Compilation séparée, make et makefile

Ce document présente une façon de découper un programme C/C++ en plusieurs fichiers dans le but de :
- ne compiler que ce qui est nécessaire à chaque compilation,
- accélérer la compilation,
- réutiliser au mieux des librairies déjà écrites,
- protéger les sources en ne distribuant que des parties déjà compilées,
- etc.

1. Compilation et édition de lien

Avant de découper un programme en plusieurs parties, un petit point rapide sur les opérations nécessaires à la traduction d'un code écrit en C/C++ vers un programme exécutable. Cette opération qu'on appelle généralement compilation contient en fait 2 étapes :
- la compilation proprement dite, qui est l'opération de traduction des instruction,
- l'édition de lien qui s'occupe de faire correspondre les adresses mémoires des variables et fonctions du programme. C'est ce traitement là qui fabrique le programme exécutable.

Dans le cas de la compilation d'un programme contenu dans un seul fichier, ces opérations s'enchaîne automatiquement lors de l'appel du compilateur (par g++ par exemple). Dans le cas d'un programme découpé en plusieurs fichiers, ces opérations seront distinguées par l'utilisation de commandes différentes ou d'option du compilateur. Par exemple pour le compilateur g++ :
- compilation seulement : g++ -c,
- édition de lien g++ ou ld.

Dans la suite le terme **compilation** désignera la **compilation seulement** et l'expression **édition de lien** désignera la **fabrication du programme exécutable**.

2. Découpage du programme

Comment faut-il découper un programme en plusieurs fichiers ?
Le premier critère de découpage est fonctionnel, c'est à dire que que on va isoler dans un (ou plusieurs) fichier(s) les parties qui concerne le même problème, ou la même structure de données. Cet ensemble de fichiers sera appelé par la suite librairie.

Exemple :
on développe un programme qui nécessite de manipuler des matrices de nombres complexes.
Toutes les opérations qui concernent ces matrices (type, initialisation, remplissage, affichage, opérations, etc.) seront isolées dans une librairie pour pouvoir :
- la mettre au point de manière indépendantes, afin de
- la réutiliser éventuellement plus tard dans un autre programme.

De même on peut isoler ce qui concerne les nombres complexes indépendamment de la problématique matrice parce que on peut vouloir réutiliser plus tard des nombres complexes sans avoir besoin de matrices et parce que les opérations qui les concernent peuvent être mises au point en dehors du contexte de matrice.

Si on pousse ce raisonnement, une façon de découper le programme, sans trop réfléchir, peut être celui du

découpage des structures de données : on crée une librairie par structure de données utilisée dans le programme.

Une fois les unités fonctionnelles définies, le découpage n'est plus qu'un problème technique guidés par le fonctionnement du compilateur.

La question à se poser est : de quoi a besoin le compilateur pour traduire du code C/C++ en code machine ? Plus précisément : qu'a t-il besoin de connaître ?
Bien évidemment il est capable de "traduire" toutes les instructions et c'est quand il rencontre une variable, une fonction, un type, une classe... qu'il ne connaît pas que cette question se pose. Si c'est dans le même fichier il suffit de le placer avant, mais si c'est dans un autre fichier il faut lui indiquer un minimum d'information :

- dans le cas d'une variable il a besoin d'en connaître le type,
- dans le cas d'une fonction il a besoin d'en connaître le nombre d'argument, leur type et le type de retour. Remarque : pour une fonction, ces informations sont contenues dans l'en-tête de la fonction qui s'écrit comme la première ligne d'une fonction (sans le bloc d'instruction), duquel on peut omettre les noms des variables mais en laissant les types et en respectant l'ordre et en terminant avec un point virgule. Dans le cas de classes, les fonctions de la classe seront elles aussi décrites par leurs en-têtes.

Ces informations doivent donc être ajoutées à chaque fichier qui sera compilé seul. La façon classique de faire cela est de présenter chaque librairie sous forme de 2 fichiers, un pour les "déclarations" et un pour les "instructions". Classiquement le premier porte l'extension ".h" et le second l'extension ".cpp".

Si on reprends notre exemple de nombres complexes, la librairie des matrices sera découpée en un fichier matcomp.h et un fichier matcomp.cpp.

Le fichier matcomp.h contient la partie déclarative c'est à dire tout ce qui est nécessaire pour compiler un autre programme utilisant matcomp et le fichier matcomp.cpp contient la partie "instruction" de la librairie qui devra être fournie pour le programme final.
Il faut bien noter aussi que pour compiler la librairie elle même il faut regrouper les 2 fichiers .h et .cpp qui la compose.

Les opération de compilation génèrent des fichiers intermédiaires dont le nom générique est « fichier objet » et qui portent comme nom de fichier le nom du fichier ".cpp" fourni mais avec l'extension ".o". Ainsi la compilation de matcomp.cpp par la commande :
g++ -c matcomp.cpp
produit le fichier : matcomp.o

Tout ce qui concerne la classe complexe de l'exemple va être réparti de la même façon dans 2 fichiers complexe.cpp et complexe.h

3. Organisation et regroupement des fichiers

Pour générer une librairie il faut donc compiler conjointement le fichier ".cpp" et le fichier ".h" de la librairie, pour cela on va les regrouper comme s'il s'agissait d'un seul fichier grâce à la directive `#include`.
Cette directive prend en argument un nom de fichier et a pour effet d'inclure le texte du fichier argument dans le fichier dans lequel elle se trouve, à l'endroit où elle se trouve. Ainsi pour compiler la librairie

précédente, il faut inclure le fichier matcomp.h dans le fichier matcomp.cpp, avant que le compilateur ait besoin de son contenu pour compiler et générer le fichier objet matcomp.o.

Remarque : vous pouvez voir l'effet de la directive grâce à l'option -E de g++.

4. Comment éviter les inclusions multiples

En incluant les fichiers ".h" dans tous les fichiers ".cpp" qui en ont besoin, il y a un risque que l'un d'eux se retrouvent inclus plusieurs fois, et cela peut générer une erreur de déclaration multiple. Pour éviter cela, le plus simple est de marquer que l'inclusion a déjà été faite par la définition d'une macro grâce à la directive `#define`. Dans tous les fichiers ".h" mettre au début :

```
#ifndef « un identificateur qui doit être unique dans le programme »
#define « un identificateur qui doit être unique dans le programme »
```

et à la fin :

```
#endif
```

L'identificateur doit être unique dans le programme, vous pouvez choisir ce que vous voulez. Il contient classiquement un ou des caractères "_", est en rapport avec le nom du fichier et éventuellement avec le nom du programmeur dans le cas d'un développement à plusieurs.

Le fonctionnement de ce dispositif est le suivant :
le compilateur lit la directive `#ifndef _MA_LIB_BOB_H_` ,
- si il connaît déjà _MA_LIB_BOB_H_ il ignore le texte jusqu'à `#endif`,
- si il n'a jamais vu _MA_LIB_BOB_H_ (**if n**ot **def**ined), alors il va lire tout ce qui suit jusqu'à `#endif` et notamment en commençant par définir _MA_LIB_BOB_H_, donc il la connaîtra à la prochaine inclusion et il sautera le bloc.

5. Compilation séparée

Le découpage en plusieurs fichiers permet de ne recompiler que ceux qui ont été modifié. Ainsi avec l'exemple sur les nombres complexes, si on modifie le fichier matcomp.cpp, il faut recompiler matcomp.cpp et main.cpp mais pas complexe.cpp. En revanche si on modifie complexe.h, il faut tout recompiler.
Cela peut être un peu fastidieux si on compile à la main, en lancant les commendes dans une console, mais heureusement il existe une commande qui s'occupe de choisir ce qu'il faut recompiler, la commande `make`

6. La commande make

La commande `make` interprète les instructions qu'elle trouve dans un fichier « `makefile` ».
Les instructions sont organisées par bloc de 2 lignes ainsi :

conséquent : liste d'antécédents
    commandes

- conséquent est la cible des 2 lignes, par exemple le programme main, ou un fichier objet point.o
- la liste d'antécédents contient tous les fichiers qui servent à fabriquer le conséquent, et notamment ceux

dont la modification doit entraîner la re-fabrication du conséquent,
- commande explique comment faire pour fabriquer le conséquent

Attention la 2e ligne commence par une tabulation.

Exemple
le fichier `makefile` de l'exemple

la 1re ligne sert à fabriquer le programme principal, qui nécessite, par exemple, matcomp.o, dont la fabrication est expliquée dans le 2e bloc, etc.

```
main : main.cpp matcomp.o complexe.o
     g++ -o main matcomp.o complexe.o main.cpp

matcomp.o : matcomp.h matcomp.cpp complexe.h
     g++ -c matcomp.cpp

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

La commande `make` cherche un fichier `makefile` ou `Makefile` et lance la réalisation du 1er conséquent, qui lancera éventuellement la réalisation des autres nécessaires.

La commande `make consequent` lance la réalisation d'un conséquent en particulier.

La commande `make -f makefile.1` utilise le fichier `makefile.1`.

Il y a plusieurs fichiers exemples de makefile pour voir d'autres possibilités de la commande.