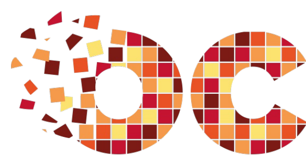
Compilez sous GNU/Linux !

Par kna

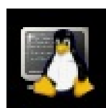


www.openclassrooms.com

Licence Creative Commons 5 2.0
Dernière mise à jour le 4/08/2009

Sommaire

Sommaire ... 2
Compilez sous GNU/Linux ! ... 3
Le compilateur GCC sous Linux ... 3
Un peu de culture ... 3
Compilez avec GCC ... 3
Les arguments de GCC ... 4
Compiler des bibliothèques avec GCC ... 6
Les bibliothèques statiques ... 6
Les bibliothèques dynamiques ... 6
Les noms usuels ... 8
Mettre à jour une bibliothèque ... 8
Compiler avec des bibliothèques tierces ... 9
La commande make et le fichier Makefile - Présentation ... 10
What does make make ? ... 10
La syntaxe du Makefile ... 10
Les directives ... 14
Makefile : un préprocesseur ? ... 15
Les variables ... 15
Les règles d'inférence ... 17
Les conditions ... 18
L'appel récursif ... 19
Pour aller plus loin... ... 21
Des Makefile automatiques : les autotools ... 21
Créer un paquet ... 22
Compiler via make sous Windows ... 23
Partager ... 23

 Compilez sous GNU/Linux !

Par kna

Mise à jour : 04/08/2009

Bonjour à tous, petits développeurs linuxiens.

Vous trouvez votre IDE limité, moche, lourd ? Ou simplement vous aimeriez comprendre qu'est-ce qui se passe derrière quand vous compilez ? Vous êtes un défenseur de la console face aux outils clickodrome ? Vous n'envisagez pas de coder sans vim ou emacs ? Ce tutoriel est fait pour vous !

A l'issue de ce tutoriel, vous serez capable de compiler vos programmes avec une simple console, grâce aux merveilleux outils GNU que sont gcc et make.
Sommaire du tutoriel :

- Le compilateur GCC sous Linux
- Compiler des bibliothèques avec GCC
- La commande make et le fichier Makefile - Présentation
- Makefile : un préprocesseur ?
- Pour aller plus loin...

Le compilateur GCC sous Linux

Un peu de culture

GCC signifie *GNU Compiler Collection* , autrefois un compilateur C ("GNU C Compiler") devenu multilangage, est un compilateur sous Linux, permettant de compiler du C, du C++, du Java, et j'en passe...
Pour plus de détails, : gcc sur wikipedia.

Je prendrai comme exemple la compilation en C sous Debian & consorts, mais vous pourrez utiliser les informations de ce tuto pour d'autres langages et sous d'autres distributions !

Compilez avec GCC

Installer GCC

Tout d'abord, avant d'entrer dans le vif du sujet, il faut que vous ayez gcc installé !

Sous Ubuntu ou Debian, vous pouvez faire (en root) :

Code : Console

```
apt-get install build-essential
```

Ceci installera gcc (pour langage C) et g++ (pour langage C++), mais aussi make qui sera utile pour la suite !

Pour les autres, vous avez sûrement un gestionnaire de paquets. Recherchez donc les paquets gcc et g++ (pour java, prenez gcj, pour fortran prenez g77 ou gfortran).

Utiliser gcc

gcc s'utilise en console (c'est en trois lettres, ça commence par g mais ce n'est pas gui !). C'est une commande qui prend des arguments :
gcc [*arguments*] [*fichiers à compiler*] (**-o** *fichier de sortie*)

Sans arguments, gcc fait toute la compilation jusqu'à l'exécutable.
L'argument -o permettra de spécifier le nom du fichier de sortie, quels que soient les autres arguments, et donc quel que soit son type !

Imaginons que notre projet soit constitué d'un main.c gérant le programme et d'un fonctions.c (avec fonctions.h) pour les fonctions qu'il appelle !

Code : Console

```
gcc main.c fonctions.c -o Programme
```

Ceci créera l'exécutable Programme basé sur main.c et fonctions.c.

Lorsque les headers sont dans le même répertoire que les sources, il est inutile de les préciser.

Et en C++ ?
Eh bien c'est pareil, sauf que vous utiliserez g++ au lieu de gcc :

Code : Console

```
g++ main.cpp fonctions.cpp -o Programme
```

Pour lancer le programme, il suffit de taper :

Code : Console

```
./Programme
```

Les arguments de GCC

La liste est non-exhaustive. Le nombre d'arguments de la fonction gcc est astronomique ! Je ne liste ici que les plus courants (et j'en oublie probablement).

-c : indique à GCC de ne pas linker.
Vous obtiendrez alors un fichier objet (.o).

Exemple :

Code : Console

```
gcc -c main.c -o main.o
```

-v : mode verbeux.
Affiche plus de détails.
Si vous voulez tout savoir : les infos sur le processus de compilation, le MD5 de la cible obtenue, ...

-I : spécifie le ou les répertoire(s) des headers.
Indique à gcc où rechercher les fichiers .h.

Par exemple, si vos .c sont dans le dossier sources et vos .h sont dans le dossier headers, il vous faudra entrer :

Code : Console

```
gcc sources/*.c -I headers -o Programme
```

-pipe : avec cette option, GCC ne génère pas de fichier temporaire entre chaque étape.
Utilise des tubes (comme les « | » en shell) de façon à ne générer que le fichier cible.

-Ox : indique le degré d'optimisation (x varie de 1 a 6).
gcc optimise votre code, c'est-à-dire qu'il va modifier le code source de façon à ce que le programme ait le même résultat par le chemin le plus court. En somme, il dit « t'aurais mieux fait de programmer cette fonction comme ça ! ».
Plus le degré est fort, plus l'optimisation est bonne, mais plus la compilation est longue et gourmande en mémoire !

Remplacez **x** par **s** (**-Os**) et gcc organisera votre code de manière à ce qu'il soit le plus court possible.

-w : supprime tous les avertissements.
Je suis un tueur en programmation, je suis plus un gamin ! Na !

-W : GCC plus exigeant quant aux warnings.
Si votre programme se compile mais ne fait pas ce qu'il devrait faire, essayez avec cet argument !

-Wall : GCC est encore plus exigeant !
La solution ultime !
Attention tout de même : avec cette option, le compilateur peut trouver des erreurs dans des bibliothèques externes qu'on utilise !

-Werror : tous les warnings deviennent des erreurs.
Seul le code parfait compilera !

-g : mode Debugger.
Ajoute des informations de déboggage à l'exécutable. Ces informations peuvent être utilisées par le deboggueur GDB. A noter que vous trouverez un tutoriel sur GDB sur ce site : http://www.siteduzero.com/tutoriel-3-3 [...] avec-gdb.html

-o [fichier] : spécifie la cible.
J'en ai parlé plus haut, sachez simplement que le fichier cible peut être aussi bien un exécutable qu'un fichier objet ou que-sais-je, suivant les autres arguments.
Enfin, c'est un argument comme un autre, vous n'êtes pas obligés de le mettre à la fin comme je l'ai fait :

Code : Console

```
gcc -o main.o -c main.c
```

équivaut à :

Code : Console

```
gcc -c main.c -o main.o
```

Personnellement, je le mets à la fin car ça me semble plus clair, mais c'est vous qui choisissez !

Si vous ne spécifiez pas de cible, la cible par défaut sera a.out dans le cas d'un exécutable, (*nom du fichier c*).o dans le cas d'un objet.

C'est tout ?

Vous en voulez plus ? Tapez `man gcc` en console.
Pour les anglophobes, vous avez la même chose en français en cliquant ici.
Tous les arguments y sont... De quoi perdre la tête !

Sous debian, pour avoir le man, il vous faut le paquet gcc-doc dans les dépôts non-free (licence GFDL).

Compiler des bibliothèques avec GCC

Le compilateur gcc vous permet aussi de compiler les bibliothèques (certains disent "librairies", car "library" en anglais).

Il existe deux types de bibliothèques : les bibliothèques statiques et les bibliothèques dynamiques.

Les bibliothèques statiques

Une bibliothèque statique (généralement d'extension .a) est une bibliothèque qui sera intégrée à l'exécutable lors de la compilation.
L'avantage est que l'exécutable produit est autonome et ne nécessite rien de plus pour fonctionner. La bibliothèque se comporte comme un autre fichier objet.

Si vous voulez faire une bibliothèque statique avec machin.c et machin.h, il suffit de faire :

Code : Console

```
gcc -c machin.c -o machin.o
ar -q libmachin.a machin.o
```

La première commande crée le fichier objet (ça, on connait). La commande ar archive tout simplement ce fichier.
Pour plus de détails, lisez man ar en français.

La bibliothèque se liera comme n'importe quel fichier objet :

Code : Console

```
gcc bidule1.o bidule2.o bidule3.o libmachin.a -o Programme
```

Les bibliothèques dynamiques

Les bibliothèques dynamiques - .so (Sharing Object) sous Linux ou .dll (Dynamic Link Library) sous Windows - sont des bibliothèques qui ne sont pas intégrées à l'exécutable lors de l'édition de liens. L'exécutable appelle alors la bibliothèque pour exécuter les fonctions.
Il en ressort plusieurs avantages :

- si la bibliothèque est utilisée par plusieurs programmes, elle n'est chargée qu'une fois en mémoire ;
- l'exécutable est plus léger ;
- on peut la mettre à jour sans recompiler le programme (à condition de ne pas modifier le header).

En revanche, il faudra fournir la bibliothèque, sans quoi le programme ne pourra pas fonctionner.

Pour créer une bibliothèque dynamique, il faut utiliser

Code : Console

```
gcc -c -fPIC truc -o truc.o
gcc -shared -fPIC truc.o -o libtruc.so
```

L'option -fPIC (Position Independent Code) compile sans indiquer d'adresse mémoire dans le code, car en fonction du programme qui l'utilisera, les adresses pourront être différentes. Ceci évitera des conflits entre les bibliothèques.

L'option -shared indique que c'est une bibliothèque partagée (autre façon de dire dynamique).

On compilera encore de la même manière (avec -fPIC en plus, par sécurité) :

Code : Console

```
gcc -fPIC bidule1.o bidule2.o bidule3.o libtruc.so -o Programme
```

Hélas, l'exécutable n'est pas prêt à être utilisé. En effet, lorsqu'un programme appelle une bibliothèque, Linux cherche si la bibliothèque est installée dans un répertoire par défaut, mais pas dans le répertoire courant.

Pour pouvoir utiliser ce programme, il y a 2 solutions :

- LA MAUVAISE : copier la bibliothèque dans /lib ou dans /usr/lib

  Code : Console

  ```
  cp libtruc.so /lib
  ```

  (nécessite d'être root)
  On laisse ces répertoires aux programmes fournis et suivis par la distribution Linux. On ne pollue pas le système avec nos programmes personnels.
  Pour les programmes personnels (ou du voisin, ou que-sais-je), on utilise le répertoire /usr/local.
  On copiera donc les bibliothèques dans /usr/local/lib !
  Hélas, ce répertoire ne sera pas trouvé non plus par le système lorsque le programme appellera la bibliothèque !
  On utilisera donc :
- LA BONNE MÉTHODE : utiliser la variable LD_LIBRARY_PATH. Cette variable donne les autres chemins où sont appelés les bibliothèques. Elle est sous la forme chemin1:chemin2:chemin3. Vous pouvez voir sa valeur en utilisant :

  Code : Console

  ```
  echo $LD_LIBRARY_PATH
  ```

  Généralement par défaut, elle n'est pas définie, la commande affichera alors un blanc. Pour ajouter un dossier, il suffit de mettre

  Code : Console

  ```
  export LD_LIBRARY_PATH=chemin:$LD_LIBRARY_PATH
  ```

  Pour indiquer le répertoire courant, utilisez le point (.).

Si vous ne voulez pas refaire la manip après chaque démarrage du système, éditez le fichier /home/moi/.profile (n'affecte que l'utilisateur "moi") ou bien /etc/profile (affecte tous les utilisateurs, nécessite donc d'être root pour le modifier).
Rajoutez la ligne export LD_LIBRARY_PATH=chemin:$LD_LIBRARY_PATH, et voilà !

Remarquez, vous pouvez le faire en une ligne de commande :

Code : Console

```
echo export LD_LIBRARY_PATH=chemin:$LD_LIBRARY_PATH >> ~/.profile
```

Cependant, ceci rajoutera la ligne à la fin du fichier. Les fichiers .profile (s'il existe) et /etc/profile étant structurés, il est préférable

de les éditer à la main.

Les noms usuels

J'ai donné comme nom à mes exemples libmachin.a et libtruc.so. J'aurais très bien plus les appeler machin.lib et truc.dll : on est sous Linux, tout est permis !

Cependant, on utilisera plutôt par convention, des noms du même type que mes exemples. De plus, cela vous permettra de compiler en utilisant les options -l et -L :

- -lmachin pour libmachin.a ou -ltruc pour libtruc.so,
  -l rajoute automatiquement lib devant et .a ou .so derrière ;
- -Lchemin pour indiquer le chemin ;
  non nécessaire si le chemin est /lib ou /usr/lib (mais ça ne devrait pas !) ou dans LD_LIBRARY_PATH.

Exemple :

Code : Console

```
gcc -fPIC bidule.o -L. -lbidule -o Programme
```

gcc créera l'exécutable Programme en liant l'objet bidule.o avec la bibliothèque libbidule.so (ou libbidule.a s'il ne trouve pas le so), située dans le répertoire courant.

Mettre à jour une bibliothèque

ld -soname

Maintenant, vous savez faire des bibliothèques dynamiques ; mais comment différencier deux versions de la même bibliothèque et faire en sorte que la nouvelle version fonctionne avec un programme compilé avec la première version ?

Par exemple, je compile le programme bidule avec la bibliothèque libtruc.so.1.1. Plus tard, je veux installer (pour un autre programme) libtruc.so.1.2. Mais je veux que le programme bidule puisse l'utiliser.

On utilise pour cela l'option -Wl,-soname, pour définir le lien libtruc.so.1 comme cela :

Code : Console

```
gcc -Wl,-soname, libtruc.so.1 -o libtruc.so.1.1
```

Ceci créera la bibliothèque libtruc.so.1.1, à partir de libtruc.so.1, et cette bibliothèque sera reconnue comme libtruc.so.1 !

Quelques explications

Je ne vous l'avais pas dit, mais gcc ne fait pas l'édition de liens ! Il appelle pour cela la commande ld !
Pour passer des options à ld avec gcc on utilise -Wl,option,

L'option -soname libtruc.so.1 permettra à l'OS de reconnaître une bibliothèque comme s'appelant libtruc.so.1.
Le nom *libtruc.so.1* sera intégré dans la cible (libtruc.so.1.1) et sera lu par ldconfig.

ldconfig

ldconfig crée des liens symboliques entre les -soname et les bibliothèques concernées.

Il inspecte les bibliothèques dans les emplacements suivants :

- /lib
- /usr/lib
- les chemins indiqués dans /etc/ld.so.conf (vous pouvez l'éditer si vous êtes root) ;
- les chemins de LD_LIBRARY_PATH.

Il crée pour chaque bibliothèque un lien ayant comme nom le -soname et ayant comme cible la bibliothèque. Si deux bibliothèques ont le même -soname, il fera le lien vers la version la plus récente.

Faites attention aux points suivants :

- le nom de la bibliothèque doit être du type **libnom.so.***version* pour être reconnue par ldconfig ;
- si vous éditez /etc/ld.so.conf, vous n'avez plus besoin de LD_LIBRARY_PATH ;
- une bibliothèque de ce type ne sera pas reconnue avec l'option -l, qui nécessite un type libnom.so (sans rien derrière) ;
- il existe certaines conventions : pour mettre à jour une bibliothèque il faut que les prototypes des fonctions soient identiques, si vous rajoutez une fonction à cette bibliothèque, il faudra mettre à votre bibliothèque -soname libtruc.so.2 ! Elle sera alors différente de libtruc.so.1 !
- il faudra exécuter ldconfig en console avant de lancer le programme.

Note : la commande ldd Programme liste les bibliothèques utilisées par votre programme.

Compiler avec des bibliothèques tierces

Si vous avez installé une bibliothèque tierce (GTK ou SDL par exemple), vous vous demandez sûrement où se trouve les fameux .so !
La réponse est qu'il n'y a pas à se poser la question !
On utilise dans ce cas la commande pkg-config.

pkg-config est un utilitaire qui donne des informations sur les bibliothèques installées.

Ainsi :

Code : Console

```
pkg-config --cflags [bibliothèque]
```

donne la liste des dossiers des headers de [bibliothèque]

Code : Console

```
pkg-config --libs [bibliothèque]
```

donne la liste des fichiers de [bibliothèque]

Certaines bibliothèques fournissent un outil spécifique pour elle-mêmes. Par exemple, avec sdl, on peut utiliser sdl-config.
sdl-config --cflags équivaut à pkg-config --cflags sdl

pkg-config donne la liste telle qu'elle peut être comprise par gcc (avec des -I et des -l). On peut donc l'intégrer directement à une commande gcc.
Donc, pour compiler un programme main.c qui utilise GTK+, on fera :

Code : Console

```
gcc -c main.c $(pkg-config --cflags gtk+-2.0) -o main.o
gcc main.o $(pkg-config --libs gtk+-2.0) -o Programme
```

(Le « **$** » permet de renvoyer la valeur de ce qu'il y a entre parenthèses à la commande)

On peut aussi le faire en une seule commande :

Code : Console

```
gcc main.c $(pkg-config --cflags --libs gtk+-2.0) -o Programme
```

Et comment je fais pour savoir que c'est gtk+-2.0 et non gtk2.0 ou encore gtk+ ?

pkg-config cherche dans /usr/lib/pkgconfig le fichier [bibliothèque].pc.
Il vous suffit de trouver le fichier de votre bibliothèque dans ce dossier.

Vous pouvez même rajouter vos propres .pc si vous constuisez des bibliothèques, en vous inspirant de la syntaxe des fichiers présents sur votre système. Vous pouvez mettre vos .pc dans un autre dossier (par exemple /usr/local/lib/pkgconfig), il faut pour celà définir le chemin dans la variable PKG_CONFIG_PATH.

Petit problème de compréhension ? Envie d'en savoir plus ?
man ldd
man ld.so
man ldconfig
man pkg-config (désolé, je ne l'ai pas trouvé en français celui-là)

Ça y est ! Vous maîtrisez gcc !

Ouah ! Mais j'ai un big programme à compiler, moi !
Faut que je me tape toutes ces commandes ?

Heureusement non, ce serait le comble du programmeur de taper sans cesse les mêmes commandes !
Il existe un outil permettant d'automatiser un peu tout ça. J'ai nommé : **make** !

La commande make et le fichier Makefile - Présentation

What does make make ?

Vous commencez à vous demander ce que sont ces *Makefiles* et ces *makes* que vous voyez partout.
Vous vous y êtes peut-être confrontés si vous avez téléchargé des programmes sous Linux au format tar.gz.

La commande make va tout simplement faire une série de commandes, située dans un fichier appelé Makefile.
En gros, ça ressemble au bon vieux script bash (à quelques exceptions près) :

- la syntaxe de make sera plus adaptée aux commandes de compilation ;
- make effectue les commandes en fonction des fichiers les plus récents ;
- make gère les dépendances ;
- et j'en passe...

Comment ? Eh bien c'est ce qu'on va voir !

La syntaxe du Makefile

Le fichier Makefile est celui dans lequel on met les commandes qui seront exécutées par make.
Vous devez donc avoir make installé (relancez votre gestionnaire de paquets si nécessaire).

Les règles

La structure de base du Makefile est :

Code : Bash

```
cible: dependances
        commandes
        ...
```

où :

- cible est le nom du fichier créé (par exemple, Programme) ;
- dependances représente la liste des fichiers (ou règles) nécessaires à la construction de la cible ;
- commandes représente les commandes à effectuer pour créer la cible.

Notez bien que la cible n'est construite que si le fichier source est plus récent.
On appelle cette structure une règle.
Le fichier Makefile n'est rien de plus qu'un ensemble de règles.

Retenez bien ce vocabulaire, c'est important pour la suite !
Sinon, ne pleurez pas si vous vous sentez perdus !

Hein ? J'ai pas tout compris ! Comment ça marche ?

Ca ira mieux avec un petit exemple :

Un Makefile minimal

Reprenons mon exemple de tout à l'heure !
Nous avons trois fichiers : main.c, fonctions.c et fonctions.h.

Les commandes à exécuter pour compiler seront :

Code : Console

```
gcc -c fonctions.c -o fonctions.o
gcc -c main.c -o main.o
gcc main.o fonctions.o -o Programme
```

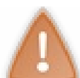
Le Makefile sera structuré comme tel :

Code : Bash

```
Programme : main.o fonctions.o
        gcc main.o fonctions.o -o Programme

main.o : main.c fonctions.c
        gcc -c main.c -o main.o

fonctions.o : fonctions.c
        gcc -c fonctions.c -o fonctions.o
```



Les tabulations avant les lignes de commandes sont obligatoires !
Eh oui ! Ne me demandez pas pourquoi, ce n'est pas moi qui ai créé make !

Regardons de plus près sur cet exemple comment fonctionne un Makefile.
Nous cherchons à créer le fichier exécutable Programme, la première dépendance (main.o) est la cible d'une des régles de notre Makefile, nous évaluons donc cette règle. Comme aucune dépendance de main.o n'est une règle, aucune autre règle n'est à évaluer pour compléter celle-ci.

Deux cas se présentent ici : soit le fichier main.c est plus récent que le fichier main.o, la commande est alors exécutée et main.o est construit ; soit main.o est plus récent que main.c et la commande n'est pas exécutée. L'évaluation de la règle main.o est terminée.

Les autres dépendances de la règle Programme (en l'occurrence ici : fonctions.o) sont examinées de la même manière (si fonctions.c est plus récent que fonctions.o, la commande qui construit ce dernier est exécutée).

Enfin, si nécessaire (si un des objets est plus récent que Programme), la commande de la règle Programme est exécutée et Programme est construit.

En résumé, les règles seront exécutées dans l'ordre inverse de l'écriture du Makefile, selon que les dépendances soient ou non plus récentes que leur cible !

Et comment je le lance ?

Il suffit d'enregistrer le projet sous Makefile (sans extension) puis de lancer dans un terminal :

Code : Console

```
make Programme
```

Facile, non ?

Notez que la commande make sans arguments exécute la première règle du Makefile, vous pouviez donc taper simplement :

Code : Console

```
make
```

Si vous utilisez vim, tapez :make pour lancer la compilation. S'il y a une erreur, le curseur se placera sur la ligne correspondante.

Un Makefile enrichi

On peut, avec cette syntaxe, ajouter d'autres fonctions à notre Makefile.

Par exemple, une fonction clean qui permet de supprimer les fichiers temporaires (objets), et une fonction mrproper qui permet un *rebuild* complet.
Pour les faire, c'est très simple, Makefile permet aussi d'exécuter des commandes du shell, c'est-à-dire les commandes classiques que vous entrez en ligne de commande (mkdir, rm, ls,...).

Notre Makefile devient alors :

Code : Bash

```
# création de l'exécutable 'Programme'
all: main.o fonctions.o
	gcc main.o fonctions.o -o Programme

main.o: main.c fonctions.h
	gcc -c main.c -o main.o

fonctions.o: fonctions.c
```

```
		gcc -c fonctions.c -o fonctions.o

	# suppression des fichiers temporaires
	clean:
		rm -rf *.o

	# suppression de tous les fichiers, sauf les sources,
	# en vue d'une reconstruction complète
	mrproper: clean
		rm -rf Programme
```

Vous remarquez que j'ai remplacé le nom de la première règle (Programme) par **all**.
C'est parce que on utilise des conventions dans les Makefile :

- all : compile tous les fichiers source pour créer l'exécutable principal ;
- install: exécute all, et copie l'exécutable, les librairies, les datas, et les fichiers en-tête s'il y en a dans les répertoires de destination ;
- uninstall : détruit les fichiers créés lors de l'installation, mais pas les fichiers du répertoire d'installation (où se trouvent les fichiers source et le Makefile) ;
- clean : détruit tout les fichiers créés par all ;
- info : génère un fichier info ;
- dvi : génère un fichier dvi ;
- dist : crée un fichier tar de distribution.

Revenons maintenant à mon exemple : pour l'exécuter, il suffit de mettre en console

Code : Console

```
make all
```

pour créer l'exécutable Programme

Code : Console

```
make clean
```

pour supprimer les fichiers temporaires

Code : Console

```
make mrproper
```

pour préparer une reconstruction complète.

Le shell Linux permet de combiner les commandes !
On peut donc tout envoyer en une seule ligne de commande !

Je vous rappelle que make seul équivaut à make all.
On obtient alors :

Code : Console

```
make && make clean
```

pour installer le programme sans laisser de fichiers temporaires.

Encore plus fort !

Code : Console

```
make mrproper && make
```

fera une reconstruction complète du programme.

Les directives

Dans certains cas, le Makefile tel que ci-dessus peut présenter quelques problèmes.
Nous allons voir lesquels et comment y pallier.

Cibles particulières : .PHONY

Reprenons la règle clean :

Code : Bash

```
clean:
	rm -rf *.o
```

Vous remarquerez que cette règle ne présente pas de dépendance.
make accepte ces règles, le fichier est alors considéré comme à jour s'il existe.

Mais il n'y a pas de fichiers clean ?

Justement, s'il y a dans le répertoire courant un fichier nommé clean, la commande ne sera jamais effectuée.

On définit alors la cible clean comme étant une cible particulière avec la directive .PHONY !
On ajoutera simplement une ligne au Makefile avant clean, voire même en tête du fichier.
Dans notre exemple, on définira clean et mrproper comme cibles spéciales :

Code : Bash

```
.PHONY: clean, mrproper
```

Les règles implicites

En fait, je ne vous l'avait pas dit, mais make est capable de générer certains fichiers même si on ne lui indique pas la commande !

Reprenons la règle all et ses dépendances.
Je vous la remets pour mémoire :

Code : Bash

```
all: main.o fonctions.o
	gcc main.o fonctions.o -o Programme

main.o: main.c fonctions.h
	gcc -c main.c -o main.o

fonctions.o: fonctions.c
	gcc -c fonctions.c -o fonctions.o
```

Si on supprime la règle main.o, make trouvera tout seul comme un grand comment créer main.o à partir de main.c. Pour des projets très simples, on peut même utiliser make sans Makefile !

On appelle cela les règles implicites. Ces règles - inexistantes dans le Makefile mais appliquées - peuvent poser des problèmes avec certains Makefile plus complexes.
On mettra alors la directive .SUFFIXES en tête du Makefile pour être tranquilles !

Mon Makefile devient alors :

Code : Bash

```
# définition des cibles particulières
.PHONY: clean, mrproper

# désactivation des règles implicites
.SUFFIXES:


# all
all: main.o fonctions.o
	gcc main.o fonctions.o -o Programme

main.o: main.c fonctions.h
	gcc -c main.c -o main.o

fonctions.o: fonctions.c
	gcc -c fonctions.c -o fonctions.o


# clean
clean:
	rm -rf *.bak rm -rf *.o

# mrproper
mrproper: clean
	rm -rf Programme
```

Ces directives ne sont pas obligatoires, mais il vaut mieux prendre l'habitude de les mettre par sécurité.

Makefile : un préprocesseur ?

Nous avons vu comment réaliser un Makefile simple permettant de contenir les informations pour compiler un programme. Cependant, nous sommes encore soumis à certaines limites :

- si on a 120 fichiers main.c, on aura 120 règles à définir ;
- s'ils sont répartis dans 50 répertoires, le Makefile va être imposant ;
- si on veut compiler avec un autre compilateur, ou bien ajouter des arguments à gcc, il faudra corriger toutes les commandes.

Heureusement pour nous, il existe des moyens de simplifier tout ça.

Les variables

Variables personnalisées

On peut introduire des variables dans make.
Bon d'accord, ce n'est pas vraiment des variables car leur valeur ne change pas !
En fait, ça ressemble aux macro-commandes #define en C !

On introduit une variable sous la forme NOM = VALEUR.
On appelle ensuite sa valeur avec $(NOM).

Un petit exemple pour être plus clair.
Définissons la variable CC pour définir le compilateur : CC = gcc.
on remplace alors gcc par $(CC) dans les commandes.

De même, imaginons qu'on veuille mettre des arguments à gcc pour la compilation.

Par exemple gcc -W -Wall -v.
On définit pareillement CFLAGS = -W -Wall -v.
On ajoute $(CFLAGS) aux commandes.

Mettons ça sur notre Makefile (toujours le même) :

Code : Bash

```
# définition des cibles particulières
.PHONY: clean, mrproper

# désactivation des règles implicites
.SUFFIXES:

# définition des variables
CC = gcc
CFLAGS = -W -Wall -v



# all
all: main.o fonctions.o
        $(CC) main.o fonctions.o -o Programme

main.o: main.c fonctions.h
        $(CC) -c main.c -o main.o $(CFLAGS)

fonctions.o: fonctions.c
        $(CC) -c fonctions.c -o fonctions.o $(CFLAGS)


# clean
clean:
        rm -rf *.bak rm -rf *.o

# mrproper
mrproper: clean
        rm -rf Programme
```

Et voilà !
Maintenant, si on veut utiliser d'autres arguments, il suffira de changer la valeur de CFLAGS en tête du fichier !

Pour les variables, on utilisera là aussi des conventions.

Pour les noms d'exécutables et d'arguments :

- AR : programme de maintenance d'archive (ar) ;
- CC : compilateur C (gcc) ;
- CXX : compilateur C++ (g++) ;
- RM : commande pour effacer un fichier (rm) ;
- TEX : programme pour créer un fichier TeX dvi à partir d'un source TeX (latex) ;
- ARFLAGS : paramètres à passer au programme de maintenance d'archives ;
- CFLAGS : paramètres à passer au compilateur C ;
- CXXFLAGS : paramètres à passer au compilateur C++ ;
- LDFLAGS : paramètres à passer au compilateur pour l'éditions de liens.

Pour les noms de répertoires et les destinations :

- prefix : racine du répertoire d'installation (= /usr/local) ;
- exec_prefix : racine pour les binaires (= $(prefix)) ;
- bindir : répertoire d'installation des binaires (= $(exec_prefix)/bin);
- libdir : répertoire d'installation des librairies (= $(exec_prefix)/lib) ;
- datadir : répertoire d'installation des données statiques pour le programme (= $(exec_prefix)/lib) ;

- statedir : répertoire d'installation des données modifiables par le programme (= $(prefix)/lib);
- includedir : répertoire d'installation des en-têtes (= $(prefix)/include) ;
- mandir : répertoire d'installation des fichiers de manuel (= $(prefix)/man) ;
- manxdir : répertoire d'installation des fichiers de la section x du manuel (= $(prefix)/manx) ;
- infodir : répertoire d'installation des fichiers info (= $(prefix)/info) ;
- srcdir: répertoire d'installation des fichiers sources (= $(prefix)/src).

Les variables automatiques

Makefile permet aussi l'utilisation de variables automatiques, calculées lors de l'exécution de chaque règle.

- $@ : nom de la cible ;
- $< : première dépendance de la liste des dépendances ;
- $? : les dépendances plus récentes que la cible ;
- $^ : toutes les dépendances ;
- $* : correspond au ' * ' simple dans le shell, i.e. représente n'importe quel nom.

On peut alors (encore !) remplacer notre Makefile par :

Code : Bash

```
# all
all: main.o fonctions.o
	$(CC) $^ -o Programme $(CFLAGS)

main.o: main.c fonctions.h
	$(CC) -c $< -o $@ $(CFLAGS)

fonctions.o: fonctions.c
	$(CC) -c $< -o $@ $(CFLAGS)
```

Pour plus de clarté, je n'ai repris que la partie qui nous intéresse ici.

J'ai pas l'impression que ça soit utile.
Ça sert pas juste à compliquer mon Makefile ?

Les variables internes sont utiles lorsqu'elles sont combinées avec ce que je vais vous apprendre tout de suite.

Les règles d'inférence

Nous pouvons spécifier des règles génériques.
Par exemple, vous avez remarqué que les règles main.o et fonctions.o se ressemblent.
On peut alors définir une seule et même règle !

On utilisera comme notation : %.o: %.c.
make lira alors "fais-moi des .o avec tous les .c à l'aide de la commande suivante".

Notre Makefile est alors simplifié.

Code : Bash

```
# all
all: main.o fonctions.o
	$(CC) $^ -o Programme $(CFLAGS)

%.o: %.c
```

```
            $(CC) -c $< -o $@ $(CFLAGS)
```

Une ligne de moins, ça fait plaisir !

Cependant, il y a un problème ici.
Vous n'avez rien remarqué ?
Le header a maintenant disparu des dépendances.

On peut corriger ça en spécifiant des dépendances en dehors des règles d'inférence :

Code : Bash

```
all: main.o fonctions.o
        $(CC) $^ -o Programme $(CFLAGS)


main.o: fonctions.h

%.o: %.c
        $(CC) -c $< -o $@ $(CFLAGS)
```

Et pourquoi je mets pas des * au lieu des % ?

Pour spécifier "tous les fichiers de type .c", on utilisera :

- %.c pour les cibles ;
- %.c dans les dépendances;
- $*.c dans les commandes.

Les bons élèves me diront :

Mais dans la commande clean de tout à l'heure
on avait utilisé *.o (et non $*.o) ?

Je le rappelle pour mémoire :

Code : Bash

```
# clean
clean:
        rm -rf *.o
```

La commande rm -rf *.o est une commande du shell.
Ici le caractère * est géré par le shell et non pas par make.

En fait on peut remplacer les caractères joker de make (% et $*) par celui du shell (*) SAUF POUR LES CIBLES !
On appelle les caractères % et $* des *patterns*.
En général, lorsqu'on utilise les patterns pour la cible, on les utilise dans toute la règle. Sinon on utilisera le caractère * du shell.

Les conditions

Pour optimiser notre Makefile, nous pouvons utiliser des conditions.
Nous avons vu que les variables fonctionnaient comme des #define en C !

Et qui dit #define, dit macros, dit #ifdef !
Je crois qu'en une phrase j'ai tout dit !

Dans un Makefile on utilsera la structure :

Code : Bash

```
ifeq ($(VARIABLE),valeur)
        # Actions

else
        # Actions
endif
```

Par exemple, imaginons que l'on veuille pouvoir compiler en mode debugger (debug) ou en mode distribution (release).

On définit une variable DEBUG dans le Makefile suivant :

Code : Bash

```
# définition des variables
CC = gcc
EXEC = Programme
DEBUG = yes

ifeq ($(DEBUG),yes)
        CFLAGS = -g -W -Wall
else
CFLAGS =
endif


# all conditionnel
all : $(EXEC)
ifeq ($DEBUG),yes)
        @echo "Génération en mode Debug"
else
        @echo "Génération en mode Release"
endif

Programme: main.o fonctions.o
        $(CC) $^ -o $(EXEC)


main.o: fonctions.h

%.o: %.c
        $(CC) -c $< -o $@ $(CFLAGS)
```

Facile, non ?

Non ? Bon allez, encore une dernière ligne droite et vous serez des pros du Makefile !

L'appel récursif

Il est possible d'appeler make à l'intérieur d'un Makefile.

Je vais prendre comme exemple cette fois un programme plus compliqué.

Nous avons un fichier main.c dans le répertoire main.
Nous avons fonctions1.c, fonctions2.c, fonctions3.c dans le répertoire fonctions (avec les headers correspondants).

On veut créer un Makefile dans chaque répertoire, et un autre qui y fera appel.

make peut appeler un autre Makefile avec la commande make -C [répertoire].

Ex. :

Code : Console

```
make -C main all
```

va exécuter la règle all dans le Makefile situé dans main.

On peut donc faire nos Makefile comme ceci :

Code : Bash

```
# Makefile racine
.SUFFIXES:
.PHONY:

LD = gcc
LDFLAGS =
OBJ = fonctions/*.o main/*.o

all:
        $(LD) $(OBJ) -o Programme

%.o
        make -C main
        make -C fonctions

clean:
        make -C main clean
        make -C fonctions clean
        rm -rf *.o
```

Code : Bash

```
# main/Makefile
.SUFFIXES:
.PHONY: clean

CC = gcc
CFLAGS = -W
OBJ = main.o

all: $(OBJ)

%o: %c
        $(CC) -c $^ -o $@ $(CFLAGS)

clean:
        rm -rf *.o
```

Code : Bash

```
# fonctions/Makefile
.SUFFIXES:
.PHONY: clean

CC =gcc
CFLAGS = -W
OBJ = fonctions1.o fonctions2.o fonctions3.o


all: $(OBJ)
```

```
%o: %c
        $(CC) -c $^ -o $@ $(CFLAGS)

clean:
        rm -rf *.o
```

Note : j'aurais pu remplacer make -C main par cd main && make.
Ça revient au même.

Pour aller jusqu'au bout :

- -n : liste ce que make va faire, sans le faire ;
- -f : indique le fichier Makefile ;
- -j3 : parallélise l'exécution de make (très utile sur biprocesseur) ;
- -p : liste les règles implicites ;
- Voyez man make

Pour aller plus loin...

Tout d'abord, pour aller plus loin, n'hésitez pas à consulter les documentations de références sur le sujet.
En plus des pages de manuel, vous pouvez visiter le site de GCC et parcourir The GNU coding standards

Ici, je vous donne quelques pistes pour vous « plonger » dans la compilation linuxienne.
Ne voulant pas trop parler sur des sujets que je ne maîtrise pas, je ne donnerai qu'un bref aperçu des différentes méthodes, avec quelques liens.

Si vous trouvez de meilleurs liens, veuillez me les envoyer par MP. Bien sûr, libre à vous de faire un tuto sur le sujet.

Des Makefile automatiques : les autotools

Il existe des outils appelés automake, autoconf et d'autres noms barbares, qui permettent de créer un Makefile à partir d'un Makefile « de base » et à l'aide d'un script généralement appelé configure.

C'est très utile lorsque les sources à compiler sont importantes. De plus, le configure peut vérifier les dépendances, et peut prendre des arguments qui permettront de modifier le Makefile en conséquence, en fonction de la configuration de la machine sur laquelle on veut installer le programme.
C'est pourquoi cette méthode est largement utilisée pour la distribution de sources. Si vous avez déjà téléchargé les sources d'un programme, vous les compilez généralement en faisant :

Code : Console

```
./configure
make
make install
```

Vous avez un aperçu de cette méthode sur cette page.
Si vous maitrisez la langue de shakespeare, voyez (entre autres) :

- http://airs.com/ian/configure/configure_toc.html
- http://www.amath.washington.edu/~lf/tu [...] nual_toc.html
- http://sources.redhat.com/automake/automake.html (ou « info automake » en console)

À noter aussi que l'IDE anjuta utilise cette méthode pour compiler les programmes que vous faites avec, si vous voulez regarder

la syntaxe des différents fichiers...

Créer un paquet

Si vous voulez faire un paquet pour votre distribution, il existe plusieurs méthodes.

checkinstall

checkinstall est une commande permettant de créer un paquet à partir des binaires et du Makefile qui les a générés, plus particulièrement la cible install.

Une fois que vous avez fait make (et peut-être ./configure avant !), utilisez simplement la commande :

pour un paquet debian .deb (debian, ubuntu, knoppix...) :

Code : Console

```
checkinstall -D make install
```

pour un paquet .rpm (fedora, mandriva...) :

Code : Console

```
checkinstall -R make install
```

pour un paquet .tgz (slackware, backtrack...) :

Code : Console

```
checkinstall -S make install
```

Cette méthode à l'avantage d'etre facile à utiliser, mais n'est pas la plus appropriée, et ne fonctionne pas toujours :
http://wiki.slackbuilds.net/faq#je_sui [...] _checkinstall

Checkinstall est en général utilisé quand on compile pour une seule machine, car c'est toujours plus propre qu'un make install brut, et n'est pas plus compliqué à faire. On ne trouvera pas de paquets créés avec checkinstall dans les dépots.

On préfèrera donc utiliser les outils de création de paquet fournis avec sa distribution :

À la sauce debian

Il existe une méthode spécifique debian (et dérivés), qui consiste à faire un Makefile un peu particulier, nommé rules et associé à d'autres fichiers, le tout mis dans un répertoire debian.
On retrouve un peu la structure du Makefile, mais on n'utilise pas make, c'est un script exécutable directement.
On peut donc construire le paquet avec la commande :

Code : Console

```
./debian/rules binary
```

Les autres fichiers du répertoire debian sont tous liés à la création du paquet par cette méthode, qui se veut la méthode "officielle" sous Debian.

Vous trouverez plus de renseignements sur le guide du nouveau responsable Debian et sur la page "créer un paquet" de la doc Ubuntu.

Vous pouvez aussi jeter un oeil au tuto sur debian-fr

Façon slackware

Slackware a aussi sa méthode de création de paquet, avec la commande makepkg. Pour faire un paquet correctement il faut faire un Slackbuild : c'est un script qui compile le programme et lance makepkg, en faisant les vérifications nécessaires.

La méthode est décrite dans le Nano-manuel d'empaquetage slackware (sur slackfr.org). Voir aussi le wiki de slackbuild.net (fr) et le HOWTO:Writing a Slackbuild script (en)

D'autres docs sont disponibles sur le net, surtout si vous n'avez pas peur de l'anglais...

Compiler via make sous Windows

Vous voulez faire un projet multiplateforme ?
Vous voulez faire un programme pour Windows et les IDE vous font vomir ?

Ne vous inquiétez pas, la solution existe : MinGW (Minimal GNU for Windows).
MinGW est un portage de gcc et make sous Windows.
Vous êtes enfin arrivé à la fin de ce tutoriel. J'espère qu'il vous a plu (ou au moins, qu'il vous a servi).

Je reste toujours prêt à corriger d'éventuelles erreurs et/ou à le remanier selon vos commentaires (ne soyez pas trop pressés quand même).

Partager