Le langage C

Le premier programme C

```
main()
   {
printf("hello, world\n");
   }
```

À l'exécution :

```
hello, world
```

Pour arriver à l'exécution du programme précédent, il est nécessaire de :

— saisir le texte du programme dans l'ordinateur

— **traduire** le programme dans un format compréhensible par l'ordinateur

— démarrer le programme traduit.

Toute erreur dans la première étape interdira les étapes suivantes.

Langage C — Termes usuels

Le texte du programme est appelé **source** du programme.

Le processus de traduction du programme est appelé **compilation** et est effectué par un programme appelé **compilateur**.

Le compilateur est chargé de décomposer le source du programme suivant les règles du langage de programmation choisi, et de générer du « code machine » (suite de 0 et de 1) qui vont contrôler l'ordinateur afin qu'il suive les intentions du programmeur.

Le résultat de la traduction ou compilation est appelé **exécutable** ou version ***binaire*** du programme.

Le source du programme est généralement conservé dans un fichier texte dont le nom termine par « .c ».

Hello World (Again)

```
1                    main()
2                       {
3                  printf("hello, world\n");
4                       }
```

Ce programme consiste en une seule entité ou morceau de code exécutable appelé **fonction**.

Tout programme C **DOIT** inclure une fonction dont le nom est « **main** ».

L'exécution de tout programme commence par l'exécution de la fonction « main ».
Sans cette fonction le programme ne peut démarrer.

La fonction s'appelle « main » et non « Main » ou « MAIN », le C étant un langage sensible à la casse des caractères.

Sur l'exemple : *« main »* apparait sur la ligne 1, ce n'est pas essentiel car le programme commence son exécution à *« main »* et non à la première ligne du source.

Les parenthèses ouvrantes et fermantes « **()** » sont **importantes** car elles précisent au compilateur que lorsque l'on écrit *« main »* on introduit ou définit une fonction.

Cette ajout est nécessaire afin de faciliter le travail du compilateur (lui-même a été écrit par un humain…).

Hello World (Again) — suite

```
1                    main()
2                       {
3               printf("hello, world\n");
4                       }
```

Sur les lignes 2 et 4 se trouvent des **accolades** ouvrantes et fermantes : elles servent à définir les limites du « **corps** » de la fonction *« main »*.

Elles doivent être présentes par paire assorties.

La ligne 3 correspond au traitement effectif du programme :
le mot « *printf* » a un rapport avec l'action d'imprimer, donc d'afficher quelque chose à l'écran.

À l'époque de la création du C, l'imprimante comme périphérique de sortie était plus répandue que l'écran…

« *printf* » est ce que l'on appel une **fonction de bibliothéque**, cela veut dire que les instructions pour afficher sur l'écran sont ***copiées*** dans le programme à partir d'une bibliothèque de fonctions utiles et pré-compilées.

Le processus consistant à regrouper dans un même programme différents morceaux précompilés est appelé **relier**, « ***link*** », et est effectué par le **relieur** ou « ***linker*** ».

Le compilateur travaille en association avec le relieur afin de produire l'exécutable final.

Les programmeurs avancés peuvent créer leur propre bibliothèque de fonctions utiles.

Instruction et expression

```
3                   printf("hello, world\n");
```

La ligne 3 est une **instruction**.

Une instruction dit à l'ordinateur d'effectuer quelque chose.

*On dit que l'instruction est **exécutée**.*

Le traitement associé à cet exemple simple consiste à utiliser une fonction de bibliothèque pour faire quelque chose.

En C, l'utilisation simple d'une fonction (de bibliothèque ou non) est techniquement une **expression**, c-à-d. quelque chose qui *possède une valeur.*

*On dit alors que l'expression est **évaluée**.*

Pour convertir une expression en instruction un point-virgule doit suivre l'expression.

Hello World (Again) — suite

Une fonction C consiste en une séquence d'instructions qui sont exécutées dans l'ordre de leur écriture.

Chaque instruction doit être correctement écrite afin d'être accepter par le compilateur.

Nouvelle version de « Hello World » :

```
main()
   {
printf("hello, ");
 printf("world");
  printf("\n");
   }
```

Le but de *printf* est d'écrire des mots sur la **sortie standard** ou écran.

La sortie standard peut également désigné un fichier…

La « liste des choses à afficher » est comprise entre les parenthèses placées immédiatement après *printf*.

Les éléments placés entre ces parenthèses sont appelés **paramètres** ou **arguments**.

On dit que les paramètres sont passés ou transmis à la fonction.

Hello World (Again) — suite

Ici la fontion printf() reçoit un seul paramètre qui est une **chaîne de caractères** (des caractères compris entre une paire de guillemets anglais « “” »).

Le caractère ‘\n’ ne s’affiche pas à l’écran : il indique en fait qu’après son insertion un retour à la ligne dans l’affichage est effectué.

La paire de caractères \n est convertie par le compilateur en un seul caractère spécial qui fait partie de la chaîne.

D’autres caractères de ce type sont disponibles :

\a	émission d’un « *beep* »
\b	retour en arrière d’un caratère
\f	passage à la page suivante
\n	passsage à la ligne suivante
\r	retour à la ligne
\t	tabulation
\v	tabulation vertical
\\	« *backslash* »
\’	apostrophe
\”	guillemet
\?	point d’interrogation.

Présentation du source

Le C n'est pas sensible à l'apparence du programme :

```
main(){        printf("hello, world\n");}
```

est équivalent à :

```
main
  (
  )
  {
printf
  (
"hello, world\n"
  )
  ;
  }
```

Les deux versions précédentes ne sont ni l'une ni l'autre recommandées.

```
/*  Premier programme en C */

main()
{
    printf("hello, world\n");
}
```

Il est important de commenter les différentes parties d'un programme afin d'en **documenter** le principe et/ou le but :

```
main()
{
    printf("hello world\n");

    /* du code tres intelligent
    que je n'ai pas encore eu le temps
    d'ecrire */

    printf("la solution au probleme de la vie, de
l\'univers et du reste est :\n");

    /* Du code encore plus intelligent qui reste
encore a developper */
}
```

Des erreurs de programmation

```
MAIN()
{
    printf("hello, world\n");
}
```

oubli que le C est sensible à la casse

```
ld: Undefined symbol _main
Compilation failed
```

```
main()
{
    printf('hello, world\n');
}
```

Utilisation d'apostrophes au lieu de guillemets

```
programme.c, line 3: too many characters
in character constant
Compilation failed
```

```
main()
{
    printf("hello, world\n")
}
```

Oubli de l'ajout du point-virgule après le printf()

```
programme.c, line 4: syntax error at or
near symbol }
Compilation failed
```

Ou bien

```
Error programme.c : statement missing ; in
    function main
```

```
main()
{
    print("hello, world\n");
}
```

Nom de fonction mal orthographié

```
ld: Undefined symbol _print
Compilation failed
```

```
main
{
    printf("hello, world\n");
}
```

Oubli de la paire de parenthèses après « main »

```
programme.c, line 2: syntax error
```

Compilation failed

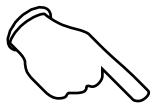
Programmer avec des entiers

Tout les ordinateurs travaillent de façon interne avec des nombres. Tous les éléments familiers comme les chaînes de caractères, les sons les images… sont en fait représentés à l'intérieur de l'ordinateur par des nombres.

Un programme simple utilisant des nombres :

Lecture de deux nombres, stockage de ces nombres et affichage de leur somme.

```
1    /*  Un programme pour lire deux nombres
2        et afficher leur somme  */
3    main()
4    {
5        int x, y; /* lieu pour stocker les nombres
6                                                  */
7        printf("Entrez x ");
8        scanf("%d", &x);
9        printf("Entrez y ");
10       scanf("%d", &y);
11       printf("la somme de x et y vaut %d\n",x+y);
12   }
```



```
Entrez x 4
Entrez y 5
la somme de x et y vaut 9
```

Le code de la ligne 5 **réserve** et **nomme** des emplacements pour stocker les nombres.

La fonction de bibliothèque *scanf()* utilisée en ligne 8 et 9 effectue une **lecture** depuis le clavier.

L'addition des deux nombres stockés se fait à la ligne 11 en calculant la valeur de x+y.

?!? « **%** » et « **&** » ligne 8, 10 et 11.

Stocker des nombres

La première tâche à accomplir lorsque l'on écrit un programme qui manipule des nombres est de décider **où** les stocker et d'en **avertir** l'ordinateur.

Les nombres sont stockés dans la mémoire de l'ordinateur.

Les emplacements de mémoire sont connus de l'ordinateur par le numéro de référence ou **adresse**.

Le programmeur (un humain…) préfére **nommer** l'emplacement ce qui permet de s'en souvenir facilement.

C'est la tâche du compilateur de gérer la **correspondance** entre le nom donné à un emplacement mémoire par le programmeur et son adresse à l'intérieur de l'ordinateur.

L'adresse d'un emplacement n'a que peu d'interêt pour le programmeur.

Ce qui est important à retenir au sujet de la mémoire de l'ordianteur est simplement que les nombres une fois stockés dans un emplacement de cette mémoire restent jusqu'à ce que le programme finisse, ou bien que d'autres nombres soient stockés dans le même emplacement, ou alors que l'ordinateur soit éteint.

Déclaration

La partie du programme qui dit au compilateur quel emplacement mémoire est requis et doit être referencé par un certain nom est appelé **déclaration**.

La forme la plus simple de déclaration consiste en un mot décrivant le type de mémoire qui est requis suivi du nom qui sera utilisé pour référencer cet emplacement :

```
int x, y;
```

Ici, deux emplacements sont réservés du type « *int* », qui seront référencés comme « x » et « y ».

L'emplacement mémoire de type « int *» permet de stocker un entier.*

Ces emplacements mémoire sont souvent appelés **variables** car il est toujours possible de changé le nombre qui y est stocké.

Le nom associé à l'emplacement mémoire est appelé **identificateur**.

On dira « la variable x », ce qui n'est pas tout à fait juste car il vaut mieux dire « le nombre stocké à l'emplacement dont l'adresse est x ».

*Toute variable utilisée dans le programme **DOIT** être déclarée.*

Toute déclaration de variable doit apparaitre avant la première instruction.

Entrée des nombres

La fonction de bibliothèque **scanf()** est chargée d'effectuer différentes tâches :

— Déterminer quels touches ont été appuyées

— Traduire le nombre saisi dans la représentation interne de l'ordinateur

— Stocker la valeur dans l'emplacement mémoire choisi

*Dans la plupart des programmes **scanf()** est utilisé pour lire depuis le clavier.*

*En fait **scanf()** lit depuis « **l'entrée standard** »qui est habituellement le clavier mais peut-être un **flux de donnée** en provenance d'un **fichier**.*

```
7       printf("Entrez x ");
8       scanf("%d", &x);
9       printf("Entrez y ");
10      scanf("%d", &y);
```

Chaque appel à la fonction **scanf()** est fait avec **deux arguments** séparés par une virgule :

argument 1 : **`"%d"`** chaîne de caractères définissant les **règles** à appliquer pour la **conversion** des touches que l'utilisateur à appuyée en une **représentation interne à l'ordinateur**.

Ces règles sont nécessaires car quand un nombre tel que « 27 » est tapé au clavier, deux caractères '2' et '7' sont transmis à l'ordinateur, il faut donc les regrouper pour former le nombre 27 dans la représentation binaire de l'ordinateur.

`"%d"` Le '**%**' indique que le caractère suivant '**d**' correspond à une commande de conversion.

argument 2 : **`&x`** c'est l'**adresse de l'emplacement** mémoire où stocké le nombre lu et converti.

*L'esperluette '**&**' ne **DOIT PAS** être omise !*

Sortie des nombres

Utilisation de la fonction de biliothèque **printf()**.

```
11      printf("la somme de x et y vaut %d\n",x+y);
```

Les paramètres associés à **printf()** (comme à **scanf()**):

argument 1 :

***Toujours** une chaîne de caractère comme premier argument.*

Cette chaîne est une **chaîne de format** ou **combinaison d'éléments** à copier directement sur la **sortie standard** et de **spécifications de conversion**.

argument 2 :

C'est une **expression** dont la valeur est **convertie** d'une **représentation interne** à l'ordinateur vers une **représentation externe** conformément à la **spécification de conversion** indiquée dans la chaîne de format.

*Différents types de conversion existent, elles **sont toujours** notées par un caractère '**%**' suivi d'un ou plusieurs caractères.*

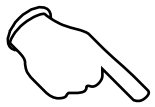
La spécification de conversion '**%d**' indique une conversion de la représentation interne de l'ordinateur vers une **valeur entière**.

La valeur de « **x+y »** est la somme des valeurs stockées dans les emplacements **x** et **y**. Techniquement, c'est est une expression.

*Quelque soit l'endroit où apparaît une **expression** dans un programme, l'effet produit est **toujours** le même :*
*L'ordinateur calcul la valeur de l'expression et **remplace l'expression** par cette **valeur**.*

Lire deux nombres

```
/* Un autre programme de somme de deux nombres */


main()
{
    int n1;
    int n2;

    printf("Entrez deux nombres ");
    scanf("%d%d", &n1, &n2);
    printf("La somme de %d et de %d est %d\n", n1,
            n2, n1+n2);
}
```



```
Entrez deux nombres 11 4
La somme de 11 et de 4 est 15

Entrez deux nombres 44 -10
La somme de 44 et de -10 est 34
```

*Bien que la chaîne de format en argument de la fonction **scanf()** soit "**%d%d**" suggére que les nombres soient concaténés, la fonction va lire deux nombres éventuellement séparés par un nombre arbitraire d'espaces, de tabulations ou de retour à la ligne.*

De même la saisi de nombre négatif est autorisé sans spécification supplémentaire.

Valeurs Initiales des variables

*Quelles sont les valeurs stockées dans les variables « **x** » et « **y** » ?*

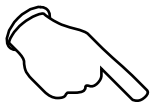
La réponse est **n'importe quelle valeur** !

On ne peut être sûr de la valeur initialement stockée dans un emplacement mémoire…

Exemple :

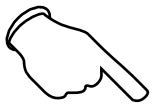
```
main()
{
    int x;
    int y;

    printf("Valeur initiale de x est %d\n", x);
    printf("Valeur initiale de y est %d\n", y);
}
```

Sur un IBM 6150 :



```
Valeur initiale de x est 0
Valeur initiale de x est 0
```

Sur une Station Sparc



```
Valeur initiale de x est 0
Valeur initiale de x est 32
```

Sur un compatible IBM PC

```
Valeur initiale de x est 0
Valeur initiale de x est 248
```

*Il est **important** d'initialiser la **valeur initiale** d'une variable dés **sa déclaration**.*

Valeurs Initiales des variables (suite)

Initialisation et **déclaration** peuvent être **combinées.**

```
main()
{
    int x = 3, y = 4;

    printf("la somme de %d et de %d est %d\n", x,
            y, x+y);
}
```

```
la somme de 3 et de 4 est 7
```

Une variable peut être initialisée par n'importe quelle expression dont la valeur est déterminée par le comilateur.

```
int x = 4 + 7;
int z = 3, y = z + 6;
```

*Il est **vivement conseillé** d'initialiser toute variable lors de sa déclaration.*

Conversion représentation interne vers représentation externe

Il est possible d'utiliser des **codes** supplémentaires lors d'une spécification de conversion, afin de **contrôler** cette conversion.

Quand une valeur est convertie de sa représentation interne par la fonction **printf()** l'ensemble des positions occupées par la forme externe est appelée un **champs de sortie**.

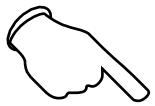
Le nombre de caractères composant ce champs est appelé la **taille** du champs de sortie.

La spécification de conversion '**%d**' définit un champs de sortie dont la taille est toujours suffisante pour représenter le nombre de chiffres et éventuellement un signe négatif.

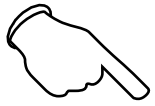
En introduisant un nombre entre le caractère '**%**' et le caractère '**d**', la taille du champs de sortie peut être définie.

*La taille du champs de sortie ne peut **jamais** être inférieure à la taille spécifiée.*

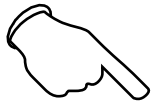
```
main()
{
    int i = 0, j = 0;
    printf("Entrez des nombres ");
    scanf("%d%d", &i, &j);
    printf("La somme est >>%3d<<\n", i + j);
}
```



```
Entrez des nombres 3 4
La somme est >>  7<<
```



```
Entrez des nombres 120 240
La somme est >>360<<
```



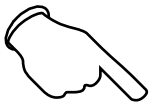
```
Entrez des nombres 999 999
La somme est >>1998<<
```

Gestion des erreurs de saisie

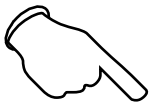
L'erreur est humaine…

Il est toujours possible d'appuyer sur la mauvaise touche :

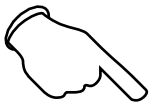
La fonction **scanf()** a un comportement précis en présence d'une saisie erronée :



```
Entrez x 3
Entrez y deux
La somme de x et de y est 3
```



```
Entrez x un
Entrez y La somme de x et de y est 0
```



```
Entrez x 1.25
Entrez y La somme de x et de y est 1
```

Exemple 1 : La fonction **scanf()** est incapable de lire la seconde entrée.

La valeur de **y** est sa valeur d'initialisation soit **0**.

Exemple 2 : Quand la fonction **scanf()** échoue elle laisse les caractères saisies.

Le second **scanf()** trouve ses caractères et échoue également, sans pouvoir recevoir d'éventuels nouveaux caractères saisis par l'utilisateur.

Exemple 3 : La conversion d'un nombre entier échoue également car la fonction **scanf()** ne s'attend pas à trouver un séparateur de décimal…

Des erreurs aux résultats imprévisibles

```
/* Un programme de lecture de deux nombres et
d'affichage de leur somme */

main()
{
    int x = 0, y = 0;

    printf("Entrez x ");
    scanf("%d", x);
    printf("Entrez y ");
    scanf("%d", y);
    printf("La somme de x et y est %d\n", x + y);
}
```

Un *plantage* de la machine est attendu !

```
Entrez x 3
Memory fault – core dumped
```

Pourquoi ?

La valeur du deuxième argument de la fonction **scanf() doit être** l'adresse de l'emplacment mémoire où le nombre doit être stocké après conversion.

En C, l'adresse d'un emplacement mémoire est **toujours** obtenu en faisant **précéder le nom de l'emplacement mémoire** par **l'esperluette** ('**&**').

Si l'esperluette est omise, la valeur transmise à la fonction **scanf() n'est pas l'adresse** de « **x** » mais la **valeur stockée** dans « **x** ».

La fonction **scanf()** interpréte cette valeur comme une adresse et essaye de stocker dans cette emplacement mémoire le nombre converti.

Des erreurs aux résultats imprévisibles (suite)

```
/* Un programme de lecture de deux nombres et
d'affichage de leur somme */

main()
{
    int x = 0, y = 0;

    printf("Entrez x ");
    scanf("%d", &x);
    printf("Entrez y ");
    scanf("%d", &y);
    printf("La somme de x et y est %d\n");
                        }
```

```
Entrez x 3
Entrez y 4
La somme de x et y est -134218272
```

Aucun message d'erreur n'est généré par le compilateur, car il n'y a pas d'erreur avec le dernier **printf()**.

Le compilateur **ne vérifie pas** que les arguments transmis à la fonction **printf() correspondent aux spécifications de conversion** inclus au niveau de la chaîne de format.

La fonction **printf()** a déterminé d'après la chaîne de format qu'elle a besoin d'un « **int** », et prend la valeur se trouvant à l'endroit courant de l'exécution du programme, malheureusement i**l n'y a aucune chance** pour que la bonne valeur soit à cet endroit…

Liste des « mots-clés » du language C

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

Certains compilateurs ajoutent des mots-clés supplémentaires.

Arithmétique

Une expression est construite à partir d'un certain nombre d'opérateurs :

`+ - * / % (`*`modulo`*`)`

Les nombres associés à ces opérateurs peuvent être :

1. des nombres appelés **constantes**
2. le nom d'une variable auquel cas la valeur contenue dans cette variable est utilisée
3. une expression auquel cas la valeur de l'expression est utilisée

Exemples :

1. **`x + 23`**
2. **`4500`**
3. **`x + y * 11`**
4. **`z`**

La valeur d'une expression composée d'entiers est elle-même un entier.

La division de deux entiers est un entier (la partie derrière la virgule est supprimée).

La division de deux entiers est tronquée ou arrondie vers zéro.

20 / 7 donne 2 et non 2.85714… ni 3.

*Attention à la **division par zéro**, dont le traitement dépend des compilateurs.*

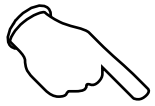
Attention à la priorité des opérateurs :
x - y + z ≠ x - (y + z)

D’autres opérateurs

L’opérateur d’affectation ‘**=**’

L’expression **x = 7** a pour valeur 7 et comme effet secondaire d’affecter cette valeur 7 à **x**.

```
main()
{
    int x = 4;

    printf("La valeur de x est %d\n", x);
    printf("La valeur de x = 8 est %d\n", x = 8);
    printf("La valeur de x est %d\n", x);
}
```



```
La valeur de x est 4
La valeur de x = 8 est 8
La valeur de x est 8
```

L’incrémentation :

x = x + 1

Le caractère « **x** »à gauche de l’opérateur **=** désigne l’emplacemet mémoire alors que le caractère « **x** » à droite désigne la valeurstockée à l’emplacement mémoire d’adresse « **x** ».

On peut remarquer que le traitement est
*« évaluer l’expression **x + 1**, en stocker la valeur dans **x** »*

et non
*« évaluer l’expression **x = x**, additionner 1 au résultat »*

*On dira que la **priorité** de l’opérateur ‘**+**’ est **supérieure** à celle de l’opérateur ‘**=**’.*

L’ opérateur ‘**++**’ : **x ++** équivaut à **x = x+ 1.**

L’ opérateur ‘**--**’ : **x --** équivaut à **x = x- 1.**

Les différents types de données

D'autres types que les entiers sont disponibles.

À chacun de ces types est associé

— un moyen de les saisir,

— de les afficher

— d'évaluer les expressions qui les contiennent.

Ces types de données **correspondent à différentes façons d'interpréter** les « 0 » et « 1 » stockés dans la mémoire de l'ordinateur.

0	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**…**

Il existe **3 types de données de base** :

— l'**entier**

— le **nombre à virgule flottante** qui permettent de stocker soit des très grand nombres, soit de très petits.

Il existe différentes manières de stocker des nombres à virgules, chacune de ces manières offrant une plus ou moins grande précision.

— le **caractère** qui correspond au moyen de stocker et de manipuler les symboles qui sont affichés ou bien saisies au clavier.

Les nombres à virgules flottante

La précision des nombres à virgules flottantes dépend du compilateur et de la machine.

Ci-dessous un tableau des minimums admis :

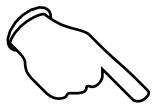
Type	Valeur maximale	Chiffres significatifs
float	**1.0 E37**	**6**
double	**1.0 E37**	**10**
long double	**1.0 E37**	**10**

Ces nombres à virgule flottante peuvent être convertis en une représentation externe à l'aide des spécifications de conversion suivantes :

f	**float**
lf	**double**
Lf	**long double**

La taille du champs de sortie peut être définie de manière à indiquer le nombre total de chiffres et le nombre de chiffres après la virgule :

```
main()
{
    double x = 213.5671435;
    double y = 0.000007234;
    printf("x = %10.5lf\n", x);
    printf("y = %10.5lf\n", y);
    printf("x = %5.2lf\n", x);
    printf("y = %10lf\n", y);
    printf("x = %3.1lf\n", x);
}
```



```
x =  213.56714
y =    0.00001
x = 213.57
y =   0.000007
```

x = 213.6

Utilisation des nombres à virgule flottante

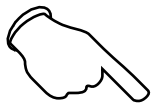
Toutes les opérations arithmétiques peuvent être employées avec des flottants à **l'exception du modulo** ('%').

Il n'existe pas d'opérateurs supplémentaires pour le traitement des flottants.

L'utilisation de l'opérateur de division donne un flottant (pas de troncage de la valeur).

La lecture et l'écriture des flottants se fait de la même manière que pour les entiers, à l'aide des **spécificateurs de conversion adaptés.**

```
main()
{
    printf("Le nombre est %10.5lf\n", 2445);
    printf("Le nombre est %10.5lf\n", -2445);
}
```



```
Le nombre est     0.00000
Le nombre est     -NaN
```

1er cas : une erreur de conversion est faite à partir de la représentation binaire de la valeur.

2e cas : la représentation binaire associée à un entier est incompatible avec celle d'un flottant.

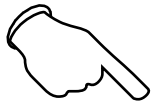
Le traitement du deuxième cas dépend de la machine, dans le meilleur des cas une indication "Not a Number" est affiché.

La définition de constante de type flottant **doit être suivi** d'une virgule même dans le cas d'une valeur entière.

```
3.0     4.7896
```

Une conversion automatique, *et correcte*, permet d'automatiquement transformer un élément de type "**float**" en élément de type "**double**".

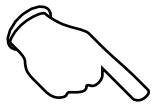
```
float x = 4.5;
printf("x sous forme de double %lf\n", x);
```



```
x sous forme de double 4.5
```

Erreurs dues à la précision limitée

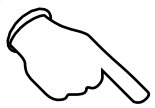
```
main()
{
    double x = 50000.0;
    x += 0.01;
    printf("x = %30.15lf\n", x);
    x += 0.0001;
    printf("x = %30.15lf\n", x);
    x += 0.000001;
    printf("x = %30.15lf\n", x);
    x += 0.00000001;
    printf("x = %30.15lf\n", x);
    x += 0.0000000001;
    printf("x = %30.15lf\n", x);
    x += 0.000000000001;
    printf("x = %30.15lf\n", x);
}
```



```
x =            50000.010000000002037
x =            50000.010099999999511
x =            50000.010100999999850
x =            50000.010101009997015
x =            50000.010101010098879
x =            50000.010101010098879
```

La précision est de 15 chiffres significatifs.

```
main()
{
    float x = 50000.0;
    x += 0.01;
    printf("x = %30.15f\n", x);

    /* ... */

    x += 0.000000000001;
    printf("x = %30.15lf\n", x);
}
```



```
x =            50000.011718750000000
x =            50000.011718750000000
x =            50000.011718750000000
x =            50000.011718750000000
x =            50000.011718750000000
x =            50000.011718750000000
```

Types entiers

```
short int
int
long int
```

Ces différents types correspondent à la **taille de la représentation binaire** que l'on veut associer à la variable.

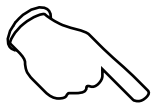
*L'utilisation du mot-clé **unsigned** permet de spécifier l'utilisation de valeurs **uniquement positives**, et d'**augmenter la valeur** la plus grande pouvant être stockée.*

Un "**short int**" est souvent représenté sur 16 bits et un "**long int**" sur 32 bits.

*Sur PC, un "**int**" est équivalent à un "**short int**".*
*Sur Unix, un "**int**" est équivalent à un "**long int**".*

Attention au dépassement de capacité !

```
main()
{
    int x = 30000;
    int y = 0;

    y = x + x;
    printf("le double de x vaut %d\", y);
}
```



```
(sous Unix)  le double de x vaut 60000
(sur PC)     le double de x vaut -5536
```

Le nombre 60000 est simplement trop grand pour être stocké dans une représentation 16 bits.

*On remarque que -5536 = 60000 - 2 * 32768*

Manipulation de la représentation binaire

Opérateurs binaires :

|	ou
&	et
^	ou exclusif
>>	décalage à droite
<<	décalage à gauche

Tables de vérités des opérateurs :

1	&	1	*donne*	1
1	&	0	*donne*	0
0	&	0	*donne*	0
0	&	1	*donne*	0
1	|	1	*donne*	1
1	|	0	*donne*	1
0	|	0	*donne*	0
0	|	1	*donne*	1
1	^	1	*donne*	0
1	^	0	*donne*	1
0	^	0	*donne*	0
0	^	1	*donne*	1

Opérateur de décalage à gauche :

11011101 << 1	*donne*	10111010
11011101 << 3	*donne*	11101000

Opérateur de décalage à droite :

11011101 >> 1	*donne*	01011101
11011101 >> 3	*donne*	00010111

Le style caractère

char

Il correspond **toujours** à un octet ou 8 bits.

L'écriture d'une constante de type caractère se fait en l'insérant entre deux guillemets simples **'**.

'a''%''g' …

La saisie et l'affichage d'un caractère est produit par le spécificateur de format '**%c**'.

Un caractère peut être **automatiquement converti en entier** :

```
printf("la taille de l'alphabet est %d", 'Z'-'A');
```

*L'association d'un caractère à une valeur numérique (comprise entre 0 et 255) se fait suivant le **code ASCII**.*
('a' est associé à la valeur 65).

*Les caractères entre **0 et 127 sont normalisés** (on parlera de codage sur 7 bits), ceux entre **128 et 256 ne le sont pas** (sur PC on parlera de pages de codes, par exemple celle codant les caractères accentués français).*

La conversion minuscule vers majuscule :

```
main()
{
    char caractere = 'a';
    char caractere_converti = 'A';

    printf("Entrez un caractere ");
    scanf("%c", &caractere);
    caractere_converti = caractere + ('A' - 'a');
    printf("\nLe caractere en majuscule est %c\n",
            caractere_converti);
}
```

Les séquences de contrôles : '\n', '\0', '\t', … sont traités comme un caractère unique.

Mélanges des types

Au sein d'une même expression, on peut mélanger les différents types de données.

Le C disposant de 45 opérateurs et de 12 types de données différents il existe 24000 combinaisons possibles.

Certains langages **interdisent le mélange** de données de différents types et fournissent des **fonctions de conversion de type**.

Ce sont des langages dits **fortement typés**.

Le C **n'est pas** un langage fortement typé.

Le C dispose de fonctions de bibliothèque pour effectuer certaines conversions de type :

atoi() pour convertir un nombre sous forme de chaîne de caractères en une valeur entière.

atof() pour convertir une chaîne de caractère en une valeur flottante.

…

Le C dispose de conversions automatiques d'un type à l'autre.

*Certaines de ces conversions **dépendent de la taille de la représentation binaire** associée au type de départ et de celle associée au type de destination.*

Préservation de la valeur numérique : conversion d'un nombre signé ou non signé d'une représentation binaire plus petite vers une représentation binaire plus grande :

short int vers **long int**.

Troncage de la représentation binaire : conversion directe d'un type de départ de représentation binaire supérieur à celle du type d'arrivée.

int vers **char**.

Conservation de la représentation binaire : conversion d'un type singé vers un type non-signé (les valeurs numériques **ne sont pas conservées**).

unsigned int vers **int**.

Troncage à la virgule : conversion d'un type flottant vers un type entier.

float vers int.

Résultat d'une expression mélangeant différents type de valeurs

```
main()
{
    double x,y;

    x = 1 + 2 / 3;
    y = 1 + 2 / 3.0;
    printf("x = %5lf, y = %5lf\n", x, y);
}
```

```
x = 1.000000, y = 1.666667
```

La priorité de l'opérateur **/** est supérieure à celle de l'opérateur **+**, la division est faite en premier :

2 / 3 les deux opérandes sont entières pas de conversion, la division réalisée est **entière** et le résultat est **0**.

2 / 3.0 la division est **flottante** le résultat est **0,66666**…
La somme est appliquée sur des opérandes de type entier et flottant, alors l'opérande **entière est convertie** en opérande **flottante**.

Opérateur de conversion de type : le **cast**.

```
x = 1 + 2 / (double) 3;
```

La valeur entière 3 est convertie (*correctement*) en la valeur flottante **3.0**, et le résultat du calcul est correcte.

Évitez de mélanger différents types de données dans une même expression.

Dans le cas d'un mélange évitez de confier la conversion des données au C…
Utilisez le cast !

Les instructions conditionnelles

Les programmes sont des **suites d'instructions** à réaliser les unes après les autres.

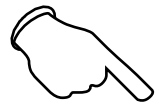
Il est possible d'écrire des programmes qui font des **traitements différents** en fonction de **conditions** détectées pendant l'exécution du programme.

Opérateurs relationnels

- **>** supérieur à
- **>=** supéieur ou égal à
- **<=** inférieur ou égal à
- **<** inférieur à

En C, il n'existe pas de type booléen, les entiers 0 et 1 sont utilisés en remplacement.

```
main()
{
    int x = 3, y = 4;

    printf("La valeur de \"%d>%d\" est %d\n", x,
y, x>y);
    printf("La valeur de \"%d>=%d\" est %d\n", x,
y, x>=y);
    printf("La valeur de \"%d<=%d\" est %d\n", x,
y, x<=y);
    printf("La valeur de \"%d<%d\" est %d\n", x,
y, x<y);

}
```



```
La valeur de "3>4" est 0
La valeur de "3>=4" est 0
La valeur de "3<=4" est 1
La valeur de "3<=4" est 1
```

Opérateurs d'égalités :

- **==** est égal à
- **!=** est différent de

Attention à ne pas confondre l'opérateur d'affectation = et l'opérateur d'égalité ==.

Les instructions conditionnelles (suite)

Les opérateurs de **combinaison** d'expression conditionnelles (*où des opérateurs relationnels apparaissent*).

&& le **ET** à ne pas confondre avec l'opérateur **&** (*ET bit à bit*)

|| le **OU** à ne pas confondre avec l'opérateur **|** (OU bit à bit)

! le **NON**.

*En C, seule valeur **0** est considérée comme **faux**, toute valeur différente de 0 est considérée comme **vrai**.*

Op1	Op2	Op1 **||** Op2	Op1 **&&** Op2	**!** Op1
0	0	0	0	1
0	non-zéro	1	0	1
non-zéro	0	1	0	0
non-zéro	non-zéro	1	1	0

```
main()
{
    int x = 1, y = 2, z = 3, p, q, r;

    p = (x > y) && (z < y);
    printf("p = %d\n", p);
    q = (y>x) || (y > z);
    printf("q = %d\n", q);
    printf("%d && %d = %d\n", p, q, p&&q);
    printf("%d || %d = %d\n", p, q, p||q);
    printf("! %d = %d\n", p, !p);
}
```

```
p = 0
q = 1
0 && 1 = 0
0 || 1 = 1
! 0 = 1
```

Combinaison d'expression conditionnelle

```
main()
{
    printf("La valeur est %d\n", 9>8>7);
}
```

```
La valeur est 0
```

La relation est fausse !

L'évaluation de l'expression commence par évaluer **9 > 8** ce qui donne la valeur **1**.

Puis il y a évaluation de l'expression **1 > 7**, dont le résultat est faux.

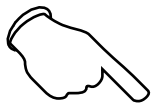
Il faut écrire : **b > x && x > a**
et non **b > x > a.**

Attention à la priorité des opérateurs :

```
test = x == 0.0 || 1.0 / x < 0.75
```

Les opérateurs arithmétiques ont une priorité supérieure à celle des opérateurs relationnels.

```
test = x == 0.0 || (1.0 / x) < 0.75
```



```
x = 1.5 test vaut 1
x = 0.75 test vaut 0
x = 0 test vaut 1
```

Les instructions conditionnelles

if (*expression***)** *instruction*
si l'*expression* à une valeure non nulle alors l'*instruction* est exécutée.

if (*expression***)** *instruction1*
else *instruction2*
si l'*expression* à une valeure non nulle alors l'*instruction1* est exécutée, sinon l'*instruction2* est exécutée.

```
main()
{
    int x;
    printf("Entrez x ");
    scanf("%d", &x);
    if (x%2) printf("%d est impair\n", x);
        else printf("%d est pair\n", x);
}

main()
{
    int x, y, z;
    printf("Entrez x y et z");
    scanf("%d%d%d", &x, &y, &z);
    if ((x < y && y < z)
        || (x > y && y > z))
        printf("%d est compris entre %d et %d", y,
x, z);
    else printf("%d n'est pas compris entre %d et
%d", y, x, z);
}
```

Si l'on désire exécuter plus d'une instruction dans la même branche de la conditionnelle :

```
main()
{
    int x;
    printf("Entrez x ");
    scanf("%d", &x);
    if (x%3)
        {
            printf("le nombre %d", x);
            printf("est divisible par 3\n");
        }
}
```

Composition d'instructions conditionnelles

```
main()
{
    double x;

    printf("Entrez un nombre ");
    scanf("%lf", &x);
    if (x > 0.0)
        {
            printf("c'est un nombre positif ");
            printf("son inverse est %10.5lf\n",
                    1.0 / x);
        }
    else
        {
            if (x == 0.0)
                printf("Zero pas d'inverse\n");
            else
                {
                    printf("c'est un nombre negatif
");
                    printf("son   inverse   est
%10.5lf\n", 1.0 / x);
                }
        }
}
```

Utilisez des tabulations pour présenter les alternatives de la conditionnelle,

*Associez toujours un **bloc** à une branche contenant **plus d'une instruction** à réaliser, **mais également si l'instruction est une conditionnelle**.*

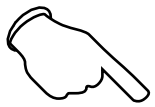
```
if (condition)    {
                      instruction;
                      instruction;
                  }
else              {
                      instruction;
                      instruction;
                  }
```

Les boucles

La boucle « Tant que »

while (*expression***)** *instruction*

```
main()
{
    int i = 1;

    while (i < 10)
        {
            printf("%d %d\n", i, i*i);
            i++;
        }
}
```



```
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
```

```
main()
{
    int i = -1;

    while (i < 0)
        {
            printf("Entrez une valeur positive");
            scanf("%d", &i);
        }
}
```

Ou plus simplement :

```
main()
{
    int i;

    do
        {
            printf("Entrez une valeur positive");
            scanf("%d", &i);
        } while (i < 0);
}
```

La boucle « for »

C'est une alternative à l'usage du « **while** » et du « **do…while** ».

for (*expression1* **;** *expression2* **;** *expression3* **)** *instruction*

1 — *expression1* est évaluée.

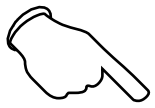
2 — si *expression2* a une valeur **différente de zéro**, **alors** *instruction* est exécutée **et** *expression3* est évaluée.

Contrairement à d'autres languages : il n'y a **pas de restriction** sur la nature des 3 expresssions.

Le « **for** » est équivalent à :

```
expression1;
while(expression2)
{
    instruction
    expression3;
}
```

```
main()
{
    int i;

    for (i=0; i<4; i++)
        printf("%d %2d\n", i, i*i);
}
```



```
0       0
1       1
2       4
3       9
```

La boucle « for » (suite)

Les différentes expressions peuvent être omises :

for (; ;) instruction

donne une boucle infinie.

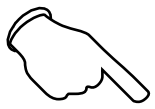
De même l'instruction peut être omise :

for (expression1; expression 2; expression 3) ;

*Attention à **ne pas insérer** sans le vouloir un '**;**' après un **for** !*

Il est également possible d'avoir plus d'une expression, chacune de ces expressions étant séparées par une virgule :

```
#include <math.h>

main()
{
    int i;
    double x;
    for (i = 0, x = 0; i < 4; i++, x += 0.5)
        printf("%d %10.7lf\n", i, sqrt(x));
}
```



```
0   0.0000000
1   0.7071068
2   1.0000000
3   1.2247449
```

Le choix multiple

C'est une alternative à l'usage de plusieurs instructions **if**.

switch (*expression entière* **)** *instruction*

L'*instruction* est composée d'une séquence d'instructions « **case** ».

case *expression-entière-constante* **:** *instruction*

L'exécution **se produit** sur le « **case** » dont la valeur associée correspond à la valeur de l'expression du « **switch** ».

L'exécution **continue jusqu'à la fin** du corps du « **switch** » ou bien jusqu'à la rencontre d'un « **break** ».

Si aucune valeur associée à un « **case** » ne correspond à la valeur du « **switch** » alors aucun partie du corps du « **switch** » n'est exécutée, à moins de définitir un traitement par défaut (« **default** »).

```
#include <stdio.h>

main()
{
    int n1, n2;
    char c;
    printf("Entrez x, y et l'operateur ");
    scanf("%d%d", &x, &y);
    scanf("%c", &c);
    switch(c)
        {
            case '+' :
                printf("%d\n", n1 + n2);
                break;
            case '-' :
                printf("%d\n", n1 - n2);
                break;
            case '*' :
                printf("%d\n", n1 * n2);
                break;
            case '/' :
                printf("%d\n", n1 / n2);
                break;
            default :
            printf("Operateur inconnu %c\n",c);
        }
}
```

Le choix multiple (suite)

```
#include <stdio.h>

main()
{
    int c;
    int nbre_chiffres = 0;
    int nbre_espaces = 0;
    int autres = 0;

    while ((c = getchar()) != EOF)
        switch(c)
            {
                case '0' : case '1' : case '2' :
                case '3' : case '4' : case '5' :
                case '6' : case '7' : case '8' :
                    nbre_chiffres ++;
                    break;
                case ' ' : nbre_espaces ++;
                           break;
                default : autres ++;
            }
    printf("%d chiffres, %d espaces, %d autres",
            nbre_chiffres, nbre_espaces, autres);
}
```

Ce programme compte les chiffres, les espaces et les autres caractères.

Les tableaux

Un tableau est une collection d'objets de même type.

La déclaration d'un tableau se fait de la même manière que celle d'une variable, avec comme indication supplémentaire le nombre d'éléments contenus dans le tableau :

type nom **[***nbre_éléments* **]**

*Le nom d'un tableau correspond à **l'adresse du premier élément** de ce tableau.*

```
main()
{
    int tableau[5];
    int i;
    int somme;

    for (i = 0, somme = 0 ; i < 5; i++)
        somme = somme + tableau[i];
    printf("La somme est %d\n", somme);
}
```

*L'indice associé du tableau **varie toujours de 0** à **la taille du tableau - 1**.*

Initialisation d'un tableau

```
main()
{
    int tableau[5] = { 0, 1, 4, 9, 16 };
    int i;
    int somme;

    for (i = 0, somme = 0 ; i < 5; i++)
        somme = somme + tableau[i];
    printf("La somme est %d\n", somme);
}
```

Tableau à n dimensions

type nom **[***nbre_éléments* **]** … **[***nbre_éléments* **]**

```
    int matrice [10] [20];
    matrice [1][5] = 58;
```

Les pointeurs

L'obtention de l'adresse d'une variable est faite en mettant l'**esperluette devant le nom de la variable**.

```
main()
{
    int a;
    double x;

    printf("L'adresse de a est %08x\n", &a);
    printf("L'adresse de x est %08x\n", &x);
}
```

```
L'adresse de a est f7fffc04
L'adresse de b est f7fffbf8
```

L'adresse d'une variable **ne peut être contenue** dans une variable de type entier, flottant ou caractère.

Il existe un type spécial permettant de stocker cette adresse associé à chaque type de base :

```
int *x;   /* permet de stocker l'adresse d'un
             entier */
float *y; /* permet de stocker l'adresse d'un
             flottant */
```

*Ces variables sont appelées « **pointeurs** ».*

Pour déclarer une variable de type pointeur il suffit de rajouter ***** devant le nom de la variable.

*Attention à toujours **initialiser un pointeur** à une valeur particulière :* la valeur **NULL**.

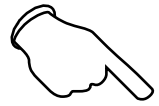
```
int *pointeur_d_entier = NULL;
```

C'est le seul moyen de détecter une erreur.!

Opérations sur les pointeurs

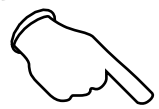
L'opérateur ***** **permet d'obtenir la valeur contenue** dans l'emplacement mémoire dont **l'adresse est stockée dans la variable**.

```
main()
{
    int *x;
    int y = 2;
    int z = 3;

    x = &y;
    printf("x pointe vers un emplacement contenant
%d\n", *x);
    x = &z;
    printf("x pointe vers un emplacement contenant
%d\n", *x);
}
```



```
x pointe vers un emplacement contenant 2
x pointe vers un emplacement contenant 3
```

```
main()
{
    int *x;
    int y = 2;
    int z = 3;

    x = &y;
    *x = 5;
    printf("La valeur de y est %d\n", y);
    x = &z;
    *x = 7;
    printf("La valeur de z est %d\n", z);
}
```



```
La valeur de y est 5
La valeur de z est 7
```

Pointeur et tableau

*Le nom d'un tableau correspond à **l'adresse du premier élément** de ce tableau.*

```
main()
{
    int tableau[] = { 1, 2, 3, 4, 5 };
    int i = 0;
    int somme;

    for (i = 0, somme = 0 ; i < 5; i++)
        somme = somme + *(tableau+i);

    printf("La somme est %d\n", somme);
}
```

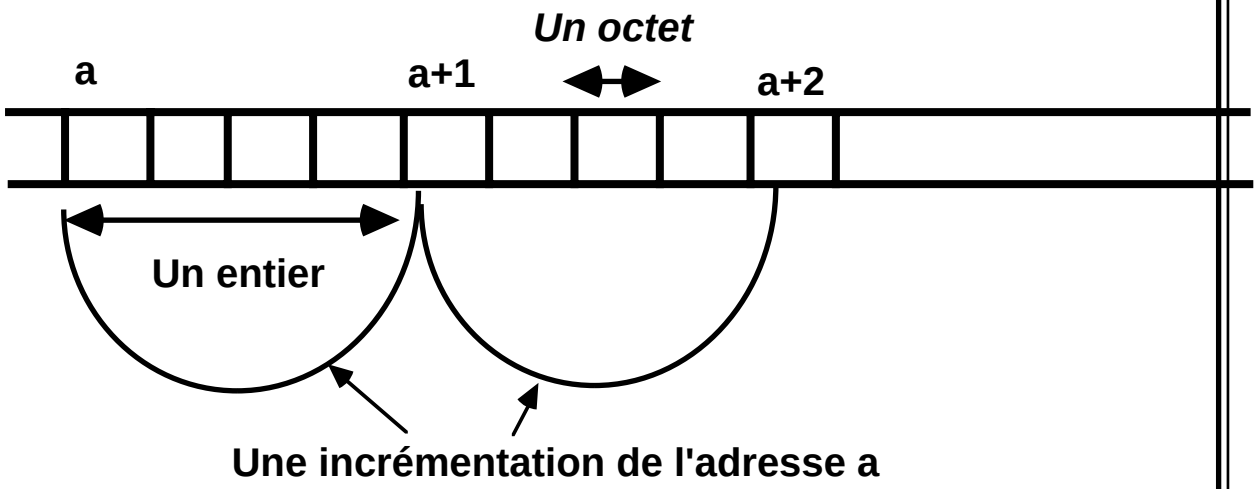
Équivalences entre accés pointeur et par indice pour un tableau :

```
tableau ~ &tableau[0]
tableau[i] ~ *(tableau + i)
```

Il est possible **d'incrémenter** ou de **décrémenter** un **pointeur**.

Ces opérations se font suivant **l'arithmétique des pointeurs**.

L'adresse, quand elle est **incrémentée** (respectivement décrémentée), est en fait **incrémentée** (resp. décrémentée) **de la taille du type de la donnée pointée**.



La chaîne de caractère

La chaîne de caractères est une collection de caractères dont le dernier caractère est nul (**'\0'**).

```
char msg[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Équivalent à

```
char msg[] = "Hello";
```

Le spécificateur de conversion associé est **%s**.

```
main()
{
    char nom[] = "Toto";

    printf("Le nom est %s\n", nom);
}

main()
{
    char saisie[50]; /* Est-ce assez ? */

    scanf("%s", saisie);
    printf("La chaine saisie est %s\n", saisie);
}
```

Attention à la reservation de l'espace nécessaire pour stocker la chaîne !

*Ne pas oublier l'espace pour le caractère '**\0**' !*

```
main()
{
    char nom[] = "Toto";
    int i = 0;
    while(nom[i] != '\0')
        printf("%c\n", nom[i]);
}
```

La chaîne de caractère (suite)

Èquivalences :

— entre un pointeur de caractère

```
char *pointeur_de_caracteres;
```

— entre un tableau

```
char tableau_de_caracteres[20];
```

Affectation

```
pointeur_de_caracteres = tableau_de_caracteres;
```

Accés

```
*pointeur_de_caractere = 'a';
```

équivalent à

```
tableau_de_caracteres[0] = 'a';
```

Initialisation

```
char *ma_chaine = "Ceci est une chaine";

char ma_chaine[] = "Ceci est une chaine";
```

Attention à **ne pas utiliser un char *** comme une chaine sans avoir vérifié qu'il référence bien une zone correcte !

```
main()
{
    char *mon_pointeur;

            /* attention plantage */
    mon_pointeur[5] = 'b';
    scanf("%s", mon_pointeur);
                /* correcte */
    mon_pointeur = "une chaine quelconque";
}
```

Fonctions associées aux chaînes de caractères

La fonction **gets()** :

```
main()
{
    /* ce programme lit une nouvelle chaine tant
        que la ligne ne commence pas par un Z */
int traitement_fini = 0; /* il est a faux */
char buffer[256]; /* on reserve un espace suffisement
                     grand */
while (!traitement_fini)
    {
        printf("Entrez une chaine ");
        gets(buffer);
        printf("La chaine est >>%s<<\n", buffer);
        if (buffer[0] == 'Z') traitement_fini = 1;
    }
}
```

*La fonction **gets()** lit une chaîne jusqu'au retour à la ligne.*

Similairement la fonction **puts()** affiche une chaine de caractère :

```
puts(buffer);
```

Fonctions de conversion :

atoi() convertit une chaîne en valeur entière
"234" —› 234

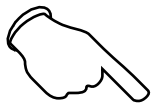
atof() convertit une chaîne en valeur flottante
"234,56" —› 234,56

```
main()
{
    int valeur = 0;
    char chaine = "3489";

    valeur = atoi(chaine);
    printf("La valeur est %d", valeur);
}
```

Fonctions associées aux chaînes de caractères (suite)

sprintf() similaire à la fonction **printf()** mais dont la sortie est dirigée dans une chaîne de caractère

```
main()
{
    char buffer[128];
    double x = 1.23456;
    char *cpos = buffer;
    int i = 0;

    sprintf(buffer, "x = %7.5lf", x);
    while(i<10)
        {
            puts(cpos+i);
            i++;
        }
}
```



```
x = 1.23456
 = 1.23456
= 1.23456
 1.23456
1.23456
.23456
23456
3456
456
56
```

Les fonctions :

strlen() renvoie la longueur d'une chaîne de caractère (*sans compter le caractère spécial de fin de chaîne '**\0**'…*)

strcpy() copie d'une chaîne dans une autre (*attention à la taille de la chaîne de destination !*)

strcmp() comparaison de deux chaînes de caractères

***Attention** à ne pas utiliser l'égalité **==** pour comparer deux chaînes.*

En effet seules les adresses des deux chaînes sont comparées et non le contenu de ses deux chaînes !

Les fonctions

Ce sont des blocs de code.

En C, on utilise toujours des fonctions, qu'elles soient de bibliothèques comme **printf()** ou **scanf()**, ou qu'elles soient définies par le programmeur par exemple la fonction principale **main()**.

Les fonctions permettent de regrouper des portions de code pour pouvoir les utiliser et les ré-utiliser.

```
main()
{
    int n1, n2, n3;
    printf("Entrez un nombre ");
    scanf("%d, &n1);
    printf("Entrez un nombre ");
    scanf("%d, &n2);
    printf("Entrez un nombre ");
    scanf("%d, &n3);
    printf("La somme est %d\n", n1+n2+n3);
}
```

Dans ce programme il n'y a pas de vérification d'éventuelles erreurs de saisie; il serait difficile de les incorporer pour chaque appel de la fonction **scanf()**.

```
main()
{
    int n1, n2, n3;
    n1 = lire_entier();
    n2 = lire_entier();
    n3 = lire_entier();
    printf("La somme est %d\n", n1+n2+n3);
}
int lire_entier()
{
    int lecture_entier_correct = 0; /* a faux */
    char buffer[256];
    int valeur_lue = 0;
    while(!lecture_entier_correct)
    {
        gets(buffer);
        valeur = atoi(buffer);
        if (errno == 0) /* pas d'erreur */
            lecture_entier_correct = 1;
    }
    return valeur_lue;
}
```

Les fonctions (suite)

Les règles pour le choix des noms de fonctions sont les mêmes que pour ceux des variables.

Une variable et une fonction **ne peuvent porter le même nom**.

Plutôt que d'afficher la valeur lue à l'écran, elle est **retournée** par l'instruction **return valeur;**

Quand l'affectation **n1 = lire_entier();** est évalué la valeur de la fonction **lire_entier()** est la valeur associée à l'instruction "**return**" qui est la **dernière instruction** exécutée de la fonction **lire_entier()**.

Un programme C normal correspond à la définition d'une série de fonctions.

La définition d'une fonction consiste en l'indication du **type de la valeur retournée** suivi du **nom de la fonction** et d'une **paire de parenthèses**.

int lire_entier ()

La fonction **lire_entier()** n'est pas inclue dans le corps de la fonction **main()**.

*Lors de l'appel de la fonction **lire_entier**, bien qu'il n'y ait pas d'arguments à lui transmettre, une paire de parenthèses sont ajoutées.*

La définition des fonctions peut se faire dans n'importe quel ordre.

La déclaration d'une fonction ne peut être faite à l'intérieur de la défintion d'une autre fonction.

L'instruction **return** peut se trouver n'importe où dans le corps de la fonction.

Si l'exécution se poursuit jusqu'à atteindre la fin de la fonction alors un "**return**" implicite est réalisé.

La fonction **lire_entier()** réalise toujours le même traitement.

Il est possible de **modifier son comportement** au cours de l'exécution en utilisant des **arguments**.

Les arguments de la fonction

Il existe les arguments dits **formels** et ceux dits **réels**.

L'argument **formel** correspond à une **variable** associée à la définition de la fonction, **réservant de l'espace de stockage** pour les arguments **réels** apparaissant dans un **appel de fonction**.

Pour déclarer un argument formel, le type et le nom de cet argument sont compris dans les parenthèses associées à la définition de la fonction :

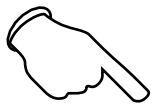
```
main()
{
    int somme = 0, n[3], i = 0;
    while(i<3)
    {
        n[i] = lire_entier(i+1);
        somme += n[i];
        i++;
    }
    printf("La somme est %d\n", somme);
}

int lire_entier(int indice)
{
    int lecture_entier_correct = 0; /* a faux */
    char buffer[256];
    int valeur_lue = 0;

    while(!lecture_entier_correct)
    {
        printf("Entrez l'entier numero %d\n",
                    indice);
        gets(buffer);
        valeur = atoi(buffer);
        if (errno == 0) /* pas d'erreur */
            lecture_entier_correct = 1;
    }
    return valeur_lue;
}
```

Ici l'argument formel est **indice**.

*L'argument formel **contient une copie** de l'argument réel.*

Les arguments de la fonction (suite)

```
main()
{
    int n[3] = { 1, 2 , 3};
    int somme = 0;
    addition(n[0], n[1], n[2], somme);
    printf("La somme est %d\n", somme);
}

addition(int a, int b, int c, int total)
{
    total = a + b + c;
    printf("total = %d\n", total);
}
```

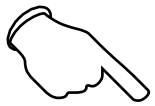


```
total = 6
La somme est 0
```

Il est normal qu'on **ne puisse modifier** à l'intérieur de la fonction la valeur des arguments réels, car le code de la fonction n'a **aucun moyen de savoir** si les arguments réels correspondent au **nom d'une variable**, ou à une **expression** ou alors à une **constante**.

Il est **possible de retourner des valeurs** par l'intermédiaire des paramètres en **transmettant à la fonction l'adresse** de l'emplacement mémoire.
Le paramètre formel doit être du type **pointeur** :

```
main()
{
    int n[3] = { 1, 2 , 3};
    int somme = 0;
    addition(n[0], n[1], n[2], &somme);
    printf("La somme est %d\n", somme);
}

addition(int a, int b, int c, int *total)
{
    *total = a + b + c;
    printf("total = %d\n", *total);
}
```



```
total = 6
La somme est 6
```

La fonction (suite)

Pour pouvoir utiliser une fonction il est nécessaire d'en connaitre le **nom**, le **type de la valeur de retour** ainsi que le **nombre d'arguments** et **le type** de chacun de ces arguments.

*Une fonction qui ne retourne rien est definie comme retournant **void**.*

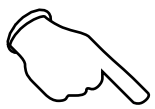
void main()

La déclaration d'une fonction précédant sa définition s'appelle un **prototype** :

double calcul(int);

Il n'est pas nécessaire de rappeler le nom des arguments.

```
int somme(int [], int);

void main()
{
    int x[] = {1, 5, 3, 2, 7, 4};
    printf("La somme des nombres = %d\n",
            somme(x, (sizeof(x)/sizeof(int))));
}

int somme(int q[], int nombre_elements)
{
    int i = 0;
    int total = 0;
    while(i<nombre_elements)
    {
        total+=q[i];
        i++;
    }
}
```



```
La somme des nombres = 22
```

*Il **n'est pas nécessaire** de transmettre la taille du tableau pour l'agument formel **int q[ ]**.*

Par contre il est obligatoire de transmettre la taille du tableau pour pouvoir le parcourir.

La fonction (suite)

```
int longueur_chaine(char *);

void main()
{
    char buffer[256];

    printf("Entrez une chaine ");
    gets(buffer);
    printf("La taille de la chaine est %d\n",
            longueur_chaine(buffer));
}

int longueur_chaine(char *chaine)
{
    int i = 0;

    while(chaine[i] != '\0')
        i++;

    return i;
}
```

*Rappel : le nom du tableau **est équivalent** à l'adresse du début de ce tableau.*

char *chaine** est équivalent à **char chaine[ ]

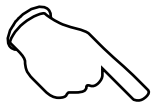
Les variables locales

Toutes les variables *devraient* être **définies à l'intérieur** des accolades ouvrantes et fermantes correspondant au **corps d'une fonction**.

Ces variables sont dites locales à la fonction où elles sont définies et ne peuvent être référencées à l'extérieur de la fonction.

Elles **n'existent plus** au sortir de la fonction !

```
void fonction(void);

void main()
{
    int i = 0;

    while(i < 3)
    {
        printf("i = %d \n", i);
        fonction();
        i++;
    }
}

void fonction(void)
{
    int i = 4;
    printf("Valeur initiale de i = %d\n", i);
    i = 7;
    printf("  Valeur finale de i = %d\n", i);
}
```



```
i = 0
Valeur initiale de i = 4
  Valeur finale de i = 7
i = 1
Valeur initiale de i = 4
  Valeur finale de i = 7
i = 2
Valeur initiale de i = 4
  Valeur finale de i = 7
```

Les variables « **i** » sont strictement indépendantes.

Gestion des fichiers

La bilbliothèque de focntions **stdio.h** fournit un certain nombre de ce fonctions permettant la gestion de fichiers.

fopen() Ouvre un fichier
fclose() Ferme un fichier
fprintf() Écriture formattée dans un fichier
fscanf() Lecture formattée depuis un fichier
fputc() Écriture d'un caractère dans un fichier
fgetc() Lecture d'un caractère depuis un fichier
fputs() Écriture d'une chaîne de caractère dans un fichier
fgets() Lecture d'une chaîne de caractère depuis un fichier
feof() Test si la fin du fichier est atteinte.

Écriture dans un fichier de valeurs d'échantillonnage :

```
void main()
{
    FILE *fichier_sortie; /* déclaration d'un
poiteur de fichier */
    double x, y;
    int saisie_terminee = 0; /* faux */

    fichier_sortie = fopen ("donnee.data", "w");
    if (fichier_sortie == NULL)
        {
            printf("Erreur dans l'ouverture du
fichier\n");
            exit(1); /* en cas d'echec on sort */
        }
    printf("Entrez la valeur d'echantillon (0.0,
0.0) pour finir");
    while(!saisie_terminee)
    {
        printf("\nDonnez x ");
        scanf("%lf", &x);
        printf("\nDonnez y ");
        scanf("%lf", &y);
        if ((x == 0.0) && (y == 0.0))
            saisie_terminee = 1; /* Vrai */
        else
        fprintf(fichier_sortie, "%lf\t%lf\n", x,
y);
    }
```

Ouverture d'un fichier

Il existe un pointeur de type spécial permettant de manipuler les informations caractérisant un fichier :

FILE **descripteur_de_fichier*

Avant toute opération de **lecture** ou **d'écriture** un fichier doit être **ouvert.**

fopen(*nom_du_fichier*, *mode_d_ouverture)*

Le **nom du fichier** doit être convenablement choisi en accord avec le système d'exploitation où s'exécutera le programme.

Le **mode d'ouverture** correspond aux **opérations possibles** sur le fichier ainsi que l'**état initial du fichier**.

Une demande de création du fichier (avec destruction du fichier s'il existait précédemment) :

"w" Ouverture du fichier pour écriture

Une ouverture pour lecture seulement :

"r" Ouverture du fichier pour lecture

Une ouverture pour ajout en fin du fichier :

"a" Ouverture du fichier pour ajout en fin

Si l'ouverture du fichier échoue (nom incorrect, plus de place sur le disque, interdiction de création, …) la fonction **fopen()** renvoie la valeur **NULL**.

Lecture/écriture dans un fichier

Par défaut les opérations d'entrée/sortie se font sur la **sortie** et **l'entrée** standard.

stdout (sortie standard) et **stdin** (entrée standard)

La fonction **printf()** est équivalente à la fonction **fprintf()** appliquée à la sortie standard :

```
printf("La valeur est %d\n", ma_valeur);
```

est équivalent à

```
fprintf(stdout, "La valeur est %d\n", ma_valeur);
```

De même :

```
scanf("%d", &ma_valeur);
```

est équivalent à

```
fscanf(stdin, "%d", &ma_valeur);
```

`fputc(c, stdout);` et `putc(c);`

`c = fgetc(stdin);` et `c = getc();`

`fputs(chaine, stdout);` et `puts(chaine);`

`fgets(chaine, stdin);` et `gets(chaine);`

La fonction **feof()** renvoie vrai ou faux selon que la fin du fichier est atteinte ou non.

Exemple de lecture dans un fichier

```
void main()
{
    FILE *fichier_entree;
    double x, y;

    fichier_entree = fopen (“donnee.data”, “r”);

    if (fichier_entree == NULL)
        {
            printf(“Erreur dans l’ouverture du
fichier\n”);
            exit(1); /* en cas d’echec on sort */
        }

    while(!feof(fichier_entree))
    {
        fscanf(“%lf\t%lf”, &x, &y);
        scanf(“%lf”, &y);
        printf(“x = %lf y = %lf\n”, x, y);
    }
}
```

Les fichiers (suite)

Un **fichier** est une **suite d'octets** situés sur un support physique.

Il existe deux sortes de fichiers :

— Les fichiers dits **structurés** ou **binaires**

Ils correspondent à une **suite d'enregistrements de taille identique**, dont la lecture et l'écriture se font de manière binaires.

L'information enregistrée ou lue **correspond directement** au contenu mémoire des variables transférées et avec la **même organisation** que ces variables (**ordre** et **type**).

*Un **nombre** est stocké dans un fichier de type binaire sous la **forme de sa représentation binaire**, il n'est donc **pas possible** de le lire directement : il faut **savoir** où sa représentation **commence** et où elle **finit** dans le fichier.*

Il n'est pas possible d'exploiter correctement le contenu d'un fichier binaire sans en transférer le contenu dans la même organisation de variables.

Ce type de fichier permet **l'accés directe** à une information :

*La **taille** d'un enregistrement étant connue, il est facile de déterminer **l'emplacement** d'un enregistrement déterminé dans le fichier.*

Il n'est donc **pas nécessaire** de **parcourir tout le fichier** pour trouver l'enregistrement voulu.

Les fichiers (suite)

— Les fichiers dits **texte**

C'est un **ensemble de lignes** séparées par des **retours à la ligne '\n'**, pouvant contenir différentes sortes de caractères.

Il conserve les informations sous forme de leur représentation « humaine ».

*Un **nombre** est stocké dans un fichier de type texte sous la **forme de chaîne de caractères**, il est donc **possible** de le lire directement dans le fichier.*

*Attention à utiliser des **délimiteurs** pour séparer les informations !*

Chaque *enregistrement* ou **ligne** est de taille variable.

Il est **obligatoire** de **parcourir l'ensemble du fichier** pour arriver sur un enregistrement particulier.

L'interêt du fichier texte est qu'il se comporte dans son **organisation** et ses **accés** comme **l'entrée** et la **sortie standard**.

Toute opération **d'entrée depuis le clavier** ou de **sortie vers l'écran** peut être **transposée** de manière transparente vers un fichier.

Les **informations stockées** dans un fichier texte peuvent être manipulées dans un **éditeur de texte** quelconque, dans un **tableur**…

Le fichier texte est le **meilleur intermédiaire** entre un programme et un autre ou bien entre un programme et l'utilisateur.

Il est moins efficace que le fichier binaire.

Des opérations sur les fichiers textes

*Lors de la **manipulation d'une chaîne de caractère** il est **important** de s'assurer que la t**aille de la chaîne est correcte** par rapport à la **taille de la zone mémoire réservée** à cette chaîne.*

Lors de la lecture d'un fichier texte ligne par ligne il est obligatoire de s'assurer que la ligne courante du fichier peut être contenue dans l'emplacement mémoire que l'on désire utiliser.

La fonction **fgets() :**

char *fgets(char *buffer, int taille_max, FILE *descripteur)

Cette fonction posséde un **deuxième argument** chargé d'indiquer à la fonction le **nombre maximum de caractères** stockables dans la chaîne de caractères passée en premier argument.

*Dans le cas où la **ligne** est **plus longue** que la **zone transmise** à la fonction **fgets()** s**euls les caractères pouvant être stockés** sont lus, l**es autres caractères étant laissés** pour une prochaine lecture.*

Le PréProcesseur

Le préprocesseur est un outil intervenant sur le source du programme avant le compilateur.

Il réalise essentiellement des **substitutions de texte** sur le code source.

Une **directive** du préprocesseur est reconnue par le fait qu'elle est **précédée** d'un caractère « # ».

Une directive se **termine** sur la **fin de la ligne** où elle apparait.

Les directives les plus utilisées sont :

#define et **#include**

La directive #include :

Elle sert à **incorporer** dans un source la **documentation** (*fichier avec une extension **« .h »**)* relative à l'utilisation d'une bibliothèque de fonctions.

#include <stdio.h>

*L'indication du fichier de documentation à inclure se fait entre **<** et **>**, si la bibliothèque fait partie des **bibliothèques standards** du C.*

*L'indication du fichier de documentation à inclure se fait entre **"** et **"** dans le cas où la bilbliothèque est une **bibliothèque écrite par le programmeur**.*

La documentation consiste en la liste de prototypes de toutes les fonctions définies dans la bibliothèque associée.

Le PréProcesseur (suite)

La directive #define :

Cette directive est utilisée pour associer une chaîne de caractères particulière à une chaîne de remplacement.

*L'opération du préprocesseur est **similaire** à l'opération de **chercher/remplacer** d'un éditeur de texte.*

Cette directive est utile pour se référer à des éléments fréquemment utilisés :

```
#define TAILLE_BUFFER 255

void main()
{
    FILE *descripteur = NULL;
    char buffer[TAILLE_BUFFER];

    descripteur = fopen ("donnee.data", "r");

    if (descripteur == NULL)
        {
            printf("Erreur dans l'ouverture du
fichier\n");
            exit(1); /* en cas d'echec on sort */
        }

    fgets(buffer, TAILLE_BUFFER, descripteur);

    printf("La premiere ligne du fichier est \n%s\n",
            buffer);

    fclose(descripteur);
}
```

Les structures en C

La **structure** permet de **regrouper** des variables de **différents types** au sein d'une même entité.

Cette structure devient un **nouveau type** dans le programme où elle est définie.

C'est une **collection d'objets** qui **partage le même identificateur** de variable, et posséde un identificateur supplémentaire (*un **champ***) permettant d'accéder à chacun des objets séparément.

```
struct date /* Le nom de la structure*/
{
    int jour;        /* un membre */
    int mois;        /* un membre */
    int annee;       /* un membre */
    char nom[16];    /* un membre */
};
```

Pour déclarer des variables de ce type :

```
struct date dates[TAILLE_MAX], aujourdhui;
```

Pour l'initialisation :

```
struct date noel = {25, 12, 1998, "Noel"};
```

Pour l'accés :

```
struct date un_jour;

un_jour.jour = 12;
un_jour.mois = 2;
```

La structure *peut-être transmise en paramètre à une fonction*
retournée depuis une fonction

Les structures en C (suite)

```
#define NBRE_ECHANTILLONS 16

struct mesure
{
    double x;
    double y;
};


struct mesure lecture_mesure()
{
    struct mesure m;

    printf("Donnez x ");
    scanf("%lf", &(m.x));
    printf("Donnez y ");
    scanf("%lf", &(m.y));

    return m;
}



void main()
{
    struct mesure tab_mesures[NBRE_ECHANTILLONS];
    int i;


    for(i = 0; i < NBRE_ECHANTILLONS; i ++)
        tab_mesures[i] = lecture_mesure();


    /* calcul sur les differentes mesure lues */



    for(i = 0; i < NBRE_ECHANTILLONS; i ++)
        printf("%lf\t%lf", tab_mesures[i].x,
                    tab_mesures[i].y);
}
```