Langage C
Processus de compilation

Sylvain Jubertie

Département Informatique
IUT Orléans

Plan

Compilation séparée

Sources et headers

Un programme C se décompose généralement (idéalement !) en plusieurs fichiers :

- les fichiers sources .c qui contiennent l'implentation des fonctions
- les fichiers headers (en-têtes) .h qui contiennent les prototypes de fonctions, définitions de structures, etc

Cette séparation permet :

- compiler les fichiers sources séparemment : on ne recompile que les fichiers sources modifiés - amélioration du temps de compilation de tout le projet
- on peut ne diffuser que les fichiers headers : distribution du code

Plan

Processus de compilation

Principe

Traduire un code source écrit dans un langage (C, C++, Fortran, Ada, ...) en un code machine exécutable par le processeur.

Processus de compilation

Entrée : fichiers `.c` et `.h` (dans le cas du C)
Sortie : fichier exécutable ou bibliothèque (code machine)

4 phases

1. préprocesseur : traitement sur le texte du fichier source
2. compilation : transformation du source en langage assembleur
3. assemblage : transformation du code assembleur en code machine
4. édition de liens : liaison entre différents codes machines pour la création du programme cible

Plan

Préprocesseur

Opérations du préprocesseur

- suppression des commentaires
- inclusion des headers

```
#include <stdio.h>
```

- expansion des macros

```
#define PI 3.14
```

- exécution des directives du préprocesseur

```
#ifdef LINUX
    ...
#else
    ...
#endif
```

Préprocesseur

Appel du préprocesseur

`gcc -E file.c`

Par défaut, affichage sur la sortie standard.

Redirection par l'option `-o file.i`

Préprocesseur

Code source

```
#include "header1.h"
#define PI 3.14
#define fois2(a) a+a

// Fonction principale
int main() {
  int val = fois2(PI);
#ifdef PLUS5
  val += 5
#else
  val -= 5;
#endif
}
```

Code “préprocessé”

...code de header1.h...

```
int main() {
  int val = 3.14+3.14;
  val -= 5;
}
```

Plan

Compilation

Opération réalisée

Transformation du code en langage assembleur propre à la machine cible
Entrée : `.i`
Sortie : `.s`

Appel du compilateur

`gcc -S file.i`
Par défaut syntaxe AT&T.
Pour la syntaxe Intel option `-masm=intel`

Compilation

Optimisations

Le compilateur peut appliquer différentes optimisations (`man gcc`) lors de la génération du code assembleur :

- inlining de fonctions : `-finline-functions`
- détection du code mort : fonctions non appelées, variables déclarées mais non utilisées
- déroulement de boucles : `-funroll-loops`
- vectorisation automatique pour unités SIMD MMX, SSE, ... : `-msse2` (pas/peu efficace en pratique)
- optimisation en fonction du processeur cible : `-march=pentium4`
- ...

Compilation

Remarques

A partir de cette étape on perd la portabilité du code :

- le code assembleur est propre à l'architecture (x86, PowerPC, ARM, MIPS, ...)
- certaines optimisations sont dépendantes du processeur (taille du cache, profondeur du pipeline, ...)

Mais on gagne en performance !

Plan

Assemblage

Opération réalisée

Transformation du code assembleur en langage machine avec symboles : fichier objet.
Entrée : `.s`
Sortie : `.o`

Appel de l'assembleur

`gcc -c` *file.s*

Assemblage

Fichier objet

Le fichier objet comporte :

- la table des symboles
- le code machine produit à partir de l'assembleur avec des symboles

Symbole

Un symbole est un nom de fonction ou de variable.
Chaque fichier `.c` définit des symboles accessibles par d'autres fichiers `.c` et peut utiliser des symboles définis dans d'autres fichier `.c`.

Assemblage

file1.c

```
int a = 0;
void fct1() {
  cout << "Hello" << endl;
}
```

file2.c

```
void fct2() {
  fct1();
  a++;
}
```

symboles file1.c

Symboles exportés :
a
fct1
Symboles importés :

symboles file2.c

Symboles exportés :
fct2
Symboles importés :
fct1
a

Assemblage

Commandes pour visualiser la table des symboles

- `objdump -t` *file.o*
- `nm` *file.o*

Plan

Edition de liens

Opération réalisée

Assemblage des fichiers objets dans le fichier exécutable.
Vérification de la définition des symboles utilisés à l'aide des tables des symboles des fichiers objets : détection des définitions multiples ou manquantes.
Entrée : fichiers `.o`
Sortie : fichier exécutable

Appel de l'éditeur de lien

`gcc -o` *exec file1.o file2.o ...*