Mémento sélectif de fonctions du langage C

version 0.2

14 novembre 2019

Table des matières

1 Introduction

Ceci n'est pas, et de loin, un répertoire complet des fonctions du C standard. Seules les fonctions les plus utiles pour le cours, les TP, le projet et l'examen sont répertoriées.
Les fonctions d'entrée/sortie `printf, fprintf, scanf, fscanf` ne sont pas incluses. Elles méritent un guide à part. Les fonctions mathématiques de `math.h` sont absentes aussi. De même, toutes les fonctions qui n'ont pas été et ne seront pas vues en cours sont absentes.
Pour aller plus loin (pour la programmation système), une source inestimable est The Single UNIX Specification `http://www.unix.org/version4/` (hélas, il faut s'enregistrer pour lire ou récupérer, mais c'est gratuit).

2 Quelques types

```
#include <stddef.h>
size_t
ptrdiff_t
```

`size_t` est un type entier non-signé utilisé souvent pour définir la taille d'un objet. Par exemple, la valeur retournée par `sizeof()` est de type `size_t`.

`ptrdiff_t` est un type entier signé, la différence de deux pointeurs est de type `ptrdiff_t`.

```
#include <stdint.h>
int8_t
int16_t
int32_t
int64_t
intmax_t

uint8_t
uint16_t
uint32_t
uint64_t
uintmax_t
```

Les types ci-dessus définissent les entiers, respectivement signés et non signés, dont le nombre de bits (8, 16, 32, 64) est fixé. (Les nombres entiers de 64 bits peuvent ne pas être implémentés.)
`intmax_t` et `unintmax_t` sont les types entiers signé/non signé de largeur maximale.
Tous ces types apparaissent en C99.

3 Fonctions de gestion de la mémoire : memchr, memcpy, memmove, memcmp, memset

```
#include <string.h>
void *memchr( const void *ptr, int val, size_t len)
int   memcmp(const vois *ptr1, const void *ptr2, size_t len)
void *memcpy(void *dest, const void *src, size_t len)
void *memmove(void *dest, const void *src, size_t len)
void *memset(void *ptr, int val, size_t len)
```

`memchr()` cherche la première occurrence du caractère `val` dans la zone de `len` octets commençant à l'adresse `ptr`.
La fonction retourne le pointeur vers le premier caractère `val` retrouvé, ou `NULL` si `val` n'a pas été trouvé. Chaque caractère `c` dans la zone mémoire recherchée est comparé à `val` par l'expression

```
(unsigned char) val == (unsigned char) c
```

La fonction `memcmp` compare un à un `len` caractères dans les deux zones mémoire d'adresse respective `ptr1` et `ptr2`. Elle retourne une valeur strictement négative si la suite de `len` caractères à l'adresse `ptr1` est inférieure (dans l'ordre lexicographique) à celle à l'adresse `ptr2`, 0 si les deux suites sont égales, et un entier stricement positif si la première suite est supérieure à la deuxième.

Les fonctions `memcpy` et `memmove` copient `len` octets de l'adresse `src` vers l'adresse `dest`. La seule différence entre les deux fonctions est que `memcpy` ne doit pas être utilisé si les deux zones mémoire se chevauchent. En cas de doute, utilisez `memmove` (ou, pour éviter des problèmes, utilisez toujours [1] `memmove`). Les deux fonctions retournent la valeur `dest`.

La fonction `memset` copie le caractère `val` sur `len` octets à partir de l'adresse `ptr`. Avant d'être copiée, la valeur `val` est transformée en `unsigned char`.

4 Fonctions d'allocation de mémoire

```
#include <stdlib.h>
void *malloc( size_t size )
void *calloc( size_t elt_count, size_t elt_size)
void *realloc(void *ptr, size_t size)
```

`malloc` alloue une zone de mémoire de taille `size` octets et retourne un pointeur vers le premier octet de la mémoire allouée. `malloc` garantit que l'adresse retournée est alignée de façon à ce que n'importe quel objet puisse être mis à cette adresse indépendamment de l'alignement exigé pour cet objet [2]. La mémoire allouée n'est pas initialisée de quelque façon que ce soit.

1. Cependant `memcpy` peut être plus rapide que `memmove`.
2. En clair, il n'est par exemple pas nécessaire de se demander s'il est possible d'y mettre un objet `int` sur une architecture qui demande que l'adresse de `int` soit multiple de 4. Et cette remarque s'étend à tout type d'objet.

`malloc` retourne `NULL` en cas d'échec.

`calloc` alloue la mémoire suffisante pour stocker `elt_count` objets de taille `elt_size` chacun. Comme `malloc`, `calloc` retourne un pointeur vers la mémoire allouée, ou NULL en cas d'échec. Tous les octets de la mémoire allouée par `calloc` sont initialisés avec la même valeur 0 (tous les bits 0).

La fonction `realloc` prend un pointeur `ptr` avec une adresse de mémoire allouée par `malloc, calloc` ou `realloc` et modifie la taille de la mémoire allouée tout en préservant le contenu.
Il est important de noter que `realloc` peut allouer une nouvelle zone de mémoire en y recopiant les octets de l'adresse `ptr` ; `realloc` retourne un pointeur vers la nouvelle zone de mémoire.
Si le pointeur retourné par `realloc` est différent de `ptr`, alors la mémoire à l'adresse `ptr` a été libérée par `realloc` et ne doit plus être utilisée (en particulier vous **ne devez pas** faire `free(ptr)`.)
Si le premier argument de `realloc` est `NULL`, alors `realloc` se comporte comme `malloc`.
Si `size` est plus petit que la taille de l'ancienne mémoire à l'adresse `ptr` seuls les `size` premiers octets sont recopiés dans la nouvelle mémoire allouée par `realloc`.
Exemple. Ajout de **INCREMENT** éléments à un tableau `tab` de `nb` éléments `int` alloué auparavant par `malloc`.

```
int *p=realloc(tab, (nb+INCREMENT) *sizeof(int) );
if( p == NULL ){
   fprintf(stderr, "agrandissement echoue\n");
}else{
 tab = p;
 nb += INCREMENT;
 ...
}
```

`free` libère la mémoire allouée par `malloc`, `calloc` ou `realloc`. L'argument de `free` doit être un pointeur retourné précédemment par une de ces fonctions.

5 Fonctions de traitement de chaînes de caractères : strlen, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strchr, strrchr, strspn, strcspn, strstr, strrchr, strpbr, strtok

Par « string », on comprend dans cette section une suite de caractères terminant par le caractère nul `'\0'` (j'utilise « string » pour éviter d'écrire chaque fois la longue expression « chaîne de caractères »).
Le paramètre `char *` de toutes ces fonctions doit être un string

```
#include <string.h>
size_t strlen( const char *s )
char * strcat( char *dest, const char *src )
char * strncat(char *dest, const char *src, size_t n)
int    strcmp( const char *s1, const char *s2)
int    strncmp(const char *s1, const char *s2, size_t n)
char * strchr( const char *s, int c)
char * strrchr(const char *s, int c)
size_t strspn( const char *s, const char *set )
size_t strcspn(const char *s, const char *set )
char * strpbrk(const char *s, const char *set )
char * strtok( char *str, const char *sep)
```

`strlen` retourne la longueur du string `s` (**sans compter le caractère nul final**).

`strcat` copie le string `src` à la suite du string `dest`. Le caractère nul à la fin du `dest` sera remplacé par le premier caractère de `src`. Le résultat est un string, donc le caractère nul est mis à la fin. La mémoire à l'adresse `dest` doit être suffisamment grande pour stocker les caractères de deux strings et le caractère nul. Cette fonction peut conduire à un débordement de tampon ; la fonction `strncat` est donc préférable.

`strncat` copie au plus `n` caractères du string `src` à la fin du string `dest`. Le caractère nul à la fin du string `dest` sera remplacé par le premier caractère de `src`. S'il n'y a pas de caractère nul parmi les `n` premiers caractères de `src`, alors tous les `n` caractères de `src` sont ajoutés à la fin de `dest`. S'il y a un caractère nul parmi les `n` premiers caractères de `src`, alors seuls les caractères jusqu'au premier caractère nul sont copiés. À la fin du nouveau string obtenu, `strncat` place le caractère nul.

`strcmp` compare les strings `s1` et `s2` et retourne une valeur strictement négative si `s1` précède `s2` (dans l'ordre lexicographique), 0 si `s1` et `s2` sont identiques, strictement positive si `s1` est plus grand que `s2`.

`strncmp` fonctionne comme `strcmp`, mais compare au maximum les `n` premiers caractères de `s1` et `s2`. En particulier, si les deux string ont au moins `n` caractères, et que les `n` premiers caractères sont identiques, alors la fonction retourne 0.

`strchr` cherche la première occurrence de `c` (converti en `char`) dans `s` et retourne un pointeur vers ce caractère (ou NULL si `s` ne contient pas `c`). `strrchr` fonctionne comme `strchr` mais cherche la dernière occurrence de `c` dans `s`.

`strspn` cherche dans `s` la première occurrence d'un caractère qui ne fait pas partie du string `set`. `strspn` retourne la longueur du plus long préfixe de `s` composé de caractères qui sont dans `set`. En particulier si `s` contient uniquement les caractères de `set`, alors `strspn` retourne la longueur de `s`.
`strcspn` est semblable à `strspn` mais cherche dans `s` la première occurrence d'un caractère qui se trouve dans le string `set` et retourne la longueur du plus long préfixe de `s` composé de caractères qui ne font pas partie de `set`.

`strpbrk` est similaire à `strspn`, mais `strpbrk` retourne le pointeur vers le premier caractère de `s` qui fait partie de `set`.

`strtok` permet de découper le string `str` en lexèmes. Notez que `str` est modifié par `strtok` (qui mettra le caractère nul à la fin de chaque lexème).
On suppose que le lexème est la plus longue suite de caractères qui ne contient aucun caractère qui se trouve dans `sep`. Intuitivement `sep` donne les caractères qui séparent les lexèmes.
Le string `str` à découper est passé comme paramètre au premier appel à `strtok`. Pour les appels suivants, le paramètre `str` doit être `NULL`. Si `strtok` ne trouve plus de lexèmes, alors la fonction retourne `NULL`. Le paramètre `sep` peut être différent d'un appel à l'autre.
Exemple :

```
char *s = " plus grand que 0.  Pour les appels suivants  ";
char *sep=" .";
char *lex = strtok(s, sep);
if(lex != NULL){
  printf("%s\n",lex);
}
while( ( lex = strtok(NULL, sep) ) != NULL ){
    printf("%s\n",lex);
}
```

imprimera

```
plus
grand
que
0
Pour
les
appels
suivants
```

6 Fonctions de test et conversion de caractères

6.1 Test de type de caractères : isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isupper, isprint, ispunct, isspace, isxdigit

```
#include <ctype.h>
int isalnum( int c )
int isalpha( int c )
int iscntrl( int c
int isdigit( int c )
int isxdigit( int c )
int islower( int c )
int isupper( int c )
int isprint( int c )
int isgraph( int c )
int ispunct( int c )
int isspace( int c )
int isblank( int c )
```

Toutes les fonctions retournent une valeur différente de 0 (vrai) si le test est positif et 0 (faux) si le test est négatif.
`isalnum` teste si `c` est un des caractères `A-Z a-z 0-9`.
`isalpha` teste si `c` est un des caractères `A-Z a-z`.
`isdigit` teste si `c` est un des caractères `0-9`.
`isxdigit` teste si `c` est un des caractères `0-9 A-F a-f`.
`islower` teste si `c` est un des caractères `a-z`.
`isupper` teste si `c` est un des caractères `A-Z`.
`isprint` teste si `c` est un caractère imprimable, c'est-à-dire si `c` n'est pas un caractère de contrôle. L'espace est considéré comme imprimable.
`isgraph` teste si `c` est un caractère imprimable différent de l'espace.
`ispunct` teste si `c` est un caractère de ponctuation, c'est-à-dire un caractère imprimable autre qu'un espace ou un caractère `isalnum`.
`isspace` teste si `c` est un caractère « blanc » : `'\t'` - tabulation, `'\r'` - carriage return, `'\n'` - new line, `'\v'` vertical tab, `'\f'` form feed, `' '` - espace.
`isblank` teste si `c` est un caractère pouvant séparer des mots. Le test est positif pour les caractères espace `' '` et tab `'\t'` et peut-être pour d'autres caractères en fonction de la localisation.

6.2 Fonctions de transformation de caractères : tolower, toupper

```
#include <ctype.h>
int tolower( int c )
int toupper( int c )
```

Si `c` est une lettre majuscule, `tolower` retourne la lettre minuscule correspondante. Si `c` est une lettre minuscule, `toupper` retourne la lettre majuscule correspondante. Dans les autres cas, la fonction retourne `c`.

7 Chaîne de caractères $\Longleftrightarrow$ nombre

7.1 Extraire un nombre d'une chaîne de caractères : atof, atoi, atol, atoll

```
#include <stdlib.h>
double atof( const char *str )
int    atoi( const char *str )
long   atol( const char *str )
long long atoll( const char *str )
```

Ces fonctions convertissent `str` en nombre en ignorant les espaces[3] au début de `str`. Par exemple dans

```
char *s="   \n -12abc";
int i = atoi( s );
```

`i` prendra la valeur -12.

7.2 Extraire un nombre d'une chaîne de caractères : strtod, strtof, strtold, strtol, strtoll, strtoul, strtoull

```
#include <stdlib.h>
double      strtod( const char * restrict str, char ** restrict ptr )
float       strtof( const char * restrict str, char ** restrict ptr )
long double strtld( const char * restrict str, char ** restrict ptr )


long strtol( const char * restrict str, char ** restrict ptr,
                    int base )
long long strtoll( const char * restrict str, char ** restrict ptr,
                     int base )
unsigned long strtoul( const char * restrict str, char ** restrict ptr,
                      int base )
unsigned long long strtoull( const char * restrict str, char ** restrict ptr,
                            int base )
```

Les fonctions `strtod, strtof strtold` transforment `str` en nombre réel en ignorant les espaces (`isspace`) au début de `str`. Si `ptr` est différent de `NULL`, ces fonctions placent à l'adresse `ptr` un pointeur vers le premier caractère non-converti.
Dans l'exemple suivant, après l'exécution du code

3. les espaces dans les sens de la fonction `isspace`

```
char str[]="   -12.78 a12 ";
char *p;
double d = strtod(str, &p);
```

`d` prend la valeur -12.78 et `p` contient un pointeur vers le caractère espace qui suit le caractère `'8'`.

Les fonctions `strtol, strtoll, strtoul, strtoull` transforment `str` en un entier en ignorant les caractères blancs au début de `str`. Le paramètre `base` permet de spécifier la base (voir la page man pour les détails). Si `base` vaut 0, les conventions habituelles s'appliquent (le préfixe `0x` désigne le nombre hexadécimal, le préfixe `0` désigne un nombre octal et dans les autres cas on suppose que `str` contient un nombre décimal).
Si `ptr` est différent de `NULL` les fonctions placent à l'adresse `ptr` un pointeur vers le premier caractère non-converti.

7.3 Nombre vers une chaîne de caractères : sprintf, snprintf

```
#include <stdio.h>
int  sprintf( char * restrict s, const char * restrict format, ... )
int  snprintf( char * restrict s, size_t len, const char * restrict format, ... )
```

La fonction `snprintf` a été ajoutée en C99 et doit être utilisée à la place de `sprintf` (cette dernière fonction peut conduire à un dépassement de tampon).
La fonction `snprintf` transforme les arguments indiqué par ... en texte selon `format` de la même façon que les fonctions `printf` et `fprintf`. La seule différence est que `snprintf` écrit le résultat dans un tampon `s` au lieu de l'envoyer dans un fichier. `snprintf` fabrique une chaîne de caractères, c'est-à-dire qu'elle ajoute un caractère nul pour terminer la suite de caractères écrits dans le tampon.
Le paramètre `len` de `snprintf` indique que `snprintf` peut écrire au plus `len` octets (y compris le caractère nul qui termine la chaîne), donc `len` ne peut pas dépasser la taille du tampon `s`.
Exemple :

```
#define T 254
  char s[T];

  int i = -22;
  snprintf(s, T, "%4d", i );
```

Après l'appel à `snprinf` le tableau `s` contient la chaîne de caractères `" -22"` (espace suivi par les trois caractères `-22`, le tout terminé par le caractère nul).
En général, `snprintf` (comme ses cousines `printf` et `fprintf`) peut transformer plusieurs objets en une chaîne de caractères.

8 Les entrées/sorties, les fichiers

8.1 Introduction

Cette section présente uniquement les opérations de haut niveau sur les fichiers. Les opérations de bas niveau, utilisant les descripteurs de fichiers, font partie de programmation système (POSIX) et ne font pas partie du C standard.

Les opérations de haut niveau de cette section sont implementées à l'aide des opération de bas niveau [4].
Cela signifie, en particulier, que l'objet flot `FILE *` utilisé dans cette section contient à l'intérieur un descripteur de fichier (au moins dans les systèmes de type UNIX).
Il existe d'ailleurs une fonction (`fileno`) qui permet de récupérer le descripteur de fichier caché dans `FILE *` comme il existe une fonction (`fdopen`) qui fabrique le flot `FILE *` à partir d'un descripteur.
A noter que le C standard ne connaît pas la notion de répertoire et il faut chercher dans POSIX pour avoir les fonctions qui permettent de parcourir un répertoire.
Le C standard ne sais rien non plus sur le propriétaire de fichier et ne peut pas gérer les problèmes liés aux droits d'accès.
Les opérations de type « redirection » et communication par les tubes (pipe) sont aussi hors de portée du C standard.
Mais, côté positif, les fonctions de haut niveau sont implémentées partout où il y a un compilateur C donc aussi sur les systèmes qui n'ont rien avoir avec UNIX/Linux. Les fonctions de haut niveau permettent de gérer les entrées/sorties formatées tandis que les entrées sorties de bas niveau sont binaires [5].
Finalement, l'objet `FILE *` gère automatiquement un tampon ce qui augmente l'efficacité des opérations d'entrée/sortie [6].

8.2 fopen, fclose, fflush

```
#include <stdio.h>
FILE *fopen( const char * restrict filename, const char *mode )
int fclose(FILE * restrict stream)
int fflush(FILE * restrict stream)
```

La fonction `fopen` prend comme argument le nom du fichier et le mode d'ouverture. La fonction retourne le pointeur `FILE *` qui permet ensuite d'effectuer différentes opérations sur le flot. En cas d'échec la fonction retourne `NULL`.
Il y a 6 modes d'ouverture possibles :

mode	description
`"r"`	ouverture en lecture
`"w"`	ouverture en écriture, création d'un nouveau fichier ou troncature d'un fichier existant
`"a"`	ouverture en écriture, création d'un nouveau fichier ou l'ajout à la fin d'un fichier existant
`"r+"`	ouverture en lecture **et** écriture d'un fichier existant, la position courante initiale au début de fichier
`"w+"`	ouverture en lecture **et** écriture, création d'un nouveau fichier ou troncature de fichier existant
`"a+"`	ouverture en lecture **et** écriture, création d'un nouveau fichier si le fichier n'existe pas, les opérations d'écriture se font toujours à la fin du fichier

Notez que le mode d'ouverture est toujours une chaîne de caractère même dans le cas où c'est une chaîne qui contient un seul caractère, donc on écrit `fopen("toto","r")` et non `fopen("toto",'r')`.

Si le fichier est ouvert en lecture et en écriture (un des modes avec `+`), les opérations de lecture doivent être séparées des opérations d'écriture par un des appels suivants : `fsetpos, fseek, rewind, fflush`.

4. Mais essayer d'implementer `fprintf` à l'aide de `write` ou `fscanf` à l'aide de `read` et vous verrez si c'est simple. Pour cela il faudrait d'abord comprendre les détails de formatage de `fprintf` et `fscanf`.

5. mais si on utilise les fonctions de bas niveau on peut toujours formater/deformater à l'aide de `snprintf` et `sscanf`.

6. Mais rien n'empêche de créer et gérer notre propre tampon et l'utiliser avec les fonctions de bas niveau, ce que font les systèmes de gestion de bases de données.

Cela signifie que, après une suite d'opérations de lecture, pour faire une écriture il faut d'abord exécuter au moins une de 4 opérations ci-dessus.
C'est la même chose dans le sens inverse : après une suite des opérations d'écriture, pour faire une lecture il faut d'abord exécuter une de 4 opérations ci-dessus.
Ces 4 opérations provoquent le vidage du tampon associé à `FILE`.

`fflush()` vide le tampon associé à `stream` en écrivant le contenu dans le fichier.

`fclose` ferme le flot `stream`. À cette occasion le tampon associé au flot sera automatiquement vidé (inutile d'appeler `fflush` avant `fclose`).

8.3 Flots stdin, stdout, stderr

Les trois variables `stdin, stdout, stderr` de type `FILE *` sont définies dans `stdio.h`. Elle contiennent respectivement les références vers les flots d'entrée/sortie/sortie erreur.
Cependant l'utilisateur peut fermer ces flots au démarrage du processus ou, en utilisant les redirections, changer les fichiers associés à ces flots.
`stdin` est ouvert en mode lecture tandis que `stdout, stderr` sont ouverts en mode écriture. Si l'utilisateur n'a pas fait de redirection `stdin`, `stdout` et `stderr` sont associés au terminal.
Vous pouvez fermer ces trois flots comme d'autre flots avec `fclose`.
Il existe des fonctions spécifiques de lecture/écriture pour les flots `stdin` et `stdout` mais vous pouvez toujours utiliser les fonctions générales de lecture/écriture sur les flots en passant `stdin`, `stdout` à la place de l'argument `FILE *`.

8.4 Gestion de la position courante : fseek, ftell, rewind, fgetpos, fsetpos

```
#include <stdio.h>
int fseek( FILE * restrict stream, long int offset, int wherefrom )
long int ftell( FILE * restrict stream )
void rewind( FILE * restrict stream )
int fgetpos( FILE * restrict stream, fpos_t *pos )
int fsetpos( FILE * restrict stream, const fpos_t *pos )
```

Les fonctions de cette section permettent de changer la position courante dans le flot `stream`.
La fonction `ftell` retourne la position courante dans le flot `stream`. Le résultat peut être utilisé comme deuxième argument de la fonction `fseek`. `ftell` retourne $-1L$ en cas d'erreur. La position 0 est la position au début du flot.
La fonction `fseek` permet de modifier la position courante d'un flot `stream`. Le deuxième argument de type `long int` spécifie la position, comptée en nombre de caractères par rapport au point de départ spécifié par le paramètre `wherefrom`. Ce paramètre prend une des trois valeurs `SEEK_SET`, `SEEK_CUR`, `SEEK_END`. Si `wherefrom` vaut `SEEK_SET`, alors `offset` est compté à partir du début du flot.
Par exemple `fseek(flot, 100L, SEEK_SET)` met la position courante 100 octets après le début du fichier.
Si `wherefrom` vaut `SEEK_CUR`, alors `offset` est compté à partir de la position courante du flot, par exemple `fseek(flot, -100L, SEEK_CUR)` recule la position courante de 100 octets.
Si `wherefrom` vaut `SEEK_END`, alors `offset` est compté à partir de la fin du flot.
`fseek(flot, -100L, SEEK_END)` met la position courante 100 octets avant la fin du fichier.
`fseek` retourne -1 en cas d'erreur.

Les fonctions `fgetpos` et `fsetpos` permettent de gérer la position courante pour les fichiers trop longs pour que la position puisse être représentée par `long int`. `fgetpos` permet de sauvegarder la position courante et `fsetpos` permettra ensuite de revenir sur cette position comme dans cet exemple :

```
fpos_t pos;
fgetpos(flot, &pos);
/* d'autres operations sur le flot */

/* revenir sur la position pos*/
fsetpos(flot, &pos);
```

Les fonctions `fgetpos` et `fsetpos` retournent -1 en cas d'erreur et 0 en cas de réussite.

8.5 Lecture d'un caractère : fgetc, getc, getchar, ungetc

```
#include <stdio.h>
int fgetc( FILE *  stream )
int getc( FILE *  stream )
int fgetchar( void )
int ungetc( int c, FILE *stream)
```

`fgetc` retourne un caractère lu dans le flot `stream`. La position courante dans `stream` est avancée d'un caractère. La fonction retourne `EOF` à la fin de `stream` ou en cas d'erreur. Il faut utiliser `ferror` et `feof` pour distinguer le cas où la valeur de retour EOF est provoquée par une erreur ou par la fin du fichier.
`getc` fait la même chose que `fgetc` mais `getc` est d'habitude implémenté comme une macro-fonction.
`getchar()` est équivalent à `fgetc(stdin)`.
`ungetc` remet le caractère `c` dans le flot. La lecture suivante avec `fgetc` ou `getc` retournera ce caractère. Il est possible de remettre plusieurs caractères dans le flot.
`ungetc` retourne le caractère `c` quand l'opération est réussie et `EOF` quand l'opération échoue.
Notez que le caractère `c` n'est pas physiquement remis dans le fichier, `ungetc` garantit juste que le `fgetc` ou `getc` qui suit retournera ce caractère.
L'appel à une des fonctions `fseek, rewind, fsetpos` annule l'effet de `ungetc`.

8.6 Lecture d'une ligne de caractères : fgets

```
#include <stdio.h>
char  *fgets(char *s, int n, FILE *stream)
```

La fonction lit jusqu'à $n-1$ caractères et les met à l'adresse `s`. La lecture s'arrête quand le caractère `'\n'` de la nouvelle ligne est lu ou quand la fonction rencontre la fin du fichier. Après tous les caractères lus la fonction place le caractère nul `'\0'` à la fin dans `s`. Si la fonction arrête sa lecture sur le caractère `'\n'`, ce caractère est aussi placé dans le tableau `s` juste avant `'\0'`.
Dans tous les cas `fgets` met dans `s` un string au sens de C.
Si l'appel à `fgets` réussit la fonction retourne `s`. `fgets` retourne `NULL` en cas d'erreur ou à la fin du fichier. Pour distinguer les deux cas (erreur ou fin de fichier) il faut utiliser les fonctions `feof` et `ferror`.

8.7 Écrire un caractère dans un fichier : fputc, putc, putchar

```
#include <stdio.h>
int fputc(int c, FILE *stream)
int putc(int c, FILE *stream)
int putchar(int c)
```

Les fonctions écrivent le caractère `c` dans le flot `stream`. Elle retournent le caractère `c` en cas de succès et EOF si l'opération échoue.
`putchar` est équivalent à `putc(c,stdout)`.

8.8 Lecture/écriture dans les fichiers binaires : fread, fwrite

```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t element_size, size_t count,
             FILE * restrict stream)
size_t fwrite(void * restrict ptr, size_t element_size, size_t count,
              FILE * restrict stream)
```

Ces fonctions permettent de lire et écrire des données binaires. `stream` est un flot ouvert en lecture ou écriture, `ptr` un pointeur vers un tableau de `count` éléments, chacun de taille `elemnt_size` octets.
`fread` retourne le nombre d'éléments lus. Ce nombre peut être inférieur à `count` si le fichier termine avant la lecture de `count` éléments. `fread` retourne 0 si la fonction n'arrive pas à lire, soit parce que la position courante est déjà à la fin du fichier soit parce qu'il y a une erreur de lecture. Pour distinguer ces deux cas (fin de fichier ou erreur) on utilisera `feof()` et `ferror()`.
`fwrite` retourne le nombre d'éléments écrits, qui sera le même que `count` sauf en cas d'erreur.
Remarque : les fichiers binaires ne sont pas portables d'une machine à l'autre.

Exemple. Écriture et lecture de tableaux de nombres `double`.

```
#define LEN 124
double tab[LEN];
/* écrire un tableau de LEN nombres double dans flot_out*/
size_t ne = fwrite(tab, sizeof(double) ,  LEN, flot_out);

/* les nombres doubles par les tranches de TAILLE nombres */
#define TAILLE 1024
double tt[TAILLE];
clearerr(flot_in);
size_t nl;
while( ( nl = fread(tt, sizeof(double) ,  TAILLE, flot_in) ) > 0 ){
   /* traiter les nl nombres lus dans tt */

}
if( ferror(flot_in) ){
    /* traiter erreur de lecture */
}
if( feof(flot_in) ){
   /* fin de fichier */

}
```

8.9 feof, ferror, clearerr

```
#include <stdio.h>
int  feof( FILE *  stream)
```

```
int ferror( FILE * stream)
void clearerr( FILE * stream)
```

Vous pouvez remarquer que les fonctions de lecture de fichier retournent la même valeur à la fin de fichier et en cas d'erreur de lecture. Chaque objet `FILE *` possède deux indicateurs : indicateur de fin de fichier et indicateur d'erreur.
L'indicateur de fin de fichier est mis à 1 si la lecture échoue à la fin de fichier.
L'indicateur d'erreur est mis à 1 si la lecture échoue à cause d'une erreur.
Les fonctions `feof` et `ferror` permettent de découvrir l'état de l'indicateur de fin de fichier et indicateur d'erreur respectivement.
Si la fin de fichier a été détectée pendant une tentative de lecture `feof(stream)` retournera une valeur différente de 0 (vrai), sinon 0 (faux) est retourné.
La fonction `ferror` retourne une valeur différente de 0 quand la lecture échoue à cause d'une erreur de lecture.

Les indicateurs ne sont pas mis à 0 automatiquement. Si un des deux indicateurs a été activé (mis à 1) par l'opération de lecture, il reste activé tant qu'on ne fait pas d'appel à `clearerr`.
`clearerr(stream)` met à 0 les deux indicateurs.

8.10 Opérations sur les fichiers : remove, rename, tmpfile, tmpnam

```
#include <stdio.h>
int  rename( const char *oldname, const char *newname)
int remove( const char *filename)
FILE *tmpfile(void)
char *tmpnam(char *s)
```

`rename` change le nom du fichier `oldname` en `newname`.

`remove` supprime le fichier. `remove` retourne 0 en cas de succès et -1 en cas d'échec.
D'après The Single UNIX Specification si `filename` est un fichier alors `remove` est équivalent à `unlink(filename)` et si `filename` est un répertoire alors `remove(filename)` est équivalent à `rmdir(filename)` (`unlink` et `rmdir` sont des fonctions POSIX mais pas de C standard).

`tmpfile` crée un fichier temporaire différent de tous les fichiers qui existent. Le fichier temporaire est ouvert en mode `"wb+"`, c'est-à-dire en lecture et écriture et en tant que fichier binaire. `tmpfile` retourne `NULL` si le fichier temporaire n'a pas pu être créé.
Le fichier temporaire est automatiquement supprimé à la fermeture par `fclose` ou quand le processus termine de façon normale (exit ou return dans main).
Quand le programme termine de façon anormale (par exemple tué par un signal), il est possible que le fichier temporaire ne soit pas supprimé.

`tmpnam` est sensé retourner un nom qui pourra ensuite être utilisé comme un nom de fichier ; `tmpnam` garantit que ce nom n'est pas un nom de fichier qui existe déjà.
`tmpnam` avec argument `NULL` mémorise le nom du fichier en interne et les appels suivants modifient ce nom.
Si l'argument `s` de `tmpnam` est non NULL alors `s` doit être un pointeur vers un tableau d'au moins `L_tmpnam` caractères et la fonction met à cette adresse le nom qu'elle a généré.

8.11 Gestion de tampon : setvbuf, setbuf

```
#include <stdio.h>
int  setvbuf( FILE * restrict stream, char * restrict buf, int mode,
              size_t size)
void setbuf( FILE * restrict stream, char * restrict buf)
```

La fonction `setvbuf` peut-être utilisée après l'ouverture du fichier mais avant toute autre opération sur le fichier.
L'argument mode détermine comment le tampon du flot sera utilisé :

- `_IOFBF` input/output buffy buffered,
- `_IOLBL` input/output line buffered,
- `_IONBF` input/output unbuffered.

Si le paramètre `buf` est NULL le tampon est alloué par `setvbuf` sinon `buf` est le pointeur vers le tampon à utiliser par le flot. Dans les deux cas `size` donne la taille du tampon.
La fonction retourne 0 si l'appel a réussi.

Si le paramètre `buf` est non NULL `setbuf` est équivalent à `setvbuf(stream, buf, _IOFBF, BUFSIZ)`.
Si le paramètre `buf` est NULL, `setbuf` est équivalent à `setvbuf(stream, NULL, _IONBF, 0)`.

8.12 Extraire le dernier élément d'un chemin : basename

```
#include <libgen.h>
char *basename(char *path)
```

La fonction `basename` n'est pas une fonction du C standard mais c'est une fonction POSIX. Elle retourne le dernier élément d'un chemin, par exemple `basename("progrs/C/toto")` retournera `"toto"`.

9 Les fonctions de recherche et tri : bsearch, qsort

```
#include <stdlib.h>
void  *bsearch( const void *key, const void *base,
                size_t nmemb, size_t size,
                int (*compar)(const void *, const void *))
void qsort(void *base, size_t nmemb, size_t size,
                int (*compar)(const void *, const void *))
```

La fonction `bsearch` cherche dans un tableau de `nmemb` objets, dont le premier élément est pointé par `base`, un élément pointé par `key`. La taille de chaque élément du tableau (et la taille de l'élément `key`) est de `size` octets. `compar` est un pointeur vers la fonction de comparaison qui permet de comparer `key` avec les éléments du tableau.

`qsort` fait le tri d'un tableau de `nmemb` éléments, chacun de taille `size` octets. Le pointeur `base` pointe vers le premier élément du tableau. `compar` est le pointeur de la fonction de comparaison.

`compar` est un pointeur vers une fonction qui doit prendre en paramètre les adresses des deux éléments à comparer. Chaque élément est de `size` octets.

`compar` doit retourner une valeur < 0 si le premier argument de `compar` est inférieur au deuxième, 0 si les deux arguments sont égaux dans l'ordre et une valeur > 0 si le premier argument est plus grand que le deuxième.

Exemple. Pour trier un tableau de chaînes de caractères dans l'ordre lexicographique, il faut définir une fonction qui permet de comparer deux chaînes de caractères pour déterminer leur ordre. La fonction `strcmp` fait ce qu'il faut mais on ne peut pas l'utiliser dans `qsort`, car les paramètres ne sont pas les bons : les paramètres de `strcmp` sont des chaînes de caractères (les données à comparer) mais `qsort` a besoin d'une fonction `compar` avec deux paramètres de type pointeur vers les données (donc des pointeurs vers les chaînes de caractères).
Donc il faut fabriquer notre propre fonction de comparaison pour `qsort` :

```
//fonction compar pour qsort
int lex(const void *mota, const void *motb){
  return strcmp( *(char **)mota, *(char **)motb);
}
```

Et maintenant nous pouvons faire le tri :

```
qsort(mots, nlem, sizeof(char *), lex);
```

trie le tableau `mots`, où `char *mots[]` est un tableau de `nlem` chaînes de caractères.

10 Les fonctions sur les entiers : abs, labs, llabs, div, ldiv, lldiv

```
#include <stdlib.h>
int abs(int i)
long  labs(long i)
long long llabs(long long i)
```

Ces fonctions retournent la valeur absolue de l'argument.

```
#include <stdlib.h>
div_t div(int number, int denom)
ldiv_t  ldiv(long number, long denom)
lldiv_t lldiv(long long number, long long denom)
```

Ces fonctions calculent en même temps le quotient `number/denom` et le reste de la division `number%denom` en mettant les résultats dans une structure avec deux champs : `quot` – le quotient – et `rem` (the remainder) – le reste de la division.

```
long a = 239876;
long b = 7;
ldiv_t d = ldiv(a,b);
/*  d.quot - le quotient  d.rem - le reste */
```

11 Vérification de condition et messages d'erreur

11.1 Vérifier si une condition est satisfaite avec assert

```
#include <assert.h>
void assert(int expression)
```

`assert` évalue `expression`. Si la valeur est 0 (condition non satisfaite), alors `assert` affiche sur `stdout`

- `expression`,
- le nom du fichier source et
- le numéro de la ligne dans le fichier source

et termine le programme avec `abort`.
`assert` sert à vérifier des conditions dans le programme, par exemple pour vérifier que i est entre 0 et $n-1$

```
assert( i <= 0 && i < n );
```

Si la condition n'est pas satisfaite, après l'affichage indiqué ci-dessus le programme s'arrête.
Pour désactiver tous les asserts, il n'est ps nécessaire de les mettre en commentaire. Il suffit d'ajouter au début du programme la ligne

```
#define NDEBUG
```

ou de compiler avec l'option

```
-DNDEBUG
```

pour rendre les asserts non opérationnels (le préprocessseur les enlèvera du programme).

11.2 Gestion d'erreurs : errno, strerror, perror

```
#include <errno.h>
extern int errno;
------------------------
#include <stdio.h>
void perror(const char *s)
------------------------
#include <string.h>
char *strerror(int errnum)
```

Les fonctions dont l'exécution termine avec une erreur [7] mettent le numéro d'erreur dans la variable globale `errno`. La page man de `errno` donne les noms symboliques pour différentes erreurs. `errno` peut être une variable de type int, mais d'autres définitions sont possibles.
La valeur 0 n'est jamais utilisée comme un code d'erreur. Donc avec `errno = 0;`, on efface le code d'erreur.

La fonction `strerror` retourne le message correspondant au numéro d'erreur `errnum`. Il est bien plus intéressant d'afficher ce message que d'afficher la valeur d'`errno`.
Exemple :

```
fprintf(stderr, "\n error in file %s in line %d : %s\n",
                __FILE__, __LINE__, strerror(errno));
```

7. La plupart de fonctions C peuvent terminer avec erreur, mais pas toutes.

affichera le message d'erreur avec le nom de fichier source et le numéro de la ligne dans le code source.
Le préprocesseur remplacera

- la macro-constante `__FILE__` par le nom de fichier source,
- la macro-constante `__LINE__` par le numéro de la ligne dans le fichier source. (Notez que `__LINE__` est une macro-constante qui n'est pas vraiment constante, la valeur change sur chaque ligne de fichier source.)

Ces deux macro-constantes font partie du C standard.

La fonction `perror` envoie sur `stderr` la valeur du paramètre `s` suivie d'un message d'erreur correspondant à la valeur actuelle de `errno`.

12 Autres fonctions

12.1 Génération de nombres pseudo-aléatoires : rand, srand

```
#include <stdlib.h>
int  rand( void )
int srand( unsigned int seed )
```

`rand` permet d'obtenir une suite pseudo-aléatoire de nombres entiers entre 0 et `RAND_MAX`.
`srand` utilise `seed` comme une graine pour initialiser la suite produite par les appels à `rand`. Si `srand` est appelé avec la même valeur `seed`, la même suite sera générée par `rand`.
Si `rand` est appelé sans faire un appel à `srand` alors la suite générée sera la même que celle générée après l'appel `srand(1)`.

12.2 Génération de nombres pseudo-aléatoires en mieux : random, srandom

La qualité du générateur implémenté dans les fonctions de la section précédente est médiocre. Un meilleur générateur s'obtient avec les fonctions

```
#include <stdlib.h>
long  random( void )
int srandom( unsigned long seed )
```

qui s'utilisent comme les fonctions `rand` et `srand` mais avec les nombres `long` au lieu de `int`.
En particulier le période de la suite générée par `random` est au moins 2^{69}. Les fonctions `random` et `srandom` font partie de POSIX mais pas du C standard.

12.3 exit, abort, getenv

```
#include <stdlib.h>
void  exit( int status )
void  abort( void )
char *getenv( const char *name )
```

`exit` termine le programme, c'est la terminaison normale. Les fonctions enregistrées à l'aide de `atexit` sont exécutées dans l'ordre inverse d'enregistrement.
Pour les fichiers ouverts, l'opération `fflush` est effectuée et les fichiers sont fermés.

Par convention la valeur `exit(0)` ou `exit(EXIT_SUCCESS)` correspond à la terminaison sans erreur et `exit(EXIT_FAILURE)` ou `exit` avec une valeur de retour supérieure à 0 à une terminaison avec erreur.
`return n;` dans la fonction `main` correspond (dans la plupart des cas) à `exit(n)`.
Dans la norme C99 il est possible de terminer la fonction `main` sans `return` ni exit et le résultat est le même que `return 0;`.
Vous pouvez utilisez n'importe quelle valeur entière comme la valeur de `exit`, cependant les fonctions de programmation système (POSIX) comme `wait` et `waitpid` permettent de voir seulement les 8 bits de poids faible de la valeur exit. Pour cette raison, il est recommandé d'utiliser des valeurs `exit` entre 0 et 255.
Dans le shell la valeur de la variable `$?` est égale à la valeur `exit` du dernier processus terminé qui a été lancée depuis ce shell [8].

`abort` provoque une terminaison anormale de programme sauf si le signal `SIGABRT` a été intercepté.

`getenv` retourne la valeur de la variable d'environnement dont le nom est donné en paramètre. Par exemple `getenv("HOME")` retourne la valeur de la variable `HOME`.
La fonction `setenv` qui permet de changer la valeur de la variable d'environnement est une fonction de POSIX mais pas de C standard.

13 Fonctions à nombre variable d'arguments

```
#include <stdarg.h>
typedef ... va_list;
#define va_start( va_list ap, type_last_fixed_param )
#define va_arg( va_list ap, type)
void va_end(va_list ap)
void va_copy(va_list dest, va_list src)
```

Une fonction à nombre variable d'arguments utilisera ces macros pour parcourir ses arguments.
Au moins un argument d'une telle fonction doit être explicitement nommé. `va_star` sert à initialiser l'argument `va_list ap`.
Le parcours s'effectue par les appels successifs à `va_arg` dont le premier argument a été initialisé par `va_start` et le deuxième argument est le type de l'argument que nous voulons récupérer.
`va_end` doit être appelé en fin de parcours.
Ces macros-fonctions ne permettent pas de déterminer le nombre d'arguments. Donc il faut déduire ce nombre par d'autres moyens.
Par exemple, si les arguments sont des pointeurs, alors on pourra adopter la convention que le dernier argument est `NULL` pour marquer la fin de la liste d'arguments.
Exemple. La fonction `add_double` prend un nombre quelconque d'arguments `double`. On termine la liste d'arguments par la constante `INFINITY`. Et la fonction calcule la somme des arguments (sauf le dernier).

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
double add_double(double i, ...){
  double d = i;
  va_list l;
```

8. Mais attention, encore une fois on ne verra que les 8 bits de poids faible. Donc après `exit(-1);` dans le processus la commande `echo $?` lancée dans le shell affichera 255. Pourqoui 255 ? Vous devez le comprendre et déduire en vous basant sur les informations données ici et sur la représentation de -1 en complement à deux.

```
  va_start(l, i);
  while(1){
    double q = va_arg(l, double);
    if( isfinite( q ) ){
      printf("%4.1f\n",q);
      d += q;
    }else{
      va_end(l);
      return d;
    }
  }
}
int main(void){

  double p = add_double(4, 7.8, -4.0, -11.1, INFINITY);
  printf("%4.1f\n",p);
  return 0;
}
```