IN 101 - Cours 11

2 décembre 2011

présenté par
Matthieu Finiasz

Les entrées-sorties en C

IN101 - 2011-2012

Ce qu'il vous reste à apprendre en C

- Les entrées-sorties :
  - pour l'instant vous savez utiliser `printf` (et un peu `scanf`) et lire des paramêtres en ligne de commande
  - nous allons voir comment lire/écrire dans des fichiers.
- Les chaînes de caractères :
  - on les a vu simplement comme des tableaux de *`char`*,
  - il existe un ensemble de fonctions pour les utiliser facilement.
- Plein de mots-clefs que vous pourrez rencontrer :
  - tests/sauts : `switch-case`, `goto`, `x? y : z`,
  - déclaration de variables : `extern`, `const`, `static`,
  - instruction spéciales : `asm`.

Descripteurs de fichiers

Le type *FILE**

- Un descripteur de fichier est "lien" vers un fichier qui permet d'y lire ou d'y écrire.
- Un programme C possède 3 descripteurs de fichiers par défaut :
  - `stdin` : l'entrée standard (où lit `scanf`),
  - `stdout` : la sortie standard (où écrit `printf`),
  - `stderr` : la sortie d'erreur (pour afficher des messages d'erreur).
  - → variables globales de type *`FILE*`* incluses avec `stdio.h`
- Un *`FILE*`* est un lien vers un fichier, mais aussi :
  - le mode : écriture ou lecture,
  - la position dans le fichier,
  - une éventuelle erreur : fin de fichier...

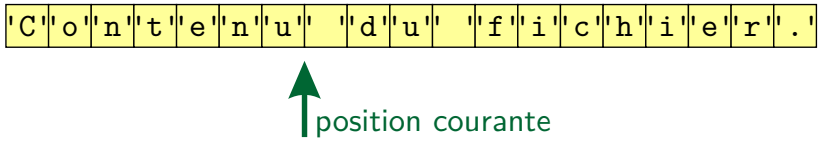
Ouverture/fermeture de fichier

- Avant d'écrire/lire dans un fichier, il faut l'ouvrir :
  - `FILE* fopen(char* path, char* mode)`
    - → ouvre le fichier `path` dans le mode choisi.
- Les différents modes :
  - `"r"` : lecture seule,
    - → commence la lecture au début du fichier,
  - `"w"` : écriture seule,
    - → crée le fichier si besoin, tronque à la longueur 0,
  - `"a"` : "append",
    - → crée le fichier si besoin, sinon, écrit à la fin du fichier,
  - `"r+", "w+", "a+"` : modes lecture et écriture,
    - → `"r+"` : pour remplacer des caractères d'un fichier,
    - → `"w+"` : comme `"r+"` mais tronque,
    - → `"a+"` : écrit toujours à la fin du fichier.

Ouverture/fermeture de fichier

- La fonction `fopen` peut échouer (problème de droits...)
  - → comme pour `malloc`, vérifier qu'elle ne renvoie pas `NULL`.
- La fonction `fclose(FILE* f)` permet ensuite de fermer `f` :
  - on ne peut alors plus lire/écrire dans `f`.
- L'écriture n'est pas instantanée, il y a un buffer :
  - `fclose(f)` finit d'écrire ce qui était en attente,
  - `fflush(f)` force l'écriture dans `f`
    - → sinon, l'écriture n'est pas garantie en cas d'interruption,
  - `fflush(0)` force l'écriture sur `stdout`
    - → en cas d'erreur, un programme peut ne pas avoir tout affiché.
- En général, il est préférable de fermer les fichiers dès qu'on ne s'en sert plus → surtout en mode écriture.

Fonctionnement des descripteurs de fichier



- Le fichier est comme un grand tableau de caractères.
- En plus, il y a une "position courante" :
  - les opérations de lecture/écriture démarrent à cette position,
  - chaque opération de lecture/écriture fait avancer la position,
  - `ftell` : permet de connaître cette position,
    - → en octets depuis de le début du fichier,
  - `fseek`, `rewind` : permettent de se déplacer dans le fichier,
  - `feof` : permet de tester si on a atteint la fin du fichier.
- Beaucoup de fonctions : n'hésitez pas à aller voir le `man`...

Lecture/écriture caractère par caractère

- Il existe beaucoup de façons de lire/écrire,
  - la méthode la plus simple est caractère par caractère.
- `int fgetc(FILE* f)` : lit un caractère dans `f` et le renvoie
  - renvoie -1 si on est au bout du fichier
    - → retourne un `int` pour que -1 et 255 ('ÿ') soient différents.
- `int fputc(int c,FILE* f)` : écrit le caratère `c` dans `f`.

copie de fichier

```
1 int c;
2 FILE* in = fopen("file1", "r");
3 FILE* out = fopen("file2", "w");
4 while ((c = fgetc(in)) != -1) {
5   fputc(c, out);
6 }
7 fclose(in);
8 fclose(out);
```

Lecture/écriture de texte

- La fonction `fprintf` fonctionne comme `printf` :
  - `printf("%d\n",a);` fait comme `fprintf(stdout,"%d\n",a);`
  - on peut mettre un autre descripteur de fichier à la place de `stdout`.
  - `fprintf(stderr,...)` sert à afficher des erreurs.
- Pour écrire des données lisibles par un humain, `fprintf` est en général la meilleure solution.
  - si les données seront lues par un programme, on peut faire mieux.
- La fonction `fscanf` fonctionne comme `scanf` :
  - `scanf("%d\n",&a);` fait comme `fscanf(stdin,"%d\n",&a);`
  - permet de lire des données formatées.

Utilisation de `scanf/fscanf`

- La chaîne de caractères en argument de `fscanf` peut contenir :
  - des conversions : %d, %f...
    - → correspondent à des chiffres (ou autre) dans le fichier,
  - des espaces : espace, tab, retour à la ligne...
    - → correspondent à 0 ou plusieurs espaces dans le fichier,
  - des caractères simples
    - → correspondent au même caractère dans le fichier
- `fscanf` renvoie le nombre de conversions réussies :
  - pensez à vérifier la valeur retournée,
  - permet de lire des données dans une boucle :

```
1 int total=0, tmp;
2 FILE* f = fopen("file", "r");
3 while (fscanf(f, "%d ", &tmp) != 0) {
4   total += tmp;
5 }
```


Utilisation de `scanf/fscanf`

- Pour lire la date : `"2 décembre 2011, 13:55:17"`
  - → `scanf("%d %s %d, %d:%d:%d",&j,&M,&a,&h,&m,&s)`
  - %s s'arrête au premier espace rencontré,
  - ⚠ si la `','` ou un `':'` manque, la suite n'est pas lue.
- Quelques autre fonctionnalités :
  - %ms laisse `scanf` allouer la mémoire nécessaire,
  - %*d lit un entier, mais ne le stocke pas,
  - %[a-z] permet de lire des lettres minuscules,
  - %5c mermet de lire 5 caractères.

Lecture/écriture binaire

- Les fonctions `fread` et `fwrite` permettent de copier un tableau de données brutes d'un fichier vers la mémoire (ou inversement).
- Par exemple (on suppose `tab` assez grand) :
  - pour lire 40 octets dans un fichier `f` :
    - → `fread(tab, 1, 40, f)`,
  - pour lire 25 entiers dans un fichier `f` :
    - → `fread(tab, sizeof(int), 25, f)`,
  - pour écrire 12 doubles dans `f` :
    - → `fwrite(tab, sizeof(double), 12, f)`.
- Il s'agit ici de données brutes :
  - on ne peut pas les lire facilement à la main,
  - mais il n'y a pas de conversion nécessaire,
    - → beaucoup plus efficaces que `fprintf` et `fscanf`.

Chaînes de caractères

Chaînes de caractères

- Une chaîne de caractères est un *char** se terminant par un '\0' :
  - le caractère de fin permet de mesurer sa longueur : `strlen`,
  - permet de les manipuler presque comme un type natif,
    - → pas besoin de passer la longueur en argument.

⚠ Ce sont quand même des tableaux avec une taille allouée :
- les fonctions continuent à lire jusqu'à trouver un '\0'
  - → risque de dépassement !
- la plupart des fonctions existent en 2 versions :
  - la version normale (non protégée) qui lit jusqu'au '\0',
  - une version protégée qui prend une limite en argument,
    - → regardez le man pour voir les arguments.

Fonctions de la bibliothèque `string.h`

- `man string.h` donne une liste des fonctions :
  - `strcmp`/`strncmp` : compare 2 chaînes (ordre alphabétique),
  - `strcpy`/`strncpy` : copie une chaîne de caractères,
  - `strcat`/`strncat` : concatène 2 chaînes,
  - `strchr`/`strrchr` : première/dernière occurence d'un *char*
  - `strstr` : recherche une chaîne dans une autre

- C'est peu par rapport à d'autres langages (Perl, PHP, Python...)
  - le C n'a pas été fait pour manipuler des chaînes de caractères,
  - mais c'est suffisant pour des choses simples,
    - → lire des arguments/options en ligne de commande.

Lecture d'arguments en ligne de commande

- Voici une façon d'ajouter des options à un programme.

```
1  int main(int argc, char* argv[]) {
2    int i, d=0;
3    FILE* input = NULL;
4    for (i=1; i<argc; i++) {
5      if (strcmp(argv[i], "-d") == 0) {
6        d = 1;
7      } else if (strcmp(argv[i], "-i") == 0) {
8        i++;
9        input = fopen(argv[i], "r");
10     } else {
11       fprintf(stderr,"Argument '%s' non reconnu.\n", argv[i]);
12       return 1;
13     }
14   }
15   if (input == NULL) {        // argument obligatoire
16     fprintf(stderr, "Donnez un nom de fichier avec -i.\n");
17     return 2;
18   }
19   ...
```

Entrée/sorties et chaînes de caractères

- Les chaînes de caractères peuvent aussi servir de buffer pour lire dans un fichier :
  - `fgets` permet de lire une ligne d'un fichier (texte),
    - → stocke la ligne dans une chaîne de caractères,
  - `sscanf` permet de faire comme `fscanf` depuis une chaîne,
    - → attention, pas de position de lecture qui avance...
- On peut faire la même chose en écriture :
  - `sptinf` permet d'écrire dans un *`char*`*,
    - → renvoie le nombre de caractères écrits (comme `printf`)
  - `fputs` permet d'écrire une ligne dans un fichier.
- `fgets` et `fputs` peuvent aussi être remplacés par `fread` et `fwrite`.

Quelques autres mots-clef du langage C

IN101 - 2011-2012

IN101 - 2011-2012

Liste des mots-clef

- Le C ANSI de 1989 reconnais les mots-clef :
  auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while
- Certains ne servent à peu près jamais, d'autres peuvent servir
  - → ceux que l'on n'a pas encore vus ne sont pas indispensables pour programmer.
- Le "GNU C Reference Manual" explique tout cela :
  http://www.gnu.org/s/gnu-c-manual/gnu-c-manual.html

IN101 - 2011-2012

Tests successifs `switch-case`

- Pour remplacer plusieurs `if-else` successifs, on peut utiliser un `switch` :

```
1  switch(a) {
2  case 0:
3    printf("a vaut 0\n");
4    break;
5  case 1:
6    printf("a vaut 1\n");
7    break;
8  default:
9    printf("a n'est ni 0 ni 1.\n");
10 }
```

- Si un `case` ne finit pas par un `break`, la suite est aussi executée,
  - → ça peut servir des fois...

Le saut goto

- Un goto permet de faire un saut à un autre endroit du code :
  - on peut tout programmer avec des goto à la place des boucles
    → c'est ce que fait le compilateur
  - on peut aussi tout programmer sans jamais utiliser de goto
    → c'est difficile à lire (humain) et à optimiser (compilateur)

Quitter une double boucle

```
1 for (i=0; i<n; i++) {
2   for (j=0; j<n; j++) {
3     if (matrice[i][j] == 1) {
4       goto sortie;
5     }
6   }
7 }
8 sortie:
9 printf("Le 1 est en position [%d,%d].\n", i, j);
```


Forme compacte des tests if-else

- La commande `(test) ? val1 : val2` s'évalue en :
  - `val1` si le test est vrai,
  - `val2` si le test est faux.
- Permet de remplacer le code :

```
1 if (a==1) {
2   b = 3;
3 } else {
4   b = 5;
5 }
```

par :

```
1 b = (a==1) ? 3 : 5;
```


Variables externes

- Pour utiliser une fonction d'une bibliothèque, on inclut le `.h` associé
  - contient des prototypes de fonctions,
    → indique que le code des fonctions sera donné plus tard,
  - `extern` permet la même chose pour des variables globales
    → `extern int a;` pour la variable globale a
  - la variable est réellement déclarée dans le `.c` de la bibliothèque.
- Sans le `extern` si l'on définit une variable globale dans le `.h`
  - elle sera créée dans chaque `.c` qui l'inclut,
  - conflit au moment de l'édition des liens.
- Un exemple d'utilisation est `stdin` dans `stdio.h`.

Variables constantes

- Le mot clef `const` permet de dire qu'une variable ne sera pas modifiée dans la fonction :
  - variables locales au moment de la déclaration,
    → `const int k = 10;`
  - argument d'une fonction,
    → `char* strcat(char* dest, const char* src);`
- Le compilateur fait :
  - une erreur si l'on essaye de modifier une variable `const`,
  - un warning si l'on passe une variable `const` comme argument non `const` d'une fonction.

⚠ Vérification faite à la compilation, pas à l'execution :
→ ça ne sert à rien, mais on peut contourner un `const`...

Variables statiques

- Une variable `static` est une variable globale, locale à une fonction :
  - cette variable n'existe que dans la fonction,
  - elle garde sa valeur entre les différents appels,
    - → permet d'éviter les conflits de noms de variables globales.
- Permet aussi de déclarer des variables/fonctions "locales" à un fichier source :
  - utilisables normalement dans le fichier,
  - invisible depuis un autre fichier,
    - → utilisable pour des fonctions internes de bibliothèques.

Utilisation d'instructions assembleur

- La fonction `asm` permet de facilement executer des commandes assembleur depuis du code C.
- Peut servir dans deux contextes différents :
  - pour écrire une portion de code très optimisée
    - → nécessite de bien connaître l'assembleur...
  - pour accéder à des fonctions internes du processeur
    - lire le compteur de cycles,
    - décalage cyclique sur certains processeurs,
    - nouvelles instructions "AES" : multiplications dans GF(2^m)...
- Cela peut faire gagner beaucoup en efficacité, sans avoir à vraiment programmer en assembleur.

```
asm("movl %1, %%eax;"
    "movl %%eax, %0;" : "=r" (b) : "r" (a): "%eax");
```


Ce qu'il faut retenir de ce cours

- La plupart des "vrais" programmes ont besoin d'entrées/sorties :
  - en C cela repose sur les descripteurs de fichiers,
    - → un *FILE** est un tableau d'octets et une position
  - lire/écrire dans des fichier est assez facile,
    - → les quelques commandes qu'on a vu suffisent,
  - il y a plusieurs façons de faire cela
    - → les plus simples ne sont pas forcément les plus efficaces...
- La bibliothèque standard du C est assez peu fournie :
  - gestion des chaînes de caractères basique,
    - → risques de dépassements, peu de fonctions...
  - entrées/sorties assez rudimentaires,
    - → il faut souvent reprogrammer des choses (buffers...)