

L3 MI — Programmation

Pierre Karpman

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

2023-09-13

Présentation

Éléments de langage

Compilation (1) : flags et fichiers multiples

Format du cours

- ▶ Cours de programmation adossé au cours d'algorithmique
- ▶ Trois grandes parties : environnement de développement ; gestion mémoire ; programmation proche de la machine
- ▶ 15 heures de TD (1 créneau par semaine)
- ▶ 30 heures de TP (2 créneaux par semaine)
 - ▶ Quelques exercices simples (au début) puis des sujets plus longs sur plusieurs séances
- ▶ Évaluation, 3 notes : contrôle continu sur papier, rendu de TPs, un examen terminal (dates à préciser)

Pour implémenter *efficacement* un algorithme, il est souvent essentiel de :

- ▶ Instancier les types concrets des données (« entier » → `int`?
`unsigned int`? `long int`? `unsigned long int`?
`short int`? `unsigned short int`? `float`? `double`? ...)
- ▶ Instancier des structures de données (tableau trié? table de hachage? arbre binaire de recherche? tas? ...)
- ▶ Gérer l'allocation mémoire
- ▶ Exploiter au mieux le jeu d'instruction et l'architecture du processeur (ou de la carte graphique...)
- ▶ (Traiter efficacement les entrées/sorties)

Langage du cours : C

- ▶ Langage des '70, avec plusieurs révisions (C89, C99, C11, C18, ...)
- ▶ Compilé
- ▶ Avec gestion explicite de la mémoire (allocation, désallocation)
- ▶ Interfaçage aisé avec de nombreux systèmes d'exploitation
- ▶ Permet d'inclure directement du code assembleur
- ▶ Syntaxe proche du Java

Le C, pourquoi ?

- ▶ Permet de rendre explicites les aspects « bas-niveau » de la programmation
- ▶ Donne beaucoup de contrôle sur le code exécuté
- ▶ Bon point d'entrée pour la programmation système ; langage utilisé par ex. dans le noyau Linux
- ▶ Langage utilisé dans beaucoup de bibliothèques (notamment de calcul scientifique)

Présentation

Éléments de langage

Compilation (1) : flags et fichiers multiples

Quelques types de base

- ▶ Absence de type : `void`
- ▶ Entiers (non) signés de taille fixe, définis dans `stdint.h` :
`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`,
`int8_t`, `int16_t`, `int32_t`, `int64_t`
- ▶ Pareil, mais de taille non nécessairement fixe :
(`unsigned`) `char`, `short`, `int`, `long`, `long long`.
Utilisation fortement déconseillée si la taille est importante
- ▶ Nombres flottants à simple et double précision et à précision étendue (non standard) : `float`, `double`, `long double`
- ▶ Un certain nombre d'alias pour des types entiers (notamment en système) : `size_t`, `time_t`, ...

Déclaration de variable

- ▶ Variables simples : suivant la syntaxe `type nom_var;`, par ex.
`uint64_t x;`
- ▶ Tableaux : suivant la syntaxe `type_des_elts nom_var[taille];`. L'indexation des tableaux commence à zéro
- ▶ Plusieurs variables de même type ou pointant vers le même type peuvent être déclarées à la suite, par ex
`double a, b[2];`
- ▶ Les variables peuvent être déclarées à n'importe quel point dans une fonction, et ont pour portée le bloc de déclaration (délimité par `{...}`). Celui-ci peut être *implicite* (par ex. autour d'un `if`).
- ▶ On peut combiner déclaration et affectation, par ex :
`uint8_t x[2] = {0x1E, 0xE7};`
- ▶ Pointeurs, types complexes : dans quelques semaines...

Expressions Booléennes

- ▶ Pas de type Booléen vrai/faux dédié *avant C99* : utilisation d'entiers où $0 \equiv \text{faux}$, $\neq 0 \equiv \text{vrai}$
 - ▶ Toute expression convertible en un type entier est utilisable, par ex. une expression d'affectation !
- ▶ Depuis C99, `stdbool.h` définit un type `bool` et des littéraux `true` et `false`
- ▶ Négation logique avec `!`
- ▶ Test d'égalité avec `==` (à ne pas confondre avec l'affectation `=`), inégalité avec `!=`
- ▶ Opérateurs logiques : `&&` (ET) et `||` (OU); ne pas confondre avec `&` (ET bit à bit) et `|` (OU bit à bit)
 - ▶ ? Les expressions composées sont évaluées avec une stratégie *early abort* : par ex. dans `A && B`, si A est fausse alors B n'est jamais évaluée \Rightarrow attention aux effets de bord !
- ▶ Plus d'exemples plus tard...

Arithmétique entière

On dispose des opérateurs arithmétiques classiques $+$, $-$, $*$, $/$, $\%$, $<$, $<=$, $>$, $>=$

- ▶ La division par $/$ est entière (pas de conversion implicite)
- ▶ Pour les entiers *non signés* de n bits, les calculs sont faits modulo 2^n ; pour les entiers signés, les *overflows* ou *underflows* sont **non définis** (“UB”) (mais des options de compilation existent pour spécifier le comportement, par ex. `-fwrapv` et `-fno-strict-overflow` pour GCC)
- ▶ L’opérateur modulo $\%$ renvoie *un* reste de la division Euclidienne, pas forcément positif :
 - ▶ `29 % 15` renvoie `14`
 - ▶ `-1 % 15` renvoie `-1`
 - ▶ `(29 % 15) == (-1 % 15)` renvoie faux
 - ▶ Mais pas de problème de comparaison si tous les arguments sont positifs par exemple

On dispose de la plupart des opérateurs classiques mais

- ▶ Certains sont remplacés par des fonctions, par ex. `fmod`
- ▶ D'autres fonctions usuelles sont aussi implémentées dans la bibliothèque mathématique standard, par ex. `log`, `cos`, `sqrt`, ...
- ▶ La comparaison de nombres flottants est *délicate* (pas spécifique au C, mais à la norme IEEE 754 sous-jacente) Par ex., la notion d'égalité est mal définie...
- ▶ Plus de détails plus tard...

Les nombres entiers ou flottants peuvent être écrits (entre autres) dans les formats suivants

- ▶ Entiers décimaux signés ou non signés par ex. `-2501`
- ▶ Entiers non signés hexadécimaux, par ex. `0xDEADAF`
- ▶ Entiers non signés en octal par ex. `0777` (un peu vicieux)
- ▶ Entiers `long` ou `long long` signés ou non, par ex. `1ULL`
- ▶ Flottants pointés décimaux, par ex. `107.3`
- ▶ Flottants pointés hexadécimaux, par ex. `0X1.6A09E667F3BCDP+0`

- ▶ `if (cond) { ... } else { ... }`
- ▶ `for (init ; cond ; iter) { ... }`
- ▶ `while (cond) { ... }`
- ▶ `do { ... } while (cond);`
- ▶ `switch ...`

L'interruption de l'exécution d'une boucle peut se faire avec `break`

Note : la délimitation des blocs avec `{}` est facultative si le bloc est composé d'une seule expression

Conditionnelle à trois opérandes

- ▶ Il existe en C une expression à trois opérandes : $A \ ? \ B \ : \ C$ qui vaut B si A est vraie et C sinon
- ▶ Utile notamment pour les affectations conditionnelles, par ex. $a = a \geq 0 \ ? \ a \ : \ -a$; (attention ici aux possibles overflows signés!)

Les deux fonctions C d'entrées/sorties les plus simples sont `printf` et `scanf`

- ▶ `printf` prend comme argument une chaîne de caractères "... " contenant éventuellement un ou plusieurs spécifieurs de format, suivie d'un nombre égal d'expressions du type correspondant. Le résultat est affiché sur la sortie standard `stdout`
- ▶ Par ex. `printf("HAI WURLD\n");` affiche le texte "HAI WURLD" suivi d'un retour à la ligne spécifié par le caractère spécial `'\n'`
- ▶ Par ex. `printf("1 + 1 = %d\n", 1 + 1)` affiche "1 + 1 = 2" suivi d'un retour à la ligne

Quelques exemples

- ▶ `"%d"` : entier signé, affichage décimal
- ▶ `"%u"` : entier non signé, affichage décimal
- ▶ `"%lu"` : entier non signé long via le modificateur `l`, affichage décimal
- ▶ `"%x"` : entier non signé, affichage hexadécimal avec lettres en minuscule
- ▶ `"%02X"` : entier non signé, affichage hexadécimal avec lettres en majuscules avec au moins deux chiffres
- ▶ `"%.2f"` : double en notation pointée décimale avec au moins deux décimales
- ▶ `"%e"` : double en notation scientifique décimale
- ▶ `"%A"` : double en notation pointée hexadécimale majuscule

- ▶ `scanf` prend comme argument une chaîne de caractères contenant un ou plusieurs spécificateurs de format, suivie d'un nombre égal de *pointeurs* de type correspondant. Les données sont lues depuis l'entrée standard `stdin`, interprétées comme spécifié par le format, et stockées aux adresses pointées. Par ex :

```
unsigned a, b;  
if (scanf("%u %x", &a, &b) == 2)...
```

attend deux entiers non signés, l'un écrit en décimal, l'autre en hexadécimal, et stocke le résultat dans `a` et `b`

- ▶ Les formats sont similaires à ceux de `printf`
- ▶ Retourne le nombre d'entrées lues avec succès ou EOF
- ▶ Plus de détails sur les pointeurs dans quelques semaines

Entrées (bis)

- ▶ Lire des entrées de façon fiable et sûre est *difficile* en C!
- ▶ Par ex. il est nécessaire de contrôler la valeur de retour de `scanf` pour éviter d'éventuelles lectures non initialisées (UB)

```
int x;  
if (scanf("%d", &x) != 1) // oui  
    // traitement d'erreur  
else  
    // ...  
...  
int y;  
scanf("%d", &y); // non
```

- ▶ Dans le cadre de ce cours : (presque) pas d'entrées → problématique secondaire (mais importante en général)

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 piègeuse 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)
{ ... }
```

Déclaration de fonctions

- ▶ Une fonction C est (généralement) déclarée suivant la syntaxe `type_retour nom(type_arg1 nom_arg1, ...)`
- ▶ On peut directement faire suivre la déclaration par la définition en faisant suivre le prototype d'un bloc `{}`, ou uniquement la déclarer en terminant le prototype par `;`
- ▶ En général le nombre d'arguments est fixe mais il existe des exceptions comme `printf`
- ▶ Pour pouvoir être utilisée dans un programme, une fonction doit avoir été déclarée « plus haut » dans le fichier

Fonctions : exemple mutuellement récursif

```
unsigned even(unsigned n); // déclaration
unsigned odd(unsigned n) { // déclaration & définition
    if (n == 0)
        return 0;
    else
        return even(n - 1);
}
unsigned even(unsigned n) { // définition
    if (n == 0)
        return 1;
    else
        return odd(n - 1);
}
```

Exercice : comment obtenir le même résultat plus efficacement ?

Politique de passage d'arguments

- ▶ Les arguments de type simple (par ex. `int`) sont passés par *valeur*
- ▶ Les arguments de type tableau sont passés par *référence*

Le *préprocesseur* joue un rôle important dans la compilation d'un programme C. Nous ne verrons aujourd'hui qu'un aspect simple mais essentiel, la directive *#include*

- ▶ *#include <filename>* recopie le contenu du fichier de nom *filename* à l'emplacement de la directive dans le fichier courant. Ce fichier doit se trouver dans un répertoire d'une liste préconfigurée par le compilateur
- ▶ *#include "path/filename"* même comportement, mais pour un fichier accessible depuis le chemin *path*
- ▶ (Dans tous les cas, la directive doit être en début de ligne sans aucune espace ou tabulation)

Inclusions préprocesseur (2)

L'usage principal de `#include` est de copier des fichiers d'en-tête de déclaration de fonctions ou de définition de types, par ex. :

- ▶ `#include <stdio.h>` rend disponible la déclaration des fonctions `printf`, `scanf`, etc.
- ▶ `#include <math.h>` rend disponible la déclaration des fonctions de la bibliothèque mathématique standard
- ▶ `#include <stdint.h>` rend disponible la déclaration des types entiers de taille fixe

La fonction main

La fonction `main` est le point d'entrée d'exécution d'un programme C

- ▶ Tout programme doit comporter un point d'entrée
- ▶ Si le programme est écrit dans un unique fichier, ce fichier doit contenir une fonction `main` de prototype `int main(int argc, char **argv)`. Pour l'instant nous pouvons ignorer les arguments (plus de détails dans quelque temps)
- ▶ Dans le cas d'une compilation séparée en plusieurs fichiers, seul l'un d'entre eux doit fournir un `main` (plus de détails plus tard)

- ▶ La valeur renvoyée par `main` indique si l'exécution du programme s'est bien passée (indiqué par `0` ou `EXIT_SUCCESS`) ou non
- ▶ Sous UNIX, on peut récupérer ce code de retour dans la variable `?`, accessible par ex. avec `> echo $?`
- ▶ Exemple d'un programme complet qui ne fait rien avec succès :

```
int main(int argc, char **argv)
{
    return 0;
}
```

- ▶ Quelques compilateurs C : gcc, clang, icc, tcc, cc (alias)
- ▶ Pour compiler un programme écrit dans un seul fichier prog.c, > cc prog.c est suffisant. Ceci génère un exécutable a.out ; pour donner un nom différent, on peut utiliser l'option -o, par ex. > cc -o prog prog.c

- ▶ Beaucoup de fonctions C sont documentées dans le manuel UNIX dans les sections 2 (appels systèmes) et 3 (bibliothèque standard C)
- ▶ Par ex. `> man 3 printf` ou `> man math` (ou `> man math.h`) sont fort utiles, de même que `> man gcc`
- ▶ Wikipedia (notamment en anglais) recense beaucoup d'information utile (notamment sur la syntaxe, les opérateurs, les types...)
- ▶ La bibliothèque (par ex. *Programmer en langage C*, Delannoy)
- ▶ Ou encore ModernC (en anglais, gratuit), Effective C (en anglais, payant), Hacker's Delight (en anglais, payant, plus édité?), bithacks (en anglais, gratuit)
- ▶ valgrind (pour débbuger, notamment les erreurs mémoire)
- ▶ T-Snippet (pour tester des fragments de code)

Présentation

Éléments de langage

Compilation (1) : flags et fichiers multiples

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)

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
```

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 exemple 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 # cible : dépendance
    cc -c prog.c # commande à exécuter pour
    ↪ produire la cible
```

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

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

Make : gestion des dépendances

La résolution d'une dépendance comme :

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

se fait en exécutant la commande si :

- ▶ le fichier prog.o n'existe pas ;
- ▶ le fichier prog.c est plus récent que prog.o (il faut recompiler prog.o pour prendre en compte des changements éventuels dans prog.c)

Dans le cas contraire (prog.o existe et est plus récent que prog.c), rien n'est fait

Pour forcer une recompilation : > rm prog.o ou > touch prog.c

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 exécutables 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

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)
```

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