# TME1 : Outils de développement de programes C



# Objectifs

L'objectif des deux premiers TMEs est de vous permettre de vérifier que vous maîtrisez les principaux outils de développement utilisés pour programmer efficacement en langage C sous UNIX. Attention : Cette U.E. suppose que vous connaissez le langage C. Si vous ne le connaissez pas, ou si vous avez des lacunes, vous devez vous formez par vous même. Les bibliothèques sont faites pour cela, et vous pouvez utiliser les machines des salles de TP en libre service pour faire des exercices.

On utilisera les outils suivants:

- vim ou emacs : éditeur de texte
- indent : outil d'indentation automatique
- gcc : préprocesseur, compilateur, éditeur de liens
- ar : constructeur d'archives
- make : génération automatique avec gestion des dépendances
- gdb : debogueur

L'application logicielle proposée est une application de manipulation d'images au format Pgm. Les algorithmes réalisent fondamentalement des parcours de tableaux à 2 dimensions.

Commencez par créer un répertoire "tme1", qui contiendra tous les fichiers utilisés dans ce TME.

# Etape 1 : programme principal

Commencez par créer un sous-répertoire tme1/etape1 dans le répertoire tme1, et placez-vous dans ce répertoire. Dans cette première étape, vous allez devoir écrire deux fichiers : un fichier 'main.c' contenant la fonction main(), et un fichier 'Makefile' permettant de compiler le programme et de générer un fichier exécutable 'pgmg'.

Le fichier Makefile doit contenir deux règles :

```
pgmg : main.c
```

qui effectue la compilation ET l'édition de liens en une seule passe.

```
clean
```

qui efface les fichiers temporaires et un éventuel 'core', pour ne conserver que les fichiers sources.

Le programme 'main()' va lire le fichier 'input_file_name', en utilisant la fonction readpgm(), et recopier chaque octet dans un tampon intermédiaire que vous devez allouer. Il va ensuite recopier le contenu de ce tampon dans un fichier 'output_file_name', en utilisant la fonction writepgm(). Le prototype de la fonction main() est défini comme suit:

```
int main(int argc, char *argv[])
```

Vous trouverez les prototypes des fonctions de lecture et d'écriture du format pgm dans la page PgmInputOutput.

L'édition de liens doit se faire avec la bibliothèque qui se trouve dans

```
/users/enseig/encadr/cao/lib/libpgmio.a
```

Un fichier contenant une image au format 'pgm' est disponible dans

```
/users/enseig/encadr/cao/tme1/Untitled.pgm
```

Ce fichier peut être visualisé en utilisant la commande

```
display Untitled.pgm
```

# Etape 2 : Analyse de la ligne de commande

Commencez par créer un second sous-répertoire tme1/etape2 dans votre répertoire tme1. Recopiez les fichiers main.c et Makefile de dans ce sous-répertoire. La seconde étape consiste à écrire un fichier 'getarg.c', qui contient la fonctions getarg(). La fonction main() ne fait plus l'analyse de la ligne de commande, cette tâche étant réalisée par la fonction getarg(). Le fichier 'getarg.c' doit contenir auss la déclaration de la variable globale mainarg, de type 'mainarg_t', qui contient les valeurs de tous les paramètres lus par la fonction getarg().

Le fichier 'getarg.h' doit contenir la définition du type mainarg_t, et la déclaration du prototype de la fonction getarg():

```
typedef struct mainarg_t {
char     *InputFileName;
char     *OutputFileName;
char     *CommandFileName
char     verbose;
} mainarg_t;

extern mainarg_t  mainarg;

extern void getarg(int argc, char argv[]);
```

On souhaite que le prototype du programme 'pgmg' soit le suivant:

```
pgmg [-c command_file_name] [-v] input_file_name output_file_name
```

Les deux arguments optionnels doivent pouvoir être placés n'importe où sur la ligne de commande.

- -v est un drapeau qui demande au programme d'afficher un message à chaque étape du traitement.
- -c command_file_name permet de définir une série de traitement à appliquer à l'image, dans un fichier de commandes.

On se contente dans cette étape d'analyser les arguments de la ligne de commande. L'analyse du des commandes contenues dans le fichier de commande sera réalisée à l'étape suivante.

# Etape 3 : Analyse du fichier de commandes

Commencez par créer un sous répertoire tme1/etape3, et à recopier vos fichiers source et votre Makefile dans ce répertoire, pour conserver les états intermédiaires de votre travail. Cette troisième étape consiste à écrire un parseur

élémentaire utilisant les fonctions fgets() et sscanf(). Vous pouvez vous inspirez de l'exemple que vous trouverez dans ScanfExample.

Le fichier de commandes contient une commande par ligne.

- S n : seuillage (si val > n, val = 255 / sinon val = 0)
- X : symétrie suivant x (x devient -x)
- Y : symétrie suivant y (y devient -y)
- P : rotation positive de 90°
- M : rotation négative de 90°
- F C00 C10 C20 C01 C11 C21 C02 C02 C12 C22 : filtrage produit de convolution avec une matrice 3*3

Voici un exemple de fichier de commandes:

```
# une ligne commencant par un # est un comentaire
X
Y
P
M
S 12
F 1 -1 1 2 10 2 1 -1 1
```

Ecriver le fichier operate.C (ainsi que le fichier associé operate.h), qui contient la fonction operate(). Cette fonction prend en entrée une image définie par un pointeur sur une structure de donnée gmap. Elle applique séquenciellement les transformations définies dans le fichier de commandes, et renvoie un pointeur sur la structure de donnée gmap contenant l'image résultante.

```
gmap *operate(FILE *commande, gmap *in);
```

On définira une fonction par type de commandes. Ces 6 fonctions prennent pour nom le caractère définissant la commande, et possèdent deux arguments: un pointeur sur l'image source, et un pointeur sur l'image résultat. Les deux structures de données doivent donc avoir été allouées en mémoire préalablement : pas de malloc() dans ces fonctions.

```
void X(gmap *mapin, gmap *mapout);
```

Pour cette étape, les fonctions de traitement se contentent de recopier mapin dans mapout, et d'afficher le message:

```
Operation XXX non implementee
```

# Etape 4 : opérations S,X,Y,P,M,F

Implémentez successivement les fonctions S, X, Y P, M. F Il faut calculer tous les pixels de l'image, en exécutant deux boucles for imbriquées pour parcourir les lignes et les colonnes. Le coeur de ces fonctions sera toujours de la forme :

```
for(y = 0 ; y < heigth ; y++) {
  for(x = 0 ; x < width ; x++) {
    ELM(mapout, U(x,y), V(x,y)) = ELM(mapin, x, y);
  }
}
```

Pour alléger l'écriture, on définira la macro ELM(map,x,y) qui permet de désigner le pixel de coordonnées (x,y) dans l'image désignée par le pointeur map.

```
#define ELM(map,x,y)  map->raster[y*map->width + x]
```

La difficulté est évidemment de définir les expressions arithmétiques U(x,y) et V(x,y) pour chacune des opérations à implémenter.

# Compte-rendu

La note de tME compte dans la note de contrôle continu. Vous n'avez pas de compte-rendu écrit à rendre, mais vous aurez à faire une démonstration de votre programme et à présenter le code au début de la séance de TME de la semaine suivante.

N'oubliez pas de commenter votre programme...