IF112 - Projet d'Informatique

2023 - 2024

Yannick Bornat - yannick.bornat@enseirb-matmeca.fr
Guillaume Bourmaud - guillaume.bourmaud@enseirb-matmeca.fr
Clémence Gillet - clemence.gillet@enseirb-matmeca.fr
Rémi Giraud - remi.giraud@enseirb-matmeca.fr

TP4 : Utilisation de *make*
Compilations séparée

Ce TP est le dernier de la série avant le projet final. Il reprend les parties précédentes et permet également d'aborder la compilation multi-fichiers.

La compilation séparée repose sur la répartition des fonctions sur plusieurs fichiers .c, chacun d'entre eux étant compilé séparément. C'est le principe utilisé pour fournir une bibliothèque de fonctions (telles `stdlib`, `stdio` ou `math`). Dans le texte de ce TP, nous appellerons *utilisateur* le programmeur qui fait appel aux fonctions des bibliothèques depuis son propre programme.

1 Séparation des fonctions

1.1 Écriture des fichiers c

Créez deux fichiers *bitmaplib.c* et *vectlib.c* qui contiendront respectivement les fonctions relatives aux image bitmap et aux descriptions vectorielles. Voici le contenu attendu pour les deux fichiers. Si, dans votre avancée des TPs précédents, vous n'avez pas eu le temps de finir une fonction, ne la créez pas dans votre fichier, cela vous évitera de perdre du temps.

bitmaplib.c devrait contenir les fonctions suivantes :
- `new_pic()` (création d'un `struct picture`)
- `save_pic()` (enregistrement d'un `struct picture` dans un fichier)
- `set_pixel()` (définir la couleur d'un pixel dans un `struct picture`)
- `draw_line()` (tracé d'une ligne dans un `struct picture`)

vectlib.c doit contenir les fonctions suivantes :
- `read_vector_file()` (ouverture d'un fichier vectoriel)
- `draw_vector()` (application d'une image vectorielle sur une image bitmap)
- `scale_vector()` (changement d'échelle d'une image vectorielle)
- `shift_vector()` (translation d'une image vectorielle)
- `flip_vector()` (miroir sur une image vectorielle)

fonctions facultatives de vectlib.c :
- `duplicate_vector()` (duplication d'une image vectorielle)
- `free_vector()` (destruction d'une image vectorielle)
- `homothety_vector()` (mise à l'échelle selon un point de référence)
- `rotate_vector()` (rotation d'une image vectorielle)

La fonction `draw_vector()` nécessite que *vectlib.c* fasse appel à une fonction de *bitmaplib.c*.

1.2 Écriture des fichiers d'en-tête (.h)

Pour rappel, les fichiers d'en-tête sont les fichiers qui contiennent l'ensemble des définitions du fichier .c auquel ils correspondent. Par convention, ils portent l'extension .h. Pour les utiliser, il suffit de les inclure grâce à la commande `#include "nom_du_fichier.h"`.

Ils contiennent (au moins) :
- la définition des types de donnée
- le prototype de chaque fonction accessible par l'utilisateur
- la déclaration des constantes nécessaires à l'utilisation des fonctions

Si certaines fonctions sont réservées pour un usage local, il est possible (voire conseillé) de ne pas les mettre dans le fichier d'en-tête.

Exemple :

La fonction `scale_vector()` est plus simple à écrire que la fonction `homothety_vector()` car elle se place dans le cas d'une homothétie par rapport à l'origine. Mais elle est plus difficile à utiliser, elle présente donc peu d'intérêt pour un utilisateur. Il est possible de construire la fonction `homothety_vector()` comme suit :

```
void homothety_vector(struct vector* l, double x, double y, double scale) {
  shift_vector(l, -x, -y);
  scale_vector(l, scale);
  shift_vector(l, x, y);
}
```

Dans ce cas, `scale_vector()` est une sous-fonction de `homothety_vector()`, elle est donc nécessaire dans le fichier .c, mais la proposer à l'utilisateur final ne présente pas d'intérêt, il est donc possible de ne pas la mettre dans le fichier d'en-tête, elle ne sera pas visible. Cela permet de réduire le nombre de fonctions visibles, et de simplifier la prise en main de l'ensemble de fonctions.

Écrivez les fichiers d'en-tête *bitmaplib.h* et *vectlib.h*. Étant donné que la définition des types de donnée est dans le fichier d'en-tête, chaque fichier .c devra utiliser son propre fichier d'en-tête.

1.3 Le piège des inclusions multiples

Le fichier *vectlib.h* contient le prototype de la fonction `draw_vector()` qui reçoit un `struct picture` comme argument. Comme `struct picture` est défini dans *bitmaplib.h*, *vectlib.h* doit inclure *bitmaplib.h*. Le problème survient quand, dans le fichier c principal, l'utilisateur inclue à la fois *bitmaplib.h* et *vectlib.h*. Dans ce cas, le fichier *bitmaplib.h* est inclus deux fois (une fois directement, et une autre fois par l'intermédiaire de *vectlib.h*). Le compilateur n'est pas capable d'identifier ce double appel, et génère une erreur car il remarque plusieurs définitions pour un même nom.

La solution consiste à protéger les définitions à l'aide des commandes de préprocesseur `#ifndef` et `#endif`. En effet, tout le code inclus entre `#ifndef TOTO` et le `#endif` qui lui correspond sera ignoré si `TOTO` a déjà été défini par un `#define`. La structure suivante permet donc de n'inclure qu'une seule fois le code `blabla`, quel que soit le nombre de fois ou elle apparaît :

```
#ifndef FOO
#define FOO
blabla;
#endif
```

Appliquez cette technique pour protéger vos fichiers d'entête des inclusions multiples.

2 Première compilation et test

2.1 Écriture du programme de démonstration

Créez un programme qui crée une image bitmap de taille 256 x 256 qui contient un dégradé de couleurs de vert (0, 255, 0) en haut vers la couleur cyan (0, 255, 255) en bas. Tracez la figure vectorielle du fichier *mouse.txt* sur cette image en utilisant la couleur noir. Ce programme doit être très court et faire appel aux bilbiothèques `bitmaplib` et `vectlib`.

2.2 Compilation

La compilation se fait en plusieurs étapes, nous supposerons pour notre exemple que le fichier principal est main.c :

```
gcc -std=c99 -c bitmaplib.c                    // compile bitmaplib.c, le résultat est bitmaplib.o
gcc -std=c99 -c vectlib.c                      // compile vectlib.c, le résultat est vectlib.o
gcc -std=c99 -c main.c                         // compile main.c, le résultat est main.o
gcc -o programme main.o bitmaplib.o vectlib.o  // fait l'édition de lien et produit le fichier final
```

Une fois la première compilation complète effectuée, les seules étapes nécessaires sont la (re)compilation des fichiers modifiés et l'édition de lien (l'étape finale).

2.3 Exécution

L'exécution du programme ne change pas, il s'agit simplement de vérifier que le résultat final est fonctionnel. Si le programme n'est pas fonctionnel, pour chaque nouvelle compilation, il faut d'abord recompiler toutes les parties du programme qui utilisent un fichier modifié :

- si c'est un .c, une seule compilation est nécessaire avant l'édition de liens
- si c'est un .h, tous les fichiers .c qui y font appel doivent être recompilés.

C'est pénible hein ?
La partie suivante va vous alléger le travail...

3 Automatisation de la compilation : *make*

L'outil *make*, est commun à la plupart des environnements de développement UNIX. Il permet de vérifier si des fichiers doivent être recompilés en comparant les dates de modification des fichiers sources et des fichiers compilés. Lorsque *make* est correctement configuré, la compilation devient beaucoup plus immédiate.

3.1 Configurer *make*

La seule configuration nécessaire à *make* est l'écriture d'un fichier dont le nom est *Makefile*. Il s'agit d'un fichier texte qui code, avec un certain format, quels fichiers sont liés pour la compilation. Voici un exemple de fichier pour un projet contenant trois fichiers .c (main.c, calcul.c et calcul.h) :

```
objects = main.o calcul.o

program :  $(objects)
    gcc -Wall -o program $(objects)
    ./program

main.o :  main.c calcul.h
    gcc -Wall -c main.c

calcul.o :  calcul.c calcul.h
    gcc -Wall -c calcul.c

clean :
    rm $(objects)
```

La première ligne contient la liste des fichiers objets (fichiers de compilation intermédiaires) à créer. Cette liste sera utile pour plusieurs lignes suivantes. Les parties écrites en rouge sont appelées les règles. Leur nom est généralement un nom de fichier. La partie en bleu est la recette de la règle. Les parties en vert sont les prérequis spécifiques à chaque règle, ce sont également des noms de fichiers.

Pour plus de détails sur la fonctionnement de make et l'automatisation du Makefile[1].

Le principe de fonctionnement :
- $(objects) est remplacé par le contenu de la variable objects définie en première ligne
- On appelle make en spécifiant une règle particulière.
- Si un prérequis correspond à un nom de règle, la règle correspondant sera d'abord appelée.
- Si un fichier prérequis a été modifié après le fichier règle, alors la recette est exécutée.
- Si aucune règle n'est spécifiée, la première règle est utilisée.

Selon ces principes, en exécutant la commande `make` en ligne de commande :
1. la règle `program` est appelée (règle par défaut)
2. la règle `main.o` est appelée
3. si *main.c* ou *calcul.h* ont été modifiés, *main.c* est recompilé
4. la règle `calcul.o` est appelée
5. si *calcul.c* ou *calcul.h* ont été modifiés, *calcul.c* est recompilé
6. (retour à la règle `main.o`) : si un des trois fichiers .o a été modifié (par une des règles précédentes, notamment), l'édition de lien est refaite et une nouvelle version de *program* est créée.

La dernière règle (`clean`) est particulière, elle n'a pas de prérequis. Si elle est appelée, sa recette sera toujours exécutée. Elle sert à nettoyer les fichiers intermédiaires si nécessaire. Pour l'appeler, il sufit d'exécuter la commande `make clean`.

Adaptez le fichier *Makefile* pour notre contexte avec les fichiers *main.c, bitmaplib.c, vectlib.c, bitmaplib.h, vectlib.h.*

3.2 Utilisation

Maintenant que *make* est opérationnel, il est beaucoup plus pratique de compiler, même si on ne se souvient pas des fichiers qui ont été modifiés. Il suffit pour cela d'exécuter `make`.

Enrichissez la bibliothèque `bitmaplib` d'une fonction `read_pic()` qui reçoit un nom de fichier en argument, et qui renvoie le `struct picture` qui correspond à l'image contenue dans ce fichier.

Pour cela, il vous faudra modifier les fichiers *bitmaplib.c*, *bitmaplib.h* et *main.c* (pour le test de votre fonction). Vous testerez votre fonction en créant une image qui superpose le fichier vectoriel *mouse.txt* au fichier bitmap *cheese.ppm.*

4 Enrichissement des bibliothèques

4.1 Où placer une nouvelle fonction ?

Si vous venez de créer une nouvelle fonction, il y a de grandes chances qu'elle soit spécifique à votre programme, ou à votre usage. Il est donc peu probable qu'elle aie sa place dans la bibliothèque de base. Mais certaines fonctions peuvent se révéler utiles, et le concepteur de la bibliothèque n'a simplement pas eu le temps/l'idée des les implémenter. Avant d'ajouter une fonction dans une bibliothèque, assurez-vous donc que cette nouvelle fonction est d'usage suffisament général pour y avoir sa place. Sinon, la fonction doit rester dans les parties spécifiques à votre programme.

Si une grosse partie de votre programme se base sur des fonctions qui ont des intérêts proches, il vous appartient de construire votre propre bibliothèque.

[1] `https://remi-giraud.enseirb-matmeca.fr/teaching/course.php?course=IF112`

Exemples :
- Ajouter des constantes qui correspondent aux couleurs de base dans la bibliothèque `bitmaplib`, ça a un sens.
- Ajouter des fonctions de représentation 3D dans `bitmaplib`, c'est beaucoup moins cohérent, autant construire une bibliothèque secondaire (qui éventuellement, se basera sur `bitmaplib` pour la représentation finale des scènes 3D)

4.2 Liste de fonctions manquantes dans `bitmaplib` et `vectlib`...

Voici une liste de fonctions qui seraient très utiles dans les bibliothèques que nous avons créées, mais que nous n'avons pas abordé car elles ne produisaient pas d'intérêt pédagogique particulier (par rapport au temps d'écriture nécessaire).

Pour `bitmaplib` :
- Agrandissement d'une image
- Rotation d'une image
- Fusion d'images (la fonction reçoit trois images, im1, im2 et mask, cette dernière est en niveau de gris. L'image renvoyée est composée de im1 là ou mask est blanc, de im2 là ou mask est noire et d'une moyenne des deux images pour les pixels ou mask est gris. La proportion des couleurs est pondérée par la noirceur du gris)

Pour `vectlib` :
- enregistrement d'un tracé dans un fichier
- Remplissage d'une zone de couleur par une autre couleur
- `Vectlib_col`, un variant de `vectlib` ou chaque ligne est associée à sa propre couleur
- Fonctions de conversion de données entre `vectlib` et `vectlib_col`

Implémentez la/les fonctions de votre choix.