Pratique du C
Bibliothèque(s) standard(s)

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 4 — 2017-2018

De quoi s'agit-il ?

La *bibliothèque standard* du C est une collection normalisée (ANSI puis ISO) de fichiers :

- d'en-têtes définissant des macros, des variables globales, des types et déclarant des prototypes de fonctions ;
- objet associés implantant — dans la *librairie standard* — des aglorithmes (e.g. tri), des structures de données (e.g. table de hachage) et des opérations courantes (e.g. entrées – sorties, gestion des chaînes de caractères, calculs, encapsulation d'appels système, etc).

Sauf demande expresse du contraire, `gcc` provoque systématiquement l'édition des liens avec la librairie standard *libc.a* habituellement située dans le répertoire `/usr/lib`. Les fichiers d'en-tête sont dans `/usr/include`.

En comparaison avec d'autres langages (e.g. Java), la bibliothèque standard est minuscule (e.g. pas d'interface graphique) ce qui facilite le portage sur de nouvelles plateformes.

Pas une mais des bibliothèques

Les différentes normes (ISO, ISO C99, POSIX, etc.) implantent dans la bibliothèque standard différentes fonctionnalités.

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

/* strtod, strtof, strtold - convert ASCII string to floating
point number float strtof(const char *, char **); */

int main(int argc, char **argv){
  printf("%f\n", strtof(argv[1], NULL)) ;
  return 0 ;
}
```

En compilant avec l'option ANSI, le prototype de la fonction `strtof` — n'existant pas dans cette norme et donc dans une directive conditionnelle — n'est pas pris en compte. Sans prototype, la valeur de retour de la fonction est supposée être un entier machine et le résultat — qui n'est pas codé comme un flottant — est donc faux.

Du coté des mathématiques

La norme ISO requière — non exhaustivement — dans la bibiliothèque standard les éléments suivants :

- `complex.h` : définition et manipulation de nombres complexes ;
- `fenv.h` : définition et manipulation de nombres en virgule flottante ;
- `float.h` : définitons spécifiant les propriétés des nombres en virgule flottante (nombre maximale de chiffre de précision, etc.) ;
- `inttypes.h` : types d'entiers indépendant des architectures matérielles ou logicielles ;
- `math.h` : définitions de fonctions mathématiques courantes (se compile avec l'option `-lm`).

Cela permet d'utiliser des structures de données et des algorithmes non triviaux.

La macro assert est définie dans le fichier d'entête `assert.h`.

```
#include<assert.h>
int
main
(void)
{
int i = 1 ;
        assert(i!=1);
        return 3/(i-1) ;
}
```

Lors de l'exécution, si l'évaluation de l'expression est 0, `assert` écrit les informations sur l'appel qui a échoué dans le flot `stderr` :

- le nom du fichier source ;
- le numéro de la ligne concernée dans le code source ;
- la fonction mise en jeu dans le code source ;
- le texte de l'expression qui a été évaluée à 0.

pour finir, `abort` est appelée. On obtient dans notre exemple :

```
Assertion failed: (i!=1), function main, file toto.c, line 7.
```


Libraire standard de gestion d'erreurs

```
#include<errno.h>
#include<stdio.h>
#include<stdlib.h>
int
main
(void)
{
  if(malloc(-1)==NULL)
  {
    perror("cela ne marche pas\n") ;
    return errno ; /* retourne le code positionn\'e */
  }                /* par malloc */
  return 0 ;
}
```

On obtient :

```
% ./a.out ; echo $?
cela ne marche pas
12
% grep 12 /usr/include/asm-generic/errno-base.h
#define ENOMEM 12 /* Out of memory */
%
```

La librairie standard n'est pas naïve

Nous pourrions définir la fonction `memcpy` comme suit :

```
void * memcpy(void *dst, const void *src, int lenght)
{
char *dest=(char *) dst ; char *srce = (char *) src ;
int i ;
for (i=0;i<lenght;i++,dest++,srce++)
*dest= *srce ;

return dst ;
}
```

alors que la version courante fait plus de 65 lignes car elle prend en compte la taille des *atomes* du segment de pile et les problèmes d'alignement pour un maximum d'efficacité suivant la plateforme utilisée.

C'est le constat pour une grande variété de fonctions tels `memset`, `strcpy`, etc.

Il n'y a pas que `strlen` dans la vie

`string.h` est l'en-tête de la bibliothèque standard qui contient les définitions des macros, des constantes et les déclarations de fonctions et de types utilisées pour la manipulation de chaînes de caractères et diverses fonctions de manipulations de la mémoire.

Ces fonctions ne sont compatibles qu'avec l'ASCII.

La manipulation des chaînes de type non-ASCII est réalisée à travers `wchar.h` (pour wide-character) qui introduit un type `wchar_t` adapté à la configuration locale de sa machine (ISO-8859, UTF-8, UTF-16, etc).

Ainsi, `string.h` contient le prototype :

```
size_t strlen(const char *);
```

alors que `wchar.h` contient le prototype :

```
size_t wcslen(const wchar_t *);
```


Quand on dépasse les bornes, il n'y a plus de limite

Il existe plusieurs versions des fonctions de manipulations de chaînes de caractères :

```
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

Il faut se servir de `strcpy` uniquement si on sait que l'espace réservé pointé par `dest` est plus grand que celui associé à `src`.

Dans le cas contraire, des données — utilisateur ou système — placées derrière le pointeur `dest` pourrait être détruites voire manipulées (on parle de débordement de tampon).

Ce qui provoque de graves problèmes de sécurité (cf. cours sur la pile d'exécution).

Donc, il faut privilégier les fonctions de type `strncpy`.

Un exemple de contamination par Java

Une mauvaise compréhension des mécanismes du C tend à écrire du code comme :

```
int i;
char *src="toto le haricot";
char dest[20] ;
for(i=0;i<strlen(src);i++)
   dest[i]=src[i] ;
```

alors que cela sous-entend une orientation objet.

C'est catastrophique en C car, si n est la longueur de la chaîne de caractères, on a $\mathcal{O}(n^2)$ accès en mémoire alors que $\mathcal{O}(n)$ suffisent.

Entrées-sorties par appels système

Les fichiers sont représentés par des structures de données :

- dans le noyau où une table regroupe les fichiers ouverts par l'ensemble des processus et contenant le déplacement (*offset*) courant dans le fichier et un mode d'ouverture (r, rw, ...) ;
- dans le *contexte* du processus où une table fait le liens entre un *descripteur* i.e. un entier identifiant unique d'une *ouverture* de fichier local au processus et un pointer dans la table des fichiers ouverts du noyau.

Les appels système de manipulation de fichiers sont :

open, read, write, close, lseek.

dont les prototypes se trouvent dans `<fcntl.h>`.
Les descripteurs ouverts par défaut sont :

- 0 : entrée standard ;
- 1 : sortie standard ;
- 2 : sortie erreur standard.

- `int open(char *name, int mode [, int perm]);` ouvre le fichier `name` suivant le mode et les permissions spécifiés, et retourne le *descripteur de fichier* correspondant ;
- `int close(int fd)` ferme le fichier associé au descripteur `fd`. À la *mort* d'un processus les fichiers sont fermés ;
- `ssize_t read(int fd, void *buf, size_t n)` lis `n` octets, à partir de l'offset courant, depuis le fichier associé au descripteur `fd` et les stocke dans `buf` (retourne le nombre d'octets lus, 0 si EOF et -1 si erreur).
- `ssize_t write(int fd,const void *buf,size_t n)` écris `n` octets provenant de `buf` dans le fichier associé au descripteur `fd` à partir de l'offset courant. La valeur retournée est le nombre d'octets écrits et -1 si erreur.
- `off_t lseek(int fd, off_t offset, int whence);` déplace l'offset courant du fichier associé au descripteur `fd` sans lire ni écrire de `offset` octets. `whence` permet de donner une origine :
  - SEEK_SET : par rapport au début du fichier ;
  - SEEK_CUR : par rapport à l'offset courant ;
  - SEEK_END : par rapport à la fin du fichier.

Encapsulation d'appels système

Il ne faut pas confondre bibliothèque standard et appels système :

- appels système : pas d'édition de liens mais seulement exécution du code de l'OS ;
- bibliothèques standard : édition de liens.

Comme un appel système est *coûteux*, il est nécessaire d'ajouter un tampon dans la gestion des entrées – sorties dans le contexte du processus.

En conséquence :

- il y a moins d'appels système pour des accès sur de petites zones ;
- on peut avoir une lecture/écriture par bloc dans le tampon ;
- mais attention à la vidange des tampons si interruption du processus.

Librairie d'entrée – sortie

Pour ce faire, on utilise un pointeur `FILE *` sur une structure identifiant un fichier ouvert sont la déclaration est dans `<stdio.h>`. On parle dans ce cas de *flot*.

- `FILE *fopen(const char *name, const char *mode);` ouvre le fichier d'identificateur `name` avec le mode d'ouverture spécifié par `mode` ("r", "w", etc.) ;
- `int fclose(FILE *stream);` ferme le fichier associé au flot `stream` ce qui provoque la vidange des tampons. Cette fonction retourne 0 en cas de succès et `EOF` si échec ;
- `FILE *freopen(const char *n, const char *m,FILE *s);` ouvre le fichier d'identificateur `n` dans le mode spécifié par `m` et lui associe le flot pointé par `s`. Le fichier associé à `stream` est préalablement fermé.

- `int fflush(FILE *stream);` procède à la vidange des tampons associés au flot de sortie `stream`. Comportement indéterminé si `stream` est un flot d'entrée. Retourne 0 en cas de succès, `EOF` si échec ;
- `size_t fread(void *ptr, size_t m, size_t n, FILE *s);` place dans le tableau pointé par `ptr` jusqu'à `n` éléments lus sur le flot pointé par `s` (la taille d'un item est spécifiée par `m`. Retourne le nombre d'éléments lus ;
- `size_t fwrite(void *ptr, size_t m, size_t n, FILE *s);` écrit à partir du tableau pointé par `ptr` jusqu'à `n` éléments sur le flot pointé par `s`. La taille d'un item est spécifiée par `m`. Retourne le nombre d'éléments écrits.

Sortie formatée

La fonction de prototype :

```
#include<stdio.h>
  int fprintf(FILE *stream, const char *format, ...);
```

écrit sur le flot pointé par `stream` au format spécifié par la chaîne `format`. `format` peut contenir des caractères ordinaires, recopiés tels quels, et des spécifications de conversion.

- `printf` provient de l'anglais *print formatted* ;
- `...` est un mot-clef du langage C qui indique que le nombre de paramètre est indéterminé (cf. second cours sur la pile d'exécution) ;
- l'instruction `printf` est dérivée de `fprintf` en indiquant comme flot, le flot prédéfini `stdout` associé à la sortie standard.

```
int i=42; char *txt="It's a wonderful life";
printf("L'\\'etoile %c est de code ascii %d.\n %s\n",i,i,txt);
```


Codage des paramètres implicites

La chaîne de caractères passée en premier argument contient des spécifications codant combien de — et comment — paramètres doivent être affichés.

Une spécification débute par un % suivi de :

- drapeaux de remplissage/justification :
  - `-` : justification à gauche,
  - `+` : impression systématique du signe,
  - `0` : remplit le début du champ avec des zéros ;
- un nombre donnant la largeur minimum du champ ;
- un caractère `.` séparateur ;
- un nombre donnant la précision ;
- une lettre : `h` pour un `short`, `l` pour un `long`, `L` pour un `long double` ;
- un caractère indiquant le type de conversion (cf. suite).

La précision ou la largeur minimum peuvent être remplacées par un astérisque (`*`) ; leur valeur sera alors prise dans la liste des paramètres. Seul le dernier caractère de conversion est obligatoire :

- `d`, `i` : `int` en notation décimale signée ;
- `x`, `X` (`o`) : `int` en notation hexadécimale (octale) non signée ;
- `u` : `int` en notation décimale non signée ;
- `c` : `int` converti en caractère non signé ;
- `f` : `double` en notation décimale signée (*dd.ddd*) ;
- `e`, `E` : `double` en notation scientifique signée ($d.ddde \pm dd$) ;
- `p` : `void *` en format pointeur (hexa. en général).

Entrée formattée

La fonction de prototype :

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

lit sur le flot pointé par `stream` au format spécifié par la chaîne `format`. `format` peut contenir des caractères ordinaires, lus comme tels dans `stream`, où des spécifications de conversion.

Les résultats des conversions sont stockés dans les variables pointées par les arguments suivant `format`.
`fscanf` reconnaît toujours la plus longue chaîne correspondant à `format`.
Une spécification débute par un % suivi de :

- `*` : supprime l'affectation ;
- un nombre donnant la largeur maximum du champ ;
- une lettre : `h`, `l` ou `L` (idem `fprintf`) ;
- un caractère indiquant le type de la conversion.

Seul le dernier caractère de conversion est obligatoire :

- `d` (`i`) : entier sous forme décimale (ou octale ou hexa.) – `int *` ;
- `o` : entier sous forme octale – `int *` ;
- `x` : entier sous forme hexadécimale – `int *` ;
- `u` : entier non signé sous forme décimale – `unsigned int *` ;
- `c` : caractère (espacement compris) – `char *` ;
- `s` : chaîne de caractères (espacement supprimé au début) – `char *` assez grand pour contenir le résultat ;
- `f`, `e` : nombre en virgule flottante – `float *` ;
- `p` : pointeur – `void *` ;
- `[..]` : plus longue chaîne composée de caractères placés entre `[]` – `char *` ;
- `[..]` : plus longue chaîne composée de caractères ne faisant pas partie de l'ensemble entre `[]` – `char *` ;

Formattage en mémoire

On factorise l'API et l'implantation d'entrée – sortie formatée pour accéder à des zones mémoires :

- utile pour "parser" des arguments de la ligne de commande ;
- lecture/écriture réalisée sur une zone mémoire (`char *`) ;
- même fonctionnement que `fprintf` et `fscanf` :
  - écriture formattée dans le tableau pointé par `s` :
    `int sprintf(char *s, const char *format, ...) ;`
  - lecture formattée dans le tableau pointé par `s` :
    `int sscanf(char *s, const char *format, ...) ;`

Entrées-sorties par fonctions mandataires

Les fonctions de prototypes :

```
#include <stdio.h>
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
```

sont des fonctions de la librairie standard — utilisant donc un tampon — qui manipulent des caractères.

Les fonctions de prototypes :

```
char *gets(char *s);
int puts(const char *s);
```

manipulent des lignes.

Et tout le reste est littérature

Les pages du manuel Unix sont divisées en plusieurs sections.
Sous Linux, on a :

1. Commandes utilisateur
2. Appels système
3. Fonctions de bibliothèque
4. Fichiers spéciaux
5. Formats de fichier
6. Jeux
7. Divers
8. Administration système
9. Interface du noyau Linux

Chaque section possède une page d'introduction qui présente la section, disponible par `man <section> intro`.
N'hésitez pas `man man` (pour savoir comment utiliser ce manuel efficacement — e.g. recherche textuelle, etc).