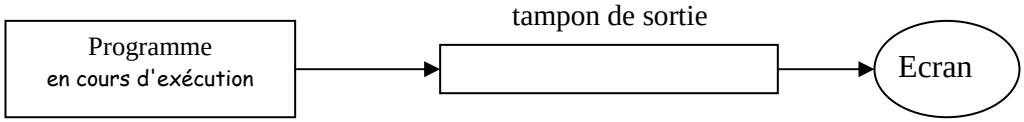
Les entrées / sorties standard en langage C

La réalisation d'entrées /sorties standard en langage C nécessite l'inclusion du fichier d'entête **<stdio.h>.**

1. La fonction printf()

Elle permet de réaliser des **affichages sur la sortie standard** (par défaut l'écran).
Les écritures sur la sortie standard se font par l'intermédiaire d'un tampon (ou buffer) :



Les flèches représentent le sens de circulation des données. Ces données qui sortent du programme en cours d'exécution et qui rentrent dans le tampon sont des **caractères**.

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

Son prototype indique que pour l'appeler, il faut lui fournir un 1er paramètre de type chaîne de caractères constante. Ce 1er paramètre correspond au *format*. Les ... indique que la liste des paramètres ne peut pas être entièrement spécifiée à l'avance.
Le format est une chaîne de caractères constituée de caractères ordinaires simplement écrits sur la sortie standard et de **spécifications de format repérées par un %.** Chaque spécification de format contient un **code de conversion** et doit correspondre à un **paramètre supplémentaire** : ce dernier doit être converti suivant le code de conversion avant son insertion dans la chaîne à écrire.
La fonction renvoie le nombre de caractères écrits en cas de succès et un nombre négatif dans le cas contraire.

Chaque spécification de format a la structure suivante :

`% [drapeaux] [largeur] [.précision] [h|l] conversion`

dans laquelle les crochets [et] signifient que ce qu'ils renferment est facultatif.
Les différentes « indications » se définissent comme suit :

- **conversion** : il s'agit d'un caractère qui précise à la fois le **type de l'expression** à afficher (indiqué en *italique*) et la **façon de présenter sa valeur**. Les types indiqués correspondent au cas où aucun modificateur (h ou l) n'est utilisé (voir ci-dessous) :

- **`d`** : *`int`*, affiché en décimal
- **`o`** : *`unsigned int`*, affiché en octal
- **`u`** : *`unsigned int`*, affiché en décimal
- **`x`** : *`unsigned int`*, affiché en hexadécimal (lettres minuscules)
- **`X`** : *`unsigned int`*, affiché en hexadécimal (lettres majuscules)
- **`f`** : *`double ou float`*, affiché en notation décimale (virgule fixe)
- **`e`** : *`double ou float`*, affiché en notation exponentielle (avec la lettre `e`)
- **`E`** : *`double ou float`*, affiché en notation exponentielle (avec la lettre `E`)
- **`c`** : `char`, pas de formatage des données (c'est à dire pas de modification du format des données)
- **`s`** : *`chaîne de caractères`*, pas de formatage des données
- **`p`** : *`pointeur`*, adresse affichée en hexadécimal (précédée du préfixe 0x)
- **`%`** : pour afficher le caractère %

- **`h|l`** : (h ou l)

`h` : Ce code précise que l'expression correspondante est de type **`short int`** ou **`unsigned short int`**. Il n'a de signification que pour les caractères de conversion : `d`, `o`, `u`, `x` ou `X`
`l` : Ce code précise que l'expression correspondante est de type **`long int`** ou **`unsigned long int`**. Il n'a de signification que pour les caractères de conversion : `d`, `o`, `u`, `x` ou `X`

Pour les autres "indications" facultatives des spécifications de format, voir les exemples ci-dessous.

1.1 Premiers exemples

```
printf ("Hello\n", n) ;    // pas de spécification de format
                           // affiche "Hello" + un saut de ligne
                           // valeur renvoyée (nombre de caractères affichés) : 5
int n = 30;
printf ("En décimal, n = %d\n", n) ; // affiche : ..............................
printf ("En hexa, n = %X\n", n) ; // affiche : ..................................

long int a = -7893; unsigned short int b = 56; // A afficher en décimal
printf ("a = %ld, b = %hu\n", a, b) ;  // affiche : ..............................

char Mot[] = "bonjour";
printf ("%s", Mot) ; // affiche : ........................
printf ("%c", Mot[2]) ; // affiche : ........................
```

1.2 Action sur le gabarit : rôle de l'indication largeur

Par défaut, les entiers sont affichés avec le nombre de caractères nécessaires (sans espaces avant ou après). Les flottants sont affichés avec six chiffres après le point (aussi bien pour le code de conversion `e` que `f`).

Un nombre placé après % dans la spécification de format précise un **gabarit d'affichage**, c'est-à-dire un nombre **minimal** de caractères à utiliser. Si le nombre peut s'écrire avec moins de caractères, `printf()` le fera précéder d'un nombre suffisant d'espaces ; en revanche, si le nombre ne peut s'afficher convenablement dans le gabarit imparti, `printf()` utilisera le nombre de caractères nécessaires.

Exemples :

```
printf ("%3d", n) ; /* entier avec 3 caractères minimum */
n = 20              Affichage : ^20
n = 3 ^^3           Affichage : ^^3
n = 2358            Affichage : 2358
n = -5200           Affichage : -5200

printf ("%f", x);   // notation décimale
                    // gabarit par défaut, 6 chiffres après le point
x = 1.2345          Affichage : 1.234500
x = 12.3456789      Affichage : 12.345679
x = -700.89         Affichage : ..................................................

printf ("%10f", x) ;       /* notation décimale - gabarit mini 10 */
                           /* (toujours 6 chiffres après point) */
x = 1.2345          Affichage :^^1.234500
x = 12.345          Affichage : ^12.345000
x = 1.2345E5        Affichage : 123450.000000

printf ("%e", x) ;         /* notation exponentielle - gabarit par défaut */
                           /* (6 chiffres après point) */
x = 1.2345          Affichage : 1.234500e+00
x = 0.0123          Affichage :
x = 123.45          Affichage : 1.234500e+02
x = 123.456789E8    Affichage : 1.234568e+10
x = -123.456789E8   Affichage : -1.234568e+10
```

1.3 Action sur la précision d'affichage des réels : rôle de l'indication précision

Pour l'affichage des réels, on peut spécifier un nombre de chiffres (éventuellement inférieur à 6) après le point décimal (aussi bien pour la notation décimale que pour la notation exponentielle).

Exemples : (avec une variable x de type float)

```
printf ("%10.3f", x) ;     /* notation décimale, gabarit mini 10 */
                           /* et 3 chiffres après point */
x = 1.2345          Affichage : ^^^^^1.235
```

```
x = 1.2345E3          Affichage : ^^1234.500
x = 1.2345E7          Affichage : 12345000.000

printf ("%12.4e", x) ;     /* notation exponentielle, gabarit mini 12*/
                           /* et 4 chiffres après point */
x = 1.2345            Affichage : ^^1.2345e+00
x = 123.456789E8      Affichage : ^^1.2346e+10
```

1.4 Présentation de 2 drapeaux

Le drapeau -, placé immédiatement après le symbole % (comme dans %-4d ou %-10.3f), demande de cadrer l'affichage à gauche au lieu de le cadrer (par défaut) à droite ; les éventuels espaces supplémentaires sont donc placés à droite et non plus à gauche de l'information affichée.

Exemple :
`printf ("%-4d", 26) ;` Affichage : 26^^

Par défaut le caractère de remplissage des gabarits d'affichage est le caractère espace. Le drapeau 0 permet de fixer comme caractère de remplissage le caractère 0.

Exemple :
`printf ("%04d", 26) ;` Affichage : 0026

1.5 La macro putchar()

Prototype : `int putchar (int caractere);`

Cette fonction permet d'afficher sur la sortie standard le *caractere* passé en paramètre. Elle renvoie le *caractere* écrit en cas de succès, et la constante EOF sinon.

Avec c une variable de type char, l'expression :
`putchar(c)`
joue le même rôle que :
`printf ("%c", c)`

1.6 La fonction sprintf()

Prototype : `int sprintf(const char *ch, const char *format, ...);`

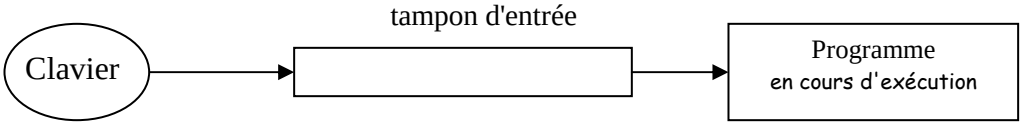
Son fonctionnement est similaire à celui de la fonction `printf()`. Au lieu d'écrire ses données sur la sortie standard (comme le fait `printf()`), cette fonction les écrit dans la chaîne de caractères `ch`.

Exemple : conversion d'un entier en chaîne de caractères

```
int nb = 56;
char ch[10];
sprintf(ch, "%d", nb);  // écrit "56" dans ch
```

2. La fonction scanf()

Elle permet de réaliser des **saisies sur l'entrée standard** (par défaut le clavier).
Les lectures sur (ou depuis) l'entrée standard se font par l'intermédiaire d'un tampon (ou buffer) :



Les flèches représentent le sens de circulation des données. Ces données qui sortent du tampon et qui rentrent dans le programme en cours d'exécution sont des **caractères**.

Prototype : `int scanf (const char * format, ...) ;`

Cette fonction permet de lire sur l'entrée standard des suites de caractères en leur appliquant une **procédure de conversion** selon le *format* fourni en 1er paramètre. Les paramètres sous-jacents (dont

l'existence est indiquée par ...) doivent correspondre aux **adresses des variables à saisir :** les variables sont modifiées avec les valeurs obtenues après les conversions. Le *format* est une chaîne de caractères essentiellement constituée de **spécifications de formats**. Chaque spécification de format est repérée par un % et doit correspondre à une adresse de variable à saisir.
La fonction renvoie le nombre de variables convenablement saisies.

Chaque spécification de format a la structure suivante :

```
% [*] [largeur] [h|l] conversion
```

dans laquelle les crochets [et] signifient que ce qu'ils renferment est facultatif. Les différentes « indications » se définissent comme suit :

- ***** : la valeur lue n'est pas prise en compte ; elle n'est donc affectée à aucun élément de la liste
- **largeur** : nombre **maximal** de caractères à prendre en compte (on peut en lire moins s'il y a rencontre d'un séparateur ou d'un caractère invalide)
- **h | l** : (h ou l)

• **h** : l'élément correspondant est l'adresse d'un `short int`. Ce modificateur n'a de signification que pour les caractères de conversion : `d`, `i`, `o`, `u`, ou `x`
• **l** : l'élément correspondant est l'adresse d'un élément de type :
- `long int` pour les caractères de conversion `d`, `i`, `o`, `u` ou `x`
- `double` pour les caractères de conversion `e` ou `f`

- **conversion** : ce caractère précise à la fois le **type de l'élément** correspondant (indiqué ici en *italique*) et la **manière dont sa valeur sera exprimée**. Les types numériques indiqués correspondent au cas où aucun modificateur (h ou l) n'est utilisé (voir ci-dessus). Il ne faut pas perdre de vue que l'élément correspondant est toujours désigné par son adresse.

Ainsi, par exemple, lorsqu'on parle de *int*, il faut lire : « adresse d'un *int* »

• **d** : *int* exprimé en décimal
• **o** : *int* exprimé en octal
• **i** : *int* exprimé en décimal, en octal ou en hexadécimal :
  - sans préfixe, le nombre est exprimé en décimal,
  - le préfixe 0 indique que le nombre est écrit en octal,
  - le préfixe 0x indique que le nombre est écrit en hexadécimal.
• **u** : *unsigned int* exprimé en décimal
• **x** : *int* exprimé en hexadécimal
• **f**, **e** : *float* écrit indifféremment en notation décimale (éventuellement sans point) ou exponentielle (avec e ou E)
• **c** : *char*
• **s** : *chaîne de caractères*
• **p** : *pointeur* exprimé en hexadécimal, sous la forme employée par `printf`

Remarques :
- Attention, certaines spécifications de format n'ont pas la même signification suivant qu'elles sont employées avec `printf()` ou avec `scanf()`.
- Certains caractères dits « **séparateurs** » (ou « espaces blancs ») jouent un rôle particulier dans les données. Les deux principaux sont l'espace et la fin de ligne (`\n`). Il en existe trois autres d'un usage beaucoup moins fréquent : tabulation horizontale (`\t`), la tabulation verticale (`\v`) et le changement de page (`\f`).
- L'information est recherchée dans un tampon, image d'une **ligne (donc terminée par un saut de ligne).** Il y a donc une certaine désynchronisation entre ce que l'on frappe au clavier (lorsque l'entrée standard est connectée à ce périphérique) et ce que lit *la fonction*. Lorsqu'il n'y a plus d'information disponible dans le tampon, il y a déclenchement de la lecture d'une nouvelle ligne.

Remarque : dans tous les exemples qui suivent `^` désigne un espace et `@` une fin de ligne.

2.1 Saisies de caractères en utilisant le code de conversion c

Ce code de conversion permet la saisie de n'importe quel caractère y compris les caractères séparateurs.
Suivant la largeur, il correspond à la saisie :
– d' un *caractère* lorsqu'aucune largeur n'est spécifiée ou que celle-ci est égale à `1`,

– d' une *suite de caractères* lorsqu'une largeur strictement supérieure à `1` est spécifiée. Dans ce cas, il ne faut pas perdre de vue que la fonction reçoit une adresse et que donc elle lira le nombre de caractères spécifiés et les rangera à partir de l'adresse indiquée. Il faut bien sûr que la place nécessaire ait été réservée au préférable.
Attention, dans ce cas, la fonction `scanf()` **ne place pas de caractère nul** après le dernier caractère saisi.

Exemples :
```
char c;             // déclaration d'une variable servant pour la saisie
scanf ("%c", &c) ;  // saisie d'un caractère
```
a@ c = ..., après la saisie, tampon = ...
^a@ c = ..., après la saisie, tampon = ...
@ c = ..., après la saisie, tampon = ...

```
char tab[] = "bonjour"; // déclaration d'une variable servant pour la saisie
scanf ("%2c", tab) ;    // saisie de 2 caractères avec un seul appel de scanf()
```
a@ tab = ..., après la saisie, tampon = ...
^a@ tab = ..., après la saisie, tampon = ...
@@ tab = ..., après la saisie, tampon = ...

2.2 Saisies de chaîne de caractères en utilisant le code de conversion s

Ce code de conversion ne permet pas la saisie de chaîne de caractères contenant des caractères séparateurs.
Dans ce cas, la fonction `scanf()` doit recevoir en paramètre une adresse de caractère.
Les caractères séparateurs, qui précède le 1er caractère qui n'en est pas un, sont ignorés. Puis, la fonction lit tous les caractères jusqu'à la rencontre d'un séparateur (ou un nombre de caractères égal à la largeur éventuellement spécifiée) et elle les range à partir de l'adresse indiquée. Enfin, elle place un caractère nul après le dernier caractère saisi. Si la saisie est "stoppée" par un caractère séparateur, ce caractère est laissé dans le tampon.

Exemples :
```
char tab[] = "bon";     // déclaration d'une variable servant pour la saisie
scanf ("%s", tab) ; // saisie d'une chaîne de caractère
```
^^@ab^@ tab = ..., après la saisie, tampon = ...
@^^@12345^@ tab = ..., après la saisie, tampon = ...

Remarque : quand on ne précise pas la largeur, les saisies ne sont pas protéger contre les éventuels débordements de tableau.

```
char tab[5];      // déclaration d'une variable servant pour la saisie
scanf ("%4s", tab) ;// saisie d'une chaîne de caractère (longueur max 4 caractères)
```
^^@ab^@ tab = ..., après la saisie, tampon = ...
@^^@12345^@ tab = ..., après la saisie, tampon = ...

2.3 Saisies de valeurs numériques

Les caractères séparateurs, qui précède le 1er caractère qui n'en est pas un, sont ignorés.
- Si ce **1er caractère différent d'un séparateur est valide**, ce caractère et les suivants qui sont valides (dans la limite d'une éventuelle largeur maximale) sont utilisés pour fabriquer une valeur numérique et cette valeur est affectée dans la variable correspondante. Dans ce cas, la **saisie réussit**. Donc en cas de succès, l'exploration du tampon s'arrête :
  - à la rencontre d'un caractère invalide par rapport à l'usage qu'on doit en faire (point décimal pour un entier, caractère différent d'un chiffre ou d'un signe, ...);
  - à la rencontre d'un séparateur;
  - lorsque la longueur (si elle a été spécifiée) a été atteinte.
- Si ce **1er caractère différent d'un séparateur n'est pas valide**, la saisie correspondante échoue, la variable correspondante n'est pas modifiée, ce caractère non valide est laissé dans le tampon (il sera donc disponible pour une prochaine saisie) et la fonction `scanf()` **s'arrête prématurément**.

Exemples :
```
int ret;                // on y affecte la valeur de retour de scanf()
short int m = 5;        // déclaration d'une variable servant pour la saisie
ret = scanf ("%hd", &m) ; // saisie d'un entier exprimé en décimal
```
^^@a8^@ m = 5, ret = 0 après la saisie, tampon = a8^@

@^^@345b^@ m = 345, ret = 1 après la saisie, tampon = b^@

```
int ret;
long int a = 10, b = 2;        // déclaration de variables servant pour la saisie
ret = scanf ("%ld%ld", &a, &b) ; // saisie de 2 entiers exprimé en décimal
```

^^@8^73^@ a = 8, b = 73, ret = 2 après la saisie, tampon = ^@
@^^@345b^@ a = 345, b = 2, ret = 1 après la saisie, tampon = b^@
@^^@z345b^@ a = 10, b = 2, ret = 0 après la saisie, tampon = z345b^@

```
int ret;
int x = 40; short int y = 9;  // déclaration de variables servant pour la saisie
ret = scanf ("%3d%2hd", &x, &y) ;// saisie de 2 entiers exprimé en décimal avec
                                 // des largeurs maximales spécifiées
```

^^@8^73^@ x = 8, y = 73, ret = 2 après la saisie, tampon = ^@
^^@8^734^@ x = 8, y = 73, ret = 2 après la saisie, tampon = 4^@
@^^@7894b^@ x = 789, b = 4, ret = 2 après la saisie, tampon = b^@
@^^@345j^@ x = 345, y = 9, ret = 1 après la saisie, tampon = j^@

2.4 Rôle d'un espace dans le format

Quand on place un espace dans le paramètre *format*, à l'instant où cet espace est pris en compte, la fonction `scanf()` extrait du tampon en les ignorant tous les caractères séparateurs précédant le prochain caractère qui n'est pas un séparateur. Un seul espace dans le paramètre *format* valide l'extraction d'une quantité quelconque de caractères séparateurs du tampon (y compris 0).

Exemples :

```
int ret;
int n = 1; char c = 'e';      // déclaration de variables servant pour la saisie
ret = scanf ("%d%c", &n, &c) ;  // saisie de l'entier et du caractère, sans espace
```

12^^a@ n = 12, c = '^', ret = 2 , après la saisie, tampon = ^a@
ab@ n = 1, c = 'e', ret = 0, après la saisie, tampon = ab@

```
int n; char c;          // déclaration de variables servant pour la saisie
scanf ("%d %c", &n, &c) ; // saisie de l'entier et du caractère, avec espace
```

12^@^a@ n = 12, c = 'a', après la saisie, tampon = @

```
int ret;
long int a = 10, b = 2;       // déclaration de variables servant pour la saisie
ret = scanf ("%ld %ld", &a, &b) ;      // saisie de 2 entiers exprimé en décimal
```

^^@8^73^@ a = 8, b = 73, ret = 2 après la saisie, tampon = ^@, Rq: ici l'espace ne change rien.

2.5 Rôle d'un caractère quelconque (différent d'un séparateur) dans le format

Tout caractère du paramètre *format* qui n'est pas un espace et qui ne fait pas partie d'une spécification de format entraîne la lecture du caractère courant du tampon :

- Si le caractère lu dans le tampon correspond au caractère du format, ce caractère est retiré du tampon et la fonction `scanf()` poursuit son travail.
- Sinon, il y a arrêt prématuré de la fonction `scanf()` (le caractère lu est alors laissé dans le tampon).

Exemple :

```
int j=0, m=0, ret;
printf("Saisir une date au format jj/mm : ");
ret = scanf ("%2d/%2d", &j, &m) ;         // saisie de la date
```

30/5@ j = 30, m = 5, ret = 2, après la saisie, tampon = @
125/6@ j = 12, m = 0, ret = 1, après la saisie, tampon = 5/6@
12^/5@ j = 12, m = 0, ret = 1, après la saisie, tampon = ^/5@
a12^/5@ j = 0, m = 0, ret = 0, après la saisie, tampon = a12^/5@

2.6 La macro getchar()

Prototype : `int getchar ();`

Cette fonction renvoie, sous forme entière, le caractère courant du tampon associé à l'entrée standard. Elle permet de lire n'importe quel caractère y compris les caractères séparateurs.

Avec c une variable de type char, l'expression :
`c = getchar()`
joue le même rôle que :
`scanf ("%c", &c)`

2.7 La fonction sscanf()

Prototype : `int sscanf(const char *ch, const char *format, ...);`

Son fonctionnement est similaire à celui de la fonction `scanf()`. Au lieu de lire ses données sur l'entrée standard (comme le fait `scanf()`), cette fonction les lit dans la chaîne de caractères `ch`.

Exemple : conversion d'une chaîne de caractères contenant une suite de chiffres en entier

```
int nb;
char Ligne[] = "269";
sscanf(Ligne, "%d", &nb);     // affecte 269 dans nb
```