Traitement de fichiers, compilation séparée, structures et autres bonnes pratiques en C

Lundi 14 Mai 2018

Michael FRANÇOIS

francois@esiea.fr

https://francois.esiea.fr/

Les fichiers

Les fichiers

- Les entrées/sorties peuvent être souvent redirigées vers un fichier.
- Traditionnellement on distingue deux types d'accès :
  - **accès séquentiel :** on accède aux données dans l'ordre du stockage
  - **Accès direct :** on se positionne à l'endroit précis que l'on veut lire
- Généralement, on peut choisir entre deux manières de représenter l'information dans un fichier :
  - Sous forme **brute** (*i.e.* binaire) : consiste à recopier dans le fichier l'information telle qu'elle figure en mémoire.
  - Sous forme **formatée** : consiste à exprimer l'information sous la forme d'une suite de caractères (nombres en décimal, ...).

Manipulation

- Le type `FILE` permet la manipulation d' "objet" de type fichiers.
- L'ouverture d'un fichier se fait via la fonction `fopen` :

```
FILE * fopen(char * nom_fichier, char * mode);
```

- Les différents modes :
  - `r` : lecture seule
  - `w` : écriture seule (si le fichier existe, il est vidé)
  - `a` : append, écriture à la suite de ce qui existait déjà dans le fichier
  - `r+, w+, a+` : lecture/écriture

Exemple :

```c
#include<stdio.h>

void main(void)
{
 FILE * FIC; /* FIC est le nom interne du fichier */
 FIC = fopen("donnees.txt", "r"); /* ouverture de donnees.txt
                                     en mode lecture seule */
if (FIC==NULL)
{
 printf("ERREUR : le fichier n'existe pas ! \n");
}
else
{
 printf("Fichier correctement ouvert ! \n");
}

/* ---ici traitement sur le fichier--- */

fclose(FIC); /* Fermeture du ficher après traitement */
}
```

La fonction `fprintf`

• La fonction `fprintf` envoie une sortie formatée vers un fichier. Contrairement à `printf` qui écrit sur la sortie standard.

• Syntaxe :

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

• La valeur de retour de `fprintf` pourra être utilisée pour détecter d'éventuelles erreurs :

- il y aura erreur dès lors que cette valeur est négative.
- sinon elle indiquera le nombre de caractères écrits dans le fichier.

Exemple :

```c
#include<stdio.h>

void main(void)
{
 int retour;
 FILE * FIC; /* FIC est le nom interne du fichier */
 FIC = fopen("donnees.txt", "w"); /* ouverture de donnees.txt
                                     en mode écriture seule */
if (FIC==NULL)
{
 printf("ERREUR : le fichier n'existe pas ! \n");
}
else
{
 printf("Fichier correctement ouvert ! \n");
}

retour = fprintf(FIC, "ESIEA %d\n", 2014);
printf("retour = %d\n", retour);

fclose(FIC); /* Fermeture du ficher après traitement */
}
```

```
$ ./EXEC
Fichier correctement ouvert !
retour = 11
```

La fonction `fscanf`

- La fonction `fscanf` lit et formate une entrée depuis un fichier. Contrairement à `scanf` qui lit depuis la console.

- Syntaxe :

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

- La valeur de retour de `fscanf` pourra être utilisée pour détecter d'éventuelles erreurs :
  - il y aura erreur dès lors que cette valeur est négative.
  - sinon elle indiquera le nombre de valeurs lues depuis un fichier.

Exemple :

```c
#include<stdio.h>

void main(void)
{
 int retour, AN1, AN2;
 char C, Nom_fic[20];
 printf("Saisir le nom du fichier : \n");
 scanf("%19s", Nom_fic);
 FILE * FIC; /* FIC est le nom interne du fichier */
 FIC = fopen(Nom_fic, "r"); /* ouverture de donnees.txt
                               en mode lecture seule */
                            /* donnees.txt contient :
                               2013 2014 E
                            */
if (FIC==NULL)
{printf("ERREUR : le fichier n'existe pas ! \n");}
else
{printf("Fichier correctement ouvert ! \n");}

retour = fscanf(FIC, "%d %d %c\n", &AN1, &AN2, &C);
printf("retour = %d\n%d %d %c\n", retour, AN1, AN2, C);
fclose(FIC); /* Fermeture du ficher après traitement */
}
```

```
$ ./EXEC
Saisir le nom du fichier :
donnees.txt
Fichier correctement ouvert !
retour = 3
2013 2014 E
```

La fonction `fseek`

• Cette fonction permet de placer le curseur à une position voulue dans le fichier.

• Syntaxe :

```
int fseek (FILE * fic, long déplacement, int origine) ;
```

Les origines utilisées :

- `SEEK_SET` : début du fichier (déplacement ≥ 0)
- `SEEK_CUR` : position actuelle du pointeur
- `SEEK_END` : fin du fichier

Remarque : cette fonction n'est utilisable que si l'on connaît la taille des données dans le fichier (impossible d'aller directement à une ligne donnée d'un texte si on ne connaît pas la longueur de chaque ligne).

Exemple :

```c
#include<stdio.h>
void main(void)
{
 int retour, AN;
 char C, Nom_fic[20];
 printf("Saisir le nom du fichier : \n");
 scanf("%19s", Nom_fic);
 FILE * FIC; /* FIC est le nom interne du fichier */
 FIC = fopen(Nom_fic, "r"); /* ouverture de donnees.txt
                               en mode lecture seule */
                            /* donnees.txt contient :
                               2013 2014 2015 2016
                            */
if (FIC==NULL) {printf("ERREUR : le fichier n'existe pas ! \n");}
else {printf("Fichier correctement ouvert ! \n");}

fseek(FIC, 15, SEEK_SET); /* on decalle le curseur de 15 positions */
fscanf(FIC, "%d", &AN);
printf("AN = %d\n", AN);

fseek(FIC, 0, SEEK_SET); /* retour au debut du fichier */
fscanf(FIC, "%d", &AN);
printf("AN = %d\n", AN);
fclose(FIC); /* Fermeture du ficher après traitement */
}
```

```
$ ./EXEC
Saisir le nom du fichier :
donnees.txt
Fichier correctement ouvert !
AN = 2016
AN = 2013
```

La fonction `fgets`

• Cette fonction permet de lire une suite de caractères à partir d'un flux quelconque.

• Syntaxe :

```
char* fgets (char* chaine, int n, FILE* flux) ;
```

• Le nombre maximal de caractères à lire est de $n - 1$, car le dernier caractère est réservé au zéro de fin de chaîne.

• Concernant l'adresse de retour, `fgets` fournit l'adresse de la chaîne lue, lorsque la lecture s'est bien déroulée. Elle renvoie le pointeur `NULL` en cas d'erreur.

Exemple :

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


int main (int argc, char* argv[])
{
 int l=6;
 char CHAINE[l];
 do
 {
   printf("Saisir une chaîne : ");
   fgets (CHAINE, l, stdin);
   puts(CHAINE);
 }while (strlen(CHAINE) != 1);

 return 0;
}
```

La fonction puts envoie sur la sortie standard les différents caractères de la chaîne d'adresse CHAINE (sans le zéro de fin), puis elle transmet un caractère de fin de ligne ('\n')

```
Saisir une chaîne : IVRY
IVRY

Saisir une chaîne : ESIEAIVRY94200
ESIEA
Saisir une chaîne : IVRY9
Saisir une chaîne : 4200

Saisir une chaîne :
```

On peut retrouver d'autres fonctions pour le traitement de fichier :

- `fwrite` : pour écrire sous forme binaire des données dans un fichier.
- `fread` : pour lire sous forme binaire des données depuis un fichier.
- `fputs` : transmet une chaîne à un flux quelconque.
- `fputc` : envoie un caractère sur un flux quelconque.
- `fgetc` : lit un caractère sur un flux quelconque.
- `ftell` : permet d'obtenir l'adresse d'un emplacement déjà parvenu.
- ...

Structures de données

Structures

• Un objet de type structure est constitué de la réunion d'un ensemble de valeurs, qui ne sont pas nécessairement d'un même type.

• L'emploi de structures permet d'accroître la clarté des programmes en rassemblant dans un même objet des informations ayant un lien, par exemple :

- les différentes infos associées à une personne : nom, prénom, adresse, date de naissance, e-mail, etc.
- les différentes infos associées à un point d'un plan : nom, abscisse, ordonnée, etc.

- Exemple
  on veut manipuler des
  nombres complexes

```
/* z = x + iy */
```

```
/* Méthode 1 */
typedef struct{
    double reel;
    double imag;
}complex1;

complex1 z1;
```

```
/* Méthode 2 */
struct complex2{
    double reel;
    double imag;
};

struct complex2 z2;
```

```
/* Méthode 3 */
struct{
    double reel;
    double imag;
} z3;
```

On accède aux membres de la structure avec l'opérateur " ."

Exemple :

```c
#include<stdio.h>

typedef struct /* Méthode 1 */
{
 double reel;
 double imag;
}complex1;


void main(void)
{
 complex1 z1;
 z1.imag = 3.0;
 z1.reel = 2.0;
 printf("z1 = %.1lf + %.1lfi\n",z1.reel, z1.imag);
}
```

```
z1 = 2.0 + 3.0i
```

Structures & Pointeurs

```c
#include<stdio.h>

typedef struct
{
 double reel;
 double imag;
}complex1;

typedef complex1 * complex1PT;

void main(void)
{
 complex1 z1;
 z1.imag = 0.0;
 z1.reel = 1.0;

 complex1PT zPT; /* équiv. à complex1 * zPT; */
 zPT = &z1;
 (*zPT).imag = 9.0; /* équiv. à  zPT->imag = 9.0; */
 (*zPT).reel = 4.0; /* équiv. à  zPT->reel = 4.0; */

 printf("partie reelle : %.2lf\n", zPT->reel);
 printf("partie imagi. : %.2lf\n", zPT->imag);
}
```

```
partie reelle : 4.00
partie imagi. : 9.00
```

Structures & allocation dynamique

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

typedef struct
{
 double reel;
 double imag;
}complex1;

typedef complex1 * complex1PT;

void main(void)
{
 complex1 z1;
 z1.imag = 8.0;
 z1.reel = 15.0;
 complex1PT zPT;
 zPT = (complex1PT) malloc(10 * sizeof(complex1));
 zPT[5] = z1;
 printf("Partie relle : %.2lf\n", zPT[5].reel);
 printf("Partie imagi : %.2lf\n", zPT[5].imag);
}
```

```
Partie relle : 15.00
Partie imagi : 8.00
```

Structures & passage d'arguments

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

typedef struct
{
 double reel;
 double imag;
}complex1;

typedef complex1 * complex1PT;

void reset(complex1PT zPT)
{
 zPT->reel = 0.0;
 zPT->imag = 0.0;
}

void main(void)
{
 complex1 Z2;
 Z2.reel = 4.8;
 Z2.imag = 5.2;
 reset(&Z2);
 printf("Z2 = %.2lf + %.2lfi\n", Z2.reel, Z2.imag);
}
```

```
Z2 = 0.00 + 0.00i
```

Quelques bonnes pratiques à adopter

Quelques bonnes pratiques à adopter

Quelques bonnes pratiques à adopter, pour une programmation plus sûre et plus efficace en langage C.

Déclaration et initialisation

Déclarer plusieurs variables dans une même déclaration peut être source de confusion par rapport à leurs types et leurs valeurs initiales.

À ne pas faire :

```
int i, j = 1 ; /* peut prêter confusion, car seulement j est initialisé à 1 */
```

À faire :

```
int i, j ; /* déclarations */
i=1; j=1 ; /* initialisation de i et j à 1 */
```

À ne pas faire :

```
char * src, c ; /* ici src est de type char *, mais c de type char */
```

À faire :

```
char * src ;
char c ;
```

À ne pas faire :

```
int ID_O ; /* lettre majuscule O */
int ID_0 ; /* chiffre zéro */
```

À faire :

```
int ID_A ;
int ID_B ;
```

Expressions

À ne pas faire :

```
X & 1 == 0 /* cette expression est prise comme X & (1 == 0) */
```

À faire :

```
(X & 1) == 0 /* permet une meilleure évaluation */
```

À ne pas faire :

```
int login ;
if (invalid login())
login=0 ;
else
printf("login valide \n ") ; /* pour le débogage */
login=1 ; /* cette ligne sera toujours exécutée, même si le login est invalide */
```

À faire :

```
int login ;
if (invalid login())
{ login=0 ; }
else
{ login=1 ;}
```

Nombres entiers

Il est important de toujours bien gérer le format des données utilisées par votre programme, pour éviter des erreurs. Voilà un exemple parmi tant d'autres.

À ne pas faire :

```c
int f(void) {
  FILE *fp;
  int x;
/* ... */
  if (fscanf(fp, "%ld", &x) < 1) {
    return -1; /* Indicate failure */
  }

/* ... */
  return 0;
}
```

À faire :

```c
int f(void) {
  FILE *fp;
  int x;
/* Initialize fp */
  if (fscanf(fp, "%d", &x) < 1) {
    return -1; /* Indicate failure */
  }

/* ... */
  return 0;
}
```

NB : Cela marche bien-sur pour des modèles où:
sizeof (int) == sizeof (long), mais pour les autres cela entraîne un débordement de tampon (i.e. buffer overflow).

Nombres réels

Quand on manipule les nombres réels, il faut faire attention à la limitation de la precision faite par la machine. Voici un petit exemple qui illustre bien la précision selon le modèle utilisé.

```
#include <stdio.h>
int main(void)
{
float f = 1.0f / 3.0f ;
printf("Float is %.50f \n ", f) ;
return 0 ;
}
```

Avec GCC 4.1 sur linux 64-bit, on obtient :

```
Float is 0.33333334326744079589843750000000000000000000000000
```

Avec Microsoft Visual Studio 2012 sur Windows 64-bit, on obtient : :

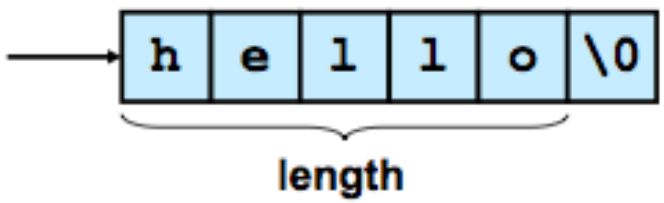
```
Float is 0.33333334326744080000000000000000000000000000000000
```

NB : il est important de bien gérer ces genres d'arrondies lors de l'écriture du code, pour assurer une meilleure portabilité de votre programme.

Chaînes de caractères

• En C, il n'existe pas de variable de type chaîne. Par contre il existe une convention de représentation des chaînes qui consiste à placer un caractère de code nul (\0) à la fin d'une succession d'octets représentant chacun des caractères de la chaîne.

• Ainsi, une chaîne de n caractères occupe en mémoire un emplacement de $n+1$ octets. En mémoire, la chaîne "hello" est représentée ainsi :



• Un pointeur sur une chaîne de caractères, pointe sur le caractère initial de la chaîne.

• Comment créer une chaîne ?

```
char ch [18]; /* permet de ranger une chaîne d'au plus 17 caractères */
char * ch = malloc (18); /* à l'adresse ch, on pourra ranger une
                            chaîne d'au plus 17 caractères */
```

Les fonctions de manipulation de chaînes :

Le langage C dispose de nombreuses fonctions standard de manipulation de chaînes qu'on peut classer en cinq catégories :

- **Copie de chaînes :**
  - `strcpy` : effectue une recopie complète d'une chaîne.
  - `strncpy` : permet de gérer le nombre de caractères à recopier.
- **Concaténation de chaînes :**
  - `strcat` : concatène deux chaînes (la 2ème est mise à la fin de la 1ère).
  - `strncat` : concatène deux chaînes en gérant le nombre de caractères.
- **Comparaison de chaînes :**
  - `strcmp` : comparaison lexicographique usuelle de deux chaînes.
  - `strncmp` : limite la comparaison au nombre de caractères indiqués.
- **Recherche de caractères ou de sous-chaînes dans une chaîne :**
  - `strchr` : cherche la 1ère occurrence d'un caractère dans une chaîne.
  - `strrchr` : cherche la 1ère occurrence d'un caractère dans une chaîne à partir de la fin.
  - `strstr` : cherche la 1ère occurrence d'une chaîne dans une autre.
  - ...
- **Conversions de chaînes en nombres :**
  - `strtol/strtod/strtoul` : conversion en `long/double/unsigned`

Limitation de la longueur des chaînes lues sur l'entrée standard :

Problème : il existe toujours un risque de voir l'utilisateur fournir une chaîne plus longue que l'emplacement prévu.

Solution (`fgets`) : on peut limiter la longueur des chaînes lues, avec élimination des caractères excédentaires, en procédant comme suit :

```
#include <stdio.h>
#include <string.h>
#define LG_MAX 5

main(){
char ch[LG_MAX];
printf("Donner une chaîne : ");
fgets (ch, LG_MAX, stdin);
printf("Chaine lue : %s\n", ch);
}
```

```
Donner une chaîne : ESIEA
Chaine lue : ESIE
Donner une chaîne : 123456789
Chaine lue : 1234
```

Solution (scanf) : on peut également imposer à %s, un gabarit maximal, c'est-à-dire un nombre maximal de caractères qui seront lus pas scanf, indépendamment des espaces blancs :

```
#include <stdio.h>
#include <string.h>
#define LG_MAX 5

main(){
char ch[LG_MAX];
printf("Donner une chaîne : ");
scanf ("%4s", ch);
printf("Chaine lue : %s\n", ch);
}
```

```
Donner une chaîne : ESIEA
Chaine lue : ESIE
Donner une chaîne : ___ESIEA
Chaine lue : ESIE
Donner une chaîne : ES_IEA
Chaine lue : ES
```

Compilation séparée

Compilation séparée

- Lorsque l'on a un gros code, il faut diviser son programme en plusieurs fichiers, pour plus de lisibilité.
- Méthode :
  - On définit les prototypes dans un fichier `".h"`
  - On inclut les définitions là où c'est nécessaire
    #include "fichier1.h"
    #include"fichier2.h"
  - On compile les fichiers `.c`
    `gcc -c fichier1.c`
    `gcc -c fichier2.c`
  - On lie le tout pour générer l'exécutable final
    `gcc -o EXEC fichier1.o fichier2.o`

NB : on peut regrouper toutes ces commandes dans un fichier `Makefile`.

Un exemple (simple) de fichier `Makefile` :

```
all : PROJET EXECUTION

code.o : code.c code.h
	gcc -c -Wall code.c
main.o : main.c code.h
	gcc -c -Wall main.c
PROJET : code.o main.o
	gcc -o PROJET code.o main.o

EXECUTION :
	./PROJET
```

Remarque : la commande `make`, permet d'exécuter par défaut la première règle qui est ici `all`. Dans ce cas, le programme sera compilé (via la règle PROJET) et l'exécutable sera également lancé (via la règle EXECUTION).

Bibliographie

• C. DELANNOY, Langage C, éditions EYROLLES, 4ème tirage 2005.

• D. Defour, "Programmation en C", Univ. de Perpignan Via Domitia.

• M. François, "Sécurité des Applications (Bonnes pratiques du langage C)", STI 5A-EO, INSA Centre Val de Loire, Oct. 2013.

• J. F. Lalande, "Programmation C", INSA Centre val de Loire, 13 Nov. 2012.

• B. W. Kernighan, D. M. Ritchie, "Le langage C (Norme ANSI)", DUNOD, 2ème édition, 2004.