Corrigé TP3

1 Arguments de la ligne de commande en C

[a] `argc` représente le nombre d'argument passés au programme par la ligne de commande. Par exemple, quand on exécute :

```
./programme abc toto 123
```

Le nombre d'arguments de cette commande, et donc `argc`, est 4.

"./programme"
"abc"
"toto"
"123"

NB : On voit que le nom du programme fait partie des arguments. Il est toujours présent et c'est toujours le premier, à l'indice 0 du tableau des arguments.

[b] Sans plus de modification, le programme affiche seulement le signe `'='` et la valeur à laquelle la somme a été initialisée.

[c] Une implémentation possible du programme `somme` :

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

int main (int argc, char *argv[]) {
   int i, somme = 0;

   if (argc == 1) {
      printf ("USAGE: ./prog number...\n");
      exit(EXIT_FAILURE);
   }

   for (i=1 ; i<argc ; i++) {
      somme += atoi(argv[i]);

      printf("%s",argv[i]);
      if (i < (argc - 1)) {
         printf(" + ");
      }
   }

   printf(" = %d\n", somme);
   return 0;
}
```

2 Lecture depuis un fichier, écriture dans un fichier

[d] Si le nombre d'argument passé est incorrect, la plupart du temps, cela signifie que l'utilisateur ne sait pas comment fonctionne le programme. On affiche donc un récapitulatif des options disponibles. On appelle cela un **USAGE**.

```
USAGE: ./prog FICHIER
```

Implémentation possible pour `mes_entrees_sorties` :

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

int main (int argc, char *argv[]) {
   if (argc != 2) {
      printf("USAGE: ./prog FICHIER\n");
      exit(1);
   }

   char *filename = argv[1];
   FILE *f = fopen(filename, "r") ;
   if (f == NULL) {
      printf("%s n'a pas pu être ouvert en lecture\n", filename) ;
      perror(filename);
      return 1 ;
   }

   char c ;
   fscanf(f, "%c", &c) ;
   while (!feof(f)) {
      printf("%c", c) ;
      fscanf(f, "%c", &c) ;
   }

   fclose(f) ;
   return 0 ;
}
```

[e] Implémentation possible pour `mes_entrees_sorties` avec la consigne complémentaire :

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

void transfer (FILE *src, FILE *dst) {
   char c;
   fscanf(src, "%c", &c);
   while (!feof(src)) {
      fprintf(dst, "%c", c);
      fscanf(src, "%c", &c);
   }
}

void cat (char *srcname) {
   FILE *src = fopen(srcname, "r");
   if (src == NULL) {
      printf("%s n'a pas pu être ouvert en lecture\n", srcname);
      perror(srcname);
      exit(1);
   }

   transfer (src, stdout);

   fclose (src);
}

void cp (char *srcname, char *dstname) {
   FILE *src = fopen(srcname, "r");
   if (src == NULL) {
      printf("%s n'a pas pu être ouvert en lecture\n", srcname);
      perror(srcname);
      exit(1);
   }

   FILE *dst = fopen(dstname, "w");
   if (dst == NULL) {
      printf("%s n'a pas pu être ouvert en écriture\n", dstname);
      perror(dstname);
      exit(1);
   }

   transfer (src, dst);

   fclose (src);
   fclose (dst);
}

int main (int argc, char *argv[]) {
   if (argc == 1 || argc >= 4) {
      printf("USAGE: ./prog FICH_SRC [FICH_DST]\n");
      return 1;
   }

   if (argc == 2) {
      cat (argv[1]);
   } else {
      cp (argv[1], argv[2]);
   }

   return 0;
}
```

[f] On observe que `cat *.c` affiche successivement tous les fichiers C expansés par l'expression `*.c`. Une modification possible de `mon_cat.c` pour reproduire ce comportement :

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

void cat (char *srcname) {
   FILE *src = fopen(srcname, "r");
   if (src == NULL) {
      printf("%s n'a pas pu être ouvert en lecture\n", srcname);
      perror(srcname);
      exit(1);
   }

   char c;
   fscanf(src, "%c", &c);
   while (!feof(src)) {
      printf("%c", c);
      fscanf(src, "%c", &c);
   }

   fclose (src);
}

int main (int argc, char *argv[]) {
   if (argc < 2) {
      printf("USAGE: ./prog FICHIER\n");
      exit(1);
   }

   for (int i=2; i<argc; i++) {
      cat (argv[i]);
   }

   return 0 ;
}
```

[g] On observe que `cat` appelé sans argument lit les données à afficher sur l'entrée standard. C'est à dire que lorsqu'on lance la commande :

```
cat
```

Le terminal se bloque en attente de l'entrée utilisateur. On peut alors taper du texte au clavier. Puis, lorsque l'on tape `entrée`, la ligne que l'on vient de terminer est ré-affichée par `cat`. Ce comportement se poursuit jusqu'à que l'on interrompe le flux en tapant Ctrl + D.

Reproduire ce comportement est un peu difficile car il n'existe pas de moyen simple pour faire un scanf sur une ligne en C, c'est à dire jusqu'à un retour chariot. On peut utiliser `"%c"` pour un caractère, `"%s"` pour un mot, mais il n'y a pas de marqueur pour la ligne entière. De plus, il est impossible de prévoir à l'avance la taille des lignes que l'utilisateur va entrer au clavier. Pour se prémunir d'un débordement de tableau, il faut donc être très vigilent. Voir à la page suivante une implémentation possible de `cat` qui reproduit le comportement de la commande originale :

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

#define MAX 100

void cat_stdin () {
   char buffer[MAX];
   int i = 0;
   scanf("%c", buffer+i++);
   while (!feof(stdin)) {
      if (buffer[i-1] == '\n') {
         buffer[i] = '\0';
         i = 0;
         printf("%s", buffer);
      }

      if (i > 98) {
         buffer[i] = '\0';
         i = 0;
         printf("%s\n", buffer);

         scanf("%c", buffer);
         while (!feof(stdin) && buffer[0] != '\n')
            scanf("%c", buffer);
      }

      scanf("%c", buffer+i++);
   }
}

void cat (char *srcname) {
   FILE *src = fopen(srcname, "r");
   if (src == NULL) {
      printf("%s n'a pas pu être ouvert en lecture\n", srcname);
      perror(srcname);
      exit(1);
   }

   char c;
   fscanf(src, "%c", &c);
   while (!feof(src)) {
      printf("%c", c);
      fscanf(src, "%c", &c);
   }

   fclose (src);
}

int main (int argc, char *argv[]) {
   if (argc < 2) {
      cat_stdin ();
   } else {
      for (int i=2; i<argc; i++) {
         cat (argv[i]);
      }
   }

   return 0 ;
}
```

3 Caractères imprimables

[h] Pour trouver l'ensemble des caractères imprimables, on utilise la commande `man ascii`, qui contient un rappel de la table ASCII. On constate que les caractères imprimables forment un bloc contigu. Ce sont les caractères compris entre le `'!'` et le `' '`. C'est à dire ceux dont le code est compris entre 33 et 126. Voilà une implémentation possible de `occurences` :

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

int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("USAGE: %s [FICHIER]", argv[0]);
      exit(1);
   }

   char *filename = argv[1];
   FILE *file = fopen(filename, "r");
   if (file == NULL) {
      printf("Impossible d'ouvrir le fichier %s en lecture\n", filename);
      perror(filename);
      exit(1);
   }

   // 93 = '~' (126) - '!' (33)
   int occurences[93] = {0};
   char c = 0;

   fscanf(file, "%c", &c);
   while (!feof(file)) {
      if (c > '!' && c < '~') {
         occurences[c - '!']++;
      }
      fscanf(file, "%c", &c);
   }

   printf("caractère\tnombre\n");
   for (int i=0; i<93; i++) {
      printf("%c\t\t%d\n", i + '!', occurences[i]);
   }
}
```

4 La commande de la semaine : find

[i] Réponse lorsque j'aurai accès à Turing !