11 Fichiers

Nous avons vu comment communiquer avec l'utilisateur, organiser nos instructions pour réaliser un programme. Cependant, à la fin de l'exécution de notre programme, nous reviendrions à zéro en le relançant. Il serait donc intéressant d'avoir un moyen de sauvegarder cet état. De même nous nous sommes limités à des entrées courtes et simples données par un utilisateur lors de l'exécution de son programme.

Pour sauvegarder et récupérer de l'information, il est possible de passer par la gestion des fichiers sur l'espace disque.

11.1 Ouverture et création de fichiers

Les fichiers vont se gérer dans le programme à l'aide du type `FILE *`. Celui-ci permettra de garder un pointeur sur un fichier ouvert par le programme. L'ouverture d'un fichier depuis l'espace disque se fait par la commande `fopen` de `stdio.h` :

```
fopen(/* chemin du fichier */, /* mode d'ouverture */)
```

- Le premier argument est le chemin du fichier sur l'espace disque.
- Le second argument est le mode d'ouverture du fichier, ceci dépend si l'on souhaite lire, écrire, s'y déplacer et autre :

Mode	Lecture	Écriture	Création	Effacement	Ajout en fin
`"r"`	√				
`"w"`		√	√	√	
`"a"`		√	√		√
`"r+"`	√	√			
`"w+"`	√	√	√	√	
`"a+"`	√	√	√		√

Une fois les opérations dans le fichier terminées, on arrête son utilisation par `fclose` sur celui-ci :

```
FILE * fichier = NULL;
/* Tentative d'ouverture / création d'un fichier */
if((fichier = fopen(/* chemin */, /* mode */)) == NULL) {
  /* Gestion de l'impossibilité d'ouverture */
}
/* Opérations avec le fichier */
/* ... */
/* Fermeture du fichier */
fclose(fichier);
fichier = NULL;
```

L'exemple suivant permet la création d'un fichier `MonFichier.txt` vide :

```
FILE * fichier = NULL;
if((fichier = fopen("MonFichier.txt", "w")) == NULL) {
  printf("Erreur de création de mon fichier.\n");
  exit(EXIT_FAILURE);
}
printf("Fichier créé avec succès.\n");
fclose(fichier);
fichier = NULL;
exit(EXIT_SUCCESS);
```

Pour alléger la lecture des codes suivants, la vérification de l'existence ou création du fichier peut être omise. Cependant, pensez à le gérer pour éviter des mauvaises surprises dans vos codes.

11.2 Lecture et écriture

11.2.1 Avec fonctions formatées

Pour communiquer avec l'utilisateur via le terminal nous avons vu des fonctions issues de `stdio` qui le permettent telles que `printf`, `scanf`. Ces fonctions existent aussi en version pour les fichiers.

fprintf

`fprintf` permet l'impression formatée de caractères dans un fichier donné. Il fonctionne de la même manière que `printf`, à la différence qu'il faut fournir un premier argument supplémentaire : le pointeur sur le fichier :

```
FILE * fichier = fopen("test.txt", "w");
fprintf(fichier, "Hello fichier !\n");
fclose(fichier);
```

De même `fprintf` peut être utilisé pour sauvegarder des données fournies par à un utilisateur :

```
int renseignerInfos(const char * pseudo) {
  int age;
  printf("Quel est votre âge ? ");
  scanf("%d", &age);
  FILE * infos = fopen(pseudo, "w");
  /* sauvegarde la valeur de age */
  fprintf(infos, "%d\n", age);
  fclose(infos);
}
```

fscanf

La fonction `fscanf` fonctionne de la même manière pour récupérer le contenu formaté d'un fichier :

```
int lireInfos(const char * pseudo, int * age) {
  FILE * infos = NULL;
  if((infos = fopen(pseudo, "r")) == NULL) {
    return 0; /* l'utilisateur est inconnu */
  }
  /* récupère la valeur de age */
  fscanf(infos, "%d", age);
  fclose(infos);
  return 1;
}
```

11.2.2 Par caractère

Dans des cas plus pratiques, nous pourrions vouloir récupérer ou écrire les informations caractère par caractère. Par exemple si vous souhaitez coder un message en conservant la mise en page. Pour ceci, nous avons à disposition les fonctions `fputc` pour écrire un caractère dans un fichier et `fgetc` pour lire un caractère depuis un fichier.

fgetc

`fgetc` va lire et renvoyer un par un les caractères d'un fichier donné. Lorsque `fgetc` est arrivé à la fin du fichier, la valeur renvoyée sera `EOF` (End Of File).

```
FILE * fichier = fopen("message.txt", "r");
int caractere;
/* tant qu'on lit un caractère dans le fichier */
while((caractere = fgetc(fichier)) != EOF) {
  putchar(caractere); /* on affiche le caractere */
}
fclose(fichier);
```

fputc

Il est tout aussi possible d'écrire dans un fichier caractère par caractère. Ceci peut se faire à l'aide de `fputc`.

```
FILE * input = fopen("message.txt", "r");
FILE * output = fopen("resultat.txt", "w");
int caractere;
const int cle = 5;
/* tant qu'on lit un caractère dans le fichier */
while((caractere = fgetc(input)) != EOF) {
  /* on le code s'il est alphabéthique */
  if(caractere >= 'a' && caractere <= 'z')
    caractere = (caractere - 'a' + cle) % 26 + 'a';
  else if(caractere >= 'A' && caractere <= 'Z')
```

```
    caractere = (caractere - 'A' + cle) % 26 + 'A';
  /* on l'écrit dans le fichier de sortie*/
  fputc(caractere, output);
}
fclose(input);
fclose(output);
```

11.2.3 Par bloc mémoire

Pour effectuer des sauvegardes, nous pourrions souhaiter de ne pas avoir nécessité que le fichier soit humainement lisible et ne prenne pas une place plus importante que celle des données réelles. Pour ceci, il est possible d'écrire dans un fichier en binaire directement.

Cette écriture en binaire se fera par octets. On lui passera un tableau ou un pointeur contenant ou recevant les données qui nous intéressent. On précisera que le fichier manipulé n'est pas un fichier texte en ajoutant un `"b"` après le mode d'ouverture `"r"`, `"w"` ou `"a"`. Les fonctions utilisées ensuite pour la lecture et l'écrire seront `fread` et `fwrite`.

fwrite

La fonction `fwrite` prend en paramètres :

- Pointeur sur les données à écrire.
- Taille d'un élément.
- Nombre d'éléments.
- Pointeur sur le fichier où écrire.

`fwrite` renvoie le nombre d'éléments écrits dans le fichier.

```
int sauvegarderListe(const char * filepath, const int * liste, int
↪  taille) {
  FILE * output = fopen(filepath, "wb");
  /* écriture de la taille : une variable */
  if(fwrite(&taille, sizeof(int), 1, output) != 1) {
    printf("Erreur écriture taille\n");
    return 0;
  }
```

```
  /* écriture de la liste : un tableau */
  if(fwrite(liste, sizeof(int), taille, output) != taille) {
    printf("Erreur écriture liste\n");
    return 0;
  }
  fclose(output);
  return 1;
}
```

fread

Une liste ainsi écrite précédemment peut être charger par le même type de procédé. `fread` permet la lecture de données et fonctionne similairement à `fwrite`. À noter que `fread` ne procède pas à l'allocation de la plage mémoire dans laquelle elle écrit. Ce sera votre rôle de vous assurer qu'elle existe et peut recevoir les données.

La fonction `fread` prend en paramètres :

- Pointeur sur les données à récupérer.
- Taille d'un élément.
- Nombre d'éléments.
- Pointeur sur le fichier où lire.

`fread` renvoie le nombre d'éléments lus depuis le fichier.

```
int chargerListe(const char * filepath, int ** liste, int *
↪  taille) {
  FILE * input = fopen(filepath, "rb");
  /* lecture de la taille : nécessaire à l'allocation */
  if(fread(taille, sizeof(int), 1, input) != 1) {
    printf("Erreur lecture taille\n");
    return 0;
  }
  /* allocation de la liste */
  if((*liste = (int *)malloc(sizeof(int) * *taille)) == NULL) {
    printf("Erreur allocation liste\n");
```

```
    return 0;
  }
  /* lecture des éléments de la liste */
  if(fread(*liste, sizeof(int), *taille, input) != *taille) {
    printf("Erreur lecture liste\n");
    return 0;
  }
  fclose(input);
  return 1;
}
```

11.3 Se déplacer dans un fichier

Des fonctions sont proposées pour jouer sur la position du curseur lors de l'écriture et de la lecture dans un fichier.

`ftell` prend pour argument un fichier et indique la position actuelle du curseur dans ce fichier.

`rewind` prend pour argument un fichier et rembobine le fichier au début.

`fseek` permet de placer le curseur à un endroit souhaité dans un fichier. `fseek` prend en arguments :

- Le fichier dans lequel déplacer le curseur.
- La position relative où déplacer le curseur par rapport à l'information donnée au paramètre suivant.
- Point de repère depuis lequel appliquer le décalage :
  - `SEEK_SET` : début du fichier.
  - `SEEK_CUR` : position actuelle dans le fichier.
  - `SEEK_END` : fin du fichier.

En exemple d'application de ces fonctions, nous proposons un code qui lit le texte pour compter le nombre de phrases présentes puis lit chaque phrase :

```
int carInChaine(char car, const char * chaine) {
  for(; *chaine != '\0'; ++chaine) {
```

```
    if(car == *chaine)
      return 1;
  }
  return 0;
}

int lirePhrase(FILE * file, long * start, long * end) {
  int car;
  while((car = fgetc(file)) != EOF) {
    if(! carInChaine(car, "\t\n ")) {
      break;
    }
  }
  if(start) /* on récupère la position du début de la phrase */
    *start = ftell(file) - 1;
  do {
    if(car == '.') {
      break;
    }
  } while((car = fgetc(file)) != EOF);
  if(end) /* on récupère la position de la fin de la phrase */
    *end = ftell(file);
  return car != EOF;
}

void afficherPortionFichier(FILE * file, long start, long end) {
  int car;
  /* on se replace dans le fichier à la position indiquée */
  fseek(file, start, SEEK_SET);
  while(ftell(file) != end) {
    putchar(fgetc(file));
  }
}
```

```c
int main() {
  FILE * file = fopen("message.txt", "r");
  int i;
  long start, end;
  for(i = 0; lirePhrase(file, NULL, NULL); ++i);
  printf("%d phrases.\n", i);
  /* on rembobine le fichier au début */
  rewind(file);
  for(i = 0; lirePhrase(file, &start, &end); ++i) {
    printf(" - Phrase %d (%ld caracteres): ", i + 1, end - start);
    afficherPortionFichier(file, start, end);
    printf("\n");
  }
  fclose(file);

  exit(EXIT_SUCCESS);
}
```

11.4 Flux standards d'entrées et sorties

En réalité, lorsque vous utilisez `printf` et `scanf`, vous travaillez également dans des `FILE *` standards qui correspondent aux entrées et sorties de votre terminal.

11.4.1 stdout

Le flux de sortie dans lequel on imprime nos caractères lors des appels à `printf` par exemple est `stdout`. On peut l'utiliser avec `fprintf` :

```c
printf("Par printf\n");
fprintf(stdout, "Par fprintf\n");
```

11.4.2 stdin

De la même manière il existe le flux de sortie standard `stdin` peut être utilisé avec `fscanf` :

```c
int nombreScanf, nombreFscanf;
scanf("%d", &nombreScanf);
fscanf(stdin, "%d", &nombreFscanf);
printf("%d %d\n", nombreScanf, nombreFscanf);
```

11.4.3 stderr

Le flux standard qui sera cependant le plus utile est `stderr`. C'est un flux standard de sortie dédié à l'écriture des erreurs ou des logs. À noter que `stdout` ne sera souvent réellement imprimé dans le terminal que lorsque son buffer est plein ou lorsqu'il imprime un retour à la ligne. `stderr` sera imprimé peu importe les caractères donnés.

```c
FILE * fichier = NULL;
if((fichier = fopen("fichier_a_ne_pas_creer", "r")) == NULL) {
  fprintf(stderr, "Erreur main() : \"fichier_a_ne_pas_creer\" est
  ↪  introuvable\n");
  exit(EXIT_FAILURE);
}
fclose(fichier);
```

À noter que cette sortie peut être redirigée vers un fichier de logs, laissant dans le terminal uniquement `stdout` :

```
# gcc -o prog main.c
# ./prog
Erreur main() : "fichier_a_ne_pas_creer" est introuvable
# ./prog 2>log
# cat log
Erreur main() : "fichier_a_ne_pas_creer" est introuvable
```

11.5 Résumé

Il est possible de gérer un fichier avec le type `FILE *`. Ce fichier s'ouvre avec `fopen` et se ferme avec `fclose` :

```
FILE * fichier = NULL;
if((fichier = fopen(/* chemin */, /* mode */)) == NULL) {
  /* traitement erreur */
}
/* utilisation fichier */
fclose(fichier);
```

Les principaux modes sont :

- `r` : lecture.
- `w` : écriture (création et effacement du fichier).
- `a` : écriture (création et ajout en fin du fichier).
- `r+` : écriture et lecture.
- `w+` : écriture et lecture (création et effacement du fichier).
- `a+` : écriture et lecture (création et ajout en fin du fichier).

Il est possible d'imprimer une chaîne de caractères formatée dans un fichier avec `fprintf` et lire une chaîne de caractères formatée avec `fscanf` :

```
fprintf(fichier, /* format */, /* variables */);
scanf(fichier, /* format */, /* adresses */);
```

Il est aussi possible de procéder caractère par caractère avec `fgetc` et `fputc` :

```
char car;
while((car = fgetc(fichier)) != EOF) {
  fputc(car, fichier);
}
```

De manière plus avancée, il est possible de passer en binaire en ajoutant `b` au mode. Ceci permet d'écrire nos données avec `fwrite` et lire avec `fread` :

```
if(fwrite(/* pointeur */, /* taille élément */, /* nombre
↪  éléments */, fichier) != /* nombre élements */)
  /* gestion erreur écriture */
if(fread(/* pointeur */, /* taille élément */, /* nombre éléments
↪  */, fichier) != /* nombre élements */)
  /* gestion erreur lecture */
```

Il est possible de jouer avec la position du curseur dans un fichier avec les fonctions suivantes :

```
ftell(fichier); /* donne la position du curseur */
rewind(fichier); /* met le curseur au début du fichier */
fseek(fichier, 1, SEEK_SET); /* se positionne après le premier
↪  octet du fichier */
fseek(fichier, 1, SEEK_CUR); /* déplace le curseur d'un caractère
↪  vers l'avant */
fseek(fichier, -1, SEEK_END); /* se positionne avant le dernier
↪  octet du fichier */
```

La bibliothèque `stdio` définie des entrées en sorties standards :

```
stdout /* sortie standard, utilisée par printf */
stdin /* entrée standard, utilisée par scanf */
stderr /* sortie d'erreur standard */
```

11.6 Entraînement

Exercice noté 42 (⋆⋆ Compteur de lancements).

Écrire un programme qui compte le nombre de fois où il a été lancé. Ceci pourrait se faire à l'aide de la sauvegarde d'un entier dans un fichier.

```
# ./prog
Programme lancé 1 fois
# ./prog
Programme lancé 2 fois
# ./prog
Programme lancé 3 fois
# ./prog
Programme lancé 4 fois
```

Exercice noté 43 (⋆ ⋆ ⋆ Codage Vigenère depuis fichier).

Écrire un programme qui lit du texte depuis un fichier texte. Puis l'encode ou le décode avec la méthode du chiffre de Vigenère pour l'afficher sans la sortie standard.

```
# ./prog
Attendu : ./prog [FICHIER MESSAGE] [CLE]
Attendu : ./prog [FICHIER MESSAGE] [CLE] decode
# ./prog message.txt ESGI
Ipkutdk li lkfxw zzsh ipsmkbxw !
Kvjat dsarà, çi hwbzeaz ti xgqvw.
```

Exercice noté 44 (⋆⋆⋆ Sauvegarde et chargement en binaire).

Écrire un programme qui sauvegarde et charge un personnage dans un fichier binaire. La manipulation du programme se fait en ligne de commande en passant des options au programme. L'écriture dans le fichier doit respecter la spécification suivante :

- (4 octets) entier : nombre de caractères constituant le nom du personnage.
- (N octets) chaîne de caractères : nom du personnage constitué de N caractères.
- (4 octets) entier : statistique de vie du personnage.
- (4 octets) entier : statistique d'attaque du personnage.
- (4 octets) entier : statistique de défense du personnage.
- (4 octets) entier : statistique de vitesse du personnage.

```
# ./prog
Attendu :
        ./prog -create [FICHIER] [NOM] [VIE] [ATK] [DEF] [VIT]
        ./prog -read [FICHIER]
# ./prog -create chouette.perso "Chouette Oiseau" 127 42 87 94
# ./prog -read chouette.perso
Personnage : {
  Nom : Chouette Oiseau
  Vie : 127
  Attaque : 42
  Défense : 87
  Vitesse : 94
}
```

```
# hexedit chouette.perso
00000000   0F 00 00 00  43 68 6F 75  65 74 74 65  ....Chouette
0000000C   20 4F 69 73  65 61 75 7F  00 00 00 2A   Oiseau....*
00000018   00 00 00 57  00 00 00 5E  00 00 00      ...W...^...
00000024
---  chouette.perso --0x0/0x23--------------------------------
```

Exercice noté 45 (⋆⋆⋆⋆ Enregistrement et recherche de numéros avec sauvegarde).

Écrire un programme qui :

- prend une option `-i [FICHIER]` pour ouvrir une liste de noms / numéros depuis un fichier existant.
- prend une option `-o [FICHIER]` pour enregistrer une liste de noms / numéros depuis la liste actuellement connue par le programme.

- Ouvre si elle existe une liste de noms indiquée.
- Propose à l'utilisateur l'ajout de nouvelles associations noms / numéros.
- Enregistre les associations dans une liste de noms indiquée.
- Permet la recherche d'un numéro depuis le nom associé.

```
# ./prog -o liste.txt
Nom (None pour arrêter) : Premier 1
Numéro : Nom (None pour arrêter) : Second 2
Numéro : Nom (None pour arrêter) : None
Nom à rechercher (None pour arrêter) :
>>> None
# ./prog -i liste.txt -o liste.txt
Nom (None pour arrêter) : Troisieme 3
Numéro : Nom (None pour arrêter) : None
Nom à rechercher (None pour arrêter) :
>>> Premier
Le numéro de "Premier" est 1
Nom à rechercher (None pour arrêter) :
>>> Second
Le numéro de "Second" est 2
Nom à rechercher (None pour arrêter) :
>>> Troisieme
Le numéro de "Troisieme" est 3
Nom à rechercher (None pour arrêter) :
>>> None
```