Logiciel de base
Première année par alternance

Responsable : Christophe Rippert
Christophe.Rippert@Grenoble-INP.fr

Chaine de compilation et d'exécution

Introduction

La compilation d'un programme écrit en C repose sur un schéma très simple (qu'on retrouve dans la plupart des langages impératifs non-interprétés).
Pour comprendre ce qui se passe à chaque étape, prenons comme exemple un fichier `bonjour.c` contenant le code suivant :

```
#include <stdio.h>

int main(void)
{
    puts("Bonjour !");
    return 0;
}
```

Compilation d'un programme écrit en C

Première étape : le pré-processeur

Le pré-processeur (qui s'appelle `cpp` sous Unix) est un module de pré-traitement qui effectue des transformations sur le code écrit avant qu'il ne soit traité par le compilateur.
On ne rentrera pas en détail dans le fonctionnement de ce module, en pratique on n'utilisera quasiment que les directives suivantes (les plus courantes) :

- la directive `#define` sert à définir des constantes et des macros (c'est à dire des bout de code qui seront recopiées tels-quels dans le code C) : dans le cadre de ce cours, on ne s'en servira que pour écrire des définitions de constantes, par exemple `#define TAILLE 10` (notez bien qu'il ne faut pas de point-virgule à la fin de la ligne) ;
- la directive `#include` sert à inclure les fichiers d'en-tête (interfaces) des modules que l'on va utiliser ou écrire (par exemple `#include <stdio.h>` si on veut utiliser la fonction `puts`, etc) : on détaillera plus loin les précautions à prendre quand on écrit soit même un fichier d'en-tête.

Il faut comprendre que le pré-processeur effectue des remplacements textuels (et non sémantiques) : le pré-processeur n'est pas le compilateur, il n'interprète pas ce qu'il manipule. On peut faire le rapprochement avec la fonction Remplacer d'un traitement de texte.
Dans le cas d'une directive `#define NOM val`, le pré-processeur remplace donc tout simplement chaque occurence de la chaine `NOM` par la valeur `val` dans tout le texte du fichier. Dans le cas d'une directive `#include <fichier.h>`, le pré-processeur fait simplement un copier-coller du contenu du fichier `fichier.h` dans le fichier contenant la macro. Ce remplacement est récursif : si le fichier inclut contient lui-même des `#include`, les fichiers qu'il inclut seront aussi inclut dans le fichier initial.
Dans le cas de notre exemple, si on tape la commande `cpp bonjour.c bonjour.i`, le pré-processeur génère un fichier `bonjour.i` qui contient, en plus du code initial que l'on a écrit, tout le contenu du fichier `stdio.h` (plus tout le contenu de tous les fichiers d'en-tête inclus dans `stdio.h`, récursivement). Vous pouvez regarder le contenu du fichier produit : c'est très verbeux car `stdio.h` est un très gros fichier d'en-tête, qui inclut lui-même beaucoup d'autres fichiers `.h` !

Deuxième étape : le compilateur

Le compilateur (le programme `cc` sous Unix) a pour rôle de traduire le code C en code assembleur.
Le compilateur C a pour particularité de faire de la compilation séparée : cela signifie que chaque fichier source est compilé sans regarder les autres fichiers, même s'ils font partie du même programme.
Pour compiler un fichier C, le compilateur a juste besoin de connaitre les prototypes de toutes les fonctions utilisées dans ce fichier, mais il n'a pas besoin de d'avoir à sa disposition le code de ces fonctions externes.
Les prototypes des fonctions sont définies dans des fichiers .h, qui sont inclus par le pré-processeur dans le fichier C qui les utilise via la directive `#include`.
Dans le cas de notre exemple, si on appelle la commande `cc -S bonjour.i` sur le fichier généré par le pré-processeur, on obtient un fichier `bonjour.s` qui contient le même programme mais écrit en langage assembleur (vous pouvez regarder le contenu de ce fichier, on comprendra dans les séances suivantes ce que cela signifie).

Troisième étape : l'assembleur

L'assembleur (appelé `as` sous Unix) prend un fichier écrit en langage assembleur et le traduit en code binaire (langage machine).
On verra dans la suite de ce cours comment on écrit un fichier directement en assembleur, pour l'instant on n'écrira que du C.
Dans le cas de notre exemple, si vous tapez la commande `as -o bonjour.o bonjour.s`, l'assembleur produit un fichier objet `bonjour.o` qui contient le même programme que `bonjour.s`, mais en langage machine (inutile de chercher à lire ce fichier, c'est du code machine, totalement illisible pour un humain !).

Quatrième étape : l'éditeur de liens

L'éditeur de lien (`ld` sous Unix) a pour rôle de fusionner les différents fichiers objets pour produire l'exécutable final. En pratique, cela veut dire qu'il va fusionner :

- les codes contenus dans les différentes fonctions de tous les fichiers objets du programme, pour obtenir une seule grosse zone de code ;
- les données statiques allouées dans tous les fichiers objets du programme, pour obtenir une grosse zone de données statiques commune à tout le programme.

Même dans le cas de notre exemple simpliste, un programme écrit en C utilise toujours d'autres fichiers objets qui font partie de la bibliothèque C standard (appelée en général `libc.a`, un fichier .a étant une archive contenant plusieurs fichiers .o) : dans notre exemple, on utilise le fichier objet contenant le code de la fonction `puts`, ainsi que d'autres fichiers objets permettant de rendre le fichier objet exécutable et qu'on ne détaillera pas (ce type de fichiers s'appellent souvent `crt0.o` sous Unix).
Comme le compilateur C fait de la compilation séparée, l'assembleur a laissé un trou dans le code objet produit, à l'endroit de l'appel à `puts` : ce trou est une indication laissée à l'éditeur de liens pour lui dire : « ici, tu dois insérer un véritable appel à la fonction `puts` que tu trouveras dans un autre fichier objet ».
C'est donc au moment de l'édition de liens qu'on peut obtenir des erreurs du type « la fonction fct est inconnue » si on a essayé d'appeler dans un fichier .c une fonction qui n'existe pas (ou si on a mal écrit le nom d'une fonction existante), et pas au moment de la compilation. C'est aussi ce qui se passe si on essaie de produire un binaire à partir d'un fichier source qui ne contient pas de fonction `main` : cette fonction est appelée par un des fichiers objets inclus automatiquement par l'éditeur de liens, et il n'arrivera pas à la trouver si elle n'est pas dans un des fichiers sources qu'on lui donne.
N'essayez pas de lancer la commande `ld` sur le fichier `bonjour.o` : elle prend des paramètres assez compliqués et dépendants du système pour réussir à produire un exécutable.

Mais beaucoup plus simplement ...

En pratique, il serait assez pénible d'avoir à effectuer à la main toutes ces étapes pour obtenir son exécutable...
La chaine de compilation GCC est en fait beaucoup plus simple à utiliser : elle fourni un programme générique (appelé simplement `gcc`) qui est capable d'appeler chacun des programmes nécessaires dans le bon ordre jusqu'à produire l'exécutable. On appelle couramment ce type de programme un *front-end* en anglais.

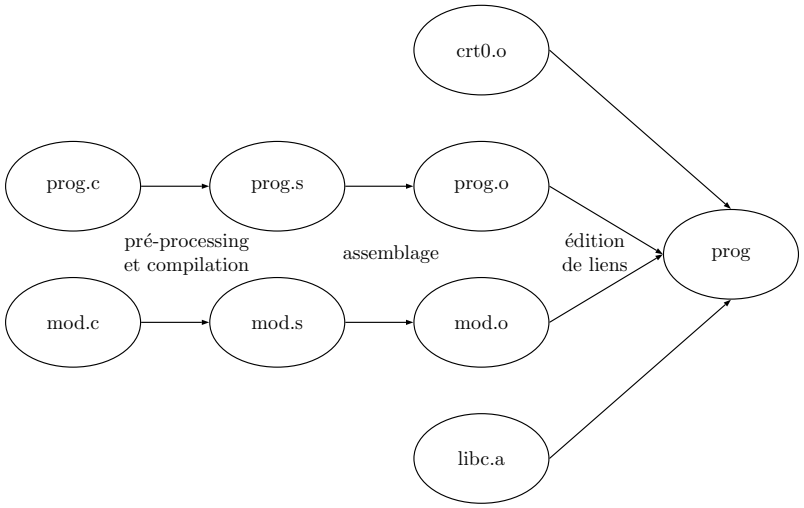
Le programme `gcc` se base simplement sur l'extension du fichier qu'on lui donne pour déterminer quels programmes il doit appeler. Par exemple, si le fichier se termine par :

- .c : il appelle dans l'ordre : le pré-processeur, le compilateur, l'assembleur et l'éditeur de lien ;
- .i : il appelle dans l'ordre : le compilateur, l'assembleur et l'éditeur de liens ;
- .s : il appelle dans l'ordre : l'assembleur et l'éditeur de liens ;
- .o : il appelle simplement l'éditeur de liens (mais en lui passant au passage tous les bons paramètres compliqués qu'on n'a pas détaillé plus haut, ce qui simplifie grandement la vie !).

On peut demander à `gcc` de produire les fichiers intermédiaires manipulés par les différents programmes qu'il appelle si on veut les observer :

- `gcc -E -o bonjour.i bonjour.c` produit le fichier en sortie du pré-processeur ;
- `gcc -S bonjour.c` produit le fichier en sortie du compilateur (après avoir appelé aussi le pré-processeur) ;
- `gcc -c bonjour.o` produit le fichier en sortie de l'assembleur (après avoir appelé aussi le pré-processeur et le compilateur) ;
- `gcc -o bonjour bonjour.c` appelle tous les programmes nécessaires pour produit directement le binaire `bonjour`.

Le schéma ci-dessous résume le processus de compilation dans le cas d'un programme composé d'un programme principal `prog.c` qui utilise un module `mod.c` :



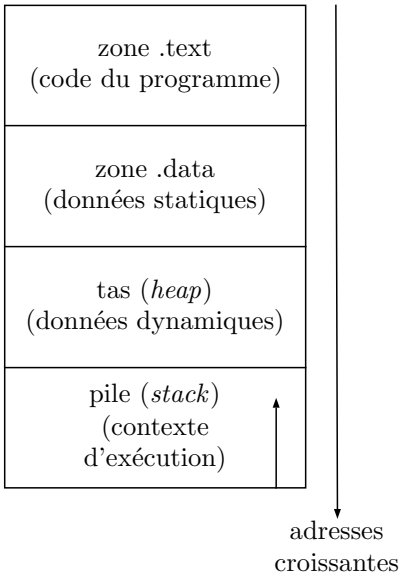
Exécution d'un programme binaire

Une fois qu'on a généré le binaire de notre programme, on peut l'exécuter en tapant la commande `./bonjour` dans un terminal.

Le système d'exploitation fait alors un certain travail que l'on vous détaillera dans le cours de Système de 2A. En résumé, on peut lister ces opérations :

1. allouer un espace mémoire pour le programme et charger le binaire (contenant le code et les données statiques) dans cet espace ;
2. résoudre les liens vers des bibliothèques dynamiques : en effet, en plus des bibliothèques statiques comme la bibliothèque C standard, un programme utilise souvent des bibliothèques partagées qui sont chargées automatiquement par le système (vous avez peut-être déjà entendu parler des fichiers .dll sous Windows ou des fichiers .so sous Unix) : on ne détaillera pas le fonctionnement de ces bibliothèques, il faut juste savoir qu'elles existent ;
3. résoudre les accès mémoires vers des destinations qui n'étaient pas connues à la compilation : dans le cas d'un code dépendant de la position mémoire, il faut traduire les accès pour qu'ils pointent vers des adresses cohérentes avec l'adresse réelle à laquelle le programme a été chargé : là encore, on ne détaillera pas ce mécanisme (on parle le plus souvent de translations) ;
4. lancer l'exécution du code du programme, qui se trouve au début de la zone qui lui a été allouée.

On détaillera beaucoup plus la structure de l'espace mémoire alloué dans la partie sur l'assembleur de ce cours. Pour l'instant, il faut juste retenir ce schéma qui détaille la structure générale du programme en mémoire (on parle en général de processus une fois que le programme a été lancé) :



Les 4 zones d'un processus sont :

- la zone de code (*text*) qui contient le code binaire (les instructions) de notre programme ;
- la zone de données statiques (*data*) qui contient les données allouées statiquement dans le programme (c'est à dire les variables globale en C) ;
- la zone de données dynamiques (appelée aussi tas, *heap* en anglais) où la fonction `malloc` va chercher de la mémoire lorsqu'on fait une allocation dynamique dans le programme ;
- la pile d'exécution (*stack*) qui contient entre autres les variables locales aux différentes fonctions du programme.