oui);
if (strcmp(oui,"oui",3) == 0) dino1.carn = 1;
else dino1.carn = 0;
printf("Nombre d'os : ");
scanf("%d",&dino1.os);
printf("Annee de decouverte : ");
scanf("%d",&dino1.annee);

// Filtrer les points-virgules
i = 0;
while (ptn->nom[i]) {
    if (ptn->nom[i] == ';') ptn->nom[i] = '_';
    i++;
}

// Formatter la fiche

sprintf(message,"%d;%s;%d;%d;%d;%d\n\r",dino1.id,dino1.nom,dino1.lng,
dino1.carn,dino1.os,dino1.annee);

// Enregistrement de la fiche
descripteur = fopen("fiches.csv","a");
if (descripteur == NULL) {
    perror ("Faute acces fiches.csv");
    return(1);
}
ret = fwrite( message, strlen(message)-1 , 1, descripteur );
if (ret != 1) {
    perror ("Faute ecriture fiches.csv");
    return(1);
}
}

```

#### SOURCE CONSULTATION.C

```

#include<stdio.h>
#include<string.h>
#include<errno.h>
char oui[10];
int main () {
    FILE *descripteur;
    char message[10000];
    char *ptn;
    int i, lng;

    // Définition
    typedef struct dinosaure {
        int id;
        char nom[51];
        int lng;
        int carn;
        int os;
        int annee;
    } DINO;

    // Déclaration
    DINO dino1;

```

```

// Lecture fiche par fiche
descripteur = fopen("fiches.csv","r");
if (descripteur == NULL) {
    perror ("Faute acces fiches.csv");
    return(1);
}
errno = 0;
while (1) {
    ptn = fgets(message, 1000, descripteur );
    if (errno != 0) {
        perror ("Faute lecture fiches.csv");
        return(1);
    }
    if (ptn == NULL) break;

    // Affichage fiche
    lng = strlen(message);
    for (i = 0; i < lng; i++) if (message[i] == ';')
message[i] = 0;
    printf("Identifiant : %s\n", ptn);
    ptn += strlen(ptn) +1;
    printf("Nom : %s\n", ptn);
    ptn += strlen(ptn) +1;
    printf("Longueur : %s\n", ptn);
    ptn += strlen(ptn) +1;
    if (*ptn == '0') printf("Non carnivore\n");
    else printf("Carnivore\n");
    ptn += strlen(ptn) +1;
    printf("Nombre d'os : %s\n", ptn);
    ptn += strlen(ptn) +1;
    printf("Année découverte : %s\n", ptn);
}
}

```

## ***XVI- LA PROGRAMMATION MULTITHREADS***

```

#define _GNU_SOURCE          /* or _BSD_SOURCE or _SVID_SOURCE */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>
char message[1000];
int verrou = 1;

/*****/
/* THREAD LECTURE */
/*****/
static void *code_lecture (void* arg) {
pid_t tid = syscall(SYS_gettid);
sleep (1);
while (1) {
    if (verrou) {
        printf("%d-lecture : ",tid);
        fgets (message, 999, stdin);
        verrou = 0;
    }
}
}
}

```

```

/*****
/* THREAD ECRITURE
*****/
static void *code_ecriture (void* arg) {
pid_t tid = syscall(SYS_gettid);
sleep (1);
while (1) {
    if (!verrou) {
        printf("%d-écriture : ",tid);
        printf("%s",message);
        verrou = 1;
    }
}
}

/*****
/* PROCESSUS PERE
*****/
int main() {
int i;
pthread_t p;
int pere = getpid();

    if (pthread_create(&p, NULL,&code_lecture, NULL) != 0)
        printf( "Je suis le processus %d, faute lancement thread
lecture\n", pere );
    else printf( "Je suis le processus %d, j'ai lancé le thread
lecture\n", pere);

    if (pthread_create(&p, NULL,&code_ecriture, NULL) != 0)
        printf( "Je suis le processus %d, faute lancement thread
écriture\n", pere );
    else printf( "Je suis le processus %d, j'ai lancé le thread
écriture\n", pere);

    while(1) {}    // Boucle sans Fin
}

```

## Annexe B : Installation du Compilateur C

### Installation sous Linux

Exemple d'installation avec `apt-get`.\*.

```
$ sudo apt update
$ sudo apt-get install gcc
```

Une fois le compilateur en place, vérifiez l'installation en affichant la version de "gcc" :

```
$ gcc --version
```

\* A vous de l'adapter en fonction de votre installateur Linux

### Installation sous Windows

- Téléchargez `mingw-get-setup.exe` sur le site de SourceForge
- Lancez `mingw-get-setup.exe` appuyez sur Installer puis lors de l'affichage de l'écran "MinGW Installation Manager" cochez les packages suivants à installer :
  - `Mingw devopper tools`
  - `Mingw base`
  - `Msys base`
- Menu `Installation -> Apply Changes`
- Rajoutez dans la variable environnementale "`path`" le chemin : `C:\MinGW\bin`
- Enfin vérifiez l'installation en affichant la version de "gcc" :

```
$ gcc --version
ou C:\MinGW\bin\gcc --version (si vous n'avez pas réussi à modifier le path)
```

### Installation sous Mac OS

Pour installer le compilateur gcc sur Mac OS X, vous devez d'abord par installer les "Outils de ligne de commande Xcode", disponibles sur la page de développement d'Apple :

- Vous devez commencer par créer un compte Apple (gratuit) :  
<https://developer.apple.com/downloads/index.action>
- Ensuite cliquez sur "Commande line tools for Xcode" dans la page de développement Apple et téléchargez le fichier `*.dmg`.
- Une fois le fichier `*.dmg` téléchargé, une boîte de dialogue vous présentera le fichier "Outils de ligne de commande", double-cliquez dessus pour l'installation.
- Une fois le compilateur en place, vérifiez l'installation en affichant la version de "gcc" :

```
$ gcc -v
```

## Annexe C : Interprétation des Déclarations

- ❖ *Attention, pour la lisibilité des codes sources (et donc leur maintenance) il est conseillé de limiter au maximum les déclarations complexes (ces dernières sont à utiliser à bon escient).*

Pour interpréter une déclaration complexe, il faut commencer par le nom de la variable ou de la fonction que l'on définit et évaluer ensuite immédiatement à droite ce sur quoi opère cet élément, puis immédiatement à gauche ce qui qualifie cet élément et ainsi de suite :

```
int (*fp)(int, int);
```

- Ici l'élément que l'on définit c'est "**fp**" : « "fp" c'est »
- rien à droite (parenthèse fermante)
- à gauche l'opérateur pointeur : « "fp" c'est un pointeur »
- on quitte la parenthèse imbricante
- à droite (**int, int**) : « "fp" c'est un pointeur sur une fonction avec en arguments 2 entiers »
- à gauche **int** : « "fp" c'est un pointeur sur une fonction avec en arguments 2 entiers qui retourne un entier »

```
void* (*depile[3])(pile**);
```

- Ici l'élément que l'on définit c'est "**depile**",
- accolades à droite «un tableau de 3 »
- à gauche l'opérateur pointeur : «un tableau de 3 pointeurs »
- on quitte la parenthèse imbricante
- à droite "**(pile\*\*)**" : « pointeurs sur des fonctions avec en argument un pointeur de pointeur sur pile »
- à gauche "**void\***" : « "depile" c'est un tableau de 3 pointeurs sur des fonctions avec en argument un pointeur de pointeur sur "pile" qui retournent un "pointeur void" »

```
int * (* (*fp1)(int) ) [10];
```

- « "fp1" est pointeur »
- « sur une fonction avec en argument 1 entier »
- « qui retourne 1 pointeur »
- « sur un tableau de 10 »
- « pointeurs sur des entiers »

```
int *( * ( *arr[5] ) ) ();
```

- « "arr" est un tableau de 5 »
- « pointeurs »
- « sur des fonctions sans argument »
- « qui retournent un pointeur »
- « sur une fonction sans argument »
- « qui retourne un pointeur sur un entier »

### Cas où la déclaration ne comporte pas d'identificateur :

Dans le cas du "**cast**" ou du prototypage des fonctions, les déclarations ne comportent pas de nom de variable ni de nom de fonction, il faut donc d'abord trouver la position de l'identificateur puis procéder comme décrit ci-dessus.

L'identificateur se trouve dans la première parenthèse en partant de la gauche, si cette parenthèse contient d'autres parenthèses alors l'identificateur se trouvera dans la première des parenthèses imbriquées et ainsi de suite.

Si la déclaration ne contient aucune parenthèse, il s'agit d'une déclaration simple et qui ne pose pas de problème d'interprétation.

----- à Valider -----

## Annexe D : Les Normes

### The C Programming Language

En 1978, Brian Kernighan et Dennis Richie publient la première spécification du langage C dans le livre « The C Programming Language ».

### La norme ANSI-ISO C-89

En 1983, l'ANSI<sup>13</sup> décide de normaliser le langage C ; ce travail s'achève en 1989 avec la publication de la norme ANSI C-89.

La norme ANSI C-89 est reprise par l'ISO en 1990.

❖ Cf. index des 32 mots clefs du C-89 à la fin du document.

### La norme ISO C-99

En 1999, une nouvelle évolution du langage est normalisée : ISO/CEI 9899:1999. Les nouveautés portent notamment sur les "tableaux de taille variable", les "pointeurs restreints", les "nombres complexes", les "littéraux composés", les "déclarations mélangées avec les instructions", les "fonctions inline", le "support avancé des nombres flottants", et la "syntaxe de commentaire de C++". La bibliothèque standard du C-99 a été enrichie de six fichiers d'en-tête depuis la précédente norme.

*Source Wikipedia*

❖ Le C-99 introduit les 5 nouveaux mots-clefs :

`_Bool`, `_Complex`, `_Imaginary`, `inline`, `restrict`.

❖ Pour activer les fonctionnalités C-99 : `gcc -std=c99`

### La norme ISO C-11

En 2011, l'ISO ratifie une nouvelle version du langage : ISO/CEI 9899:2011. Cette évolution introduit notamment le "support de la programmation multi-thread", les "expressions à type générique", et un meilleur "support d'Unicode".

*Source Wikipedia*

❖ Le C-11 introduit les 7 nouveaux mots-clefs :

`_Alignas`, `_Alignof`, `_Atomic`, `_Generic`, `_Noreturn`, `_Static_assert`, `_Thread_local`.

❖ Pour activer les fonctionnalités C-11 : `gcc -std=c11` (à partir de la version 4.7 de gcc)

---

<sup>13</sup> (American National Standards Institute)

## Bibliothèques Standards du C-89

<code>&lt;assert.h&gt;</code> :	Diagnostic des assertions
<code>&lt;ctype.h&gt;</code> :	Classification des caractères
<code>&lt;errno.h&gt;</code> :	Gestion des erreurs
<code>&lt;float.h&gt;</code> :	Portabilité des nombres flottants
<code>&lt;limits.h&gt;</code> :	Bornes des variables
<code>&lt;locale.h&gt;</code> :	Paramètres locaux : heure, monnaie ...
<code>&lt;math.h&gt;</code> :	Fonctions mathématiques
<code>&lt;setjmp.h&gt;</code> :	Basculement de piles
<code>&lt;signal.h&gt;</code> :	Gestion des signaux inter-processus
<code>&lt;stdarg.h&gt;</code> :	Fonctions variadiques
<code>&lt;stddef.h&gt;</code> :	Types et macros complémentaires
<code>&lt;stdio.h&gt;</code> :	Entrées / Sorties standard
<code>&lt;stdlib.h&gt;</code> :	Conversion de chaînes de caractères
<code>&lt;string.h&gt;</code> :	Manipulation de chaînes de caractères, accès mémoire
<code>&lt;time.h&gt;</code> :	Manipulation des dates

## Index des 32 mots-clefs du C-89

auto, 47  
break, 15, 16  
case, 14  
char, 19, 20  
const, 10  
continue, 16  
default, 14, 15  
do, 16  
double, 19  
else, 14  
enum, 33  
extern, 47  
float, 19  
for, 15  
goto, 15  
if, 14  
int, 19  
long, 19  
register, 53  
return, 39  
short, 19  
signed, 19  
sizeof, 20  
static, 40, 47  
struct, 32  
switch, 14  
typedef, 48  
union, 33  
unsigned, 19  
void, 37  
volatile, 53  
while, 16

Référence C89	
<i>auto</i>	<i>int</i>
<i>break</i>	<i>long</i>
<i>case</i>	<i>register</i>
<i>char</i>	<i>return</i>
<i>const</i>	<i>short</i>
<i>continue</i>	<i>signed</i>
<i>default</i>	<i>sizeof</i>
<i>do</i>	<i>static</i>
<i>double</i>	<i>struct</i>
<i>else</i>	<i>switch</i>
<i>enum</i>	<i>typedef</i>
<i>extern</i>	<i>union</i>
<i>float</i>	<i>unsigned</i>
<i>for</i>	<i>void</i>
<i>goto</i>	<i>volatile</i>
<i>if</i>	<i>while</i>