make

Programmation modulaire

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 écrit 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.

En C/C++, 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);`. Les déclarations sont fournies dans des fichiers en-tête ou *header* (d'extension `.h`).
- la **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 ; }`. Les définitions sont fournies dans des fichiers source (d'extension `.c` ou `.cpp`).
- l'**appel** (invocation) qui est son utilisation. Elle doit correspondre à la déclaration faite au compilateur qui la vérifie : `int res = plus(2, 2);`. Les utilisations des fonctions sont réalisées dans des fichiers source (d'extension `.c` ou `.cpp`).

> 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.

Compilation séparée

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.

Fabrication

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

- compilation séparée (l'option `-c` de `gcc/g++`) de tous les fichiers sources (d'extension `.c` ou `.cpp`) pour génèrer les fichiers objet (d'extension `.o` ou `.obj`)
- édition de liens de tous les fichiers objets (d'extension `.o` ou `.obj`) et des bibliothèques avec les options `-L` et `-l` pour fabriquer un exécutable (d'extension `.out` ou `.exe`) en utilisant l'option `-o` (*output*)

Makefile

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` (voir l'option `-f` pour choisir un nom de fichier différent).

make

`make` est un logiciel traditionnel d'UNIX. C'est un "**moteur de production**" : il sert à appeler des commandes créant des fichiers.

Lien : https://www.gnu.org/software/make/

Documentation : https://www.gnu.org/software/make/manual/html_node/index.html

Support de cours : http://tvaira.free.fr/dev/cours/cours-c-programmation-modulaire.pdf et https://btssn-lasalle84.github.io/guides-developpement-logiciel/guides-pdf/make.pdf

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.

`make` est un outil indispensable aux développeurs car :

- `make` assure la compilation séparée automatisée : plus besoin de saisir 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
- `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

Notion de règle

Une règle est une suite d'instructions qui seront exécutées pour construire une cible (*target*) 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)
```

> `make` attend une tabulation et non des espaces !

Lien : https://www.gnu.org/software/make/manual/html_node/Rule-Introduction.html#Rule-Introduction

Exemples

Le projet d'exemple contient les trois fichiers suivants et doivent permettre de fabriquer l'exécutable `test-calculs.out` :

- `main.cpp` contient la définition de la fonction principale `main()` qui utilise les fonctions `additionner()` et `multiplier()`

```cpp
#include <iostream>
#include "calcul.h"

int main()
{
    int x        = 40;
```

```
    int y        = 2;
    int resultat = 0;

#if QUESTION == 3
    x = 64;
    y = 2;
#endif
    resultat = additionner(x, y);
    std::cout << "x + y = " << resultat << std::endl;

#if QUESTION == 3
    x = 33;
    y = 2;
#endif
    resultat = multiplier(x, y);
    std::cout << "x * y = " << resultat << std::endl;

    return 0;
}
```

- `calcul.h` contient la déclaration des fonctions `additionner()` et `multiplier()`

```
#ifndef CALCUL_H
#define CALCUL_H

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

#endif // CALCUL_H
```

- `calcul.cpp` contient la définition des fonctions `additionner()` et `multiplier()`

```
#include "calcul.h"

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

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

> Inclusion unique : 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 habituellement une structure `#ifndef ... #define ... #endif`. Il est aussi possible d'utiliser la directive `#pragma once` avec certains compilateurs .

On va élaborer le fichier `Makefile` en plusieurs itérations pour ce projet.

Makefile de base

Le fichier `Makefile.v1` va utiliser trois règles :

- une règle pour fabriquer la cible `test-calculs.out` qui dépend de `main.o` et `calcul.o`
- une règle pour fabriquer la cible `main.o` qui dépend de `main.cpp`
- une règle pour fabriquer la cible `calcul.o` qui dépend de `calcul.cpp`

```
test-calculs.out: main.o calcul.o
	g++ -o test-calculs.out main.o calcul.o

main.o: main.cpp
	g++ -c main.cpp

calcul.o: calcul.cpp
	g++ -c calcul.cpp
```

Lien : https://www.gnu.org/software/make/manual/html_node/Simple-Makefile.html#Simple-Makefile

- Première fabrication :

```
$ make -f Makefile.v1
g++ -c main.cpp
g++ -c calcul.cpp
g++ -o test-calculs.out main.o calcul.o

$ ./test-calculs.out
x + y = 42
x * y = 80
```

- Si rien n'a changé :

```
$ make -f Makefile.v1
make: « test-calculs.out » est à jour.
```

- Si on modifie `main.cpp` :

```
$ touch main.cpp
$ make -f Makefile.v1
g++ -c main.cpp
g++ -o test-calculs.out main.o calcul.o
```

- Si on modifie `calcul.cpp` :

```
$ touch calcul.cpp
$ make -f Makefile.v1
g++ -c calcul.cpp
g++ -o test-calculs.out main.o calcul.o
```

- Si on supprime `calcul.o` :

```
$ rm -f calcul.o
$ make -f Makefile.v1
g++ -c calcul.cpp
g++ -o test-calculs.out main.o calcul.o
```

- Si on modifie `calcul.h` :

```
$ touch calcul.h
$ make -f Makefile.v1
make: « test-calculs.out » est à jour.
```

L'exécutable n'est pas refabriqué car il n'y a aucune dépendance sur le fichier `calcul.h` !

On ajoute la dépendance sur le fichier `calcul.h` :

```
test-calculs.out: main.o calcul.o
	g++ -o test-calculs.out main.o calcul.o

main.o: main.cpp calcul.h
	g++ -c main.cpp

calcul.o: calcul.cpp calcul.h
	g++ -c calcul.cpp
```

- Vérification :

```
$ touch calcul.h
$ make -f Makefile.v1
g++ -c main.cpp
g++ -c calcul.cpp
g++ -o test-calculs.out main.o calcul.o
```

Notions de 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)`

Liens :

- https://www.gnu.org/software/make/manual/html_node/Reference.html#Reference
- https://www.gnu.org/software/make/manual/html_node/Makefile-Basics.html#Makefile-Basics

Il existe un ensemble de variables prédéfinies dans `make` : `TARGET`, `CC`, `CXX`, ...

Liste des variables prédéfinies : https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html

```
# le nom de l'exécutable :
TARGET = test-calculs.out
MAIN = main
MODULE = calcul

# Les variables génériques
#CC = gcc -c # la commande de compilation en C
CXX = g++ -c # la commande de compilation en C++
LD = g++ -o # la commande pour l'édition de liens
CFLAGS = -Wall -DQUESTION=3 # les options de compilation, exemple : -Ixxx -Ox
LDFLAGS = # les options pour l'édition de liens, exemple : -lxxx -Lxxx

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

$(TARGET): $(MAIN).o $(MODULE).o
	$(LD) $(TARGET) $(LDFLAGS) $(MAIN).o $(MODULE).o

$(MAIN).o: $(MAIN).cpp $(MODULE).h
	$(CXX) $(CFLAGS) $(MAIN).cpp

$(MODULE).o: $(MODULE).cpp $(MODULE).h
	$(CXX) $(CFLAGS) $(MODULE).cpp
```

Les variables automatiques

La deuxième amélioration est d'utiliser les **variables automatiques**.

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
- `$^` : toutes les dépendances
- `$?` : les dépendances plus récentes que la cible
- `$*` : correspond au nom de base (sans extension) de la cible courante

Elles sont très utilisées dans les fichiers `Makefile` pour simplifier les commandes.

```
# le nom de l'exécutable :
TARGET = test-calculs.out
MAIN = main
MODULE = calcul

# Les variables génériques
CC = gcc -c # la commande de compilation en C
CXX = g++ -c # la commande de compilation en C++
LD = g++ -o # la commande pour l'édition de liens
```

```
CFLAGS = -Wall # les options de compilation, exemple : -Ixxx -Ox
LDFLAGS = # les options pour l'édition de liens, exemple : -lxxx -Lxxx

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

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

$(MAIN).o: $(MAIN).cpp $(MODULE).h
	$(CXX) $(CFLAGS) $<

$(MODULE).o: $(MODULE).cpp $(MODULE).h
	$(CXX) $(CFLAGS) $<
```

Lien : https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables

Les règles génériques

Certaines règles sont systématiques : 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 $@ $<
```

Lien : https://www.gnu.org/software/make/manual/html_node/Pattern-Intro.html#Pattern-Intro

Quelques utilisations pratiques des fonctions fournies par `make` :

```
# Génération de la liste des fichiers sources :
SRC := $(wildcard *.c)
# Génération de liste de fichiers objets à partir de SRC :
OBJ := $(patsubst %.c, %.o, $(SRC))

# Affichage de la liste
$(info LISTE_SRC : $(SRC))
```

Lien : https://www.gnu.org/software/make/manual/html_node/Wildcard-Function.html#Wildcard-Function

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`). La cible `cleanall` supprime en plus l'exécutable (`.out` ou `.exe`).

Lien : https://www.gnu.org/software/make/manual/html_node/Standard-Targets.html#Standard-Targets

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) *.o

cleanall:
	$(RM) *.o $(TARGET)
```

Un fichier `Makefile` qui intègre :

- a. une liste des fichiers header `.cpp` dans une variable `SRC`
- b. une liste des fichiers header `.h` dans une variable `HEADERS`
- c. une compilation générique des fichiers `.cpp`
- d. les cibles `clean` et `cleanall`

```
# le nom de l'exécutable :
TARGET = test-calculs.out
MAIN = main
MODULE = calcul

SRC := $(wildcard *.cpp)
$(info LISTE_SRC : $(SRC))
HEADERS := $(wildcard *.h)
$(info LISTE_HEADERS : $(HEADERS))

# Les variables génériques
CC = gcc -c # la commande de compilation en C
CXX = g++ -c # la commande de compilation en C++
LD = g++ -o # la commande pour l'édition de liens
CFLAGS = -Wall # les options de compilation, exemple : -Ixxx -Ox
LDFLAGS = # les options pour l'édition de liens, exemple : -lxxx -Lxxx

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

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

%.o: %.cpp $(HEADERS)
	$(CXX) $(CFLAGS) -o $@ $<

.PHONY: clean cleanall

clean:
	$(RM) *.o
```

```
cleanall:
	$(RM) *.o $(TARGET)
```

> Généralement, on ajoute une cible `install` qui permet de réaliser l'installation de l'exécutable fabriqué par le `Makefile`. Sous GNU/Linux, la commande `install` permet de copier des fichiers en attribuant les droits nécessaires.

De nos jours ...

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`, `qmake`, etc. qui facilitent la génération de `Makefile` complexes et spécifiquement adaptés à l'environnement dans lequel les actions de production sont censées se réaliser.

Liens :

- https://fr.wikipedia.org/wiki/Autoconf
- https://fr.wikipedia.org/wiki/CMake et http://tvaira.free.fr/dev/cours/cmake.pdf
- https://doc.qt.io/qt-6/qmake-manual.html

Voir aussi : https://fr.wikipedia.org/wiki/Gradle est un moteur de production fonctionnant sur la plateforme Java. Il permet de construire des projets en Java, Scala, Groovy voire C++.
