Programmation en langage C

Allocation de la mémoire et utilisation des fonctions d'entrée/sortie vers un fichier

Plan du cours

1/ Allocation de la mémoire

En langage C, le programmeur a à sa charge la gestion de la mémoire dans le programme qu'il développe. Si le programmeur souhaite manier un ensemble de nombre flottants de taille *n*, variable, la seule possibilité que lui offre le langage C est l'allocation dynamique de la mémoire via la création d'un pointeur sur *float,* objet de type *float* *. C'est ce que fait la procédure suivante :

```
int n;
float *px;
n = 100;
px = (float *) malloc(n * sizeof(float));
```

Pour allouer dynamiquement de la mémoire on utilise les fonctions *calloc* ou *malloc*. Lorsque la mémoire allouée n'est plus utile il faut la libérer en utilisant la fonction *free*. Ces fonctions sont disponibles dans la bibliothèque standard *stdlib*. On trouve la description des fonctions d'allocation et de libération de la mémoire dans les pages de manuel de Linux. Voici un extrait (ne pas faire attention à la fonction *realloc* qui n'est pas au programme) :

```
NOM
       malloc,  calloc,  free, realloc - Allocation et libération
       dynamiques de mémoire.

SYNOPSIS
       #include <tstdlib.h>

       void *calloc (size_t nmemb, size_t size);
       void *malloc (size_t size);
       void free (void *ptr);
       void *realloc (void *ptr, size_t size);

DESCRIPTION
       calloc() alloue la  mémoire  nécessaire  pour  un  tableau
       nmemb  éléments, chacun d'eux représentant size octets, et
       renvoie un pointeur vers la mémoire allouée.   Cette  zone
       est remplie avec des zéros.

       malloc() alloue size octets, et renvoie un pointeur sur la
       mémoire allouée.  Le contenu de la zone de  mémoire  n'est
       pas initialisé.

       free()  libère  l'espace mémoire pointé par ptr, qui a été
       obtenu lors d'un appel antérieur à malloc(),  calloc()  ou
       realloc().  Si le pointeur ptr n'a pas été obtenu par l'un
       de ces appels, ou si il a déjà été libéré avec free(),  le
       comportement  est  indéterminé.   Si  ptr est NULL, aucune
       tentative de libération n'a lieu.

      [...]

VALEUR RENVOYÉE
       Pour calloc() et malloc(), la valeur renvoyée est un poin
       teur sur la mémoire allouée, qui est correctement  alignée
       pour  n'importe  quel  type  de  variable,  ou  NULL si la
       demande échoue.

       free() ne renvoie pas de valeur.

      [...]
```

Cet exemple permet au lecteur de se familiariser avec la présentation des pages de manuel (accessible par la ligne de commande *man <mot clé>*) de Linux. La première partie indique les noms des différentes fonctions qui sont décrites dans la page. La deuxième partie est le synopsis de chaque fonction. C'est ainsi qu'elles sont déclarées dans la bibliothèque – ici la bibliothèque *stdlib* – qui les implémente. Ensuite, on peut lire ce que chaque fonction fait et enfin on peut lire les informations sur la valeur renvoyée. Ces pages sont très détaillées, et dans l'extrait ci-dessus, des informations n'apparaissent pas.

Exercice 1 :

Créer une chaîne de caractères de taille suffisante pour pouvoir y écrire «Une chaîne de caractères» (attention au caractère de fin de chaîne '\0'). Libérer la mémoire allouée pour cette chaîne et réallouer un espace suffisant pour y écrire « Une chaîne de caractères plus longue».

Réponse à l'ex. 1 :

```
char *str = (char *) malloc(25*sizeof(char));
strcpy(str, "Une chaîne de caractères");
free(str);
str = (char *) malloc(37*sizeof(char));
strcpy(str, "Une chaîne de caractères plus longue");
```

Exercice 2 :

Créer dynamiquement un tableau *t* de pointeurs sur *int* de taille 10. Initialiser chaque pointeur *t[i],* du tableau en affectant à ce dernier un tableau d'entier de taille *i+2*. Initialiser toute la mémoire allouée avec les coefficients du binôme $C(i, j) = C_i^j$. (Rappel $C(i+1, j+1) = C(i,j)+C(i,j+1)$).

Réponse à l'ex. 2 :

```
int **t = (int **) malloc(10*sizeof(int *));
int i, j;
for (i = 0; i < 10; ++i)
  t[i] = (int *) malloc((i+2)*sizeof(int));
t[0][0] = t[0][1] = 1;
for (i = 1; i < 10; ++i)
  {
    t[i][0] = 1;
    for (j = 1; j < i+1; ++j)
      t[i][j] = t[i-1][j] + t[i-1][j];
    t[i][i-1] = 1;
  }
```

2/ Entrées / sorties

Le langage C est très utilisé pour communiquer avec les fichiers. La bibliothèque standard propose une large variété de fonctions d'entrées / sorties vers un fichier. En ce qui concerne le traitement d'entrées / sorties, on distingue souvent les fichiers texte des fichier binaires. Un fichier texte est une suite d'octets codant chacun un caractère de l'alphabet étendu. Un fichier binaire est tout sauf un fichier texte.

2.1/ Entrées / sorties formatées avec *fscanf* et *fprintf*

Pour lire et écrire des données dans un fichier texte, on utilise les fonction *fscanf* et *fprintf* :

```
int fscanf (FILE *stream, const char *format, ...);
int fprintf (FILE *stream, const char *format, ...);
```

La fonction *fscanf* analyse les entrées conformément à la chaîne *format*. Elle lit ses entrées depuis le flux pointé par *stream*. Les arguments pointeurs successifs (les « ... » dans la déclaration ci-dessus) doivent correspondre correctement aux indicateurs de conversion fournis dans la chaîne

format. Les indicateurs de conversion sont introduit par le caractère '%'.

La fonction *printf* produit des sorties en accord avec le format décrit par la chaîne *format*. Elle écrit sa sortie vers le flux de sortie pointé par *stream*. Les arguments qui suivent la chaîne *format* sont les éléments dirigés vers le flux de sortie et doivent correspondre à la chaîne *format*. Les fonctionnalités offertes grâce à l'utilisation de la chaîne *format* sont nombreuses et sont décrites en détails dans les pages de manuel de Linux (voir 4.2 et 4.3). Quelques exemples (exemples a/ à e/) valent mieux qu'un long discours pour montrer le principes et les fonctionnalités de base.

a) L'indicateur de conversion permettant de lire un entier est « i ». Ainsi, cet appel de *fscanf* :

```
int j;
fscanf(stream, "%i", &j);
```

fait la lecture d'un entier depuis le flux pointé par *stream* et stocke sa valeur dans l'entier *j* (*&j* est un pointeur vers un entier). Les autres indicateurs de conversions très utiles sont :

- nombre flottant en simple précision : 'f';
- caractère : 'c';
- chaîne de caractères : 's';

b) Chaque occurrence du caractère '%' dans la chaîne format indique le début d'une conversion. Ainsi cet appel de *fscanf* :

```
int j;
float x;
char str[256];
fscanf(stream, "%i %f %s", &j, &x, str);
```

fait la lecture d'un entier suivie de la lecture d'un nombre flottant en simple précision suivie de la lecture d'une chaîne de caractères. La présence d'au moins un caractère d'espacement entre chaque champ dans le fichier pointé par *stream* garantit que l'analyse se déroulera correctement.

c) On peut insérer dans la chaîne de format de conversion des caractères quelconques. Ces derniers devront être lus tels quels dans la chaîne à analyser pour que celle-ci se déroule correctement. Par exemple le code suivant :

```
fscanf(stream, "j=%i, x=%f, str=%s", &j, &x, str);
```

suppose que la chaîne à analyser commence, aux espacements près et aux valeurs choisies près, par ceci :

```
j=12,x=45.87, str=uneChaine
```

d) On peut insérer entre l'indicateur de conversion '%' et l'indicateur de type de conversion un nombre entier qui indique le nombre maximum de caractères à analyser. Ainsi l'appel :

```
char carray[10];
fscanf(stream, "%10c", carray);
```

limite le remplissage de *carray* à 10 caractères. Ainsi si le flux d'entrée commence par :

```
abcdefghijklm
```

carray sera égal à *'a'b'c'd'e'f'g'h'i'j'* (et sera dépourvu de caractère de fin de chaîne). Par contre, si le flux d'entrée commence par :

```
abcdefgh ijklm
```

carray sera égal à *'a'b'c'd'e'f'g'h'*, et les valeurs suivantes *carray[8]* et *carray[9]* ne seront pas initialisée. Si on remplace l'indicateur de conversion 'c' par 's', le caractère de fin de chaîne '\0' est ajouté après le dernier caractère lu. Le programmeur doit veiller à ce que l'espace mémoire réservé soit suffisant.

e) On peut utiliser derrière le '*%*' un champ du type *[sequence]*. La séquence désigne un ensemble de caractères auquel doit appartenir chaque caractère lu dans le flux d'entrée pour que la conversion continue. Ainsi le code suivant :

```
fscanf(stream, "%[0123456789]", str);
```

indique de mettre dans *str* les caractères du flux d'entrée tant qu'aucun espace n'est trouvé et tant que le caractère lu est un chiffre. Le tiret '-' permet de remplacer toute séquence de caractères contigus (au sens du codage acsii) par le caractère du début suivi du tiret suivi du caractère final. Ainsi *%[0123456789]* équivaut à *%[0-9]*. Le caractère '^' permet d'exclure au lieu d'inclure. Le code suivant :

```
fscanf(stream, "%[^a-z]", str);
```

met à la suite les uns des autres dans *str* tout caractère n'étant pas une lettre de l'alphabet minuscule. Ce type de formatage (*%[sequence]*) ajoute le caractère '\0' de fin de chaîne. On peut également faire précéder le crochet ouvrant de l'indicateur de taille maximum.

L'utilisation de la chaîne *format* est rigoureusement la même en écriture qu'en lecture si bien que tout ce qui vient d'être dit peut être utilisé avec *fprintf*. L'indicateur de taille précédant l'indicateur de type de conversion est particulièrement utile en écriture pour produire des sorties propres et alignées sur différentes colonnes. Supposons que l'on veuille ranger sur trois colonnes de largeur 20 caractères des nombres flottants. Voici ce qui peut être fait :

```
int x[10], y[10], z[10];
int i;

// initialisation de x, y, z
...
// fin d'initialisation
for (i = 0; i < 10; ++i)
  fprintf(stream, "%-20i%-20i%-20i\n", x[i], y[i], z[i]);
```

Le '-' précédant le nombre 20 impose l'alignement à gauche avec remplissage à droite avec des espaces. Sans le '-', le programme effectue un alignement à droite avec remplissage à gauche par

des zéros. On a le même comportement avec les autres type de conversions 'f', pour l'écriture d'un nombre flottant en notation classique, et 'e' pour l'écriture d'un nombre flottant en notation scientifique (avec exposant). Il est à noter que *fscanf* ne lit pas la notation scientifique.

2.2/ Entrées / sorties binaires avec *fread* et *fwrite*

Pour lire et écrire des données binaires, on utilise les fonctions *fread* et *fwrite*. Finalement ces fonctions sont beaucoup plus simples que *fscanf* et *fprintf* car il n'y a pas de formatage. Voici comment ces fonctions sont déclarées dans le fichier d'entête de la bibliothèque standard *stdio*.

```
size_t  fread  (void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite (const void *ptr, size_t size, size_t nmemb,FILE *stream);
```

La fonction *fread* lit depuis le flux pointé par *stream nmemb* éléments de données, chacun d'eux représentant *size* octets. Elle stocke dans le flux pointé par *stream* les valeurs lues dans *ptr*. La fonction *fwrite* écrit *nmemb* éléments de données chacun d'eux de taille *size* octets. *fread* et *fwrite* renvoient le nombre d'éléments correctement lus ou écrits. Si tout se passe bien, la valeur renvoyée est donc *nmemb*.

2.3/ Ouverture, fermeture et déplacement dans un fichier

Pour utiliser les fonctions de lecture et d'écriture dans un fichier, il faut mentionner l'existence des fonctions d'ouverture, de déplacement et de fermeture. Un flux d'entrée / sortie vers un fichier est manipulable en C à l'aide d'un pointeur *stream* de type *FILE* *. Pour faire pointer *stream* vers le flux nommé *fileName*, on utilise la fonction *fopen* :

```
FILE *stream;
stream = fopen(fileName, "r");
```

La fonction *fopen* est déclarée comme suit :

```
FILE *fopen (const char *path, const char *mode);
```

l'argument mode est le mode d'ouverture du fichier («r» = lecture seule, «r+» = lecture et écriture, «w» = écriture seule, «a» écriture à la fin du fichier). Le lecteur est renvoyé aux pages de manuel pour plus de détail sur le mode d'ouverture. Si l'ouverture échoue, la valeur spéciale *NULL* est renvoyée. La fonction *fclose* sert à fermer le fichier et est déclarée comme suit :

```
int fclose (FILE *stream);
```

Le lecteur est averti qu'après l'appel de :

```
fclose(stream);
```

stream ne pointe plus vers aucun flux, et tout appel de *fscanf*, *fprintf*, *fread*, *fwrite*, *fseek* générerait une erreur.

Pour se déplacer dans un fichier, la bibliothèque standard propose la fonction *fseek* :

```
int fseek (FILE *stream, long offset, int whence);
```

l'argument *whence* est la position de départ dans le fichier. Cet argument peut prendre les valeurs prédéfinies *SEEK_CUR,* qui indique la position courante dans le fichier à l'appel de *fseek*, *SEEK_SET*, qui indique le début du fichier et *SEEK_END*, qui indique la fin du fichier. L'argument *offset* est le déplacement signé en octets dans le fichier depuis la position de départ indiquée par *whence*.

3/ Exercice de synthèse

Dans cet exercice, on se propose d'implémenter la lecture et l'écriture d'un fichier image au format PGM.

a) Proposer une structure C permettant de traiter les images. Cette structure permettra d'allouer dynamiquement un tableau bidimensionnel de nombres flottants en simple précision. Le nombre de lignes, le nombre de colonnes et la taille du tableau devront être renseignés dans la structure.

b) Le format PGM est un format d'image bien reconnu par la plupart des logiciels d'imagerie sous Unix. Voici le format d'un fichier PGM. Un fichier PGM dit « raw » est constitué d'une entête ASCII suivie des octets de l'image parcourue ligne par ligne. L'entête a le format suivant :

```
P5
# ...
# ...
256 256
255
ligne 1
...
ligne 256
```

La première ligne est toujours P5, c'est le « magic number » de PGM « raw ». Viennent ensuite des lignes de commentaires optionnelles, commençant par #. La ligne suivante donne les dimensions – largeur et hauteur – de l'image. La dernière ligne doit normalement contenir 255 qui exprime la dynamique des valeurs. Ceci étant dit, proposer une fonction :

```
int read_pgm(image **img, const char *fileName);
```

qui effectue la lecture du fichier PGM *fileName*. La valeur renvoyée sera 0 si la lecture s'effectue normalement, -1 sinon.

c) Ecrire une fonction :

```
int write_pgm(image *img, const char *fileName);
```

qui permet l'écriture de l'image *img* au format PGM dans le fichier nommé *fileName*. La valeur renvoyée sera également 0 ou -1 selon que l'opération s'effectue correctement ou non.

d) Supposons qu'une image *img* soit créée à l'intérieur du programme principal.

- rechercher le minimum et le maximum de l'image et mémoriser ces valeurs dans les variables (de type *float*) *minValue*, *maxValue*;
- soit *nbEch* le plus petit entier supérieur ou égal à la racine carrée de la taille de l'image. Soit *Delta = (maxValue – minValue) / nbEch*. Calculer l'histogramme *h* de l'image, c'est à dire le graphe qui indique pour tout *k* compris entre *0* et *nbEch -1* le nombre de pixels de l'image dont la valeur est compris entre *minV + k Delta* et *maxV + k Delta*. *h* sera un tableau d'entiers de taille *nbEch;*
- créer alors un fichier texte «Histogram.dat» dans lequel vous écrirez les infirmations suivante :

```
#histogramme de l'image <nom de l'image>
#dynamique = [<minV>; <maxV>]
#<nbEch> échantillons
<minV>                  <h[0]>
<minV+Delta>            <h[1]>
...
<minV+(nbEch-1)*Delta>  <h[1]>
```

e) Ce format est compatible avec le format *gnuplot*. Dans le répertoire de «*Histogram.dat*» lancer le logiciel *gnuplot* en ligne de commande :

```
~$ gnuplot
```

puis lancer la ligne de commande *gnuplot* :

```
gnuplot> plot 'Histogram.dat' w lp
```

4 / Annexes

Cette partie reprend les descriptions complètes des fonctions utilisées dans ce cours données dans les pages de manuel de Linux traduites en français. On peut trouver ces pages à l'adresse : http://www.linux-france.org/article/man-fr/man3.

4.1/ Description détaillée de *malloc*, *calloc*, *free*

```
NOM
       malloc,  calloc,  free, realloc - Allocation et libération
       dynamiques de mémoire.

SYNOPSIS
       #include <tstdlib.h>

       void *calloc (size_t nmemb, size_t size);
       void *malloc (size_t size);
       void free (void *ptr);
       void *realloc (void *ptr, size_t size);

DESCRIPTION
       calloc() alloue la  mémoire  nécessaire  pour  un  tableau
       nmemb  éléments, chacun d'eux représentant size octets, et
       renvoie un pointeur vers la mémoire allouée.   Cette  zone
```

est remplie avec des zéros.

malloc() alloue *size* octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé.

free() libère l'espace mémoire pointé par *ptr*, qui a été obtenu lors d'un appel antérieur à malloc(), calloc() ou realloc(). Si le pointeur *ptr* n'a pas été obtenu par l'un de ces appels, ou si il a déjà été libéré avec free(), le comportement est indéterminé. Si *ptr* est NULL, aucune tentative de libération n'a lieu.

realloc() modifie la taille du bloc de mémoire pointé par *ptr* pour l'amener à une taille de *size* octets. realloc() conserve le contenu de la zone mémoire minimum entre la nouvelle et l'ancienne taille. Le contenu de la zone de mémoire nouvellement allouée n'est pas initialisé. Si *ptr* est NULL, l'appel de realloc() est équivalent à mal loc(size). Si *size* vaut zéro, l'appel est équivalent à free(ptr). Si *ptr* n'est pas NULL, il doit avoir été obtenu par un appel antérieur à malloc(), calloc() ou realloc().

VALEUR RENVOYÉE
Pour calloc() et malloc(), la valeur renvoyée est un poin teur sur la mémoire allouée, qui est correctement alignée pour n'importe quel type de variable, ou NULL si la demande échoue.

free() ne renvoie pas de valeur.

realloc() renvoie un pointeur sur la mémoire nouvellement allouée, qui est correctement alignée pour n'importe quel type de variable, et qui peut être différent de *ptr*, ou NULL si la demande échoue, ou si *size* vaut zéro. Si real loc() échoue, le bloc mémoire original reste intact, il n'est ni libéré ni déplacé.

CONFORMITÉ
ANSI-C

VOIR AUSSI
brk(2)

NOTES
Le standard Unix98 réclame que malloc(), calloc(), et realloc() positionne *errno* à ENOMEM en cas d'échec. La Glibc suppose qu'il en est ainsi (et les versions glibc de cette routine le font). Si vous utilisez une implémenta tion personnelle de malloc qui ne positionne pas *errno*, certaines routines de bibliothèques peuvent échouer sans donner de raison dans *errno*.

Lorsqu'un programme se plante durant un appel à malloc(), calloc() ou realloc(), ceci est presque toujours le signe d'une corruption du tas (zone de mémoire dans laquelle sont allouées les variables dynamiques). Ceci survient généralement en cas de débordement d'un bloc mémoire alloué, ou en libérant deux fois le même pointeur.

Les versions récentes de la bibliothèque C de Linux (libc

postérieures à 5.4.23) et la bibliothèque GNU libc 2.x incluent une implémentation de malloc() dont on peut con figurer le comportement à l'aide de variables d'environ nement. Quand la variable MALLOC_CHECK_ existe, les appels à malloc() emploient une implémentation spéciale, moins efficace mais plus tolérante à l'encontre des bugs simples comme le double appel de free() avec le même argu ment, ou un débordement de buffer d'un seul octet (bugs de surpassement d'une unité, ou oubli d'un caractère nul final d'une chaîne). Il n'est toutefois pas possible de pallier toutes les erreurs de ce type, et l'on risque de voir des fuites de mémoire se produire.

Si la variable MALLOC_CHECK_ vaut zéro, toutes les corrup tions du tas détectées sont ignorées silencieusement; Si elle vaut 1 un message de diagnostique est affiché sur *stderr*. Si cette variable vaut 2, la fonction abort() est appelée immédiatement. Ce comportement est partic ulièrement utile car un crash pourrait sinon se produire ultérieurement, et serait très difficile à diagnostiquer.

4.2/ La fonction *fscanf*

NOM

scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf - Entrées formatées.

SYNOPSIS

```
#include <tstdio.h>
int scanf (const char *format, ...);
int fscanf (FILE *stream, const char *format, ...);
int sscanf (const char *str, const char *format, ...);

#include <tstdarg.h>
int vscanf (const char *format, va_list ap);
int vsscanf (const char *str, const char *format, va_list ap);
int vfscanf (FILE *stream, const char *format, va_list ap);
```

DESCRIPTION

Les fonctions de la famille scanf analysent leurs entrées conformément au *format* décrit plus bas. Ce format peut contenir des *indicateurs de conversion*. Les résultats des conversions, s'il y en a, sont stockés dans des arguments *pointeurs*. La fonction scanf lit ses données depuis le flux d'entrée standard *stdin*, fscanf lit ses entrées depuis le flux pointé par *stream*, et sscanf lit ses entrées dans la chaîne de caractères pointée par *str*.

La fonction vfscanf est analogue à vfprintf(3) et lit ses arguments depuis le flux pointé par *stream* en utilisant une liste variable d'arguments pointeurs, voir stdarg(3). La fonction vscanf examine l'entrée standard en utilisant une liste variable d'arguments pointeurs et la fonction vsscanf examine une chaîne. Elles sont respectivement ana logues aux fonctions vprintf et vsprintf.

Les arguments *pointeurs* successifs doivent correspondre correctement aux indicateurs de conversion fournis (voir néanmoins l'attribut '*' plus bas). Toutes les conver sions sont introduites par le caractère % (symbole pour cent). La chaîne *format* peut également contenir d'autres

caractères. Les blancs (comme les espaces, les tabulations ou les retours chariots) dans la chaîne *format* correspondent à un nombre quelconque de blancs (et même aucun) dans la chaine d'entrée. Tous les autres caractères ne peuvent correspondre qu'a eux-même. L'examen de l'entrée s'arrête dès qu'un caractère d'entrée ne correspond pas à un caractère du format. L'examen s'arrête également quand une conversion d'entrée est impossible (voir ci-dessous).

CONVERSIONS

A la suite du caractère % introduisant une conversion, il peut y avoir un nombre quelconque de caractères *attributs* de la liste suivante :

* Ne pas stocker le résultat. La conversion est bien effectuée comme d'habitude, mais le resultat est éliminé au lieu d'être memorisé dans un pointeur.

a Indique que la conversion sera de type s, la mémoire nécessaire pour la chaine sera allouée avec malloc(3) et le pointeur sera assigné à la variable de type *char* qui n'a pas besoin d'être initialisée auparavant. Cet attribut n'existe pas en *C ANSI*.

h Indique que la conversion sera de type dioux ou n et que le pointeur suivant est un pointeur sur un *short int* (plutôt que sur un *int*).

l Indique que la conversion sera de type dioux ou n et que le pointeur suivant est un pointeur sur un *long int* (plutôt que sur un *int*), ou que la conversion sera de type efg et que le pointeur suivant est un pointeur sur un *double* (plutôt que sur un *float*). Indiquer deux attributs l successifs est équivalent à indiquer l'attribut L.

L Indique que la conversion sera de type efg et que le pointeur suivant est un pointeur sur un *long double* ou que la conversion sera de type dioux et que le pointeur suivant est un pointeur sur un *long long*. (ce type n'existe pas en *C ANSI*. Un programme l'utilisant ne sera pas portable sur toutes les machines).

q est équivalent à L. Cet attribut n'existe pas en *C ANSI*.

En plus de ces attributs peut se trouver un champ optionnel de longueur maximale, exprimée sous forme d'entier, entre le caractère % et l'indicateur de conversion. Si aucune longueur n'est donnée, une valeur infinie est utilisée par défaut (avec une exception, voir plus bas). Autrement, la conversion examinera au plus le nombre de caractères indiqués. Avant que les conversions ne commencent, la plupart d'entre elles éliminent les blancs. Ces espaces blancs ne sont pas comptés dans le champ de largeur maximale.

Les conversions suivantes sont disponibles :

% Correspond à un caractère `%'. Ceci signifie qu'un indicateur `%%' dans la chaîne de format cor

respond à un seul caractere `%' dans la chaine d'entrée. Aucune conversion, et aucune assignation n'a lieu.

d Correspond à un entier décimal éventuellement signé, le pointeur correspondant doit être du type *int **.

D Equivalent à ld, utilisé uniquement pour compati bilité avec des versions précédentes. (Et seule ment dans libc4. Dans libc5 et glibc le %D est ignoré silencieusement, ce qui conduit d'anciens programmes à échouer mystérieusement).

i correspond à un entier éventuellement signé. Le pointeur suivant doit être du type *int **. L'entier est en base 16 (héxadécimal) s'il commence par `0x' ou `0X', en base 8 (octal) s'il commence par un `0', et en base 10 sinon. Seuls les caractères correspondants à la base concernée sont utilisés.

o Correspond à un entier octal non signé. Le pointeur correspondant doit être du type *unsigned int **.

u Correspond à un entier décimal non signé. Le poin teur suivant doit être du type *unsigned int **.

x Correspond à un entier héxadécimal non signé. Le pointeur suivant doit être du type *unsigned int **.

X Equivalent à x

f Correspond à un nombre réel éventuellement signé. Le pointeur correspondant doit être du type *float **.

e Equivalent à f.

g Equivalent à f.

E Equivalent à f

s Correspond à une séquence de caractères différents des caractères blancs. Le pointeur correspondant doit être du type *char **, et la chaine doit être assez large pour accueillir toute la séquence, ainsi que le caractère NUL final. La conversion s'arrête au premier caractère blanc, ou à la longueur maximale du champ.

c Correspond à une séquence de *width* caractères (par défaut 1). Le pointeur associé doit être du type *char **, et il doit y avoir suffisament de place dans la chaîne pour tous les caracteres. Aucun car actère NUL final n'est ajouté. Les caractères blancs de début ne sont pas supprimés. Si on veut les éliminer, il faut utiliser un espace dans le format.

[Correspond à une séquence non vide de caractères appartenant à un ensemble donné. Le pointeur corre spondant doit être du type *char **, et il doit y

avoir suffisament de place dans le tableau de car actères pour accueillir la chaîne ainsi qu'un car actère NUL final. Les caractères blancs du début ne sont pas supprimés. La chaîne est constituées de caractères inclus ou exclus d'un ensemble donné. L'ensemble est composé des caractères compris entre les deux crochets [et]. L'ensemble *exclut* ces caractères si le premier après le crochet ouvrant est un accent circonflexe ^. Pour inclure un cro chet fermant dans l'ensemble, il suffit de le placer en première position après le crochet ouvrant, ou l'accent circonflexe ; à tout autre emplacement il servira à terminer l'ensemble. Le caractère tiret - a également une signification particulière. Quand il est placé entre deux autres caractères, il ajoute à l'ensemble les caractères intermédiaires. Pour inclure un tiret dans l'ensem ble, il faut le placer en dernière position avant le crochet fermant.Par exemple, `[^]0-9-]' corre spond à l'ensemble `Tout sauf le crochet fermant, les chiffres de 0 à 9, et le tiret'. La chaine se termine dès l'occurence d'un caractère exclus (ou inclus s'il y à un accent circonflexe) de l'ensem ble, ou dès qu'on atteint la longueur maximale du champ.

p Correspond à une valeur de pointeur (comme affichée par `%p' dans printf(3). Le pointeur correspondant doit être du type *void **.

n Aucune lecture n'est faite. Le nombre de caractères déjà lus est stocké dans le pointeur correspondant, qui doit être de type *int **. Ce n'est *pas* une con version, mais le stockage peut quand même être sup primé avec un attribut *. Le standard C indique : `L'exécution d'une directive %n n'incrémente pas le compteur d'assignations renvoyé à la fin de l'exécution'. Mais il semble qu'il y ait des con tradictions sur ce point. Il est probablement sage de ne pas faire de suppositions sur l'effet de la conversion %n sur la valeur renvoyée.

VALEUR RENVOYÉE

Ces fonctions renvoient le nombre d'éléments d'entrées correctement assignés. Ce nombre peut être plus petit que le nombre d'élements attendus, et même être nul, s'il y a une erreur de mise en correspondance. La valeur zéro indique qu'aucune conversion n'a été faite bien que des caractères étaient disponibles en entrée. Typiquement c'est un caractère d'entrée invalide qui en est la cause, par exemple un caractère alphabétique dans une conversion `%d'. La valeur EOF est renvoyée si une erreur d'entrée a eu lieu avant toute conversion, par exemple une fin de fichier. Si une erreur fin-de-fichier se produit après que les conversions aient commencé, le nombre de conversions réussies sera renvoyé.

VOIR AUSSI

strtol(3), strtoul(3), strtod(3), getc(3), printf(3)

STANDARDS

Les fonctions fscanf, scanf, et sscanf sont conformes à ANSI C3.159-1989 (``C ANSI'').

L'attribut q est une notation *BSD 4.4* pour *long long*, alors que ll ou l'utilisation de L dans les conversions entières sont des notations GNU.

Les versions Linux de ces fonctions sont basées sur la bibliothèque *libio GNU*. Jetez un oeil sur la documentation *info* de la *libc GNU (glibc-1.08)* pour une description complète.

BUGS

Toutes ces fonctions sont totalement conformes à ANSI C3.159-1989, mais lui ajoutent les attributs q et a ainsi que des comportements supplémentaires des attributs L et l. Ce derniers doivent être considérés comme des bugs, car ils modifient le comportement d'attributs définis dans ANSI C3.159-1989.

Certaines combinaisons d'attributs n'ont pas de sens en *C ANSI* (par exemple %Ld). Bien qu'elles aient un comportement bien défini sous Linux, ce n'est peut être pas le cas sur d'autres architectures. Il vaut donc mieux n'utiliser que des attributs définis en *C ANSI*, par exemple, utilisez q à la place de L avec les conversions diouxX ou ll.

L'utilisation q n'est pas la même sous *BSD 4.4*, car il peut être utilisé avec des conversions de réels de manière équivalente à L.

[NDT] La conversion %s devrait toujours être accompagnée d'une longueur maximale de chaîne de caractères. En effet, il existe un risque de débordement de buffer, qui peut conduire à un trou de sécurité important dans un programme Set-UID ou Set-GID.

4.3/ La fonction *fprintf*

NOM

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf - Formatage des sorties.

SYNOPSIS

```
#include <tstdio.h>

int printf (const char *format, ...);
int fprintf (FILE *stream, const char *format, ...);
int sprintf (char *str, const char *format, ...);
int snprintf (char *str, size_t size, const char  *format,
...);

#include <tstdarg.h>

int vprintf (const char *format, va_list ap);
int  vfprintf  (FILE  *stream, const char *format, va_list
ap);
int vsprintf (char *str, const char *format, va_list ap);
int vsnprintf (char *str, size_t size, const char *format,
va_list ap);
```

DESCRIPTION

Les fonctions de la famille printf produisent des sorties en accord avec le *format* décrit plus bas. Les fonctions printf et vprintf écrivent leur sortie sur *stdout*, le flux de sortie standard. fprintf et vfprintf écrivent sur le flux *stream* indiqué. sprintf, snprintf, vsprintf et vsnprintf écrivent leurs sorties dans la chaîne de car actères *str*.

Ces fonctions créent leurs sorties sous le contrôle d'une chaîne de *format* qui indique les conversions à apporter aux arguments suivants (ou accessibles à travers les argu ments de taille variable de stdarg(3)).

Ces fonctions renvoient le nombre de caractères imprimés, sans compter le caractère nul `\0' final dans les chaînes. snprintf et vsnprintf n'écrivent pas plus de *size* octets (y compris le '\0' final), et renvoient -1 si la sortie a été tronquée à cause de cette limite.

La chaîne de format est composée d'indicateurs : les car actères ordinaires (différents de %), qui sont copiés sans modification sur la sortie, et les spécifications de con version. Les spécifications de conversion sont intro duites par le caractère %. Les arguments doivent corre spondre correctement (après les promotions de types) avec les indicateurs de conversion.

Après le caractère %, les éléments suivant doivent apparaître, dans l'ordre :

· Zéro ou plusieurs attributs :

indique que la valeur doit être convertie en une autre forme. Pour les conversions c, d, i, n, p, s, et u cette option n'a pas d'effet. Pour la conversion o la précision est incrémentée pour forcer le premier car actère de la chaîne de sortie à valoir 0 (sauf si une valeur nulle est imprimée avec une précision explicite de 0). Pour les conversions x et X une valeur non nulle reçoit le préfixe `0x' (ou `0X' pour l'indi cateur X). Pour les conversions e, E, f, g, et G le résultat contiendra toujours un point décimal même si aucun chifre ne le suit (normalement, un point décimal n'est présent avec ces conversions que si des décimales le suivent). Pour les conversions g et G les zéros en tête ne sont pas éliminés, contrairement au comportement habituel.

0 indique le remplissage avec des zéros. Pour toutes les conversions sauf n, la valeur est complétée à gauche avec des zéros plutot qu'avec des espaces. Si une précision est fournie avec une conversion numérique (d, i, o, u, i, x, et X), l'attribut 0 est ignoré.

- (un attribut de largeur négatif) indique que la valeur doit être justifiée sur la limite

gauche du champ. Sauf pour la conversion n, les valeurs sont complétées à droite par des espaces, plutôt qu'a gauche par des zéros ou des blancs. Un attribut - surcharge un attribut 0 si les deux sont fournis.

´ ´ (un espace) indique qu'un espace doit être laissé avant un nombre positif produit par une conversion signée (d, e, E, f, g, G, ou i).

\+ indique que le signe doit toujours être imprimé avant un nombre produit par une con version signée. Un attribut + surcharge un attribut 'espace' si les deux sont fournis.

' indique que les chiffres d'un argument numérique doivent être groupés en fonction de la localisation. Remarquez que de nom breuses versions de gcc n'accepte pas cet attribut et déclencheront un avertissement (warning).

- Un nombre optionnel, indiquant une largeur minimale de champ. Si la valeur convertie occupe moins de caractères que cette largeur, elle sera complétée par des espaces à gauche (ou à droite si l'attribut d'alignement à gauche a été fourni).

- Une précision eventuelle, sous la forme d'un point (`.') suivi par un nombre. Si ce nombre est absent, la précision est fixée à 0. Cette précision indique un nombre minimum de chiffres à faire apparaître lors des conversions d, i, o, u, x, et X, le nombre de décimales à faire apparaître pour les conversions e, E, et f, le nombre maximum de chiffres significatifs pour g et G, et le nombre maximum de caractères à imprimer pour la conversion s.

- Le caractère optionnel h, indiquant que la conver sion d, i, o, u, x, ou X suivante correspond à un argument *short int* ou *unsigned short int*, ou que la conversion n suivante correspond à un argument pointeur sur un *short int*.

- Le caractère optionnel l (elle) indiquant que la conversion d, i, o, u, x, ou X suivante s'applique à un argument *long int* ou *unsigned long int*, ou que la conversion n suivante correspond à un pointeur sur un *long int*. Linux fournit une possibilité, non conforme ANSI, d'utiliser deux attributs l comme synonyme à q ou L. Ainsi ll peut être utilisé avec les conversions de nombres réels. Cette méthode est néanmoins fortement déconseillée.

- Le caractère L indiquant que la conversion e, E, f, g, ou G suivante correspond à un argument *long dou ble*, ou que la conversion d, i, o, u, x, ou X suiv ante correspond à un argument *long long*. Remarquez que *long long* n'est pas spécifié par *ANSI C* et n'est donc pas portable sur toutes les architec

tures.

· Le caractère optionnel q. Il est équivalent à L. Voir les sections STANDARDS et BUGS pour des détails sur l'utilisation de ll, L, et q.

· Un caractère Z indiquant que la conversion d'entier suivante (d, i, o, u, i, x, et X), correspond à un argument *size_t*.

· Un caractère indiquant le type de conversion à appliquer.

Le champ de largeur ou de précision, ou les deux, sont parfois indiqués par un astérisque `*' à la place d'un nombre. Dans ce cas, un argument *int* fournit la valeur du champ largeur ou précision. Un champ de largeur négative est traité de manière identique à l'argument d'ajustement à gauche avec une largeur positive. Une précision négative est ignorée.

Les indicateurs de conversion, et leurs significations sont :

diouxX L'argument *int* (ou une variante appropriée) est convertie en un chiffre décimal signé (d et i), un chiffre octal non-signé (o), un chiffre décimal non-signé (u), un chiffre héxadécimal non-signé (x et X). Les lettres abcdef sont utilisées pour les conversions avec x, les lettres ABCDEF sont utilisées pour les conversions avec X. La précision, si elle est indiquée, donne un nombre minimal de chiffres à faire apparaître. Si la valeur convertie nécessite moins de chiffres, elle est complétée à gauche avec des zéros.

eE L'argument réel, de type *double*, est arrondi et présenté avec la notation scientifique [-]9.999e99 dans lequel se trouve un chiffre avant le point, puis un nombre de décimales égal à la précision demandée. Si la précision n'est pas indiquée, l'affichage contiendra 6 décimales. Si la précision vaut zéro, il n'y a pas de point décimal. Une con version E utilise la lettre E (plutôt que e) pour introduire l'exposant. Celui-ci contient toujours au moins deux chiffres. Si la valeur affichée est nulle, son exposant est 00.

f L'argument réel, de type *double*, est arrondi, et présenté avec la notation classique [-]999.999, où le nombre de décimales est égal à la précision réclamée. Si la précision n'est pas indiquée, l'affichage se fera avec 6 décimales. Si la précision vaut zéro, aucun point n'est affiché. Lorsque le point est affiché, il y a toujours au moins un chiffre devant.

g L'argument réel, de type *double*, est converti en style f ou e (ou E pour la conversion G) La précision indique le nombre de décimales significa tives. Si la précision est absente, une valeur par défaut de 6 est utilisée. Si la précision vaut 0,

elle est considérée comme valant 1. La notation scientifique e est utilisée si l'exposant est inférieur à -4 ou supérieur ou égal à la précision démandée. Les zéros en fin de partie décimale sont supprimés. Un point decimal n'est affiché que s'il est suivi d'au moins un chiffre.

c L'argument entier, de type *int*, est converti en un *unsigned char*, et le caractère correspondant est affiché.

s L'argument de type ``*char **'' est supposé être un pointeur sur un tableau de caractères (pointeur sur une chaîne). Les caractères du tableau sont écrits jusqu'au caractère NUL final, non compris. Si une précision est indiquée, seul ce nombre de car actères sont écrits. Si une précision est fournie, il n'y a pas besoin de caractère nul. Si la précision n'est pas donnée, ou si elle est supérieure à la longueur de la chaîne, le caractère NUL final est nécessaire.

p L'argument pointeur, du type ``*void **'', est affiché en héxadécimal, comme avec %#x ou %#lx.

n Le nombre de caractères déjà écrits est stocké dans l'entier indiqué par l'argument pointeur de type ``*int **''. Aucun argument n'est converti.

% Un caractère `%' est écrit. Il n'y a pas de conver sion. L'indicateur complet est `%%'.

En aucun cas une petite largeur de champ ne causera une troncature d'un champ. Si le résultat de la conversion est plus grand que le champ prévu, celui-ci est étendu pour contenir le résultat.

EXEMPLES

Pour afficher une date et une heure sous la forme `Dimanche 10 Novembre, 23:15', ou *jour_semaine* et *mois* sont des pointeurs sur des chaînes :

```
#include <tstdio.h>
fprintf (stdout, "%s %d %s, %.2d:%.2d\n",
      jour_semaine, jour, mois, heure, minute);
```

Pour afficher Pi avec 5 décimales :

```
#include <tmath.h>
#include <tstdio.h>
fprintf (stdout, "pi = %.5f\n", 4 * atan(1.0));
```

Pour allouer 128 octets de chaînes de caractères, et écrire dedans :

```
#include <tstdio.h>
#include <tstdlib.h>
#include <tstdarg.h>
char *
newfmt (const char *fmt, ...)
{
  char *p;
  va_list ap;
  if ((p = malloc(128)) == NULL)
```

```
                  return (NULL);
                va_start(ap, fmt);
                (void) vsnprintf(p, 128, fmt, ap);
                va_end(ap);
                return (p);
              }
```

VOIR AUSSI
printf(1), scanf(3)

STANDARDS
Les fonctions fprintf, printf, sprintf, vprintf, vfprintf, et vsprintf sont conformes à ANSI C3.159-1989 (``ANSI C'').

L'attribut q est une notation de *BSD 4.4* pour *long long*, alors que ll ou l'utilisation de L pour les conversions d'entiers est une extension GNU.

Les versions Linux de ces fonctions sont basées sur la bibliotheque *libio GNU*. Jetez un oeil à la documentation *info* de la *libc (glibc-1.08) GNU* pour une description plus précise.

BUGS
Certaines conversions de nombres réels, sous Linux, peu vent causer des fuites de mémoire.

Toutes les fonctions sont totalement conformes à ANSI C3.159-1989, mais proposent des attributs q, Z et ' supplémentaires, ainsi qu'un comportement additionnel pour les attributs L et l. Ces derniers comportements doivent être considérés comme des bugs, car ils modifient des attributs définis par ANSI C3.159-1989.

L'effet d'ajustement du format %p avec des zéros (soit avec l'attribut 0, soit en indiquant une précision), et l'effet bénin (en clair : aucun) de l'attribut # sur les conversions %n et %p sont non standards. De telles combi naisons doivent être évitées.

Certaines combinaisons d'attributs *ANSI C* n'ont pas de sens (par exemple %Ld). Bien qu'elles aient un comporte ment bien défini sous Linux, ce n'est pas obligatoirement le cas sur d'autres architectures. Il vaut mieux n'utiliser que les attributs définis par *ANSI C*, par exem ple, l'utilisation de q est preferable à L pour les con versions diouxX ou ll.

L'utilisation de l'attribut q n'est pas la même que sous *BSD 4.4*, où il peut être utilisé pour les conversions de nombres réels de manière identique à L.

Comme sprintf et vsprintf ne font pas de suppositions sur la longueur des chaînes, le programme appelant doit s'assurer de ne pas déborder l'espace d'adressage. C'est souvent difficile.

4.4/ Les fonctions *fread* et *fwrite*

NOM

fread, fwrite - Entrées/sorties binaires sur un flux.

SYNOPSIS

```
#include <tstdio.h>

size_t fread (void *ptr, size_t size, size_t nmemb, FILE
*stream);

size_t fwrite (const void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

DESCRIPTION

La fonction fread lit *nmemb* éléments de données, chacun d'eux représentant *size* octets de long, depuis le flux pointé par *stream*, et les stocke à l'emplacement pointé par *ptr*.

La fonction fwrite écrit *nmemb* éléments de données, chacun d'eux représentant *size* octet de long, dans le flux pointé par *stream*, après les avoir lus depuis l'emplacement pointé par *ptr*.

VALEUR RENVOYÉE

fread et fwrite renvoient le nombre d'éléments correcte ment lus ou écrits (et non pas le nombre d'octets). Si une erreur se produit, ou si la fin du fichier est atteinte en lecture, le nombre renvoyé est plus petit que *nmemb* et peut même être nul.

fread traite la fin du fichier comme une erreur, et l'appelant devra appeler feof(3) ou ferror(3) pour dis tinguer ce cas.

VOIR AUSSI

feof(3), ferror(3), read(2), write(2)

CONFORMITÉ

Les fonctions fread et fwrite sont conformes à ANSI C3.159-1989 (``ANSI C'').