Programmation modulaire en C, Makefile

Rappel préliminaire : l'excellent polycopié sur le langage C de H. Garetta est disponible à l'url `http://www.dil.univ-mrs.fr/~garreta/Polys/PolyC.pdf`

1 Introduction

Lorsque l'on écrit un programme de plusieurs milliers de lignes (ce qui peut éventuellement être le cas pour ce projet de détection de visages dans des images), il est évidemment hors de question que ce programme soit entièrement contenu dans un grand, illisible et unique fichier source (**hors de question !**). Une démarche naturelle est de découper le projet en plusieurs parties ou *modules*, chacun de ces modules étant associé à un ensemble de fonctionnalités déterminées. Dans une approche comme celle-ci, deux questions viennent notamment se poser : 1. comment définir les modules ?, 2. comment compiler efficacement les modules – généralement associés à un ou plusieurs fichiers – pour en faire un programme exécutable ?

La réponse à la première question n'est pas évidente et, pour une application donnée il n'y a que rarement un seul et unique découpage possible ; en revanche, dans le cas d'un application graphique telle que celle qui nous intéresse, il est possible de faire reposer le découpage retenu sur une architecture bien connue sous le nom de « Modèle-Vue-Contrôleur » (MVC), décrite brièvement ci-dessous.

Concernant la seconde question, la réponse repose sur l'utilisation de Makefiles, qui permettent de définir la « recette » de création d'un exécutable : les instructions codées dans un Makefile définissent les modules (et les fichiers qui les constituent), i.e. les ingrédients, dont l'exécutable a besoin pour être créé, et la manière de les utiliser et de les combiner, i.e. de les cuisiner, pour obtenir ledit exécutable. On notera que les notions de fichiers objets (fichiers `.o`), fichiers en-tête (fichiers `.h`), librairies (fichiers `.a` et `.so`) et éditions de liens sont les élément de base de ce paradigme de *compilation séparée*.

Dans ce document nous donnons quelques principes de base de programmation modulaire, avec un découpage en modules qui s'applique particulièrement bien aux programmes proposant une interface graphique.

2 Un découpage modulaire reposant sur l'architecture MVC

La programmation modulaire consiste, comme son nom l'indique à découper un projet informatique en plusieurs modules qui se définissent par les fonctionnalités qu'ils offrent. Dans le cadre de notre projet, il est par exemple facile d'identifier un découpage naturel en trois modules : le module de reconnaissance proprement dit, le module d'affichage graphique et le module de gestion des interactions avec l'utilisateur (clics de souris, notamment). Dans le jargon MVC, le premier module correspond à ce qui est appelé le *modèle*, le second à la *vue* et le dernier au *contrôleur*, pour des raisons que les noms justifient d'ores et déjà un peu mais sur lesquelles on reviendra ultérieurement.

2.1 Fichiers en-tête (header)

Avec un minimum d'expérience en C, on sait très vite qu'il y a souvent des instructions qui se placent en tête d'un programme et qui prennent la forme suivante :

```
#include <stdio.h>
#include <math.h>
```

Ces instructions sont des directives qui précisent au compilateur, qui va s'occuper de transformer le programme en langage C en langage machine (comprenez en langage binaire), que certaines des fonctions que vous allez utiliser n'ont pas été écrites par vous-même mais d'autres programmeurs et que les signatures de ces fonctions se trouvent dans les fichiers stdio.h et math.h. (La signature d'une fonction

est composée de son nom, ou identifiant, la liste et types de paramètres qu'il est nécessaire de lui fournir pour l'appeler et son type de retour.) De façon simplifiée, ces instructions disent donc au compilateur de ne pas s'étonner de ne pas trouver explicitement la déclaration et la définition de certaines fonctions que vous utilisez dans votre programme mais que ces fonctions existent par ailleurs et qu'elles sont bien programmées. Les headers sont des « contrats » qui assurent aux programmeurs qui y ont recours que les fonctions proposées sont réellement implémentées et font précisément ce qu'elle promettent de faire. Les deux directives **#include** proposées ci-dessus permettent l'utilisation de routines d'entrée/sortie (affichage d'un texte sur l'écran, lecture d'un texte au clavier, lecture/écriture dans un fichier) et l'utilisation de routines simples de mathématique (calcul de logarithmes, fonctions trigonométriques, etc).

Les headers constituent une partie des modules (dans une vision simplifiée des choses, on peut considéré que stdio.h fait partie du module d'entrée/sortie du langage C et math.h fait partie du module de mathématiques) et un des objectifs du projet est que chacun écrive ses propres headers.

Un fichier en-tête est simplement un fichier texte dont la structure est donnée ci-après. Comme il se doit, vous écrirez des fichiers en-tête qui correspondront à chacun des modules fonctionnels que vous aurez identifiés. Chaque fichier contiendra les signatures de fonctions que vous programmerez et, éventuellement la déclaration des types de données (les **struct**). Par exemple, si l'on veut faire un module pour le traitement proprement dit du tableau de pixels d'une image, on créera un fichier en-tête pixbuf.h, du type suivant (attention, ici pixbuf est un nom de fichier que vous avez choisi, il pourrait s'appeler tableaupixels et n'est pas directement relié au type GdkPixbuf de Gdk) :

```
#ifndef __PIXBUF__
#define __PIXBUF__

/* fonction qui calcule l'histogramme des couleurs
   d'une image, renvoie 0 si la fonction echoue (par
   exemple par ce que le pixbuf passe en parametre
   est NULL et 1 sinon. Le resultat est stocke dans le
   tableau histo
*/
int histogramme(GdkPixbuf *pixbuf, int *histo);

/* fonction qui transforme le tableau de pixels
   d'une image de telle sorte que celle-ci soit
   en niveaux de gris. Renvoie 0 si echec et 1 sinon.
   Le tableau de pixels est change en place.
*/
int gris(GdkPixbuf *pixbuf);

/* fonction qui dit si la portion du tableau de pixels
   correspondant a une certaine partie d'une image contient
   un visage ou non. renvoie 1 si elle contient un visage
   et 0 sinon.
*/
int visage(GdkPixbuf *pixbuf, int x, int y, int width, int height);

#endif
```

qui spécifie le type de fonctionnalités qui seront offertes par le module de traitement de pixels (en anticipant un petit peu, ce module correspond précisément au modèle, dans la terminologie MVC). Encore une fois, un header est une sorte de contrat qui précise quelles sont les fonctions qui peuvent être utilisées et ce qu'elles font. Pour que ces fonctions puissent être utilisées dans un fichier toto.c, il est nécessaire que ce fichier ait une ligne du type **#include** "pixbuf.h". Par conséquent, une première phase du projet doit être un processus de réflexion quant aux différentes fonctions que vous devrez écrire. Une fois listées et catégorisées selon le module dont elles sont censées faire partie, elles sont rangées par fichier en-tête (nous considérerons dans une première phase du projet qu'il y aura un fichier en-tête par

module).

Comme n'étant que des contrats, ces fichiers header ne font rien que signifier qu'il est possible d'utiliser les fonctions qui y sont déclarées. Mais le vrai travail est de programmer ou définir les fonctions en tant que telles, c'est-à-dire d'écrire le code qui permet d'assurer que les fonctions font effectivement ce qu'elles promettent de réaliser. A chaque fichier en-tête on associe par conséquent un (ou éventuellement plusieurs) fichier(s) qui implémentent les fonctions. Dans ce projet, on écrira le code qui correspond au fichier titi.h dans le fichier titi.c. Et donc, dans le cas du fichier pixbuf.h, on créera un fichier pixbuf.c qui contiendra par exemple les lignes suivantes :

```
#include "pixbuf.h"

int histogramme(GdkPixbuf *pixbuf, int *histo){
  /* Boucle qui parcourt l'ensemble des pixels du
     tableau et compte les différentes intensités
     des canaux RVB et les stocke dans histor
  */
}

int gris(GdkPixbuf *pixbuf){
  /* Boucle qui fait une transformation de l'image
     en niveaux de gris
  */
}

int visage(GdkPixbuf *pixbuf, int x, int y, int width, int height){
  /* Traitement plus ou moins sophistiqué qui détermine
     si le tableau de pixels correspond à une image
     de visage.
  */
}
```

Un des intérêts de la séparation du programme en plusieurs modules dans le cadre d'un travail en équipe est qu'une fois que les participants d'un projet ont décidé des fonctions qu'ils devaient programmer et qu'ils les ont réparties en modules (pour nous, un module correspond à un fichier en-tête .h et son fichier .c), la charge d'un module particulier peut être dévolue à un groupe particulier qui, sachant exactement ce que doivent faire les fonctions qu'il doit programmer, peut travailler (i.e. programmer son module) indépendamment de ce qui se passe au sein des autres modules. Le projet est ainsi découpé en plusieurs parties qui peuvent avancer de façon (presque) indépendante et donc parallèlement. Un deuxième avantage de la programmation modulaire a trait à la possibilité de réutiliser le code programmé dans plusieurs applications. Ainsi, il est possible d'utiliser le module mathématique du langage C dans plusieurs programmes ; de plus il n'est pas nécessaire lorsqu'on ne veut utiliser que ce module d'utiliser les modules d'entrée/sortie. La disponibilité de module permet donc de faire un programme « à la carte », où seuls les modules contenant les fonctionnalités dont on a besoin nécessitent d'être utilisés.

Cette section décrit brièvement comment mettre en œuvre la modularité en C. En d'autres termes, cette section explique comment créer des boîtes à outils dans lesquelles on peut piocher des fonctions d'intérêt. La section suivante donne des liens sur la façon dont on peut exploiter ces modules et les intégrer dans un programme proprement dit.

2.2 Compilation séparée et Makefiles

Une fois les modules programmés, il est nécessaire de disposer d'une procédure pour pouvoir les utiliser dans le programme principal, que nous appellerons detection.c. Pour cela, il est évidemment nécessaire d'insérer les **#include** idoines au début de detection.c de telle sorte que son contenu ressemble à (les fichiers en-tête control.h view.h sont décrits un peu plus loin) :

```
#include "pixbuf.h"
#include "control.h"
#include "view.h"
#include <gdk-pixbuf/gdk-pixbuf.h>
#include <gtk/gtk.h>

int main(int argc, char *argv[]){
  ... /* programme principal                    */
  ... /* qui fait appel aux fonctions déclarées */
  ... /* dans les fichiers en-tête              */
  return 1;
}
```

Mais demeure la question de la compilation de ce projet en un seul programme à partir des multiples fichiers .c et .h. L'intérêt du fichier Makefile et de l'utilitaire make est précisément d'aider à cette tâche. En particulier, ces deux outils (bien que le seul programme soit en fait make, mais puisqu'il fonctionne avec un fichier qui s'appelle généralement Makefile, ce dernier est également considéré comme un outil), permettent de ne compiler que les parties qui l'exigent : si un module a été modifié depuis la dernière compilation complète du projet, alors make et le Makefile associé travailleront de concert pour ne recompiler que le module modifié et en intégrer les changements au programme principal.

Un Makefile élémentaire pour le programme qui nous intéresse peut être le suivant :

```
# Spécification du compilateur
CC=gcc

# Options de compilation
CFLAGS=`pkg-config --cflags gtk+-2.0` -Wall
LDFLAGS=`pkg-config --libs gtk+-2.0`

all: detection

detection: detection.o pixbuf.o control.o view.o
	$(CC) $(LDFLAGS) detection.o pixbuf.o view.o -o detection

detection.o: detection.c
	$(CC) -c $(CFLAGS) detection.c

pixbuf.o: pixbuf.c pixbuf.h
	$(CC) -c $(CFLAGS) pixbuf.c

control.o: control.c control.h
	$(CC) -c $(CFLAGS) control.c

view.o: view.c view.h
	$(CC) -c $(CFLAGS) view.c
```

La simple commande make all, pour peu que les commandes ci-dessus soient bien enregistrées dans un fichier nommé Makefile ou makefile permet d'effectuer la compilation du programme pour donner l'exécutable detection (on remarque que le Makefile proposé ici suppose que le main est dans le fichier detection.c). Si le fichier s'était appelé compilation, il aurait alors fallu le préciser à make en lançant la commande make -f compilation all.

Une explication de texte rapide sur les Makefile. Les lignes importantes sont celles de la forme :

```
cible : dependances
    commande  # ligne précédée par une tabulation !!!
```

où il est possible d'avoir plusieurs commandes (sur plusieurs lignes). La cible est généralement l'objet que l'on veut obtenir (sauf pour la cible all), les dépendances sont les fichiers nécessaires à l'obtention de cette

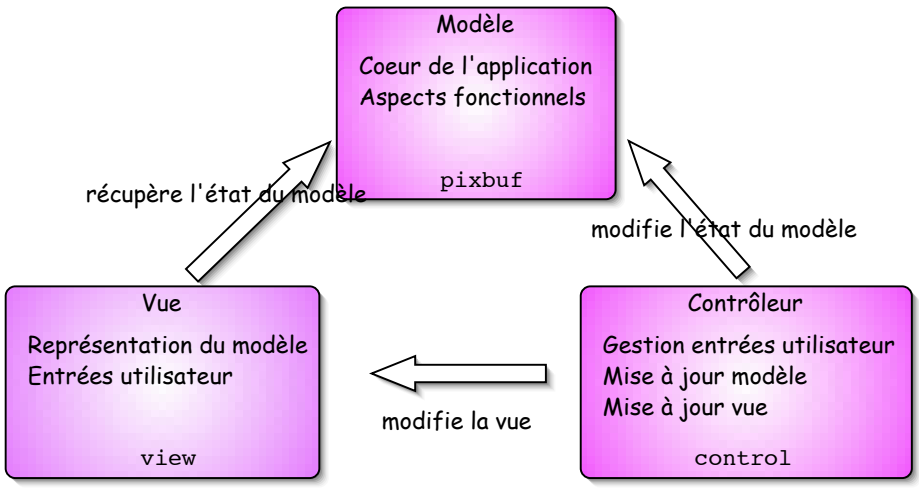
cible (les ingrédients) et la commande l'action à effectuer pour générer la cible à partir des dépendances (la méthode de cuisson). make permet de prendre en compte n'importe quelle cible et il serait possible de faire make pixbuf.o : si le fichier pixbuf.o existe déjà (comme provenant d'une compilation précédente) et que la date de dernière modification de pixbuf.o est postérieure à celle de ses dépendances alors make n'exécute pas les commande permettant de générer pixbuf.o et le fichier existant n'est pas modifié (la logique est que le fichier pixbuf.o est à jour par rapport aux fichiers qui permettent de le construire) ; inversement si les dépendances ont une date de modification ultérieure à celle de la cible alors celle-ci est recompilée (l'ancien pixbuf.o est alors remplacé par un nouveau pixbuf.o).

Notons que le Makefile proposé ici n'est pas correct si les différents fichiers en-tête sont amenés à changer et que des inclusions croisées de ces fichiers ont lieu au sein des fichiers .c. Pour corriger ce problème, il faudrait ajouter des dépendances de ces fichiers .c par rapport aux fichiers .h en question.

Pour ce projet, nous nous limiterons à l'écriture de fichiers Makefile assez simples et vous êtes donc invités à consulter le tutoriel très simple disponible à l'url suivante : `http://gl.developpez.com/tutoriel/outil/makefile/`. Ce document ne s'étendra pas plus outre sur l'écriture de Makefile.

2.3 Architecture MVC

L'architecture Modèle-Vue-Contrôleur est une architecture communément utilisée par les développeurs d'interfaces graphiques et/ou de boîtes à outils graphiques. Cette architecture se schématise de la façon suivante :



(On notera que le sens des flèches n'est pas nécessairement celui que l'on trouvera partout ailleurs sur internet ; ceci provient de la flexibilité de sémantique que l'on peut attacher à celles-ci.) L'idée de ce architecture de conception est de séparer une application graphique en trois entités (que l'on pourra faire correspondre aux modules) :

le modèle : il s'agit de la partie du programme qui représente l'état interne de l'application ; cette partie regroupe les données manipulées par le programmes ainsi que les méthodes pour les manipuler. Dans le cas du programme de détection de visage, l'objet principal à manipuler est un tableau de pixels, sur lequel et à partir duquel diverses opérations (des calculs pour les histogrammes, la détection de visages proprement dite, entre autres) seront effectuées. Pour simplifier le travail, on pourra considérer qu'une variable (on serait tenté d'appeler cela une instance, si l'on prend l'approche « objet » proposée par Gtk+) de type GdkPixbuf est un tableau de pixels.

la vue : c'est l'interface avec laquelle l'utilisateur accède à l'application ; typiquement, elle fournit une représentation (à l'écran) du modèle et généralement, on souhaite que cette représentation soit toujours synchronisée avec l'état courant du modèle. Dans notre cas, il s'agit simplement de

l'interface graphique qui représente l'image correspondant au tableau de pixels qu'est le modèle. Cette interface est sensbile aux actions de l'utilisateur (clics de souris, entrées d'information au clavier, etc).

le contrôleur : c'est lui, en quelque sorte, qui fait le lien entre la vue et le modèle. Il traite les interactions reçues par la vue provenant de l'utilisateur et transmet au modèle les opérations qui doivent être effectuées ; il synchronise la vue et le modèle. Dans notre application, le contrôleur est constitué des callbacks, c'est-à-dire des fonctions qui sont exécutées en réponse à des actions de l'utilisateur (il s'agit par exemple des fonctions `configure_event` et `expose_event` du petit programme `hello.c` fourni précédemment).

Une multitude de documents concernant cette architecture de développement est disponible sur le Web. Il est sage d'essayer de respecter au mieux le découpage en modules qui en résulte (ce qui n'ôte pas la possibilité de découper ces modules en sous-modules, si nécessaire).

3 Conclusion et travail à faire

Dans ce documents, les rudiments de la programmation modulaire et de son intérêt ont été présentés. En particulier, l'importance des `Makefiles` a été soulignée ainsi que la pertinence de l'architecture MVC.

Un exercice à faire s'inscrit dans la continuité de la phase de découverte de Gtk+ par le biais du fichier `hello.c` : il consiste à découper ce fichier suivant l'architecture MVC (en utilisant par exemple les noms de modules `pixbuf`, `view`, `control` proposés plus haut, et en renommant `hello.c` en `detection.c`), à ajouter les petites fonctionnalités de traitement d'images que vous aurez programmées et à définir un `Makefile` permettant la compilation des fichiers ainsi obtenus.