TP4: Correction

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ac-toulouse.fr

Exercice 1

L'archive du TP contient deux fichiers `quarante.txt` et `quarante.pas_txt`. Le premier contient le nombre 40 encodé en ASCII, c'est à dire le caractère '4' suivi du caractère '0'. Le second contient le nombre 40 encodé en entier signé sur 4 octets. L'archive contient également le code `ecrire_40.c` utilisé pour générer ce fichier. Il utilise des fonctions que l'on n'a pas encore vu, mais vous pouvez déjà l'ouvrir et constater que l'on recopie bien le contenu d'un `int` valant 40.

Q1. Les trois octets `0x34`, `0x30` et `0x0a` correspondent respectivement au code ASCII du caractère '4', au code ASCII du caractère '0', et au code ASCII du retour à la ligne.

Q2. La commande `od -t x1 -A x quarante.pas_txt` affiche que le fichier contient **quatre** octets: `0x28`, `0x00`, `0x00`, et `0x00`. `0x28` vaut $2\times16+8 = 40$, on reconnaît donc l'écriture de 40 sur 4 octets, en **petit-boutiste**.

Exercice 2

Effectivement, le fichier a bien été créé. Si jamais on inspecte son contenu avec la commande `od` de l'exercice précédent, on y voit les codes ASCII des différents caractères. Il est important de noter que lorsque l'on écrit un entier n avec le format `%d`, on n'écrit pas l'encodage de n, mais la suite des encodages individuels de ses chiffres en base 10. En particulier, si l'on avait écrit un plus grand nombre comme 123456, on aurait bien vu 6 octets et pas 4.

Exercice 3

Exercice 4

Q1. Pour la fonction `int premier_zero(char* filename)`, on peut commencer par reprendre le code de la fonction `somme` de l'exercice précédent, et l'adapter:

```c
/* Renvoie le nombre d'entiers strictement positifs lus dans le fichier
   `nom_fichier` avant d'y lire un 0. Renvoie -1 si le fichier ne contient
   aucun 0.
   Précondition: `nom_fichier` ne doit contenir que des entiers. */
int premier_zero(char* nom_fichier){
  FILE* f = fopen(nom_fichier, "r"); // ouverture en mode lecture
  assert(f != NULL);
  int x = 0; // entier lu dans le fichier
  int tot_positif = 0; // nombre d'entiers > 0 lus
  while (fscanf(f, "%d", &x) != EOF){
    if (x == 0){
      fclose(f);
      return tot_positif;
    } else if (x > 0){
      tot_positif++;
    }
  }
  // si l'on sort de la boucle, c'est qu'aucun 0 n'a été croisé
  fclose(f);
  return -1;
}
```

Notons que dans cette fonction, il y a deux endroits dans le code où l'on peut `return` et sortir de la fonction: il faut penser à fermer le fichier ouvert avant **chacun** des `return` !

Dans le main, en plus d'une version interactive du programme où l'on demande un nom de fichier à l'utilisateur et où l'on affiche le résultat, on peut écrire une version "test", où l'on appelle la fonction sur plusieurs fichiers qui ont été créés au préalable:

```
5 33 1 816 8 0 48 12 0 3
```

Figure 1: Fichier `test1.txt`

```
9 -5 7 -2 6 0
```

Figure 2: Fichier `test2.txt`

```
60 42 -12 17 198 -5
```

Figure 3: Fichier `test3.txt`

Les trois fichiers servent à tester la fonction dans les cas suivants:

- que des entiers positifs suivi d'un 0, et éventuellement d'autres données après, pour vérifier que le programme s'arrête au premier 0;
- un mélange d'entiers positifs et négatifs suivi d'un 0, pour vérifier que le programme ne compte pas les entiers négatifs;
- pas de 0, pour vérifier que la fonction renvoie bien -1 comme demandé.

D'où la fonction main:

```c
int main(){
  // TESTS
  assert(premier_zero("test1.txt") ==  5);
  assert(premier_zero("test2.txt") ==  3);
  assert(premier_zero("test3.txt") == -1);

  // MODE INTERACTIF

  char nom_fichier[51];
  // récupérer le nom du fichier
  printf("Veuillez entrer un nom de fichier:\n");
  scanf("%s", nom_fichier);

  int p = premier_zero(nom_fichier);
  if (p == -1){
    printf("Le fichier ne contient pas de 0\n");
  } else {
    printf("Le fichier contient %d entiers > 0 avant le premier 0\n", p);
  }
  return 0;
}
```

Q2. On peut lire plusieurs données d'un coup avec `fscanf`. On peut donc encore une fois adapter la boucle de lecture:

```c
/* pour chaque ligne de la forme:
   n mot
   dans le fichier filename, répète n fois `mot` */
void repeter(char* filename){
  int n;
  char mot[50];

  FILE* f = fopen(filename, "r");
  while(fscanf(f, "%d %s", &n, mot) != EOF){
    for (int i = 0; i < n; ++i){
      printf("%s ", mot);
    }
    printf("\n");
  }
  fclose(f);
}
```

Exercice 5

Q1. On voit bien que le programme affiche chacun des arguments qu'on lui fournit. De plus, il affiche systématiquement son propre nom: on peut donc en conclure que `argv[0]` est le nom de l'exécutable.

Q2. En suivant l'indication du sujet: on lit caractère par caractère, en recopiant chaque caractère lu depuis le fichier d'entrée directement dans le fichier de sortie:

```c
/* Copie in_fn dans out_fn */
void copieur(char* in_fn, char* out_fn){
  FILE* in_f  = fopen(in_fn,  "r");
  FILE* out_f = fopen(out_fn, "w");
  assert(in_f  != NULL);
  assert(out_f != NULL);
  char c; // caractère lu
  while (fscanf(in_f, "%c", &c) != EOF){
    fprintf(out_f, "%c", c);
  }
  fclose(out_f);
  fclose(in_f);
}
```

Puis, dans le main, il suffit d'appeler cette fonction avec les deux premiers arguments du programme. Si l'utilisateur n'a pas fourni assez d'arguments, on affiche une erreur:

```c
int main(int argc, char** argv){
  if (argc < 3){
    printf("Veuillez fournir deux noms de fichiers.\n");
    return 0;
  }
  copieur(argv[1], argv[2]);
  return 0;
}
```

Exercice 6

Q1. Il faut vérifier les tableaux case par case, on ne peut pas se contenter de tester si `t1 == t2`. En effet, `t1 == t2` teste l'égalité des adresses mémoires, mais pas le contenu des tableaux !

```c
/* Renvoie true si les n premiers éléments
   de t1 et t2 sont égaux */
bool egaux(int* t1, int* t2, int n){
  for (int i = 0; i < n; ++i){
    if (t1[i] != t2[i]){
      return 0;
    }
  }
  return true;
}
```

Q2. La fonction:

```c
/* Renvoie un tableau de n zéros puis m uns */
int* zeros_uns(int n, int m){
  assert(n >= 0 && m >= 0);
  int* res = malloc((n + m) * sizeof(int));
  for (int i = 0; i < n; ++i){
    res[i] = 0;
  }
  for (int i = n; i < n+m; ++i){
```

```
    res[i] = 1;
  }
  return res;
}
```

Un exemple de test unitaire dans le main:

```
  int* test_q2 = zeros_uns(3, 5);
  int attendu_q2[8] = {
    0, 0, 0,
    1, 1, 1, 1, 1
  };
  assert(egaux(test_q2, attendu_q2, 8));
  free(test_q2);
```

Q3. La fonction:

```
/* Renvoie un tableau de n cases contenant 0, 1, ..., n-1 */
int* range(int n){
  assert(n >= 0);
  int* res = malloc(n * sizeof(int));
  for (int i = 0; i < n; ++i){
    res[i] = i;
  }
  return res;
}
```

et un test unitaire:

```
  int* test_q3 = range(6);
  int attendu_q3[6] = {0, 1, 2, 3, 4, 5};
  assert(egaux(test_q3, attendu_q3, 8));
  free(test_q3);
```

Exercice 7

Dans cet exercice, on veut lire dans un fichier jusqu'à ce que l'une des deux conditions suivantes soit atteinte:

- On a lu n entiers
- On a atteint la fin du fichier

Autrement dit, on lit **tant que** les deux conditions sont fausses. Cela donner une solution possible:

```
/* Renvoie un tableau d'au plus *n entiers lus
   dans le fichier filename. Stocke dans *n le
   nombre d'entiers effectivement lus. */
int* lire_entiers(char* filename, int* n){
  FILE* f = fopen(filename, "r");
  assert(f != NULL);
  int i = 0; // nb d'entiers lus
  int* t = malloc(*n * sizeof(int)); // entiers lus

  bool fini_de_lire = false;
  while(!fini_de_lire && i < *n){
```

```
12      int nb_lus = fscanf(f, "%d", &t[i]);
13      if (nb_lus == EOF){
14        fini_de_lire = true; // termine la boucle
15      } else {
16        i++;
17      }
18    }
19    // en sortie: soit on a atteint EOF, soit on a lu n entiers.
20    // Dans tous les cas, i contient le nombre d'entiers lus
21    *n = i;
22    fclose(f);
23    return t;
24  }
```

Un point important: quand vous ouvrez un fichier, pensez à **vérifier qu'il n'est pas NULL**: si le fichier est NULL, c'est que l'ouverture a échoué (typiquement, vous avez mal écrit le nom du fichier, ou vous ne vous trouvez pas au même endroit dans le terminal et vous n'avez pas mis le chemin menant au fichier).

Dans les tests, on peut penser à trois cas à tester:

- un cas où la fonction atteint la fin du fichier avant d'avoir lu n entiers;
- un cas où la fonction lit n entiers avant d'avoir atteint la fin du fichier;
- un fichier vide.

(Voir fonction main dans le dossier de code).

Une remarque: dans la solution proposée, on renvoie un tableau de la taille demandée par l'utilisateur, y compris si on a lu beaucoup moins d'entiers. On pourrait ajouter quelques instructions à la fin de la fonction pour allouer un deuxième tableau faisant pile la bonne taille, et y recopier les valeurs:

```
1   // réallouer un autre tableau faisant pile la bonne taille
2   int* res = malloc(i*sizeof(int));
3   for (int j = 0; j < i; ++j){
4     res[j] = t[j];
5   }
6   free(t);
7   fclose(f);
8   return res;
```

Vous pouvez aussi vous renseigner sur la fonction `realloc` (en tapant `man realloc` dans le terminal par exemple, ou en regardant sur internet) et voir comment on aurait pu l'utiliser dans cet exercice !