L3 MI — Programmation

Pierre Karpman
pierre.karpman@univ-grenoble-alpes.fr
https://www-ljk.imag.fr/membres/Pierre.Karpman/tea.html

2019–09–11

Compilation multi-fichiers

Retour sur les expressions

Retour sur la conversion de types

Préprocesseur & macros

Principe

Un programme C peut être écrit sur plusieurs fichiers, si :

- Exactement un fichier contient une fonction `main`
- Toute fonction utilisée dans un fichier a été déclarée au préalable *dans ce fichier*
- Toute fonction utilisée a été définie dans un des fichiers (ou une bibliothèque externe)

Structure habituelle

Un programme écrit sur plusieurs fichiers comporte généralement :

- Plusieurs fichiers `.c`
- Pour chaque fichier `file.c`, un fichier `file.h` correspondant qui déclare les fonctions (pas forcément toutes) de `file.c` qui peuvent être visibles dans d'autres fichiers `.c`
  - Un fichier `.c` « inclura » les fichiers `.h` nécessaires (pas forcément tous)

Compilation séparée (pour de faux)

Un programme écrit sur deux fichiers `prog.c`, `file2.c` peut être compilé ainsi :

```
# inutile d'ajouter les fichiers .h éventuels
> cc -o prog prog.c file2.c
```

qui produira un fichier de sortie `prog`

- Chaque fichier est (re)compilé à chaque appel à `cc`
- Relativement peu d'intérêt

Compilation séparée en plusieurs étapes

On procède habituellement comme suit :

1. Compilation séparée des fichiers `.c` en fichiers `.o`

   ```
   > cc -c prog.c
   > cc -c file2.c
   ```

2. Édition des liens des fichiers `.o` pour produire un exécutable

   ```
   > cc -o prog prog.o file2.o
   ```

3. Si seul `prog.c` est modifié, on peut reproduire un exécutable en faisant uniquement

   ```
   > cc -c prog.c
   > cc -o prog prog.o file2.o
   ```

Compilation séparée en plusieurs étapes (suite)

Quelques avantages

- Permet de ne pas systématiquement tout recompiler $\rightsquigarrow$ gain de temps pour les gros programmes
- Permet de fournir un bout de programme uniquement sous forme de `.o` (en pratique on utilisera plutôt un format de bibliothèque partagée)

Quelques inconvénients

- Peut conduire à une multiplication des fichiers
- Complexifie le processus de compilation $\rightsquigarrow$ utilisation d'outils dédiés

Un outils pour la compilation : make

Objectifs de make :

- Permettre une compilation modulaire efficace (faire le strict nécessaire) et facilement configurable
- Passe par un langage simple exprimant des cibles, des dépendances, et des règles de dérivation

Exemple

Dans notre exmple précédent, on avait les dépendances suivantes

- `prog.o` ne peut être créé que si `prog.c` existe
- `file2.o` ne peut être créé que si `file2.c` existe
- `prog` ne peut être créé que si `prog.o` et `file2.o` existent

Ce qui se traduit par le Makefile suivant (attention, il faut utiliser des *tabulations* ⇧) :

```
prog.o: prog.c
        cc -c prog.c

file2.o: file2.c
        cc -c file2.c

prog: prog.o file2.o
        cc -o prog prog.o file2.o
```

Utilisation de make

- En ligne de commande, `> make target` si les instructions sont dans un fichier appelé `Makefile`, ou
  `> make -f otherfile target` sinon
- Avec `target` un des labels se trouvant dans le fichier (par ex. `prog` dans l'exemple précédent)
- Si on ne spécifie pas de `target`, la cible en début du fichier est utilisée par défaut

Quelques cibles classiques

On ajoute souvent quelques « fausses » cibles à un makefile :

```
all: prog1 prog2 prog3 # tous les executables finaux

clean:
        rm *.o

run: prog1
        ./prog1

check:
        # lance des tests...
```

Quelques options de compilation

Nous avons déjà vu `-c` et `-o` ; d'autres options courantes sont :

- `-Wall`, `-Wextra`... : options d'avertissement
- `-std=c89`, `-std=c99`... : options de standard
- `-S` : émission du code assembleur intermédiaire
- `-O2`, `-O3`... : options d'optimisation
- `-mavx`, `-mpclmul`, `-march=native`... : options d'architecture

Ainsi que :

- `-I` : option du préprocesseur permettant d'ajouter des dossiers explorés pour la directive `#include <....>`
- `-L` : option du *linker* permettant d'ajouter des dossiers explorés pour la recherche de bibliothèques partagées
- `-l...` : option du linker spécifiant une bibliothèque (par ex. `-lm` pour la bibliothèque mathématique standard) à utiliser

Make : règles implicites

Pour des programmes C, make dispose d'un certain nombre de règles implicites ; le makefile précédent peut se simplifier en :

```
prog.o: prog.c

file2.o: file2.c

prog: prog.o file2.o
```

Le compilateur utilisé est celui spécifié dans la variable d'environnement `CC`, ou `cc` par défaut

Personnalisation de l'environnement

On peut spécifier dans un makefile la valeur de variables d'environnement, par ex. :

```
# variables utilisées dans les règles implicites
# = : redéfinition complète, += : «augmentation»
CC=clang
CPPFLAGS= -I/home/karpman/sw/soft/include
CFLAGS= -O3 -mavx2 -mavx512vl -maxv512bw
LDFLAGS+= -L/usr/local/lib -lm4ri
# variable quelconque
LOL=cat Makefile

printmk:
        $(LOL)
```

Personnalisation à l'invocation

Les variables d'un Makefile peuvent être redéfinies à l'invocation :

```
# suffisant pour une variable d'environnement prédéfinie
> CC=gcc-9 make
# -e : utilise la définition de l'environnement
> LOL="echo 'hai'" make -e printmk
```

Quelques options pratiques :

```
# affiche les commandes sans les exécuter
> make --dry-run
# exécute les commandes en parallèle (32 au plus)
> make -j 32
```

Compilation multi-fichiers

Retour sur les expressions

Retour sur la conversion de types

Préprocesseur & macros

Dualité expressions/instructions

On dispose en C :

- d'expressions, par ex. `3*x + 15`
- d'instructions, par ex. `x = 3` ou encore `fun(6)`

Une certaine particularité du langage est que :

- certaines expressions ont des effets de bord, par ex. d'affectation comme `i++`
- les instructions retournent une valeur, comme les expressions ; on peut par ex. faire `a = (b = 3)`

Exemple : les opérateurs unaires (pour référence)

- `i++` : la valeur de l'expression est `i`, qui est ensuite incrémenté
- `++i` : i est incrémenté, et la valeur de l'expression est `i`
- `i--` : la valeur de l'expression est `i`, qui est ensuite décrémenté
- `--i` : i est décrémenté, et la valeur de l'expression est `i`

Une conséquence piégeuse classique

- L'instruction d'affectation est aussi une expression, dont la valeur est la quantité affectée (si c'est un nombre)
- Il est donc licite d'écrire `if (x = 1) { ... }` mais le résultat n'est pas forcément celui attendu
- (Un compilateur moderne émettra généralement un avertissement)

La virgule

- Les instructions/expressions s'enchaînent généralement avec un `;`
- On peut aussi combiner plusieurs expressions avec une `,` : toutes les instructions sont exécutées, mais seule la valeur de la dernière expression sera retournée
- Utilisation principalement dans les blocs de `for`, etc. par ex :

```
for (int i = 0, j = 0; i < n; i++, j += i)
{ ... }
```

Compilation multi-fichiers

Retour sur les expressions

Retour sur la conversion de types

Préprocesseur & macros

Types de conversions

Il y a au moins trois types de conversion de type rencontrés couramment en C :

- Entier $\leftrightarrow$ flottant
- Changement de précision, par ex. `uint64_t` vers `uint32_t`
- Entier signé $\leftrightarrow$ non-signé

Toutes peuvent se faire implicitement, et toutes peuvent engendrer une perte de précision ou des erreurs

Entier ↔ flottant

Quelques exemples :

- `double x = 1337 // tout va bien`
- `double x = 0x123456789ABCDEF0 // perte de précision`
- `double x = 12/5 // division entière`
- `int x = 14. // tout va bien`
- `int x = (int)14. // pareil, explicitement`
- `int x = 14./5. // troncation`
- `int x = 123456789123. // overflow/erreur`

Changement de précision, signes

Quelques exemples :

- ```
  uint32_t x = 2;
  uint8_t y = x; // tout va bien
  ```
- ```
  uint32_t x = 257;
  uint8_t y = x; // réduction modulo
  ```
- ```
  int32_t x = 128;
  int8_t y = x; // non défini
  ```
- ```
  int32_t x = 128;
  uint8_t y = x; // tout va bien
  ```
- ```
  int32_t x = -1;
  uint32_t y = x; // renormalisation ; dépend de
  ↪  l'architecture
  ```
  ```

Quelques conseils

- Utilisez des types homogènes (taille, signe) pour éviter les conversions (implicites)
- Si une conversion spécifique est nécessaire faites la explicitement, par ex.

```
int8_t x = (int8_t)(y % 128); // y de type uint8_t
```

- Faites attention aux types des constantes numériques, par ex.
  - `1` est un entier signé « standard »
  - `0x1` est un entier non signé « standard »
  - `1ULL` est un entier non signé `long long`

Compilation multi-fichiers

Retour sur les expressions

Retour sur la conversion de types

Préprocesseur & macros

Préprocesseur

- Le préprocesseur est appelé en début de compilation avant les différentes phases de traduction
- Il exécute notamment les directives `#include`
- Il dispose aussi d'un langage de macros avancé, et de symboles prédéfinis

Quelques points sur les symboles

- On peut définir des symboles, avec ou sans valeurs, par ex. :

```
#define MTHREAD
#define MAX_THREADS 64
```

- On peut faire appel à ces symboles dans tout fichier où ils sont définis ; chaque occurence (isolée, hors d'une chaîne de caractères) de la chaîne `MAX_THREADS` sera remplacée par `64` par le préprocesseur
- On peut annuler la définition d'un symbole pour la suite d'un fichier :

```
#undef SIMD
```

Symboles prédéfinis

- Il existe plusieurs symboles prédéfinis, dont notamment deux utiles pour le débuggage
  - `__LINE__` est remplacé par le numéro de la ligne où il se trouve
  - `__func__` est remplacé par le nom de la fonction où il se trouve (si pertinent)
- Exemple d'utilisation :

```
printf("Coucou from %s @%d\n", __func__, __LINE__);
```

Compilation conditionelle

- Le préprocesseur possède aussi des tests `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`...
- Permet de facilement commenter un gros bloc de code :

```
#if 0
....
#endif
```

- Permet d'activer du code en fonction de contraintes extérieures :

```
#ifdef __SIMD_AVX
....
#elif defined(__SIMD_SSSE3) || defined(__X86_64)
....
#endif
```

Inclusion conditionelle

- En C, on ne doit pas déclarer plusieurs fois une fonction donnée
- Mais avant d'inclure un fichier `.h`, on ne sait pas forcément s'il a déjà été inclu ou pas, ce qui peut mener à des erreurs
- Une solution classique :

```
#ifndef __HAI_H
#define __HAI_H
        void print_hai();
#endif
```

- Nécessite une absence de conflit des symboles, et une « coopération » des développeurs/ses

Définitions en ligne de commande

- On peut définir un symbole, y compris avec une valeur à l'appel au compilateur :

  ```
  cc -DMAX_THREADS=128 p.c
  ```

- Nécessite éventuellement un test *#ifndef* dans le source pour ne pas être écrasé

Macros à arguments

- On peut aussi définir des macros à argument, *qui ne sont pas des fonctions*
- L'expression correspondant au résultat est calculée par le préprocesseur et substituée à l'appel
- Par ex.

```
#define MIN(X,Y) X < Y ? X : Y
...
        MIN(NTHREADS, 12);
...
```

Macros à arguments : quelques pièges

- Dans le cas suivant :

```
#define SQ(X) X*X
...
        SQ(a+b);
        SQ(i++);
...
```

  la première expression sera traduite en `a+b*a+b` qui n'a pas la valeur attendue, et dans la seconde `i++` sera évalué deux fois et `i` incrémenté deux fois
- Le premier cas peut se régler en (sur)parenthésant la définition : `#define SQ(X) ((X)*(X))`
- Le second en s'abstenant d'utiliser des effets de bords dans les arguments