Initiation au C
cours n°6

Antoine Miné

École normale supérieure

22 mars 2007

Plan du cours

Communications :

- les arguments en ligne de commande,
- les variables d'environnement,
- les fichiers, les flux d'entrée-sortie `FILE*`.

Arguments en ligne de commande

Arguments en ligne de commande

Ligne de commande :
on peut passer un nombre arbitraire d'arguments à un programme.

Le *shell* effectue plusieurs opérations sur la ligne de commande :

- Expansion des caractères `*`, `?`,
  les noms des fichiers du répertoire courant sont utilisés.
- Découpage des arguments au niveau des espaces,
  on peut se protéger des caractères spéciaux par `"` ou `\`.

Exemples

`$ ./a.out a b c`	passe `a`, `b` et `c`	(3 arguments)
`$ ./a.out "a b" c`	passe `a b` et `c`	(2 arguments)
`$ ./a.out *`	passe la liste des fichiers	
`$ ./a.out \*`	passe juste `*`	(1 argument)

Lire les arguments depuis le C

En C : ils sont passés en argument à `main` sous forme de chaînes.

Prototype de `main`

```
int main(int argc, char* argv[]);
```

- `argc` : nombre d'arguments disponibles,
- `argv` : tableau de chaînes de caractères :
  - `argv[0]` 1er argument = nom du programme,
  - `argv[1]` 1er argument supplémentaire,
  - ⋮
  - `argv[argc-1]` dernier argument,
  - `argv[argc]` pointeur NULL (invalide !).

Exemple

Exemple de programme

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  int i;
  printf( "%i args\n", argc );
  for ( i=0; i<argc; i++ )
    printf( "%i: %s\n",
            i, argv[i] );
  return 1;
}
```

Exemple de session

```
$ gcc toto.c -Wall -Wextra
$ ./a.out hello world 42 !
5 args
0: ./a.out
1: hello
2: world
3: 42
4: !
$
```

Attention

Un programme ne peut pas faire d'hypothèses sur argc.

Passage d'arguments numériques

Arguments numériques : aussi passés comme des chaînes !

$\Longrightarrow$ c'est au programme C de faire la conversion.

Fonctions prédéfinies

```
#include <stdlib.h>
int    atoi (const char *nptr);
double atof (const char *nptr);
```

Exemple : `int a = atoi(argv[2]);`

Les variables d'environnement

Les variables d'environnement

Environnement : table d'association gérée par le système.

Permet d'affecter à une variable (chaîne) une valeur (chaîne).

L'environnement est :

- initialisé par le *shell*,
  Exemple : `USERNAME` = votre login.
- modifiable par l'utilisateur,
  Exemples : `export VAR=val`, fichier `.login` (pour `bash`).
- hérité par le programme exécuté,
  $\Longrightarrow$ utilisable par votre programme.

Accès à l'environnement depuis le C

Fonction prédéfinie

```
#include <stdlib.h>
char* getenv(const char* name);
```

- retourne (un pointeur sur) une chaîne : la valeur de `name`,
- retourne `NULL` si `name` n'est pas définie,
- la chaîne retournée ne doit pas être modifiée !

Exemple

```
char* nom = getenv("USERNAME");
printf("Bonjour %s\n", nom ? nom : "X");
```

Les flux d'entrée-sortie

Les fichiers

Fichier = bloc de données, non volatile, stocké sur disque

- Chaque fichier a un nom et vit dans un répertoire,
- Données = suite d'octets non structurée,
  les programmes se mettent d'accord sur des formats d'échange.
- Par convention, la fin du nom indique le format,
  e.g. : `.txt` = texte brut, `.png` = image PNG, etc.

Unix = système multi-utilisateurs, donc chaque fichier a :

- un utilisateur et un groupe propriétaire,
- des droits : lecture, écriture, exécution,
  qui concernent : le propriétaire, le groupe, les autres.

(Voir `ls -l` et `chmod`.)

Les flux en C

flux = objet C permettant d'accéder à un fichier.

Type C des flux

```
FILE *
```

Géré par la bibliothèque C,
pour le programmeur, c'est un type abstrait (ne pas déréférencer).

Opérations sur les `FILE*` :

- ouverture : `fopen`
- lecture : `fgetc`, `fgets`, `fscanf`, `fread`
- écriture : `fputc`, `fputs`, `fprintf`, `fwrite`
- navigation : `ftell`, `fseek`, `feof`
- fermeture : `fclose`

Premier exemple

Exemple

```
#include <stdio.h>
int main()
{
  FILE* f = fopen("toto.txt", "w");    /* ouverture */
  if (f==NULL) return 1;               /* vérification */
  fputs("coucou\n", f);                /* écriture */
  fclose(f);                           /* fermeture */
  return 0;
}
```

Effet : écrit la ligne coucou dans le fichier `toto.txt`.

Le pointeur `FILE*` est passé en argument de toutes les fonctions.

Ouverture des flux

Note : toutes les fonctions sont dans `stdio.h`.

Prototype

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

Arguments :

- `path` = nom (chemin) du fichier à ouvrir,
- `mode` = mode d'ouverture (lecture, écriture, ...).

Valeur de retour :

- $\neq$ NULL $\Rightarrow$ nouveau flux représentant le fichier,
- $=$ NULL $\Rightarrow$ échec de l'ouverture,
  (fichier introuvable, droits insuffisants,..., voir `errno`)

$\Longrightarrow$ toujours vérifier la valeur de retour !

Chemins et noms de fichiers

`path` peut représenter un chemin complet :

- liste de répertoires séparés par /
  terminée par le nom de fichier,
- chemin absolu, commençant par /
  e.g. : `/home/mine/truc.c`,
- chemin relatif, ne commençant pas par /
  e.g. : `truc.c`, `bidule/muche/truc.c`,
- le répertoire `..` représente le parent, *e.g.* : `../tmp/x.c`.

Sous Windows :
le séparateur est \ au lieu de /
un chemin absolu commence par un lecteur, *e.g.* : `c:\truc\` $\cdots$

Sous les vieux MacOS :
le séparateur est : au lieu de /.

Modes d'ouverture

`mode` = chaîne décrivant comment le fichier doit être ouvert :

`"r"`	lecture seule
`"w"`	écriture seule
`"r+"` ou `"w+"`	lecture et écriture
`"a"`	écriture à la fin du fichier

Si le fichier existe déjà :

- `"r"` et `"r+"` se placent en début de fichier,
- `"w"` et `"w+"` tronquent le fichier,
- `"a"` se positionne en fin de fichier.

Si le fichier n'existe pas :

- `"r"` et `"r+"` échouent,
- `"w"`, `"w+"` et `"a"` créent le fichier.

Sous Windows : ajouter b pour indiquer un fichier binaire et non texte.

Fermeture des flux

Prototype

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

Effet :

- s'assure que les données sont écrites sur disque (buffering),
- ferme le flux pointé par `fp`,
- le flux pointé par `fp` ne doit plus être utilisé,
  mais on peut faire : `fclose(fp); fp = fopen(...);`

Valeur de retour : 0 si tout s'est bien passé (généralement le cas).

Les fichiers ouverts sont automatiquement fermés si le programme se termine normalement... pas en cas de `Segmentation fault`.

Écritures simples

Prototypes

```
int fputc(int c, FILE *stream);
int  putc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
```

Effet :

- `fputc` écrit un seul caractère `c`,
- `putc` identique à `fputc` (plus rapide),
- `fputs` écrit une chaîne entière, sans le \0 final.

Valeur de retour :

en cas d'erreur, la valeur spéciale `EOF` est retournée,
sinon, une valeur ≥ 0.

Écritures formatées

Prototype

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

S'utilise exactement comme `printf`, avec `stream` en plus :

- `format` : chaîne avec des caractères % magiques,
- on ajoute autant d'arguments que de %.

Valeur de retour :

- nombre de caractères écrits,
- valeur négative en cas d'erreur.

Exemple

```
fprintf(f, "%i * 2 = %i\n", x, x*2);
```

Lecture de caractères

Prototypes

```
int fgetc(FILE *stream);
int  getc(FILE *stream);
```

Effet : lit un seul octet.

`getc` est identique à `fgetc` (en plus rapide).

Valeur de retour :

- entier entre 0 et 255 (`unsigned char`),
- valeur spéciale `EOF` en cas de fin de fichier ou d'erreur.

Exemple

Copie de fichier caractère par caractère

```
void copie(const char* src, const char* dst)
{
  FILE* s = fopen( src, "r" );
  FILE* d = fopen( dst, "w" );
  if (!s || !d) { printf("erreur!\n"); exit(1); }
  while (1) {
    int x = getc(s);
    if (x == EOF) break;
    if (putc(x,d) == EOF) { printf("erreur!\n"); break; }
  }
  fclose(s);
  fclose(d);
}
```

Lecture de lignes complètes

Prototype

```
char* fgets(char* s, int size, FILE *stream);
```

Effet :

- lit une ligne complète, terminée par `\n`,
- stocke la ligne dans la chaîne pointée par `s`,
  inclut le `\n`, puis un `\0` final,
- retourne `NULL` si erreur ou plus rien à lire, `s` sinon.

Cas particuliers : lignes non terminées par `\n`

- la dernière ligne du fichier ne se termine pas forcément par `\n`,
- lit au plus `size` caractères, avant d'ajouter le `\0`.

Attention

Prévoir de la place pour `size`+1 octets dans `s` !

Lectures formatées

Prototype

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

S'utilise exactement comme `scanf`, avec `stream` en plus :

- `format` : chaîne avec des caractères % magiques,
- on ajoute autant de pointeurs que de %.

Valeur de retour :

- nombre d'éléments correctement lus,
- `EOF` en cas d'erreur ou de fin de ficher.

Exemple

```
fscanf(f, "%i,%i,%i", &a, &b, &c);
```

Erreurs sur les fichiers

En cas d'erreur, une valeur spéciale est retournée :

- `NULL` pour un type de retour pointeur,
- `EOF` pour un type de retour entier.

De plus, la variable globale entière `errno` est positionnée.

Codes d'erreur courants

Valeurs symboliques définies dans `errno.h`

`EACCES`	fichier introuvable
`EEXIST`	fichier déjà existant
`EISDIR`	le fichier est un répertoire
`EIO`	erreur matérielle
`EINVAL`	position invalide

(`errno` n'est pas modifié en cas de succès)

Fonctions sur les erreurs

Fonctions permettent de rendre ces codes d'erreur intelligibles.

Prototypes

```
void perror(const char *s);
char* strerror(int errnum);
```

Effet de `perror` :
affiche `s` et un message expliquant l'erreur de code `errno`.

Valeur de retour de `strerror` :
description textuelle de l'erreur de code `errnum`.
$\Longrightarrow$ généralement appelée avec `errno` comme argument.

Fin de fichiers

Lecture en fin de fichier :

- on peut lire moins que ce qui est demandé (`fgets`, `fscanf`),
- si on ne peut rien lire, on retourne une erreur (`NULL`, `EOF`).

Comment différencier vraie erreur et fin de fichier ?

Prototypes

```
int feof(FILE *stream) ;
int ferror(FILE *stream) ;
```

- `feof` retourne $\neq 0$ si on est en fin de fichier, 0 sinon,
- `ferror` retourne $\neq 0$ si il y a eu une erreur, 0 sinon.

Écriture en fin de fichier : pas d'erreur, le fichier est agrandi.

Exemple

Exemple de gestion des erreurs

```
void echo(const char* fname)
{
  FILE* f = fopen(fname, "r");
  if (f) {
    while (1) {
      int c = getc(f);
      if (c == EOF) {
        if (ferror(f)) perror("échec de getc");
        fclose(f); return;
      }
      printf("%c", c);
    }
  }
  else printf("échec de open(%s): %s\n",
              fname, strerror(errno));
}
```

Position courante dans un flux

Chaque flux a une position courante en octets.

- initialisée à 0 à l'ouverture (sauf mode "a"),
- mise à jour à chaque lecture et écriture,
- un seul curseur pour la lecture et l'écriture.

Prototype

```
long ftell(FILE* stream);
```

Valeur de retour :

- position en octets depuis le début du fichier,
- -1 en cas d'erreur.

Positionnement dans un flux

Si le mode n'est pas "a", on peut changer la position du curseur.

Prototype

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

Effet : se déplace de `offset` octets dans le fichier.
La valeur symbolique `whence` indique un point de référence :

- `SEEK_SET` à partir du début du fichier,
- `SEEK_CUR` à partir de la position courante,
- `SEEK_END` à partir de la fin du fichier.

⇒ `offset` peut être négatif,
on peut se positionner au-delà du fichier !

Exemple

Détermination de la taille d'un fichier

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  FILE* f;
  if (argc <= 1) return 1;
  f = fopen( argv[1], "r" );
  if (!f) printf("fichier %s introuvable\n", argv[1]);
  else {
    fseek(f, 0, SEEK_END);
    long p = fteel(f);
    printf("%li octets\n", p);
  }
}
```

Opérations sur les fichiers

Opérations agissant directement sur le système de fichiers.

Destruction de fichiers

```
#include <unistd.h>
int remove(const char *pathname);
```

Renommage de fichiers

```
int rename(const char *oldpath, const char *newpath);
```

Valeur de retour : 0 en cas de succès, -1 en cas d'erreur (`errno`)

Pas d'ouverture de fichier $\Longrightarrow$ pas de flux `FILE*`.

Les flux stdin, stdout et stderr

Trois flux déjà ouverts sont prédéfinis par `stdio.h` :

- `stdin` entrée standard, en lecture seule,
- `stdout` sortie standard, en écriture seule,
- `stderr` sortie d'erreur, en écriture seule.

Par défaut, `stdin` est branché sur le clavier,
`stdout` et `stderr` sont branchés sur l'écran.

Mais on peut les rediriger depuis le *shell* avec <, > et 2>. (bash)

Exemple

```
$ ./a.out < entree.txt > sortie.txt
```

Les flux stdin, stdout et stderr

`stdin`, `stdout` et `stderr` sont de type `FILE*`.

On peut les utiliser avec les fonctions d'entrée-sortie !

D'ailleurs

`printf(...)`	est un raccourci de	`fprintf(stdout,...)`
`scanf(...)`	est un raccourci de	`fscanf(stdin,...)`

Attention :

- `stdin`, `stdout` et `stderr` n'ont pas de curseur,
  $\Rightarrow$ `ftell` et `fseek` ne fonctionnent pas...
- `feof` fonctionne sur `stdin`, (frappe de contrôle-D)
- `gets` et `puts` ne sont pas équivalents à `fgets` et `fputs`...

Mémoire tampon

Mémoire tampon *(buffer)* **=**
zone de stockage temporaire en attendant un traitement.

Les entrées-sorties sont bufferisées. (raisons d'efficacité)

`stdin` :
le système attend d'avoir une ligne complète avant de la transmettre au programme.

`stdout` :
le C attend d'avoir une ligne complète avant de l'écrire à l'écran.

`stderr` : non bufferisé.

Flux sur des fichiers : bufferisés par blocs.

Vidange

On peut forcer l'écriture du tampon.

Prototype

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

Exemple : `fflush(stdout) ;`

Autre méthode de vidange : `fseek`.

- marche pour le tampon d'écriture et de lecture
  $\Longrightarrow$ entre une lecture et une écriture sur un même fichier, on fait `fseek(f, 0, SEEK_CUR) ;`
- ne marche pas sur `stdout`, `stdin`...