La compilation en C

Yann Rotella

Université de Versailles Saint Quentin en Yvelines
Université Paris Saclay
yann.rotella@uvsq.fr

1 décembre 2021

À la différence des langages *interprétés* comme `Python` par exemple, le langage `C` est un langage dit *compilé*. Les fichiers source sont des fichiers lisibles par l'utilisateur. La compilation en `C` se fait en deux temps. Dans un premier temps les fichiers source (avec l'extension `.c`) sont d'abord transcrits en des fichiers objets (avec l'extension `.o`). Ensuite et enfin, ces fichiers objets, explicités sous la forme d'un langage machine sont liés par le compilateur de manière à produire un fichier `exécutable`. Le but de cette fiche est de revoir les notions basiques de compilation et notamment celles de la dernière étape qui s'occupe de `lier` les fichiers objets.

1 La fonction main et lien avec le bash

La fonction `main` en `C` possède de manière générale la signature suivante:

```
int main(int argc, char** argv) {
    /* CODE */
    return 1;
}
```

Le type de sortie `int` peut aussi être `void`, cela a pour notre cours peu d'importance.

En revanche, les *arguments* `argc` et `argv` sont très utiles. En effet, lorsqu'on écrit un programme, on a parfois (souvent) besoin de le tester un grand nombre de fois, et sur potentiellement des paramètres différents, afin d'évaluer par exemple le coût de notre programme, ou tout simplement faire des tests. Plus précisément, on ne veut pas recompiler le programme à chaque fois qu'on souhaite utiliser celui-ci. Les arguments de la focntion `main` permettent ceci.

- `argc` est un entier qui vaut le nombre d'arguments passés en ligne de commande via le bash, nom de la commande incluse (i.e. le nom du programme executable).
- `argv` est un tableau de `str` (chaînes de caractères) qui contient les dits arguments. La taille de `argv` vaut donc `argc`.

À partir de maintenant, on exécutera tout le temps nos programmes par la ligne de commande du bash.

2 Bibliothèques et Fichiers séparés

Les bonnes pratiques d'un programmeur consistent à séparer en petit bouts (modules) le code. Ceci permet de réutiliser des modules dans plusieurs applications, de faciliter un travail à plusieurs ou d'avoir un code plus lisible. Dans le cas d'un langage compilé (`C`, `Rust` par exemple), il y a une quatrième utilité, c'est de rendre plus rapide la compilation du programme, puisque cela permet de ne pas recompiler les modules non-modifiés depuis la dernière compilation.

Dans `gcc`, on distingue deux étapes: la phase d'assemblage et l'édition de liens.

2.1 Phase d'assemblage

En `C`, les fichiers *source* possèdent l'extension `.c`, les fichiers compilés possèdent l'extension `.o`. On parle de fichiers *objet*. La première étape de la compilation consiste à produire un fichier objet pour chaque fichier source correspondant. La syntaxe dans le bash est la suivante.

```
gcc -c fichier1.c
gcc -c fichier2.c
gcc -c fichier3.c
gcc *.c
```

2.2 Édition de liens

Comme nous prenons des bonnes pratiques, on a donc plusieurs fichiers objets. Or, pour produire notre programme, nous avons besoin de chacun de ces fichiers objets, et de les *lier* ensemble. C'est la deuxième étape réalisée par le compilateur `gcc`: linking. Elle est réalisée naturellement par le *linker* de gcc. La syntaxe dans le bash est la suivante.

```
gcc -o my_prog.exe *.o
```

Naturellement, quand on n'a qu'un seul fichier, il n'y a pas besoin de séparer ces deux étapes, et nous pouvons utiliser la syntaxe suivante.

```
gcc -o my_prog.exe main.c
```

L'option `-o` permet de spécifier le nom du fichier produit. Si cette dernière n'est pas mise, le fichier sorti est par défaut un `a.out`.

3 Fichiers d'entête et dépendances

Au début de la compilation, il y a une phase de *preprocessing*. Dans cette phase, certaines vérifications sont réalisées, notamment une partie qui vérifie les types des arguments des fonctions, qui vérifie que les variables sont définies,...

Que ce soit pour le compilateur ou bien pour la lisibilité du code, les bonnes pratiques de compilation consistent à déclarer les prototypes des fonctions, avant de coder les-dites fonctions. Normalement, un bon fichier source a la forme suivante.

```
int fonction1(void);
unsigned int fonction2(int *T);
void fonction3(unsigned int x);

int fonction1(void){
    /* CODE */
    return r;
}

unsigned int fonction2(int *T){
    /* CODE */
    return r;
}

void fonction3(unsigned int x){
    /* CODE */
}
```

Cependant, si l'on veut mettre les trois fonctions dans trois fichiers source séparés et que les fonctions s'appellent entre elles, alors on a un problème. En effet, le compilateur doit connaître a minima les prototypes des fonctions pour pouvoir réaliser la phase de preprocessing. Pour résoudre cela, on utilise des entêtes (headers), qui contiennent les prototypes des fonctions. C'est le préprocesseur de `gcc` qui gère cela.

Note 1. Pour voir ce que fait le préprocesseur, on peut utiliser l'otpion `-E` de `gcc`.

Les fichiers d'entête possèdent l'extension `.h`, et il faut rajouter la macro suivante au début de chaque fichier source qui nécessite les prototypes des fonctions définies dans l'entête correspondant.

```
#include "mon_entete.h"
```

En reprenant l'exemple précédant, cela donne un fichier `entete.h` dans lequel on mettra:

```
int fonction1(void);
unsigned int fonction2(int *T);
void fonction3(unsigned int x);
```

ainsi qu'un fichier `fonction1.c` dans le lequel il y aura:

```
#include "entete.h"
int fonction1(void){
    /* CODE */
    return r;
}
```

ainsi qu'un fichier `fonction2.c` contenant

```
#include "entete.h"
unsigned int fonction2(int *T){
    /* CODE */
    return r;
}
```

et de même pour `fonction3.c`.

Il peut y avoir plusieurs entêtes pour un seul gros programme, et dans un fichier `.c` il peut y avoir plusieurs fonctions. Ce qui compte le plus ici, c'est de réaliser quelque chose de logique, et de couper son code en des bouts qui sont cohérents.

Pour compiler le code avec des entêtes, il n'y a rien de particulier à réaliser. Le préprocesseur de gcc trouve automatiquement les entêtes, sans avoir à préciser dans la ligne de commande les `.h`.

3.1 Gestion des headers avec ifndef/define

Dans des gros projets, on peut avoir un fichier `fichier1.h` qui a besoin de fonctions définies dans `fichier2.h`, qui lui même a besoin de fonctions définies dans `fichier1.h`. Dans ce cas, le préprocesseur peut ne pas comprendre car il va avoir besoin de définir les prototypes des fonctions contenues dans le premier fichier `fichier1.h` potentiellement deux fois. En reîtérant cela, on pourrait même avoir un code où la phase du préprocesseur ne termine même pas.

L'astuce très classique pour résoudre ce problème consiste à utiliser la syntaxe suivante dans chaque header.

Premier fichier `fichier1.h`:

```
#ifndef MON_FICHIER_1
#define MON_FICHIER_1

#include "fichier2.h"
/* Macros */
/* Prototypes */

#endif
```

Deuxième fichier `fichier2.h`:

```
#ifndef MON_FICHIER_2
#define MON_FICHIER_2

#include "fichier1.h"
/* Macros */
/* Prototypes */

#endif
```

En faisant cette astuce, on remarque que l'inclusion de chaque header n'aura lieu qu'une et une seule fois, gérant *de facto* le problème.

4 Les bibliothèques

Une bibliothèque est un ensemble de fichiers objets (`.o`). Nous avons deux types de librairies: *statiques* et *dynamiques*.

- Les bibliothèques statiques sont, quand elles sont utilisées, liées au programme **avant** l'exécution du programme. Par conséquent, l'éxécutable en question contient le code objet de la bibliothèque, ce qui peut être assez lourd. L'extension des bibliothèques statiques est en `.a`.
- Les bibliothèques dynamiques sont elles liées seulement à l'**exécution** du programme. Par conséquent, le programme ne contient que l'adresse mémoire (un pointeur) de chaque fonction à éxécuter. C'est lors de l'exécution que le système d'exploitation va chercher le code objet desdites fonctions à appeler. Sous Windows, on utilise l'extension `.dll`. Sous Unix, on utilise l'extension `.so` (shared object).

Note 2. Aujourd'hui, la plupart des bibliothèques utilisées sont dynamiques.

La commande que l'on utilise pour créer une bilbiothèque statique s'appelle `ar` et cela donne les lignes de commandes suivantes.

```
gcc -c *.c
ar r libma_lib.a *.o
```

N'hésitez pas à fouiller sur internet les différentes options de `ar`.

Pour les bibliothèques dynamiques, on doit utiliser l'option `-fpic` qui permet de générer du code relogeable (portabilité).

```
gcc -fpic -c *.c
```

Enfin, la bibliothèque dynamique peut être créée avec l'option `-shared`

```
gcc -shared -o libma_lib.so *.o
```

Dans tous les cas, nous allons avoir des programmes qui nécessitent à un moment d'appeler le code présent dans ces librairies. Ainsi, il faut à chaque fois indiquer à gcc que l'on a besoin de la bibliothèque. Il a du vous arriver d'avoir besoin d'utiliser `-lm` à la compilation, afin d'utiliser la librairie maths `libm.a` située a priori dans le dossier `lib`. Ainsi, produire l'éxecutable qui utilise une librairie (ou plusieurs) peut se faire de la manière suivante.

```
gcc -o main main.o -lma_lib
```

- Inconvénient des librairies statiques: si la librairie a une mise à jour, il faut recompiler et la taille du code source est volumineuse.
- Inconvénient des librairies dynamiques: si la librairie a été supprimée ou est illisible dans l'ordinateur, le programme ne fonctionne plus.

Chemins de recherche Naturellement, lorsqu'on produit une bibliothèque, on ne veut pas que celle-ci se situe uniquement dans le dossier d'un code spécifique. Si nous créeons des bibliothèques, c'est pour les réutiliser dans des programmes différents.

La recherche des *headers* se fait par défaut dans le même dossier que le fichier source et aussi dans les dossiers `/usr/include` et `/usr/local/include`. Les bibliothèques elles sont cherchées dans `/lib`, `/usr/lib/` et `/usr/local/lib`. Pour une librairie qui serait dans un chemin absolu dans l'ordinateur nommé ici `chemin/absolu`, on utilisera l'option `-L` de `gcc` et cela donne la chose suivante.

```
gcc -o main main.o -Lchemin/absolu -lma_lib
```

Pour finir sur le chemin de recherche, la variable `LD_LIBRARY_PATH` de votre système contient les chemins dans lesquels le système ira chercher pour lier les librairies. Utiliser cette variable système est une utilisation avancée de votre ordinateur. C'est à vous de voir comment vous voulez organiser votre environnement de travail.

GMP En cryptographie asymétrique, les entiers sur lesquels on travaille ont des tailles bien plus grandes que 32 ou 64 bits. Par conséquent, même les opérations de base peuvent être assez compliquées à implémenter directement. La bibliothèque GMP pour Gnu Multiple Precision arithmetic library permet de faire des calculs sur des entiers de taille arbitraires en C. Pour la prendre en main, il faut aller sur sa documentation: `https://gmplib.org/manual/`.

Pour aller plus loin Lorsque voius ferez un premier gros projet, vous allez devoir compiler plusieurs (beaucoup de) fois votre programme. Par conséquent, si le projet est gros, on a l'habitude d'utiliser un `Makefile` dans le dossier du projet qui va gérer les étapes de compilation à chaque fois. N'hésitez pas à trainer sur internet au moment voulu pour écrire votre premier `Makefile`.