Initiation au C
cours n°9

Antoine Miné

École normale supérieure

26 avril 2007

Plan du cours

- compilation séparée,
- modularité,
- bibliothèques `.a` et `.so`,
- recompilation automatique avec `make`.

Compilation séparée

Étapes de la compilation

Compilation = génération d'un programme exécutable à partir d'un fichier `.c`.

Se décompose en plusieurs étapes.

Étapes de la compilation

- pré-traitement (gestion de `#include`, `#define`),
- analyse syntaxique,
- typage,
- génération de code assembleur,
- assemblage,
- édition de liens (ajout de la bibliothèque C).

Programmes utilisés lors de la compilation

Plusieurs programmes interviennent lors de la compilation :

Chaîne de compilation

Pré-traiteur : `cpp`

- pré-traitement (gestion de `#include`, `#define`),

Compilateur C : `cc1`

- analyse syntaxique,
- typage,
- génération de code assembleur,

Assembleur : `as`

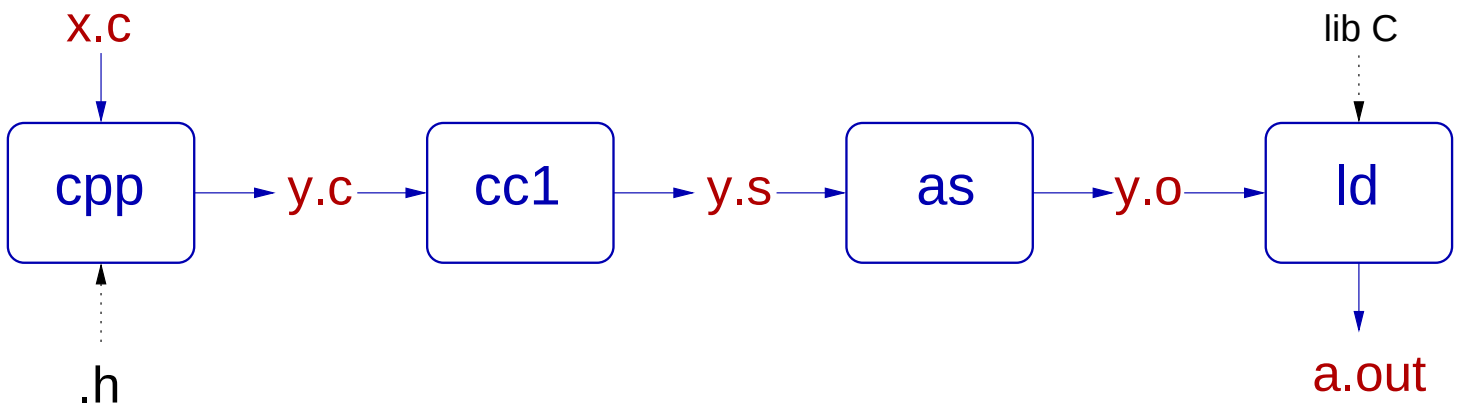
- assemblage,

Éditeur de liens : `ld`

- édition de liens (ajout de la bibliothèque C).

Chaîne de compilation

Ces programmes communiquent par des fichiers intermédiaires :



Types de fichiers :

- `.c` : fichier source C, avant et après pré-traitement,
- `.s` : fichier source assembleur,
- `.o` : fichier objet, (`.obj` sous Windows),
- `a.out` : fichier exécutable (`.exe` sous Windows).

On parlera également des bibliothèques : `.a`, `.so`, `.dll`.

Compilation simplifiée grâce à gcc

En pratique, on n'appelle pas cpp, cc1,... à la main, on utilise gcc.

gcc = interface sur la chaîne de compilation.

Ligne de commande classique

```
$ gcc toto.c -Wall -Wextra
```

Effet :

- pré-traite, compile, assemble et lie,
- détruit les fichiers intermédiaires `.c`, `.s` et `.o` après usage,

$\Longrightarrow$ on obtient automatiquement un exécutable.

Note : gcc détermine les actions à effectuer grâce à l'extension `.c`.

Compilation séparée

Compilation séparée : on décompose en deux étapes

- 'compilation' = pré-traitement + compilation + assemblage,
- édition de liens.

Exemple

```
$ gcc -c toto.c -Wall -Wextra
$ gcc toto.o
```

Effet :

1. `gcc -c` compile et génère un fichier objet `toto.o`,
   (pas d'édition de liens, pas d'exécutable généré)
2. `gcc` lie le `.o` en argument et génère un exécutable `a.out`.
   (pas de compilation, `toto.c` n'est pas examiné)

Note : le fichier `toto.o` n'est pas détruit à la fin du processus.

Modularité

Programmes multi-fichiers

Un programme peut être composé de plusieurs sources `.c`.

Avantages

- facilite l'écriture et la compréhension de gros programmes,
- facilite le travail à plusieurs,
- rend les recompilations plus rapides (compilation séparée),
- permet la modularité et l'abstraction,
- permet la réutilisation dans d'autres projets.

Exemple : noyau Linux 2.6.18

- 8531 fichiers `.c`,
- taille moyenne : 646 lignes, taille totale : 5.5 Mlignes,
- taille maximale : 18227 lignes, taille médiane : 330 lignes.

Compilation multi-fichiers

Méthode simple : en une seule étape.

Exemple

```
$ gcc main.c utils.c mon_print.c -Wall -Wextra
```

Effet :

- pré-traite, compile, assemble et lie tous les `.c`,
- génère un exécutable `a.out`,
- efface tous les `.o` et autres fichiers intermédiaires.

Notes :

- l'ordre des fichiers et des options n'est pas important,
- si un seul `.c` change, `gcc` recompile tout !

Compilation séparée multi-fichiers

Méthode avancée : compilation séparée.

Exemple

```
$ gcc -Wall -Wextra -c main.c
$ gcc -Wall -Wextra -c utils.c
$ gcc -Wall -Wextra -c mon_print.c
$ gcc main.o utils.o mon_print.o
```

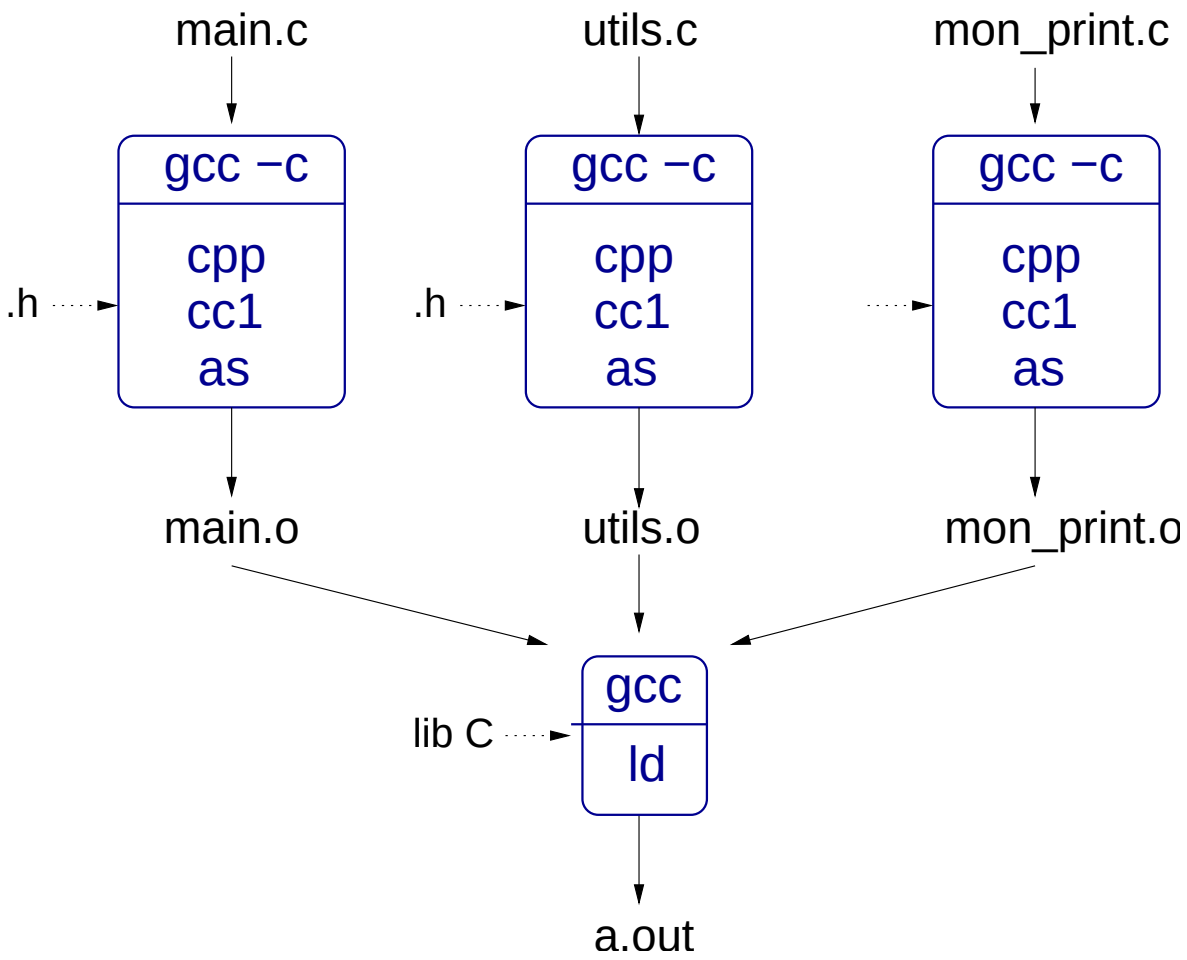
Effet :

- compile chaque `.c` en un `.o` avec `gcc -c`,
- lie tous les `.o` en un exécutable `a.out`.

Attention

Avec `-c`, on ne peut compiler qu'un fichier à la fois !

Diagramme de compilation séparée multi-fichiers



Bénéfice de la compilation séparée

Avantage

Les fichiers `.o` restent disponibles.

En cas de modification du programme :

- seuls les fichiers `.c` modifiés doivent être recompilés,
- l'édition de liens doit également être refaite.

$\Longrightarrow$ gain de temps.

Exemple : après modification de `main.c`

```
$ gcc -Wall -Wextra -c main.c
$ gcc main.o utils.o mon_print.o
```

(automatisation possible grâce à `make`)

Contenu d'un fichier objet

Un fichier objet `.o` contient une **table de symboles** :

- variables globales avec leur taille et valeur d'initialisation,
- fonctions compilées en langage machine.

Le format des fichiers `.o` est :

- binaire, (consultable par `objdump -Ds`),
- non portable, (dépend du type de processeur et d'OS),
- standard pour tous les langages compilés,
  (inter-opérabilité avec le C++, le OCaml, l'assembleur, etc.)
- non typé,
  (types des variables et prototypes des fonctions perdu).

Symboles non définis et édition de liens

Un fichier `.o` n'est pas un programme complet.
Il peut référencer des symboles non définis.

Exemples :

- fonction ou variable dans une bibliothèque,
  (*e.g.* : `printf` dans la bibliothèque standard,
  `sin` dans la bibliothèque mathématique)
- fonction ou variable définie dans un autre fichier `.o`.

Édition de liens :

- pioche dans les `.o` et les bibliothèques passés en argument,
- résout les symboles non définis, (ou indique une erreur)
- comportement indéfini en cas de définitions multiples,
- génère un exécutable (presque) autonome.

Notion d'unité de compilation en C

Unité de compilation C = source `.c` qui sera compilé en un `.o`.

Quelle que soit la méthode de compilation employée,
chaque fichier `.c` est compilé indépendemment en un `.o` :

- pas d'accès aux autres fichiers sources `.c`,
- pas d'accès aux fichiers objets `.o` ni aux bibliothèques.

Les variables et fonctions non définies dans le `.c` se retrouvent comme symboles non définis dans le `.o`...

Conséquence

Pour obtenir un `.o` correct, le `.c` doit préciser les types des variables et prototypes des fonctions utilisées mais non définies :
à défaut de définir, il faut déclarer.

Exemple incorrect

mon_print.c

```
#include <stdio.h>
#include <stdlib.h>
void err(const char*s) {
  printf("%s!!!\n",s);
  exit(1);
}
```

utils.c

```
#include <stdio.h>
void lit(const char*s) {
  FILE* f = fopen(s,"r+");
  if (!f) err("open");
  while (fget(...))
    ...
```

main.c

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  if (argc<2) err("argc");
  lit(argv[1]);
  ...
```

- alerte de typage à la compilation (avec `-Wall`),
- pas d'erreur de liaison, mais programme généré incorrect !

Premier essai de correction : ajout de prototypes

mon_print.c

```
#include <stdio.h>
#include <stdlib.h>
void err(const char*s) {
  printf("%s!!!\n",s);
  exit(1);
}
```

utils.c

```
#include <stdio.h>
void err(const char*s);
void lit(const char*s) {
  FILE* f = fopen(s,"r");
  if (!f) err("open");
  while (fget(...))
    ...
```

main.c

```
#include <stdio.h>
void err(const char*s);
void lit(const char*s);
int main(int argc, char* argv[]) {
  if (argc<2) err("argc");
  lit(argv[1]);
  ...
```

Meilleur correction : utilisation d'une en-tête .h

mon_print.c

```
#include "header.h"

void err(const char*s) {
  printf("%s!!!\n",s);
  exit(1);
}
```

utils.c

```
#include "header.h"

void lit(const char*s) {
  FILE* f = fopen(s,"r");
  if (!f) err("open");
  while (fget(...))
    ...
```

main.c

```
#include "header.h"

int main(int argc, char* argv[])
{
  if (argc<2) err("argc");
  lit(argv[1]);
  ...
```

header.h

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

void err(const char*s);
void lit(const char*s);
```

Utilisation des en-têtes `.h`

Inclusion d'une en-tête utilisateur :

Syntaxe

```
#include "fichier"
```

Effet :

- cherche *fichier* dans le répertoire courant,
- remplace la ligne `#include` par le contenu du fichier.

Similaire à `#include <···>`, sauf pour le répertoire de recherche.

Avantages :

- pas de recopie inutile de prototypes,
- un seul endroit à mettre à jour en cas d'évolution des `.c`,
- une en-tête documentée donne un *résumé* des fonctionnalités.

Contenu d'une en-tête

Une en-tête `.h` est un **fichier C** avec pour conventions tacites :

- l'en-tête n'est pas compilée, elle est #incluse,
- l'en-tête contient :
  - des définitions de types et de macro-instructions,
  - des déclarations de fonctions (prototypes) et variables,
  - l'inclusion d'autres en-têtes, standards ou utilisateurs.

Attention

Ne pas mettre de définitions de fonctions ou de variables dans une en-tête.

Si l'en-tête est incluse par plusieurs `.c`, toute déclaration sera dupliquée dans plusieurs `.o` $\Longrightarrow$ problèmes à l'édition de liens !

Déclarations de variables extern

Pour une déclarer une **variable globale** sans la définir,
on utilise le mot-clé `extern`.

Exemple :

truc.h
```
/* déclaration */
extern int toto;
```

truc.c
```
#include "truc.h"

/* définition */
int toto = 2;
```

main.c
```
#include "truc.h"
...
  /* utilisation */
  toto = 12;
```

Dans l'unité de compilation `truc.c`,
`toto` est déclaré `extern` et défini,
$\Longrightarrow$ permet à `gcc` de vérifier la cohérence.

`extern` est facultatif pour les déclarations de fonctions (prototypes).

Variables globales et fonctions `static`

Symbole statique = local à l'unité de compilation :

- non visible depuis les autres unités de compilation,
- plusieurs unités de compilation peuvent avoir des symboles statiques de même nom,
- fonctionne pour les variables globales et les fonctions.
  (Pour les variables locales, `static` a un autre sens...)

Exemple :

truc.c

```
#include "header.h"
static int nb = 0;
void truc()
{
  nb++;
  ...
```

bidule.c

```
#include "header.h"
static int nb = 12;
void bidule()
{
  nb++;
  ...
```

header.h

```
void truc();
void bidule();
```

Utilisation des en-têtes `.h`

Exemple : noyau Linux 2.6.18

- 8613 fichiers `.h`,
- taille moyenne : 154 lignes, total : 1.3 Mlignes.

L'utilisation des en-têtes est très libre :

- on peut utiliser une en-tête `X.h` pour chaque fichier `X.c`,
  ou une seule en-tête pour tout le projet,
  ou regrouper les déclarations dans des `.h` thématiques.
- l'en-tête peut omettre certains types et déclarations.

Exemple d'application

Rappels sur les types `struct`

Rappel `struct` = types enregistrements.

Définition de type

```
struct s {
   type1   champ1 ;
      ⋮
   typeN   champN ;
} ;
```

Effet : définit un nouveau type de structure

- de nom *s*,
- de champs nommés *champ1* à *champN*,
- les champs ont pour type *type1* à *typeN*.

Types `struct` incomplets

Il est possible de déclarer un type sans le définir.

Déclaration de type

`struct s;`

Déclare l'existence du type `struct s` sans préciser ses champs.

Le type est incomplet :

- on ne peut pas déclarer de variable de type `struct s` :

  e.g. :
  ```
  struct s truc;
  void affiche_s(struct s truc);
  ```

- on peut déclarer une variable de type `struct s *` :

  e.g. :
  ```
  struct s* ptr;
  void affiche_s(struct s* ptr);
  ```

- on ne peut pas déréférencer une variable de type `struct s *` :

  e.g. : `*ptr`, `ptr->titi`.

Application à l'abstraction de types

compte.c

```
struct compte { int num; ... };

void affiche(struct compte* c)
{
  printf("%i\n",c->num);
  ...
}

struct compte* cree()
{
  struct compte* c;
  c = malloc(sizeof(*c));
  ...
  return c;
}
```

compte.h

```
struct compte;

void affiche
  (struct compte* c);

struct compte* cree();
```

main.c

```
#include "compte.h"
...
  struct compte* x;
  x = cree();
  affiche(x);
  free(x);
```

Les bibliothèques

Bibliothèques

Bibliothèque = archive de symboles compilés.

Ressemble aux fichiers objets `.o`, mais :

- regroupe plusieurs unités de compilation dans un seul fichier,
  $\Longrightarrow$ facilite la distribution et l'utilisation,
- optimisée pour être liée de nombreuses fois.

Un fichier bibliothèque :

- a l'extension `.a` (statique) ou `.so` (dynamique),
  (`.dll` sous Windows)
- commence toujours par `lib` (*e.g.* : `libm.so`, `libgmp.a`),
- se trouve généralement dans le répertoire `/usr/lib`.

On expliquera ici l'utilisation des bibliothèques, pas leur création...

Utilisation de bibliothèques prédéfinies

Option d'édition de liens : `-l`*bibli* :

bibli est le nom de la bibliothèque :

- sans l'extension,
- sans le préfixe `lib`.

Cas particulier :
la bibliothèque C (`libc.so`) est toujours liée par défaut.
(option `-nostdlib` pour *ne pas* la lier)

Note : un fichier bibliothèque vient généralement accompagné de son lot d'en-têtes `.h`.

Exemple : utilisation de la bibliothèque mathématique

Bibliothèque mathématique = `libm.so` ou `libm.a` :

Contient la définition des fonctions de `math.h`.
(`sin`, `cos`, `pow`, etc.)

Compilation séparée

```
$ gcc -c exemple.c -Wall -Wextra
$ gcc exemple.o -lm
```

Compilation en une passe

```
$ gcc exemple.c -Wall -Wextra -lm
```

Options usuelles de compilation de gcc

Options de compilation de gcc

`-c`	compilation seule, pas de liaison
`-W`*xxx*	alertes supplémentaires à la compilation *e.g.* : `-Wall`, `-Wextra`
`-O`*x*	optimisation *e.g.* : `-O`, `-O1`, `-O2`, `-O3`, `-Os`
`-g`	ajout d'informations de débogage
`-I` *répertoire*	où `#include` cherche les en-têtes
`-D`*var* `-D`*var*`=`*val*	équivalent à `#define` *var* équivalent à `#define` *var val*

Options usuelles de liaison de gcc

Options de liaison de gcc

`-o` *fichier*	l'exécutable s'appellera *fichier* au lieu de `a.out`
`-l`*lib*	lie avec la bibliothèque *lib*
`-L` *répertoire*	où `-l` cherche les bibliothèques

Exemple de compilation complexe

Exemple : programme utilisant la bibliothèque **gmp** installée chez l'utilisateur : `gmp.h` + `libgmp.a`.

Compilation

```
$ gcc -c proj1.c -Wall -Wextra -I /users/mine/include
$ gcc -c proj2.c -Wall -Wextra -I /users/mine/include
$ gcc -o proj proj1.o proj2.o -L /users/mine/lib -lgmp -lm
```

Note : l'ordre des options `-l` peut être important...

Automatisation de la compilation avec `make`

L'outils make

make = utilitaire de compilation automatique :

- lit une liste de règles de compilation dans un fichier `Makefile`,
- compare les dates de dernière modification des fichiers,
- détermine ceux qu'il faut régénérer,
- n'effectue que le strict minimum d'actions.

Utilisation de make

```
$ make
```

Format du fichier `Makefile`

Forme des règles

but : *prérequis*
    commande

- *but* est le nom du fichier généré,
- *prérequis* est la liste des fichiers dont dépend *but*,
- *commande* est la commande à exécuter pour générer *but*.

Attention l'espacement compte dans les règles :

- *commande* est précédé d'un caractère tabulation,
- les lignes ne doivent pas être coupées.

Exemple de Makefile

Makefile

```
proj: proj1.o proj2.o
	gcc -o proj proj1.o proj2.o -lgmp -lm -L ...

proj1.o: proj1.c
	gcc -c proj1.c -Wall -Wextra -I ...

proj2.o: proj2.c
	gcc -c proj2.c -Wall -Wextra -I ...
```

Effet de `make` : cherche à régénérer `proj`.
Si nécessaire, commence par régénérer `proj1.o` et `proj2.o`.

Note : on peut simplifier le `Makefile` en utilisant des variables et des règles génériques...