

Programmation en C et Méthodes Numériques

Fichiers et Bibliothèques

T. Dietenbeck (thomas.dietenbeck@upmc.fr)

Sorbonne Université



- 1 Programmation modulaire
- 2 Fichiers
- 3 Exercice

1 Programmation modulaire

- Prototype
- Création de librairie
- Inclusion d'une librairie
- Compilation séparée

2 Fichiers

3 Exercice

Problèmes

Jusqu'à présent, on avait un seul fichier .c qui contenait la fonction `main` et toutes les autres fonctions (acceptable pour un petit projet)

- **Lisibilité** : fichier de plusieurs milliers (millions ?) de lignes
⇒ comment retrouver rapidement la méthode recherchée ?
- **Réutilisabilité** : lors d'un nouveau projet, certaines fonctions peuvent resservir
⇒ comment s'en servir sans faire un copier-coller du code ?
- **Modification concurrente** : plusieurs développeurs travaillent généralement sur un même projet
⇒ comment permettre à chacun de faire ses modifications ?



Problèmes

Jusqu'à présent, on avait un seul fichier `.c` qui contenait la fonction `main` et toutes les autres fonctions (acceptable pour un petit projet)

- **Lisibilité** : fichier de plusieurs milliers (millions ?) de lignes
⇒ comment retrouver rapidement la méthode recherchée ?
- **Réutilisabilité** : lors d'un nouveau projet, certaines fonctions peuvent resservir
⇒ comment s'en servir sans faire un copier-coller du code ?
- **Modification concurrente** : plusieurs développeurs travaillent généralement sur un même projet
⇒ comment permettre à chacun de faire ses modifications ?

Solution : Programmation modulaire

- Séparation du code en plusieurs fichiers (appelées bibliothèques)
- Une bibliothèque regroupe des méthodes ayant un point commun (e.g. fonctions mathématiques, gestion d'une liste d'entiers, ...)
- Le fichier principal inclut ces bibliothèques pour utiliser les méthodes

Réversivité croisée

On souhaite implémenter les fonctions suivantes :

$$f(x) = \begin{cases} 1, & \text{si } x \leq 1 \\ g(x+2), & \text{si } x > 1 \end{cases} \quad \text{et} \quad g(x) = f(x-3) + 4$$

- Qui déclarer en premier : f ou g ?

Récurtivité croisée

On souhaite implémenter les fonctions suivantes :

$$f(x) = \begin{cases} 1, & \text{si } x \leq 1 \\ g(x+2), & \text{si } x > 1 \end{cases} \quad \text{et} \quad g(x) = f(x-3) + 4$$

- Qui déclarer en premier : f ou g ?

Solution : Prototype

Idée :

- Dire au programme qu'une fonction existe et comment l'appeler
- **mais ne pas dire comment elle fonctionne**
⇒ donner le mode d'emploi de la fonction



Principe

Pour pouvoir appeler une fonction (vision client), il suffit de connaître :

- son nom
- ses paramètres d'entrée : les variables dont elle a besoin pour fonctionner
- son type de retour : le résultat que la fonction produit

Il n'est pas nécessaire de savoir comment le résultat est produit à partir des données d'entrée !

Syntaxe

```
typeRetour nomFonction( listeParametres );
```

- **Déclaration usuelle** d'une fonction mais
 - Se termine toujours par un ";"
 - Pas de code associé
- Un prototype se déclare au début d'un fichier (après les `#define`) pour que les autres fonctions le connaissent

Remarques

L'ordre dans lequel

- les prototypes sont déclarés
- les fonctions (dont le prototype a été donné) sont implémentées

n'a pas d'importance

Exemple

On souhaite implémenter les fonctions :

$$f(x) = \begin{cases} 1, & \text{si } x \leq 1 \\ g(x+2), & \text{si } x > 1 \end{cases} \quad \text{et} \quad g(x) = f(x-3) + 4$$

Code

```
// Inclusion des bibliotheques standards
#include <stdio.h>
#include <stdlib.h>

// Definition de constantes
#define X 5

// Declaration des prototypes
int f( int x );
int g( int x );
```

Exemple

On souhaite implémenter les fonctions :

$$f(x) = \begin{cases} 1, & \text{si } x \leq 1 \\ g(x+2), & \text{si } x > 1 \end{cases} \quad \text{et} \quad g(x) = f(x-3) + 4$$

Code

```
// Programme principal
int main( ) {
    printf( "f(%d) = %d // g(%d) = %d\n", X, f(X), X, g(X) );
    return 0;
}

// Implementation de f et g
int g( int x ) {
    return f(x-3) + 4;
}

int f( int x ) {
    if( x <= 1 ) return 1;
    else return g(x+2);
}
```

Programmation modulaire

- Séparation du code en plusieurs fichiers (appelées librairies)
- Une librairie regroupe des méthodes ayant un point commun (e.g. fonctions mathématiques, gestion d'une liste d'entiers, ...)
- Le fichier principal inclut ces librairies pour utiliser les méthodes

Définition

Une librairie se compose de 2 fichiers :

- un fichier "header" qui contient le prototype de toutes les méthodes
- un fichier "source" qui contient le code des méthodes

Composition

Un header contient

- la déclaration de la librairie

```
#ifndef nomLibrairie_h_  
#define nomLibrairie_h_
```

- l'inclusion d'autres librairies dont elle a besoin (`#include`)
- les définitions de constantes (`#define`) et de type (`#typedef`)
- les prototypes des méthodes
- la déclaration de fin de librairie (`#endif`)

Un header a pour extension de fichier ".h"

Exemple : Fonctions trigonométriques

```
// Declaration de la librairie (obligatoire!)
#ifndef trigonometrie_h_
#define trigonometrie_h_

// Inclusion de bibliotheques
#include <stdio.h>
#include <stdlib.h>

// Definition de constantes
#define PI 3.1415926
#define EPS 1e-5

// Prototypes
double sin( double x );
double cos( double x );
double tan( double x );

// Fin de la declaration (obligatoire)
#endif
```

Composition

Un fichier source contient

- l'inclusion du header (`#include "nomLibrairie.h"`)
 - le nom du fichier header est donné entre guillemets "
 - il s'agit toujours de la 1ère ligne du fichier source
- l'implémentation (*i.e.* le code) de toutes les fonctions déclarées dans le header

Un fichier source a pour extension de fichier ".c"

Exemple : Fonctions trigonométriques

```
// Inclusion du header (obligatoire)
#include "trigonometrie.h"

// Implementation
double tan( double x ) { return sin(x) / cos(x); }

double sin( double x ) {
    // Initialisation des variables
    int n = 0;
    double sX = x, uN = x, x2 = -x*x;
    // Calcul de sin
    do{
        uN = uN * x2 / ( (2*n+2)*(2*n+3) );
        sX += uN;
        n++;
    } while( (uN > EPS) || (uN < -EPS) );
    return sX;
}

double cos( double x ) { ... }
```


Principe

- Pour que le programme principal puisse utiliser des fonctions d'une librairie, il faut inclure celle-ci au début du programme.
- Ajout en début de fichier d'une commande `#include` suivi du nom de la librairie
 - soit entre chevrons `< >`, s'il s'agit d'une librairie standard (e.g. `<stdio.h>`, `<stdlib.h>`)
 - soit entre guillemets `" "`, s'il s'agit d'une librairie personnelle (e.g. `"trigonometrie.h"`)

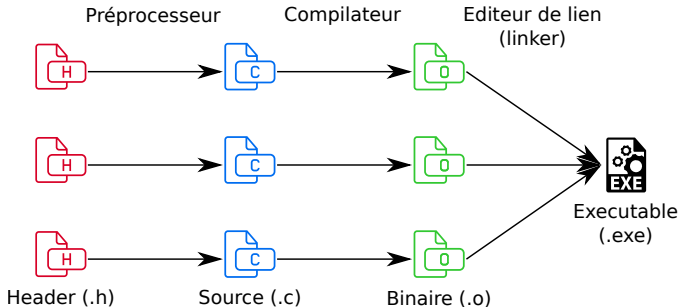
Exemple : Fonctions trigonométriques

```
// Inclusion de bibliotheques
#include <stdio.h>    // Facultatif (include par trigonometrie)
#include <stdlib.h>    // Facultatif (include par trigonometrie)
#include "trigonometrie.h" // Obligatoire (pour connaitre les
    ↪ fonctions sin, cos et tan)

int main( ) {
    printf( "sin(%f) = %f\n", PI/4, sin(PI/4) );
    printf( "cos(%f) = %f\n", PI/4, cos(PI/4) );
    printf( "tan(%f) = %f\n", PI/4, tan(PI/4) );
    return 0;
}
```

Principe

- 1 Le préprocesseur ajoute les headers aux fichiers source et détermine toutes les libraires dont le programme a besoin
- 2 Le compilateur traduit chaque fichier .c en langage machine (fichiers dit objets ou binaires .o)
- 3 Le linker lie tous les fichiers .o ensemble pour créer le fichier exécutable (.exe)



1 Programmation modulaire

2 Fichiers

- Définitions
- Fichiers textes
- Fichiers binaires
- Exemple

3 Exercice

Gestion des entrées / sorties

- Interactions avec l'utilisateur via la console :
 - `printf(...)` \Rightarrow affichage d'un message / résultat
 - `scanf(...)` \Rightarrow lecture d'une valeur donnée par l'utilisateur

Exemple

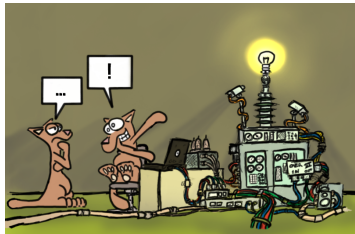
```
int unEntier;  
  
printf( "Veuillez entrer un entier" );  
scanf( "%d", &unEntier );  
printf( "Vous avez entre %d", unEntier );
```

Gestion des entrées / sorties

- Interactions avec l'utilisateur via la console :
 - `printf(...)` \Rightarrow affichage d'un message / résultat
 - `scanf(...)` \Rightarrow lecture d'une valeur donnée par l'utilisateur

Limitations

- Sauvegarde impossible des résultats
- Gestion de masse de données peu pratique
 - Affichage d'un tableau $n \times m$?
 - À chaque utilisation du programme, l'utilisateur doit donner les valeurs de travail \Rightarrow long et fastidieux (e.g. initialisation d'un tableau $n \times m$)



Intérêt

- Autre interface de gestion des entrées / sorties
- Permet la sauvegarde de données pour des utilisations futures
- Lecture / écriture rapide
- Meilleure lisibilité (e.g. fichier texte, csv pour excel, ...)

2 types de fichiers

- ① Fichier texte
- ② Fichier binaire

Syntaxe

- **Nouveau type** : `FILE`
- **Déclaration** : `FILE *fichier`
⇒ `fichier` contient **l'adresse** du début du fichier

Remarques

- Le type `FILE` ainsi que toutes les fonctions de manipulation de fichiers sont définies dans la bibliothèque `stdio.h`.
- Il faut donc penser à l'inclure au début du programme !

Syntaxe

- **Nouveau type** : FILE
- **Déclaration** : FILE *fichier
⇒ fichier contient **l'adresse** du début du fichier

Utilisation

Utilisation identique quelque soit le type de fichier

- 1 Déclaration d'une variable de type FILE*
- 2 **Ouverture** du fichier
- 3 Lecture / Écriture
- 4 **Fermeture** du fichier

Syntaxe

FILE *fopen(**char** *nom, **char** *mode) où

- nom : chaîne de caractères contenant le nom du fichier
- mode : mode d'ouverture du fichier
 - "r" : lecture seule (read)
 - "w" : écriture (write)
 - "a" : écriture à la fin (append)

Nom de fichier

- Si on ne donne que le nom du fichier (e.g. "charlie.txt"), le fichier doit être présent dans le même répertoire que le programme C (i.e. le ".exe")
- Si le fichier est ailleurs, il faut donner son chemin d'accès
 - soit absolu (e.g. "C :/data/charlie.txt" sous Windows ou "~ /data/charlie.txt" sous Linux)
 - soit relatif (i.e. par rapport au dossier où se trouve le fichier ".exe" :
" ../.. /data/charlie.txt")

Syntaxe

`FILE *fopen(char *nom, char *mode)` où

- `nom` : chaîne de caractères contenant le nom du fichier
- `mode` : mode d'ouverture du fichier
 - `"r"` : lecture seule (read)
 - `"w"` : écriture (write)
 - `"a"` : écriture à la fin (append)

Remarques

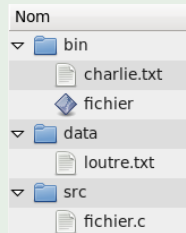
- Pour les modes `"r"` et `"a"`, le fichier doit exister
- Pour le mode `"w"`, si le fichier existe déjà, son contenu sera écrasé. Sinon, il sera créé.
- Si l'ouverture du fichier échoue (e.g. fichier inexistant ou verrouillé), `fopen` renvoie `NULL`.

Exemple

```
// Declaration d'une variable de type fichier
FILE *fichier1;
/* Ouverture en lecture seule du fichier texte
 * "charlie.txt" present dans le meme repertoire
 * que le fichier ".exe" */
fichier1 = fopen( "charlie.txt", "r" );

/* On verifie que l'ouverture s'est bien passee
 * avant d'utiliser le fichier */
if( fichier1 != NULL ) {
    // Utilisation
} else { // Sinon on affiche un message
    printf( "Erreur d'ouverture du fichier!" );
}

/* Declaration d'une variable de type fichier et
 * ouverture en ecriture d'un fichier "loutre.txt"
 * dans le repertoire "../data" (chemin relatif) */
FILE *fichier2 = fopen( "../data/loutre.txt", "w" );
```



Syntaxe

```
int fclose( FILE* fichier )
```

- renvoie 0 si le fichier a été correctement fermé

Remarques

Il faut toujours fermer un fichier après l'avoir utilisé

- Pour libérer la mémoire utilisée
- Pour permettre la réouverture du fichier (sinon il reste verrouillé)

Syntaxe

```
int fclose( FILE* fichier )
```

- renvoie 0 si le fichier a été correctement fermé

Exemple

```
// Declaration et ouverture d'un fichier
FILE *fichier = fopen( "../data/loutre.txt", "w" );
if( fichier != NULL ) { // L'ouverture s'est bien passée?
    /* Utilisation du fichier
     * ...
     */

    // Fermeture du fichier
    fclose( fichier );
}
```

Définition

- Extension : ".txt", ".csv" (comma separated value, pour les tableurs par exemple)
- Stocke les données sous forme de caractères
- + Peut être ouvert avec un éditeur de texte \Rightarrow bonne lisibilité
 - Peu compact
 - Lecture / écriture des données les unes après les autres

Syntaxe

`int` `fprintf`(`FILE` *`fichier`, `char` *`str`, ...) où

- `str` : la chaîne de caractères à écrire
- ... : variables dont on souhaite écrire le contenu (optionnel)
⇒ Syntaxe proche de la fonction `printf`

Exemple

```
int varI = 42;
float varF = 13.37;

fprintf( f, "Vive les loutres!\n" ); // Ecrit la chaine de
    ↪ caracteres "Vive les loutres!" dans f puis va a la ligne

fprintf( f, "%d\n", 42 );           // Ecrit l'entier 42 dans f
fprintf( f, "%f\n", varF );         // Ecrit dans f le reel varF

// Ecrit dans f un entier (varI) et un reel (varF)
fprintf( f, "%d et %f\n", varI, varF );
```


Syntaxe

`int fscanf(FILE *fichier, char *str, ...)` où

- `str` : chaîne de caractères contenant les types à lire
- `...` : adresses des variables où l'on veut stocker le contenu
- `fscanf` renvoie le nombre de données lues
⇒ Syntaxe proche de la fonction `scanf`
- Si on arrive à la fin du fichier, le nombre de données lues peut être différents du nombre de données demandées

Exemple

```
int varI, nbData;
float varF;

// Lit un entier dans f et stocke le resultat dans varI
nbData = fscanf( f, "%d", &varI ); // nbData = 1

// Lit un reel (float) et stocke la valeur dans varF
nbData = fscanf( f, "%d %f", &varI, &varF ); // nbData = 2
```

Rappel

| Code | Format |
|--------------------|---|
| <code>"%d"</code> | Entier (<code>int</code>) |
| <code>"%f"</code> | Réel (<code>float</code>) |
| <code>"%lf"</code> | Réel (<code>double</code>) |
| <code>"%e"</code> | Écriture scientifique (<code>float</code>) |
| <code>"%le"</code> | Écriture scientifique (<code>double</code>) |
| <code>"%c"</code> | Caractère (<code>char</code>) |
| <code>"%s"</code> | Chaîne de caractères (<code>char*</code>) |

Remarque

- Une chaîne de caractères lue avec `fscanf` s'arrête au premier espace trouvé (" ", "\n") ou à la fin du fichier

Définition

- Extension : ".bin"
- Sauvegarde les données telles qu'elles sont présentes en mémoire
- + Compact
- + Lecture / écriture de blocs de données
- Ne peut pas être ouvert avec un éditeur de texte

Ouverture de fichiers binaires

- Lors de l'appel à la fonction `fopen`, il faut rajouter un "b" au mode pour préciser qu'il s'agit d'un fichier binaire
- **Exemple** : Ouverture en lecture d'un fichier binaire

```
FILE *fBinaire = fopen( "charlie.bin", "rb");
```

Syntaxe

`int fwrite(void *src, int sizeType, int nbDonnees, FILE *fichier)` où

- `src` : pointeur vers les données à écrire
- `sizeType` : taille (en octet) du type de données à écrire
- `nbDonnees` : nombre de données à écrire
- `fichier` : fichier où écrire les données
- `fwrite` renvoie le nombre de données effectivement écrites.

Remarque

- Pour connaître la taille d'un type, on peut utiliser l'opérateur `sizeof`

Exemple : `sizeof(int)` renvoie 4

- `fwrite` permet d'écrire un bloc de données

Attention : toutes les données doivent être de même type

Syntaxe

```
int fwrite( void *src, int sizeType, int nbDonnees, FILE *fichier )
```

Exemple

```
#define DIM 42
...

int varI = 42;
float tabF[DIM];
/* Du code pour remplir tabF */
...

// Ouverture en ecriture du fichier binaire charlie.bin
FILE *fBin = fopen( "charlie.bin", "wb" );

// Ecriture d'1 entier (varI) dans fBin
fwrite( &varI, sizeof(int), 1, fBin );
// Ecriture de DIM reels (tabF) dans fBin
fwrite( tabF, sizeof(float), DIM, fBin );
```

Syntaxe

`int fread(void *dest, int sizeType, int nbDonnees, FILE *fichier)` où

- `dest` : pointeur vers les données à écrire
 - `sizeType` : taille (en octet) du type de données à écrire
 - `nbDonnees` : nombre de données à écrire
 - `fichier` : fichier où lire les données
 - `fread` renvoie le nombre de données effectivement lues
- ⇒ Syntaxe identique à `fwrite`

Remarque

- Le nombre de données lues peut être différent de `nbDonnees` si on a atteint la fin du fichier
 - `fread` permet de lire un bloc de données
- Attention** : toutes les données doivent être de même type

Syntaxe

```
int fread( void *dest, int sizeType, int nbDonnees, FILE *fichier )
```

Exemple

```
int varI;  
float tabF[DIM];  
  
// Ouverture en lecture du fichier binaire charlie.bin  
FILE *fBin = fopen( "charlie.bin", "rb" );  
  
// Lecture d'1 entier de fBin dans varI  
fread( &varI, sizeof(int), 1, fBin );  
// Lecture de DIM reels de fBin dans tabF  
fread( tabF, sizeof(float), DIM, fBin );
```

| | | Fichier texte | Fichier binaire |
|-----------|--------------------|---|-----------------|
| Taille | | peu compact | compact |
| Lisible | éditeur de texte | oui | non |
| | code C | oui | oui |
| Lecture / | Écriture par blocs | non | oui |
| Fonctions | Ouverture | fopen(nom, mode), avec mode = "r", "w" ou "a" "rb", "wb" ou "ab" | |
| | Fermeture | fclose(...) | |
| | Lecture | fscanf(...) | fread(...) |
| | Écriture | fprintf(...) | fwrite(...) |

Remarque

L'écriture du nombre 1.414213562373095 occupe :

- 8 octets (taille d'un **double**) dans un fichier binaire
- 17 octets (1 octet par caractère) dans un fichier texte

Énoncé

- On souhaite écrire dans un fichier texte et dans un fichier binaire le contenu d'un tableau de réels.
- La taille du tableau est définie comme une constante au début du programme.
- Après écriture dans les fichiers, on relira les données et les affichera.

En-tête

```
#include <stdio.h>    // Gestion des fichiers
#include <stdlib.h>
#include <time.h>

// Emplacement + Nom des fichiers
#define FNAME_TXT "exemple.txt"
#define FNAME_BIN "exemple.bin"
// Dimension du tableau
#define DIM 10

...
```

Déclaration des variables et initialisation

```
...  
int main( ) {  
    int i;  
    /// Initialisation  
    srand(time(NULL)); // Generateur de nombres aleatoires  
    // Initialisation des tableaux  
    float tab[DIM];  
    for( i = 0; i < DIM; i++ )  
        tab[i] = (float)rand()/(float)RAND_MAX;  
    float tabTxt[DIM], tabBin[DIM];  
    ...  
}
```

Écriture des fichiers

```
...  
// Fichier texte  
FILE *fTxt = fopen( FNAME_TXT, "w" ); // Ouverture en ecriture  
if( fTxt != NULL ) {  
    fprintf( fTxt, "%d ", DIM ); // Dimension du tableau  
    for( i = 0; i < DIM; i++ ) // Contenu du tableau  
        fprintf( fTxt, "%f ", tab[i] );  
    fclose( fTxt ); // Fermeture du fichier  
}  
  
// Fichier binaire  
FILE *fBin = fopen( FNAME_BIN, "wb" ); // Ouverture en ecriture  
if( fBin != NULL ) {  
    fwrite( &i, sizeof(int), 1, fBin ); // Dimension du tableau  
    fwrite( tab, sizeof(float), DIM, fBin ); // Contenu  
    fclose( fBin ); // Fermeture du fichier  
}  
...
```

Lecture du fichier texte

```
...
// Fichier texte
fTxt = fopen( FNAME_TXT, "r" ); // Ouverture en lecture
int dimTxt = 0;
if( fTxt != NULL ) {
    i = 0;
    fscanf( fTxt, "%d", &dimTxt );
    // Tant que l'on n'est pas arrive au bout du fichier
    while( fscanf( fTxt, "%f", &(tabTxt[i]) ) == 1 )
        i++;
    fclose( fTxt ); // Fermeture du fichier
}

// Fichier binaire
fBin = fopen( FNAME_BIN, "rb" ); // Ouverture en lecture
int dimBin = 0;
if( fBin != NULL ) {
    fread( &dimBin, sizeof(int), 1, fBin ); // Taille du tableau
    fread( tabBin, sizeof(float), dimBin, fBin ); // Contenu
    fclose( fBin ); // Fermeture du fichier
}
...
```

Affichage des tableaux

```
...
// Tableau initial
printf("Taille du tableau initial %d \n", DIM );
for( i = 0; i < DIM; i++ )
    printf( " | %11.10f ", tab[i] ); // 10 chiffres apres la ,
printf(" | \n" );



// Fichier texte
printf("Taille du tableau texte %d \n", dimTxt );
for( i = 0; i < dimTxt; i++ )
    printf( " | %11.10f ", tabTxt[i] );
printf(" | \n" );

// Fichier binaire
printf("Taille du tableau binaire %d \n", dimBin );
for( i = 0; i < dimBin; i++ )
    printf(" | %11.10f ", tabBin[i] );
printf(" | \n" );

return 0;
}
```

Comparaison des fichiers

```
bash-4.1$ ./fichier
Taille du tableau initial 10
| 0.0239615683 | 0.4439047575 | 0.3254126608 | 0.9166337252 | 0.7428848743 |
| 0.2844643593 | 0.8016647100 | 0.4131768048 | 0.9195679426 | 0.5095171928 |
Taille du tableau texte 10
| 0.0239620004 | 0.4439049959 | 0.3254129887 | 0.9166340232 | 0.7428849936 |
| 0.2844640017 | 0.8016650081 | 0.4131770134 | 0.9195680022 | 0.5095170140 |
Taille du tableau binaire 10
| 0.0239615683 | 0.4439047575 | 0.3254126608 | 0.9166337252 | 0.7428848743 |
| 0.2844643593 | 0.8016647100 | 0.4131768048 | 0.9195679426 | 0.5095171928 |
```

| Nom | Taille | Type |
|---|-----------|---------------------|
|  exemple.bin | 44 octets | bin document |
|  exemple.txt | 93 octets | document texte brut |

- Pour le fichier texte, seuls les 5 premiers chiffres après la virgule sont corrects !
- Le fichier texte est 2 fois plus volumineux

Solution : écriture de 10 chiffres après la virgule dans le fichier texte

```
Code : fprintf( fTxt, "%11.10f ", tab[i] );
```

```
bash-4.1$ ./fichier
Taille du tableau initial 10
| 0.0239615683 | 0.4439047575 | 0.3254126608 | 0.9166337252 | 0.7428848743 |
| 0.2844643593 | 0.8016647100 | 0.4131768048 | 0.9195679426 | 0.5095171928 |
Taille du tableau texte 10
| 0.0239615683 | 0.4439047575 | 0.3254126608 | 0.9166337252 | 0.7428848743 |
| 0.2844643593 | 0.8016647100 | 0.4131768048 | 0.9195679426 | 0.5095171928 |
Taille du tableau binaire 10
| 0.0239615683 | 0.4439047575 | 0.3254126608 | 0.9166337252 | 0.7428848743 |
| 0.2844643593 | 0.8016647100 | 0.4131768048 | 0.9195679426 | 0.5095171928 |
```

| Nom | Taille | Type |
|---|------------|---------------------|
|  exemple.bin | 44 octets | bin document |
|  exemple.txt | 133 octets | document texte brut |

- Le résultat est correct
- mais le fichier texte est maintenant 3 fois plus volumineux

Erreurs fréquentes

```
#define D 67
...

// Ouverture en binaire d'un fichier texte
FILE *fTxt = fopen( "erreur.txt", "rb" );

int i;
FILE *fTxt2 = fopen( "erreur.txt", "r" );
// Erreur a l'execution si erreur.txt n'existe pas
fscanf( fTxt2, "%d ", &i );

if( fTxt2 != NULL ) {
    fprintf( fTxt2, "%d ", D ); // Ouverture en lecture seule
    fscanf( fTxt2, "%d", i );   // i n'est pas un pointeur
    // Oubli de la fermeture du fichier => fichier verrouille
}

FILE *fBin = fopen( "erreur.bin", "wb" );
if( fBin != NULL ) {
    // D est une constante => pas d'adresse
    fwrite( &D, sizeof(int), 1, fBin );
    fwrite( &i, sizeof(double), 1, fBin ); // Mauvaise taille
}
```



- 1 Programmation modulaire
- 2 Fichiers
- 3 Exercice**

Mesure de tensions

- On souhaite calculer la valeur moyenne et la valeur efficace d'une tension dont l'évolution au cours du temps est stockée dans un fichier texte sous la forme :

```
0 0
0.1 2
0.3 2.5
0.8 1
...
```

- La première colonne du fichier correspond à l'instant t où la mesure est effectuée et la deuxième colonne à la valeur de la tension à cette instant.
- On supposera qu'une seule période est conservée dans le fichier.

Question

- Écrire le programme permettant de résoudre ce problème

Suivi des examens d'un patient

- Pour faciliter le suivi d'un patient, un médecin souhaite pouvoir consulter le résultats des examens précédents puis ajouter les nouvelles données recueillies.
- Les données à conserver sont : la date (chaîne de caractères), le poids, la taille et la tension (2 entiers) du patient.

Question

- Proposer un programme satisfaisant ce cahier des charges.