Compiler un programme C :

quelques informations sur gcc et make …

Joëlle Luter

Table des matières

1. Création d'un exécutable avec gcc

GCC regroupe un ensemble d'outils permettant de générer des programmes exécutables à partir de programmes sources écrits dans différents langages (C, C++, Objective C, Fortran, Java, Ada).
Ce document donne quelques éléments sur la création d'un exécutable à partir de programmes sources écrits en C.
Pour plus d'informations, il faudra se référer à l'aide en ligne de l'environnement de développement ou consulter la page https://gcc.gnu.org/onlinedocs/ du site GCC.

Etapes de génération d'un exécutable

La création d'un exécutable à partir de sources C se déroule en 4 étapes : preprocessing, compilation, assemblage et édition des liens.

En fonction de l'extension du nom de fichier donné en entrée, gcc déterminera à quelle étape commencer le traitement.

Extension	Type de fichier	Etape de début du traitement
fichier.**c**	code C avec préprocessing	préprocessing
fichier.**i**	code C sans préprocessing	compilation
fichier.**s**	code assembleur	assemblage
fichier.**o**	code objet	édition des liens

gcc démarre donc le traitement à l'étape indiquée par le type de fichier d'entrée et continue jusqu'à la création d'un exécutable nommé par défaut `a.out`.

Il est possible de préciser par une option à quelle étape gcc doit arrêter son traitement :

Option	Dernière étape	Fichier généré par défaut (type et nom)
aucune	édition des liens	Code exécutable : a.out
-c	assemblage	Code objet : fichier.o
-S	compilation	Code assembleur : fichier.s
-E	préprocessing	Code source affiché sur la sortie standard

Le nom du fichier généré peut être précisé grâce à l'option **–o** . Par exemple la commande **`gcc prog.c –o prog`** crée un programme exécutable appelé **`prog`** (au lieu de `a.out`).

Quelques options de gcc

GCC offre de nombreuses options contrôlant le préprocesseur, le compilateur, l'assembleur, l'éditeur de liens, le debugger et permettant de préciser les modes d'optimisation, l'architecture cible, etc…

Nous ne citerons ici que les options les plus fréquemment utilisées pour la création d'un exécutable à partir de sources écrits en C.

Option	Rôle
-o	Précision du nom du fichier de sortie
-g	Ajout des informations de debug dans un exécutable
-c	S'arrêter avant l'édition de liens (création d'un fichier objet)
-**Wall**	Affichage de tous les warnings de compilation
-g	Génération des informations de debugging
-I	ajout d'un répertoire à la liste où le préprocesseur ira rechercher les fichiers d'entête (headers) lors des directives de type *#include <nomHeader.h>*
-L	ajout d'un répertoire à la liste où l'éditeur de liens ira rechercher les bibliothèques.
-l	ajout d'une bibliothèque à la liste à prendre en compte lors de l'édition des liens (en plus de la bibliothèque C standard)
-D	Ajout d'une macro du préprocesseur. Revient à ajouter une directive de type *#define* dans le code source.

Exemple d'utilisation :

```
arm-linux-gnueabihf-gcc.exe -I/c/SDL/INSTALLEURS/SDLINC
-L/c/SDL/usr/local/lib *.c -o h3da -lSDL -lbcm_host -lvcos
-lvchiq_arm -lm -lpthread –lrt -D_REENTRANT
```

Cette commande, passée sous Windows :

- effectue la cross-compilation : `arm-linux-gnueabihf-gcc.exe`
- de tous les fichiers source C présents dans le dossier courant : `*.c`
- génère un fichier exécutable nommé *h3da* : `-o h3da`

La directive *#define _REENTRANT* doit être traitée par le préprocesseur : `-D_REENTRANT`

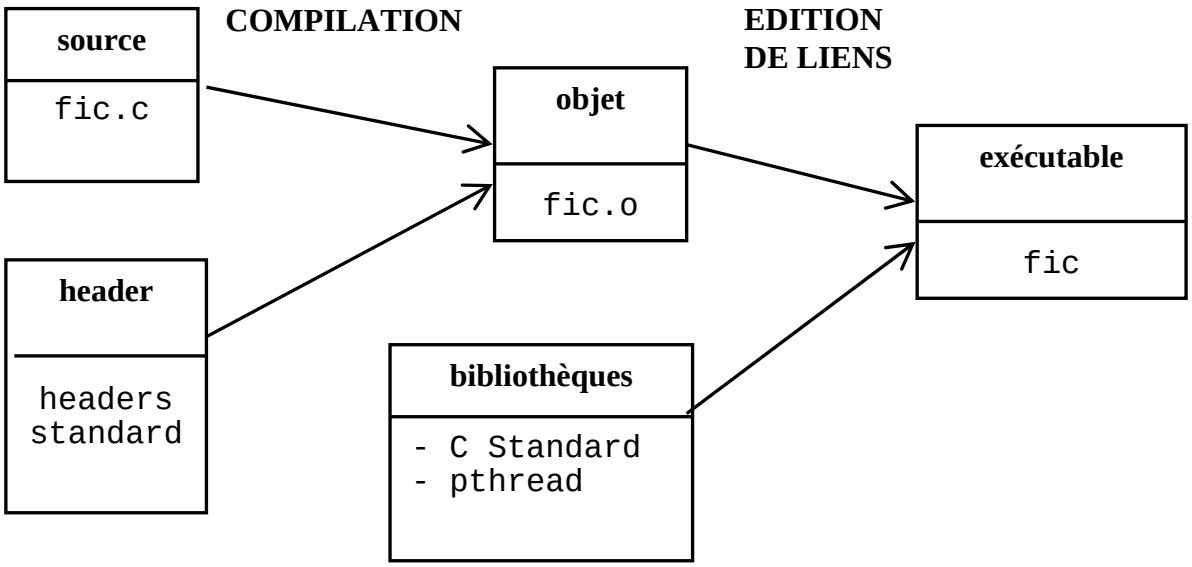
Les fichiers d'entête doivent être recherchés par le préprocesseur dans le dossier standard et dans le dossier *c/SDL/INSTALLEURS/SDLINC* : `-I/c/SDL/INSTALLEURS/SDLINC`

Les bibliothèques doivent être recherchés par le linker dans le dossier standard et dans le dossier */c/SDL/usr/local/lib* : `-L/c/SDL/usr/local/lib`

Les bibliothèques utilisées en plus de la bibliothèque C standard sont : SDL.so, bcm_host.so, vcos.so, vchiq.arm.so, math.so, pthread.so et rt.so : `-lSDL -lbcm_host -lvcos -lvchiq_arm -lm -lpthread –lrt`

Exemple 1 : création d'un exécutable à partir d'un seul fichier source

Graphe de dépendances



Compilation

```
gcc -Wall –c fic.c
```

Edition de liens

```
gcc –lpthread -o fic fic.o
```

ou, en une seule étape

```
gcc –Wall –lpthread –o fic fic.c
```

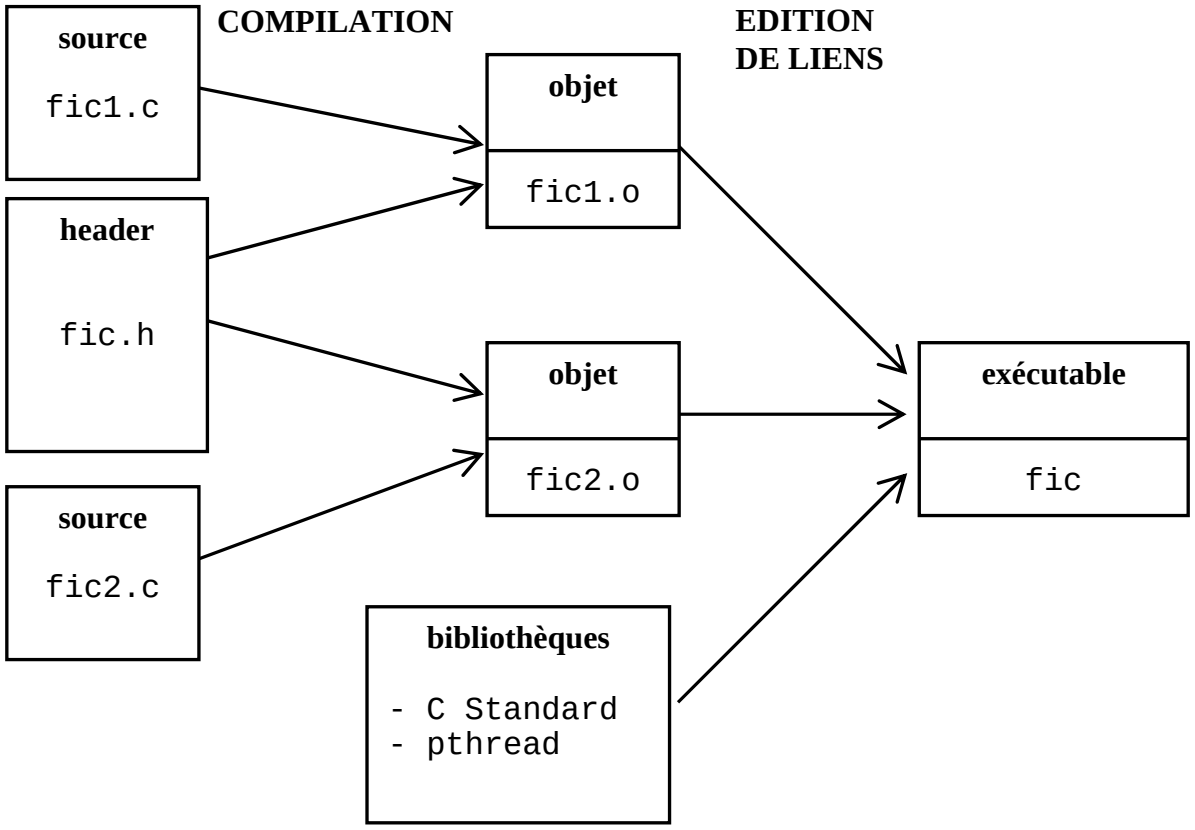
Rappels:

- Les fichiers d'en-tête ("headers", "fichiers .h") sont inclus dans les fichiers source par le préprocesseur. Ils ne sont donc pas explicitement compilés.

- Les bibliothèques sont des regroupements de programmes objets.

Exemple 2 : création d'un exécutable à partir de plusieurs fichiers source

Graphe de dépendances



Compilation

```
gcc -Wall –c fic1.c
gcc -Wall –c fic2.c
```

Edition de liens

```
gcc fic1.o fic2.o –lpthread -o fic
```

ou, en une seule étape

```
gcc –Wall fic1.c fic2.c –lpthread –o fic
```

2. Make

Make est un outil permettant d'effectuer un traitement conditionné par la date de dernière modification des fichiers dont dépend ce traitement.
Il peut donc par exemple être utilisé pour générer un exécutable en ne recompilant que les fichiers source ayant subi une modification depuis la dernière compilation.

Il se sert pour cela de règles de dépendances et de commandes associées qui lui seront précisées dans un fichier appelé **`makefile`** (ou **`Makefile`**).
Ce fichier doit se trouver dans le répertoire d'où est lancée la commande **`make`**.

Chaque règle a la syntaxe suivante :

`cible : liste de dépendances` (autre nom de cible ou de fichier)
`commande à exécuter` (la commande est précédée d'une tabulation)

- La cible peut être une étiquette (phony target, fausse cible) ou un nom de fichier.
- Lorsqu'une règle est évaluée par la commande **`make`**, la liste de dépendances est analysée.
- Toute cible présente dans les dépendances est à son tour évaluée.
- La commande ne sera exécutée que si la cible ne correspond pas à un fichier existant ou que la date de modification d'au moins une dépendance est plus récente que celle de la cible.

Un makefile exprime donc un graphe de dépendances.

Syntaxe de lancement

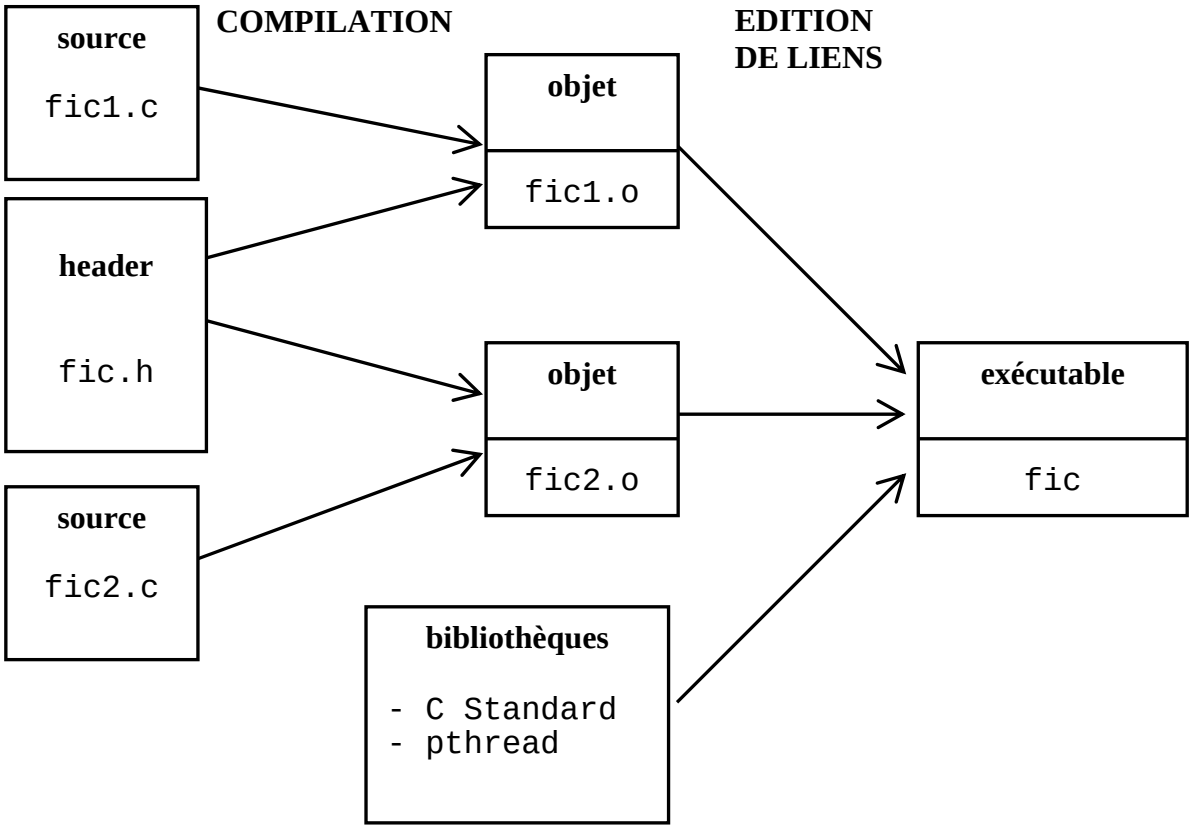
`make`
Exécute la première règle rencontrée dans le fichier `makefile` (ou `Makefile`) ainsi que les règles nécessaires à la résolution de ses dépendances.

`make nom_de_cible`
Exécute la règle de cible `nom_de_cible` présente dans le fichier `makefile` (ou `Makefile`) ainsi que les règles nécessaires à la résolution de ses dépendances. On peut ainsi avoir plusieurs types de traitements présents dans le même makefile et choisir lequel lancer en ligne de commande.

`make –f nom_fichier_makefile`
Exécute la première règle rencontrée dans le fichier `nom_fichier_makefile` ainsi que les règles nécessaires à la résolution de ses dépendances.

Makefile de l'exemple 2

Graphe de dépendances



On voit apparaître dans le graphe 3 règles de dépendances :

- construction de `fic` qui dépend des fichiers `fic1.o` et `fic2.o`.
- construction de `fic1.o` qui dépend des fichiers `fic1.c` et `fic.h`.
- construction de `fic2.o` qui dépend des fichiers `fic2.c` et `fic.h`.

On considérera ici qu'on ne tient pas compte d'éventuelles modifications des fichiers d'entête et des bibliothèques standard. Ces fichiers n'apparaîtront donc pas dans les dépendances.

Fichier makefile traduisant ce graphe de dépendances

```
fic: fic1.o  fic2.o
   gcc fic1.o fic2.o –lpthread –o fic

fic1.o: fic1.c  fic.h
   gcc –Wall –c fic1.c

fic2.o: fic2.c  fic.h
   gcc –Wall –c fic2.c
```

Etiquettes standard

Le nom des cibles n'est pas imposé. Cependant certaines étiquettes ont un nom associé usuellement à un traitement particulier.

Quelques exemples d'étiquettes standards :

- **`all`** : exécute toutes les règles de haut niveau
- **`clean`** : détruit tout les fichiers créés par **`all`**
- **`tar`** : crée une archive contenant les fichiers sources

Le fichier `makefile` précédent devient:

```
all: fic

fic: fic1.o  fic2.o
   gcc fic1.o fic2.o –lpthread –o fic

fic1.o: fic1.c  fic.h
   gcc –Wall –c fic1.c

fic2.o: fic2.c  fic.h
   gcc –Wall –c fic2.c

clean:
   rm –f fic fic1.o fic2.o
```

La commande `make` lance par défaut la première règle rencontrée dans le fichier, ici `all`.

La commande `make clean` lancera uniquement la suppression des fichiers.

Les variables

Il est possible de définir des variables dans un `makefile`. Leur utilité et leur fonctionnement est similaire aux macros du préprocesseur C.
On notera que l'accès au contenu d'une variable se fait par **`$(nom_de_la_variable).`**

Par convention, on utilisera les noms de variables suivants :

Nom	Usage
CC	compilateur à utiliser
LD	linker à utiliser
CFLAGS	options de compilation C
LDFLAGS	options d'édition de liens
OBJS	programmes objet
SRCS	programmes source
LIBS	bibliothèques
INCL	fichiers d'entête
BIN	exécutable à générer par la règle **all**

Le `makefile` de l'exemple 2 devient :

```
CC = gcc
INCL = fic.h
OBJS = fic1.o  fic2.o
BIN = fic
CFLAGS = -Wall
LDFLAGS = -lpthread

all: fic

fic: $(OBJS)
   $(CC) $(OBJS) $(LDFLAGS) –o $(BIN)

fic1.o: fic1.c  $(INCL)
   $(CC) $(CFLAGS) –c fic1.c

fic2.o: fic2.c  $(INCL)
   $(CC) $(CFLAGS) –c fic2.c

clean:
   rm –f $(BIN) $(OBJS)
```

Ces variables permettent d'utiliser une règle prédéfinie par make, qui est que pour créer un fichier de suffixe .o, il faut appliquer la commande **`$(CC) $(CFLAGS) –c`** sur le fichier .c correspondant.
Le `makefile` précédent devient:

```
CC = gcc
INCL = fic.h
OBJS = fic1.o  fic2.o
BIN = fic
CFLAGS = -Wall
LDFLAGS = -lpthread

all: fic

fic: $(OBJS)
   $(CC) $(OBJS) $(LDFLAGS) –o $(BIN)

fic1.o: fic1.c  $(INCL)

fic2.o: fic2.c  $(INCL)

clean:
   rm –f $(BIN) $(OBJS)
```

Variables spéciales

$@ nom de la cible à reconstruire
$* nom de la cible sans suffixe
$< nom de la 1ère dépendance
$? liste des dépendances plus récentes que la cible

```
CC = gcc
INCL = fic.h
OBJS = fic1.o  fic2.o
BIN = fic
CFLAGS = -Wall
LDFLAGS = -lpthread

all: fic

fic: $(OBJS)
   $(CC) $(OBJS) $(LDFLAGS) –o $@

fic1.o: fic1.c  $(INCL)

fic2.o: fic2.c  $(INCL)

clean:
   rm –f $(BIN) $(OBJS)
```