Bibliothèque dynamique sous linux, utilisation sous CodeBlocks

Michel Billaud (michel.billaud@laposte.net)

4 février 2022

Table des matières

- Les notes sont publiées dans https://www.mbillaud.fr/notes/
- Sources dans https://github.com/MichelBillaud/notes-diverses

1 Une question fréquemment posée

Dans de nombreux cours de programmation en C on demande aux élèves

- de travailler avec CodeBlocks (CB),
- d'utiliser une bibliothèque "maison" qu'on me fournit.

Je m'abstiens de discuter le choix de CB . Je vais juste montrer comment le configurer pour arriver, sous CB, à compiler et faire fonctionner un projet qui utilise une bibliothèque.

Mais d'abord, comment on fait une bibliothèque. C'est sous Linux.

2 Une bibliothèque simple

2.1 Objectif

Void comment réaliser proprement (relativement) une bibliothèque dynamique (`.so`) utilisable par d'autres projets.

Ici la bibliothèque contiendra un seul "module"

```c
#include <stdio.h>
#include "foo.h"

void foo(void)
{
    printf("Foo !\n");
}
```

qui certes ne casse pas trois pattes à un canard, mais suffira pour la démonstration.

Pour cela il faut fournir aux projets "utilisateurs" :

- des fichiers d'entête (ici `foo.h`), qu'on met dans un répertoire `include/` ;
- la bibliothèque compilée (`libfoo.so`), dans `lib/`.

Le fichier `foo.h` est sans surprise

```c
#ifndef FOO_H
#define FOO_H

void foo(void);

#endif
```

2.2 Organisation du sous-projet Foo

Tout ce qui concerne la bibliothèque est dans un répertoire `Foo`, qui contient tout ceci après compilation

```
Foo/
|-- Makefile            # fabrication de la bibliothèque
|-- build               # répertoire de travail
|   `-- foo.o
|-- include
|   `-- foo.h           # entête du module       ("exportée")
|-- lib
|   `-- libfoo.so       # bibliotheque compilée  ("exportée")
|-- src
|   `-- foo.c           # source du module
`-- tests
    |-- Makefile
    |-- main            # un programme de test
    |-- main.o
    `-- main.c          # source
```

```
5 directories, 8 files
```

Le programme de test sert à vérifier que la bibliothèque fonctionne.

2.3 Le Makefile

```
CFLAGS   += -fPIC
CPPFLAGS += -Iinclude
LDFLAGS  += -shared

lib/libfoo.so: build/foo.o
    $(LINK.c) $^ -o $@

build/%.o: src/%.c
        $(COMPILE.c) $< -o $@

run-tests:
    (cd tests; make mrproper ; make run)

clean:
    $(RM) *~ */*~ build/*
```

Explications :

- les deux premières cibles fabriquent `lib/libfoo.so` en compilant `src/foo.c` qui inclut `include/foo.h`.
- la troisième fait exécuter le programme qui est dans `tests/`, après avoir lancé sa reconstruction complète. Voir les détails plus loin.

2.4 Le programme de tests

Comporte un source on ne peut plus simple

```
#include "foo.h"

int main()
{
    foo();
}
```

et un `Makefile`

```
CFLAGS = -std=c17 -Wall -Wextra -Werror

FOO_LIB_DIR =     ../lib
FOO_INCLUDE_DIR = ../include

CPPFLAGS += -I$(FOO_INCLUDE_DIR)

LDFLAGS +=  -L$(FOO_LIB_DIR)
LDLIBS  = -lfoo
```

```
main: main.o

run: main
    LD_LIBRARY_PATH=$(FOO_LIB_DIR) ./main

mrproper:
    $(RM) *~ *.o main
```

qui montre les options nécessaires à la compilation (dans `CPPFLAGS`), à l'édition des liens (`LDFLAGS` et `LDLIBS`), ansi que le positionnement de la variable d'environnement `LD_LIBRARY_PATH` pour que le "chargeur dynamique" trouve la bibliothèque au moment de l'exécution.

Cette étape est utile pour les bibliothèques qui n'ont pas été installées par les procédures standard, qui enregistrent (par la commande `ldconfig`, voir manuel) les bibliothèques disponibles dans le fichier `/etc/ld.so.cache`.

3 Utilisation de la bibliothèque *Foo* à partir de CodeBlocks

Essentiellement on souhaite faire la même chose que le programme de test (`tests/main.c`), à savoir appeler le fonction `foo()` de la bibliothèque, mais dans le cadre de l'IDE CodeBlocks.

3.1 Création du projet

On crée un projet (nommé `Projet`, pourquoi pas) **à côté** du répertoire qui contient la bibliothèque. En fait n'importe où, sauf dans le répertoire de la bibliothèque lui-même.

Étapes :

1. lancer CB
2. Menu : File / New / Project… / Console Application / Next / C / Next
3. Donner le titre "`Projet`" et sélectionner le "folder" qui est "au dessus" de la bibliothèque, Next
4. Sélectionner le compilateur (`gcc` pour moi), et Finish.

Dans les sources du projet, un fichier `main.c` a été rempli automatiquement. Normalement vous pouvez le compiler et le faire exécuter. Sinon, vous avez mal installé/configuré CB, et il va falloir vous débrouiller sans moi pour arranger ça, parce que ce n'est pas l'objet de ce document.

3.2 Inclusion de l'entête

Première étape : modifions légèrement ce fichier source pour qu'il inclue l'entête

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

int main()
{
```

```
    printf("Hello world!\n");
    return 0;
}
```

Si on tente de compiler, on a bien sûr un message d'erreur dans le **Build log** :

```
gcc -Wall -g -pedantic -Wextra -Wall  -c .../Projet/main.c -o obj/Debug/main.o
gcc  -o bin/Debug/Projet obj/Debug/main.o
.../Projet/main.c:2:10: fatal error: foo.h: Aucun fichier ou dossier de ce type
    2 | #include "foo.h"
      |          ^~~~~~~
compilation terminated.
Process terminated with status 1 (0 minute(s), 0 second(s))
1 error(s), 0 warning(s) (0 minute(s), 0 second(s))
```

Pour y remédier, il faut indiquer où se trouve le fichier d'entête.

1. menu Project / Build options / **Search Directories** …
2. Dans l'onglet **Compiler**, cliquer sur Add
3. Naviguer jusqu'au répertoire voulu
4. Je choisis "keep relative path", et ça me propose "../Foo/include", ce qui parait correct.
5. Ok.

Maintenant, le programme passe l'étape de la compilation, et de l'exécution. Ne crions pas victoire trop tôt, il n'appelle pas encore la fonction `foo()`. C'est l'étape suivante.

3.3 Appel de la fonction `foo()`

Si on a bien sauvé le projet, l'auto-complétion est active : si on tape `fo` suivi de contrôle-espace, l'éditeur propose de compléter par `foo()` parce qu'il a chargé les définitions de fonctions qui sont dans l'entête. C'est pratique.

Maintenant compilons :

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

int main()
{
    printf("Hello world!\n");
    foo();
    return 0;
}
```

et bam, encore une erreur : le **Build log** indique de quoi il s'agit précisément :

```
gcc  -o bin/Debug/Projet obj/Debug/main.o
/usr/bin/ld : obj/Debug/main.o : dans la fonction « main » :
.../Projet/main.c:7 : référence indéfinie vers « foo »
collect2: error: ld returned 1 exit status
```

La fonction `foo` n'a pas été trouvée lors de l'édition des liens. Ca ne vous surprendra pas parce que vous voyez que la commande `gcc` ne mentionne pas la

bibliothèque. Il va donc falloir arranger ça avec les options d'édition des liens

1. menu Project / Build options / **Search libraries** …
2. Dans l'onglet **Linker**, cliquer sur Add
3. Naviguer jusqu'au répertoire `lib/` de la bibliothèque
4. Open, keep as a relative path, Yes, "`../Foo/lib`", Ok Ok

Après le répertoire de la bibliothèque, la bibliothèque elle-même :

5. menu Project / Build options / **Linker settings** …
6. Other Linker Options : `-lfoo`, Ok

La compilation se fait maintenant sans problème.

3.4 Exécution

L'exécution fonctionne aussi sans nécessiter d'autres interventions.

Ce merveilleux mystère s'explique en consultant le `Build log`. Pour la lisibilité j'ai fragmenté les lignes et remplacé un long chemin d'accès par "…" :

```
Checking for existence: .../Projet/bin/Debug/Projet
Set variable: LD_LIBRARY_PATH=.:.../Foo/lib:
Executing: xterm -T Projet -e /usr/bin/cb_console_runner \
  LD_LIBRARY_PATH=:.:.../Foo/lib .../Projet/bin/Debug/Projet \
  (in .../Projet/.)
```

Pour lancer l'exécution de "Projet" dans une fenêtre terminal (`xterm`), CB a positionné la variable `LD_LIBRARY_PATH` avec le chemin d'accès qu'on lui a donné. Il l'a fait avec un chemin absolu plutôt que relatif, mais peu importe : ça marche.