Programmation système I
Les entrées/sorties

DUT 1re année

Université de Marne La vallée

Les entrées-sorties : E/O

Entrées/Sorties : Opérations d'échanges d'informations dans un système informatique.

Les systèmes d'exploitations ont des mécanismes/fonctions d'entrées sorties, les langages de programmation aussi. Dans le langage C, la plupart de ces fonctionnalités se situent dans la librairie `<stdio.h>`. (stdio : standard input output).

Sans entrées/sorties, il est impossible de produire des programmes évolués ayant de grands impacts sur l'environnement, le système et la mémoire.

Les flux d'informations

Toutes interactions avec l'environnement et la mémoire (et leurs modifications) nécessite de récupérer ou de constituer un flux d'informations.

Flux classiques :

- Texte tapé au clavier (entrée)
- Mouvement de la souris et clic (entrée)
- Fichiers sur le disque dur (entrée/sortie)
- Position de la souris sur l'écran (entrée/sortie)
- Texte affiché sur l'écran (sortie)
- Réseau (entrée/sortie)
- Webcam (entrée)
- ...

Les fichiers dans le langage C

Il est possible de manipuler (créer / lire / écrire / détruire) des fichiers du disque dur via des fonctions d'Entrées/Sorties de la librairie `<stdio.h>`.

Dans le langage C, les fichiers sont manipulables via un pointeur sur une structure `FILE`.

Procédé de manipulation de fichier:

- Ouverture du fichier, constitution de l'objet (`FILE *`) et placement du pointeur.
- Fonctions d'Entrées/Sorties du langage C pour modifier des caractères sur le pointeur et déplacer ce pointeur dans le fichier ouvert.
- Fermeture du fichier.

Le dernier caractère d'un fichier ouvert est **`EOF`** pour (End Of File).

Ouverture d'un fichier: `fopen`

Avant toute opération sur un fichier, il est nécessaire d'obtenir un pointeur de type `FILE *`.

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

`-path` contient l'adresse du fichier à ouvrir, `-mode` précise le mode d'ouverture.

- `"r"`: mode lecture, pointeur au début
- `"w"`: mode écriture, pointeur au début, création du fichier si non existant, effacement sinon. (Chevron `UNIX` >)
- `"r+"`: mode lecture-écriture, pointeur au début
- `"w+"`: mode lecture-écriture, pointeur au début, création du fichier si non existant, effacement sinon.
- `"a"`: mode écriture, pointeur à la fin, création du fichier si non existant. (Chevron `UNIX` >>)
- `"a+"`: mode lecture-écriture, pointeur à la fin, création du fichier si non existant.

Fermeture d'un fichier: `fclose`

Une fois les modifications terminées, il convient de fermer les fichiers ouverts.

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

retourne 0 en cas de succès et **`EOF`** sinon.

Réouverture d'un fichier: `freopen`

Ré-ouvre un fichier en fermant l'instance précédemment ouverte.

```
FILE *freopen(const char *path, const char *mode, FILE
*stream);
```

A l'origine, cette fonction est conçue pour permettre les redirections. On l'utilisera rarement en pratique mais son recours est parfois contraint.

Les flux standards

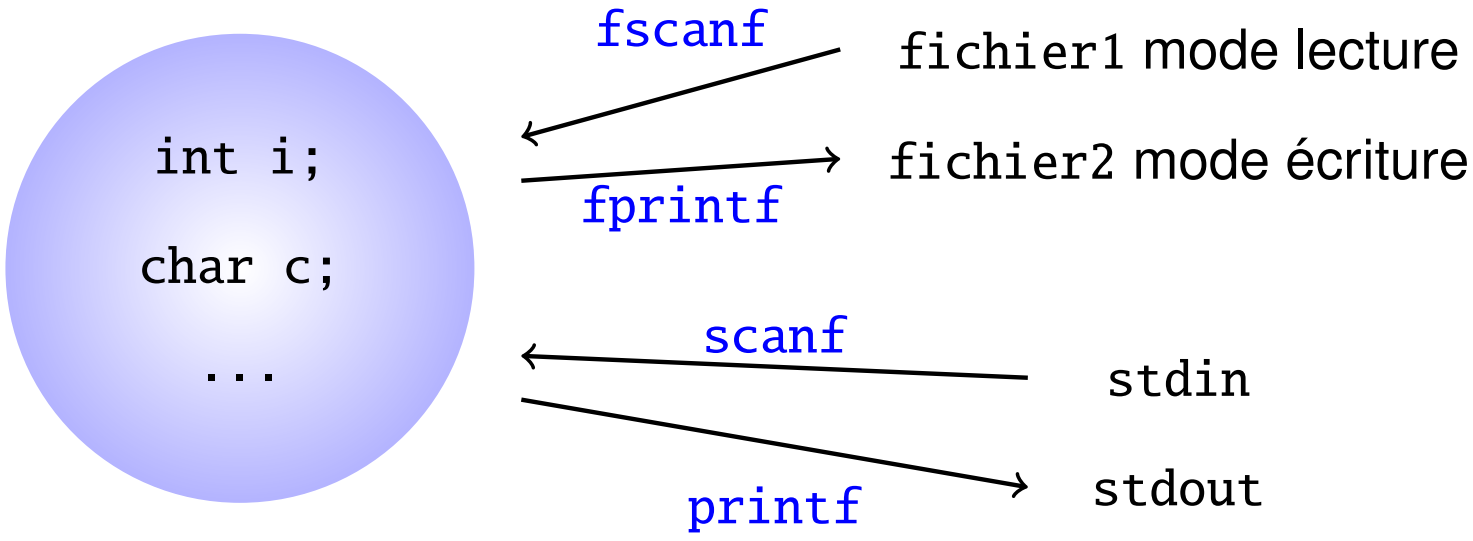
Dans le langage C et pour `UNIX` (programmé en C), il existe des flux standards qui se comportent comme des fichiers pour lesquels l'utilisateur n'a pas à se soucier de l'ouverture et fermeture.

- stdin: (standard input) est l'entrée standard. Tout caractère tapé au clavier durant l'exécution d'un programme se retrouve dans ce flux.
- stdout: (standard output) est la sortie standard. Tout caractère écrit dans ce flux sera affiché dans la console lors de l'exécution d'un programme. ATTENTION A LA BUFFERISATION.
- stderr: (standard error) est la sortie d'erreur standard.

Les fonctions d'Entrées/Sorties du langage C opérant sur les fichiers et flux standards sont très liées. (printf - fprintf, scanf - fscanf, ...).

Fonctions d'Entrées/Sorties du langage C

Les fonctions d'entrées/sorties servent à écrire et récupérer des données dans les fichiers ouverts et les flux standards.



Les fonctions primitives systèmes sont très difficiles à utiliser (`open`, `close`, `read`, `write`). Notamment, elles ne font que de la lecture octet par octet (8 bits). On utilisera les fonctions E/S formatées de `<stdio.h>`.

`scanf` : lecture formatée sur l'entrée standard

```
int scanf(const char *format, ...);
```

On remplace les ... par une liste d'adresses de conteneurs mémoires adaptés. Si on a une variable `int i`, l'adresse **`&i`** de `i` est un conteneur mémoire adapté à la lecture d'un entier.

Type	Lettre
int	%d
long	%ld
float - double	%f - %lf
char	%c
string (char*)	%s

```
1 char nom[50];
2 int age;
3 printf("Saisissez votre nom, un espace, puis votre age");
4 scanf("%s %d", nom, &age);
```

`scanf` retourne le nombre de variables affectées par la saisie, cela permet de vérifier le bon déroulement de la saisie.

`printf` : écriture formatée sur la sortie standard

```
int printf(const char *format, ...);
```

On remplace les ... par une liste de variables adaptées aux différents types apparaissant dans le format. Encore une fois, il faut respecté l'ordre de lecture de gauche à droite.

Le texte formaté avec les bonnes substitutions est ensuite affiché à l'écran (modulo la bufferisation).

```
1  float t=37.12;
2
3  printf("La temperature de %s est %f", "Martin", f);
```

En cas de succès, `printf` retourne le nombre de caractères écrits sur la sortie standard (sans compter le caractère de fin de chaîne).

`fscanf` : lecture formatée sur un flux

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

On remplace les ... par une liste d'adresses de conteneurs mémoires adaptés. On récupère ainsi des données dans le flux qui se place en valeur des adresses de variables données.

```
1    char nom[80];
2    int age;
3    FILE * fichier_entree;
4
5    fichier_entree = fopen("nom_age.txt","r");
6    fscanf (fichier_entree , "%s %d", nom, &f);
```

`fscanf` retourne le nombre de variables affectées par la saisie, cela permet de vérifier le bon déroulement de la saisie.

Les flux standards se comportent comme des fichiers ouvert, ainsi : `scanf(format, ...)` est équivalent à `fscanf(stdin, format, ...)`.

`fprintf` : écriture formatée sur un flux

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

On remplace les ... par une liste de variables adaptées aux différents types apparaissant dans le format. Encore une fois, il faut respecté l'ordre de lecture de gauche à droite.

Le texte formaté avec les bonnes substitutions est ensuite écrit sur le flux.

```
1    int age=12;
2    FILE * fichier_sortie;
3
4    fichier_sortie = fopen("nom_age.txt","w");
5    fscanf (fichier_sortie , "%s %d", "Martin", age);
```

En cas de succès, `fprintf` retourne le nombre de caractères écrits sur le flux (sans compter le caractère de fin de chaîne).

`printf(format, ...)` est équivalent à `fprintf(stdout, format, ...)`.

Il existe de nombreuse autre fonctions d'Entrées/Sorties.

Lorsque l'on ne connaît pas/plus une fonction `foo` en langage C, **LE SEUL RÉFLEXE:** on tape `man 3 foo` dans un terminal.

Autre fonctions: fgetc, fgets, getc, getchar, gets, ungetc, fputc, fputs, putc, putchar, puts, ...

`man 3 fgetc`

```
GETS(3)                    Linux Programmer's Manual                    GETS(3)

NAME
       fgetc,  fgets,  getc,  getchar,  gets, ungetc - input of characters and
       strings

SYNOPSIS
       #include <stdio.h>

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

Soyez astucieux et perspicaces

Les noms de fonctions n'ont pas été choisis au hasard.

- `print-f` : écrire, format
- `f-scan-f` : flux, analyser, format
- `f-get-c` : flux, obtenir, caractère
- `put-char` : placer, caractère
- `f-put-s` : flux, placer, chaîne de caractères

- Lorsque le nom ne commence pas par un `f`, la fonction affecte les flux standards, sinon elle aura un flux parmi ses arguments.
- En fin de nom, `c` et `char` désigne des caractères, `s` désigne une chaîne de caractères, `f` que la fonction peut prendre des écritures formatées (types différents).

Des paramètres dans la fonction `main`

Voici le prototype standard d'un `main` acceptant des arguments:

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

- `argc` est un entier, il donne le nombre d'arguments passés à l'exécutable (en comptant le nom de l'exécutable comme premier argument).
- `argv` est un tableau de chaînes de caractères.
  - `argv[0]`: le nom de l'exécutable (dépend du nom donné à la compilation).
  - `argv[1]`: le nom du premier argument.
  - `argv[2]`: le nom du second argument.
  - ...

Similarité avec le langage du `bash`:
`$#` se comporte comme `argc`
`$0` se comporte comme `argv[0]`
`$1` se comporte comme `argv[1]`
... (et tout est, a priori, chaînes de caractères)

Pour le programme suivant : `test.c`

```c
#include <stdio.h>

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

  for (i=0 ; i<argc ; i++){
    printf("%s\n", argv[i]);
  }
  return 0;
}
```

```
nicolas@lancelot:~/test$ gcc -o test test.c -Wall -ansi
nicolas@lancelot:~/test$ ./test fich1.txt fich2.txt 23
./test
fich1.txt
fich2.txt
23
```

Les arguments des exécutables passés par la console sont des chaînes de caractères. Toutefois, le langage C possède des fonctions de conversions entre ces types.

`int atoi(const char *nptr);`
retourne un entier à partir d'une chaîne de caractères.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void){
5    char* nombre="12543";
6
7    printf("%d\n", atoi(nombre));
8    return 0;
9  }
```

```
nicolas@lancelot:~/test$ gcc -o test test.c -Wall -ansi
nicolas@lancelot:~/test$ ./test
12543
```

Comment alors faire un exécutable en console qui attend des entiers en arguments et retourne la somme de ces derniers.

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

int main(int argc, char* argv[]){
  int i,somme=0;

  for (i=1 ; i<argc ; i++){
    somme += atoi(argv[i]);
  }
  printf("%d\n", somme);
  return 0;
}
```

```
nicolas@lancelot:~/test$ gcc -o test test.c -Wall -ansi
nicolas@lancelot:~/test$ ./test
0
nicolas@lancelot:~/test$ ./test 1 2
3
nicolas@lancelot:~/test$ ./test 1 2 5 5
13
nicolas@lancelot:~/test$ ./test foo bla
0
```

L'œuf ou la poule, `UNIX` ou le langage C ?

Le système d'exploitation `UNIX` et le langage C ont été conçu de manière relativement conjointe. Notamment, les flux standards (stdin, stdout, stderr) SONT LES MÊMES!

Un exécutable compilé à partir de source écrite en C bien programmé peut s'enchaîner dans des pipes `UNIX` à volonté.

conséquences:
Toutes les données récupérées sur l'entrée standard (`scanf`, `getchar`, ...) peuvent en fait être envoyé au programme via un chevron entrant < d'`UNIX` ou un pipe `|`.

```c
#include <stdio.h>

int main(void){
  char nom[50];
  int age;

  printf("Donner votre nom et votre age!\n");
  scanf("%s %d", nom, &age);
  printf("Ok, vous etes %s et vous avez %d ans\n", nom, age);
  return 0;
}
```

```
nicolas@lancelot:~/test$ gcc -o test test.c -Wall -ansi
nicolas@lancelot:~/test$ echo "Dupont 45"
Dupont 45
nicolas@lancelot:~/test$ echo "Dupont 45" | ./test
Donner votre nom et votre age!
Ok, vous etes Dupont et vous avez 45 ans
nicolas@lancelot:~/test$ ./test < nom_age.txt
Donner votre nom et votre age!
Ok, vous etes Durant et vous avez 78 ans
```

```c
#include <stdio.h>

int main(void){
  int c=getchar();

\* Recopiage de l entree standard sur la sortie standard *\
  while (c != EOF){
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

```
nicolas@lancelot:~/test$ gcc -o test test.c -Wall -ansi
nicolas@lancelot:~/test$ echo "pouet fou bla" > fichier
nicolas@lancelot:~/test$ echo "une seconde ligne" >> fichier
nicolas@lancelot:~/test$ ./test < fichier
pouet fou bla
une seconde ligne
nicolas@lancelot:~/test$ ./test < fichier | ./test
pouet fou bla
une seconde ligne
```

Bufferisation

La sortie standard est bufferisée: Les fonctions d'Entrées/Sortie de `<stdio.h>` n'envoient pas directement les éléments sur l'écran.

`UNIX` incite à l'enchaînement de commandes ou exécutables via des pipes. Mais la communication entre les processus a un coût. Surtout si l'on perd son temps à communiquer sans arrêt des messages tout petits.

`printf` écrit sur le buffer de la sortie standard (de taille 8192 Octets par défaut) et une fois le buffer plein (ou un retour à ligne, ou un `flush`, ou ...), on sollicite le processus d'affichage à l'écran.

Note: La sortie d'erreur n'est pas bufferisé et affiche les caractères dès l'ajout dans le flux.

```c
#include <stdio.h>

int main(void){
  char* nom="Jean";

  printf("Ou est l'erreur de segmentation");
  nom[34] = 123;
  return 0;
}
```

Ce programme donne le comportement suivant

```
nicolas@lancelot:~/test$ gcc -o test test.c -Wall -ansi
nicolas@lancelot:~/test$ ./test
Erreur de segmentation (core dumped)
```

Où est passé le `printf` ? l'erreur de segmentation se trouve avant ?

Solution 1:

```
1  #include <stdio.h>
2
3  int main(void){
4    char* nom="Jean";
5
6    /* Un saut de ligne force la vidange du buffer */
7    printf("Ou est l'erreur de segmentation\n");
8    nom[34] = 123;
9    return 0;
10 }
```

Solution 2:

```
1  #include <stdio.h>
2
3  int main(void){
4    char* nom="Jean";
5
6    /* Drapeau de debogage sur la sortie d'erreur */
7    fprintf(stderr, "Ou est l'erreur de segmentation");
8    nom[34] = 123;
9    return 0;
10 }
```