TP N°6 – Entrées/Sorties bas niveau
Les fichiers BMP

Attention à respecter les instructions suivantes :

1. vous **devez** utiliser l'outil `make` pour compiler vos programmes. Une introduction à cet outil est disponible sur moodle ;
2. chaque programme doit être composé **d'au moins** deux fichiers `.c` (plus les éventuels `.h`). Un fichier contenant le `main` et un ou plusieurs fichiers contenant le reste du TP ;
3. chaque fichier **doit** compiler avec les options `-Wall -Wextra -Werror`.

Si une de ces condition n'est pas vérifiée, le TP ne sera pas validé par votre enseignant.

Vous devez écrire une commande :

```
modif_bmp [options] input output
```

qui prend en entrée un fichier au format `bmp` et qui génère en sortie un fichier `bmp` aux mêmes caractéristiques (codage de l'image, etc…) qui représente l'image originale après divers traitements qui seront appliqués en fonction des options de la ligne de commande. Vous trouverez à la fin du sujet une description précise du format de fichier.

Indications. Un découpage très simple vous est proposé, vous pouvez bien entendu utiliser toute autre méthode qui vous paraît plus judicieuse.

Exercice 1 : Lecture de l'entête

Afin de retenir les informations stockées dans l'entête, nous allons utiliser les structures suivantes qui correspondent à la description du format de fichier BMP.

```
/* On déclare des types dont on est sûr de la taille.
   Si on doit implémenter le TP sur une architecture différente, il
   suffit de changer ces déclarations
*/
typedef unsigned short uint16;
typedef unsigned int uint32;

typedef struct
{
      uint16 signature;
      uint32 taille_fichier;
      uint32 reserve;
      uint32 offset_donnees;
} entete_fichier;

typedef struct
{
      uint32 taille_entete;
      uint32 largeur;
      uint32 hauteur;
      uint16 nombre_plans;
      uint16 profondeur;
      uint32 compression;
      uint32 taille_donnees_image;
      uint32 resolution_horizontale;
      uint32 resolution_verticale;
      uint32 taille_palette; /* en nombre de couleurs */
      uint32 nombre_de_couleurs_importantes; /* 0 */
} entete_bitmap;

typedef struct
{
      entete_fichier fichier;
      entete_bitmap bitmap;
} entete_bmp;
```

À cause des alignements des champs des structures en mémoire, nous ne pouvons pas faire une simple lecture de l'entête dans le fichier à l'aide de `read(desc,&bmp,sizeof(entete_bmp));`. Nous allons devoir lire (ou écrire) les champs de la structure un à un [1].

Q 1. Écrire une fonction

```
int lire_deux_octets(int fd, uint16 *val)
```

qui lit deux octets dans le fichier dont le descripteur est donné en paramètre et retourne le nombre d'octets effectivement lus, -1 en cas d'erreur et 0 si le fichier est terminé.

Q 2. Écrire une fonction

```
int lire_quatre_octets(int fd, uint32 *val)
```

qui lit quatre octets dans le fichier dont le descripteur est donné en paramètre et retourne le nombre d'octets effectivement lus, -1 en cas d'erreur et 0 si le fichier est terminé.

Q 3. Écrire une fonction

```
int lire_entete(int de, entete_bmp *entete);
```

qui lit l'entête du fichier en entrée et stocke les informations dans la structure passée en paramètre en se servant des fonctions précédentes. La fonction doit retourner -1 si l'entête n'a pas pu être lue correctement.

La commande `~hauspiem/public/m3101/bmp` vous permet d'afficher les valeurs des champs contenus dans un fichier bmp afin de vérifier votre lecture.

Exercice 2 : Écriture de l'entête

Q 1. Écrire une fonction

```
int ecrire_deux_octets(int fd, uint16 val)
```

qui écrit deux octets dans le fichier dont le descripteur est donné en paramètre et retourne le nombre d'octets effectivement écrits ou -1 si une erreur est survenue.

Q 2. Écrire une fonction

```
int ecrire_quatre_octets(int fd, uint32 val)
```

qui écrit quatre octets dans le fichier dont le descripteur est donné en paramètre et retourne le nombre d'octets effectivement écrits ou -1 si une erreur est survenue.

Q 3. Écrire une fonction

```
int ecrire_entete(int vers, entete_bmp *entete);
```

qui écrit l'entete passée en paramètre dans le fichier de destination en se servant des fonctions précédentes. La fonction doit retourner -1 si l'entête n'a pas pu être lue correctement.

Exercice 3 : Limitation de notre commande

Pour ce TP, nous n'allons lire que des fichiers BMP dont la profondeur d'image est 24 bits, c'est à dire dont chaque pixel est codé sur 3 octets, 1 octet par composante de couleur (bleu, vert, rouge).

Écrire une fonction

```
int verifier_entete(entete_bmp *entete);
```

qui vérifie que le fichier bmp chargé précédemment est bien une image ayant une profondeur de 24 bits et retourne 1 si c'est le cas et 0 sinon.

Quand vous écrirez votre commande, si cette fonction retourne 0, la commande quittera en affichant un message d'erreur.

Exercice 4 : Allocation de l'espace mémoire nécessaire à la lecture des pixels

Écrire une fonction

```
unsigned char* allouer_pixels(entete_bmp *entete);
```

qui retourne un tableau suffisamment grand pour contenir tous les pixels de l'image décrite par l'entête passée en paramètre.

1. On pourrait forcer le compilateur à ne pas ajouter d'octets de « bourrage », mais c'est en général une mauvaise idée si l'on souhaite écrire du code portable. De plus, cela ne réglerait pas les problèmes d'endianess.

Exercice 5 : Lecture des pixels

Écrire une fonction

```
int lire_pixels(int de, entete_bmp *entete, unsigned char *pixels);
```

qui copie les pixels contenus dans le fichier `de` dans le tableau `pixels` qui aura été alloué au préalable. Le paramètre `entete` permet de savoir combien de données il faut lire dans le fichier. **Attention**, les pixels de l'image ne sont pas nécessairement stockés immédiatement après l'entête. Observez bien le rôle du champ `offset_donnees` et utilisez la fonction `lseek` pour le prendre en compte.

Exercice 6 : Ecriture des pixels

Écrire une fonction

```
int ecrire_pixels(int vers, entete_bmp *entete, unsigned char *pixels);
```

qui écrit les pixels contenus dans le tableau `pixels` dans le fichier `vers`. Le paramètre `entete` permet de savoir quelle est la taille du tableau `pixels`. De même que pour la question précédente, vous utiliserez la fonction `lseek` pour prendre en compte le champ `offset_donnees` de l'entête.

Exercice 7 : Premier test, copie d'un fichier bmp

Pour tester toutes les fonctions précédentes, écrivez la fonction suivante. **Attention** : ici, aucun test n'est fait pour maintenir le sujet le plus clair possible, pensez à **TOUJOURS** effectuer tous les tests nécessaires de façon à ce que votre programme ne plante pas...

```
int copier_bmp(int de, int vers)
{
  entete_bmp entete;
  unsigned char *pixels;

  /* lecture du fichier source */
  lire_entete(de, &entete);
  pixels = allouer_pixels(&entete);
  lire_pixels(de, &entete, pixels);

  /* écriture du fichier destination */
  ecrire_entete(vers, &entete);
  ecrire_pixels(vers, &entete, pixels);

  /* on libère les pixels */
  free(pixels);
  return 1; /* on a réussi */
}
```

Cette fonction suppose bien évidemment que les fichiers source et destination ont été ouverts correctement (`de` et `vers` sont les descripteurs de ces fichiers).

Récupérez le fichier `~hauspiem/public/m3101/test24.bmp` et essayez d'en créer une copie à l'aide de votre programme.

Exercice 8 : Premier filtre

Écrire une fonction

```
void rouge(entete_bmp *entete, unsigned char *pixels);
```

qui ne conserve **que** la composante rouge de l'image. N'oubliez pas que l'image est une succession de pixels et qu'un pixel est constitué de trois octets, un pour le bleu, un pour le vert et un pour le rouge (dans cet ordre). Pensez également au problème de la taille des lignes (en octets) qui doit être un multiple de 4 et de la notion de *padding* (bourrage) qui en découle.

Exercice 9 : Le main et les options

Écrire la fonction `main` de façon à ce que votre commande respecte le format d'appel suivant :

```
modif_bmp [options] input output
```

Votre programme devra supporter (pour l'instant) l'option suivante :

— `-r` : applique un filtre rouge sur l'image

Le comportement de votre programme doit être de copier l'image `input` vers l'image `output` en appliquant, si l'option est présente, le filtre indiqué dans les options.

Exercice 10 : Filtres supplémentaires

Écrire une fonction

```
void negatif(entete_bmp *entete, unsigned char *pixels);
```

qui effectue le négatif de l'image. Pour effectuer le négatif d'une image, il suffit de remplacer chacun de ses pixels par leur complément à un [2]

Écrire une fonction

```
void noir_et_blanc(entete_bmp *entete, unsigned char *pixels);
```

qui passe les pixels de l'image en noir et blanc [3]

Écrire une fonction

```
void moitie(entete_bmp *entete, unsigned char *pixel, int sup);
```

qui ne conserve que la moitié supérieure de l'image si `sup` vaut 1 et la moitiée inférieure sinon. Attention, il faudra sûrement modifier l'entete passée en paramètre cette fois.

Exercice 11 : Le programme complet

Modifiez votre programme pour qu'il accepte maintenant les options suivantes :

—

— `-n` : passe l'image en négatif

— `-b` : passe l'image en noir et blanc

— `-s` : récupère uniquement la moitié supérieure de l'image

— `-i` : récupère la moitiée inférieure de l'image

Évidemment, les modifications seront appliquées à l'image sauvegardée. Si plusieurs options sont passées, les traitements doivent être effectués dans l'ordre (par exemple `-n -b` écrit une image passée en noir et blanc puis en négatif). Si aucune option n'est donnée, vous devez toujours faire une copie simple.

Le script `~hauspiem/public/m3101/test_bmp.sh` vous permet de tester votre programme.

2. Pour rappel, le complément à un est le nombre entier que l'on obtient en remplacant les 0 par des 1 et les 1 par des 0 dans la représentation binaire. L'opérateur permettant ceci en C est l'opérateur~

3. Pour passer les pixels d'une image en noir et blanc, il suffit d'y réfléchir très fort, au bout d'un moment on trouve... :-)

Le format

Entête du fichier

L'entête du fichier fournit des informations sur le type de fichier (Bitmap), sa taille et indique où commencent les informations concernant l'image à proprement parler.

L'entête du fichier est codée sur 18 octets :

— 2 octects de signature indiquent la nature du fichier, nous considérerons que le fichier utilisé est correct ;
— 4 octets codant la taille totale du fichier ;
— 4 octets pour un champs réservé ;
— 4 octets pour l'offset de l'image.

Entête de l'image

L'entête de l'image indique les caractéristiques de l'image :

— 4 octets pour la taille de l'entête de l'image ;
— 4 octets pour la largeur de l'image (en nombre de pixels) ;
— 4 octets pour la hauteur de l'image (en nombre de pixels) ;
— 2 octets pour le nombre de plans (toujours 1) ;
— 2 octets pour la profondeur de codage de la couleur (i.e. 1, 4, 8, 16, 24 ou 32 bits par pixel) ;
— la méthode de compression (sur 4 octets). Cette valeur vaut 0 lorsque l'image n'est pas compressée, ou bien 1, 2 ou 3 suivant le type de compression utilisé ;
— la taille totale de l'image en octets (sur 4 octets) ;
— la résolution horizontale (sur 4 octets), c'est-à-dire le nombre de pixels par mètre horizontalement ;
— la résolution verticale (sur 4 octets), c'est-à-dire le nombre de pixels par mètre verticalement ;
— le nombre de couleurs de la palette (sur 4 octets). Si l'image est palettisée et que ce nombre vaut 0 alors le nombre de couleurs est $2^{profondeur}$;
— le nombre de couleurs importantes de la palette (sur 4 octets). Ce champ peut être égal à 0 lorsque chaque couleur a son importance.

La palette de l'image optionnelle sur 4 octets.

Codage de l'image

— Le codage de l'image se fait en écrivant successivement les bits correspondant à chaque pixel, ligne par ligne en commençant par le pixel en bas à gauche.
— Les images en 2 couleurs utilisent 1 bit par pixel, ce qui signifie qu'un octet permet de coder 8 pixels.
— Les images en 16 couleurs utilisent 4 bits par pixel, ce qui signifie qu'un octet permet de coder 2 pixels.
— Les images en 256 couleurs utilisent 8 bits par pixel, ce qui signifie qu'un octet code chaque pixel.
— Les images en couleurs réelles utilisent 24 bits par pixel, ce qui signifie qu'il faut 3 octets pour coder chaque pixel, en prenant soin de respecter l'ordre de l'alternance bleu, vert et rouge.
— Chaque ligne de l'image doit comporter un nombre total d'octets qui soit un multiple de 4 ; si ce n'est pas le cas, la ligne doit être complétée par des 0 de telle manière à respecter ce critère.