# Initiation au langage C

par Jessee Michaël C. Edouard (Accueil)

Date de publication : 24 mars 2008

Dernière mise à jour : 13 juin 2010

Ce tutoriel va vous apprendre les concepts de base du langage C. Il n'est ni une référence ni une définition du langage mais a été écrit dans le but d'aider le lecteur à le prendre en main avant d'étudier les concepts avancés de ce langage. Bonne lecture.
Commentez cet article :

# I - Introduction

## I-A - Historique

L'histoire du langage C est intimement liée à celle du système d'exploitation UNIX. En 1969, Ken Thompson, qui travaillait pour Bell Laboratories, mit au point sur un DEC PDP-7 un système d'exploitation semblable à Multics mais plus simple et plus modeste. Ce système reçut par la suite le nom d'Unics (Uniplexed Information and Computing System) qui ne tarda pas à être définitivement changé en **UNIX**.
A l'époque, le seul langage vraiment adapté à l'écritute de systèmes d'exploitation était le langage d'assemblage. Ken Thompson développa alors un langage de plus haut niveau (c'est-à-dire plus proche du langage humain), le **langage B** (dont le nom provient de **B**CPL, un sous-ensemble du langage CPL, lui-même dérivé de l'Algol, un langage qui fut populaire à l'époque), pour faciliter l'écriture d'UNIX. C'était un langage faiblement typé (un langage non typé, par opposition à un langage typé, est un langage qui manipule les objets sous leur forme binaire, sans notion de type (caractère, entier, réel, ...)) et trop dépendant du PDP-7 pour permettre de porter UNIX sur d'autres machines. Alors Denis Ritchie et Brian Kernighan, de Bell Laboratories également, améliorèrent le langage B pour donner naissance au **langage C**. En 1973, UNIX fut réécrit entièrement en langage C. Pendant 5 ans, le langage C fut limité à l'usage interne de Bell jusqu'au jour ou Brian Kernighan et Denis Ritchie publièrent une première définition du langage dans un ouvrage intitulé The C Programming Language. Ce fut le début d'une révolution dans le monde de l'informatique. Grâce à sa puissance, le langage C devint rapidement très populaire et en 1983, l'**ANSI** (American National Standards Institute) décida de le normaliser en ajoutant également quelques modifications et améliorations, ce qui donna naissance en 1989 au langage tel que nous le connaissons aujourd'hui. L'année suivante, le langage C a été adopté par l'**ISO** (International Organization for Standardization) et actuellement, par l'ISO et l'**IEC** (International Electrotechnical Commission), et a connu au fil du temps certaines révisions (dans l'immense majorité des cas toujours compatibles avec les versions antérieures) dont les plus importantes sont celles de 1990 (adoption par l'ISO), de 1995 (amélioration du support du développement de programmes internationnaux) et de 1999 (ajout d'un nombre important de nouvelles fonctionnalités). La version de 1999 est cependant actuellement encore peu diffusée c'est pourquoi dans ce document, nous nous placerons dans l'apprentissage des versions de 1990 et 1995.

## I-B - Caractéristiques du langage C

### I-B-1 - Universalité

Langage de programmation par excellence, le C n'est pas confiné à un domaine particulier d'applications. En effet, le C est utilisé dans l'écriture

- de systèmes d'exploitations (comme Windows, UNIX et Linux) ou de machines virtuelles (JVMs, Runtimes et Frameworks .NET, logiciels de virtualisation, etc.)
- de logiciels de calcul scientifique, de modélisation mathématique ou de CFAO (Matlab, R, Labview, Scilab, etc.)
- de bases de données (MySQL, Oracle, etc.)
- d'applications en réseau (applications intranet, internet, etc.)
- de jeux vidéo (notamment avec OpenGL ou la SDL et/ou les différents moteurs (de jeu, physique ou 3D) associés))
- d'assembleurs, compilateurs, débogueurs, interpréteurs, de logiciels utilitaires et dans bien d'autres domaines encore.

Oui, le C permet tout simplement de tout faire, ce qui lui a valu, à juste titre, l'appelation courante "the God's programming language" (le langage de programmation de Dieu). A cause de son caractère proche du langage de la machine, le C est cependant peu productif, ce qui signifie qu'il faut souvent écrire beaucoup pour faire peu. C'est donc le prix à payer pour l'utilisation de ce langage surpuissant !

## I-B-2 - Concision, souplesse

C'est un langage concis, très expressif, et les programmes écrits dans ce langage sont très compacts grâce à un jeu d'opérateurs puissant.

## I-B-3 - Puissance

Le C est un langage de haut niveau mais qui permet d'effectuer facilement des opérations de bas niveau et d'accéder aux fonctionnalités du système, ce qui est la plupart du temps impossible ou difficile à réaliser dans les autres langages de haut niveau. C'est d'ailleurs ce qui fait l'originalité du C, et aussi la raison pour laquelle il est autant utilisé dans les domaines où la performance est cruciale comme la programmation système, le calcul numérique ou encore l'informatique embarquée.

## I-B-4 - Portabilité

C'est un langage qui ne dépend d'aucune plateforme matérielle ou logicielle, c'est-à-dire qui est entièrement portable. De plus, de par sa simplicité, écrire un compilateur C pour un processeur donné n'est pas beaucoup plus compliqué que d'écrire un assembleur pour ce même processeur. Ainsi, là où l'on dispose d'un assembleur pour programmer, on dispose aussi généralement d'un compilateur C, d'où l'on dit également que le C est un "assembleur portable".

## I-B-5 - Omniprésence

La popularité du langage mais surtout l'élégance des programmes écrits en C est telle que son style et sa syntaxe ont influencé de nombreux langages :

- C++ et Objective C sont directement descendus du C (on les considère souvent comme des extensions du C)
- C++ a influencé Java
- Java a influencé JavaScript
- PHP est un mélange de C et de langage de shell-scripting sous Linux
- C# est principalement un mix de C++ et de Java, deux langages de style C, avec quelques influences fonctionnelles d'autres langages comme Delphi et Visual Basic (le # de C# vient-il dit-on de deux "++" superposés)
- Et on risque d'y passer la journée ...

Connaître le C est donc très clairement un atout majeur pour quiconque désire apprendre un de ces langages.

## I-C - A propos de ce document

Ce document contient des cours et des exercices. Chaque fois que vous tombez sur des exercices, traitez-les avant de passer au chapitre ou au paragraphe suivant.

## I-D - Remerciements

J'adresse un remerciement tout particulier à **JagV12** pour sa relecture très attentive de cet article ainsi qu'à **Franck.h** pour ses conseils et son soutien.

# II - Instruction, programme et fonction

## II-A - Structure générale d'un programme C

### II-A-1 - Un premier programme

Nous allons écrire un programme qui affiche **Hello, world** en langage C.

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

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

Voici la sortie de ce programme (c'est-à-dire, ce que nous, utilisateurs, verrons affiché sur le périphérique de sortie standard) :

```
Hello, world
```

Le **caractère** '\n' dans la **chaîne de caractères** "Hello, world\n" représente le caractère de fin de ligne. Il permet d'insérer une nouvelle ligne (**n**ew line).
En langage C, la « brique » qui permet de créer un programme est la **fonction**. Un programme écrit en C est donc constitué d'une ou plusieurs fonctions, une fonction étant généralement composée d'une ou plusieurs **instructions**, chaque instruction élémentaire devant se terminer par un point-virgule.
**printf** est une fonction qui permet d'afficher du texte sur la **sortie standard**, par défaut l'écran. **main** est également une fonction, c'est celle qui sera automatiquement appelée à l'exécution. On l'appelle le **point d'entrée** du programme ou encore la **fonction principale**. Ici, elle est seulement composée de deux instructions :

```
1    printf("Hello, world\n");
2    return 0;
```

Selon la norme officielle du langage C, **main** est une fonction qui **doit retourner un entier** (**int**). Chez de nombreux systèmes (dont Windows et UNIX), cet entier est appelé le **code d'erreur** de l'application. En langage C, bien que cela ne soit pas forcément le cas pour le système d'exploitation, on retourne **0** pour dire que tout s'est bien passé.
Le langage C impose (à quelques exceptions près que nous verrons plus bas) qu'une fonction doit avoir été déclarée avant de pouvoir être utilisé (nous verrons plus tard ce que c'est qu'une **déclaration**). Dans notre exemple, puisque nous utilisons la fonction **printf**, nous devons tout d'abord la déclarer. Le moyen le plus simple de le faire est d'inclure le fichier **stdio.h**, qui contient entre autres la déclaration de cette fonction, à l'aide de la **directive include** :

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

### II-A-2 - Les commentaires

On peut insérer des commentaires n'importe où dans du code C à l'aide des délimiteurs /* et */. Les commentaires permettent de rendre les programmes plus lisibles. Voici un exemple d'utilisation de commentaires :

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

hello.c
```
int main()
{
    /* Ce programme affiche "Hello, world" */
    printf("Hello, world\n");
    return 0;
}
```

## II-A-3 - Sensibilité à la casse

Le C est sensible à la casse : il fait la distinction entre les majuscules et les minuscules. Printf et printf sont deux choses complètement différentes et qui n'ont strictement rien à voir ...

## II-A-4 - Le format libre

En langage C, les espaces (y compris les tabulations et les retours à la ligne) peuvent être utilisées à volonté (ou pas du tout) avant ou après un séparateur (# < > ( ) { } ; □). On traduit ce fait en disant que le C est un langage au **format libre**. Une chaîne ("...") doit cependant toujours tenir sur une seule ligne (nous reviendrons sur ce point plus tard). Ainsi, notre programme qui affiche "Hello, world" aurait également pu s'écrire :

```
#   include    <stdio.h>
int
main ( )
{   printf    (
        "Hello, world\n"
    );return

    0

;}
```

Dans certains cas, les espaces peuvent cependant être utilisés de manière précise. Par exemple, avec :

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

Vous demandez d'inclure le fichier **<espace>stdio.h<espace>** (qui a de fortes chances de ne pas exister, ce qui génèrerait des erreurs à la compilation ...) et non le fichier **stdio.h** !

# II-B - Du fichier source à l'exécutable

## II-B-1 - La compilation

Avant de définir la compilation, il y a deux expressions qu'il faut déjà avoir défini. Il s'agit de "fichier source" et "fichier exécutable".

- Un **fichier source** est un fichier contenant, dans le cas qui nous intéresse, des lignes de code écrits en C. Le code contenu dans le fichier source est appelé du **code source**. Les fichiers sources C portent généralement l'extension "**.c**" (par exemple : hello.c).
- Un **fichier exécutable** est un fichier contenant, entre autres, du code directement exécutable par le processeur. Ce code est appelé **code machine**. L'extension portée par les fichiers exécutables varient selon le système. Sous Windows par exemple, ils portent généralement l'extension "**.exe**" (par exemple : hello.exe). Sous Linux, ils ne portent généralement pas d'extension.

La **compilation** c'est, en première approximation, le processus pendant lequel le fichier source (qui ne contient que du texte) est transformé en fichier exécutable (qui contient du code compréhensible par le processeur). Une définition plus précise sera donnée dans les cours plus avancés.
Puisque les programmes C doivent être compilés avant de pouvoir être exécutés, on dit que le C est un **langage compilé**. Le programme qui effectue la compilation est appelé **compilateur**.
Nombreux programmeurs utilisent cependant un **EDI** (Environnement de Développement Intégré) au lieu d'un simple compilateur pour compiler leurs programmes. Un EDI est un logiciel qui intègre un éditeur de texte pour taper les codes sources, un compilateur pour les traduire en exécutable ainsi que d'autres outils aidant à la mise au point et à la distribution des programmes. Il existe de nombreux EDIs pour le langage C. Certains sont gratuits (Code::Blocks Studio, Visual C++ Express, etc.), d'autres payants (CodeGear RAD Studio, Microsoft Visual Studio, etc.). Téléchargez ou achetez et installez le logiciel qui vous convient donc avant de continuer. Pour les manipulations pratiques, nous supposerons dans la suite que nous travaillons sous Windows, mais que cela ne vous empêche de lire ce tutoriel même si vous utilisez un autre système.

## II-B-2 - Applications consoles

Le C voulant être un langage entièrement portable, la création d'interfaces graphiques n'est pas supportée en standard par le langage. En effet, la notion d'interface graphique n'existe pas forcément dans tous les systèmes. Dans les systèmes proposant un environnement graphique, comme Windows par exemple, les applications qui en sont dépourvues sont souvent appelées **applications consoles**. Ces applications, quand elles sont lancées, s'exécutent dans une fenêtre (la fameuse "console") dans laquelle l'utilisateur n'utilise que le clavier pour commander le programme). Dans ce tutoriel, c'est toujours ce type d'applications que allons développer.

## II-B-3 - Manipulations avec Visual C++ et Code::Blocks

### II-B-3-a - Avec Visual C++ 6

1. Lancez **Visual C++**.
2. Créez un projet d'application console que vous allez nommer **helloworld** en choisissant **File > New > Project > Win32 Console Application**. Un dossier nommé helloworld est alors créé. Ce sera votre répertoire par défaut. Deux fichiers, entre autres, sont également créés : **helloworld.prj** (votre projet) et **helloworld.dsw** (votre espace de travail).
3. Choisissez **An empty project** pour qu'aucun fichier ne soit automatiquement ajouté à notre projet.
4. Ajoutez au projet un nouveau fichier que vous allez nommer **hello.c** avec la commande **Project > Add to Project > New > C++ source file**. Nommez-le bien hello.c car si vous omettez l'extension, VC6 va automatiquement ajouter l'extension .cpp et vous aurez donc un fichier source C++ (hello.cpp) au lieu d'un fichier source C, à moins bien sûr que c'est justement ce que vous cherchiez ...
5. Dans l'explorateur de projet (normalement c'est sur votre gauche), cliquez sur **File View** pour afficher une vue des fichiers qui constituent votre projet. Ouvrez ensuite le dossier **Source Files** puis double cliquez sur **hello.c** pour l'ouvrir dans l'éditeur.
6. Saisissez maintenant le code puis compilez avec la commande **Build > Build helloworld.exe (F7)**. Le fichier **helloworld\Debug\helloworld.exe** est alors créé.
7. Pour tester votre programme sans quitter VC, choisissez **Build > Execute helloworld.exe (Ctrl + F5)**.
   Le message de type "Appuyez sur une touche pour continuer" ne fait évidemment pas partie de votre programme. C'est juste un message provenant de votre EDI pour éviter la fermeture immédiate de la console à la fin de l'exécution du programme, ce qui ne vous aurait pas permis de "contempler" sa sortie.

### II-B-3-b - Avec Visual Studio .NET

La procédure est presque les même que sous VC6. Créez un nouveau projet d'application Windows (**Win32 Project**) puis dans l'étape **Applications Settings** de l'assistant, choissez **Console Application** puis **Empty Project**. Compilez puis testez.

## II-B-3-c - Avec Code::Blocks 1.0

1 Lancez **Code::Blocks**.
2 Créez un nouveau projet d'application console que vous aller nommer **helloworld** en choisissant **File > New Project > Console Application**. Enregistrez-le dans un répertoire de votre choix qui sera alors votre répertoire par défaut. Avant de valider, cochez l'option **Do not create any files** afin qu'aucun fichier ne soit automatiquement ajouté à notre projet. Une fois que vous avez validé, un fichier **helloworld.cbp** (votre projet) sera créé dans le répertoire de votre projet.
3 Créez un nouveau fichier que vous allez nommer **hello.c** avec la commande **File > New File**. Acceptez que ce fichier soit ajouté au projet.
4 Saisissez maintenant le code puis compilez avec la commande **Build > Build (Ctrl + F9)**. Le fichier **helloworld.exe** est alors créé.
5 Pour tester votre programme sans quitter Code::Blocks, choisissez **Build > Run (Ctrl + F10)**. Le message de type "Appuyez sur une touche pour continuer" ne fait évidemment pas partie de votre programme. C'est juste un message provenant de votre EDI pour éviter la fermeture immédiate de la console à la fin de l'exécution du programme, ce qui ne vous aurait pas permis de "contempler" sa sortie.

# II-C - Les fonctions

## II-C-1 - Introduction

Comme nous l'avons déjà dit plus haut, les fonctions sont les briques avec lesquelles on construit un programme en langage C. La vie d'un programmeur C se résume donc à créer puis utiliser des fonctions. Vous allez adorer ! Nous allons donc maintenant voir de plus près ce que c'est qu'une fonction.

## II-C-2 - Exemple avec une fonction "mathématique"

En mathématiques, on définit une fonction comme suit :

```
f(x) = x² - 3
```

Cela signifie que **f** est une **fonction** qui reçoit en **argument** un réel **x** et qui **retourne** un réel : $x^2 - 3$.
Ecrivons une fonction C que nous appellerons f, qui reçoit en argument un entier x et qui retourne également un entier : $x^2 - 3$.

```
int f(int x)
{
    return x*x - 3;
}
```

Le code ci-dessus constitue ce qu'on appelle la **définition** ou encore l'**implémentation** de la fonction f. Voyons maintenant un exemple d'utilisation de cette fonction.

```
#include <stdio.h>

int f(int); /* declaration de la fonction f */

int main()
{
    int x = 4;
    printf("f(%d) = %d\n", x, f(x));
    return 0;
}
```

```c
int f(int x)
{
    return x*x - 3;
}
```

La ligne :

```c
int f(int);
```

tient lieu de **déclaration** de la fonction f. Sans le point-virgule, on a ce qu'on appelle le **prototype** ou la signature de la fonction f. Ici, le prototype (la signature) indique que "f est une fonction qui nécessite en argument un int (int f(**int**)) et qui retourne un int (**int** f(int))". Un prototype suivi d'un point-virgule constitue ce qu'on appelle une déclaration (mais ce n'est pas la seule forme possible de déclaration d'une fonction).
Attention ! Nous avons vu jusqu'ici qu'une instruction élémentaire se termine par un point-virgule. Cela ne signifie en aucun cas que tout ce qui se termine par un point-virgule est donc une « instruction ». Une déclaration est une déclaration, pas une instruction. Nous définirons ce que c'est exactement une instruction encore plus loin.
Le **%d** dans la chaîne passée en premier argument de printf est ce qu'on appelle un **spécificateur de format**. Il renseigne sur la manière dont nous souhaitons afficher le texte. Ici, on veut afficher les nombres 4 et 13 (f(4)). Nous disons donc à printf d'utiliser le format « nombre entier » (%d) pour les afficher. Le premier %d correspond au format que nous voulons utiliser pour afficher x et le deuxième pour f(x). Nous y reviendrons un peu plus loin.
Sachez également que la variable x dans la fonction main n'a absolument rien à voir avec la variable x en paramètre de la fonction f. Chaque fonction peut avoir ses propres variables et ignore complètement ce qui se passe chez les autres fonctions. On vient ainsi d'introduire le concept de "variables locales" que nous étudierons également plus tard.

## II-C-3 - Exemple avec une "procédure"

Certains langages de programmation (Pascal, VB (Visual Basic), etc.) font la distinction entre "fonction" et "procédure". En langage C et ses dérivés, il n'y a pas cette distinction. Ce que chez certains langages on appelle **procédure** est en effet ni plus ni moins qu'une fonction qui effectue un traitement mais qui ne retourne pas de valeur, c'est-à-dire une fonction qui ne retourne rien. En langage C, ce "rien" est indiqué par le mot-clé **void** (terme anglais signifiant "vide"). Voici un exemple de fonction qui ne requiert aucun argument et qui ne retourne rien : une fonction qui affiche "Bonjour." 3 fois (en 3 lignes).

```c
#include <stdio.h>

void Bonjour3Fois(void);

int main()
{
    /* Ce programme affiche Bonjour 6 fois */
    Bonjour3Fois();
    Bonjour3Fois();
    return 0;
}

void Bonjour3Fois(void)
{
    printf("Bonjour.\n");
    printf("Bonjour.\n");
    printf("Bonjour.\n");
}
```

## II-C-4 - Remarques

**Dans une déclaration** on peut :

- Mettre le nom des arguments de la fonction (bon uniquement pour la déco et rien d'autre). Par exemple :

exemple de déclaration d'une fonction

```c
int Surface(int Longueur, int largeur);
```

exemple de déclaration d'une fonction

- Ne pas préciser les arguments de la fonction (déconseillé). Dans ce cas, il faut aller à la définition de la fonction pour connaître les arguments qu'elle requiert. Attention ! Ne pas préciser les arguments d'une fonction n'est pas la même chose que déclarer une fonction qui ne requiert aucun argument. Exemple :

exemple de déclaration incomplète d'une fonction (déconseillé)

```
int Surface(); /
* Surface est une fonction, point. Voir sa definition pour plus de details. */
```

exemple de déclaration d'une fonction qui ne requiert aucun argument

```
int Surface(void);
```

**De plus** on peut :

- Ne pas déclarer une fonction retournant un int (déconseillé).
- La déclaration d'une fonction n'est nécessaire que lorsque son utilisation précède sa définition. En effet, une fois définie, la fonction est entièrement connue et donc n'a plus besoin d'être déclarée. Il est cependant toujours conseillé de ne définir une fonction qu'après son utilisation (ce qui requiert donc une déclaration) ne serait-ce que pour la lisibilité du programme (en effet c'est le programme qu'on veut voir à première vue, pas les petits détails).

**Dans une définition**:

- On peut ne pas préciser le type de retour d'une fonction (déconseillé). Dans ce cas celle-ci est supposée retourner un int.
- Si la fonction a déjà été déclarée à l'aide d'une déclaration complète (c'est-à-dire avec prototype), une paire de parenthèses vide signifie que la fonction n'accepte aucun argument. Sinon, les parenthèses vides signifient que les arguments requis par la fonction ne sont pas renseignés. Cette manière de définir une fonction est à éviter sauf pour des fonctions qui ne sont jamais appelées depuis le programme comme main() par exemple.

## II-D - Les macros

### II-D-1 - Le préprocesseur

Avant d'être effectivement compilés, les fichiers sources C sont traités par un **préprocesseur** qui résout certaines directives qui lui sont données comme l'inclusion de fichier par exemple. Le préprocesseur, quoi qu'étant un programme à priori indépendant du compilateur, est un élément indispensable du langage.
Une directive donnée au préprocesseur commence toujours par **#**. Nous avons déjà rencontré la directive **include** qui permet d'inclure un fichier. La directive **define** permet de définir des **macros**.

### II-D-2 - Définition

Une **macro**, dans sa forme la plus simple, se définit de la manière suivante :

```
#define <macro> <le texte de remplacement>
```

Par exemple :

```
#define N 10
```

commande au préprocesseur de remplacer toutes les occurrences de le lettre N par 10. Bien sûr il n'y a pas que les lettres et les nombres qu'on peut utiliser, rien ne nous empêche par exemple d'écrire :

```
#define PLUS +
```

Ainsi, toutes les occurrences de PLUS seront remplacés par +.
Dans certaines circonstances, les macros peuvent également avantageusement remplacer les fonctions. Par exemple :

```
#define carre(x) x * x
```

Dans ce cas, une expression telle que

```
carre(3)
```

sera remplacé par :

```
3 * 3
```

Si cela semble fonctionner à première vue, voyons un peu ce qu'on obtiendrait avec :

```
carre(1 + 1)
```

Et bien, on obtiendrait :

```
1 + 1 * 1 + 1
```

soit 1 + 1 + 1 soit 3 car la multiplication est prioritaire par rapport à l'addition. Nous verrons plus tard l'ordre de priorité des opérateurs. C'est pourquoi on utilise généralement beaucoup de parenthèses dans le corps d'une macro. Ainsi, dans l'exemple précédent, on aurait du par exemple écrire :

```
#define carre(a) ((a) * (a))
```

Ce qui nous donnerait pour carre(1 + 1) par exemple :

```
((1 + 1) * (1 + 1))
```

## II-E - Exercices

### II-E-1 - La lettre X

Ecrire un programme qui dessine la lettre dans une grille de dimensions 5 x 5 en utilisant des espaces et le caractère '*'. Cela signifie qu'on veut avoir comme sortie :

```
*   *
 * *
  *
 * *
*   *
```

### II-E-2 - Périmètre d'un rectangle

Ecrire une fonction qui permet d'avoir le périmètre d'un rectangle de longueur et largeur données en arguments et utiliser cette fonction dans un programme arbitraire pour afficher le périmètre d'un rectangle de dimensions 100 x 60. Exemple de sortie :

```
Le perimetre d'un rectangle de longueur 100 et de largeur 60 est 320.
```

## III - Expressions et instructions

## III-A - Introduction aux types de données du langage C

### III-A-1 - Les types de base du langage C

Au niveau du processeur, toutes les données sont représentées sous leur forme binaire et la notion de **type** n'a pas de sens. Cette notion n'a été introduite que par les langages de haut niveau dans le but de rendre les programmes plus rationnels et structurés. Mais même parmi les langages de haut niveau, on distingue ceux qui sont dits fortement typés (qui offrent une grande variété de types), comme le Pascal par exemple, de ceux qui sont dits faiblement ou moyennement typés (et plus proches de la machine) comme le C par exemple. Le langage C ne dispose que de 4 types de base :

| Type C | Type correspondant |
|---|---|
| **char** | caractère (entier de petite taille) |
| **int** | entier |
| **float** | nombre flottant (réel) en simple précision |
| **double** | nombre flottant (réel) en double précision |

Devant char ou int, on peut mettre le modificateur **signed** ou **unsigned** selon que l'on veut avoir un entier signé (par défaut) ou non signé. Par exemple :

```
char ch;
unsigned char c;
unsigned int n; /* ou tout simplement : unsigned n */
```

La plus petite valeur possible que l'on puisse affecter à une variable de type entier non signé est 0 alors que les entiers signés acceptent les valeurs négatives.
Devant int, on peut mettre également **short** ou **long** auxquels cas on obtiendrait un entier court (**short int** ou tout simplement **short**) respectivement un entier long (**long int** ou tout simplement **long**). Voici des exemples de déclarations valides :

```
int n = 10, m = 5;
short a, b, c;
long x, y, z = -1;
unsigned long p = 2;
```

**long** peut être également mis devant **double**, le type résultant est alors **long double** (quadruple précision).

# III-A-2 - Règle d'écriture des constantes littérales

## Nombres entiers

Toute constante littérale « pure » de type entier (ex : 1, -3, 60, 40, -20, ...) est considérée par le langage comme étant de type **int**.
Pour expliciter qu'une constante littérale de type entier est de type **unsigned**, il suffit d'ajouter à la constante le suffixe **u** ou **U**. Par exemple : 2**u**, 30**u**, 40**U**, 50**U**, ...
De même, il suffit d'ajouter le suffixe **l** ou **L** pour expliciter qu'une constante littérale de type entier est de type **long** (on pourra utiliser le suffixe **UL** par exemple pour **unsigned long**).
Une constante littérale de type entier peut également s'écrire en octal (base 8) ou en hexadécimal (base 16). L'écriture en hexa est évidement beaucoup plus utilisée.
Une constante littérale écrite en **octal** doit être précédée de **0** (zéro). Par exemple : **0**12, **0**20, **0**30**UL**, etc.
Une constante littérale écrite en **hexadécimal** doit commencer par **0x** (zéro x). Par exemple **0x**30, **0x**41, **0x**61, **0x**FF**L**, etc.

## Nombres flottants

Toute constante littérale « pure » de type flottant (ex : 0.5, -1.2, ...) est considérée comme étant de type double.
Le suffixe **f** ou **F** permet d'expliciter un **float**. Attention, 1f n'est pas valide car 1 est une constante entière. Par contre 1.0f est tout à fait correcte. Le suffixe **l** ou **L** permet d'expliciter un **long double**.
Une constante littérale de type flottant est constituée, dans cet ordre :

- d'un signe (+ ou -)
- d'une suite de chiffres décimaux : la partie entière
- d'un point : le séparateur décimal
- d'une suite de chiffres décimaux : la partie décimale
- d'une des deux lettres e ou E : symbole de la puissance de 10 (notation scientifique)
- d'un signe (+ ou -)
- d'une suite de chiffres décimaux : la puissance de 10

Par exemple, les constantes littérales suivantes représentent bien des nombres flottants : 1.0, -1.1**f**, 1.6E-19, 6.02e23**L**, 0.5 3e8

## Caractères et chaînes de caractères

Les caractères sont placés entre simple quottes (ex : 'A', 'b', 'c', ...) et les chaînes de caractères entre double quottes (ex : "Bonjour", "Au revoir", ...). Certains caractères sont spéciaux et il faut utiliser la technique dite d'**échappement** pour les utiliser. En langage C, le caractère d'échappement est le caractère \. Exemples :

| Caractère | Valeur |
|---|---|
| '\t' | Le caractère 'tabulation' |
| '\r' | Le caractère 'retour chariot' |
| '\n' | Le caractère 'fin de ligne' |
| '\\' | Le caractère 'antislash' |

# III-A-3 - Spécification de format dans printf

## III-A-3-a - Généralités

Voici la liste des codes format que nous utiliserons les plus souvent par la suite :

| Code format | Utilisation |
|---|---|
| c | Afficher un caractère |
| d | Afficher un int |
| u | Afficher un unsigned int |
| x, X | Afficher un entier dans le format hexadécimal |
| f | Afficher un float ou un double en notation décimale |
| e | Afficher un float ou un double en notation scientifique avec un petit e |
| E | Afficher un float ou un double en notation scientifique avec un grand E |
| g, G | Afficher un float ou un double (utilise le format le plus adapté) |
| % | Afficher le caractère '%' |

Lorsqu'on affiche une donnée de type flottant dans le format "scientifique", il n'y a qu'un chiffre, ni moins ni plus, devant le point décimal (et ce chiffre est 0 si et seulement si la donnée vaut 0), le nombre de chiffres après ce point est variable et le nombre de chiffres utilisés pour représenter l'exposant dépend de l'implémentation mais il y en a toujours au moins deux.
D'autre part :

- h devant d ou u indique que l'argument est un short
- l devant d, u, x ou X indique que l'argument est de type long
- L devant f, e, E ou g indique que l'argument est de type long double

La spécification de format dans printf va cependant beaucoup plus loin qu'une simple spécification de type. En règle générale, une spécification de format a la structure suivante : **%[options]specificateur_de_type**. Les "options" ont été mis entre crochets car elles sont toujours facultatives. Les options disponibles dépendent du code de conversion (le "spécificateur de type") utilisé. Nous n'allons pas toutes les présenter (la documentation de référence est là pour ça) mais allons quand même voir quelques options que nous aurons assez souvent l'occasion d'utiliser.
Enfin, pour afficher un caractère, il vaut mieux utiliser putchar (ex : putchar('*')) qui est plus simple plutôt que printf.

## III-A-3-b - Les "options"

Une spécification de format dans printf a le plus souvent (cas pratique) la structure suivante :

```
%[[+][-][0]][gabarit][.precision][modificateur_de_type]specificateur_de_type
```

Les drapeaux + et 0 n'ont de sens que pour les données à imprimer sous forme numérique.
Découvrons le rôle des paramètres les plus utiles pour les types les plus utilisés.

## III-A-3-c - Le type entier

Pour le type entier (code de conversion d, u, x ou X) :

- h devant le spécificateur de type indique que l'argument attendu est de type short et l indique que l'argument est de type long.
- La précision indique le nombre maximum de caractères à utiliser pour imprimer la donnée.
- Le gabarit indique le nombre minimum de caractères à utiliser.
- Le drapeau 0 indique qu'il faut ajouter autant de 0 qu'il en faut avant le premier chiffre significatif de sorte que tout l'espace reservé au texte (défini par le gabarit) soit occupé.

- Le drapeau - indique que le texte doit être aligné à gauche. L'alignement par défaut est donc bien évidemment à droite.
- Le drapeau + indique qu'on veut que le signe du nombre soit toujours affiché, même si le nombre est positif ou nul.

Voici un programme proposant quelques applications :

```
#include <stdio.h>

int main()
{
    int n = 1234;

    printf("+------------+\n");
    printf("|%12d|\n", n); /* 12 caracteres minimum, alignement a droite */
    printf("+------------+\n");
    printf("|%-12d|\n", n); /* 12 caracteres minimum, alignement a gauche */
    printf("+------------+\n");
    printf("|%012d|\n", n); /* 12 caracteres minimum, caractere de remplissage : '0' */
    printf("+------------+\n");

    /* Affichage obligatoire du signe */

    printf("|%+12d|\n", n);
    printf("+------------+\n");
    printf("|%+-12d|\n", n);
    printf("+------------+\n");
    printf("|%+012d|\n", n);
    printf("+------------+\n");

    return 0;
}
```

Voici la sortie de ce programme :

```
+------------+
|        1234|
+------------+
|1234        |
+------------+
|000000001234|
+------------+
|       +1234|
+------------+
|+1234       |
+------------+
|+00000001234|
+------------+
```

## III-A-3-d - Le type flottant

Pour le type flottant (code de conversion f, e, E, g ou G) :

- L devant le spécificateur de type indique que l'argument attendu est de type long double.
- La précision indique le nombre de caractères à afficher après le point décimal pour les codes f, e et E (la valeur par défaut est 6) et le nombre maximum de chiffres significatifs pour les codes g et G.
- Le gabarit indique le nombre minimum de caractères à utiliser.
- Le drapeau 0 indique qu'il faut ajouter autant de 0 qu'il en faut avant le premier chiffre significatif de sorte que tout l'espace reservé au texte (défini par le gabarit) soit occupé.
- Le drapeau - indique que le texte doit être aligné à gauche. L'alignement par défaut est donc bien évidemment à droite.
- Le drapeau + indique qu'on veut que le signe du nombre soit toujours affiché, même si le nombre est positif ou nul.

Voici un programme proposant quelques applications :

```
#include <stdio.h>

int main()
{
    double x = 12.34;

    /* Affichage de flottant avec 4 chiffres apres le point decimal. */

    printf("%.4f\n", x);
    printf("%.4e\n", x);

    /* Affichage dans le format le plus approprie. */

    printf("%g\n", x);

    return 0;
}
```

Voici la sortie de ce programme (en supposant que l'implémentation utilise 3 chiffres pour imprimer l'exposant) :

```
12.3400
1.2340e+001
12.34
```

## III-A-4 - Les variables et les constantes

Si vous avez déjà fait un peu de math dans votre vie, vous devriez connaître ce que c'est qu'une variable. La notion est pratiquement la même en programmation. La déclaration d'une variable en langage C, dans sa forme la plus simple, se fait de la manière suivante :

```
<type> variable;
```

Par exemple :

```
int n;
```

Voici d'autres exemples beaucoup plus généralistes :

```
int a, b, c;
int i = 0, j, n = 10;
double x, y, z = 0.5;
```

Une déclaration de variable de la forme :

```
int n = 0;
```

est appelée une **déclaration avec initialisation**. Ce n'est pas la même chose que :

```
int n;
n = 0;
```

qui est une déclaration suivie d'une instruction d'affectation. Pour vous en convaincre rapidement, une variable constante (ou varaible en lecture seule) est une variable qui ne peut être qu'initialisée puis lue (la valeur d'une constante ne peut pas être modifiée). On obtient une **constante** en ajoutant le mot-clé **const** devant une déclaration. Par exemple :

```
const int n = 10;
```

On ne pourra jamais écrire :

```
n = <quoi que ce soit>;
```

Le mot-clé const être placé avant ou après le type ou encore avant le nom de la variable. Voici deux exemples :

```
const int n = 10;
int const m = 20;
```

## III-A-5 - Définition de nouveaux types

Le C dispose d'un mécanisme très puissant permettant au programmeur de créer de nouveaux types de données en utilisant le mot clé **typedef**. Par exemple :

```
typedef int ENTIER;
```

Définit le type ENTIER comme n'étant autre que le type int. Rien ne nous empêche donc désormais d'écrire par exemple :

```
ENTIER a, b;
```

Bien que dans ce cas, un simple #define aurait pu suffire, il est toujours recommandé d'utiliser typedef qui est beaucoup plus sûr.

# III-B - Les pointeurs

## III-B-1 - Définition

Comme nous le savons très bien, l'endroit où s'exécute un programme est la mémoire donc toutes les données du programme (les variables, les fonctions, ...) se trouvent en mémoire. Le langage C dispose d'un opérateur - **&** - permettant de récupérer l'**adresse** en mémoire d'une variable ou d'une fonction quelconque. Par exemple, si **n** est une variable, &n désigne l'**adresse de n**.
Le C dispose également d'un opérateur - ***** - permettant d'accéder au **contenu de la mémoire** dont l'adresse est donnée. Par exemple, supposons qu'on ait :

```
int n;
```

Alors les instructions suivantes sont strictement identiques.

```
n = 10;
```

```
*( &n ) = 10;
```

Un **pointeur** (ou une variable de type pointeur) est une variable destinée à recevoir une adresse. On dit alors qu'elle **pointe** sur un emplacement mémoire. L'accès au contenu de la mémoire se fait par l'intermédiaire de l'opérateur *.
Les pointeurs en langage C sont typés et obéissent à l'**arithmétique des pointeurs** que nous verrons un peu plus loin. Supposons que l'on veuille créer une variable - p - destinée à recevoir l'adresse d'une variable de type int. p s'utilisera alors de la façon suivante :

```
p = &n;
...
*p = 5;
...
```

*p est un int. Vous suivez ? Alors comment devrait-on déclarer le pointeur ? Si vous avez bien suivi, vous auriez répondu :

```
int *p;
```

Et de plus, si vous avez très bien suivi, vous auriez certainement ajouté : et cela est strictement équivalent à :

```
int * p;
```

Car en C, les espaces sont totalement gratuites ! Donc voilà, le type d'un **pointeur sur int** est **int ***. Mais cela peut être moins évident dans certaines déclarations. Par exemple, dans :

```
/* cas 1 */
int * p1, p2, p3;
```

seul p1 est de type int * ! Le reste, p2 et p3, sont de type int. Si vous voulez obtenir 3 pointeurs, vous devrez utilisez la syntaxe suivante :

```
/* cas 2 */
int *p1, *p2, *p3;
```

Ou avec un typedef :

```
typedef int * PINT;
PINT p1, p2, p3;
```

Par contre cela n'aurait pas marché si on avait défini PINT à l'aide d'un #define car cela nous amènerait au cas 1.

## III-B-2 - Saisir des données tapées au clavier avec la fonction scanf

On va encore faire un petit exercice de logique, plutôt de bon sens.
Si nous voulons afficher un entier par exemple, que doit-on fournir à la fonction d'affichage (printf par exemple) : la valeur de l'entier que nous voulons afficher ou l'adresse de la variable contenant le nombre que nous voulons afficher ?
La bonne réponse est : l'entier que nous voulons afficher bien sûr, autrement dit sa valeur. Personne n'a demandé son adresse !
Maintenant, si nous voulons demander à l'utilisateur (celui qui utilise notre programme) de taper un nombre puis ranger le nombre ainsi tapé dans une variable, que devons-nous fournir à la fonction permettant la saisie : l'adresse de la variable dans laquelle nous souhaitons ranger le nombre que l'utilisateur a entré, ou la valeur que contient actuellement cette variable ?
La bonne réponse est bien sûr : l'adresse de la variable dans laquelle nous souhaitons ranger le nombre entré
La fonction **scanf**, déclarée dans le fichier **stdio.h**, permet de lire des données formatées sur l'**entrée standard**, par défaut le clavier.
Voici un programme qui demande à l'utilisateur d'entrer 2 nombres puis affiche leur somme :

```
#include <stdio.h>

int main()
{
    int a, b, c;

    printf("Ce programme calcule la somme de 2 nombres.\n");

    printf("Entrez la valeur de a : ");
    scanf("%d", &a);

    printf("Entrez la valeur de b : ");
    scanf("%d", &b);

    c = a + b;
```

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

    return 0;
}
```

Mais faites gaffe avec la fonction scanf ! "%d" par exemple n'a rien à voir avec " %d ". En effet dans ce dernier cas, le format attendu par la fonction est :

```
<un espace> <un entier> <puis un autre espace>
```

Où espace désigne en fait n'importe quel caractère ou série de caractères blancs (espace, tabulation, etc.). Voici un exemple beaucoup plus convaincant :

```
#include <stdio.h>

int main()
{
    int mon_age = 23, votre_age;

    printf("Bonjour, j'ai %d ans. Et vous? ", mon_age);
    scanf("%d ans", &votre_age);
    printf("Ah! vous avez %d ans?\n", votre_age);

    return 0;
}
```

L'utilisateur devra alors taper par exemple :

```
22 ans
```

Peu importe le nombre de caractères blancs entre 22 et ans.
C'est ce qu'on appelle une **saisie formatée**. scanf retourne le nombre de conversions de format réussies. La valeur de retour de scanf permet donc déjà de voir si l'utilisateur a bien respecté le format attendu ou non. Dans notre exemple, il n'y a qu'une seule conversion de format à faire (%d). Si l'utilisateur respecte bien le format, scanf retournera 1.
Les fonctions telles que scanf sont plutôt destinées à être utilisées pour lire des données provenant d'un programme sûr (par l'intermédiaire d'un fichier par exemple), pas celles provenant d'un humain, qui sont sujettes à l'erreur. Les codes format utilisés dans scanf sont à peu près les mêmes que dans printf, sauf pour les flottants notamment.

| Code format | Utilisation |
|---|---|
| f, e, g | float |
| lf, le, lg | double |
| Lf, Le, Lg | long double |

Voici un programme qui permet de calculer le volume d'un cône droit à base circulaire selon la formule : V = 1/3 * (B * h) où B est la surface de base soit pour une base circulaire : B = PI*$R^2$, où R est le rayon de la base.

```
#include <stdio.h>

double Volume(double r_base, double hauteur);

int main()
{
    double R, h, V;

    printf("Ce programme calcule le volume d'un cone.\n");

    printf("Entrez le rayon de la base : ");
    scanf("%lf", &R);

    printf("Entrez la hauteur du cone : ");
    scanf("%lf", &h);
```

```
    V = Volume(R, h);
    printf("Le volume du cone est : %f", V);

    return 0;
}

double Volume(double r_base, double hauteur)
{
    return (3.14 * r_base * r_base * hauteur) / 3;
}
```

## III-B-3 - Exemple de permutation des contenus de deux variables

Voici un programme simple qui permute le contenu de deux variables.

```
include <stdio.h>

int main()
{
    int a = 10, b = 20, c;

    /* on sauvegarde quelque part le contenu de a */
    c = a;

    /* on met le contenu de b dans a */
    a = b;

    /* puis le contenu de a que nous avons sauvegardé dans c dans b */
    b = c;

    /* affichons maintenant a et b */
    printf("a = %d\nb = %d\n", a, b);

    return 0;
}
```

Maintenant, réécrivons le même programme en utilisant une fonction. Cette fonction doit donc pouvoir **localiser** les variables en mémoire autrement dit nous devons passer à cette fonction les adresses des variables dont on veut permuter le contenu.

```
#include <stdio.h>

void permuter(int * addr_a, int * addr_b);

int main()
{
    int a = 10, b = 20;

    permuter(&a, &b);
    printf("a = %d\nb = %d\n", a, b);

    return 0;
}

void permuter(int * addr_a , int * addr_b)
/***************\
* addr_a <-- &a *
* addr_b <-- &b *
\***************/
{
    int c;

    c = *addr_a;
    *addr_a = *addr_b;
    *addr_b = c;
}
```

# III-C - Les expressions

## III-C-1 - Introduction

En langage C, la notion d'**expression** est beaucoup plus riche que dans n'importe quel autre langage. De ce fait en donner une définition est longue et nous allons donc plutôt l'introduire (comme nous l'avons d'ailleurs toujours fait) de manière intuitive. Pour cela, analysons l'instruction suivante :

```
c = a + b;
```

Dans cette instruction (où nous supposons que les 3 variables sont tous de type int), il y a tout d'abord **évaluation** de a + b, puis **affection** du résultat dans c. On dit que :

```
a + b
```

est une expression, plus précisément une **expression de type int** (car a et b sont de type int) et dont la **valeur** est la somme de a et b (a + b), et que :

```
c = a + b;
```

est une instruction (remarquez bien le point-virgule).
L'expression a + b est composée de deux expressions à savoir a et b. On dit alors que c'est une **expression complexe**. En langage C, les **opérateurs** permettent de construire des expressions complexes. Dans notre exemple, l'**opérateur d'addition** à savoir + nous a permis de construire une expression complexe, a + b, à partir des expressions simples a et b.
Or en langage C, **=** est également un opérateur, l'**opérateur d'affectation**, et permet donc de construire des expressions complexes. Ainsi, dans notre exemple :

```
c = a + b
```

est une expression (remarquez bien l'absence de point-virgule, sinon ce serait une instruction) et dont la valeur et le type sont celle et celui de l'expression situé à gauche à savoir c.

## III-C-2 - lvalue et rvalue

Une **lvalue** (de **l**eft **value**) est tout ce qui peut figurer à gauche de l'opérateur d'affectation. Une variable, par exemple, est une lvalue. Si p est un pointeur (disons vers int), p est une lvalue (en effet un pointeur est une variable comme toutes les autres) et *p est également une lvalue.
Une **rvalue** (de **r**ight **value**) est tout ce qui peut figurer à droite de l'opérateur d'affectation. Une variable, une constante littérale, l'adresse d'une variable en sont des exemples. Si a est une variable, disons de type int, alors a est une lvalue tandis que a, -a ou a + 1 par exemple sont des rvalues. Cet exemple nous montre bien qu'une variable est à la fois une lvalue et une rvalue.
Une rvalue possède une valeur mais pas une adresse. Ainsi par exemple : a étant une lvalue, &a a bien un sens (désigne l'adresse mémoire de la variable a). Par contre, une expression insensée telle que &(-a) ou &(a + 1) sera rejetée (avec plaisir) par le compilateur.

## III-C-3 - Opérations usuelles

### III-C-3-a - Les opérateurs arithmétiques courants

Les opérateurs arithmétiques courants +, -, * et / existent en langage C. Toutefois, la division entière est un tout petit peu délicate. En effet, si a et b sont des entiers, a / b vaut le **quotient** de a et b c'est-à-dire par exemple, 29 /

5 vaut 5. Le reste d'une division entière s'obtient avec l'opérateur **modulo** %, c'est-à-dire, en reprenant l'exemple précédent, 29 % 5 vaut 4.

## III-C-3-b - Les opérateurs de comparaison

Les opérateurs de comparaison sont

| Opérateur | Rôle |
|---|---|
| < | Inférieur à |
| > | Supérieur à |
| == | Egal à |
| <= | Inférieur ou égal à |
| >= | Supérieur ou égal à |
| != | Différent de |

Par exemple, l'expression :

```
1 < 1000
```

est vraie. Vraie ? Et que vaut « vrai » au juste ? Quel est le type d'une expression « vraie » ? En langage C, la valeur d'une expression « vraie » est non nulle et celle d'une expression « fausse » zéro. Et réciproquement, toute valeur non nulle (24, -107, 2.9, ...) peut être interprétée comme vrai et zéro comme faux. Par exemple :

```
int prop;
prop = (1 < 1000); /* alors prop = VRAI */
```

## III-C-3-c - Les opérateurs logiques

On peut construire des expressions logiques complexes à l'aide des opérateurs

| Opérateur | Rôle |
|---|---|
| && | ET |
| \|\| | OU |
| ! | NON |

Par exemple :

```
int prop1, prop2, prop_ou, prop_et, prop_vrai;
prop1 = (1 < 1000);
prop2 = (2 == -6);
prop_ou = prop1 || prop2; /* VRAI, car prop1 est VRAI */
prop_et = prop1 && prop2; /* FAUX, car prop2 est FAUX */
prop_vrai = prop1 && !prop_2 /* VRAI car prop1 et !prop2 sont VRAI */
```

Les opérateurs logiques jouissent des propriétés suivantes :

- Dans une opération ET, l'évaluation se fait de gauche à droite. Si l'expression à gauche de l'opérateur est fausse, l'expression à droite ne sera plus évaluée car on sait déjà que le résultat de l'opération sera toujours FAUX.
- Dans une opération OU, l'évaluation se fait de gauche à droite. Si l'expression à gauche de l'opérateur est vrai, l'expression à droite ne sera plus évaluée car on sait déjà que le résultat de l'opération sera toujours VRAI.

## III-C-4 - L'opérateur virgule

On peut séparer plusieurs expressions à l'aide de l'opérateur virgule. Le résultat est une expression dont la valeur est celle de l'expression la plus à droite. L'expression est évaluée de gauche à droite. Par exemple, l'expression :

```
(a = -5, b = 12, c = a + b) * 2
```

vaut 14.

## III-C-5 - Taille des données. L'opérateur sizeof

La taille d'une donnée désigne la taille, en octets, que celle-ci occupe en mémoire. Par extension de cette définition, la taille d'un type de données désigne la taille d'une donnée de ce type. Attention ! **octet** désigne ici, par abus de langage, la taille d'un élément de mémoire sur la machine cible (la machine abstraite), c'est-à-dire la taille d'une case mémoire (qui vaut 8 bits dans la plupart des architectures actuelles), et non un groupe de 8 bits. En langage C, un octet (une case mémoire) est représenté par un char. La taille d'un char n'est donc pas forcément 8 bits, même si c'est le cas dans de nombreuses architectures, mais dépendante de la machine. La norme requiert toutefois qu'un char doit faire au moins 8 bits et que la macro **CHAR_BIT**, déclarée dans **limits.h**, indique la taille exacte d'un char sur la machine cible.
Le C dispose d'un opérateur, **sizeof**, permettant de connaître la taille, en octets, d'une donnée ou d'un type de données. La taille d'un char vaut donc évidemment 1 puisqu'un char représente un octet. Par ailleurs, il ne peut y avoir de type dont la taille n'est pas multiple de celle d'un char. Le type de la valeur retournée par l'opérateur sizeof est **size_t**, déclaré dans **stddef.h**, qui est inclus par de nombreux fichiers d'en-tête dont stdio.h.
Comme nous l'avons déjà dit plus haut, la taille des données est dépendante de la machine cible. En langage C, la taille des données n'est donc pas fixée. Néanmoins la norme stipule qu'on doit avoir :

```
sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeofd(long)
```

Sur un processeur Intel (x86) 32 bits par exemple, un char fait 8 bits, un short 16 bits, et les int et les long 32 bits.

## III-C-6 - Les opérateurs d'incrémentation et de décrémentation

Il s'agit des opérateurs ++ (**opérateur d'incrémentation**) et -- (**opérateur de décrémentation**). Ils existent sous forme **préfixée** ou **postfixée**. L'opérande doit être une lvalue de type numérique ou pointeur. Soient i et j deux variables de type int :

```
j = i++;
```

est équivalent à :

```
j = i;
i = i + 1;
```

On voit donc bien que la valeur de i++ est celle de i avant l'évaluation effective de l'expression (**post incrémentation**). Inversement :

```
j = ++i;
```

est équivalent à :

```
i = i + 1;
j = i;
```

Où l'on voit bien que la valeur de ++i est celle de i après l'évaluation effective de l'expression (**pré incrémentation**).

L'opérateur -- s'utilise de la même manière que ++ mais l'opérande est cette fois-ci décrémentée.

### III-C-7 - Expressions conditionnelles

Une expression conditionnelle est une expression dont la valeur dépend d'une condition. L'expression :

```
p ? a : b
```

vaut a si p est vrai et b si p est faux.

### III-C-8 - Autres opérateurs d'affectation

Ce sont les opérateurs : +=, -=, *=, /=, ...

```
x += a;
```

par exemple est équivalent à :

```
x = x + a;
```

### III-C-9 - Ordre de priorité des opérateurs

Les opérateurs sont classés par ordre de priorité. Voici les opérateurs que nous avons étudiés jusqu'ici classés dans cet ordre. Opérateur Associativité

| Opérateur | Associativité |
|---|---|
| Parenthèses | de gauche à droite |
| ! ++ -- - (signe) sizeof | de gauche à droite |
| * / % | de gauche à droite |
| + - | de gauche à droite |
| < <= > >= | de gauche à droite |
| == != | de gauche à droite |
| & (adresse de) | de gauche à droite |
| && | de gauche à droite |
| \|\| | de gauche à droite |
| Opérateurs d'affectation (= += ...) | de droite à gauche |
| , | de gauche à droite |

Ce n'est pas parce que cet ordre existe qu'il faut le retenir par coeur. Pour du code lisible, il est même conseillé de ne pas trop en tenir compte et d'utiliser des parenthèses dans les situations ambiguës.

## III-D - Considérations liées à la représentation binaire

### III-D-1 - Généralités

L'ordinateur traite les données uniquement sous forme numérique. A l'intérieur de l'ordinateur, ces nombres sont représentés en binaire. La manière de représenter une donnée dans la mémoire de l'ordinateur est appelée **codage**. Pour bien comprendre le C, vous devez connaître comment les données sont représentées en mémoire, c'est-à-dire sous leur forme binaire. Voir à ce sujet votre cours de numérique.

### III-D-2 - Les caractères

La représentation numérique des caractères définit ce qu'on appelle un jeu de caractères. Par exemple, dans le jeu de caractères **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) qui est un jeu de caractères qui n'utilise que 7 bits et qui est à la base de nombreux codes populaires de nos jours, le caractère 'A' est représenté par le code 65, le caractère 'a' par 97 et '0' par 48. Hélas, même ASCII n'est pas adopté par le langage C. En effet si le C dépendait d'un jeu de caractères particulier, il ne serait alors pas totalement portable. Le standard définit néanmoins un certain nombre de caractères que tout environnement compatible avec le C doit posséder parmi lesquels les 26 lettres de l'alphabet latin (donc en fait 52 puisqu'on différencie les majuscules et les minuscules), les 10 chiffres décimaux, les caractères # < > ( ) etc. Le programmeur (mais pas le compilateur) n'a pas besoin de connaître comment ces caractères sont représentés dans le jeu de caractères de l'environnement. Le standard ne définit donc pas un jeu de caractères mais seulement un ensemble de caractères que chaque environnement compatible est libre d'implémenter à sa façon (plus éventuellement les caractères spécifiques à cet environnement). La seule contrainte imposée est que leur valeur doit pouvoir tenir dans un char.
Concernant la technique d'échappement, sachez également qu'on peut insérer du code octal (commençant par 0) ou hexadécimal (commençant par x) après le caractère d'échappement \ pour obtenir un caractère dont le code dans le jeu de caractères est donné. L'hexadécimal est de loin le plus utilisé. Par exemple : '\x30', '\x41', '\x61', ... Et enfin pour les caractères de code 0, 1, ... jusqu'à 7, on peut utiliser les raccourcis '\0', '\1', ... '\7'.

### III-D-3 - Dépassement de capacité

Le dépassement de capacité a lieu lorsqu'on tente d'affecter à une lvalue une valeur plus grande que ce qu'elle peut contenir. Par exemple, en affectant une valeur sur 32 bits à une variable ne pouvant contenir que 16 bits.

## III-E - La conversion de type

### III-E-1 - Conversion implicite

En langage C, des règles de **conversion** dite **implicite** s'appliquent aux données qui composent une expression complexe lorsqu'ils ne sont pas de même type (entier avec un flottant, entier court avec entier long, entier signé avec un entier non signé, etc.). Par exemple, dans l'expression :

```
'A' + 2
```

'A' est de type char et 2 de type int. Dans ce cas, 'A' est tout d'abord converti en int avant que l'expression ne soit évaluée. Le résultat de l'opération est de type int (car un int + un int donne un int). Ici, il vaut 67 (65 + 2). En fait, les char et les short sont toujours systématiquement convertis en int c'est-à-dire que dans l'addition de deux char par exemple, tous deux sont tout d'abord convertis en int avant d'être additionnés, et le résultat est un int (pas un char). Un unsigned char sera converti en unsigned int, et ainsi de suite.
En règle générale : le type le plus « faible » est convertit dans le type le plus « fort ». Par exemple, les entiers sont plus faibles que les flottants donc 1 mélangé à un flottant par exemple sera tout d'abord converti en 1.0 avant que l'opération n'ait effectivement lieu.

### III-E-2 - Conversion explicite (cast)

Il suffit de préciser le type de destination entre parenthèses devant l'expression à convertir. Par exemple :

```
float f;
f = (float)3.1416;
```

Dans cet exemple, on a converti explicitement 3.1416, qui est de type double, en float. Lorsqu'on affecte un flottant à un entier, seule la partie entière, si elle peut être représentée, est retenue.

# III-F - Les instructions

## III-F-1 - Introduction

L'**instruction** est la plus petite unité qui compose une fonction. Le seul type d'instruction que nous avons vu jusqu'ici est l'**expression-instruction** qui n'est autre qu'une expression suivie d'un point-virgule. Comprendre les notions d'expression et d'instruction est indispensable pour bien comprendre et maîtriser la syntaxe du langage C. Nous allons maintenant généraliser la notion d'instruction en langage C.

## III-F-2 - Bloc d'instructions

Un **bloc**, abusivement appelé bloc d'instructions, ou plus précisément une **instruction-bloc** consiste en un groupe de **déclarations** et d'**instructions** (qui peuvent être également des blocs). Un bloc est délimité par des **accolades**. Une fonction est toujours constituée d'une instruction-bloc.
Une variable déclarée à l'intérieur d'un bloc est locale à (c'est-à-dire n'est connue qu'à l'intérieur de) ce bloc et masque toute variable de même nom déclarée à l'extérieur de celui-ci (une variable globale est une variable déclarée à l'extérieur de toute fonction et est connue partout dans le fichier source). D'autre part, une déclaration doit toujours se faire en début de bloc. Les arguments effectifs d'une fonction sont également des variables locales à cette fonction.
Voici un exemple de programme (présentant peu d'intérêt je le conçois) comportant plusieurs blocs :

```c
#include <stdio.h>

int main()
{
    printf("Les instructions en langage C\n");
    printf("-----------------------------\n\n");

    {
        /* Un bloc d'instructions */
        printf("**********\n");
        printf("* Bloc 1 *\n");
        printf("**********\n");
    }

    putchar('\n');

    {
        /* Un autre bloc */
        printf("**********\n");
        printf("* Bloc 2 *\n");
        printf("**********\n");

        {
            /* Encore un autre bloc */
            printf("**********\n");
            printf("* Bloc 3 *\n");
            printf("**********\n");
        }
    }

    return 0;
}
```

Retenez donc bien ceci : **un bloc constitue bien une et une seule instruction**. Bien que cette instruction soit en fait un bloc d'instructions (et de déclarations), il est conseillé de la voir comme une et une seule instruction plutôt que comme un bloc d'instructions. Une instruction-bloc ne se termine pas par un point-virgule. Conclusion : Il est faux de dire qu'une instruction doit toujours se terminer par un point-virgule. Le point-virgule marque tout simplement la fin de certaines instructions, dont les expressions-instructions.

## III-F-3 - L'instruction if

Permet d'effectuer des choix conditionnels. La syntaxe de l'instruction est la suivante :

```
if ( <expression> )
    <une et une seule instruction>
else
    <une et une seule instruction>
```

Une instruction if peut ne pas comporter de else. Lorsqu'on a plusieurs instructions if imbriquées, un else se rapporte toujours au dernier if suivi d'une et une seule instruction. Par exemple : écrivons un programme qui compare deux nombres.

```
#include <stdio.h>

int main()
{
    int a, b;

    printf("Ce programme compare deux nombres.\n");

    printf("Entrez la valeur de a : ");
    scanf("%d", &a);

    printf("Entrez la valeur de b : ");
    scanf("%d", &b);

    if (a < b)
        printf("a est plus petit que b.\n");
    else
        if (a > b)
            printf("a est plus grand que b.\n");
        else
            printf("a est egal a b.\n");

    return 0;
}
```

## III-F-4 - L'instruction do

Permet d'effectuer une boucle. La syntaxe de l'instruction est la suivante :

```
do
    <une et une seule instruction>
while ( <expression> );
```

L'instruction do permet d'exécuter une instruction tant que <expression> est vraie. **Le test est fait après chaque exécution de l'instruction**. Voici un programme qui affiche 10 fois Bonjour.

```
#include <stdio.h>

int main()
{
    int nb_lignes_affichees = 0;

    do
        {
            printf("Bonjour.\n");
            nb_lignes_affichees++;
        }
    while (nb_lignes_affichees < 10);

    return 0;
}
```

## III-F-5 - L'instruction while

Permet d'effectuer une boucle. La syntaxe de l'instruction est la suivante :

```
while ( <expression> )
    <une et une seule instruction>
```

L'instruction while permet d'exécuter une instruction tant que <expression> est vraie. **Le test est fait avant chaque exécution de l'instruction**. Donc si la condition (<expression>) est fausse dès le départ, la boucle ne sera pas exécutée.

## III-F-6 - L'instruction for

Permet d'effectuer une boucle. La syntaxe de l'instruction est la suivante :

```
for ( <init> ; <condition> ; <step>)
    <instruction>
```

Elle est pratiquement identique à :

```
<init>;
while ( <condition> )
{
    <instruction>
    <step>
}
```

Par exemple, écrivons un programme qui affiche la table de multiplication par 5.

```
#include <stdio.h>

int main()
{
    int n;

    for(n = 0; n <= 10; n++)
        printf("5 x %2d %2d\n", n, 5 * n);

    return 0;
}
```

## III-F-7 - Les instructions switch et case

Ces instructions permettent d'éviter des instructions if trop imbriquées comme illustré par l'exemple suivant :

```
#include <stdio.h>

int main()
{
    int n;

    printf("Entrez un nombre entier : ");
    scanf("%d", &n);

    switch(n)
    {
    case 0:
        printf("Cas de 0.\n");
        break;

    case 1:
```

```
        printf("Cas de 1.\n");
        break;

    case 2: case 3:
        printf("Cas de 2 ou 3.\n");
        break;

    case 4:
        printf("Cas de 4.\n");
        break;

    default:
        printf("Cas inconnu.\n");
    }

    return 0;
}
```

## III-F-8 - L'instruction break

Permet de sortir immédiatement d'une boucle ou d'un switch. La syntaxe de cette instruction est :

```
break;
```

## III-F-9 - L'instruction continue

Dans une boucle, permet de passer immédiatement à l'itération suivante. Par exemple, modifions le programme table de multiplication de telle sorte qu'on affiche rien pour n = 4 ou n = 6.

```
#include <stdio.h>

int main()
{
    int n;

    for(n = 0; n <= 10; n++)
    {
        if ((n == 4) || (n == 6))
            continue;

        printf("5 x %2d %2d\n", n, 5 * n);
    }

    return 0;
}
```

## III-F-10 - L'instruction return

Permet de terminer une fonction. La syntaxe de cette instruction est la suivante :

```
return <expression>; /* termine la fonction et retourne <expression> */
```

ou :

```
return; /* termine la fonction sans spécifier de valeur de retour */
```

## III-F-11 - L'instruction vide

Une instruction peut être vide. Par exemple :

```
/* Voici une instruction vide * / ;
/* Voici une autre instruction vide */ ;
{ /* Encore une instruction vide */ }
```

# III-G - Exercices

## III-G-1 - Valeur absolue

Ecrire un programme qui demande à l'utilisateur d'entrer un nombre et qui affiche ensuite la "valeur absolue" de ce nombre. Il y a plusieurs façons de définir la valeur absolue d'un nombre x mais nous utiliserons la suivante : c'est x si x est supérieur ou égal à 0 et -x si x est inférieur à 0. La valeur absolue d'un nombre est donc un nombre toujours positif ou nul. Voici un exemple d'exécution :

```
Ce programme permet de determiner la valeur absolue d'un nombre.
Entrez un nombre : -6
La valeur absolue de -6 est 6.
Merci d'avoir utilise ce programme. A bientot !
```

## III-G-2 - Moyenne

Ecrire un programme qui demande à l'utilisateur d'entrer une note. La note doit être comprise entre 0 et 20 sinon le programme affichera "Cette note n'est pas valide" puis se terminera. Après que l'utilisateur ait entré une note valide, si la note est supérieure ou égale à 10, le programme affichera "Vous avez eu la moyenne." sinon, il affichera "Vous n'avez pas eu la moyenne.". Voici quelques exemples d'exécution :

```
Ce programme permet de determiner si vous avez eu la moyenne ou non.
Entrez votre note (0 a 20) : 12
Vous avez eu la moyenne.
Merci d'avoir utilise ce programme. A bientot !
```

```
Ce programme permet de determiner si vous avez eu la moyenne ou non.
Entrez votre note (0 a 20) : 22
Cette note n'est pas valide.
Merci d'avoir utilise ce programme. A bientot !
```

## III-G-3 - L'heure dans une minute

Ecrire un programme qui demande à l'utilisateur d'entrer l'heure et la minute actuelles et qui affiche ensuite l'heure et la minute qu'il fera une minute après. Voici quelques exemples d'exécution :

```
Ce programme permet de determiner l'heure qu'il sera une minute plus tard.
Entrez l'heure actuelle : 14
Entrez la minute actuelle : 38
Dans une minute il sera : 14:39.
Merci d'avoir utilise ce programme. A bientot !
```

```
Ce programme permet de determiner l'heure qu'il sera une minute plus tard.
Entrez l'heure actuelle : 23
Entrez la minute actuelle : 59
Dans une minute il sera : 00:00.
Merci d'avoir utilise ce programme. A bientot !
```

Pour simplifier l'exercice, on supposera que les valeurs entrées par l'utilisateur seront toujours correctes.

## III-G-4 - Rectangle

a. Ecrire une fonction affiche_car qui reçoit en arguments un caractère c et un entier n - void affiche_car(char c, int n) - et qui imprime ce caractère n fois. Utilisez cette fonction pour écrire un programme qui demande à l'utilisateur d'entrer un nombre n puis qui affiche une ligne de longueur n en utilisant le caractère '*', c'est-à-dire une ligne composée de n '*'. n doit être compris entre 0 et 20. Voici un exemple d'exécution :

```
Ce programme dessine une ligne.
Entrez la longueur de la ligne (0 a 20) : 10
**********
Merci d'avoir utilise ce programme. A bientot !
```

b. En utilisant la fonction affiche_car, écrire un programme qui demande à l'utilisateur d'entrer deux nombres L et h puis qui dessine un rectangle de longueur L et de hauteur h à l'aide du caractère '*'. L et h doivent être compris entre 0 et 20. Voici un exemple d'exécution :

```
Ce programme dessine un rectangle plein.
Entrez la longueur du rectangle (0 a 20) : 10
Entrez la hauteur du rectangle (0 a 20) : 4
**********
**********
**********
**********
Merci d'avoir utilise ce programme. A bientot !
```

c. Reprendre l'exercice b en dessinant cette fois ci un rectanle creux. Voici un exemple d'exécution :

```
Ce programme dessine un rectangle creux.
Entrez la longueur du rectangle (0 a 20) : 10
Entrez la hauteur du rectangle (0 a 20) : 4
**********
*        *
*        *
**********
Merci d'avoir utilise ce programme. A bientot !
```

## III-G-5 - Triangle isocèle

a. Ecrire un programme qui demande à l'utilisateur d'entrer un nombre h puis qui dessine un triangle isocèle de hauteur h et de surface minimale à l'aide du caractère '*'. h doit être compris entre 0 et 20. b. Afficher après le dessin le nombre d'étoiles affichées. Voici un exemple d'exécution :

```
Ce programme dessine un triangle.
Entrez la hauteur du triangle (0 a 20) : 4
   *
  ***
 *****
*******
Il y a 16 etoile(s) affichee(s).
Merci d'avoir utilise ce programme. A bientot !
```

## III-G-6 - Somme

Ecrire un programme qui demande à l'utilisateur d'entrer au plus 10 nombres puis qui affiche la somme des nombres entrés. L'utilisateur peut à tout moment terminer sa liste en entrant 0. Voici quelques exemples d'utilisation :

```
Ce programme permet de calculer une somme.
Entrez la liste des nombres a additionner (terminez en entrant 0).
1
3
```

```
5
7
0
Le total est : 16.
Merci d'avoir utilise ce programme. A bientot !
```

```
Ce programme permet de calculer une somme.
Entrez la liste des nombres a additionner (terminez en entrant 0).
1
2
3
4
5
6
7
8
9
10
Le total est : 55.
Merci d'avoir utilise ce programme. A bientot !
```

# IV - Tableaux, pointeurs et chaînes de caractères

## IV-A - Les tableaux

### IV-A-1 - Définition

Un **tableau** est un regroupement d'une ou plusieurs données de même type contigus en mémoire. L'accès à un élément du tableau se fait par un système d'indice, l'indice du premier élément étant 0. Par exemple :

```
int t[10];
```

déclare un tableau de 10 éléments (de type int) dont le nom est t. Les éléments du tableau vont donc de t[0], t[1], t[2] ... à t[9]. t est une variable de type tableau, plus précisément (dans notre cas), une variable de type **tableau de 10 int** (int [10]). Les éléments du tableau sont des int. Ils peuvent être utilisés comme n'importe quelle variable de type int.

### IV-A-2 - Initialisation

On peut initialiser un tableau à l'aide des accolades. Par exemple :

```
int t[10] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};
```

Bien évidemment, on n'est pas obligé d'initialiser tous les éléments, on aurait donc pu par exemple nous arrêter après le 5ème élément, et dans ce cas les autres éléments du tableau seront automatiquement initialisés à 0. Attention ! une variable locale non initialisée contient « n'importe quoi », pas 0 !
Lorsqu'on déclare un tableau avec initialisation, on peut ne pas spécifier le nombre d'éléments car le compilateur le calculera automatiquement. Ainsi, la déclaration :

```
int t[] = {0, 10, 20, 30};
```

est strictement identique à :

```
int t[4] = {0, 10, 20, 30};
```

## IV-A-3 - Création d'un type « tableau »

Tout d'abord, étudions un peu la logique du mot-clé typedef. Pour cela, supposons que l'on veuille pouvoir utiliser indifféremment les formules suivantes pour déclarer un entier :

```
int x;
```

et

```
ENTIER x;
```

Dans la première forme, on remplace x par ENTIER et on la fait précéder du mot-clé typedef, ce qui nous donne :

```
typedef int ENTIER;
```

Concrètement, cela signifie : un ENTIER est tout ce qui suit le mot int dans une déclaration. On peut également formuler une phrase similaire (genre : un TABLEAU est tout ce qui est précédé du mot **int** et suivi de **[10]** dans sa déclaration) pour définir un type TABLEAU mais il y a plus malin :
On remplace t par TABLEAU dans sa déclaration puis on ajoute devant le mot-clé typedef, exactement comme ce qu'on a fait avec x et ENTIER. Ainsi, le type tableau de 10 int se définit de la manière suivante :

```
typedef int TABLEAU[10];
```

Désormais :

```
int t[10];
```

et :

```
TABLEAU t;
```

sont strictement équivalents.

## IV-A-4 - Les tableaux à plusieurs dimensions

On peut également créer un tableau à plusieurs dimensions. Par exemple :

```
int t[10][3];
```

Un tableau à plusieurs dimensions n'est en fait rien d'autre qu'un tableau (tableau à une dimension) dont les éléments sont des tableaux. Comme dans le cas des tableaux à une dimension, le type des éléments du tableau doit être parfaitement connu. Ainsi dans notre exemple, t est un tableau de 10 tableaux de 3 int, ou pour vous aider à y voir plus clair :

```
typedef int TRIPLET[3];
TRIPLET t[10];
```

Les éléments de t vont de t[0] à t[9], chacun étant un tableau de 3 int.
On peut bien entendu créer des tableaux à 3 dimensions, 4, 5, 6, ...
On peut également initialiser un tableau à plusieurs dimensions. Par exemple :

```
int t[3][4] = { {0, 1, 2, 3},
                {4, 5, 6, 7},
                {8, 9, 10, 11} };
```

Qu'on aurait également pu tout simplement écrire :

```
int t[][4] = { {0, 1, 2, 3},
               {4, 5, 6, 7},
               {8, 9, 10, 11} };
```

### IV-A-5 - Calculer le nombre d'éléments d'un tableau

La taille d'un tableau est évidemment le nombre d'éléments du tableau multiplié par la taille de chaque élément. Ainsi, le nombre d'éléments dans un tableau est égal à sa taille divisée par la taille d'un élément. On utilise alors généralement la formule sizeof(t) / sizeof(t[0]) pour connaître le nombre d'éléments d'un tableau t. La macro définie ci-dessous permet de calculer la taille d'un tableau :

```
#define COUNT(t) (sizeof(t) / sizeof(t[0]))
```

## IV-B - Les pointeurs

### IV-B-1 - Les tableaux et les pointeurs

Pour nous fixer les idées, considérons le tableau t suivant :

```
char t[10];
```

Mais les règles que nous allons établir ici s'appliquent à n'importe quel type de tableau, y compris les tableaux à plusieurs dimensions.
Définissons ensuite le type TABLEAU par :

```
typedef char TABLEAU[10];
```

Mais avant d'aller plus loin, j'aimerais déjà préciser que les tableaux et les pointeurs n'ont rien de commun, à part peut-être le fait qu'on peut pointer sur n'importe quel élément d'un tableau (ou sur un tableau ...), tout comme on peut pointer sur n'importe quoi. Mais il y a quand même une chose qui lie les deux notions, c'est que : lorsqu'elle n'est pas utilisée en unique argument de **sizeof** ou en unique argument de **l'opérateur &** ("adresse de"), une **expression de type tableau** est toujours convertie par le compilateur en l'adresse de son premier élément. Cela signifie, dans ces conditions, que si t est un tableau de disons 10 éléments, l'écriture **t** est strictement équivalente à **&t[0]**, donc t + 1 (qui est strictement équivalent à &(t[0]) + 1) est strictement équivalent à &t[1] et ainsi de suite. Ainsi :

```
t[5] = '*';
```

est strictement équivalent à :

```
*(t + 5) = '*';
```

Et ainsi de suite.
Dans la pratique, on utilise un pointeur sur un élément du tableau, généralement le premier. Cela permet d'accéder à n'importe quel élément du tableau par simple calcul d'adresse. En effet, comme nous l'avons dit plus haut : t + 1 est équivalent à &(t[1]), t + 2 à &(t[2]), etc.
Voici un exemple qui montre une manière de parcourir un tableau :

```
#include <stdio.h>

#define COUNT(t) (sizeof(t) / sizeof(t[0]))

void Affiche(int * p, size_t nbElements);

int main()
```

```
{
    int t[10] = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90};

    Affiche(t, COUNT(t));

    return 0;
}

void Affiche(int * p, size_t nbElements)
{
    size_t i;

    for(i = 0; i < nbElements; i++)
        printf("%d\n", p[i]);
}
```

## IV-B-2 - L'arithmétique des pointeurs

L'arithmétique des pointeurs est née des faits que nous avons établis précédemment. En effet si p pointe sur un élément d'un tableau, p + 1 doit pointer sur l'élément suivant. Donc si la taille de chaque élément du tableau est par exemple de 4, p + 1 déplace le pointeur de 4 octets (où se trouve l'élément suivant) et non de un.
De même, puisque l'on devrait avoir (p + 1) - p = 1 et non 4, la différence entre deux adresses donne le nombre d'éléments entre ces adresses et non le nombre d'octets entre ces adresses. Le type d'une telle expression est **ptrdiff_t**, qui est défini dans le fichier **stddef.h**.
Et enfin, l'écriture p[i] est strictement équivalente à *(p + i).
Cela montre à quel point le typage des pointeurs est important. Cependant, il existe des pointeurs dits **génériques** capables de pointer sur n'importe quoi. Ainsi, la conversion d'un pointeur générique en un pointeur d'un autre type par exemple ne requiert aucun cast et vice versa.

## IV-B-3 - Pointeurs constants et pointeurs sur constante

L'utilisation du mot-clé const avec les pointeurs est au début un peu délicate. En effet, en C on a ce qu'on appelle des pointeurs constants et des pointeurs sur constante (et donc aussi des pointeurs contants sur constante ...). Il faut bien savoir les différencier.

- **Pointeur constant** : Le pointeur, qui est une variable comme toute les autres, est déclaré const.
- **Pointeur sur constante** : C'est l'objet pointé qui est constant.

Dans le deuxième cas, l'objet pointé n'est pas nécessairement une constante. Seulement, on ne pourra pas le modifier via le pointeur (qui considère que l'objet est une constante). De même, on peut pointer sur une mémoire en lecture seule avec n'importe quel pointeur (et non nécessairement un pointeur sur constante) mais cela ne signifie pas que la mémoire devient désormais accessible en écriture (puisque la mémoire pointée est en lecture seule. Ce n'est pas en pointant là-dessus avec quoi que ce soit qu'on pourra changer cela.). Si le pointeur n'est pas un pointeur sur contante alors qu'il pointe sur une constante, le compilateur acceptera évidemment de modifier le contenu de la mémoire pointée (puisque l'écriture est syntaxiquement correcte), mais le problème se manifestera à l'exécution, lorsque le programme tentera de modifier le contenu de la zone en lecture seule.
Comme int * est le type d'un pointeur sur int, un pointeur constant sur un int est de type

```
int * const
```

Si on avait placé le mot-clé const avant int *, on obtiendrait :

```
const int *
```

qui correspond plutôt, contrairement à ce qu'on attendait, au type pointeur sur const int (pointeur sur constante) ! Voici deux illustrations de l'utilisation de const avec les pointeurs :

```
int n, m;
```

```
int * const p = &n;
*p = 10;
/* p = &m : INTERDIT ! p est une constante ! */
```

```
int n, m;
int const * p = &n; /* Ou const int * p = &n; */
p = &m;
/* *p = 10 : INTERDIT ! *p est une constante ! */
```

## IV-B-4 - Pointeurs génériques

Le type des pointeurs génériques est **void ***. Comme ces pointeurs sont génériques, la taille des données pointées est inconnue et l'arithmétique des pointeurs ne s'applique donc pas à eux. De même, puisque la taille des données pointées est inconnue, l'opérateur d'indirection * ne peut être utilisé avec ces pointeurs, un cast est alors obligatoire. Par exemple :

```
int n;
void * p;

p = &n;
*((int *)p) = 10; /
* p etant desormais vu comme un int *, on peut alors lui appliquer l'operateur *. */
```

Etant donné que la taille de toute donnée est multiple de celle d'un char, le type char * peut être également utilisé en tant que pointeur universel. En effet, une variable de type char * est un **pointeur sur octet** autrement dit peut pointer n'importe quoi. Cela s'avère pratique des fois (lorsqu'on veut lire le contenu d'une mémoire octet par octet par exemple) mais dans la plupart des cas, il vaut mieux toujours utiliser les pointeurs génériques. Par exemple, la conversion d'une adresse de type différent en char * et vice versa nécessite toujours un cast, ce qui n'est pas le cas avec les pointeurs génériques.
Dans **printf**, le spécificateur de format **%p** permet d'imprimer une adresse (void *) dans le format utilisé par le système.
Et pour terminer, il existe une macro à savoir **NULL**, définie dans stddef.h, permettant d'indiquer q'un pointeur ne pointe nulle part. Son intérêt est donc de permettre de tester la validité d'un pointeur et il est conseillé de toujours initialiser un pointeur à NULL.

## IV-B-5 - Exemple avec un tableau à plusieurs dimensions

Soit :

```
int t[10][3];
```

Définissons le type TRIPLET par :

```
typedef int TRIPLET[3];
```

De façon à avoir :

```
TRIPLET t[10];
```

En dehors du cas sizeof et opérateur & ("adresse de"), t représente l'adresse de t[0] (qui est un TRIPLET) donc l'adresse d'un TRIPLET. En faisant t + 1, on se déplace donc d'un TRIPLET soit de 3 int.
D'autre part, t peut être vu comme un tableau de 30 int (3 * 10 = 30) puisque les éléments d'un tableau sont toujours contigus en mémoire. On peut donc accéder à n'importe quel élément de t à l'aide d'un pointeur sur int.
Soit p un pointeur sur int et faisons :

```
p = (int *)t;
```

On a alors, numériquement, les équivalences suivantes :

| | |
|---|---|
| t | p |
| t + 1 | p + 3 |
| t + 2 | p + 6 |
| ... | |
| t + 9 | p + 27 |

Prenons alors à présent, le 3ème TRIPLET de t soit t[2].
Puisque le premier élément de t[2] se trouve à l'adresse t + 2 soit p + 6, deuxième se trouve en p + 6 +1 et le troisième et dernier en p + 6 + 2. A près cet entier, on se retrouve au premier élément de t[3], en p + 9.
En conclusion, pour un tableau déclaré :

```
<type> t[N][M];
```

on a la formule :

```
t[i][j] = *(p + N*i + j) /* ou encore p[N*i + j] */
```

Où évidemment : p = (int *)t.
Et on peut bien sur étendre cette formule pour n'importe quelle dimension.

### IV-B-6 - Passage d'un tableau en argument d'une fonction

Nous avons déjà vu que le passage d'un tableau en argument d'une fonction se fait tout simplement en passant l'adresse de son premier élément. Sachez également que, **en argument d'une fonction**,

```
<type> <identificateur>[]
```

est **strictement équivalent à** :

```
<type> * <identificateur>
```

Alors que dans une « simple » déclaration, la première déclare un tableau (qui doit être impérativement initialisé) et la deuxième un pointeur. On peut également préciser le nombre d'éléments du tableau mais cette information sera complètement ignorée par le compilateur. Elle n'a donc d'intérêt que pour la documentation.

## IV-C - Les chaînes de caractères

### IV-C-1 - Chaîne de caractères

Par définition, une **chaîne de caractères**, ou tout simplement : chaîne, est une suite finie de caractères. Par exemple, "Bonjour", "3000", "Salut !", "EN 4", ... sont des chaînes de caractères. En langage C, une chaîne de caractères littérale s'écrit entre double quottes, exactement comme dans les exemples donnés ci-dessus.

### IV-C-2 - Longueur d'une chaîne

La longueur d'une chaîne est le nombre de caractères qu'elle comporte. Par exemple, la chaîne "Bonjour" comporte 7 caractères ('B', 'o', 'n', 'j', 'o', 'u' et 'r'). Sa longueur est donc 7. En langage C, la fonction **strlen**, déclarée dans le fichier **string.h**, permet d'obtenir la longueur d'une chaîne passée en argument. Ainsi, strlen("Bonjour") vaut 7.

## IV-C-3 - Représentation des chaînes de caractères en langage C

Comme nous l'avons déjà mentionné plus haut, les chaîne de caractères littérales s'écrivent en langage C entre double quottes. En fait, le langage C ne dispose pas vraiment de type chaîne de caractères. Une chaîne est tout simplement représentée à l'aide d'un **tableau de caractères**.
Cependant, les fonctions manipulant des chaînes doivent être capables de détecter la fin d'une chaîne donnée. Autrement dit, toute chaîne de caractères doit se terminer par un caractère indiquant la fin de la chaîne. Ce caractère est le caractère '\0' et est appelé le **caractère nul** ou encore **caractère de fin de chaîne**. Son code ASCII est 0. Ainsi la chaîne "Bonjour" est en fait un tableau de caractères dont les éléments sont 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0', autrement dit un tableau de 8 caractères et on a donc "Bonjour"[0] = 'B', "Bonjour"[1] = 'o', "Bonjour"[2] = 'n', ... "Bonjour"[7] = '\0'. Toutefois, la mémoire utilisée pour stocker la chaîne peut être accessible qu'en lecture, le il n'est donc pas portable de tenter de la modifier.
Les fonctions de manipulation de chaîne de la bibliothèque standard du langage C sont principalement déclarées dans le fichier **string.h**. Voici un exemple d'utilisation d'une de ces fonctions.

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

int main()
{
    char t[50];

    strcpy(t, "Hello, world!");
    printf("%s\n", t);

    return 0;
}
```

Dans cet exemple, la chaîne t ne peut contenir tout au plus que 50 caractères, caractère de fin de chaîne inclus. Autrement dit t ne peut que contenir 49 caractères « normaux » car il faut toujours réserver une place pour le caractère de fin de chaîne : '\0'. On peut aussi bien sûr initialiser une chaîne au moment de sa déclaration, par exemple :

```c
char s[50] = "Bonjour";
```

Qui est strictement équivalente à :

```c
char s[50] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0'};
```

Puisque, vu d'un pointeur, la valeur d'une expression littérale de type chaîne n'est autre que l'adresse de son premier élément, on peut utiliser un simple pointeur pour manipuler une chaîne. Par exemple :

```c
char * p = "Bonjour";
```

Dans ce cas, p pointe sur le premier élément de la chaîne "Bonjour". Or, comme nous l'avons déjà dit plus haut, la mémoire allouée pour la chaîne "Bonjour" est en lecture seule donc on ne peut pas écrire par exemple :

```c
p[2] = '*'; /* Interdit */
```

Avec un tableau, ce n'est pas l'adresse en mémoire de la chaîne qui est stockée, mais les caractères de la chaîne, copiés caractère par caractère. La mémoire utilisée par le tableau étant indépendante de celle utilisée par la chaîne source, on peut faire ce qu'on veut de notre tableau. La fonction strcpy permet de copier une chaîne vers un autre emplacement mémoire.
Le paragraphe suivant discute des fonctions de manipulation de chaînes en langage C.

# IV-C-4 - Les fonctions de manipulation de chaîne

## strcpy, strncpy

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

int main()
{
    char t1[50], t2[50];

    strcpy(t1, "Hello, world!");
    strcpy(t2, "**************");
    strncpy(t1, t2, 3);
    printf("%s\n", t1);

    return 0;
}
```

Attention ! Si t1 n'est pas assez grand pour pouvoir contenir la chaîne à copier, vous aurez un **débordement de tampon** (**buffer overflow**). Un **tampon** (ou **buffer**) est tout simplement une zone de la mémoire utilisée par un programme pour stocker temporairement des données. Par exemple, t1 est un buffer de 50 octets. Il est donc de la responsabilité du programmeur de ne pas lui passer n'importe quoi ! En effet en C, le compilateur suppose que le programmeur sait ce qu'il fait !
La fonction strncpy s'utilise de la même manière que strcpy. Le troisième argument indique le nombre de caractères à copier. Aucun caractère de fin de chaîne n'est automatiquement ajouté.

## strcat, strncat

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

int main()
{
    char t[50];

    strcpy(t, "Hello, world");
    strcat(t, " from");
    strcat(t, " strcpy");
    strcat(t, " and strcat");
    printf("%s\n", t);

    return 0;
}
```

## strlen

Retourne le nombre de caractères d'une chaîne.

## strcmp, strncmp

On n'utilise pas l'opérateur == pour comparer des chaînes car ce n'est pas les adresses qu'on veut comparer mais le contenu mémoire. La fonction strcmp compare deux chaînes de caractères et retourne :

- zéro si les chaînes sont identiques
- un nombre négatif si la première est "inférieure" (du point de vue lexicographique) à la seconde
- et un nombre positif si la première est "supérieure" (du même point de vue ...) à la seconde

Ainsi, à titre d'exemple, dans l'expression

```
strcmp("clandestin", "clavier")
```

La fonction retourne un nombre négatif car, 'n' étant plus petit que 'v' (dans le jeu de caractères ASCII, ça n'a rien à voir avec le langage C), "clandestin" est plus petit que "clavier".

## IV-C-5 - Fusion de chaînes littérales

Le C permet de fusionner deux chaînes littérales en les plaçant simplement côte à côte. Cela se révèle particulièrement utile lorsqu'on doit composer une chaîne et que cette dernière soit trop longue a écrire sur une seule ligne. Par exemple :

```
#include <stdio.h>

int main()
{
    printf( "Voici une chaine de caracteres particulierement longue, tres longue, "
            "tellement longue (oui, longue !) qu'il a fallu la decouper en deux !" );
    return 0;
}
```

# IV-D - Exercices

## IV-D-1 - Recherche dans un tableau

Nous nous proposons d'écrire un programme qui permet de trouver toutes les occurrences d'un nombre entré par l'utilisateur parmi un un ensemble de nombres précédemment entrés. L'utilisateur entrera dix nombres (que vous placerez dans un tableau), ensuite il entrera le nombre à rechercher et le programme devra afficher toutes les occurrences de ce nombre s'il y en a ainsi que le nombre d'occurrences trouvées. Voici un exemple d'exécution :

```
Ce programme permet de trouver toutes les occurrences d'un nombre.
Entrez 10 nombres :
t[0] : 2
t[1] : -1
t[2] : 3
t[3] : 28
t[4] : 3
t[5] : -8
t[6] : 9
t[7] : 40
t[8] : -1
t[9] : 8
Entrez le nombre a rechercher : 3
t[2]
t[4]
2 occurrence(s) trouvee(s).
Merci d'avoir utilise ce programme. A bientot !
```

## IV-D-2 - Calcul de la moyenne

Nous nous proposons d'écrire un programme permettant de calculer la moyenne d'un élève. L'utilisateur entrera 5 "notes". Une note est constituée d'une note sur 20 (de type entier) et d'un coefficient compris entre 1 et 5 (entier également). La note définitive pour une matière est la note sur 20 multipliée par le coefficient. La moyenne de l'étudiant est égale au total des notes définitives divisé par la somme des coefficients.
On créera un tableau t de 5 tableaux de 3 entiers, c'est-à-dire int t[5][3]. Les 5 éléments de t sont destinés à accueillir les 5 notes. Pour chaque note t[i], t[i][0] contiendra la note sur 20, t[i][1] le coefficient et t[i][2] la note définitive. Le programme demandera à l'utilisateur d'entrer les notes puis affichera la moyenne de l'élève avec 2 chiffres après la virgule et les calculs intermédiaires. Voici un exemple d'exécution :

```
Ce programme permet de calculer votre moyenne scolaire.
```

```
Note 1 (0 a 20) : 14
Coef 1 (1 a 5)  : 5
Note 2 (0 a 20) : 10
Coef 2 (1 a 5)  : 5
Note 3 (0 a 20) : 16
Coef 3 (1 a 5)  : 3
Note 4 (0 a 20) : 8
Coef 4 (1 a 5)  : 1
Note 5 (0 a 20) : 12
Coef 5 (1 a 5)  : 1
+------+------+----------+
| Note | Coef | Note Def |
+------+------+----------+
|   14 |    5 |       70 |
+------+------+----------+
|   10 |    5 |       50 |
+------+------+----------+
|   16 |    3 |       48 |
+------+------+----------+
|    8 |    1 |        8 |
+------+------+----------+
|   12 |    1 |       12 |
+------+------+----------+
| Tot. |   15 |      188 |
+------+------+----------+
| Moy. |           12.53 |
+------+-----------------+
Merci d'avoir utilise ce programme. A bientot !
```

### IV-D-3 - Manipulation de chaînes

Ecrire les fonctions de manipulation de chaînes suivantes :

- str_cpy, de rôle et d'utilisation semblables à la fonction strcpy de la bibliothèque standard
- str_len, de rôle et d'utilisation semblalbles à la fonction strlen de la bibliothèque standard
- str_equals, qui reçoit en arguments de chaînes de caractères et qui retourne VRAI si elles sont égales et FAUX dans le cas contraire

## V - Les entrées/sorties en langage C

### V-A - Introduction

Les **entrées/sorties** (**E/S**) ne font pas vraiment partie du langage C car ces opérations sont dépendantes du système. Cela signifie que pour réaliser des opérations d'entrée/sortie en C, il faut en principe passer par les fonctionnalités offertes par le système. Néanmoins sa bibliothèque standard est fournie avec des fonctions permettant d'effectuer de telles opérations afin de faciliter l'écriture de code portable. Les fonctions et types de données liées aux entrées/sorties sont principalement déclarés dans le fichier **stdio.h** (**st**an**d**ard **i**nput/**o**utput).

### V-B - Les fichiers

Les entrées/sorties en langage C se font par l'intermédiaire d'entités logiques, appelés **flux**, qui représentent des objets externes au programme, appelés **fichiers**. Un fichier peut être ouvert en lecture, auquel cas il est censé nous fournir des données (c'est-à-dire être lu) ou ouvert en écriture, auquel cas il est destiné à recevoir des données provenant du programme. Un fichier peut être à la fois ouvert en lecture et en écriture. Une fois qu'un fichier est ouvert, un flux lui est associé. Un **flux d'entrée** est un flux associé à un fichier ouvert en lecture et un **flux de sortie** un flux associé à un fichier ouvert un écriture. Tous les fichiers ouverts doivent être fermés avant la fin du programme. Lorsque les données échangées entre le programme et le fichier sont de type texte, la nécessité de définir ce qu'on appelle une ligne est primordiale. En langage C, une **ligne** est une suite de caractères terminée par le **caractère de fin de ligne** (inclus) : '\n'. Par exemple, lorsqu'on effectue des saisies au clavier, une ligne correspond à une suite de

caractères terminée par **ENTREE**. Puisque la touche ENTREE termine une ligne, le caractère généré par l'appui de cette touche est donc, en C standard, le caractère de fin de ligne soit '\n'.

## V-C - Les entrée et sortie standards

Lorsque le système exécute un programme, trois fichiers sont automatiquement ouverts :

- l'**entrée standard** par défaut le clavier
- la **sortie standard**, par défaut l'écran (ou la console)
- et l'**erreur standard**, par défaut associé à l'écran (ou la console)

Respectivement associés aux flux **stdin**, **stdout** et **stderr**. Ils sont automatiquement fermés avant la fin du programme.
Chez la majorité des systèmes, dont Windows et UNIX, l'utilisateur peut rediriger les entrée et sortie standards vers un autre fichier à l'aide des symboles < et >. Par exemple, exécutez le programme qui affiche « Hello, world » à partir de l'interpréteur de commandes à l'aide de la commande suivante (nous supposons que nous sommes sous Windows et que le programme s'appelle hello.exe) :

```
hello > sortie.txt
```

et vous verrez que le message sera imprimé dans le fichier sortie.txt et non à l'écran. Si le fichier n'existe pas, il sera créé. S'il existe déjà, son ancien contenu sera effacé.
Un deuxième et dernier exemple : écrire un programme qui affiche **Hello, world** en écrivant explicitement sur stdout, autrement dit avec un printf beaucoup plus explicite. On utilisera alors la fonction **fprintf** qui permet d'écrire du texte sur un flux de sortie, dans notre cas : stdout, ce qui nous donne :

```c
#include <stdio.h>

int main()
{
    fprintf(stdout, "Hello, world\n");
    return 0;
}
```

## V-D - Exemple : lire un caractère, puis l'afficher

La macro **getc** permet de lire un caractère sur un flux d'entrée. La macro **putc** permet d'écrire un caractère sur un flux de sortie. Voici un programme simple qui montre comment utiliser les macros getc et putc :

```c
#include <stdio.h>

int main()
{
    int c; /* le caractere */

    printf("Veuillez taper un caractere : ");
    c = getc(stdin);

    printf("Vous avez tape : ");
    putc(c, stdout);

    return 0;
}
```

Vous vous demandez certainement la raison pour laquelle on a utilisé int plutôt que char dans la déclaration de c. Et bien tout simplement parce que getc retourne un int (de même putc attend un argument de type int). Mais justement : Pourquoi ? Et bien parce que getc doit pouvoir non seulement retourner le caractère lu (un char) mais aussi une valeur qui ne doit pas être un char pour signaler qu'aucun caractère n'a pu être lu. Cette valeur est **EOF**. Elle est définie dans le fichier stdio.h. Dans ces conditions, il est clair qu'on peut utiliser tout sauf un char comme type de retour de getc.

Un des cas les plus fréquents où getc retourne EOF est lorsqu'on a rencontré la **fin du fichier**. La fin d'un fichier est un point situé au-delà du dernier caractère de ce fichier (si le fichier est vide, le début et la fin du fichier sont donc confondus). On dit qu'on a rencontré la fin d'un fichier après avoir encore tenté de lire dans ce fichier alors qu'on se trouve déjà à la fin, pas juste après avoir lu le dernier caractère. Lorsque stdin est associé au clavier, la notion de fin de fichier perd à priori son sens car l'utilisateur peut très bien taper n'importe quoi à n'importe quel moment. Cependant l'environnement d'exécution (le système d'exploitation) offre généralement un moyen de spécifier qu'on n'a plus aucun caractère à fournir (concrètement, pour nous, cela signifie que getc va retourner EOF). Sous Windows par exemple, il suffit de taper en début de ligne la combinaison de touches Ctrl + Z (héritée du DOS) puis de valider par ENTREE. Evidemment, tout recommence à zéro à la prochaine opération de lecture.
Les macros **getchar** et **putchar** s'utilisent comme getc et putc sauf qu'elles n'opèrent que sur stdin, respectivement stdout. Elles sont définies dans stdio.h comme suit :

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

Et enfin **fgetc** est une fonction qui fait la même chose que getc (qui peut être en fait une fonction ou une macro ...). De même **fputc** est une fonction qui fait la même chose que putc.

## V-E - Saisir une chaîne de caractères

Il suffit de lire les caractères présents sur le flux d'entrée (dans notre cas : stdin) jusqu'à ce que l'on ait atteint la fin du fichier ou le caractère de fin de ligne. Nous devrons fournir en arguments de la fonction l'adresse du tampon destiné à contenir la chaîne de caractère saisie et la taille de ce tampon pour supprimer le risque de débordement de tampon.

```
#include <stdio.h>

char * saisir_chaine(char * lpBuffer, size_t nBufSize);

int main()
{
    char lpBuffer[20];

    printf("Entrez une chaine de caracteres : ");
    saisir_chaine(lpBuffer, sizeof(lpBuffer));

    printf("Vous avez tape : %s\n", lpBuffer);

    return 0;
}

char * saisir_chaine(char * lpBuffer, size_t nBufSize)
{
    size_t nbCar = 0;
    int c;

    c = getchar();
    while (nbCar < nBufSize - 1 && c != EOF && c != '\n')
    {
        lpBuffer[nbCar] = (char)c;
        nbCar++;
        c = getchar();
    }

    lpBuffer[nbCar] = '\0';

    return lpBuffer;
}
```

La fonction **scanf** permet également de saisir une chaîne de caractères ne comportant aucun espace (espace, tabulation, etc.) grâce au spécificateur de format %s. Elle va donc arrêter la lecture à la rencontre d'un espace (mais avant d'effectuer la lecture, elle va d'abord avancer jusqu'au premier caractère qui n'est pas un espace). scanf ajoute enfin le caractère de fin de chaîne. Le gabarit permet d'indiquer le nombre maximum de caractères à lire (caractère de fin de chaîne non compris). Lorsqu'on utilise scanf avec le spécificateur %s (qui demande de lire une chaîne sans

espace), il ne faut jamais oublier de spécifier également le nombre maximum de caractères à lire (à mettre juste devant le s) sinon le programme sera ouvert aux attaques par débordement de tampon. Voici un exemple qui montre l'utilisation de scanf avec le spécicateur de format %s :

```
#include <stdio.h>

int main()
{
    char lpBuffer[20];

    printf("Entrez une chaine de caracteres : ");
    scanf("%19s", lpBuffer);

    printf("Vous avez tape : %s\n", lpBuffer);

    return 0;
}
```

Et enfin, il existe également une fonction, **gets**, déclarée dans stdio.h, qui permet de lire une chaîne de caractères sur stdin. Cependant **cette fonction est à proscrire** car elle ne permet pas de spécifier la taille du tampon qui va recevoir la chaîne lue.

## V-F - Lire une ligne avec fgets

La fonction **fgets** permet de lire une ligne (c'est-à-dire y compris le '\n') sur un flux d'entrée et de placer les caractères lus dans un buffer. Cette fonction ajoute ensuite le caractère '\0'. Exemple :

```
#include <stdio.h>

int main()
{
    char lpBuffer[20];

    printf("Entrez une chaine de caracteres : ");
    fgets(lpBuffer, sizeof(lpBuffer), stdin);

    printf("Vous avez tape : %s", lpBuffer);

    return 0;
}
```

Dans cet exemple, deux cas peuvent se présenter :

- l'utilisateur entre une chaîne comportant 18 caractères tout au plus puis valide le tout par ENTREE, alors tous les caractères de la ligne, y compris le caractère de fin de ligne, sont copiés dans lpBuffer puis le caractère de fin de chaîne est ajouté
- l'utilisateur entre une chaîne comportant plus de 18 caractères (c'est-à-dire >= 19) puis valide le tout par ENTREE, alors seuls les 19 premiers caractères sont copiés vers lpBuffer puis le caractère de fin de chaîne est ajouté

## V-G - Mécanisme des entrées/sorties en langage C

### V-G-1 - Le tamponnage

#### V-G-1-a - Les tampons d'entrée/sortie

En langage C, les entrées/sorties sont par défaut **bufférisées**, c'est-à-dire que les données à lire (respectivement à écrire) ne sont pas directement lues (respectivement écrites) mais sont tout d'abord placées dans un tampon (buffer) associé au fichier. La preuve, vous avez certainement remarqué par exemple que lorsque vous entrez des données

pour la première fois à l'aide du clavier, ces données ne seront lues qu'une fois que vous aurez appuyé sur la touche ENTREE. Ensuite, toutes les opérations de lecture qui suivent se feront immédiatement tant que le caractère '\n' est encore présent dans le tampon de lecture, c'est-à-dire tant qu'il n'a pas été encore lu. Lorsque le caractère '\n' n'est plus présent dans le buffer, vous devrez à nouveau appuyer sur ENTREE pour valider la saisie, et ainsi de suite.
Les opérations d'écriture sont moins compliquées, mais il y a quand même quelque chose dont il serait totalement injuste de ne pas en parler. Comme nous l'avons déjà dit plus haut, les entrées/sorties sont par défaut bufferisées c'est-à-dire passent par un tampon. Dans le cas d'une opération d'écriture, il peut arriver que l'on souhaite à un certain moment forcer l'écriture physique des données présentes dans le tampon sans attendre que le système se décide enfin de le faire. Dans ce cas, on utilisera la fonction **fflush**. Nous verrons dans le paragraphe suivant un exemple d'utilisation de cette fonction.

## V-G-1-b - Les modes de tamponnage

Le langage C permet de spécifier le mode de tamponnage à utiliser avec un flux donné à l'aide de la fonction **setvbuf**. Elle doit être appelée avant toute utilisation du fichier.
Il existe 3 modes de tamponnage des entrées/sorties :

- **Pas de tamponnage** (**_IONBF** (*no buffering*)), dans lequel le flux n'est associé à aucun tampon. Les données sont directement écrites sur ou lues depuis le fichier.
- **Tamponnage par lignes** (**_IOLBF** (*line buffering*)), le mode par défaut (d'après la norme), dans lequel le flux est associé à un tampon vidé que lorsqu'il est plein, lorsque le caractère de fin de ligne a été envoyé, lorsqu'une opération de lecture sur un flux en mode "Pas de tampon" a été efectuée ou lorsqu'une opération de lecture sur un flux en mode "Tamponnage par lignes" nécessite son vidage. Par exemple, en pratique, lorsque stdout est dans ce mode, toute demande de lecture sur stdin provoquera l'écriture physique des caractères encore dans le tampon. Cela permet d'avoir les questions afichées à l'écran avant que l'utilisateur puisse entrer la réponse.
- **Tamponnage strict** (**_IOFBF** (*full buffering*)), dans lequel le flux associé à un tampon vidé que lorsqu'il est plein.

Dans tous les cas, la fermeture d'un fichier ouvert en écriture entraîne également l'écriture des caractères qui sont encore dans le tampon, s'il y en a (ce qui a bien évidemment lieu avant sa fermeture).
Le prototype de la fonction setvbuf est le suivant :

```
int setvbuf(FILE * f, char * buf, int mode, size_t size);
```

L'argument mode indique évidemment le mode à utiliser (_IONBF, _IOLBF ou _IOFBF) et l'argument size la taille du tampon (buf) à associer au flux. Si buf vaut NULL, un tampon de taille size sera alloué et associé au fichier. Les arguments buf et size sont évidemment ignorés lorsque mode vaut _IONBF.
Voici un exemple d'utilisation de cette fonction :

```
#include <stdio.h>

#define N 1

void loop(unsigned long n);

int main()
{
    setvbuf(stdout, NULL, _IOFBF, 8);

    printf("a\n"); /* Contenu du buffer : [a\n]. */
    loop(N); /* Faire une quelconque longue boucle */

    /* Le buffer n'est pas encore plein, rien ne s'affichera donc sur la sortie standard. */

    printf("b\n"); /* Contenu du buffer : [a\nb\n]. */
    fflush(stdout); /* Vider le buffer. */

    /* Le buffer n'est pas encore plein mais fflush a ete appelee. On aura donc :   */
```

```
    /* - Sur la sortie standard : + [a\nb\n].                                    */
    /* - Dans le buffer : [] (rien).                                            */

    loop(N); /* Faire une quelconque longue boucle */

    printf("azertyuiop\n");

    /* On aura :                                          */
    /* - Dans le buffer : [azertyui].                     */
    /* Le buffer est plein, le vidage s'impose. On aura : */
    /* - Sur la sortie standard : + [azertui].            */
    /* - Dans le buffer : [] (rien).                      */
    /* Il reste encore les carateres [op\n]. On aura :    */
    /* - Dans le buffer : [op\n].                         */

    loop(N); /* Faire une quelconque longue boucle */

    return 0;

    /* Au dela de l'accolade : Fin du programme.                                        */
    /* Tous les fichiers encore ouverts (dont la sortie standard) seront fermes. On aura : */
    /* - Sur la sortie standard : + [op\n].                                             */
    /* - Dans le buffer : [] (rien).                                                    */
}

void loop(unsigned long n)
{
    unsigned long i, j, end = 100000000;

    for(i = 0; i < n; i++)
        for(j = 0; j < end; j++);
}
```

## V-G-2 - Lire de manière sûre des données sur l'entrée standard

Tout d'abord, analysons le tout petit programme suivant :

```
#include <stdio.h>

int main()
{
    char nom[12], prenom[12];

    printf("Entrez votre nom : ");
    fgets(nom, sizeof(nom), stdin);

    printf("Entrez votre prenom : ");
    fgets(prenom, sizeof(prenom), stdin);

    printf("Votre nom est : %s", nom);
    printf("Et votre prenom : %s", prenom);

    return 0;
}
```

Dans ce programme, si l'utilisateur entre un nom comportant moins de 10 caractères puis valide par ENTREE, alors tous les caractères rentrent dans nom et le programme se déroule bien comme prévu. Par contre si l'utilisateur entre un nom comportant plus de 10 caractères, seuls les 11 premiers caractères seront copiés dans nom et des caractères sont donc encore présents dans le buffer du clavier. Donc, à la lecture du prénom, les caractères encore présents dans le buffer seront immédiatement lus sans que l'utilisateur n'ait pu entrer quoi que ce soit. Voici un deuxième exemple :

```
#include <stdio.h>

int main()
{
    int n;
    char c;
```

```
    printf("Entrez un nombre (entier) : ");
    scanf("%d", &n);

    printf("Entrez un caractere : ");
    scanf("%c", &c);

    printf("Le nombre que vous ave entre est : %d\n", n);
    printf("Le caractere que vous ave entre est : %c\n", c);

    return 0;
}
```

Lorsqu'on demande à scanf de lire un nombre, elle va déplacer le pointeur jusqu'au premier caractère non blanc, lire tant qu'elle doit lire les caractères pouvant figurer dans l'expression d'un nombre, puis s'arrêter à la rencontre d'un caractère invalide (espace ou lettre par exemple).
Donc dans l'exemple ci-dessus, la lecture du caractère se fera sans l'intervention de l'utilisateur à cause de la présence du caractère '\n' (qui sera alors le caractère lu) due à la touche ENTREE frappée pendant la saisie du nombre.
Ces exemples nous montrent bien que d'une manière générale, il faut toujours **vider le buffer** du clavier après chaque saisie, sauf si celui-ci est déjà vide bien sûr. Pour vider le buffer du clavier, il suffit de manger tous les caractères présents dans le buffer jusqu'à ce qu'on ait rencontré le caractère de fin de ligne ou atteint la fin du fichier.
A titre d'exemple, voici une version améliorée (avec vidage du tampon d'entrée après lecture) de notre fonction **saisir_chaine** :

```
char * saisir_chaine(char * lpBuffer, int nBufSize)
{
    char * ret = fgets(lpBuffer, nBufSize, stdin);

    if (ret != NULL)
    {
        char * p = lpBuffer + strlen(lpBuffer) - 1;
        if (*p == '\n')
            *p = '\0'; /* on ecrase le '\n' */
        else
        {
            /* On vide le tampon de lecture du flux stdin */
            int c;

            do
    c = getchar();
            while (c != EOF && c != '\n');
        }
    }

    return ret;
}
```

# VI - L'allocation dynamique de mémoire

## VI-A - Les fonctions malloc et free

L'intérêt d'**allouer dynamiquement de la mémoire** se ressent lorsqu'on veut créer un tableau dont la taille dont nous avons besoin n'est connue qu'à l'exécution par exemple. On utilise généralement les fonctions **malloc** et **free**.

```
int t[10];
...
/* FIN */
```

Peut être remplacé par :

```
int * p;
```

```
p = malloc(10 * sizeof(int));
...
free(p); /* libérer la mémoire lorsqu'on n'en a plus besoin */
/* FIN */
```

Les fonctions malloc et free sont déclarées dans le fichier **stdlib.h**. malloc retourne NULL en cas d'échec. Voici un exemple qui illustre une bonne manière de les utiliser :

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

int main()
{
    int * p;

    /* Creation d'un tableau assez grand pour contenir 10 entiers */
 p = malloc(10 * sizeof(int));

    if (p != NULL)
    {
        printf("Succes de l'operation.\n");
        p[0] = 1;
        printf("p[0] = %d\n", p[0]);
        free(p); /* Destruction du tableau. */
    }
    else
        printf("Le tableau n'a pas pu etre cree.\n");

    return 0;
}
```

## VI-B - La fonction realloc

La fonction **realloc** :

```
void * realloc(void * memblock, size_t newsize);
```

permet de « redimensionner » une mémoire allouée dynamiquement (par malloc par exemple). Si memblock vaut NULL, realloc se comporte comme malloc. En cas de réussite, cette fonction retourne alors l'adresse de la nouvelle mémoire, sinon la valeur NULL est retournée et la mémoire pointée par memblock reste inchangée.

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

int main()
{
    int * p = malloc(10 * sizeof(int));

    if (p != NULL)
    {
        /* Sauver l'ancienne valeur de p au cas ou realloc echoue. */
        int * q = p;
        /* Redimensionner le tableau. */
        p = realloc(p, 20 * sizeof(int));

        if (p != NULL)
        {
            printf("Succes de l'operation.\n");
            p[0] = 1;
            printf("p[0] = %d\n", p[0]);
            free(p);
        }
        else
        {
            printf("Le tableau n'a pas pu etre redimensionne.\n");
```

```
            free(q);
        }
    }
    else
        printf("Le tableau n'a pas pu etre cree.\n");

    return 0;
}
```

# VI-C - Exercices

## VI-C-1 - Calcul de la moyenne (version 2)

Reprendre l'exercice IV-D-2 en demandant cette fois-ci à l'utilisateur le nombre de notes qu'il désire entrer (limité à exactement 5 dans l'ancien sujet). Voici un exemple d'exécution :

```
Ce programme permet de calculer votre moyenne scolaire.
Entrez le nombre de notes que vous voulez entrer : 5
Note 1 (0 a 20) : 14
Coef 1 (1 a 5)  : 5
Note 2 (0 a 20) : 10
Coef 2 (1 a 5)  : 5
Note 3 (0 a 20) : 16
Coef 3 (1 a 5)  : 3
Note 4 (0 a 20) : 8
Coef 4 (1 a 5)  : 1
Note 5 (0 a 20) : 12
Coef 5 (1 a 5)  : 1
+------+------+----------+
| Note | Coef | Note Def |
+------+------+----------+
|   14 |    5 |       70 |
+------+------+----------+
|   10 |    5 |       50 |
+------+------+----------+
|   16 |    3 |       48 |
+------+------+----------+
|    8 |    1 |        8 |
+------+------+----------+
|   12 |    1 |       12 |
+------+------+----------+
| Tot. |   15 |      188 |
+------+-----------------+
| Moy. |           12.53 |
+------+-----------------+
Merci d'avoir utilise ce programme. A bientot !
```

## VI-C-2 - Recherche dans un tableau (version 2)

Reprenez l'exercice IV-D-1 en remplaçant les nombres par des chaînes de caractères. L'utilisateur entrera en premier lieu le nombre de chaînes qu'il va entrer.
Aucune chaîne ne doit excéder 20 caractères mais la mémoire que vous utiliserez pour stocker chaque chaîne entrée doit être juste assez grande pour la contenir. Autrement dit, votre objectif sera de minimiser la quantité de mémoire utilisée. Voici un exemple d'exécution :

```
Ce programme permet de trouver toutes les occurrences d'une chaine.
Combien de chaines voulez-vous entrer ? 10
Entrez 10 chaines de caracteres (20 caracteres par chaine au plus) :
t[0] : programmation
t[1] : langage
t[2] : C
t[3] : ANSI
t[4] : ISO
t[5] : IEC
```

```
t[6] : programmation
t[7] : programme
t[8] : code
t[9] : programme
Entrez la chaine a rechercher : programmation
t[0]
t[6]
2 occurrence(s) trouvee(s).
Merci d'avoir utilise ce programme. A bientot !
```

# VI - Solutions des exercices

## VI-A - La lettre X (II-E-1)

Voici la solution la plus évidente et la plus lisible :

```c
#include <stdio.h>

int main()
{
    printf("*   *\n");
    printf(" * * \n");
    printf("  *  \n");
    printf(" * * \n");
    printf("*   *\n");
    return 0;
}
```

On peut cependant remarquer que les espaces après le dernier * de chaque ligne sont inutiles. De plus, rien ne nous oblige à faire un appel à printf pour chaque ligne. On peut faire tout le dessin avec un seul appel à cette fonction. Voici donc une autre solution, un peu moins évidente que la première :

```c
#include <stdio.h>

int main()
{
    printf("*   *\n * *\n  *\n * *\n*   *\n");
    return 0;
}
```

## VI-B - Périmètre d'un rectangle (II-E-2)

```c
#include <stdio.h>

#define LONGUEUR 100
#define HAUTEUR 60

int perimetre(int L, int h);

int main()
{
    printf(
        "Le perimetre d'un rectangle de longueur %d et de hauteur %d est %d.\n",
        LONGUEUR, HAUTEUR, perimetre(LONGUEUR, HAUTEUR)
    );
    return 0;
}

int perimetre(int L, int h)
{
    return 2 * (L + h);
}
```

Ayez l'habitude de représenter les valeurs figées par des macros. Cela permet d'avoir un code à la fois lisible et facilement maintenable : si un jour la spécification change, il suffit de changer les définitions des macros et non plusieurs parties du programme.

## VI-C - Valeur absolue (III-G-1)

Il y a plusieurs façons d'y arriver. Voici deux exemples :

```c
#include <stdio.h>

int main()
{
    double x, y;

    printf("Ce programme permet de determiner la valeur absolue d'un nombre.\n");
    printf("Entrez un nombre : ");
    scanf("%lf", &x);

    if (x >= 0)
        y = x;
    else
        y = -x;

    printf("La valeur absolue de %f est : %f\n", x, y);

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

```c
#include <stdio.h>

int main()
{
    double x;

    printf("Ce programme permet de determiner la valeur absolue d'un nombre.\n");
    printf("Entrez un nombre : ");
    scanf("%lf", &x);

    printf("La valeur absolue de %f est : %f\n", x, (x >= 0 ? x : -x));

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-D - Moyenne (III-G-2)

```c
#include <stdio.h>

int main()
{
    int note;

    printf("Ce programme permet de determiner si vous avez eu la moyenne ou non.\n");
    printf("Entrez votre note (0 a 20) : ");
    scanf("%d", &note);

    if (0 <= note && note <= 20)
    {
        if (note >= 10)
            printf("Vous avez eu la moyenne.\n");
        else
            printf("Vous n'avez pas eu la moyenne.\n");
    }
```

```
    else
        printf("Cette note n'est pas valide.\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-E - L'heure dans une minute (III-G-3)

```
#include <stdio.h>

int main()
{
    int h, m;

    printf("Ce programme permet de determiner l'heure qu'il sera une minute plus tard.\n");
    printf("Entrez l'heure actuelle : ");
    scanf("%d", &h);
    printf("Entrez la minute actuelle : ");
    scanf("%d", &m);

    m++;

    if (m == 60)
    {
        m = 0;
        h++;

        if (h == 24)
            h = 0;
    }

    printf("Dans une minute il sera : %02d:%02d\n", h, m);

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-F - Rectangle (III-G-4)

Le programme a est complet. Pour les programmes b et c, seule la fonction main() est donnée (les autres parties du programme restant inchangées par rapport à a).
a.

```
#include <stdio.h>

void affiche_car(char c, int n);

int main()
{
    int L;

    printf("Ce programme dessine une ligne.\n");
    printf("Entrez la longueur de la ligne (0 a 20) : ");
    scanf("%d", &L);

    if (0 <= L && L <= 20)
    {
        affiche_car('*', L);
        putchar('\n');
    }
    else
        printf("Cette valeur est invalide.\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");
```

```
    return 0;
}

void affiche_car(char c, int n)
{
    if (c > 0 && n > 0)
    {
        int i; /* i : nombres de caracteres deja affiches */

        for(i = 0; i < n; i++) /* on arrete des que i a atteint n */
            putchar(c);
    }
}
```

Vous ne vous attendiez certainement pas à ce que la valeur de c soit testée avant de le passer à putchar. La raison est tout simplement que l'argument attendu par putchar est un int (qui doit avoir une valeur positive) et non un char. Comme char peut faire référence à signed char, il vaut mieux tester la valeur du caractère reçu avant de faire quoi que ce soit avec. putchar sera étudiée dans plus de détails dans le chapitre V.
b.

```
int main()
{
    int L;

    printf("Ce programme dessine un rectangle plein.\n");
    printf("Entrez la longueur du rectangle (0 a 20) : ");
    scanf("%d", &L);

    if (0 <= L && L <= 20)
    {
        int h;

        printf("Entrez la hauteur du rectangle (0 a 20) : ");
        scanf("%d", &h);

        if (0 <= h && h <= 20)
        {
            /* On dessine h lignes (lignes 1 a h) */

            int i; /* i : numero de la ligne en cours */

            for(i = 1; i <= h; i++)
            {
                affiche_car('*', L);
                putchar('\n');
            }
        }
        else
            printf("Cette valeur est invalide.\n");
    }
    else
        printf("Cette valeur est invalide.\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

c.

```
int main()
{
    int L;

    printf("Ce programme dessine un rectangle creux.\n");
    printf("Entrez la longueur du rectangle (0 a 20) : ");
    scanf("%d", &L);
```

```
    if (0 <= L && L <= 20)
    {
        int h;

        printf("Entrez la hauteur du rectangle (0 a 20) : ");
        scanf("%d", &h);

        if (0 <= h && h <= 20)
        {
            int i;

            for(i = 1; i <= h; i++)
            {
             /* Pour les lignes 1 et h, dessiner une ligne continue */
                if (i == 1 || i == h)
                    affiche_car('*', L);
                else
                {
                    /* Pour le reste, seules les extreimites sont visibles */
                    putchar('*');
                    affiche_car(' ', L - 2);
                    putchar('*');
                }

                putchar('\n');
            }
        }
        else
            printf("Cette valeur est invalide.\n");
    }
    else
        printf("Cette valeur est invalide.\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-G - Triangle isocèle (III-G-5)

Ici aussi, seule la fonction main() est donnée pour les mêmes raisons que dans VII-F.
a. Il faut juste remarquer qu'à la ligne i (i allant de 1 à h), il y a *h - i* espaces suivi de *1 + 2 * (i - 1)* étoiles.

```
int main()
{
    int h;

    printf("Ce programme dessine un triangle.\n");
    printf("Entrez la hauteur du triangle (0 a 20) : ");
    scanf("%d", &h);

    if (0 <= h && h <= 20)
    {
        int i;

        for(i = 1; i <= h; i++)
        {
            affiche_car(' ', h - i);
            affiche_car('*', 2 * i - 1);
            putchar('\n');
        }
    }
    else
        printf("Cette valeur est invalide.\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
```

```
}
```

b.

```
int main()
{
    int h;

    printf("Ce programme dessine un triangle.\n");
    printf("Entrez la hauteur du triangle (0 a 20) : ");
    scanf("%d", &h);

    if (0 <= h && h <= 20)
    {
        int i, n, nb_etoiles; /* nb_etoiles : nombre d'etoiles deja affichees */

        for(i = 1, nb_etoiles = 0; i <= h; i++)
        {
            n = h - i;
            affiche_car(' ', n);

            n = 2 * i - 1;
            affiche_car('*', n);

            nb_etoiles += n;

            putchar('\n');
        }

        printf("Il y a %d etoile(s) affichee(s).\n", nb_etoiles);
    }
    else
        printf("Cette valeur est invalide.\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-H - Somme (III-G-6)

```
#include <stdio.h>

int main()
{
    int n, saisis = 0, fini = 0, total = 0;

    printf("Ce programme permet de calculer une somme.\n");
    printf("Entrez la liste des nombres a additionner (terminez en entrant 0).\n");

    while (!fini)
    {
        scanf("%d", &n);
        saisis++;
        total += n;
        fini = (saisis == 10 || n == 0);
    }

    printf("Le total est : %d\n", total);

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-I - Recherche dans un tableau (IV-D-1)

```c
#include <stdio.h>

int main()
{
    int t[10], i, n, nb_occurrences = 0;

    printf("Ce programme permet de trouver toutes les occurrences d'un nombre.\n");
    printf("Entrez 10 nombres :\n");

    for(i = 0; i < 10; i++)
    {
        printf("t[%d] : ", i);
        scanf("%d", t + i);
    }

    printf("Entrez le nombre a rechercher : ");
    scanf("%d", &n);

    for(i = 0; i < 10; i++)
    {
        if (t[i] == n)
        {
            printf("t[%d]\n", i);
            nb_occurrences++;
        }
    }

    printf("%d occurrence(s) trouvee(s)\n", nb_occurrences);

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-J - Calcul de la moyenne (IV-D-2)

```c
#include <stdio.h>

int main()
{
    int t[5][3], i, total_coefs = 0, total_notes = 0;
    double moyenne;

    printf("Ce programme permet de calculer votre moyenne scolaire.\n");

    for(i = 0; i < 5; i++)
    {
        /* Saisie des notes */

        printf("Note %d : ", i + 1);
        scanf("%d", &(t[i][0]));
        printf("Coef %d : ", i + 1);
        scanf("%d", &(t[i][1]));
        t[i][2] = t[i][0] * t[i][1];

        /* Calcul des totaux */

        total_notes += t[i][2];
        total_coefs += t[i][1];
    }

    moyenne = ((double)total_notes) / total_coefs;

    /* Affichage du resultat */

    printf("+------+------+----------+\n");
    printf("| Note | Coef | Note Def |\n");
```

```
    printf("+------+------+----------+\n");

    for(i = 0; i < 5; i++)
    {
        printf("| %4d | %4d | %8d |\n", t[i][0], t[i][1], t[i][2]);
        printf("+------+------+----------+\n");
    }

    printf("| Tot. | %4d | %8d |\n", total_coefs, total_notes);
    printf("+------+-----------------+\n");
    printf("| Moy. | %15.2f |\n", moyenne);
    printf("+------+-----------------+\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}
```

## VI-K - Manipulation de chaînes (IV-D-3)

```
char * str_cpy(char * dest, const char * source)
{
    int i;

    for(i = 0; source[i] != '\0'; i++)
        dest[i] = source[i];

    dest[i] = '\0';

    return dest;
}

size_t str_len(const char * t)
{
    size_t len;

    for(len = 0; t[len] != '\0'; len++)
        /* On ne fait rien, on laisse seulement boucler */ ;

    return len;
}

int str_equals(const char * s1, const char * s2)
{
    int i = 0, equals = 1;

    while (equals && s1[i] != '\0' && s2[i] != '\0')
    {
        if (s1[i] != s2[i])
            equals = 0;
        i++;
    }

    if (equals && (s1[i] != '\0' || s2[i] != '\0'))
        equals = 0;

    return equals;
}
```

Remarquez bien que nous avons choisi char * comme type de retour de str_cpy et non void. La valeur retournée n'est autre que la valeur du pointeur dest. Le prototype de str_cpy est strictement le même que celui de la fonction standard strcpy. Cela permet d'écrire du code compact lorsqu'on a envie, par exemple :

```
char s[50];
strcat(strcpy(s, "Bonjour"), " tout le monde !");
```

Le deuxième argument de la fonction (s2) est déclaré const char * et non simplement char *. En effet, les données pointées par s2 doivent être uniquement lues par la fonction, cette dernière ne doit pas avoir le droit de les modifier. L'ajout du qualificateur const ici est donc plus que recommandé. De plus, cela a un aspect documentaire : la présence ou l'absence de const permet aux utilisateurs d'une fonction de savoir quelles données pourraient être modifiées par la fonction et quelles données seront justes lues.

## VI-L - Calcul de la moyenne (version 2) (VI-C-1)

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

typedef int NOTE[3];

int main()
{
    int n;

    printf("Ce programme permet de calculer votre moyenne scolaire.\n");
    printf("Entrez le nombre de notes que vous voulez entrer : ");
    scanf("%d", &n);

    if (n > 0)
    {
        NOTE * t = malloc(n * sizeof(NOTE));

        if (t != NULL)
        {
            int i, total_coefs = 0, total_notes = 0;
            double moyenne;

            for(i = 0; i < n; i++)
            {
                printf("Note %d : ", i + 1);
                scanf("%d", &(t[i][0]));
                printf("Coef %d : ", i + 1);
                scanf("%d", &(t[i][1]));
                t[i][2] = t[i][0] * t[i][1];
                total_notes += t[i][2];
                total_coefs += t[i][1];
            }

            moyenne = ((double)total_notes) / total_coefs;

            printf("+------+------+----------+\n");
            printf("| Note | Coef | Note Def |\n");
            printf("+------+------+----------+\n");

            for(i = 0; i < n; i++)
            {
                printf("| %4d | %4d | %8d |\n", t[i][0], t[i][1], t[i][2]);
                printf("+------+------+----------+\n");
            }

            free(t);

            printf("| Tot. | %4d | %8d |\n", total_coefs, total_notes);
            printf("+------+------+----------+\n");
            printf("| Moy. | %15.2f |\n", moyenne);
            printf("+------+-----------------+\n");
        }
        else
            printf("Le tableau n'a pas pu etre cree.\n");
    }
    else
        printf("Cette valeur est invalide.\n");

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
```

```
}
```

## VI-M - Recherche dans un tableau (version 2) (VI-C-2)

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

typedef char * PCHAR;

size_t get_string(char * lpBuffer, int nBufSize);

int main()
{
    int n;
    char s[21];

    printf("Ce programme permet de trouver toutes les occurrences d'une chaine.\n");
    printf("Combien de chaines voulez-vous entrer ? ");
    scanf("%d", &n);
    get_string(s, sizeof(s)); /* Pour purger stdin */

    if (n > 0)
    {
         PCHAR * t = malloc(n * sizeof(PCHAR));

         if (t != NULL)
         {
            int i, nb_occurrences = 0, succes = 1;

            printf("Entrez %d chaines de caracteres (20 caracteres par chaine au plus) :\n", n);

            for(i = 0; succes && i < n; i++)
            {
             size_t len;

                printf("t[%d] : ", i);
                len = get_string(s, sizeof(s));
                t[i] = malloc(len + 1);

                if (t[i] != NULL)
                    strcpy(t[i], s);
                else
                {
                    int j;

                    for(j = 0; j < i; j++)
                        free(t[j]);

                    succes = 0;

                    printf("Memoire insuffisante pour continuer.\n");
                }
            }

            if (succes)
            {
                printf("Entrez la chaine a rechercher : ");
                get_string(s, sizeof(s));

                for(i = 0; i < n; i++)
                {
                    if (strcmp(t[i], s) == 0)
                    {
                        printf("t[%d]\n", i);
                        nb_occurrences++;
                    }

                    free(t[i]);
                }
```

```
                printf("%d occurrence(s) trouvee(s)\n", nb_occurrences);
            }

            free(t);
         }
         else
            printf("Le tableau n'a pas pu etre cree.\n");
    }

    printf("Merci d'avoir utilise ce programme. A bientot !\n");

    return 0;
}

size_t get_string(char * lpBuffer, int nBufSize)
{
    size_t len = 0;

    if (fgets(lpBuffer, nBufSize, stdin) != NULL)
    {
     char * p;
        len = strlen(lpBuffer);

        p = lpBuffer + len - 1;
        if (*p == '\n')
        {
            *p = '\0'; /* on ecrase le '\n' */
            len--;
        }
        else
        {
            /* On vide le tampon de lecture du flux stdin */
            int c;

            do
    c = getchar();
            while (c != EOF && c != '\n');
        }
    }

    return len;
}
```

## VIII - Conclusion

Ce tutoriel vous a présenté les éléments indispensables pour bien démarrer avec le langage C. Pour maîtriser un langage (et la programmation), il ne faut cependant pas s'arrêter à une bonne compréhension, il faut aussi acquérir de l'expérience. Améliorer les exemples et les solutions des exercices, se fixer des projets, sont des moyens très efficaces pour obtenir cette expérience et ensuite passer à la suite du cours. Alors, à vos projets !