Compilation C sous *Unix*

Episode 1: compilation et bibliothèques

Le rôle de gcc

`gcc` est un enchaîneur de passes permettant de compiler un programme. L'option `-v` de `gcc` permet de savoir l'ensemble des actions exécutée par la commande:

```
$ gcc -v single_main.c

Reading specs from /usr/lib/gcc-lib/specs
gcc version egcs-2.91.66 (egcs-1.1.2 release)
cpp -lang-c -v -undef -Di386 -Dlinux single_main.c /tmp/ccAwzMtX.i
GNU CPP version egcs-2.91.66 (egcs-1.1.2 release) (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/include
End of search list.

cc1 /tmp/ccAwzMtX.i -quiet -dumpbase single_main.c -version
    -o /tmp/ccH2w9od.s
GNU C version egcs-2.91.66 (egcs-1.1.2 release) (i486-linux)

as -V -Qy -o /tmp/ccnyD87k.o /tmp/ccH2w9od.s
GNU assembler version 2.9.1 (i486-linux), using BFD version 2.9.1

collect2 -m elf_i386 -dynamic-linker /lib/ld-linux.so.2
    /usr/lib/crt1.o -L/usr/i486-linux/lib /tmp/ccnyD87k.o -lgcc
    -warn-common -lc /usr/lib/crtend.o /usr/lib/crtn.o
```

`gcc` est en fait l'enchaînement de 4 passes:

- un préprocesseur : `cpp`
- un compilateur C : `cc1` (ou `c0` ou `c1`).
- un assembleur : `as`
- un éditeur de lien : `collect2` (ou `ld`).

Le résultat d'une passe est transmis à la passe suivante grâce à un fichier temporaire placé dans `/tmp` puis effacé quand le fichier n'est plus nécessaire. Par défaut la sortie est placée dans le fichier `a.out`.

Le préprocesseur `cpp`

But du préprocesseur: transformation du fichier `.c` en appliquant toutes les instructions du préprocesseur (i.e. celle commençant par `#`) contenues dans le code. Le résultat est le source texte tel qu'il sera compilé.

Les fichiers en entrée du préprocesseur ont pour extension `.i`

Rappel: principales instructions préprocesseur pouvant être utilisées dans un source `C`.

- `#include` : Inclusion de fichiers d'entête (headers).
```
#include <stdio.h>
#include "myprog.h"
```
- `#define` : Définition de constantes symboliques ou de macros.
```
#define METHOD1
#define VMAX 100
#define PLUS(a,b) (a+b)
```
- `#undefine` : L'inverse de `#define`.
- `#ifdef` : Compilation conditionnelle.
```
#ifdef METHOD1
   ...
#else
   ...
#endif
```
ou
```
#ifdef METHOD1
   ...
#endif
```

Options du préprocesseur: les options suivantes peuvent être passées directement à `gcc` mais n'agissent que lors du préprocessing:

- **-E**
  Permet d'analyser le résultat en sortie du préprocesseur. Celui-ci est envoyé sur la sortie standard.
- **-D***var*
  Equivalent à ajouter dans le fichier la ligne suivante
```
#define var
```
  L'écriture **-D***var*=*val* permet de spécifier en plus une valeur à la variable.

Le compilateur `cc1`

But du compilateur: génération du code assembleur à partir du langage `C`.

Option du compilateur : **-S**
Cette option dit à `gcc` de s'arrêter après la phase de compilation en générant un fichier assembleur d'extension `.s`

Exemple:

```
$ ls
single_main.c
$ cat single_main.c
#include <stdio.h>

int
main (int argc, char *argv[])
{
  printf ("hello world\n");

  return 0;
}
$ gcc -S single_main.c
$ ls
single_main.c  single_main.s
$
```

```
$ cat single_main.s
        .file   "single_main.c"
        .version        "01.01"
gcc2_compiled.:
.section        .rodata
.LC0:
        .string "hello world\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        pushl $.LC0
        call printf
        addl $4,%esp
        xorl %eax,%eax
        jmp .L1
        .align 16
.L1:
        movl %ebp,%esp
        popl %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) egcs-2.91.66"
```

L'assembleur `as`

But de l'assembleur: produire un fichier objet (code binaire) contenant le code translatable des fonctions définies à partir du fichier assembleur.

Option de l'assembleur : **`-c`**
Cette option dit à `gcc` de conserver le fichier objet d'extension `.o`

La table des symboles: le fichier objet obtenu contient l'ensemble des symboles définis ainsi que les symboles nécessaires à ceux définis. L'ensemble de ces symboles peut être affiché en utilisant la commande `nm`.

Exemple:

- Source `test.c`

```
#include <stdio.h>
int u;
extern int v;

int Func(int a, int b) {
        return a*u+b*v;
}

void View(int a) {
        printf("Valeur = %d\n",a);
}
```

- Symboles contenus dans le fichier `test.o`

```
$ gcc -c test.c
$ nm test.o
00000000 T Func
00000030 T View
00000000 t gcc2_compiled.
         U printf
00000004 C u
         U v
```

où les lettres majuscules ont la signification suivante: `T`=symbole défini dans le code, `U`=symbole non défini, `C`=symbole commun (global). `t` indique le compilateur utilisé.

L'editeur de liens `ld` (ou `collect2`)

But de l'éditeur de liens: construction d'un code exécutable (dont le nom par défaut est `a.out`) à partir des fichiers objets, des "libraries" (bibliothèques de fonctions translatables), et des runtimes (codes de démarrage situés dans `/usr/lib/*crt*.o`).

Options l'éditeur de liens:

-o *nom*	Donne comme nom *nom* au fichier de sortie plutôt que `a.out`.
-L*rep*	Ajoute le répertoire *rep* à la tête de la liste des répertoires dans lesquelles se trouvent les libraries.
-l*nom*	Utiliser la librarie **lib***nom***.a**.
-s	Suppression des informations sur les symboles.

Exemple 1

```
$ gcc -c single_main.c
$ ld /lib/ld-linux.so.2 /usr/lib/crt1.o single_main.o -lgcc -lc
     /usr/lib/crtend.o /usr/lib/crtn.o -o single_main
```

Attention: `ld` est monopasse et ne va chercher que ce dont il a besoin. Il peut être nécessaire de passer plusieurs fois le même objet (ou library) en cas de dépendance croisée.

Exemple 2

```
$ gcc -c single_main.c
$ ld /usr/lib/crt1.o /usr/lib/crti.o single_main.o
/usr/lib/crt1.o(.text+0x1d): undefined reference to `__libc_start_mai
single_main.o: In function `main':
single_main.o(.text+0xd): undefined reference to `printf'
$ ld /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/libc.a single_main.o
single_main.o: In function `main':
single_main.o(.text+0xd): undefined reference to `printf'
$ ld /usr/lib/crt1.o /usr/lib/crti.o single_main.o /usr/lib/libc.a
$
```

L'enchaînement des passes, avec un `gcc` bien configuré, a pour conséquence que `gcc single_main.c` produit le même résultat que l'exemple 1.

Notions de liens statiques/dynamiques

Bibliothèque statique (static library)

Bibliothèque incluse à l'exécutable lors de l'édition de lien pour permettre son exécution. Elle a pour extension: `.o` ou `.a`

Bibliothèque dynamique (dynamic library)

Bibliothèque non incluse à l'exécutable lors de l'édition de lien. L'exécutable contient alors la référence aux bibliothèques nécessaires à sa propre exécution. Elle a pour extension `.so` et `.sa`

Options de `ld` (et de `gcc`)

`-static` force une édition de lien statique.
`-shared` force une édition de lien dynamique.

Conséquences

- Les liens statiques rendent l'exécutable portable.
- Les liens dynamiques réduisent la taille de l'exécutable.

Exemple

```
$ gcc -c single_main.c
$ ld /usr/lib/crt1.o /usr/lib/crti.o test2.o /usr/lib/libc.a
$ ls -l a.out
-rwxr-xr-x   1 pascal   users      945668 Jan 25 14:20 a.out
$ ld /usr/lib/crt1.o /usr/lib/crti.o test2.o /usr/lib/libc.so
$ ls -l a.out
-rwxr-xr-x   1 pascal   users       10200 Jan 25 14:20 a.out
```

La commande `nm` sur le `a.out` obtenu donne une idée du nombre de fonctions intégrées à l'exécutable.

Rappel de compilation séparée en `C`

Les fonctionnalités de `gcc` et le langage `C` permettent de décomposer un programme comme une suite de modules assemblables grâce au linker.

- Régle 1: En `C`, tout objet (types, variables, constantes symboliques, fonctions) doit être défini **avant** son utilisation.

- Régle 2: Pour tout fichier module *nom*`.c`, un fichier d'entête (ou header) *nom*`.h` existe, et celui-ci contient les informations suivantes **spécifiques au corps du modules**:
  - la définition des constantes symboliques et des macros .
  - la définition des nouveaux types (structure, union, ...).
  - la définition des variables globales liées à ce module (rappel: les variables globales **doivent** être évitées).
  - la définition des fonctions définies dans le fichier `.c` associé sous forme de prototypes.

  On définit le fichier *ext_nom*`.h` comme une copie du fichier *nom*`.h` à la différence que les définitions des variables et des fonctions sont précédées du mot-clé[1] `extern`.

- Régle 3: Toute utilisation dans le module *nom*`.c` d'une fonction définie dans le module *nom2*`.c` **implique** la présence de la ligne: `#include "`*ext_nom2*`.h"` dans le fichier *nom*`.c`.

Remarque: il est possible en pratique de ne gérer qu'un seul fichier `.h` par fichier `.c` en utilisant une astuce du préprocesseur.

[1] Attention La non-utilisation du mot-clé `extern` peut provoquer des phénomènes des confusions lors de la compilation.

Exemple de compilation séparée

Source	Entête	Entête externe

/* mod1.c */
#include "mod1.h"
 	
int func1(int a) {
  c++;
  return a+c;
}
	/* mod1.h */
	int c=0;
	int func1(int);
		/* ext_mod1.h */
		extern int c;
		extern int func1(int);

/* mod2.c */
#include "ext_mod1.h"
#include "mod2.h"
 	
int func2(int a) {
  c--;
  return func1(a)+c;
}
	/* mod2.h */
	int func2(int);
		/* ext_mod2.h */
		extern int func2(int);

/* main.c */
#include <stdio.h>
#include "ext_mod2.h"
 	
int main(int an, char **av) {
  printf("%d\n",func2(an));
}		

Commandes de compilation et de linkage des modules:

```
$ gcc -c mod1.c
$ gcc -c mod2.c
$ gcc -c main.c
$ gcc -o main mod1.o mod2.o main.o
```

Création et gestion de bibliothèques statiques `ar`

La commande `ar` est une commande permettant la gestion complète d'une bibliothèque statique de fonctions compilées à partir de leurs `.o`.

Syntaxe de `ar`:

`ar` **-options** *nom.a* [*obj1.o obj2.o ...*]

Options de la commande `ar`:

`-v`	mode verbose (détail des opérations).
`-r`	insertion ou remplacement des objets spécifiés ayant le même nom.
`-d`	suppression des objets spécifiés.
`-x`	extraction de l'archive des objets spécifiés.
`-ru`	comme `-r` mais seulement si l'objet est plus récent (update).
`-t`	affiche la liste des objets contenus dans l'archive.

Attention: penser à faire le `.h` correspondant au contenu de la bibliothèque.

Exemple:

```
$ ls -l *.o
-rw-r--r-- 1 pascal users 864 Jan 25 19:45 mod1.o
-rw-r--r-- 1 pascal users 908 Jan 25 19:45 mod2.o
$ ar -r modn.a mod1.o mod2.o
$ ar -t modn.a
mod1.o
mod2.o
$ ar -d modn.a mod2.o
$ ar -t modn.a
mod1.o
$ ar -rv modn.a mod1.o mod2.o
r - mod1.o
a - mod2.o
$ ar -tv modn.a
rw-r--r-- 500/100 864 Jan 25 19:45 2000 mod1.o
rw-r--r-- 500/100 908 Jan 25 19:45 2000 mod2.o
```

```
$ nm -s modn.a

Archive index:
c in mod1.o
func1 in mod1.o
func2 in mod2.o

mod1.o:
00000000 D c
00000000 T func1
00000000 t gcc2_compiled.

mod2.o:
         U c
         U func1
00000000 T func2
00000000 t gcc2_compiled.
```

Création de bibliothèques dynamiques

La création d'une bibliothèque dynamique simplement en utilisant des options spécifiques du compilateur `gcc`. Elle se fait en deux étapes:

1. Compilation des objets avec l'option `-fPIC` de `gcc`. L'option `PIC` signifie littéralement "Position Independent Code", et permettra au code d'être chargé pendant l'exécution.

2. Linkage des objets dans la librarie en spécifiant l'option `-shared`.

Exemple

```
$ gcc -fPIC -c mod1.c
$ gcc -fPIC -c mod2.c
$ gcc -shared -o libmodn.so mod1.o mod2.o
$ su
# cp libmodn.so /usr/lib
# ldconfig
# exit
$ gcc -c main.c
$ gcc -o main -lmodn main.o
$ main a b c d e
6
$
```

Notes

- L'inconvénient principal des bibliothèques partagées pour une utilisation locale est que la bibliothèque doit pouvoir être retrouvée par le système au moment de l'exécution.
- Le fichier système `/etc/ld.so.conf` contient la liste des répertoires contenant les bibliothèques partagées (**seules** celles contenues dans ce répertoire sont considérées lors des exécutions). Tout changement de ce fichier doit être suivi par l'exécution de `ldconfig` pour mettre à jour la liste de références aux bibliothèques.