INF203 - Compilation

Victor Lambret

2017

Introduction

Le Processeur

- Dans un ordinateur les programmes s'exécutent sur un ou plusieurs processeurs.
- Un processeur est un composant matériel permettant d'effectuer des opérations simples.

1. Lire ou écrire de la mémoire
2. Effectuer des opérations arithmétiques ou logiques
3. Sauter à une instruction précise (pour faire des if, boucles, appels de fonction, etc.)

Une instruction

- Les instructions des processeurs prennent quelques octets. Chaque bit y a une signification particulière.
- Ces instructions sont conçues pour être rapides à décoder et exécuter par le processeur. Elles ne sont pas faites pour être lisibles par un humain !

Exemple d'instruction en binaire :

```
10110000 01100001
```

L'assembleur

- Le binaire n'étant pas facile à manipuler
- On lui associe une représentation textuelle compréhensible par un humain lorsque c'est nécessaire.

Voici l'instruction précédente en assembleur :

```
movb $0x61,%al
```

Pas tellement plus clair...

Programmer en assembleur

Programmer en assembleur est possible, mais ce n'est pas pratique.

- Les instructions étant simples, il en faut beaucoup pour faire des choses compliquées
- Un jeu d'instruction étant spécifique à une famille de processeurs on ne peut pas écrire des programmes portables sur plusieurs machines.
- Au delà de programmes triviaux un développeur ne produit pas du code efficace et fonctionnel en assembleur

Le compilateur

Rôle

- Les développeurs souhaitent programmer dans des langages de haut niveau (C, C++, Java, Ocaml, etc.)
- Les développeurs souhaitent écrire un programme une fois et pouvoir l'exécuter sur des machines différentes
- Un composant doit donc s'occuper de traduire le langage de haut niveau vers les instructions bas niveau d'un processeur : c'est le compilateur !

Le code source

- Le code source est fait pour être compris par un humain.
- Il doit aussi être compréhensible pour un programme : le compilateur. Il est donc plus strict qu'un langage humain.

Analyse lexicale et syntaxique

Afin de transformer un code source en binaire exécutable le compilateur va effectuer différentes étapes :

1. Il analyse le code source pour identifier des éléments de base (mots clefs du langage, noms, etc. . .)
2. Il vérifie que ces éléments de base sont associés dans le respect de la syntaxe du langage

Pour faire le parallèle avec le français :

1. Vérifier que les mots appartiennent bien au dictionnaire
2. Vérifier que la grammaire des phrases est correcte

Représentation interne

- Une fois le code source analysé le compilateur transforme le programme dans son format interne
- Ce format interne est une représentation pratique pour que le compilateur effectue des optimisations du programme
- La représentation interne est encore indépendante du processeur

Création de l'exécutable

- Une fois le programme optimisé, le compilateur va traduire sa représentation interne en un programme exécutable
- Pour cela il fait appel à un traducteur capable de transformer la représentation interne du programme en suite d'instructions pour le processeur ciblé

Compilation ou interprétation ?

L'interpréteur

- L'interpréteur fonctionne a peu prêt comme le compilateur sauf pour la dernière étape.
- Au lieu de créer un exécutable à partir de sa représentation interne il va chercher à l'exécuter directement

Nombre d'appel

- Pour un programme compilé le compilateur est appelé une fois puis le programme est exécuté indépendament.
- Un programme compilé doit donc être entièrement traduit
- Dans le cas d'un programme interprété l'interpréteur analyse le programme à chaque fois qu'il est exécuté.
- Afin de gagner du temps un interpréteur se limite souvent à analyser et exécuter seulement ce qui est utile

Qu'est-ce qui est utile ?

- Quand on exécute un programme il est rare qu'on exécute l'ensemble de ses instructions.
- A chaque exécution le programme va passer dans une partie du code. Par exemple quand on passe dans une branche d'un `if else` l'autre branche n'est pas exécutée.
- Pour un compilateur ce code non utilisé doit compiler. S'il contient une erreur on ne peut donc pas exécuter le programme.
- Pour un interpréteur ce code non utilisé est inutile pour son exécution, s'il contient une erreur cela ne pose pas de problème.

gcc

Utilisation simple

On peut compiler un programme C assez facilement

```
gcc main.c
```

Mais l'exécutable alors généré se nomme `a.out`, ce n'est pas très clair

Nommer l'exécutable de sortie

gcc permet de nommer l'exécutable généré à l'aide de l'option -o :

```
gcc main.c -o main
```

Erreur stupide à éviter

- Faire attention à ne pas mettre le nom d'un fichier source après -o sinon le code source est écrasé.
- L'auto-complétion n'est pas toujours votre amie. (les gestionnaires de version de code oui par contre)

```
gcc main.c -o main.c
gcc -o main.c main
```

Afficher des erreurs

- Le C a été conçu pour de la programmation système. Un développeur système peut faire des choses dangereuses, le langage part du principe qu'il sait ce qu'il fait.
- En pratique, il existe de nombreuses constructions dangereuses mais légales. En voici une :

```
if (result = 2)
    return 1;
else
    return 0;
```

Dans ce code le if teste une affectation, c'est légal en C.

Afficher des erreurs

Deux explications sont ici possibles :

- Le développeur s'est trompé et a mis $=$ au lieu de $==$ (très probable)
- Le développeur souhaite vraiment tester une affectation (peu probable).

La compilation de ce bout de code ne provoque aucune erreur puisque c'est du code C valide :

```
gcc main.c
```

L'option -Wall

En ajoutant l'option `-Wall` (Warnings, all) à gcc il est un peu plus bavard :

```
user@machine:~$ gcc -Wall main.c
main.c: In function ‘main’:
main.c:11:2: warning: suggest parentheses around
   assignment used as truth value [-Wparentheses]
  if (result = 2)
    ^
```

Il dit ici que c'est légal mais qu'il faut rajouter des parenthèses pour confirmer qu'on souhaite tester l'affectation.

L'option -Werror

- Par défaut les warnings ne sont que des avertissements.
- Un adage souvent vérifié en C est : "Warning à la compilation, erreur à l'exécution"
- Cela conduit de nombreux projets à considérer les warnings comme des erreurs.
- On utilise très souvent `-Wall` et `-Werror` ensemble

```
gcc -Wall -Werror main.c
```

L'option –pedantic

- L'option `--pedantic` (attention, le tiret est double ici !) peut-être utilisée en complément de `-Wall`.
- Une remarque de `-Wall` est toujours pertinente à regarder
- Une remarque de `--pedantic` n'est pas toujours un problème. Des fois il fait remarquer que notre code n'est pas compatible avec d'anciennes versions de la norme C.

```
main.c:7:1: warning: C++ style comments are
not allowed in ISO C90
 // commentaire
 ^
```

Quels flags utiliser ?

- Un bon développeur C utilise les flags `-Wall` et `-Werror` afin de détecter les erreurs au plus tôt.
- Il compile souvent pour détecter les problèmes rapidement

Compiler plusieurs fichiers sources

Le problème

- Il est possible de faire des programmes à l'aide d'un unique fichier source.
- Toutefois, à partir d'une certaine complexité il est peu pratique de travailler sur un unique fichier source volumineux. On cherche à découper le programme en modules répondant chacun à un problème.
- Il faut donc être capable de compiler plusieurs fichiers source pour créer le programme.

Première solution

Il est possible d'indiquer plusieurs fichiers C à gcc

```
gcc -Wall -Werror fichier1.c fichier2.c main.c -o main
```

Limites de la première solution

- Pour quelques fichiers ça marche !
- Pour de vrais projets c'est limité.
- Le noyau linux comporte ~18000 fichiers sources C. Un être humain sain d'esprit ne souhaite pas utiliser 18000 fichiers dans une seule commande.
- Si la compilation échoue, on recommence de zéro !
- Si on modifie une ligne dans un fichier on recompile tout !

Seconde solution : la compilation intermédiaire

- Un module C qui ne contient pas de fonction `main` ne peut pas servir à créer un exécutable
- Mais on peut quand même transformer son code source en instructions, on crée alors un fichier objet
- L'option -c de gcc permet de créer un fichier objet :

```
gcc -Wall -Werror -c fichier1.c
```

Un fichier fichier1.o est alors créé

Compiler avec des fichiers objets

On peut ensuite compiler ces fichier objets ensemble pour créer l'exécutable.

```
gcc -Wall -Werror -c fichier1.c
gcc -Wall -Werror -c fichier2.c
gcc -Wall -Werror -c fichier3.c
gcc fichier1.o fichier2.o fichier3.o -o main
```

Avantages

- En cas de modification on recompile juste ce qu'il faut pour faire marcher le programme
- En cas d'erreur on conserve tout ce qui marche

L'édition de lien

Quand on découpe son programme en modules, les modules communiquent entre eux :

- par des appels de fonctions
- par le partage de variables globales

Question : Si un module fait appel à une fonction, comment la compilation peut marcher si elle ne connait pas la fonction ?

Interface d'une fonction

Voici une fonction C

```
int add(int n1, int n2) {
    return n1 + n2;
}
```

- Ce qui est entre accolades c'est le corps de la fonction
- Ce qui est avant l'accolade c'est l'interface à laquelle répond la fonction

Prototype de fonction

Pour appeler une fonction il est inutile de connaître sa mécanique interne. Il est juste nécessaire de savoir comment elle s'utilise :

1. Comment elle s'appelle ?
2. Quels sont ses paramètres d'entrée ?
3. Qu'est-ce qu'elle retourne ?

Le prototype de la fonction répond à ces questions

Prototype de fonction

Le prototype de la fonction précédente est le suivant :

```
int add(int n1, int n2);
```

Utilisation du prototype

Voici le contenu du fichier `main.c` :

```
int add(int n1, int n2);

void main() {
    return add(n1, n2);
}
```

- Ce fichier peut-être compilé en fichier objet car le compilateur dispose de toutes les informations pour compiler le corps de main : il sait comment il devra appeler la fonction.

Problème : où trouver les fonctions ?

Si on compile en fichier objet `math.c` et `main.c` on obtient deux fichiers objets : `math.o` et `main.o`

- main.o utilise la fonction add
- math.o contient la fonction add

`main.o` n'est pas un exécutable complet car il fait appel à un symbole encore inconnu : `add`

Etape finale de compilation

```
gcc add.o main.o -o main
```

- Lors de cette étape, le compilateur cherche à créer un exécutable complet : c'est à dire que tous les symboles sont connus
- Il va donc chercher dans les fichiers objets quels sont les symboles non résolus et quels sont les symboles disponibles
- Si chaque symbole non résolu est lié à son implémentation dans un fichier objet la compilation est réussie

On appelle l'étape de résolution de ces symboles l'édition de lien.