Cours Langage C/C++
Programmation modulaire

Thierry Vaira

BTS IRIS Avignon

tvaira@free.fr v0.1

Programmation modulaire (1/2)

- Le découpage d'un programme en sous-programmes est appelée **programmation modulaire**.
- La programmation modulaire se justifie par de multiples raisons :
  - un programme écrit d'un seul tenant devient très difficile à comprendre dès lors qu'il dépasse une page de texte
  - la programmation modulaire permet d'éviter des séquences d'instructions répétitives
  - la programmation modulaire permet le partage d'outils communs qu'il suffit d'avoir écris et mis au point une seule fois
  - des fonctions convenablement choisies permettent souvent de dissimuler les détails d'un calcul contenu dans certaines parties du programme qu'il n'est pas indispensable de connaître.
- D'une manière générale, la programmation modulaire clarifie l'ensemble d'un programme et facilite les modifications ultérieures.

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Programmation modulaire (2/2)

- Pour conclure, il faut distinguer :
  - la **déclaration** d'une fonction qui est une instruction fournissant au compilateur un certain nombre d'informations concernant une fonction. Il existe une forme recommandée dite **prototype** :

    `int plus(int, int) ;` ← fichier en-tête (.h)
  - sa **définition** qui revient à écrire le **corps** de la fonction (qui définit donc les traitements effectués dans le bloc {} de la fonction)

    `int plus(int a, int b) { return a + b ; }` ← fichier source (.c ou .cpp)
  - l'**appel** qui est son utilisation. Elle doit correspondre à la déclaration faite au compilateur qui vérifie.

    `int res = plus(2, 2) ;` ← fichier source (.c ou .cpp)
- *Remarque :* La définition d'une fonction tient lieu de déclaration. Mais elle doit être "connue" avant son utilisation (un appel). Sinon, le compilateur génère un message d'avertissement (warning) qui indiquera qu'il a lui-même fait une déclaration implicite de cette fonction : `"attention : implicit declaration of function 'multiplier'"`

Passage d'argument à une fonction

- Voici les différentes déclarations en fonction du contrôle d'accès désiré sur un paramètre reçu :
  - passage par valeur → accès en lecture seule à la variable passée en paramètre : `void foo(int a) ;`
  - passage par adresse → accès en lecture seule à la variable passée en paramètre : `void foo(const int *a) ;`
  - passage par adresse → accès en lecture et en écriture à la variable passée en paramètre : `void foo(int *a) ;`
  - passage par adresse → accès en lecture et en écriture à la variable passée en paramètre (sans modification de son adresse) : `void foo(int * const a) ;`
  - passage par adresse → accès en lecture seule à la variable passée en paramètre (sans modification de son adresse) : `void foo(const int * const a) ;`
  - passage par référence → accès en lecture et en écriture à la variable passée en paramètre : `void foo(int &a) ;`
  - passage par référence → accès en lecture seule à la variable passée en paramètre : `void foo(const int &a) ;`

Fichiers séparés

- La programmation modulaire entraîne la **compilation séparée**.
- Le programmeur exploitera la programmation modulaire en séparant ces fonctions dans des fichiers distincts. Généralement, il regroupera des fonctions d'un même "thème" dans un fichier séparé.
- C'est l'étape d'**édition de liens (*linker*)** qui aura pour rôle de regrouper toutes les fonctions utilisées dans un même exécutable.
- Regroupons les **définitions des fonctions** `multiplier()` et `plus()` dans un fichier `calcul.c` :

```c
int multiplier(int a, int b) {
   return a * b;
}

int plus(int a, int b) {
   return a + b;
}
```

Déclaration et définitions des fonctions

- Si un programme `main.c` désire utiliser une (ou plusieurs) des fonctions disponibles dans `calcul.c`, il doit accéder à leurs déclarations pour pouvoir se compiler.
- L'utilisation recommandée est de lui fournir un fichier contenant (seulement) les déclarations des fonctions.
- C'est le rôle des **fichiers en-tête (*header*)** qui portent l'extension `.h`. On aura donc un couple de fichiers : un `.h` (pour les **déclarations**) et un `.c` (pour les **définitions**) portant le même nom.
- Plaçons les déclarations des 2 fonctions précédentes dans un fichier `calcul.h` :

```
int multiplier(int a, int b);
int plus(int a, int b);
```

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

La directive include

- Maintenant, on peut écrire et compiler le programme `main.c` :

```
#include <stdio.h> /* pour la déclaration de printf */
#include "calcul.h" /* pour la déclaration de multiplier */
int main() {
   printf("2*2 = %d\n", multiplier(2, 2));
   return 0;
}
```

Remarque : `#include` est une **directive de pré-compilation** réalisée par le préprocesseur AVANT la compilation. Toutes les directives de pré-compilation commencent par un dièse '#'. La directive `include` a pour rôle d'inclure le fichier indiqué. On peut indiquer le fichier à inclure entre <> dans ce cas le fichier sera cherché dans les répertoires standards connus du compilateur (`/usr/include` sons GNU/Linux). Pour ses propres fichiers, on préfère indiquer son nom entre "". Dans ce cas, le fichier sera cherché dans le répertoire courant. On peut indiquer un chemin particulier en utilisant l'option `-Ichemin` de `gcc` (cf. `man gcc`).

Fabrication

Pour fabriquer un exécutable, il faut réaliser les étapes suivantes :

- compilation de `calcul.c` en utilisant l'option `-c` (compilation) et génèration d'un fichier objet (`calcul.o`)
- compilation de `main.c` en utilisant l'option `-c` (compilation) et génèration d'un fichier objet (`main.o`)
- édition de liens des fichiers objets (`.o`) en utilisant l'option `-o` (*output*) pour fabriquer un exécutable

Les étapes successives avec `gcc` :

```
$ gcc -c calcul.c
$ gcc -c main.c
$ gcc calcul.o main.o -o main
```

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Inclusion unique

Important : le compilateur n'"aime" pas inclure plusieurs fois les mêmes fichiers `.h` car cela peut entraîner des déclarations multiples.

- Pour se protéger des inclusions multiples, on utilise la structure suivante pour ses propres fichiers *header* :

```
#ifndef CALCUL_H // si CALCUL_H n'est pas défini
#define CALCUL_H // alors on le définit

int multiplier(int a, int b);
int plus(int a, int b);

#endif // fin si
```

Lors de l'inclusion suivante, le contenu ne sera pas inclus car `CALCUL_H` a déjà été défini et donc ses déclarations sont déjà connues. Il est aussi possible d'utiliser la directive `#pragma once`.

Makefile (1/2)

Les étapes peuvent être automatisées en utilisant l'outil `make` et en écrivant les règles de fabrication dans un fichier `Makefile`.

- Le fichier `Makefile` contient la description des opérations nécessaires pour générer une application. Pour faire simple, il contient les règles à appliquer lorsque les dépendances ne sont plus respectées.
- Le fichier `Makefile` sera lu et exécuté par la commande `make`.
- Une règle est une suite d'instructions qui seront exécutées pour construire une cible si la cible n'existe pas ou si des dépendances sont plus récentes.
- La syntaxe d'une règle est la suivante :

```
cible : dépendance(s)
<TAB>commande(s)
```

Makefile (2/2)

- Pour notre exemple, la structure du fichier `Makefile` sera :

```
main: main.o calcul.o
    gcc main.o calcul.o -o main

main.o: main.c
    gcc -c main.c

calcul.o: calcul.c
    gcc -c calcul.c
```

Pour fabriquer l'application, il suffira de faire : `make -f Makefile` (on lui dit de lire le fichier `Makefile`) ou plus simplement `make` (car par défaut il lit le fichier `Makefile` dans le répertoire courant).

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

make (1/2)

- `make` est un logiciel traditionnel d'UNIX. C'est un "**moteur de production**" : il sert à appeler des commandes créant des fichiers. À la différence d'un simple script shell, make exécute les commandes seulement si elles sont nécessaires. Le but est d'arriver à un résultat (logiciel compilé ou installé, documentation créée, etc.) **sans nécessairement refaire toutes les étapes**.
- `make` fut à l'origine développé par le docteur Stuart Feldman, en 1977. Ce dernier travaillait alors pour *Bell Labs*.
- De nos jours, les fichiers `Makefile` sont de plus en plus rarement générés à la main par le développeur mais construit à partir d'outils automatiques tels qu'`autoconf`, `cmake` ou `qmake` qui facilitent la génération de `Makefile` complexes et spécifiquement adaptés à l'environnement dans lequel les actions de production sont censées se réaliser.

make (2/2)

- `make` est un outil indispensable aux développeurs car :
  - `make` assure la compilation séparée automatisée → plus besoin de taper une série de commandes
  - `make` permet de ne recompiler que le code modifié → optimisation du temps et des ressources
  - `make` utilise un fichier distinct contenant les règles de fabrication (Makefile) → mémorisation de règles spécifiques longues et complexes à retaper
  - `make` permet d'utiliser des commandes (ce qui permet d'assurer des tâches de nettoyage, d'installation, d'archivage, etc ...) → une seule commande pour différentes tâches
  - `make` utilise des variables, des directives, ... → facilite la souplesse, le portage et la réutilisation

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Makefile : les variables

- La première amélioration du fichier Makefile est d'utiliser des variables.
- On déclare une variable de la manière suivante : `NOM = VALEUR`
- Et on l'utilise comme ceci : `$(NOM)`

```
# le nom de l'exécutable :
TARGET=main # l'espace n'est pas obligatoire
MODULE = calcul
# le reste est "générique"
CC = gcc -c   # la commande de compilation
LD = gcc -o   # la commande pour l'édition de liens
CFLAGS = -Wall # les options de compilation
LDFLAGS =     # les options pour l'édition de liens

all: $(TARGET) # la cible par défaut est la cible de la première règle
     trouvée par make (ici all)

$(TARGET): $(TARGET).o $(MODULE).o
    $(LD) $(TARGET) $(LDFLAGS) $(TARGET).o
$(TARGET).o: $(TARGET).c
    $(CC) $(CFLAGS) $(TARGET).c
$(MODULE).o: $(MODULE).c
    $(CC) $(CFLAGS) $(MODULE).c
```

Makefile : les variables automatiques (1/2)

- Les variables automatiques sont des variables qui sont actualisées au moment de l'exécution de chaque règle, en fonction de la cible et des dépendances :

```
$@ : nom complet de la cible
$< : la première dépendance
$? : les dépendances plus récentes que la cible
$^ : toutes les dépendances
$* : correspond au nom de base (sans extension) de la cible courante
$$ : le caractère $

... (consulter la doc de make)
```

- Elles sont très utilisées dans les fichiers Makefile pour éviter d'écrire en "dur" les noms de fichiers dans les commandes.

Makefile : les variables automatiques (2/2)

- La deuxième amélioration du fichier Makefile est d'utiliser les variables automatiques.
- Cela permet d'ajouter notamment une dépendance sur un fichier *header* sans qu'il soit compiler :

```
...

$(TARGET): $(TARGET).o $(MODULE).o
    $(LD) $@ $(LDFLAGS) $^

$(TARGET).o: $(TARGET).c $(MODULE).h
    $(CC) $(CFLAGS) $<

$(MODULE).o: $(MODULE).c $(MODULE).h
    $(CC) $(CFLAGS) $<
```

Frères des Ecoles Chrétiennes Saint-Jean-Baptiste de la Salle - Avignon

Makefile : les règles génériques

- Certaines règles sont systématique : tous les fichiers `.c` sont compilés avec `gcc -c` par exemple.
- On peut alors utiliser une règle générique de la manière suivante :

```
...

%.o: %.c
    $(CC) $(CFLAGS) -o $@ $<

# ou pour des anciennes versions :
.c.o:
    $(CC) $(CFLAGS) -o $@ $<
```

Makefile : divers

- Quelques utilisations pratiques :

```
# Génération de liste de fichiers objets à partir de SRC :
SRC = main.c calcul.c
OBJ = $(SRC:.c=.o)

# Ou de manière plus "robuste" :
OBJ = $(strip $(patsubst %.c, %.o, $(wildcard *.c)))

# Génération de la liste des fichiers sources :
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)
```

Makefile : la cible .PHONY, clean et cleanall

- Dans un Makefile basique, la cible `clean` est la cible d'une règle ne présentant aucune dépendance et qui permet d'effacer tous les fichiers objets (`.o` ou `.obj`)
- Problème : si un fichier porte le nom d'une cible, il serait alors forcément plus récent que ses dépendances et la règle ne serait alors jamais exécutée.
- Solution : il existe une cible particulière nommée `.PHONY` dont les dépendances seront systématiquement reconstruites.

```
.PHONY: clean cleanall

clean:
    rm -f *.o

cleanall:
    rm -f *.o $(TARGET)
```