Notes de cours : Compilation séparée et makefiles

Les indications données ici sont valables autant pour des programmes en C++ que pour des programmes en C.

Lorsque la taille et la complexité des programmes augmente, il devient nécessaire de séparer le code en plusieurs fichiers sources. On définit alors les fonctions et classes en deux parties. D'un côté, un *prototype* décrit un élément (méthode ou fonction) sans lui donner de corps. On déclare généralement les prototypes dans un *fichier d'en-têtes* (portant l'extension .h) que l'on inclut dans les fichiers sources qui en ont besoin. Un prototype s'écrit de la même manière qu'une déclaration traditionnelle, mais ne contient pas de corps. Il est donc terminé par un point-virgule au lieu d'avoir un bloc définit par une paire d'accolades { … }.

```
// Function prototype
void myFunc(int a); // Terminated by ; instead of { }

// Class prototype
// The class prototype must specify all member variables and methods,
// but the body of the methods can be kept empty.
class MyClass
{
public :
    MyClass();
    int myMethod();
};
```

La *définition du corps*, ou l'*implémentation* de la fonction ou méthode se fait généralement dans un fichier source distinct, qui inclut le fichier en-tête prototypant les éléments implémentés. Si on implémente une méthode, le nom de la classe préfixe le nom de la méthode, en étant séparée par ::, comme illustré dans l'exemple suivant.

```
#include <stdio.h>
#include "exHeader.h"

void myFunc(int a)
{
    printf("%d\n",a);
}

MyClass::MyClass()
{
    printf("Constructing MyClass object\n");
}

int MyClass::myMethod()
{
    return 5;
}
```

Téléchargez les fichiers *exMain.cpp*, *exHeader.h*, et *exImpl.cpp*, qui illustrent le fonctionnement de prototypes. Vous pouvez compiler le programme en spécifiant tous les noms

des fichiers à compiler sur la ligne de commande. Regardez ce que vous dit le compilateur lorsque vous enlevez le prototype ou l'implémentation d'une fonction.

Au lieu de compiler tous les fichiers en une fois, il est aussi possible de séparer les phases de la compilation. On compile d'abord chaque fichier source en un fichier objet, puis on *lie* (link) tous les fichiers objets pour constituer notre exécutable. On ajoute pour cela l'option –c lorsque l'on compile avec gcc ou g++. Ainsi, il n'est pas nécessaire de recompiler tout le programme lorsqu'un seul fichier source a été modifié. Lorsque les programmes deviennent complexes, le gain de temps est substantiel.

```
> g++ -Wall –c -o exMain.o exMain.cpp
> g++ -Wall –c –o exImpl.cpp exImpl.o
> g++ -o ex exMain.o exImpl.o
```

Afin de ne pas devoir compiler chaque fichier à la main, on automatise le processus en utilisant des *makefiles*. Le principe de base est d'écrire une liste de règles auxquelles on associe des instructions de compilation. Pour chaque règle, on indique les autres règles ou les noms de fichiers dont elle dépend. Lorsqu'un fichier est modifié, les règles qui dépendent de ce fichier sont invalidées et les instructions de compilations sont exécutées. On nomme donc généralement les règles d'après le nom du fichier qu'elles génèrent. La règle suivante est ainsi exécutée chaque fois que le fichier *exMain.cpp* est plus récent que le fichier *exMain.o*.

```
# Rule exMain.o depends on exMain.cpp and exHeader.h
#  (because exMain.cpp includes exHeader.h)
exMain.o: exMain.cpp exHeader.h
    g++ -Wall -c –o exMain.o exMain.cpp
```

Les règles peuvent ainsi être chainées jusqu'à l'obtention de l'exécutable. Le makefile ci-dessous compile et lie notre programme d'exemple. Il contient une règle supplémentaire *clean* (c'est une convention), très pratique pour faire de l'ordre ou pour s'assurer que tous les fichiers objets sont bien compilés avec les mêmes options de compilation.

```
ex: exMain.o exImpl.o
    g++ -o ex exMain.o exImpl.o

exImpl.o: exImpl.cpp exHeader.h
    g++ -Wall –c –o exImpl.o exImpl.cpp

exMain.o: exMain.cpp exHeader.h
    g++ -Wall -c –o exMain.o exMain.cpp

clean:
    rm –f ex *.o
```

Téléchargez le makefile ci-dessus sur le site du cours. On exécute un makefile en utilisant la commande *make*. Cette commande prend par défaut un fichier nommé *Makefile* ou *makefile*, mais on peut utiliser un autre nom de fichier et l'exécuter avec *make –f makefileWithAnotherName*. On peut spécifier la règle à exécuter en passant le nom de la règle en argument de *make*, par exemple *make clean* pour supprimer les exécutables et fichiers objets.

```
> make clean
rm –f ex *.o
> make
g++ -Wall  -c –o exMain.o exMain.cpp
g++ -Wall  -c –o exImpl.o exImpl.cpp
g++ -o ex exMain.o exImpl.o
> make
make: ‘ex’ is up to date.
```

Comme aucun fichier n’a été modifié entre la première et la deuxième exécution de *make*, il ne fait rien la deuxième fois. Si l’on modifie maintenant le fichier *exMain.cpp*, *make* recompile uniquement ce fichier et génère un nouvel exécutable en utilisant le nouveau fichier objet.

```
> make
g++ -Wall  -c –o exMain.o exMain.cpp
g++ -o ex exMain.o exImpl.o
```

On peut aussi compiler un seul objet en spécifiant la règle correspondante. Cet objet ne sera plus recompilé tant que les fichiers sources associés n’auront pas été modifiés.

```
> make clean
rm –f ex *.o
> make exImpl.o
g++ -Wall  -c –o exImpl.o exImpl.cpp
> make
g++ -Wall  -c –o exMain.o exMain.cpp
g++ -o ex exMain.o exImpl.o
```

Il n’y a aucune restriction sur le nombre de règles qui peuvent être déclarées ainsi que sur leurs dépendances. On peut ainsi générer des exécutables différents en liant ensemble d’autres fichiers objets.

```
# Generate executable ex using exMain.o and exImpl.o
ex: exMain.o exImpl.o
    g++ –o ex exMain.o exImpl.o

# Generate executable that uses a different main function
ex2: exOtherMain.o exImpl.o
    g++ –o ex exOtherMain.o exImpl.o
```

Le choix de l’exécutable à générer peut ensuite être fait en appelant *make ex* ou *make ex2* explicitement. Autrement, on peut ajouter une règle qui dépend de *ex* et *ex2*, auquel cas les deux exécutables sont maintenus à jour simultanément.