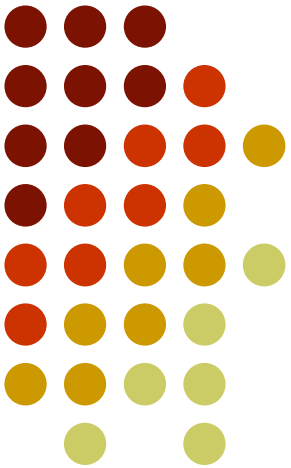
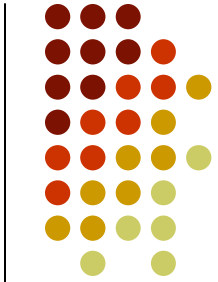
Programmation en C

Esisar - CS210

Entrées-Sorties simplifiées



Avant-propos



- Ce support de cours doit être travaillé en lien avec le poly « Introduction au langage C » de Cassagne, notamment ce qui se trouve dans le chapitre 5.

Introduction

- Le langage C “ pur ” se limite aux déclarations, expressions, instructions, structures de contrôle et fonctions.
- Les fonctions d’entrées-sorties ne font pas partie à proprement parler du langage. (Initialement elles pouvaient varier d’un système d’exploitation à l’autre.)
- La norme ANSI, reprise ensuite par la norme ISO (« ISO C89 ») décrit toutes les fonctions devant être définies dans un environnement standard.
- On regroupe sous le terme d’ « entrées-sorties » les opérations de lecture (read) et d’écriture (write) et les opérations de contrôle associées.

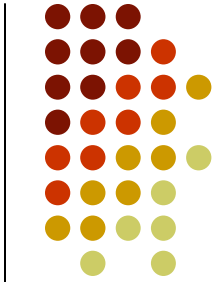
lecture/écriture

- Une opération d'écriture est un transfert de données effectué depuis une zone mémoire du programme vers une unité périphérique (de sortie) :
  - terminal
  - disque
  - imprimante
  - boîtier sur une carte, etc….
- Une opération de lecture est l'opération symétrique, c'est-à-dire un transfert de données depuis une unité périphérique (d 'entrée) vers une zone mémoire du programme :
  - clavier
  - disque
  - pile réseau, etc….

Implémentation des entrées/sorties

- Le langage C “ pur ” n’a pas d’instructions d’entrées-sorties.
- Celles-ci peuvent être réalisées soit :
  - par des appels systèmes de « bas niveau » gérant des descripteurs de fichiers.
  - soit par des fonctions « de haut niveau » d’ une bibliothèque standard, <stdio.h>, manipulant les « flux » ou « flots » (A privilégier!!) . Le terme de « haut niveau » indique qu’elles intègrent des mécanismes de formatage/déformatage et d’optimisation de transfert physique et qu’elles gèrent un certain nombre de problèmes et contrôles via le système de façon transparente à l’utilisateur.

flots, flux et fichiers (1)

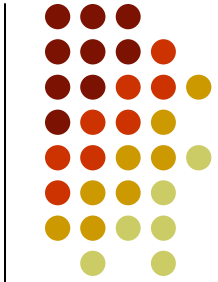


- Les entrées/sorties peuvent donc s'effectuer sur des unités périphériques aux comportements très différents.
- Afin d'unifier ces comportements, toutes les unités physiques sont vues à travers une abstraction unique appelée fichier (fichier ordinaire, répertoire ou spécial).
- De plus, afin d'unifier les mécanismes d'entrée/sorties, celles-ci s'effectuent au moyen d'un seul type d'unité logique appelé flot ou flux *(stream)*, qui rajoute à l 'abstraction précédente des tampons (buffers), des verrous, des rapports d 'états et d 'erreurs plus ou moins évolués.

flots, flux et fichiers (2)

- Le plus simple pour effectuer une entrée/sortie est de procéder caractère après caractère, mais on peut aussi procéder par blocs.
- On appelle “tampon” (*buffer*) une zone mémoire dans laquelle sont stockés provisoirement les caractères en attente (lors des entrées/sorties par blocs).
- Il existe trois modes de fonctionnements des flux :
  - Non mémorisés.
  - Mémorisés.
  - Mémorisés par ligne.

Flots : initialisation, fonctionnement



- A chaque flot est associé une paire d'indicateurs composé de l'indicateur de fin de fichier et l'indicateur d'erreur.
- Chaque flot possède également une position courante, généralement initialisée au début de fichier lors de l'ouverture, indiquant la position de l'octet à partir duquel sera effectuée la prochaine opération d'écriture/lecture, et qui sera mise à jour automatiquement.
- Un flot est codé par une structure de type « **`FILE`** », définie dans <stdio.h>. (cf cours suivant)
- Dans un programme un flot sera donc déclaré par un « **`FILE*`** ».

Ouverture / fermeture d 'un flot

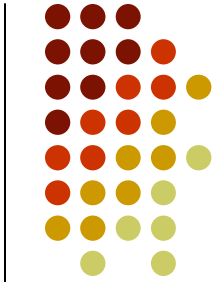
- Pour accéder à un fichier par l 'intermédiaire d 'un flot, il faut l ' « ouvrir » à l 'aide de la fonction « **fopen**», ce qui a pour effet l 'initialisation d 'un flot (cf poly 5.2)

prototype :

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

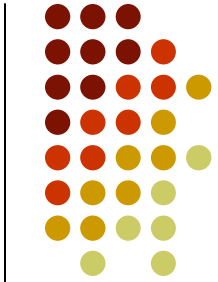
- le « chemin » est le nom du fichier à associer au flot.
- Ce flot peut être initialisé en mode :
  - « lecture » : utilise un fichier existant
  - « écriture » : crée un nouveau fichier ou réinitialise un fichier existant
  - « lecture/écriture » : mettre à jour (partiellement) un fichier existant

<stdio.h> : modes pour ouverture fichier



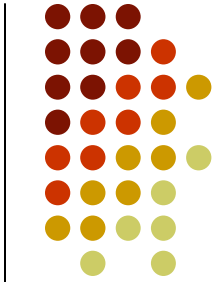
- « r » : lecture. Le fichier **doit** exister
- « w » : écriture seule, si le fichier existe, sa taille est ramenée à 0.
- « a » : écriture seule en fin de fichier. Si le fichier existe, il n 'est pas modifié, sinon il est créé
- « r+ » : lecture et écriture, mais les écritures commencent au début (écrasement s 'il y a déjà quelque chose.
- « w+ » : lecture et écriture, le fichier est ramené à une taille nulle s 'il existait.
- « a+ » : ajout et lecture : les lectures s 'effectuent au début, mais les écritures se feront en fin de fichier.

Flots standard d 'un processus



- Au lancement d'un processus, trois flots sont initialisés par défaut :
- « **stdin** » : flot « bufferisé » initialisé en lecture sur l'entrée standard (en général, le clavier).
- « **stdout** » : flot « bufferisé » initialisé en écriture sur la sortie standard (en général, l'écran).
- « **stderr** » : flot non « bufferisé » initialisé en sortie immédiate sur la sortie standard.
- Ces trois flots sont des variables globales du processus, définies dans le < stdio.h >.(cf plus tard pour notion variable globale)

exemple 1 : écriture sur stdout



```
int main(void){
   printf(″IUT Bonjour\n″);
   return EXIT_SUCCESS ;
   }
```

Est équivalent à :

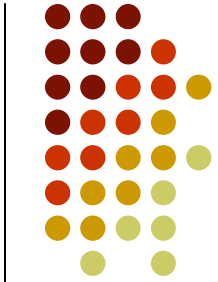
```
int main(void){
   fprintf(stdout,″IUT Bonjour\n″);
   return EXIT_SUCCESS ;
   }
```

Où stdout est une variable globale déclarée, définie dans <stdio.h> et initialisée par le système

exemple 2 : écriture dans un fichier, sans gestion d ’erreur

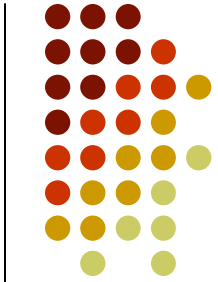
```
int main(void){
  FILE* fd ;   /*déclaration du flot */
        /* ouverture du flot : création du fichier */
  fd = fopen(″toto.txt″,″w″) ;
    /* utilisation du flot : écriture dans le fichier */
  fprintf(fd,″IUT Bonjour\n″);
                    /* fermeture du flot */
  fclose(fd);
  return EXIT_SUCCESS ;
  }
```

exemple fondamental : avec gestion d ’erreurs (1)



- Le programmeur ne peut pas espérer que les accès fichiers qu’il tente d ’exécuter vont toujours réussir. Lors de l ’exécution, peuvent survenir des problèmes variés : fichiers inexistants, droits d ’accès incorrects, disquettes enlevées/éjectées, plus de papier dans l ’imprimante…
- Il est donc indispensable de gérer les erreurs pouvant survenir, au minimum en récupérant les compte-rendus d ’erreurs, pour éviter au moins les dysfonctionnements grossiers, notamment pour les ouvertures de flots!! (cf cours « erreurs »)

Exemple fondamental : avec gestion d 'erreurs (2)



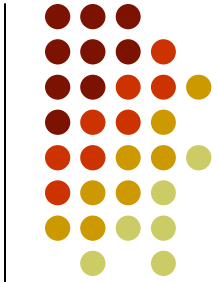
```
const char* fich = "toto.txt";
 FILE* fd = fopen(fich,"w") ;
 if ( NULL == fd ) {/* test impératif !! */
        fprintf(stderr,
              "erreur ouverture %s\n",fich);
        return EXIT_FAILURE ;
        } /* NE PAS LAISSER CONTINUER ! */
 fprintf(fd,"IUT Bonjour\n");
  if (EOF == fclose(fd)){/*moins impératif*/
        fprintf(stderr,
              "erreur fermeture %s\n",fich);
  return EXIT_FAILURE ;
        }
 …
```

Exemple fondamental : avec gestion d 'erreurs (3)

- Il existe une fonction « **perror** » de la bibliothèque <stdio.h> qui donne des messages d 'erreurs complémentaires sur « **stderr**», ainsi la remontée d 'erreur sur l 'ouverture de fichier peut s 'écrire :

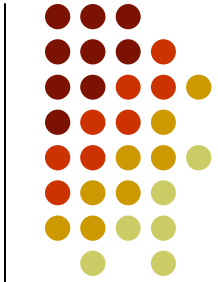
```
FILE* fd = fopen(fich,"w") ;
  if ( NULL == fd )
    {
    fprintf(stderr,"erreur fopen %s\n",fich);
    perror("");
    return EXIT_FAILURE ;
    } /* NE PAS LAISSER CONTINUER ! */
```

Retour sur la gestion d ’erreurs



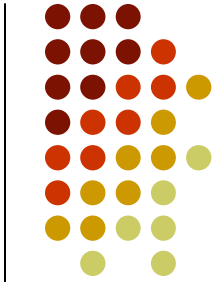
- L’exemple ci-après illustre les différentes stratégies possibles pour le programmeur concernant la mise en œuvre de gestion d’erreurs, notamment la stratégie « ne rien faire ».
- On souhaite écrire un simple code de recopie d’un fichier texte « `toto` » dans un autre fichier « `copie.maj` » :

Un exemple



```
int main(int argc, char* argv[]){
     FILE * fdin, fdout ;
     int c ;
     const char* fichierIn = "toto" ;
     const char* fichierOut = "copie.maj" ;

     /* ouverture du fichier à lire */
     fdin = fopen(fichierIn,"r");
     if ( NULL == fdin ){
          fprintf(stderr,"erreur ouverture
                         %s\n",fichierIn);
          perror("");
          return EXIT_FAILURE ;
     }
```

Un exemple

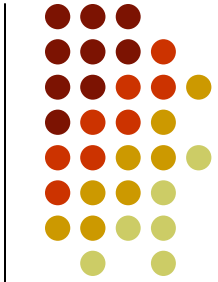


```
/* ouverture du fichier à écrire */
    fdout = fopen(fichierOut,"w");
    if ( NULL == fdout ){
          fprintf(stderr,"erreur ouverture
                        %s\n",fichierOut);
          perror("");
          (void)fclose(fdin);
/* ici pas de gestion d'erreur, le programmeur
commence à craquer ???? */
          return EXIT_FAILURE ;
          }

/* recopie */
   while ( EOF != (c = getc(fdin)) )
          putc(toupper(c),fdout);
```

Un exemple

```
    /* fermeture des fichiers */
    if (EOF == fclose(fdin)){
        fprintf(stderr,"erreur fermeture
                              %s", fichierIn);
        return EXIT_FAILURE ;
        }

    if (EOF == fclose(fdout)){
        fprintf(stderr,"erreur fermeture
                              %s",fichierOut);
        return EXIT_FAILURE ;
        }

    /* ouf , c 'est fini ! */
    return EXIT_SUCCESS ;
}
```

Un exemple : conclusion



- algo effectif simple, clair, 7 lignes:

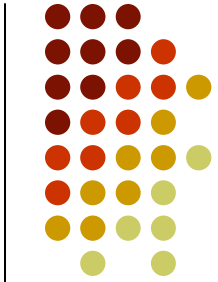
```
/* ouverture du fichier à lire */
/* ouverture du fichier à écrire */
/* tant que je peux lire un caractère */
      /* je le transforme */
      /* je l 'écris */
/* fermeture du fichier à lire */
/* fermeture du fichier à écrire */
```

- Codage : entre 20 et 30 lignes, le traitement est noyé dans la gestion d 'erreur (et encore il en manque!)

Un exemple : conclusion

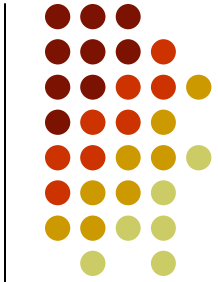
- Ici, la stratégie « ne pas tester les erreurs » est :
  - inapplicable pour les ouvertures de fichiers !!
  - acceptable pour fermeture fichier lu.
  - pas terrible pour fermeture fichier créé.
- Le code reste donc important au regard de l ’algo, on pourrait être tenté par la version :

Un exemple (V2)



```
static FILE * ouvreOuQuitte(const char *nom,
                            const char* mode)
static void fermeOuQuitte(const char * nom,
                          FILE* fd) ;

int main(int argc, char* argv[]){
    FILE * fdin, fdout ;
    int c ;
    const char* fichierIn = "toto" ;
    const char* fichierOut = "copie.maj" ;
```

Un exemple (V2)



```
/* ouverture du fichier à lire */
fdin = ouvreOuQuitte(fichierIn,"r");
/* ouverture du fichier à écrire */
fdout = ouvreOuQuitte(fichierOut,"w");
/* recopie */
while ( EOF != (c = getc(fdin)) )
      putc(toupper(c),fdout);
/* fermeture des fichiers */
fermeOuQuitte(fichierIn,fdin);
fermeOuQuitte(fichierOut,fdout);

return EXIT_SUCCESS ;
}
```

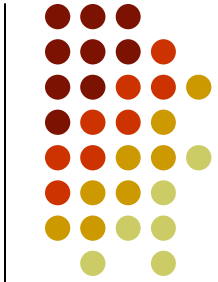
Un exemple (V2)

```
FILE * ouvreOuQuitte(const char *nom,
                     const char* mode)  {

    FILE * fd = fopen(nom,mode);

    if ( NULL == fd ){
          fprintf(stderr,"erreur ouverture
                                  %s\n",nom);
          perror("");
          fprintf(stderr,"impossible de
                          continuer !!\n");
          exit ( EXIT_FAILURE ) ;
          }

    return fd ;
    }
```

Un exemple (V2)

```
void fermeOuQuitte(const char * nom,
                                    FILE* fd) {

    if (EOF == fclose(fd)){
          fprintf(stderr,"erreur fermeture
                                          %s", nom);
          exit ( EXIT_FAILURE ) ;
          }
}
```

Un exemple : conclusion 2



- le code devient plus lisible, mais gestion d ’erreur moins fine : si la seconde ouverture se passe mal, on ne ferme plus le 1° fichier, idem si la 1° fermeture se passe mal.
- De plus si le code était plus complexe, il faudrait peut-être aussi fermer d ’autres fichiers, libérer des zones mémoires, etc, d ’où la stratégie parfois de centraliser ces traitements, mais cela signifie qu ’une fonction connaît toutes les informations nécessaires!

fonctions d ’entrées/sorties « formatées »

- Les fonctions de base sont pour l’écriture : *printf*, et pour la lecture : *scanf* (cf poly 5.5)

prototypes :

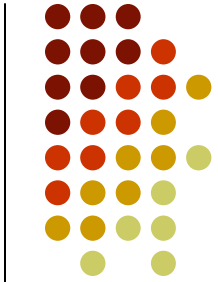
```
int printf(const char*, … “liste de paramètres” …);
int scanf(const char*, … “liste de paramètres” …) ;
```

printf retourne le nombre de caractères écrits ou un nombre négatif si erreur.

scanf retourne EOF sur erreur, ou le nombre de paramètres interprétés.

Ces deux fonctions rajoutent à la lecture/écriture des procédures de conversion (entre format interne et format externe)

utilisation de printf



- Voir poly ch5 (5.5).
- Les options minimales à connaître du paramètre de format sont :
  - %d , % i : entiers
  - %f (%g) : réels (double précision)
  - %s : chaînes
  - %c : caractères (isolés)
  - %p : adresse (pointeur)

exemple de scanf

- *Attention !* scanf utilise un passage de paramètre par référence (par adresse). Exemple :

```
int main(void) {
   float x ;
   int i, j ;
   if ( 3 != scanf("%f , %d %d", &x, &i, &j))
       printf("erreur de saisie\n");
   else
       printf("Saisi : %f , %d %d \n ",x,i,j);
   return EXIT_SUCCESS ;
   }
```

Fonctions dérivées : *fprintf, fscanf*

- A côté fonctions de base « printf, scanf » (dont en fait on ne se sert quasiment jamais), il existe de nombreuses fonctions dérivées, beaucoup plus utilisées : fprintf, fscanf , sprintf, snprintf, sscanf, etc...
- On a déjà vu que « printf, scanf » utilisent des flots implicites :

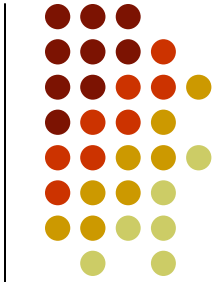
```
printf(" bonjour ! ") ;
```
est équivalent à
```
fprintf(stdout, " bonjour ! ") ;
```

- et de même :

```
scanf("%d",&i) ;
```
est équivalent à
```
fscanf(stdin, "%d", &i) ;
```

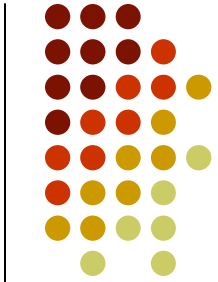
- Et on peut écrire dans (ou lire à partir d ') un flot quelconque.

Fonctions dérivées : *sprintf, sscanf*



- Les fonctions « sprintf , sscanf » ont le même comportement que « printf , scanf » mais elles s ’appliquent non pas sur des flots , mais sur des « tampons » (en fait sur de simples zones mémoires!).
- Elles permettent de gérer au niveau utilisateur les conversions de formatage.

Exemple de *sprintf, sscanf*



```
char buf[16] ;
  int i = 3 ;
  int j ;
  sprintf(buf,"i vaut %d\n",i) ;
  /*ou snprintf(buf,sizeof(buf),...); */
```

Va écrire dans le tampon « buf » la chaîne :

`"i vaut 3\n"`

i		v	a	u	t		3	\n	\0	x	x	x	x	x	x

Fonctions dérivées : *sprintf, sscanf*

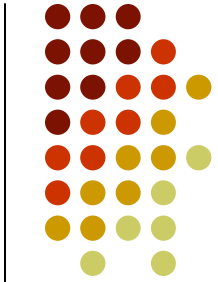
- De même si « buf » est un tampon contenant les caractères suivants :

-	3	.	5	2	\0	x	x	x	x	x	x	x	x	x	x

```
float x ;
sscanf(buf,"%f",&x) ;
```

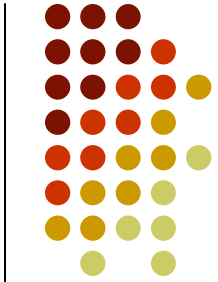
- permet de lire un réel à partir de sa représentation ASCII présente dans le buffer « buf ».

Lecture et écriture par lignes



- La fonction «scanf » étant mal spécifiée, et peu adaptée à un traitement correct des erreurs de saisie, il est conseillé pour les entrées (aussi bien provenant du clavier qu ’un fichier!) d ’utiliser des lectures « lignes » (grâce à la fonction “ fgets ”) et de gérer au niveau utilisateur l ’analyse de la ligne saisie (avec “ sscanf ”)
- fgets() permet de rapatrier dans un tampon du programme une ligne complète provenant du flot considéré (par exemple stdin!)

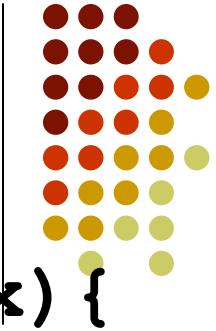
Lecture et écriture par lignes



- Ex : lecture d ’un fichier ligne à ligne, et écriture sur la sortie standard, en rajoutant le numéro de ligne :

```
char buf[1024] ;
int i = 0 ;
FILE * fdSource ;
…          /* initialisation de fdsource */
/* puis lecture ligne à ligne:
tant qu ’on est pas en fin de fichier,
fgets ne rend pas NULL!*/
while(NULL!=fgets(buf,sizeof(buf),fdSource)
    fprintf(stdout,”Ligne %d: %s\n”,i++,buf);
```

Exemple : lireEntier( ) (simplifié)

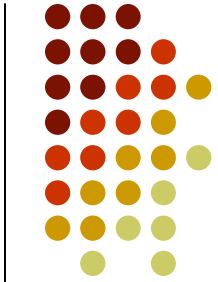


```
int lireEntierMess(int min,int max){
  int cr ;
  bool fini ;
  int n ;
  char buf[256] ;
  fini = false ;
  assert ( min <= max ) ;
  while ( !fini ){
     printf ( "Entrer un entier entre
          %d et %d\n",min, max ) ;
     assert( buf ==
```

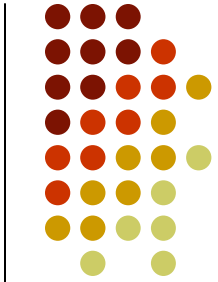
Exemple : lireEntier( ) (simplifié)

```
    if ( cr != 1 || n < min || n > max ){
          printf("Saisie incorrecte,
                      recommencez!\n") ;
          printf ( "Entrer un entier compris
                entre %d et %d\n",min, max ) ;
          }
    else
          fini = true ;
    }     /* fin tant que */
    assert ( min <= n && n <= max ) ;
    return n ;
}
```

entrées/sorties binaires



- Les fonctions précédemment décrites sont réservées à la lecture/écriture de données de type « texte » (de l ’ASCII !, visualisable sur un écran…)
- Il existe aussi deux fonctions permettant de lire (« fread ») et d’écrire (« fwrite ») des blocs de données sans aucun transcodage (de façon binaire!).
- Attention les fichiers résultants ne sont en général pas portables (la taille des objets « C » n ’est pas définie !)

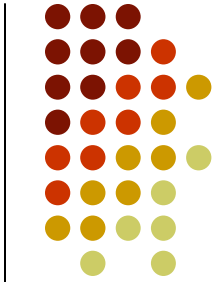


lecture : *fread*

- prototype :

```
size_t fread(void *tampon, size_t taille,
                    size_t nombre, FILE* flot) ;
```

- *tampon* est l'adresse du début de la mémoire où est placé le résultat de la lecture.
- Valeur rendue : nombre d'éléments lus ou EOF lors d'une erreur.

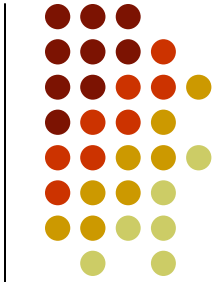


écriture : *fwrite*

- prototype :

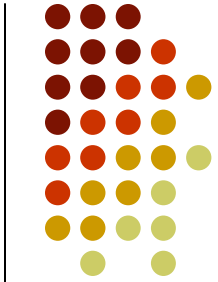
```
size_t fwrite(void *tampon, size_t taille,
                    size_t nombre, FILE* flot) ;
```

- Fonction symétrique de la précédente.
- Valeur rendue : nombre d'éléments écrits.



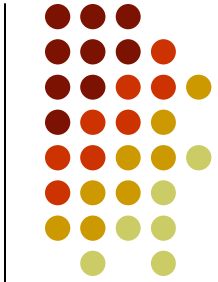
exemple écriture (simplifié)

```
  float notes[NBELEVES] ;
  int nb = NBELEVES ;
  int nbEcrits ;
  FILE* fd ;
  /* par ex. « notes » initialisé par saisie op */
  /* ouverture du flot (sans gestion d 'erreur (=bug!),
  puis écriture du tableau dans fichier */
  fd = fopen("notes.bin","w") ;
  nbEcrits = fwrite(notes,sizeof(float),nb,fd);
/* traitement erreurs écriture */
  if( nbEcrits != nb )
        fprintf(stderr,"Erreur écriture!\n) ;
/* fermeture du flot (sans gestion d 'erreur) */
  fclose(fd);
```



exemple écriture (simplifié) (2)

```
  float notes[NBELEVES] ;
  int nb = NBELEVES ;
  int nbEcrits ;
  FILE* fd ;
  /* par ex. « notes » initialisé par saisie op */
  /* ouverture du flot (sans gestion d 'erreur (=bug!),
  puis écriture du tableau dans fichier */
  fd = fopen("notes.bin","w") ;
  nbEcrits = fwrite(notes,sizeof(notes),1,fd);
/* traitement erreurs écriture */
  if( nbEcrits != 1 )
        fprintf(stderr,"Erreur écriture!\n) ;
/* fermeture du flot (sans gestion d 'erreur) */
  fclose(fd);
```



exemple lecture (simplifié)

```
  float notes[NBELEVES] ;
  int nb = NBELEVES ;
  int nblus ;
  FILE* fd ;
  /* ouverture du flot (sans gestion d 'erreur (=bug!),
  puis lecture du tableau à partir du fichier */
  fd = fopen("monFichier","r") ;
  nblus = fread(notes,sizeof(notes),1,fd);
/* traitement erreurs lecture */
  if( nblus != 1 )
        fprintf(stderr,"Erreur lecture!\n") ;
/* fermeture du flot (sans gestion d 'erreur) */
  fclose(fd);
/* puis, utilisation des données…  */
```

Contrôle de terminaison et d'erreur

- Rappel : à chaque flot est associé l'indicateur de fin de fichier et l'indicateur d'erreur.
- Les fonctions suivantes permettent de tester ou modifier les indicateurs:

« feof » rend « vrai » si on est arrivé en fin de fichier. Prototype : *int* feof(*FILE* flot*) ;

« ferror » rend « vrai » si une erreur s 'est produite sur le flot. Prototype : *int* ferror(*FILE* flot*) ;

« clearerr » réinitialise les deux indicateurs. Prototype : *void* clearerr(*FILE* flot*) ;

Contrôle du positionnement

- Les fonctions de lecture/écriture opèrent en général sur un mode séquentiel. Mais on peut aussi dans le cas d'un fichier ordinaire accéder à n'importe quel endroit du fichier en modifiant la position courante du flot (accès direct).
- Pour se positionner dans un flux :

  int fseek(*FILE* flot, long décalage, int origine*) ;
  int rewind(*FILE* flot*) ; (position au début)
- Pour savoir où on est dans un flux :

  int ftell(*FILE* flot*) ;
- (cf « man »!!)