Compiler du C++

Titou

5 mai 2013

1 Distinction entre les fichiers

Les fichiers .hpp sont appelés **fichiers headers**, les fichier .cpp les **fichiers d'implémentation**.

Le rôle des headers est de déclarer l'interface, c'est à dire présenter les types de données et les signatures de variables/fonctions[1] à l'utilisateur *(ie c'est d'habitude là qu'on met la documentation)*, mais surtout au compilateur, qui pourra savoir si un type de donnée/variable/fonction existe ou non.

Les fichiers d'implémentation contiennent quant à eux le code des fonctions. Pour utiliser des variables/fonctions définies ailleurs (librairie standard, autre fichier, ...), ils ont besoin de connaître la signature de cette variable. Ils pourront la connaître en incluant le fichier header la contenant. Exemple :

```
#include <cstring>

int main(int argc, const char **argv){
  const char *origine = "Hello world";
  char copie[12];

  /* La fonction strcpy, qui copie une chaine dans une autre
     est declaree dans cstring. Le code qui la fait tourner se
     trouve lui dans la librairie standard. */
  strcpy(copie, origine);

  return 0;
}
```

2 Etapes de compilation

Quand on compile, on peut distinguer 3 grandes étapes (il y en a plus) :

2.1 Préprocesseur

Toutes les directives de prérocesseur sont executées : les `#include` sont remplacés par le contenu du fichier qu'ils incluent, les valeurs définie par `#define` sont remplacées, ...

Types d'erreurs rencontrées à cette étape

- `file XXX.hpp not found`
- `Error in macro`

Exécuter juste cette étape : `$ g++ -E fichierEntree.cpp -o fichierSortie.ii`

1. La distinction importe peu : en C/C++, un nom de fonction est en fait une variable de type `pointeur sur fonction` qui pointe sur la première instruction de la fonction.

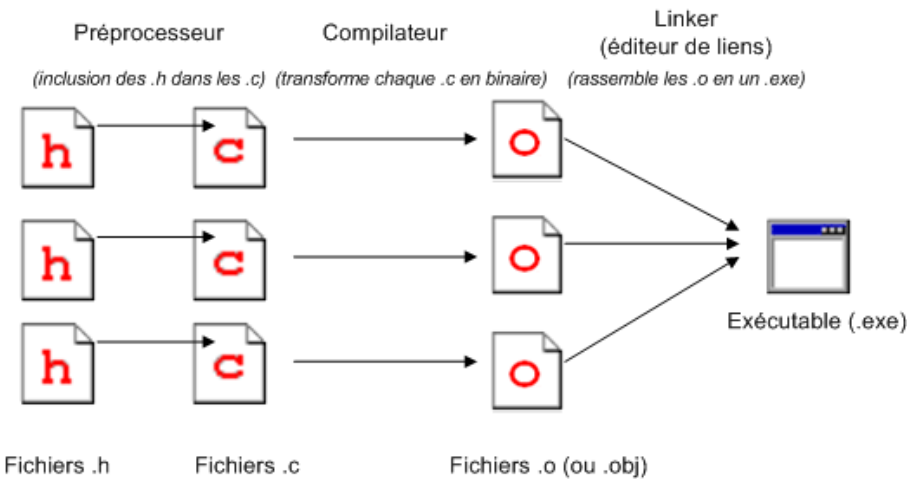

FIGURE 1 – Source : Tutoriel C du Site du Zéro

2.2 Compilation et assemblage

Un fichier d'implémentation est ensuite transformé : les instructions C sont transformées en instructions machine. Le fichier résultant n'est pas encore tout à fait exécutable, on l'appelle **fichier objet**, et il porte généralement l'extension **.o**

Types d'erreurs rencontrées à cette étape

- `syntax error`
- `... was not declared in this scope`
- `expected ...`
- `unknow type or identifier`

Exécuter juste cette étape : `$ g++ -c fichierEntree.cpp -o fichierSortie.o`

2.3 Edition des liens

Enfin, différents fichiers objet sont rassemblés ensemble en un seul **fichier exécutable**, le programme final. Pour ce faire, toutes les variables/fonctions dont les signatures ont été incluses grâce aux fichiers headers sont remplacées par leur adresse dans le programme.

Types d'erreurs rencontrées à cette étape

- `Undefined reference to ...`

Exécuter juste cette étape : `$ g++ fichiersObjets -o fichierExecutable`

2.4 Raccourcis

`g++` permet d'exécuter les 3 étapes en une seule, ainsi on peut très bien écrire, par exemple :

```
$ g++ -o mainProjLang mainProjLang.cpp Tree.cpp SplitGame.cpp
```

3 Automatisation avec make

La commande `make` sur Linux ou Mac OS X permet d'automatiser le processus de compilation, tout en gardant les fichiers intermédiaire pour ne produire que ceux qui doivent être mis à jour *(si l'heure du fichier de sortie est inférieure à l'heure du fichier d'entrée)*.

Les instructions pour la *recette* finale sont données dans un fichier nommé **Makefile**. Toutes les étapes ne doivent pas être décrites explicitement, puisque `make` *connaît* des règles de base (comme .cpp => .o).

3.1 Exemple

Le fichier Makefile suivant permet, en tapant simplement la commande `make` de compiler l'exécutable final mainProjLang. En tapant `make test`, et à condition que le fichier test.cpp existe, on construit un executable avec les memes fichiers objets, mais où la fonction `main()` vient du fichier test.cpp. Note : les lignes indentées commencent par des tabulations.

```
#Compilateur C++
CXX = g++
#Flags C++
CXXFLAGS = -x c++ -pedantic -Wall -Wextra -O3

#Executables finaux
TARGETS = main mainProjLang
#Fichiers objets necessaires aux executables finaux
OBJECTS = Tree.o SplitGame.o

all: ${TARGETS}
% : %.cpp

#Compilation d'un executable avec les objets
% : %.o ${OBJECTS}
	${CXX} -o $@ $^

#Efface les fichiers objets
.PHONY : clean
clean:
	rm -f *.o

#Efface tous les produits de compilation
mrproper: clean
	rm -f ${TARGETS}
```

4 Références

- http://www.siteduzero.com/informatique/tutoriels/compilez-sous-gnu-linux
- http://www.gnu.org/software/make/manual/make.html
- A. Tannebaum, Structured Computer Organization, 5ème édition, *Section 7.3 : The assembly process*
- http://www.siteduzero.com/informatique/tutoriels/apprenez-a-programmer-en-c/la-compilation-sepa