Corrigé TP4

[a] Code décimal du caractère 'A' : **65**
Code décimal du caractère '5' : **53**
Code décimal du caractère ' ' (espace) : **32**
Caractère représenté par le code décimal 109 : **m**
Caractère représenté par le code décimal 77 : **M**

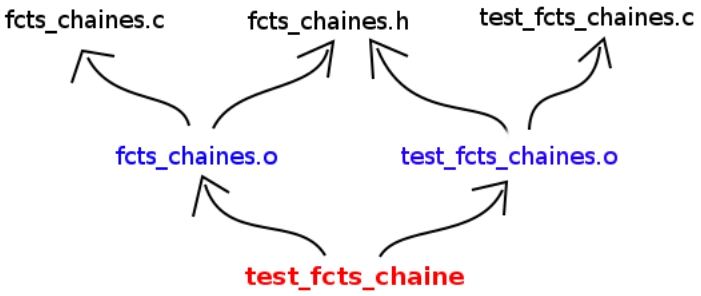
[b] Intervalle des codes décimaux correspondant à des caractères visibles : de **33** ('!') à **126** ('~')

[c] Programme imprimant les caractères visibles ainsi que leur code décimal :

```c
/**
 * Fichier de démonstration pour l'Université Grenoble Alpes
 *
 * @autor <corentin@marciau.fr>
 */

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

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

   for (i=33; i<=126; i++) {
      printf("Code décimal du caractère '%c' : %d\n", i, i);
   }

   return 0;
}
```

[d] Les chaînes de caractères doivent se terminer par le caractère \0 en C.

[e] Graphe des dépendances de `test_fcts_chaine` :



[f] La fonction min_2_maj supplémentaire n'implique pas d'ajout de fichier ni modification des dépendances des fichiers. Il n'y a donc pas besoin de changer le Makefile pour la prendre en compte.

[g] Les fonctions `mon_strcmp` et `min_2_maj` prennent en arguments deux pointeurs vers des tableaux de `char`. Considérons le bout de code suivant :

```
int mon_strcmp(char s1[], char s2[]) {
  [...]
}

int main(int argc, char * argv[]) {
  char chaine1[10] = "abcdefghi";
  char chaine2[10] = "jklmnopqr";

  mon_strcmp(chaine1, chaine2);
}
```

En langage C, le passage des paramètres se fait **par copie**.

On pourrait donc penser que lors de l'appel de `mon_strcmp`, tout le contenu des deux tableaux de caractères `chaine1` et `chaine2` serait copié dans les nouvelles variables `s1` et `s2`. **Ce n'est pas le cas !** En effet, les paramètres de la fonction sont de type `char[]` (équivalent à `char*`) et sont donc des pointeurs. On a vu en TD que lorsqu'on déclare par exemple :

```
int tab[10];
```

on obtient dans la variable `tab` un pointeur contenant l'adresse de la première case du tableau.

Ainsi, lorsqu'on appelle `mon_strcmp` avec comme arguments chaine1 et chaine2, ce ne sont pas les tableaux entiers qui sont copiés mais simprement les variables contenant leurs adresses.

[h] Taille du type `unsigned char` : **1** octet, c'est à dire **8** bits
Taille maximale d'un entier représenté avec un `unsigned char` : **255**
Intervalle d'entiers tenant sur un `unsigned char` : de **0** à **255**

[i] Taille du type `short int` : **2** octet, c'est à dire **16** bits
Taille maximale d'un entier représenté avec un `short int` : **32 767**
Intervalle d'entiers tenant sur un `short int` : de **-32 768** à **32 767**

[j] Le temps d'exécution de `debordechar` est de l'ordre de la miliseconde. Il n'est donc pas perceptible. Idem pour `debordeshort`. En revanche, e temps d'exécution de debordeint est d'environs 24 secondes selon l'ordinateur sur lequel le programme est exécuté.
On peut mesurer le temps d'exécution d'une commande shell à l'aide de la commande `time`. Ainsi :

```
user@computer$ time ./debordechar

real    0m0.002s
user    0m0.000s
sys     0m0.000s
```

La taille d'un `long` (8 octets) est plus grande que celle d'un `int` d'un facteur

$$2^{64}/2^{32}$$

soit environs 4 milliards. Il n'est donc pas possible de mesurer le temps d'exécution de `debordelong` car le DLST ferme la nuit.

[k] Conversion d'une chaine de caractères en entier selon l'algorithme de Horner :

```c
/**
 * Fichier de démonstration pour l'Université Grenoble Alpes
 *
 * @autor <corentin@marciau.fr>
 */

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

int mon_atoi (char num[]) {
   int i = 0;
   int resultat = 0;

   while (num[i] != '\0') {
      int digit = num[i] - '0';
      resultat = (resultat * 10) + digit;
      i++;
   }

   return resultat;
}

int main (int argc, char *arcv[]) {
   char input[10] = "";

   printf ("Entrez un nombre : \n");
   scanf ("%s", input);

   int resultat = mon_atoi (input);
   printf ("Vous avez entré : %d\n", resultat);

   return 0;
}
```

[l] Le script `make_gen.sh` se décompose en plusieurs blocs :

Bloc 1 Comme vu en TP3, le bloc 1 déclare une fonction qui prend un nom de fichier en argument, en extrait à l'aide de la commande `grep` toutes les lignes contenant "#include", puis extrait de chaque ligne le nom du fichier inclu à l'aide de la commande `sed`. La liste de tous les fichiers obtenus séparés par des espaces est ensuite imprimée sur la sortie standard.

Bloc 2 Le bloc 2 renvoie une erreur si le script n'a pas été appelé avec un unique argument. Un message d'erreur indique que cet argument doit être le nom de l'exécutable que l'on souhaite obtenir.

Bloc 3 Le bloc 3 déclare la variable PRGM qui contient le premier argument passé au script, soit le nom du programme qu'on souhaite obtenir.

Bloc 4 Le bloc 4 effectue les actions suivantes :

- Il déclare la variable `DEPPRGM` qui contient une cible de Makefile pour fabriquer le programme principal que l'utilisateur a donné en premier argument du script.
- Il déclare la variable `COMPRGM` qui contient la commande Makefile à exécuter pour la cible contenue dans `DEPPRGM`, la ligne commançant bien par une tabulation ("`\t`") et pas par des espaces.
- Ensuite, pour chaque fichier se terminant par `.c`, le nom du fichier sans son extension est obtenu à l'aide de la commande `basename` et on y ajoute ensuite l'extension `.o`. On obtient ainsi le fichier objet que l'on souhaite construire pour satisfaire les dépendances du programme principal.
- Ces fichiers objet sont ensuite ajoutés à la fin des deux variables `DEPPRGM` et `COMPRGM`.
- Enfin, la cible ainsi construite et sa commande sont ajouté dans un fichier Makefile qui est écrasé s'il existait déjà.

Bloc 5 Le dernier bloc ajoute au Makefile les cibles pour construire tous les fichiers objet dont le programme principal va dépendre.

[m] Script `make_gen.sh` complété :

```
#!/bin/bash

#####################################
# Bloc 1
extract_headers()
{
    for i in $(grep '#include "' $1 | sed 's/#include "//' | sed 's/"//')
    do
       echo -n $i" "
    done
    echo
}

#####################################
# Bloc 2
if [ $# -ne 1 ]
then
    echo Donnez le nom du programme principal en argument
    exit 1
fi

#####################################
# Bloc 3
PRGM=$1

#####################################
# Bloc 4

DEPPRGM="$PRGM: "
COMPRGM="\tgcc -o $PRGM "
for FILE in *.c
do
    FILEO=$(basename $FILE .c).o
    DEPPRGM=$DEPPRGM"$FILEO "
    COMPRGM=$COMPRGM"$FILEO "
done

echo -e $DEPPRGM > Makefile
echo -e $COMPRGM >> Makefile

#####################################
# Bloc 5
for source_file in *.c
do
    object="$(basename $source_file .c).o"
    headers=$(extract_headers $source_file)
    echo -e "$object: $source_file $headers" >> Makefile
    echo -e "\tgcc -c $source_file" >> Makefile
done
```