IN 101 - TD 03

Solution

Attention : désormais l'implémentation des exercices sera à faire en fin de TD, après que nous ayons examiné toutes les questions de réflexion et de structuration des algorithmes. Laissez le clavier...

1 Point fixe d'un tableau

≪Écrivez une fonction qui vérifie s'il existe un point fixe dans un tableau ≫

On appelle ≪point fixe d'un tableau ≫ un indice `i` tel que `t[i] == i`.

Soit un tableau `t` contenant, **par hypothèse**, uniquement des entiers naturels tous différents et triés en ordre croissant strict.

Q 1.1. Proposez un algorithme pour une fonction `fixpt` qui prend en argument un tel tableau et détermine s'il existe un point fixe dedans.

Solution

Si à un endroit dans le tableau, on a t[i] == i, par exemple, t[5] == 5, alors vu que tous les naturels sont strictement croissants, il ne peut y avoir dans les **5 cases précédentes**, que **5 autres** valeurs **inférieures strictement à 5**, **positives** et **différentes entre elles**.

Donc 0, 1, 2, 3 et 4. Comme elles doivent être en ordre **strictement croissant**, elles ne peuvent être que dans l'ordre 0, 1, 2, 3 et 4.

Donc forcément, s'il existe un point fixe quelque part, toutes les cases précédentes ont aussi un point fixe, en particulier la case 0. Il n'y a donc qu'à tester si la case 0 est un point fixe.

Attention, pour accéder à la case 0, il faut que le tableau contienne au moins une valeur. Donc si le tableau est vide, il faut gérer le cas particulier. Un tableau vide ne contient pas de point fixe manifestement. L'algorithme est alors simplement :

```
fixpt (t, len) =
  Si len = 0 retourner faux
  Retourner t[0] == 0
```

ou encore plus court :

```
fixpt (t, len) =
  Retourner (len <> 0) et (t[0] == 0)
```

2 Fou aux échecs

≪Écrivez une fonction qui vérifie si le déplacement d'un fou d'une position initiale à une position finale en **un coup** est conforme aux règles du jeu d'échecs. ≫

On rappelle que les échecs se jouent sur un damier de taille 8×8 et qu'un fou ne peut se déplacer qu'en **diagonale**.

On modélise le damier par une matrice de coordonnées comprises entre 0 et 7 inclus.

On **ne cherche pas** à vérifier si la position initiale est atteignable par les règles du jeu d'échec (par exemple, de par la disposition initiale des pièces, il est impossible de trouver un des deux fous noirs sur une case blanche au cours d'une partie respectant les règles).

L'échiquier ne contient que le fou (pas d'autres pièces qui pourraient le bloquer).

Q 2.1. Quel sont les domaines des entrées et des sorties ?

Solution

Le jeu d'échec se jouant sur un damier, chaque position est un point du plan, donc un **couple** d'entiers. Chaque coordonnée doit appartenir à $[0; 7]$. La fonction doit prendre en argument la position initiale et la position finale, donc elle doit prendre 2 **couples** d'entiers. Plutôt que fabriquer une `struct` à cet effet, nous nous simplifierons la tâche en passant 4 entiers à notre fonction.

Nous **ne tenterons pas** de restreindre ces entiers à être positifs en utilisant des `unsigned int` pour une raison que nous verrons après avoir répondu à la question suivante.

La fonction devant dire si « oui » ou « non » le déplacement est légal, elle retourne une valeur de vérité donc un **booléen**.

Q 2.2. Quelle sont les relations entre les positions initiales et finales d'un déplacement légal ?

Solution

Notons (x, y) les coordonnées de la position initiale et (x', y') celles de la position finale. Si le déplacement est diagonal, il faut que pour un certain k l'on ait :

$$\begin{aligned} x' &= x \pm k \\ y' &= y \pm k \end{aligned}$$

Donc :

$$\begin{aligned} x' &= x + k \quad \Rightarrow \quad k = x' - x \\ &\text{ou} \\ x' &= x - k \quad \Rightarrow \quad k = x - x' \end{aligned}$$

et

$$\begin{aligned} y' &= y + k \quad \Rightarrow \quad k = y' - y \\ &\text{ou} \\ y' &= y - k \quad \Rightarrow \quad k = y - y' \end{aligned}$$

Plutôt que de réfléchir en terme des combinaisons de cas, il suffit de remarquer que $x - x'$ et $x' - x$ se regroupent en $\mid x - x' \mid$. Donc le déplacement est légal si l'on a :

$$\begin{aligned} k &= \mid x - x' \mid \\ k &= \mid y - y' \mid \end{aligned}$$

et finalement si $\mid x - x' \mid = \mid y - y' \mid$.

Nous voyons ici pourquoi nous n'avons pas décidé d'utiliser des `unsigned int`. En effet, la soustraction de 2 `unsigned` donnerait un `unsigned` (avec quelle valeur si $x < y$?) et rendrait donc la fonction de valeur absolue totalement inutilisable !

Tout cela ne contraint pas la position d'arrivée à rester dans le damier. Il faudra donc en plus vérifier que x' et $y' \in [0; 7]$.

Et un ≪déplacement nul ≫ ? Avoir les positions initiale et finale égales, est-ce un déplacement conforme aux règles du jeu ? Le choix d'interprétation est libre mais **il faut se poser la question**. Pour la correction de cet exercice le choix est de considérer que c'est illégal puisqu'il n'y a pas à proprement parler de ≪déplacement ≫.

3 État des finances

≪On souhaite écrire un programme permettant de connaître le solde de chacune des personnes d'un groupe où les membres se doivent de l'argent. ≫

Exemple pratique

— Charlie doit à Alice 10.
— Alice doit à Bob 15.
— Charlie doit à Bob 7.
— Bob doit à Charlie 5.

La question à laquelle répondre est ≪quel est le solde de chacun après règlement de ces dettes ? ≫ La réponse (affichage souhaité) :

```
$ ./money.x bill2.txt
Alice : -5
Bob : 17
Charlie : -12
```

Q 3.1. Proposez une esquisse d'algorithme pour résoudre le problème de calcul des soldes (on s'occupera de l'acquisition des données plus loin).

Solution

Lorsqu'une personne A rembourse sa dette d à une personne B, on retire d du solde de A et on ajoute d à celui de B. Il suffit donc de mémoriser le solde de chaque personne et d'itérer sur les dettes en effectuant l'addition et la soustraction entre soldes concernés.

Pour mémoriser les soldes, on utilisera un **tableau** `balances` où chaque case correspond au solde d'une personne. Il faudra être capable de savoir quelle case (quel indice de tableau) correspond à quelle personne. On peut donc utiliser un second tableau `names` dans lequel on stockera les noms. Ainsi, à un indice i, `balances[i]` correspondra au solde de la personne de nom `names[i]`.

names		**balances**
Alice	0	
Bob	1	
Charlie	2	

Notons que l'on pourrait également utiliser un tableau à 2 dimensions, ou un tableau mémorisant des couples (par des `struct`) $(nom, solde)$.

```
Pour chaque ligne de transaction
  Le nom devant de l'argent est le premier
  Celui recevant et le montant est le second
  index_debit <- trouver l'index du nom devant de l'argent
  index_credit <- trouver l'index du nom recevant de l'argent
  balances[index_debit] <- balances[index_debit] - montant
  balances[index_credit] <- balances[index_credit] + montant
```

Nous n'avons pas envie de saisir au terminal toutes les informations, nous allons utiliser un fichier **texte** comme entrée. Pour manipuler un fichier **texte** il faut :

1. L'ouvrir : fonction `fopen`.
2. Y lire (ou y écrire) ce que l'on veut : fonction `fscanf` (ou `fprintf`).
3. Le fermer une fois que l'on n'en a plus besoin : fonction `fclose`.
4. Pour tester si l'on est arrivé à la **fin du fichier**, il faut avoir fait **au moins une lecture** avant d'utiliser la fonction dédiée : `feof`. C'est important car ça joue sur la forme de l'algorithme.

On ne s'intéresse pas à toutes les fonctionnalités des fichiers, on ne regarde que ce dont nous avons besoin. Nous détaillerons ces fonctions dans la partie implémentation. Il faut actuellement juste savoir que `fscanf` fonctionne comme `scanf` (que vous connaissez déjà), sauf qu'au lieu de récupérer les données au clavier, elle les récupère dans un fichier.

La structure d'un fichier de données est fixe. La **première** ligne contient, le nombre n de personnes impliquées dans le problème. Ensuite, séparés par des espaces ou des retours à la ligne (ce qui ne change rien pour `scanf`/`fscanf`), les n noms des personnes.

Suivent un nombre **quelconque** de lignes comportant à chaque fois : le nom de la personne **devant** de l'argent suivi du nom de celle **recevant** l'argent suivi du **montant** (un entier), le tout séparé par des **espaces**.

```
$ more bill2.txt
3
Alice Bob Charlie
Charlie Alice 10
Alice Bob 15
Charlie Bob 7
Bob Charlie 5
MyMachine:~
```

Q 3.2. Complétez votre esquisse d'algorithme de la question Q3.1 pour acquérir les données d'entrée.

Solution

```
Ouvrir le fichier
Lire la première valeur qui est le nombre de participants (nb_persons)
Créer le tableau de taille nb_person pour mémoriser les noms
Lire nb_person noms et les stocker dans le tableau
```

```
Créer le tableau de taille nb_person pour stocker les soldes, en mettant
                    0 dans chaque case
Lire 2 noms et un montant
Tant que la fin de fichier n'est pas atteinte
  Le nom devant de l'argent est le premier lu
  Celui recevant le montant est le second lu
  Rechercher pour chaque nom à quel indice du tableau de noms il se trouve
  Faire l'addition et la soustraction qui vont bien à ces indices dans
               le tableau des soldes
  Lire 2 noms et un montant
Fermer le fichier
Parcourir les 2 tableaux en paralèlle
  Afficher le nom et le solde final de la personne concernée
Libérer les tableaux créés
```

Q 3.3. Comment allez-vous retrouver l'indice du solde d'une personne dans le tableau de soldes à partir de son nom ? Quels sont les domaines d'entrée(s) et de sortie(s) de la fonction de reherche d'indice ?

Solution

Pour trouver à quel indice se trouve le solde d'une personne il faut trouver à quel indice se trouve son nom dans le tableau de noms. Il faut donc balayer le tableau jusqu'à trouver le nom recherché et retourner l'indice courant. La fonction prend donc en entrée le tableau de noms (donc de chaînes de caractères), la taille du tableau et le nom à rechercher (donc une chaîne de caractères) et retourne une valeur entière positive.

Attention, il faut gérer le cas où l'on ne trouve pas le nom recherché. C'est une erreur et l'on pourra choisir de retourner un indice impossible, par exemple -1. Cela signifie qu'il faudra gérer cette erreur dans la fonction qui calcule les soldes.

```
find_index_by_name (names, len, name)
  Parcourir names avec i de 0 à len - 1
    Si names[i] est égal à name alors on a trouvé, retourner i
  Retourner -1
```

Q 3.4. À quoi devrait-on faire attention en lisant une ≪ligne de transaction ≫ afin que le programme soit robuste ?

Solution

Pour être plus robuste, il serait intéressant de vérifier que chaque ligne de ≪transaction ≫ est bien formée (avec 2 chaînes et un entier). Dans le cas contraire, ce serait une erreur et il faudrait penser à libérer les ressources allouées (fichier et mémoire).

4 Implémentation

Q 4.1. Écrivez en `C` l'algorithme esquissé dans l'exercice 2 pour vérifier la légalité d'un déplacement. Votre programme prendra ses arguments sur la ligne de commande. Vous pourrez tester votre programme comparant ses résultats avec ceux qui suivent :

```
move_bishop (1, 1, 1, 1)    → false
move_bishop (1, 5, 4, 5)    → false
move_bishop (1, 5, 2, 4)    → true
move_bishop (2, 4, 1, 5)    → true
move_bishop (7, 6, 1, 0)    → true
move_bishop (8, 7, 1, 0)    → false
move_bishop (2, 2, -1, -1)  → false
```

Solution

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

int abs (int i) {
  if (i < 0) return (-i) ;
  return i ;
}


bool move_bishop (int x, int y, int x_prim, int y_prim) {
  if ((x < 0) || (y < 0) || (x > 7) || (y > 7) || (x_prim < 0) ||
      (y_prim < 0) || (x_prim > 7) || (y_prim > 7) ||
      (x == x_prim && y == y_prim))
    return false ;
  return (abs (x - x_prim) == abs (y - y_prim)) ;
}


int main (int argc, char *argv[]) {
  if (argc != 5) {
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }

  if (move_bishop
      (atoi (argv[1]), atoi (argv[2]), atoi (argv[3]), atoi (argv[4])))
    printf ("Legal\n") ;
  else printf ("Illegal\n") ;
  return 1 ;
}
```

Manipulation de fichiers en C

ATTENTION : Savoir lire dans un fichier est à maîtriser **absolument** car cela resservira dans d'autres TDs et possiblement en **examen**.

— L'ouverture d'un fichier se fait grâce à la fonction

FILE *fopen (**char** *filename, **char** *mode)

qui prend en argument le nom du fichier et une chaîne de caractères représentant le mode d'accès (`"rb"` pour lecture, `"wb"` pour écriture) et renvoie un pointeur sur un « *descripteur* » du fichier (de type `FILE`). Si l'ouverture échoue, `fopen` retourne le pointeur `NULL`.

— La lecture dans un fichier se fait par la fonction

int fscanf (FILE *stream, **char** *format, ...)

qui opère comme `scanf`, mais en lisant dans le fichier désigné par `stream`. Par rapport

à `scanf`, il faut juste passer en plus en premier argument le descripteur du fichier dans lequel lire. Chaque lecture consécutive «avance» automatiquement dans le fichier. Cette fonction retourne le nombre d'éléments effectivement lus (nombre de % réussis dans le format). Autrement dit, une lecture partiellement réussie retournera un entier strictement inférieur au nombre de % présents dans le format. Par contre, en cas d'échec total dû à l'atteinte de fin de fichier, elle retournera la valeur `EOF` (égale à -1).

— Pour savoir si l'on a atteint la fin du fichier, il faut interroger la fonction

int feof (FILE *stream)

après lecture. Elle renvoie «vrai» si la fin du fichier a été atteinte suite à cette tentative de lecture, «faux» sinon. Autrement dit, une lecture qui lit le dernier « mot » d'un fichier **ne** provoquera **pas** le signalement de fin de fichier. Ce sera la **prochaine** lecture qui, en échouant, permettra à `feof` de répondre «vrai».

— Finalement, lorsque l'on n'a plus besoin de travailler sur un fichier ouvert, il faut le fermer en utilisant la fonction

int fclose (FILE *stream)

Q 4.2. Écrivez la fonction `find_index_by_name` qui implémente l'algorithme de la question Q3.3.

Solution

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


int find_index_by_name (char *in_array[], int tlen, char *name) {
  for (int i = 0; i < tlen; i++) {
    if (strcmp (in_array[i], name) == 0) return i ;
  }
  return -1 ;
}
```

Q 4.3. Écrivez la fonction `compute_amount` qui implémente le reste de l'algorithme de l'exercice 3. Vous rajouterez un `main` prenant le nom du fichier de transactions en argument via la ligne de commande.

Solution

On remarquera dans ce programme le schéma de lecture caractéristique en C pour vérifier l'atteinte de fin de fichier. Puisque `feof` n'est significative que suite à la dernière tentative lecture, et qu'elle ne renvoie « *fin de fichier atteinte* » que si la tentative de lecture a **échoué**, il convient d'adopter un algorithme de la forme suivante.

```c
fscanf (in, "%d", &foo) ;
while (! feof (in)) {
  traitement qui va bien ;
  fscanf (in, "%d", &foo) ;
}
```

En particulier, le schéma suivant **est incorrect**.

```c
while (! feof (in)) {
  fscanf (in, "%d", &foo) ;    /* NON NON NON NON NON NON ! */
```

```
  traitement qui va bien ;
}
```

```
void compute_amounts (char *fname) {
  unsigned int nb_persons ;
  char **names ; /* Array of names of persons. */
  /* Temporary strings to fetch names. We could save one but let's keep 3
     instead of 2 for sake of readability. */
  char tmp_name[256], name_debit[256], name_credit[256] ;
  int *balances ;       /* Array of balances. */
  int amount ;     /* Read amount at each iteration in the file. */

  /* Open the file. */
  FILE *file = fopen (fname, "rb") ;
  if (file == NULL) {
    printf ("Error. Unable to open file.\n") ;
    return ;
  }

  /* Get the number of persones. */
  fscanf (file, "%u", &nb_persons) ;
  /* Create the array of persons. */
  names = malloc (nb_persons * sizeof (char*)) ;
  if (names == NULL) {
    printf ("Error. Unable to allocate the array of persons.\n") ;
    fclose (file) ;
    return ;
  }

  /* Fill the array of persons. */
  for (int i = 0; i < nb_persons; i++) {
    fscanf (file, "%s", tmp_name) ;
    names[i] = malloc ((strlen (tmp_name) + 1) * sizeof (char)) ;
    if (names[i] == NULL) {
      printf ("Error. Unable to allocate a name.\n") ;
      /* Free the names already allocated at this point. */
      for (i = i - 1; i >= 0; i--) free (names[i]) ;
      free (names) ;
      fclose (file) ;
      return ;
    }
    strcpy (names[i], tmp_name) ;
  }

  /* Create an array with a balance of 0 for each person. */
  balances = malloc (nb_persons * sizeof (int)) ;
  if (balances == NULL) {
    printf ("Error. Unable to allocate the array of balances.\n") ;
    for (int i = 0; i < nb_persons; i++) free (names[i]) ;
    free (names) ;
    fclose (file) ;
    return ;
  }
  for (int i = 0; i < nb_persons; i++) balances[i] = 0 ;

  /* Loop on the file until its end... */
  fscanf (file, "%s %s %d", name_debit, name_credit, &amount) ;
  while (!feof (file)) {
    /* Get the index of the person to debit. */
    int index_debit = find_index_by_name (names, nb_persons, name_debit) ;
    /* Get the index of the person to credit. */
    int index_credit = find_index_by_name (names, nb_persons, name_credit) ;
    if ((index_credit == -1) || (index_debit == -1)) {
      printf ("Error. Unable to find a person.\n") ;
      /* TODO : must free arrays and close the file. See how it is done
         at the end of the function. I'm lazy in this solution. */
      return ;
    }
```

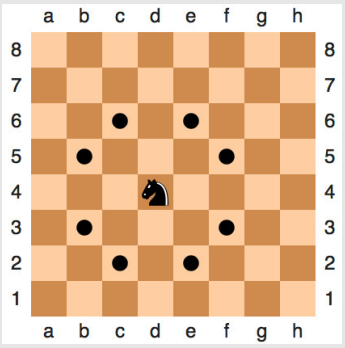
```
    /* Subtract / add the amount. */
    balances[index_debit] = balances[index_debit] - amount ;
    balances[index_credit] = balances[index_credit] + amount ;
    /* Read next line if any. */
    fscanf (file, "%s %s %d", name_debit, name_credit, &amount) ;
  }
  /* Close the opened file. */
  fclose (file) ;
  /* Summarize the debts/credits. */
  for (int i = 0; i < nb_persons; i++)
    printf ("%s : %d\n", names[i], balances[i]) ;

  /* Free the allocated memory. */
  for (int i = 0; i < nb_persons; i++) free (names[i]) ;
  free (names) ;
  free (balances) ;
}


int main (int argc, char *argv[]) {
  if (argc != 2) {
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }

  compute_amounts (argv[1]) ;
  return 0 ;
}
```

S'il vous reste du temps ou pour continuer après la séance.

5 Cavalier aux échecs

« Écrivez une fonction qui vérifie si le déplacement d'un cavalier d'une position initiale à une position finale en un coup est conforme aux règles du jeu d'échecs. »

Les déplacements possibles d'un cavalier sur un damier d'échec sont rappelés sur l'image ci-dessous (provenance Wikipédia) :



Q 5.1. Quelle sont les relations entre les positions initiales et finales d'un déplacement légal ?

Solution

L'image ci-dessus nous montre qu'il y a 8 déplacements possibles :

$$\begin{array}{ll} x' & = x+1 \\ & y' = y+2 \\ & y' = y-2 \end{array} \qquad \begin{array}{ll} x' & = x+2 \\ & y' = y+1 \\ & y' = y-1 \end{array}$$

$$\begin{array}{ll} x' & = x-1 \\ & y' = y+2 \\ & y' = y-2 \end{array} \qquad \begin{array}{ll} x' & = x-2 \\ & y' = y+1 \\ & y' = y-1 \end{array}$$

Encore une fois, $x' = x+1$ et $x' = x-1$ se réduisent à $\mid x - x' \mid = 1$ (et similairement pour ± 2 et pour y et y'. Ainsi un déplacement est légal si :

$$\begin{array}{lll} \mid x - x' \mid = 1 & \text{et} & \mid y - y' \mid = 2 \\ & \text{ou} & \\ \mid x - x' \mid = 2 & \text{et} & \mid y - y' \mid = 1 \end{array}$$

Tout comme dans l'exercice précédent, Il faut contraindre la position d'arrivée à rester dans le damier et rejeter le ≪déplacement nul ≫.

Q 5.2. Écrivez en `C` la fonction `move_knight` qui vérifie la légalité d'un déplacement. Vous pourrez tester votre programme comparant ses résultats avec ceux qui suivent :

`move_knight (1, 1, 1, 1)` → *false*
`move_knight (4, 5, 3, 7)` → *true*
`move_knight (3, 5, 2, 7)` → *true*
`move_knight (0, 3, -1, 2)` → *false*
`move_knight (4, 6, 5, 6)` → *false*
`move_knight (5, 5, 6, 7)` → *true*

Solution

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

int abs (int i) {
  if (i < 0) return (-i) ;
  return i ;
}


bool move_knight (int x, int y, int x_prim, int y_prim) {
  if ((x < 0) || (y < 0) || (x > 7) || (y > 7) || (x_prim < 0) ||
      (y_prim < 0) || (x_prim > 7) || (y_prim > 7) ||
      ((x == x_prim) && (y == y_prim)))
    return false ;
  return (((abs (x - x_prim) == 1) && (abs (y - y_prim) == 2))
          ||
          ((abs (x - x_prim) == 2) && (abs (y - y_prim) == 1))) ;
}


int main (int argc, char *argv[]) {
  if (argc != 5) {
    printf ("Error. Wrong number of arguments.\n") ;
    return 1 ;
  }

  if (move_knight
      (atoi (argv[1]), atoi (argv[2]), atoi (argv[3]), atoi (argv[4])))
    printf ("Legal\n") ;
  else printf ("Illegal\n") ;
  return 1 ;
```

```
}
```

6 Textruction

Vous avez peut-être déjà lu ce résultat d'études qui mnotre que l'rdore des letrtes dnas un mot n'a pas d'icnopmrtae, la sulee cohse iornapttme est que la prmeière et la drenière sinoet à la bnnoe palce.

On se propose d'écrire un programme permettant d'afficher le texte contenu dans un fichier en mélangeant les lettres des mots comme dans la phrase ci-dessus. Les permutations de lettres doivent ne pas changer la première ni la dernière lettre d'un mot. Bien évidemment, la ponctuation du texte ainsi que d'éventuels chiffres ne sont pas affectés par les permutations.

Q 6.1. Décrivez les grandes étapes de ce programme.

Solution

```
Vérifier présence de l'argument
Ouvrir fichier
Lire un mot
Tant que fin de fichier non atteinte
  Permuter les lettres du mot si besoin
  Afficher le mot modifié
  Lire un mot
Fermer fichier
```

Q 6.2. Partant du principe qu'un mot est une chaîne de caractères, décrivez le processus de permutation des lettres.

Solution

Puisque la première et la dernière lettre ne doivent pas bouger, il va falloir balayer les caractères d'indice 1 à « *la longueur du mot* **-2** ». En effet, les tableaux (donc chaînes) commençant à l'indice 0, le dernier caractère significatif d'une chaîne est à l'indice « *longueur du mot - 1* ». Comme nous ne devons pas toucher à la dernière lettre du mot, on s'arrête bien au caractère d'indice « *longueur du mot* **- 2** » inclu.

À chaque étape, on échangera le caractère à l'indice courant avec un autre du mot au hasard.

Puisque nous souhaitons permuter au hasard, il nous faut une fonction générant des nombres (pseudo-)aléatoires. La bibliothèque standard de C fournit la fonction `rand` à cet effet. Consultez sa documentation en utilisant la commande Unix `man` :

```
man rand
```

Q 6.3. Que fait cette fonction ? Quel fichier d'en-tête doit être utilisé pour y accéder ? Comment obtenir un nombre entre 0 et autre chose que le fameux `RAND_MAX` ?

Solution

Cette fonction génère un nombre entre 0 et `RAND_MAX`. Pour utiliser cette fonction, comme indiqué dans le `man`, il faut inclure le fichier d'en-tête `stdlib.h` de la bibliothèque standard de C. Il faut donc faire un **#include** <stdlib.h>

Pour ramener le nombre généré de l'intervalle $[0; \texttt{RAND\_MAX}]$ dans un intervalle $[0; n[$, on utilisera l'opération de modulo (reste de la division entière) : l'opérateur `%` de C. `(rand () % n)` donne bien un nombre dans entre 0 inclus et n exclu.

Q 6.4. Donnez l'expression qui permet de calculer un indice de caractère tiré au hasard parmi les lettres d'un mot dont on connaît la longueur (par exemple stockée dans une variable `len`).

Solution

`1 + (rand () % (len - 2))`

Puisque `rand` génère un nombre entre 0 et ... inclus, puisque les caractères à permuter commencent à l'indice 1, il faut bien rajouter 1 au nombre généré.

Pour la borne supérieure, il ne faut pas dépasser l'avant-dernière lettre du mot. Elle se trouve donc à l'indice `len - 2`. Si l'on fait un modulo par `(len - 2)`, alors on obtient un nombre **strictement inférieur** à `(len - 2)` donc qui désignera un indice de caractère strictement inférieur à l'avant-dernier caractère. MAIS, puisque nous avons rajouté 1, notre borne supérieure sera bien égale à `(len - 2)`.

Q 6.5. Décrivez l'algorithme de la fonction `shuffle` qui prend en argument une chaîne de caractères et effectue les permutations directement dedans. On sait qu'en C, une chaîne passée en argument peut être modifiée en place (implicitement un passage d'argument par adresse, comme pour tout tableau).

Solution

```
shuffle (str)
  len <- longueur de str
  Si len > 3
    len_moins_2 <- len - 2
    Pour i de 1 à len_moins_2 inclus
      j <- 1 + (rand () % len_moins_2)
      tmp <- str[i]
      str[i] <- str[j]
      str[j] <- tmp
```

Précédemment dans ce TD, les fonctions de manipulation de fichiers vous ont été présentées. Pour la suite il va falloir être capable de lire le contenu d'un fichier caractère par caractère. Il est bien entendu possible d'utiliser `fscanf` avec le format `%c`. Mais vous pouvez également utiliser la fonction

int getc (FILE *stream)

qui lit un seul caractère dans le fichier et le retourne en résultat (sous forme d'un entier entre 0 et 255). Si la lecture échoue car la fin du fichier a été atteinte, la fonction renverra la valeur `EOF` (autre façon dans ce type de lecture de tester la fin de fichier).

On souhaite maintenant écrire une fonction `next_word` qui devra lire le prochain mot dans un fichier dont le descripteur est passé en argument, transmettre ce mot à la fonction appelante et retourner « vrai » s'il reste des choses à lire dans le fichier, et « faux » sinon.

Q 6.6. Quels sont les domaines d'entrée et de sortie de cette fonction ?

Solution

Puisque la fonction doit retourne « vrai » ou « faux », elle doit retourner un `bool`.

Puisqu'elle doit lire dans un fichier, elle doit prendre en argument son descripteur (un `FILE*` en C).

Mais elle doit aussi transmettre à l'appelant le mot qu'elle aura lu. Comme elle retourne déjà un `bool`, elle ne peut pas retourner autre chose en C. Donc on va utiliser un passage par adresse. Ainsi, en sortie la chaîne prendra en argument une chaîne dans laquelle stocker l'éventuel mot lu.

Un mot est une suite contiguë de caractères **alphabétiques** (nommés ici « lettres »). Autrement dit, lecture et construction d'un mot s'arrêtent si le caractère lu est autre chose qu'une **lettre**.

Q 6.7. Décrivez le processus le lecture d'un mot.

Solution

On lit un caractère. Tant que le caractère lu est une **lettre** et que l'on n'a pas atteint la fin du fichier, on stocke ce caractère, on lit un caractère et on recommence.

Lorsque l'on arrête, soit c'est parce qu'on n'a rien lu car la fin du fichier est atteinte, soit que le caractère lu n'est pas une « lettre ». Donc on clôt la chaîne courante et on retourne « vrai » ou « faux » selon la raison pour laquelle la lecture s'est arrêtée.

Attention, dès que l'on arrête sous prétexte que le caractère lu n'est plus une lettre, le caractère aura été **lu** (« consommé » dans le fichier). Il faut donc s'en souvenir pour le retourner la prochaine fois que la fonction est appelée. Sinon on perdrait toute la ponctuation, les chiffres, les blancs du texte !

Et donc, au début de la fonction, avant de lire réellement un caractère dans le fichier, il faut regarder si un caractère autre que « lettre » n'avait pas été mémorisé le coup précédent. Si oui, c'est lui que l'on considérera comme étant le mot à retourner pour le présent appel de la fonction. On pourra utiliser une variable globale pour conserver cette information d'un appel sur l'autre à la fonction. Quand aucun caractère n'est en attente, on pourra initialiser cette variable à `'\0'` puisque ce caractère ne peut pas faire partie du contenu d'un fichier (c'est le marqueur de fin de chaîne).

```
pending_char <- '\0'

next_word (fichier, buffer) =
  Si pending_char <> '\0'
    Fabriquer dans buffer la chaîne avec uniquement le caractère pending_char
    Retourner vrai
  c <- lire un caractère
  Tant que fin de fichier non atteinte et c est alphabétique
    Rajouter c à la suite de buffer
    c <- lire un caractère
  Clore la chaîne buffer
  Si fin de fichier atteinte
    Si un mot a été construit, retourner vrai
    Sinon retourner faux
  Sinon
    pending_char <- c
    Retourner vrai
```

Pour savoir si un caractère est une « lettre », vous pouvez utiliser la fonction
int isalpha (**int** c)
disponible via un `#include <ctype.h>` et dont vous obtiendrez la documentation avec la commande `man` d'Unix.

Q 6.8. Comment allez-vous savoir si un mot qui vient d'être lu doit voir ses lettres permutées ou bien si l'on ne doit pas y toucher car c'est de la ponctuation, des chiffres ou des blancs ?

Solution

En fait, la question ne se pose même pas. Si l'on tombe sur un caractère qui n'est pas une « lettre », on retourne directement cet unique caractère comme chaîne. Ainsi, si l'on a le contenu de fichier « `Il y a 100 ans` », lorsque nous arriverons sur le caractère `'1'`, il sera retourné comme la chaîne `"1"`. Puis à l'appel suivant la chaîne `"0"` sera retournée et ainsi de suite. Et il en sera d'ailleurs de même pour les espaces que l'on aura rencontrés plus tôt.

Ainsi, on remarque que toute ponctuation ou autre séquence ne représentant pas un mot « alphabétique » sera retournée comme une chaîne de 1 caractère. Et vu que la permutation n'affecte que les chaînes d'au moins 4 caractères, on n'aura aucun cas particulier à gérer.

Q 6.9. Imnleptaz en C les fnocitons précédnmeemt conçues et rtujaeor un `main` qui premet de pndrere en aurmengt de ligne de comnmdae le nom du feihcir à taetirr, pius ahiffce son ctneonu en aiunqapplt les pmatitournes.
Ouch, le dernier mot n'est quand même pas facile à retrouver !

Solution

```c
#include <stdio.h>      /* For file operations. */
#include <stdbool.h>    /* For booleans. */
#include <ctype.h>      /* For isalpha. */
#include <stdlib.h>     /* For srand. */
#include <unistd.h>     /* For getpid. */
#include <string.h>



void shuffle (char *buffer) {
  int len = strlen (buffer) ;
  /* Only need to work of the word has more than 3 letters since first and
     last must not move. */
  if (len > 3) {
    int i ;
    int len_minus_two = len - 2 ;
    /* Iterate on all the letters between the first and the last ones, bounds
       excluded. */
    for (i = 1; i <= len_minus_two; i++) {
      /* Randomly chose 1 character index in the string between first and last
         letters excluded . */
      int j = 1 + (rand () % len_minus_two) ;
      /* Exchange it with the current character. */
      char tmp = buffer[i] ;
      buffer[i] = buffer[j] ;
      buffer[j] = tmp ;
    }
  }
}


/* If different from the null character ('\0'), this represent an already
   read character that is not alphabetic, hence that stopped the input
```

```
   of a string. Since this character was not put in the ended string but was
   really read, we must remind it to return it next time the lexer will be
   called. Otherwise, we would loose it.
   We don't export this variable in the .h since it is purely "local" to the
   next_word function. */
char pending_char = '\0' ;


bool next_word (FILE *in, char *buffer) {
  int i = 0 ;  /* Where to add in the string under construction the current
                  read character. */
  int found_char ;  /* Character currently read from the file. */

  /* If the previous time we already read a character that stopped the
     previous string construction, then we must use it to build the current
     string. */
  if (pending_char != '\0') {
    /* Make a string with this only character. */
    buffer[0] = pending_char ;
    buffer[1] = '\0' ;      /* Close the string. */
    /* Character is not anymore to process. Clear it. */
    pending_char = '\0' ;
    return (true) ;    /* End. */
  }

  /* "Else" ... no pending character, hence really read in the file. */
  found_char = fgetc (in) ;
  while (!feof (in) && isalpha (found_char)) {
    /* While the end of file is not reached and the read character is
       alphabetic, store it in the string under construction. */
    buffer[i] = found_char ;
    i++ ;
    /* Read next character. */
    found_char = fgetc (in) ;
  }

  /* Always close the current string. Even if we didn't read anymore
     characters. */
  buffer[i] = '\0' ;
  /* If we stopped because end of file we must ensure that we do not still
     have already read a word. If so, return true. */
  if (feof (in)) {
    if (i != 0) return (true) ;
    else return (false) ;/* Otherwise really nothing remaining, return false. */
  }
  else {
    /* Stopped *not* because end of file. Hence inevitably because a non
       alphabetic character was read. Hence, remind it for next turn and
       return telling that there is possibly something that remains to be
       read. String under construction was closed just above, hence is
       complete for this turn. */
    pending_char = found_char ;
    return (true) ;
  }
}


int main (int argc, char *argv[]) {
  FILE *in ;
  char buffer[256] ;

  /* Ensure there is only one argument on the command line. */
  if (argc != 2) {
    printf ("Error: wrong number of arguments. Expected <file name>.\n") ;
    return (1) ;
  }

  /* Attempt to open the file whose name was given on the command line. Open
     it in "read" mode. */
```

```
  in = fopen (argv[1], "rb") ;
  if (in == NULL) {
    /* Unable to open the file. */
    printf ("Error: unable to open file '%s'.\n", argv[1]) ;
    return (1) ;
  }

  /* Initialize random generator. */
  srand (getpid ()) ;
  /* While the lexing function doesn't tell we reached the end of the file... */
  while (next_word (in, buffer)) {
    /* Hack the letters of the read word. */
    shuffle (buffer) ;
    /* Print the hacked word. */
    printf ("%s", buffer) ;
  }

  /* Close the input file before exiting. */
  fclose (in) ;
  return (0) ;
}
```

Q 6.10. Lancez votre programme 2 fois de suite sur un même texte. Que remarquez-vous ?

Solution

Les permutations « aléatoires » sont les mêmes d'une exécution à une autre. Cela est dû au fait que la fonction `rand` n'est pas réellement un générateur aléatoire : seulement pseudo-aléatoire. Elle génère sa suite de nombres par un calcul, donc de manière totalement déterministe à partir d'un nombre initial : la graine.

Dans la solution proposée, on adopte cette dernière technique en utilisant la fonction `getpid` qui retourne le numéro unique du programme courant en cours d'exécution (c.f. ligne 11 dans `textruction.c`).