# Bases sur le langage C

I. Ferrané et M.C. Lagasquie

16 septembre 2021

# Table des matières

# 1 Les prérequis

Tout étudiant est censé avoir acquis ce que nous appelons ici les concepts algorithmiques de base. Ceci est absolument nécessaire pour aborder un langage de programmation quel qu'il soit.
L'écriture d'un programme dans un langage de programmation passe d'abord par une réflexion sur le problème à résoudre et sur la méthode à développer pour y arriver. Cette phase de réflexion préalable indispensable se traduit par l'écriture d'un *algorithme*. Celui-ci permet de décrire la logique du programme c'est-à-dire de décrire l'enchaînement des instructions telles qu'elles seront exécutées au final. L'algorithme ainsi obtenu est ensuite traduit dans le langage de programmation souhaité.
Ces concepts algorithmiques de base indispensables à l'écriture d'un algorithme correspondent aux notions :

- de *variable* et de *type de variable* ;
- d'*affectation* : instruction simple permettant la manipulation de variables ;
- de *lecture* et d'*écriture* : instructions de communication entre la machine et le programme (transmission des données de l'un vers l'autre) ;
- et d'*instructions de contrôle* : instructions permettant de structurer l'ensemble des actions qui constituent un programme. Elles donnent la possibilité de faire des choix (exécution d'un bloc d'instructions seulement en fonction d'une condition), de proposer une alternative (exécution d'un premier bloc d'instructions en fonction d'une condition ou d'un second bloc en fonction de la condition contraire) ou de répéter un nombre déterminé ou indéterminé de fois le même bloc d'instructions.

Pour plus de complément sur ces notions primordiales, vous pouvez consulter [Del97].
Ce document a été bâti en s'appuyant essentiellement sur les ouvrages suivants : [Dri90, Del92, Got93, RS94].

# 2 Concepts fondamentaux du langage C

Les éléments de syntaxe du langage C et les exemples présentés ici sont relatifs à la norme ANSI (voir [Dri90]). Cette norme présente un certain nombre d'avantages par rapport au langage non normalisé, notamment au niveau de la définition de constantes, de l'initialisation de certains types de données (tableaux et structures) et au niveau des contrôles de la conformité des appels de fonctions (contrôle sur le type et le nombre de paramètres, ...).

## 2.1 Jeu de caractères

Il s'agit simplement de l'ensemble des caractères admis pour l'écriture d'un programme en langage C.

- Lettres :
  ```
  a, ..., z, A, ..., Z
  ```
- Chiffres :
  ```
  0, 1, ..., 9
  ```
- Caractères spéciaux :
  ```
  ! + - * / % < > ( ) { }
  [ ] ~ # = ; , : ^ \ ' " .
  | & _ ? espace
  ```
- Commentaires :
  ```
  /* ceci est un commentaire */
  ```

Remarque : un commentaire aide à la lisibilité du programme, il ne correspond pas à une action.

## 2.2 Identificateurs et mots-clés

Les "mots" utilisés pour écrire un programme sont de 2 types : les identificateurs et les mots-clés.

- Un *identificateur* est une suite de caractères alphanumériques (lettres et chiffres) choisie par le programmeur. Cette suite de caractères doit impérativement commencer par une lettre et peut éventuellement contenir un ou plusieurs blancs soulignés (ou caractère underscore) : ex : `var1`, `id_var`, ...Un identificateur ne doit pas correspondre à l'un des mots-clés ci-dessous.
- Un *mot-clé* est un mot réservé ayant une signification prédéfinie pour le langage considéré. Un mot-clé ne peut pas être utilisé comme identificateur. Les mots-clés du langage C sont :

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

## 2.3 Types de données

Le rôle d'un programme est d'effectuer une tâche particulière à partir de données diverses. La nature d'une donnée est caractérisée par son type.

- Le *type* définit l'ensemble des valeurs possibles d'une donnée ainsi que l'ensemble des opérateurs qui peuvent lui être appliqués.
- La *représentation mémoire* d'une donnée correspond au nombre d'octets nécessaires pour stocker l'information correspondante. Ce nombre d'octets correspond à la taille de la donnée et dépend du type de celle-ci et le cas échéant du compilateur ou de la machine.

En langage C, seulement 4 types scalaires existent :

- le type entier (`int`),
- le type caractère (`char`),
- le type réel simple précision (`float`) et double précision (`double`).

À noter, concernant les types que :

- Le type booléen tel qu'il peut être défini dans d'autres langages n'existe pas explicitement dans le langage C. Cependant, la valeur de vérité *faux* est représentée par la valeur entière 0 (zéro) et *vrai* par la valeur entière 1 ou plus généralement par toute valeur différente de 0 (zéro).
- Le type chaîne de caractères n'est pas défini explicitement. Il faut passer impérativement par une donnée structurée c'est-à-dire par un tableau de caractères.
- Le type `void` est un autre type prédéfini dans la langage C et qui a la particularité de ne pas avoir de taille. Il est utilisé généralement pour indiquer l'absence de données (en particulier lors de la définition de fonctions).

**Le type entier : `int`** La taille de la représentation mémoire d'un entier dépend non seulement de la machine et du compilateur associé, mais également du spécificateur de type éventuellement utilisé.

- Représentation mémoire d'une valeur entière : 2 ou 4 octets (ou plus, cela dépend de la machine).
- Intervalle des valeurs entières (sur 2 octets) : -32768...+32767.
- Spécificateurs de type associés au type `int` : il s'agit de 3 attributs (`short`, `long` et `unsigned`) qui influent sur la représentation mémoire de la donnée considérée :
  - `short int` : représentation mémoire d'un entier court sur 2 octets,
  - `long int` : représentation mémoire d'un entier long sur 4 octets.
    Bien que la représentation mémoire associée à toute valeur dépende de la machine, l'inégalité suivante se vérifie toujours :
    ```
    sizeof(short) <= sizeof(int) <= sizeof(long)
    ```
    L'opérateur `sizeof` est un opérateur défini dans le langage C et qui, appliqué à un type de donnée, renvoie le nombre d'octets occupés par toute valeur appartenant à ce type.
  - `unsigned int` : la taille de la représentation mémoire d'un entier (`int`) n'est pas modifiée, mais le bit de signe est utilisé comme un chiffre significatif. Ceci a pour effet d'augmenter l'intervalle des valeurs possibles pour les entiers positifs. On a par exemple sur 2 octets :

    ```
    int                -32768 ... +32767
    unsigned int        0 ... +65535
    ```

    Remarques : le spécificateur `unsigned` peut être combiné avec `short` et `long`. Avec les spécificateurs de type, le mot-clé `int` peut être omis.

**Le type caractère : `char`**

- Représentation mémoire d'un caractère : 1 octet.

Chaque donnée de type caractère a un équivalent entier (codé sur 1 octet) qui correspond au code ASCII du caractère.
— Intervalle des valeurs des codes ASCII[1] : -128 ...+127 ou 0..255 suivant le codage utilisé.

**Les types réels : `float` et `double`** Les réels sont représentés en virgule flottante (partie mantisse et partie exposant). On parle plus généralement de nombre "flottant".
— Représentation mémoire d'un nombre flottant : 4 octets (ou plus, suivant la machine).
— Représentation d'un nombre flottant double précision : 8 octets.

## 2.4 Valeurs constantes ou littéraux

**Valeurs entières** La notation la plus fréquemment utilisée pour représenter un entier est la notation décimale (en base 10). Cependant, il peut être nécessaire de représenter un entier suivant la notation octale (base 8) ou hexadécimale (base 16), lorsque celui-ci représente une adresse par exemple.
En langage C, on peut utiliser une de ces trois notations. Le préfixe `0` (zéro) ou `0x` indique que le nombre n'est pas exprimé en base 10, mais en octal ou en hexadécimal. Exemple : représentation du nombre entier 15 :
— notation décimale (base 10 : 0..9) ex : `15`,
— notation octale (base 8 : 0..7) ex : `017`,
— notation hexadécimale (base 16 : 0..9,a..f) ex : `0xf` ou `0Xf`.
Une valeur entière sera terminée par le suffixe `L` et/ou `U` suivant qu'elle correspond à la valeur d'un entier long (`long int`) et/ou non signé (`unsigned int`). Donc, en fonction du spécificateur de type utilisé, on aura l'une des représentations suivantes pour le nombre décimal 15 :

```
short int          :        15,
long int           :        15L,
unsigned int       :        15U ou 017U,
unsigned long int  :        15UL ou 0xFUL.
```

**Valeurs flottantes** Elles contiennent un point décimal et/ou un exposant. Exemple :
`12.3, 123E-1, 1.23E1`

**Caractères** En plus des caractères alphanumériques (lettres et chiffres), on dispose d'autres types de caractères. Tous ont un équivalent entier qui est le code ASCII.
— Caractère simple : exprimé entre quotes (' ') ou bien représenté par son code ASCII, ici entre parenthèses. Exemple :

```
'A'        (65)
'$'        (36)
' '        (32) caractère espace
```

— Caractère non visualisable : exprimé suivant une notation symbolique. Il correspond généralement à un effet de mise en page :

| | | | |
|---|---|---|---|
| sonnerie | \a | retour arrière | \b |
| tabulation horiz. | \t | tab. verticale | \v |
| retour à la ligne | \n | nouvelle page | \f |
| retour chariot | \r | caractère nul | \0 |

Remarque : une notation équivalente utilise la valeur octale du code ASCII associé au caractère : \t équivaut à \11. Dans ce cas la notation octale n'est pas précédée du marqueur 0.
ATTENTION ! Le caractère nul '\0' (anti slash zéro) est le caractère de code ASCII zéro. Ne pas confondre avec la lettre O ni avec le chiffre 0 (zéro, code ASCII 48).
— caractère spécial : caractère qui a une signification particulière dans la syntaxe du langage C. Aussi, lorsqu'il doit être utilisé comme simple caractère et non comme caractère spécial il doit être précédé du caractère \ (anti slash).

| | | | |
|---|---|---|---|
| guillemet | \" | point d'interrogation | \? |
| quote | \' | anti slash | \\ |

1. A noter qu'il existe plusieurs versions étendues des codes ASCII qui permettent, entre autres, d'encoder des caractères accentués. Du coup, l'encodage des caractères se fera sur 2 octets.

**Chaînes de caractères** La valeur d'une chaîne de caractères est représentée par une suite de caractères délimitée par des guillemets. Elle peut comporter des caractères non visualisables et des caractères spéciaux. Exemple :

```
""                                    /* représente la chaîne vide */
"chaîne de caractères"
"\"chaîne entre guillemets\""
"retour_a_la_ligne\n"
```

ATTENTION ! Le compilateur place automatiquement le caractère '\0' à la fin de chaque valeur de chaîne de caractères.
Exemple : "chaine" comporte 6 caractères visualisables mais est en fait représentée de manière interne par une suite de 7 caractères, soit :

```
'c' 'h' 'a' 'i' 'n' 'e' '\0'
```

Il faut absolument tenir compte de cela lors du stockage d'une chaîne de caractères dans un tableau de caractères puisque le type chaîne n'est pas défini de manière standard.

## 2.5 Constantes

Il est possible en langage C ANSI de définir des constantes :

- soit en utilisant la directive `#define` (voir section 4.1 page 16),
- soit en associant un identificateur avec un type et une valeur constante (non modifiable au cours du programme) :

```
const TYPE ident_constante = valeur ;
```

## 2.6 Variables

Les données manipulées par un programme sont stockées en mémoire centrale. Il y a 2 façons d'accéder à une donnée : soit en utilisant le nom (ou identificateur) qui lui est attribué, soit directement à partir de l'adresse de la zone mémoire où la donnée est stockée.
Une variable représente une donnée dont la valeur peut être modifiée au cours de l'exécution du programme. Une variable ne peut être utilisée qu'après avoir été déclarée.

**Déclaration d'une variable** Une déclaration associe une variable (identificateur) ou un groupe de variables à un type de données. Ce dernier détermine alors la représentation mémoire et la taille de la donnée qui sera manipulée lorsque l'identificateur associé sera utilisé. Une déclaration de variable se situe forcément en début d'un bloc d'instructions, avant toute instruction exécutable.

```
TYPE nom_variable ;
TYPE nom_var1, nom_var2, ..., nom_varN;
```

`TYPE` sera un des types de base (voir section 2.3 page 2) ou bien un type défini par l'utilisateur (voir l'instruction `typedef` présentée plus tard).

**Initialisation d'une variable** L'initialisation est une opération qui consiste à attribuer une valeur initiale à une variable. Ceci peut être fait directement lors de la déclaration ou bien avant la première utilisation de la variable. Dans tous les cas, c'est une opération indispensable si on doit utiliser sa valeur dans une condition ou dans un calcul.

```
TYPE nom_variable = val_initiale ;
```

La valeur d'initialisation doit être du même type que la variable ou d'un type compatible. Dans le second cas, il y aura conversion implicite de la valeur initiale dans le type souhaité (voir section 2.7 page ci-contre).

**Exemples :**

```
int i, j, indice = 0 ;    /* seul indice est initialisé */
char car = '$', c;        /* seul car est initialisé */
float x = 0.0, y = 0.0 ;  /* x et y sont initialisés */
long int res = 0L ;       /* valeur 0 sur 4 octets (ou plus, suivant la machine) */
float z = 0;              /* conversion implicite de 0 en 0.0 */
```

## 2.7 Expressions

Une expression représente une donnée dont il faut déterminer la valeur. Cette valeur est fonction du type de la donnée. Une expression est constituée d'un ou plusieurs opérandes combinés entre eux au moyen d'opérateurs (à chaque type de données correspond un jeu d'opérateurs autorisés - voir sections 2.8 à 2.12 page 7). Une expression simple correspond soit à une constante, soit à une variable, soit à un élément de tableau (voir section 2.15 page 9) ou bien au résultat de l'appel d'une fonction (voir la suite du cours sur les compléments du langage C) : `5, X, T[i], f(X)`, ...
Une expression complexe est une combinaison d'expressions plus simples appelées opérandes et reliées par des opérateurs : `X + 5, Y > (Z+2)`, ...

**Valeur d'une expression** Le rôle d'une expression est d'être évaluée, puis remplacée par sa valeur. Après évaluation d'une expression, si le type de la valeur obtenue n'est pas le même que celui des autres expressions auxquelles elle est combinée, deux cas de figure se présentent :

- soit les types sont incompatibles et cela provoque une erreur de compilation ;
- soit les types sont compatibles (comme `int` et `float` ou `char` et `int` à cause du code ASCII) et une conversion doit être effectuée sans quoi certains résultats pourraient être faussés (voir section 2.14 page 8).

## 2.8 Opérateurs arithmétiques

Ils permettent l'écriture d'expressions arithmétiques permettant le calcul d'une valeur numérique (entière ou réelle).

**Opérateurs binaires (2 opérandes)**

| | | | |
|---|---|---|---|
| `+` | addition | `-` | soustraction |
| `*` | multiplication | `/` | division entière ou réelle |
| `%` | modulo ou reste de la division entière | | |

ATTENTION !

- L'opérateur `%` ne s'utilise qu'avec des opérandes de type entier.
- Si les 2 opérandes de l'opérateur `/` sont des entiers alors le résultat obtenu est le quotient de la division entière.
- Les opérateurs `+, -` et `*` s'utilisent avec des entiers ou des réels.

Remarque : si `c1` et `c2` sont des données de type caractère (`char`) alors `c1+c2` (resp. `c1-c2`) revient à faire la somme (resp. la différence) des valeurs entières correspondant respectivement au code ASCII de `c1` et à celui de `c2`.
Exemple :

```
'B'-'A' /* expression dont la valeur est l'entier 1 */
    /* différence des codes ASCII du caractère A et du */
    /* caractère B ; les codes ASCII des lettres majuscules se suivent */

'0'+1 /* expression dont le résultat est */
    /* l'entier correspondant au code ASCII du caractère */
    /* '1' ; les codes ASCII des chiffres se suivent */
```

**Opérateurs unaires (1 opérande)**

| | | |
|---|---|---|
| `-` | valeur opposée | ex : `Y = -X ;` |
| `++` | incrémentation | ex : `I++` ou `++I` |
| `--` | décrémentation | ex : `I--` ou `--I` |

ATTENTION ! Les opérateurs `++` et `--` sont aussi des opérateurs d'effectation (voir section 2.11 page 7). Et leur position joue un rôle dans l'évaluation du résultat.
`I++` signifie que l'on utilise la valeur de `I` d'abord, puis que celle-ci est incrémentée de 1 (`I = I + 1`) ensuite.
`++I` signifie que l'on incrémente la valeur de `I` de 1 (`I = I + 1`) puis que l'on utilise cette nouvelle valeur.
Exemple :

```
int I = 5, J = 3, K;

K = J * I++ ;         /* K = 15 ; I = 6 ; J = 3 */
K = J * ++I ;         /* I = 6 ; K = 18; J = 3  */
```

## 2.9 Opérateurs relationnels

Ces opérateurs sont utilisés pour construire des expressions logiques élémentaires dont la valeur correspond à une valeur de vérité : *vrai* ou *faux*. En C, le type booléen n'existe pas, aussi, la valeur de vérité *faux* est représentée par la valeur entière 0 (zéro) et la valeur de vérité *vrai* par toute valeur entière différente de 0. Les expressions logiques permettent d'exprimer des conditions et les opérateurs relationnels renvoient la valeur entière 0 ou 1 suivant le cas.

```
<     inférieur                 >     supérieur
<=    inférieur ou égal         >=    supérieur ou égal
==    égalité                   !=    différent
```

ATTENTION!!! Ne pas confondre l'égalité avec l'affectation représentée par un simple signe `=`.
Exemples :

```
int A = 15, B = 10 ;

A < B   /* expression dont la valeur est 0 => faux */
A > B   /* expression dont la valeur est 1 => vrai */
```

## 2.10 Opérateurs logiques

Ces opérateurs permettent la combinaison (et, ou) ou la négation (non) d'expressions logiques (conditions) pour constituer des expressions logiques plus complexes.

### Opérateurs binaires

```
&&    et logique (conjonction)          ||   ou logique (disjonction)
```

Remarque : Les expressions logiques combinées avec ces opérateurs sont évaluées de gauche à droite.

### Opérateur unaire

```
!    non logique (négation)
```

Exemples :

```
int A = 15, B = 10 ;

(A < B)  /* expression logique dont la valeur est 0 => faux */

(B%2 == 0)
   /* expression logique dont la valeur est 1 => vrai */
   /* 10 est divisible par 2 => le reste de la */
   /* division entière de B par 2 est bien égal à 0 */


(A < B) && (B%2 == 0)
   /* expression logique dont la valeur est 0 => faux */
   /* la première expression logique étant fausse */
   /* le résultat d'une conjonction est faux */


(A < B) || (B%2 == 0)
```

```
   /* expression logique dont la valeur est 1 => vrai */
   /* la seconde expression logique étant vraie */
   /* le résultat d'une disjonction est vrai */

!(B%2 == 0)
   /* expression logique dont la valeur est faux */
   /* car négation d'une expression logique vraie */
```

## 2.11 Opérateurs d'affectation

L'affectation est l'opérateur qui permet d'attribuer une valeur à une variable. L'ancienne valeur est alors remplacée par la nouvelle.

### Affectation simple

```
nom_var = expression ;
```

ATTENTION ! Ne pas confondre cet opérateur avec l'opérateur testant l'égalité de 2 expressions (==).
Une expression d'affectation a une valeur : celle de la valeur affectée. Aussi peut-on la trouver dans une expression plus complexe. Il est conseillé cependant de ne pas abuser du procédé pour ne pas diminuer la lisibilité du programme.

```
while (( c = getchar() ) != '$' ) ...

/* c reçoit la valeur du caractère saisi au clavier */
/* Cette valeur est ensuite comparée au caractère $ */
```

### Affectation combinée

```
nom_var = nom_var + expression ;
```

peut se réécrire

```
nom_var += expression ;
```

Même chose avec les autres opérateurs arithmétiques binaires :
`-, *, /, %` utilisés dans des affectations peuvent se réécrire respectivement `-=, *=, /=, %=`.

## 2.12 Opérateurs sur les chaînes de bits

Le langage C est un langage de programmation de haut niveau qui comporte un certain nombre de fonctionnalités bas niveau (proche de la machine). Aussi il est possible de travailler directement sur les chaînes de bits représentant l'information en mémoire à l'aide d'opérateurs spécifiques, utilisables seulement avec des opérandes entiers.

### Opérateur unaire

```
~    complément à un
```

### Opérateurs binaires

```
&    et binaire
|    ou binaire
^    ou exclusif binaire

>>   décalage à droite
<<   décalage à gauche
```

Dans les opérations de décalage, le second opérande doit être un entier non signé. Ces opérateurs peuvent également être combinés avec l'opérateur d'affectation (`&=`, `|=`, `^=`, `>>=`, `<<=` ).
ATTENTION ! ! ! Ne pas confondre les opérateurs `&` et `|` avec les opérateurs logiques `&&` et `||`.

## 2.13 Priorité des opérateurs

```
Priorité maximale        ( )
                          -      ++      --      !      (TYPE)          (op. unaires)
                          *      /       %
                          -      +
                          <      <=      >       >=
                          ==     !=
                          &&
                          ||
Priorité minimale         =      +=      -=      *=     /=     %=     (affectations)
```

Les opérateurs les plus prioritaires sont appliqués en premier. Aussi, dans certain cas, il est indispensable de parenthéser les expressions pour ne pas entraîner des erreurs de calcul.
Exemple :

`A + B * C` est équivalent à `A + (B * C)` et non à `(A + B) * C`.

## 2.14 Conversions

**Conversion implicite** Conversion effectuée automatiquement. Il est cependant déconseillé au programmeur de se "reposer" sur cette possibilité.
Exemple :

```
float X = 0 ;
    /* 0 est converti automatiquement en 0.0  */
int A = 15;
X = X + A ;
    /* A est évalué à 15 et remplacé par sa valeur, */
    /* puis, pour pouvoir être additionné à la */
    /* valeur de X qui est une donnée flottante, */
    /* 15 est converti automatiquement en 15.0 */
```

**Conversion explicite ou cast** Voulue par le programmeur, cette conversion se traduit par le typage de l'expression à convertir. Ainsi, après évaluation, la valeur obtenue est convertie dans le type indiqué avant d'être utilisée.

```
(TYPE) expression
```

ATTENTION ! Lors d'une conversion implicite ou explicite, le type de l'expression n'est pas modifié, seule sa valeur est convertie.
Exemple :

```
float MOY = 0.0;
int A = 15, B = 26;

MOY = (float) (A + B) / 2 ;
```

Commentaire : Le résultat de l'expression `A + B` est ici converti explicitement en valeur flottante (41 devient 41.0). Ainsi, la valeur 2 fait alors l'objet d'une conversion implicite (2 devient 2.0). Sans cela, les deux opérandes étant entiers, c'est la division entière qui aurait été effectuée. Le résultat obtenu aurait alors été erroné puisque la partie décimale aurait été tronquée (`MOY` récupérerait alors 20.0 au lieu de 20.5). C'est un exemple simple de résultat faussé par la combinaison d'expressions de types différents mais compatibles (`MOY` est de type `float` alors que `A` et `B` sont de type `int`). Le problème ne se pose pas si on écrit :

```
MOY = (A + B) / 2.0 ;
    /* conversion implicite du résultat de A+B => division réelle effectuée */
```

## 2.15 Tableaux

Un tableau correspond à une zone mémoire où des données variables de *même type* (élémentaire ou non) sont stockées consécutivement. Ainsi, l'accès à chaque élément du tableau peut être fait au moyen d'un indice. Un tableau peut être représenté schématiquement par une suite de "cases", chaque case correspondant à une donnée et désignée par une valeur d'indice.

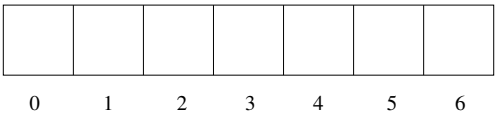



Tableau de 7 éléments maximum (= 7 "cases") indicés de 0 à 6

**Indice d'un tableau**

- la valeur de l'indice d'un tableau est obligatoirement une valeur entière et positive,
- la première valeur d'indice (= indice de la première case) est obligatoirement 0 (zéro) et non 1,
- la dernière valeur d'indice (= indice de la dernière case) correspond donc au nombre d'éléments que peut contenir le tableau moins 1. Si on définit un tableau de 10 éléments l'indice du dernier élément sera 9.

Par conséquent, l'indice d'un tableau de `N` éléments est une valeur entière qui doit obligatoirement prendre sa valeur dans l'intervalle `0..N-1`.
Exemple :
Si `TAB` est le nom du tableau alors on note `TAB[i]` l'élément d'indice `i`. Ainsi, `TAB[0]` désigne le premier élément du tableau (= valeur stockée dans la 1ère case).

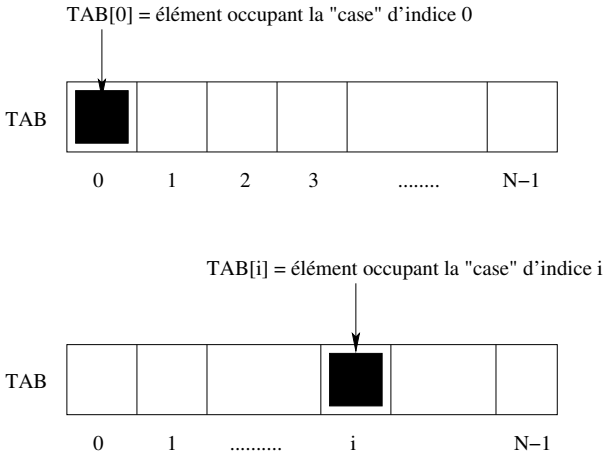
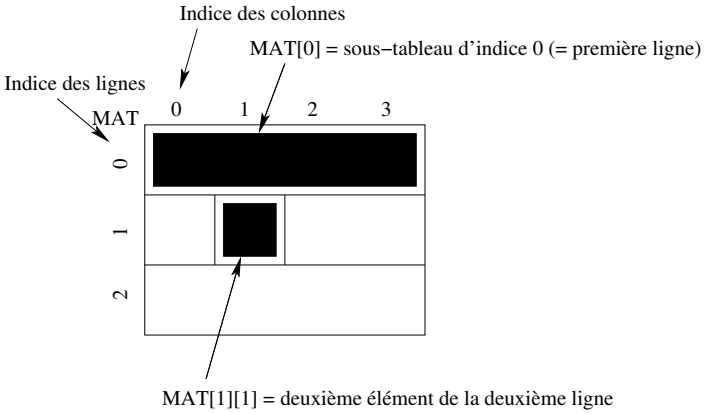



Tableau de N éléments maximum (= N "cases") indicés de 0 à N–1

**Tableau multidimensionnel :** C'est un tableau ayant plus d'une dimension. Par exemple, un tableau à 2 dimensions permet la représentation et la manipulation de matrices. On peut accéder soit à un élément de la matrice (accès doublement indicé), soit à une ligne complète (accès simplement indicé).
En fait, un tableau à 2 dimensions peut être défini comme un tableau d'éléments qui sont eux-mêmes des tableaux. Ainsi, une matrice peut être vue comme un tableau de lignes, une ligne étant un tableau unidimensionnel d'éléments.
Exemple :
Si `MAT` est une matrice (= tableau à 2 dimensions), `DIM1` représentant le nombre maximum de lignes et `DIM2` le nombre maximum de colonnes, alors pour toute valeur de `i` comprise entre 0 et `DIM1-1`, et pour toute valeur de `j` comprise entre 0 et `DIM2-1` :

- `TAB[i]` représente une ligne de la matrice (= tableau unidimentionnel comportant `DIM2` éléments) ;
- `TAB[i][j]` représente un élément de la matrice et plus exactement l'élément se trouvant à l'indice `j` de la ligne d'indice `i`.




Représentation d'un tableau à 2 dimensions : 3 lignes x 4 colonnes

Ceci peut être généralisé à des tableaux de dimension supérieure à 2.

**Déclaration d'un tableau** Lors de la déclaration d'un tableau, on spécifie le type des éléments, le nom du tableau et pour chaque dimension, le nombre maximum d'éléments. Ce nombre doit être connu au moment de la compilation du programme. Par conséquent, elle doit correspondre à une constante entière strictement positive.
ATTENTION ! ! Ne pas utiliser une variable, même initialisée, pour définir la taille d'un tableau, car une variable ne prend sa valeur qu'au moment de l'exécution du programme.

```
/* Tableau à une dimension */
TYPE nom_tableau [taille];

/* Tableau à 2 dimensions */
TYPE nom_tableau [dim1][dim2];

/* Tableau à N dimensions */
TYPE nom_tableau [dim1][dim2]...[dimN];
```

Exemple :

```
char chaine[20];
int tab[5] ;
float nb_reels[10][5];
```

Commentaire : `tab` est un tableau de 5 entiers, `chaine` un tableau de 20 caractères et `nb_reels` un tableau à 2 dimensions (10 lignes x 5 colonnes), chaque élément étant un nombre flottant.

**Initialisation d'un tableau** Un tableau peut être initialisé lors de sa déclaration. Pour cela, on spécifie dans l'ordre et entre accolades, les valeurs de chacun des éléments du tableau. Si la déclaration du tableau s'accompagne d'une initialisation alors, dans ce cas et seulement dans ce cas, la taille du tableau peut être omise. Elle est alors déduite du nombre de valeurs présentes dans la liste.

```
/* Tableau à une dimension */
TYPE nom_tab [taille] = { liste de valeurs } ;
TYPE nom_tab [ ] = { liste de valeurs };

/* Tableau à 2 dimensions */
TYPE nom_tab [dim1][dim2] = { liste de listes} ;
TYPE nom_tab [][dim2] = {liste de listes} ;
```

Chaque sous-liste correspond à la valeur d'un sous-tableau (ou ligne). Ainsi, la première sous-liste ira dans la première ligne (indice 0 pour la première dimension), la 2ème dans la 2ème ligne, ainsi de suite. Lors de l'initialisation d'un tableau, seule la valeur de dernière dimension est indispensable. Les autres, si elles sont omises (= crochets vides `[]`), seront déduites de l'ensemble des valeurs d'initialisation.
Exemples :

```
int filtre_binaire [] = {0, 1, 0, 0, 1} ;
char chaine[] = "initialisation" ;
char voyelle[6] = {'a','e','i','o','u','y'} ;

float matrice[2][2] = { {0.0, 1.0} {1.0,0.0} };
char l_mot[][20] = {"rappel", "de", "cours"};
```

Commentaire : `filtre_binaire` est un tableau de 5 entiers ; `chaine` un tableau de 15 caractères (14 visualisables + 1 marquant la fin de chaîne) ; `voyelle` est un tableau de 6 caractères (non utilisable comme chaîne puisqu'il n'y a pas le marqueur de fin de chaîne '\0') ; `l_mot` est un tableau de 3 chaînes de caractères (chaque chaîne ayant au maximum 19 caractères + celui de fin de chaîne).

# 3 Instructions

Un programme correspond à un ensemble d'instructions permettant la manipulation de différentes données. Une instruction correspond à une action que doit effectuer la machine lors de l'exécution d'un programme. Il existe 3 catégories d'instructions pour traduire une action élémentaire ou plus complexe : les instructions simples, les instructions composées et les instructions de contrôle.

## 3.1 Instructions simples

Toute instruction simple se termine par un `;`. Il s'agit généralement d'une expression d'affectation ou bien d'un appel à un sous-programme ne renvoyant pas de valeur ou dont on n'utilise pas la valeur de retour.
Exemple :

```
a = 3 ;                                 /* affectation d'une valeur constante */
c = a + b ;                             /* affectation de la valeur d'une
                                              expression arithmétique */
i++ ;                                   /* équivalent à i = i + 1  */
j += i ;                                /* équivalent à j = j + i */
printf("resultat = %d\n", c) ;          /* appel à la fonction d'écriture */
;                                       /* instruction vide ou nulle */
```

Il est possible d'enchaîner les instructions simples en utilisant un opérateur spécifique : l'opérateur `virgule` (`,`). Cet opérateur permet d'évaluer des expressions l'une après l'autre, de la gauche vers la droite, et renvoie la valeur de la dernière expression évaluée. Par exemple :

```
int a, b=10, c ;
printf("resultat = %d\n", (a = 3 , c = a+b)) ;   /* les deux affectations sont
                                                    realisees et la valeur affichee
                                                    est le résultat de la derniere
                                                    affectation (ici 13) */
```

Attention, la virgule servant aussi de séparateur (par exemple lors de l'initialisation d'un tableau – voir section 2.15), cet opérateur est à utiliser avec parcimonie (le code risquant de devenir illisible).

## 3.2 Instructions composées ou bloc d'instructions

Séquence d'instructions simples, composées et/ou de contrôle, encadrée par des accolades ( {...} ) et appelée généralement bloc d'instructions. Cette séquence constitue alors une action complexe à réaliser par le programme.
Exemple :

```
{ /* debut de bloc */
        printf("Message d'erreur") ;
        traiter_erreur ();
} /* fin de bloc : pas de ; */
```

Les déclarations de variables, de tableaux et autres données nécessaires à l'exécution des instructions doivent être faites en début de bloc (après l'accolade ouvrante et avant la première instruction du bloc). Dans ce cas, elles sont locales au bloc et n'existent plus une fois sorti du bloc (après accolade fermante).

## 3.3 Instructions de contrôle

Il s'agit de l'implémentation des instructions définies du point de vue algorithmique et dont le rôle est de structurer le programme : choix (simple, multiple, avec alternative), et répétition.
Notation : dans ce qui suit, `instruction` représente une instruction quelconque (simple, composée ou complexe). L'expression représentant la condition qui va déterminer le choix ou la répétition doit être parenthésée.

### 3.3.1 Les sélections

Il en existe deux types.

**Instruction IF ELSE** Utilisée pour effectuer une action particulière en fonction du résultat d'un test logique (= valeur de vérité d'une condition).

***Test simple ou choix*** `if (expression) instruction`

`expression` : expression logique simple (variable, appel de fonction, ...) ou complexe (conjonction et/ou disjonction et/ou négation de condition)

`instruction` : n'est exécutée que si `expression` a une valeur différente de zéro (donc *vrai*), ignorée sinon.

ATTENTION ! En langage C il n'y a pas de mot-clé `then`

Exemple :

```
int X, Y;

if ( Y != 0 ) X = X/Y ;
   /* X reçoit le quotient de la division entière */
   /* de X par Y seulement si Y n'est pas nul */

if (trouve_solution) printf("OK !");
   /* le message OK ! n'est affiché que si la variable */
   /* trouve_solution a une valeur non nulle = vrai */

if ( (X>0) && (Y>0) )
   {       printf("OK !");
           Z = X % Y;
   }
   /* le message OK ! est affiché et Z reçoit le reste */
   /* de la division entière de X par Y seulement */
   /* si X et Y sont des entiers positifs non nuls */
```

***Test avec alternative :*** `if (expression) instruction_1 else instruction_2`

La clause `else` exprime une alternative. `instruction_1` et `instruction_2` sont des instructions qui seront exécutées en fonction de la valeur de vérité de `expression`.

Si `expression` a une valeur différente de 0 (= *vrai*) alors `instruction_1` est exécutée et `instruction_2` ignorée. Sinon si `expression` vaut 0 (= *faux*) `instruction_2` est exécutée et `instruction_1` ignorée.

Exemple :

```
if ( Y != 0 ) X = X/Y ;
else printf("Division par zéro \n");
   /* la division n'est effectuée que si le diviseur */
   /* est différent de 0 sinon un message est affiché */
```

ATTENTION ! On peut imbriquer plusieurs instructions de ce type, mais il faut faire attention aux ambiguïtés, c'est-à-dire savoir à quel `if` se rapporte un `else`. Par défaut, une clause `else` est rattachée au `if` précédent.

***Opérateur conditionnel " ? : "*** Certaines alternatives simples peuvent s'écrire au moyen d'un opérateur plutôt qu'avec l'instruction `if else`.

```
(expression_1) ? expression_2 : expression_3 ;
```

équivaut à

```
if (expression_1) expression_2 ; else expression_3 ;
```

Exemple :

```
if ( A < B ) MIN = A ;
else MIN = B ;
```

se traduira par

```
MIN = ( A < B ) ? A : B ;
```

**Instruction SWITCH** Ce type d'instruction permet de sélectionner un groupe précis d'instructions parmi plusieurs en fonction de la valeur d'une expression.

```
switch (expression)
{   case valeur1 :  suite_instructions_1
                    break;
    case valeur2 :  suite_instructions_2
                    break;
    ...
    default :       suite_instructions_N
}
```

`expression` est évaluée et doit renvoyer un entier ou un caractère. La valeur obtenue détermine directement la `suite_instructions_i` à exécuter.
Si aucune valeur ne correspond, aucune séquence d'instructions n'est exécutée à moins qu'une clause `default` ait été définie. Dans ce cas, ce sont les instructions associées qui sont exécutées.

**Instruction break** Elle permet de forcer la sortie du bloc d'instruction courant. Utilisée dans un `switch`, elle permet, après l'exécution de la séquence d'instructions associée à une valeur donnée, de reprendre l'exécution du programme à la première instruction suivant l'accolade fermante du `switch`. Si l'instruction `break` est absente alors les séquences d'instructions correspondant aux valeurs suivantes sont également exécutées.
Pour une utilisation du `switch` équivalente à des choix (`if ...else`) imbriqués, il faut systématiquement mettre l'instruction `break` à la fin de chaque séquence d'instructions.
Remarque : On peut exprimer le fait qu'une séquence d'instructions soit commune à plusieurs valeurs en spécifiant chacune de ces valeurs au moyen d'une clause `case` et en associant la suite d'instructions à exécuter (terminée par `break`) à la dernière valeur.
Exemple :

```
/* séquence d'instructions correspondant à une action */
/* à effectuer en fonction de la réponse de */
/* l'utilisateur */

char choix ;
int fin ;

printf("Voulez-vous continuer ?: ");
```

```
choix = getchar();
switch(choix)
{       case 'o' :
        case 'O' :  fin = 0;
                    /* la même action est à faire */
                    /* si la réponse est O majuscule */
                    /* ou minuscule */
                    break;
        case 'n' :
        case 'N' :  fin = 1;
                    /* la même action est à faire */
                    /* si la réponse est N majuscule */
                    /* ou minuscule */
                    break;
        default :   printf("Repondre par o / n ");
                    /* action a faire si la réponse */
                    /* de l'utilisateur n'est ni O ni N */
}
```

### 3.3.2 Les répétitions

Il en existe 3 types.

**Instruction WHILE** Permet de répéter une action tant qu'une condition est vérifiée. La condition est exprimée par une expression dont la valeur détermine la valeur de vérité de la condition.

`while (expression) instruction`

La partie `instruction` est exécutée tant que `expression` a une valeur différente de 0 (= *vrai*).
IMPORTANT : Pour effectuer un premier passage (= "entrer dans la boucle") il faut s'assurer que la condition exprimée par `expression` est vraie au moment du test. On "sort" de la boucle lorsque la condition correspondant à `expression` est fausse. La partie `instruction` doit donc obligatoirement modifier à un moment donné la valeur de l'expression sinon la boucle sera infinie !
ATTENTION ! Il n'y a pas de mot-clé `do` associé à une boucle `while`
Exemple :

```
int chiffre = 0;
   /* chiffre est initialisé à 0 ;        */

   /* 0 est bien inférieur ou égal à 9    */
   /* alors on peut entrer dans la boucle */
while( chiffre <= 9)
     {
        printf("%d\n", chiffre);
        chiffre++;
           /* la valeur de chiffre est incrémentée de 1 */
     }
   /* au fur et a mesure des incrémentations      */
   /* chiffre est passé de 0 à 10 ; chiffre n'est */
   /* pas inférieur ou égal à 9 alors on sort     */
   /* de la boucle                                */
```

Remarque : `expression` en condition est optionnelle. Si elle est absente, on obtient une boucle infinie.

**Instruction DO WHILE** Autre type de répétition ayant pour rôle d'exécuter l'instruction tant que la condition est vraie, c'est-à-dire tant que l'expression associée a une valeur différente de 0.
ATTENTION ! ! ! Il ne s'agit pas exactement de l'implémentation de l'instruction algorithmique

`repéter ...jusqu'à`

mais plutôt d'un

répéter ...tant que

Par conséquent, la seule différence qui existe entre une boucle WHILE et une boucle DO WHILE est que, dans le premier cas, le test est fait *avant* de rentrer dans la boucle et dans le second il est fait *après* le premier passage. Par conséquent, la partie `instruction` d'une boucle DO WHILE est exécutée au moins une fois.

```
do instruction while (expression);
```

Exemple :

```
int chiffre = 0;

do
    printf("%d\n", chiffre++);
while ( chiffre <= 9) ;
```

Commentaire : On a condensé ici en une seule instruction les 2 instructions constituant le bloc d'instructions à répéter de la version précédente. `chiffre` est affiché puis incrémenté. Si `chiffre` n'avait pas été initialisé, sa valeur aurait été quelconque et, à supposer que cette valeur quelconque soit supérieure à 9, elle aurait été affichée quand même.
Remarque : `expression` en condition est optionnelle. Si elle est absente, on obtient une boucle infinie.

**Instruction FOR** Troisième type de répétition existant en C. L'instruction `for` permet d'exécuter une instruction un nombre donné de fois.

```
for (expression_1 ; expression_2 ; expression_3) instruction
```

`expression_1` : initialise la variable de boucle (affectation).
`expression_2` : représente la condition à satisfaire pour exécuter la partie `instruction` (Attention ! Ce n'est pas l'expression de la condition d'arrêt).
`expression_3` : sert à modifier la variable de boucle (incrémentation, décrémentation ou affectation).
`instruction` : séquence de code à exécuter plusieurs fois.
ATTENTION ! ! ! Pas de mot-clé `do` dans un `for`.
Exemple :

```
int chiffre;

for (chiffre = 0 ; chiffre <= 9 ; chiffre++)
     printf("%d\n", chiffre);
```

L'instruction `for` a une utilisation plus large que l'équivalent algorithmique POUR . . .FAIRE. De fait, une boucle `for` est toujours traduisible en une séquence de code incluant une boucle `while` :

```
for (expression_1 ; expression_2 ; expression_3) instruction
```

se traduit en :

```
expression_1;
while (expression_2)
    {
     instruction ;
     expression_3 ;
    }
```

Remarque : les 3 expressions (`expression_1`, `expression_2` et `expression_3`) sont optionnelles. Si `expression_2` est absente, on obtient une boucle infinie. Par exemple, on peut écrire `for(;;)`, ce qui correspond à une boucle infinie sans initialisation.

# 4 Structure d'un programme en langage C

Un programme en langage C se décompose au minimum en deux parties : les directives et la fonction principale `main`. Mais sa structure peut être plus complexe :

```
/* les directives */
/* définition de variables globales */
/* définition des fonctions (y compris la fonction main) */
```

Les variables globales seront connues de *toutes* les fonctions déclarées ensuite. Chaque fonction sera connue de *toutes* les fonctions déclarées ensuite. La fonction `main` est, en général, définie en dernier puisqu'elle doit connaître tout le reste.
Dans ce document, nous allons nous intéresser uniquement aux directives et à la définition de la fonction `main`. La possibilité de définir d'autres fonctions sera vue ultérieurement et la définition des variables globales a déjà été présentée en section 2.6 page 4.

## 4.1 Les directives

Ces directives débutent par le caractère `#` et s'adressent au préprocesseur. Celui-ci opère juste avant la phase de compilation proprement dite. Son rôle est de remplacer chaque directive par ce qu'elle représente.

**Inclusion de fichiers : `#include`** Cette directive doit être suivie d'un nom de fichier. Toute ligne correspondant à la directive `#include` est alors remplacée par le contenu du fichier mentionné. Il s'agit en général d'un fichier contenant des déclarations de fonctions et de variables externes et suffixé par `.h`. Ces fichiers sont appelés "header". Ce mécanisme permet de référencer les fichiers contenant la définition ou la déclaration de données et/ou de fonctions externes au programme mais utilisées dans celui-ci.
Si le fichier inclus correspond à une bibliothèque de fonctions standards (fonctions C prédéfinies), son nom doit être spécifié entre `<` et `>` :

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

S'il s'agit d'un autre type de fichier, son nom sera donné entre guillemets :

```
#include "mon_fichier.h"
```

**Définition de constantes symboliques : `#define`** Cette directive permet la définition d'une constante symbolique. Avant la compilation, chaque occurrence de cette constante est remplacée par la valeur qui lui a été associée.

```
#define NOM_CONST valeur
```

Dans d'autres cas cette directive permet simplement de définir une constante comme existante. Son existence pourra être testée avec les directives conditionnelles comme `#ifdef` ou `#ifndef`. Ce point ne sera pas abordé dans ce document.

## 4.2 La fonction principale `main`

Un programme C est constitué d'un ensemble de sous-programmes systématiquement appelés fonctions. La fonction principale a un nom prédéfini : `main`. Elle correspond au point d'entrée du programme, c'est-à-dire que l'exécution du programme commencera toujours par le corps de cette fonction.
La structure de n'importe quelle fonction (y compris de la fonction `main`) est la suivante :

```
/* en tête de la fonction : */
type_de_retour nom_fonction (liste de paramètres)
{
   /* définition de variables locales */
   /* instructions */
}
```

Les variables locales d'une fonction ne seront connues que dans les instructions de cette fonction.
Dans le cas de la fonction `main`, on peut avoir trois types d'en-tête. Pour un programme peu complexe, la fonction `main` n'admet pas de paramètre et ne renvoie pas de valeur :

```
/* définition d'une fonction main sans valeur de retour*/

void main(void)
{ /* définition de variables locales */
```

```
  char mot[] = "Bonjour !";
  /* instructions */
  puts(mot); /* affichage de la chaîne mot */
}
```

Cependant, suivant les besoins, il peut être nécessaire que le programme renvoie un code de retour qui pourra éventuellement être exploité par le système :

```
/* définition d'une fonction main renvoyant un code */
/* de retour entier à la manière des commandes UNIX */

int main(void)
{  /* définition de variables locales */
   ...
   /* instructions */
   ...
   /* renvoi de la valeur de retour */
   return(...);
}
```

Attention, dans la norme ANSI, la fonction `main` doit être déclarée comme renvoyant un `int` même si elle ne retourne rien (donc un compilateur respectant cette norme renverra une erreur non bloquante sur le premier exemple d'en-tête).
Il peut également s'avérer nécessaire de paramétrer un programme. Dans ce cas, la fonction `main` devra admettre 2 paramètres de noms prédéfinis : `argv` et `argc`. Les valeurs des paramètres seront alors fournies lors de l'appel du programme. Ce point n'est pas abordé ici.

# 5 Compilation d'un programme en langage C

Un programme est la transcription en langage source (ici le langage C) d'un algorithme. Il est saisi par le programmeur sous un éditeur de texte (`vi`, `emacs`, `xemacs`, `nedit`, ...), puis sauvegardé dans un fichier dont le nom doit obligatoirement être terminé par l'extension `.c`. Un tel fichier est appelé fichier ou programme source.
Pour pouvoir exécuter un programme C, il faut obtenir un fichier exécutable (code binaire) à partir du programme source. Cette étape correspond à la compilation et à l'édition de liens.

## 5.1 Vérification de la syntaxe d'un programme

```
lint [-options] fichier_source
```

Lorsqu'elle est disponible, la commande `lint` appliquée à un programme source (fichier suffixé par `.c`) ne génère aucun code (c'est-à-dire pas de fichier exécutable). Elle détecte les erreurs de syntaxe et avertit le programmeur ("warning") lorsque certaines instructions risquent de nuire à la portabilité du programme ou à son exécution (cf. le manuel en ligne : `man lint`).

## 5.2 Compilation et édition de liens

Plusieurs compilateurs peuvent être disponibles : `cc, gcc`, ...Ils sont chargés de la traduction d'un programme source en langage machine (code binaire) et de la création d'un fichier exécutable. Ce processus comprend plusieurs étapes.

**Compilation** Consiste à vérifier si le programme a été écrit conformément à la syntaxe du langage utilisé. Si aucune erreur de compilation n'est détectée alors le compilateur génère un fichier objet portant le même nom que le fichier source mais avec l'extension `.o`.

**Edition de liens** Partant du fichier objet (.o), le compilateur vérifie que toutes les variables et les fonctions utilisées dans le programme sont bien définies quelque part, soit dans le programme, soit dans une des bibliothèques de fonctions utilisées (fichiers inclus). Il s'agit de faire le lien entre les objets utilisés et leur définition. Si aucune erreur d'édition de liens n'est détectée alors le compilateur peut générer le fichier. Le nom d'un programme exécutable est a.out par défaut.
Une option de compilation permet de dissocier les 2 étapes sinon elles s'enchaînent automatiquement et on obtient directement l'exécutable à partir du programme source.

**Commandes de compilation** On suppose ici que le programme écrit est constitué d'un seul fichier source.
Utilisée sans option et appliquée à un fichier source la commande `cc` (ou `gcc` compilateur du GNU) génère un fichier exécutable dont le nom par défaut est a.out.

```
cc [-options] fichier_source
ou
gcc [-options] fichier_source
```

***Principales options :***

- **o** donne un nom à l'exécutable autre que celui attribué par défaut.
- **g** génère l'exécutable de façon à ce que l'on puisse utiliser un débogueur pour effectuer la mise au point du programme : exécution pas à pas, trace, ...
- **c** supprime l'appel automatique à l'éditeur de liens. Le seul fichier généré correspond au fichier objet (fichier .o). L'édition de liens doit alors être faite séparément.

***Exemples de commandes de compilation :***

| | |
|---|---|
| `cc prog.c` | compile et fait l'édition de liens du programme prog.c ; s'il n'y a pas d'erreur, l'exécutable de nom a.out est généré. |
| `cc prog.c -o prog.out` | compile et fait l'édition de liens du programme prog.c ; s'il n'y a pas d'erreur, l'exécutable de nom prog.out est généré. |
| `cc -g prog.c -o prog.out` | idem, mais l'exécutable est construit de façon à pouvoir être utilisé sous un débogueur (option g : génération d'une table de symboles,...). |
| `cc -c prog.c` | compile le programme prog.c et génère le fichier objet correspondant prog.o (mais pas d'exécutable), l'édition de liens doit être faite ultérieurement (avec la commande `cc prog.o`). |

# 6 Les entrées-sorties (bibliothèque stdio.h)

Différentes bibliothèques de fonctions standards sont à la disposition des développeurs de programmes en langage C. Ces bibliothèques sont constituées de fichiers .h regroupant les fonctions de même type. Exemple :

**stdio.h :** fonctions de gestion des entrées-sorties

**math.h :** fonctions mathématiques

**string.h :** fonction de gestion des chaînes de caractères

Ces fichiers contiennent généralement la déclaration de chaque fonction standard ainsi que, dans certains cas, la définition de types de données (ex : le type `FILE` défini dans stdio.h) ou de constantes symboliques (ex : `NULL`).
Pour pouvoir utiliser une fonction standard, il faut inclure le fichier correspondant à la bibliothèque contenant la déclaration de la fonction. Pour cela il faut utiliser la directive `#include` (voir section 4.1 page 16).
La première bibliothèque dont on a besoin dans un programme est celle permettant l'échange d'informations entre la machine et les dispositifs d'entrées-sorties (clavier, écran, fichiers textes, fichiers binaires, ...) : la bibliothèque stdio.h.
Certaines des fonctions de cette bibliothèque lisent sur l'entrée standard (clavier par défaut) ou écrivent sur la sortie standard (écran par défaut). D'autres permettent le transfert d'informations depuis ou vers des fichiers de données.

Dans cette partie, nous ne présentons que les fonctions de lecture et d'écriture élémentaires. Les opérations sur les fichiers de données seront présentées ultérieurement.
À noter : Pour plus de détails sur la notion d'entrées-sorties standard se référer à la partie UNIX du cours (voir le polycop correspondant).

## 6.1 Entrées-sorties de caractères

**Fonction getchar :** `int getchar(void)`

N'admet pas de paramètre (`void`). Lit un caractère sur l'entrée standard (celle-ci correspondant par défaut au clavier) et renvoie le code ASCII du caractère lu (`int`).

**Fonction putchar :** `int putchar( char c)`

Écrit sur la sortie standard (écran par défaut) le caractère `c` donné en paramètre et renvoie le code ASCII du caractère écrit (`int`).

## 6.2 Entrées-sorties de chaînes de caractères

**Fonction gets :** `char * gets( char * ch)`

Lit une suite de caractères saisie au clavier et terminée par un retour à la ligne ('`\n`'). La chaîne lue est placée dans le tableau de caractères désigné par le paramètre fourni (`ch`). Le caractère de fin de ligne '`\n`' est normalement remplacé par le marqueur de fin de chaîne de caractères '`\0`'.

**Fonction puts :** `int puts( const char * ch)`

Écrit sur la sortie standard (écran) la chaîne de caractères donnée en paramètre, suivie d'un retour à la ligne. Si l'opération d'écriture s'est déroulée sans problème, la valeur renvoyée est celle du nombre de caractères écrits, sinon c'est la valeur correspondant à `EOF` (-1) qui est renvoyée.

## 6.3 Entrées-sorties formatées

Indispensables pour lire ou écrire des données numériques (entiers et réels) ou bien pour lire ou écrire un flot de données mixte : valeurs numérique, caractères et chaînes de caractères.
Compte tenu de la nature diverse des données, il est donc obligatoire de spécifier le format de chaque donnée à traiter. C'est pourquoi on parle d'entrées-sorties formatées.

**Fonction printf :** `int printf( const char * format, [arg1, ..., argN])`

Écrit des données sur la sortie standard. Si des arguments `arg1, ..., argN` sont spécifiés [2], leur valeur respective est écrite sur la sortie standard conformément à la spécification de format fournie dans la chaîne `format` donnée en paramètre.

- *chaîne* `format` : il s'agit d'une chaîne de caractères directement donnée entre " " et précisant comment le flot de données doit être interprété pour être écrit sur la sortie standard. Elle est composée de texte et/ou de spécificateurs de format.
- *spécificateur de format* : il définit le type et éventuellement la taille de la donnée à écrire. Il débute par le caractère `%` suivi :
  - d'une valeur entière, qui, lorsqu'elle est précisée correspond à la *taille minimum de la donnée* (= nombre de caractères s'il s'agit d'une chaîne ou nombre de chiffres s'il s'agit d'une donnée numérique).
  - d'une seconde valeur entière, qui, lorsqu'elle est précisée indique soit le *nombre de chiffres* à écrire après le point décimal lorsque la donnée est de type `float` ou `double`, soit le *nombre maximum de caractères* de la chaîne à écrire réellement. Afin de la distinguer de la taille minimum, cette valeur, appelée précision, est toujours précédée d'un point.
  - du *caractère de conversion* précisant la forme selon laquelle la donnée devra être écrite.

  (ex : `%s, %15s, %.3f, %5.3f,` ...)

2. le fait qu'ils apparaissent entre crochets dans la définition précédente signifie qu'ils sont optionnels.

| Principaux caractères de conversion | Signification | Type correspondant |
|---|---|---|
| `c` | simple caractère | `char` |
| `s` | chaîne de caractères terminée par '\0' | `char *` |
| `d` ou `i` | entier décimal signé | `int` |
| `o` | entier sous forme octale (non précédé de 0) | `unsigned int` |
| `x` | entier sous forme hexadécimale (non précédé de 0x) | `unsigned int` |
| `f` | Valeur réelle ( avec point décimal mais sans exposant) | `float` ou `double` |
| `e` | Valeur réelle ( avec un seul chiffre avant le point décimal et une partie exposant) | `float` ou `double` |
| `g` | `e` ou `f` avec indication d'une précision | |
| `l` | Combiné avec `d,u,o,i` ou `x` pour un entier long | `long int` |

— *arguments* : désignent les valeurs à écrire. Ils peuvent correspondre à une constante, une expression, un appel de fonction, une variable, un tableau de caractères, ...
IMPORTANT : Le type de la donnée à écrire doit être compatible avec le format spécifié. Le nombre de spécificateurs de type donnés dans la chaîne `format` doit correspondre au nombre de données à écrire.

Exemple :

```
int X = 10, Y = 3;
char ch[] = "EXEMPLE";

   /* évaluation de la moyenne et affichage   */
   /* du résultat 3 chiffres après la virgule */
printf("moyenne de %d et %d = %.3f", X, Y, (float) X + Y /2);

   /* affichage de la chaîne EXEMPLE complétée    */
   /* de 3 espaces pour arriver à 10 caractères   */
   /* minimum                                     */
printf("%10s", ch);

   /* affichage d'un simple texte avec         */
   /* tabulation \t et retour à la ligne \n    */
printf("\tceci est uniquement du texte\n");
```

La fonction `printf` renvoie le nombre de données émises sur la sortie standard. En cas de problème elle renvoie -1 (`EOF`).

**Fonction scanf :** `int scanf( const char * format, [arg1, ..., argN])`

Lit des données sur l'entrée standard. Ces données seront interprétées suivant les spécifications fournies dans la chaîne `format` donnée en paramètre, et éventuellement stockées dans les variables représentées par `arg1, ..., argN`.

— *chaîne* `format` : il s'agit d'une chaîne de caractères directement donnée entre " " et qui précise comment le flot de données présent sur l'entrée standard doit être interprété. Elle est composée de groupes de caractères appelés spécificateurs de format.

— *spécificateur de format* : il définit le type et éventuellement la taille de la donnée à lire. Il débute par le caractère `%`, suivi éventuellement d'une valeur entière correspondant à la taille maximum de la donnée (= nombre de caractères s'il s'agit d'une chaîne ou nombre de chiffres s'il s'agit d'une donnée numérique) et se termine par un caractère indiquant en quel type la donnée doit être convertie (ex : `%c`, `%5d`, ...).

| Principaux caractères de conversion | Signification | Type correspondant |
|---|---|---|
| `c` | simple caractère | `char` |
| `s` | chaîne de caractères terminée par '\0' | `char *` (tableau de caractères) |
| `d` | entier décimal | `int` |
| `o` | entier sous forme octale | `unsigned int` |
| `x` | entier sous forme hexadécimale | `unsigned int` |
| `i` | entier sans distinction de forme | `int` |
| `h` | entier court | `short` |
| `u` | entier décimal non signé | `unsigned int` |
| `l` | entier long | `long` |
| `e, f` ou `g` | Nombre à virgule flottante | `float` |
| `lf` | Nbre à virg. flottante double précision | `double` |

Ainsi la donnée 1 peut être, selon le format utilisé, considérée comme un caractère (`%c '1'`), une chaîne de caractères (`%s "1"`), un entier (`%d 1`) ou un réel (`%f 1.0`).

— *arguments* : ils désignent l'endroit où la donnée lue va être stockée. Dans le cas d'une donnée simple (entier, réel ou caractère), il ne s'agit pas directement du nom de la variable qui va recevoir la valeur mais de son adresse (à voir plus tard avec l'utilisation de l'opérateur d'adressage `&`). Dans le cas de l'utilisation d'un tableau de caractères pour la lecture d'une chaîne de caractère (format `%s`) le nom du tableau suffit (cf. tableaux).

IMPORTANT : Le type de la donnée où doit être stockée la valeur lue doit être compatible avec celui du format associé. Il doit y avoir autant d'arguments que de spécificateurs de types.

Exemple :

```
int X, Y, N;
float Z;
char C, ch[20];

scanf("%d", &N);
   /* lecture d'un entier stocké dans la variable N : */
   /* &N = adresse mémoire de la variable N */

scanf("%d %f", &X, &Z);
   /* lecture d'un entier et d'un réel                 */
   /* respectivement stockés dans les variables X et Z */

scanf("%c%d%s", &C, &Y, CH);
   /* lecture d'un caractère suivi d'un entier puis   */
   /* d'une chaîne de caractères respectivement       */
   /* stockés dans les variables C et Y et dans le    */
   /* tableau de caractère de nom CH */

scanf("%c%c%c", &CH[0], &CH[1], &CH[2]);
   /* lecture de 3 caractères consécutifs             */
   /* respectivement stockés dans les première,       */
   /* deuxième et troisième cases du tableau de       */
   /* caractère CH. Ceci est différent de la lecture  */
   /* d'une chaîne de 3 caractères                    */
```

ATTENTION ! La chaîne `format` peut contenir éventuellement d'autres caractères que ceux composant les spécificateurs de format. Cela signifie alors que ceux-ci devront être trouvés tels quels dans le flot de données. S'il n'y a pas correspondance alors la lecture sera interrompue. Donc, ne pas mélanger le message destiné à informer l'utilisateur de ce qu'il doit faire et la saisie de la donnée demandée.

```
printf("Donner la valeur de N : ");
scanf("%d", &N);
```

et non

```
scanf("Donner la valeur de N : %d", &N);
```

La fonction `scanf` renvoie le nombre de données lues. S'il y a un problème de correspondance entre un format et une donnée, le nombre renvoyé est 0. Si le flot de données prend fin alors que d'autres données restent à lire la valeur renvoyée est -1 (`EOF`).

Des indicateurs de format peuvent modifier l'opération de lecture ou d'écriture. Ceux-ci sont alors placés après le caractère `%` débutant le spécificateur de type.

| Indicateur | Exemple | Action correspondante |
|---|---|---|
| `-` | `%-d` | l'écriture de la donnée est justifiée à gauche. |
| `*` | `%*s` | la donnée est lue mais pas mémorisée |
| `%` | `%%` | écriture du caractère `%` |
| `#` | `%#o` | les valeurs octales ou hexadécimales sont précédées de leurs marqueurs respectifs `0` et `0x` |

# 7 Premiers exercices

Ces exercices font appel à l'ensemble des concepts algorithmiques de base supposés acquis et aux concepts élémentaires du langage C présentés dans les sections précédentes. N'utiliser de tableau que si c'est demandé dans l'énoncé.
***Un conseil : au moins pour les premiers exercices, écrire d'abord l'algorithme en utilisant la syntaxe vue en cours, puis le traduire en C.***

**EXERCICE 1 :** Lire et comparer 2 entiers $A$ et $B$ quelconques et afficher un message annonçant le résultat de la comparaison :
$A$ est plus grand que $B$
ou $A$ et $B$ sont identiques
ou $A$ est plus petit que $B$

**EXERCICE 2 :** Lire 2 valeurs entières en les plaçant dans deux variables $x$ et $y$. Echanger les valeurs entre $x$ et $y$ puis afficher les valeurs de $x$ et $y$.

**EXERCICE 3 :** Lire 3 valeurs entières en les plaçant dans trois variables $x$, $y$ et $z$. Trier ces trois valeurs de telle sorte que, au final, la plus petite des 3 valeurs soit placée dans $x$, la plus grande dans $z$ et la valeur intermédiaire dans $y$.

**EXERCICE 4 :** Traduire la boucle `for` suivante en une boucle `while` :

```
for(((C = 0)&&(I = 1)); I > 0 ; I--) C+=I ;
```

Analyser le résultat obtenu et comparer-le avec celui fourni par la boucle :

```
for(((C = 0),(I = 1)); I > 0 ; I--) C+=I ;
```

**EXERCICE 5 :** Traduire la boucle `while` suivante en une boucle `for` :

```
X = 0 ;
Y = 0.25 ;
while((X == 0) || (Y != 2.)) Y = Y*2 ;
```

**EXERCICE 6 :** Calculer la factorielle de 10.

**EXERCICE 7 :** Déterminer le minimum et le maximum d'une série de $N$ nombres entiers quelconques (la valeur de $N$ sera fixée par l'utilisateur mais sera $>$ à 0).

**EXERCICE 8 :** Calculer la moyenne d'une série de $N$ nombres entiers positifs.

1. 1ère version : le nombre d'éléments à traiter est demandé à l'utilisateur,
2. 2ème version : la saisie s'arrête dès que la valeur -1 est lue.

**EXERCICE 9 :** Calculer la somme des $N$ premiers nombres impairs.

**EXERCICE 10 :** Nombres premiers

1. Déterminer si un nombre entier positif $N$ est ou non un nombre premier. Méthode : Un nombre est premier si et seulement si il n'admet aucun diviseur autre que 1 et lui-même. Autrement dit pour prouver que $N$ n'est pas premier, il suffit de trouver un nombre compris entre 2 et $N-1$ (ou entre 2 et $N/2$ si on veut optimiser) qui divise $N$.
2. Afficher les $N$ premiers nombres entiers qui ont la caractéristique d'être des nombres premiers. Exemple : si $N = 8$ alors 1 2 3 5 7 11 13 17 seront affichés.

**EXERCICE 11 :** Utilisation des tableaux
Trier une suite de $N$ nombres entiers ($N \leq 20$) suivant leur parité. Le programme affichera ensuite la liste des nombres pairs saisis, puis celle des nombres impairs.

**EXERCICE 12 :** Utilisation des tableaux
Considérer que le tableau $T$ d'entiers de taille $N$ est donné. Calculer la moyenne des valeurs dont l'indice est un multiple de 3.

**EXERCICE 13 :** Utilisation des tableaux
Considérer que le tableau $T_1$ d'entiers de taille $N$ est donné. Calculer le tableau $T_2$ de flottants de dimension $N$ tel que chaque case $i$ de $T_2$ contienne la moyenne des cases 0 à $i$ de $T_1$.

**EXERCICE 14 :** Utilisation des tableaux
Considérer que le tableau $T$ d'entiers de taille $N$ est donné. Renverser les valeurs dans $T$ (par exemple, au départ $T = \{3, 2, 5, 10, 8\}$ et à l'arrivée $T = \{8, 10, 5, 2, 3\}$). Proposer deux versions de votre programme : une utilisant un tableau auxilliaire et l'autre sans utiliser de tableau auxilliaire.

**EXERCICE 15 :** Comparaison de caractères (on pourra utiliser des tableaux)
Afficher la liste des lettres communes à 2 mots saisis par l'utilisateur. On considérera successivement les 2 cas suivants :

1. 1ère version : une lettre est commune à 2 mots si elle se trouve à la même position dans les 2. Exemple :

```
mot1 --> exemple
mot2 --> exercice
      lettre 1 : e
      lettre 2 : x
      lettre 3 : e
```

2. 2ème version : une lettre est commune à 2 mots si elle se trouve dans l'un et dans l'autre quelle que soit la position.
   Exemple :

```
mot1 --> exemple
mot2 --> complexe
      lettre(s) commune(s) : e x m p l
```

# Références


[Del92] Delannoy (Claude). – *Exercices en langage C.* – Eyrolles, 1992. ISBN : 2-212-08251-7.

[Del97] Delannoy (Claude). – *Initiation à la programmation.* – Eyrolles, 1997. ISBN : 2-212-08983-X.

[Dri90] Drix (Philippe). – *Langage C norme ANSI : vers une approche orientée objet.* – Masson, 1990. ISBN : 2-225-81912-2.

[Got93] Gottfried (Byron S.). – *Programmation en C : cours et problèmes.* – Ediscience international, Mc Graw Hill, 1993. ISBN : 2-7042-1230-9.

[RS94] Rigaud (Jean-Marie) et Sayah (Amal). – *Programmation en langage C.* – Cépaduès, 1994.

# Index pour le langage C

# V

# A Quelques bibliothèques et fonctions standards

Vous trouverez ici la description de quelques bibliothèques et fonctions standard couramment utilisées en langage C.

## A.1 Fichier stdio.h : Fonctions standards d'entrées-sorties

Quelques fonctions de cette bibliothèque ont été présentées dans la section 6 page 18.

## A.2 Fichier math.h : Fonctions mathématiques

Pour utiliser les fonctions mathématiques, il faut non seulement inclure ce fichier en début de programme mais également compiler avec l'option `-lm`.

`int abs(int)` Renvoie la valeur absolue de l'entier.

`int ceil(double)` Renvoie l'entier immédiatement supérieur.

`int floor(double)` Renvoie l'entier immédiatement inférieur.

`double cos(double)` Renvoie le cosinus.

`double sin(double)` Renvoie le sinus.

`double tan(double)` Renvoie la tangente.

`double exp(double)` Élève $e$ (nombre de Néper) à la puissance donnée en paramètre.

`double log(double)` Renvoie le logarithme népérien du nombre spécifié.

`double log10(double)` Renvoie le logarithme décimal du nombre spécifié.

`double pow(double, double)` Élève le premier nombre à la puissance indiquée par le second.

`double sqrt(double)` Renvoie la racine carrée.

## A.3 Fichier string.h : Manipulation de chaînes de caractères

Toute chaîne doit être terminée par le caractère nul '`\0`'.

`int strcmp(char*, char*)` Comparaison lexicographique de 2 chaînes de caractères, renvoie un nombre négatif si la première est avant la seconde, positif si la première est après la seconde et nul si les chaînes sont identiques.

`char* strcat(char*, char*)` Concatène la seconde chaîne à la chaîne première.

`char* strcpy(char*, char*)` Copie la chaîne `s2` dans la chaîne `s1`.

`int strlen(s)` Renvoie la longueur de la chaîne donnée en paramètre.

## A.4 Fichier ctype.h : Fonctions testant la nature d'un caractère

Elles renvoient une valeur non nulle si le test est positif, 0 sinon.

`int isalnum(char)` Teste si le caractère est un caractère alphanumérique.

`int isalpha(char)` Teste si le caractère est un caractère alphabétique.

`int isascii(char)` Teste si le caractère est un code ASCII.

`int isdigit(char)` Teste si le caractère est un chiffre décimal.

`int isodigit(char)` Teste si le caractère est un chiffre octal.

`int isxdigit(char)` Teste si le caractère est un chiffre hexadécimal.

`int islower(char)` Teste si le caractère est une lettre minuscule.

`int isupper(char)` Teste si le caractère est une lettre majuscule.

`int ispunct(char)` Teste si le caractère est un caractère de ponctuation.

`int isspace(char)` Teste si le caractère est un caractère d'espacement.

`int isprint(char)` Teste si le caractère est un caractère imprimable.

`int toascii(char)` Convertit le caractère en son code ASCII.

`int tolower(char)` Convertit la lettre en minuscule.

`int toupper(char)` Convertit la lettre en majuscule.

## A.5 Fichier stdlib.h

`void exit(unsigned int)` Termine le programme en fermant fichiers et buffers. Le code de retour du programme correspond à la valeur donnée en paramètre.

`int system(char*)` Transmet une commande au système d'exploitation. Si la commande décrite par la chaîne s'est exécutée sans problème, la valeur retournée est nulle sinon -1. Un sous-shell `sh` est lancé pour exécuter cette commande (voir le polycop sur le système UNIX).