# Guide pour la compilation C & Makefile

## Chaine de compilation monofichier classique

Pour la plupart des langages, on dispose de compilateurs libres: GCC pour C, G++ pour C++, GCJ pour Java, GNAT pour Ada, Gfortran pour Fortran, GPC pour Pascal,...

GCC a été porté sur un nombre considérable de systèmes d'exploitation (pratiquement toutes les variantes d'Unix, VMS, Windows) et de microprocesseurs.

### *Exemple*

La syntaxe de base utilisée par les compilateurs de GCC est :

```
gcc fichierSource.c -o fichierExecutable.exe
```

Cette commande soumet le fichier « fichierSource.c » au compilateur et spécifie que le fichier l'exécutable devra s'appeler « fichierExecutable.exe » (option –o pour output suivie du nom de l'exécutable).

On utilisera le plus souvent la commande suivante :

```
gcc FichierSource.c -Wall -o FichierExecutable
```

et, si on doit accéder à la bibliothèque mathématique (sqrt, pow…):

```
gcc FichierSource.c -Wall -o FichierExecutable -lm
```

## GCC : Les options

Le tableau qui suit présente quelques options courantes applicables au compilateur GCC pour une programmation traditionnelle sous linux. On peut distinguer deux types d'options:

- Options de compilation:

| Option | Définition | Complément |
|---|---|---|
| -c | Compiler ou assembler les fichiers sources, mais ne pas éditer les liens. | L'étape d'édition des liens n'est pas effectuée. La sortie finale du compilateur correspond à un fichier objet pour chaque fichier source. Elle permet la compilation séparée. |

| | | |
|---|---|---|
| -o fichier | Placer la sortie dans fichier. | Fixe le nom du fichier objet généré lors de la compilation d'un fichier source. |
| -ansi<br>-std=c89 | Supporter tous les programmes ISO C89. | Cela désactive certaines fonctionnalités de GCC qui sont incompatibles avec le C89 ISO pendant la compilation de code C, comme les mots-clés « asm » et « typeof », et les macros prédéfinies comme « unix » et «vax » qui identifient le type de système que vous utilisez. |
| -Wall | Affiche tous les messages d'avertissement. | En général, ces avertissements sont une aide appréciable pour le programmeur. |
| -pedantic | émettre tous les avertissements demandés par le standard C ISO | Rejeter tous les programmes qui utilisent des extensions interdites, et d'autres programmes qui ne suivent pas le C ISO. |
| -g | Génère les informations symboliques de déboguage. | Cela permet en utilisant un debugger (gdb,ddd...) d'avoir accès aux noms de variables utilisées dans le source, de savoir pour chaque instruction machine exécutée à quel fichier source, et à quelle ligne de code cela correspond. |
| -std=c99 | Supporter tous les programmes ISO C99. | Elle permet : l'utilisation de commentaires //, la déclaration de variables de contrôle de boucles « For (char C='a'...) et gère les fichiers texte en UTF8 (gestion des accents dans un terminal). |
| -Ichemin | Préciser le chemin de répertoires où gcc doit chercher des headers inclus. | L'écriture du chemin dépend du système d'exploitation, le chemin peut être absolu ou relatif:<br>Linux:<br>• chemin absolu -I/usr/local/include<br>• chemin relatif -I../../headers (on remonte de 2 niveaux...)<br>-I./headers (à partir du répertoire courant)<br><br>Windows:<br>• chemin absolu -IC:\programs\headers<br>• chemin relatif -I..\..\headers (on remonte de 2 niveaux...) |

- Options de l'édition de liens:

| Option | Définition | Complément |
|---|---|---|
| -lm | Utilisation de la bibliothèque mathématique. | |
| -Lchemin | Préciser le chemin de répertoires où gcc doit chercher des fichiers nécessaires au link | L'écriture du chemin absolu ou relatif dépend du système d'exploitation:<br>Linux: -L/usr/local/include<br>Windows: -L\C\bureau |

# Compilation multi-fichiers

Prenons un exemple : vous réalisez un programme permettant de gérer une liste de mots. Vous spécifiez le type des éléments de la liste et implémentez la liste elle-même. De plus, cette liste va être utilisée par un programme principal. On a les cinq fichiers :

- element.h
- element.c
- liste.h
- liste.c
- main.c

Il est nécessaire de tenir compte des dépendances entre les fichiers. En C, ces dépendances sont matérialisées par les instructions #include.

On peut compiler manuellement, de la manière suivante :

```
//création des fichiers objet
gcc –c element.c
gcc –c liste.c
gcc –c main.c
//édition de liens et création de l'exécutable
gcc element.o liste.o main.o –o appli.exe
```

## Makefile :

Make est un logiciel traditionnel d'UNIX. Il sert à appeler des commandes créant des fichiers. A la différence d'un simple script shell, make n'exécute les commandes seulement que si elles sont nécessaires. Le but est d'arriver à un résultat (logiciel compilé ou installé, documentation créée, etc.) sans nécessairement refaire toutes les étapes.

La compilation multi-fichiers en ligne de commande devient vite fastidieuse lorsque le nombre de fichiers de votre application augmente. Make permet d'automatiser la compilation multi-fichiers (compilation séparée). Sa principale fonction est de prendre en compte les dépendances entre fichiers pour qu'après la modification de l'un d'entre-eux, tous les fichiers qui en dépendent, mais seulement ceux-là, soient recompilés.

Pour ce faire, Make utilise les informations contenues dans un fichier texte, le `makefile`. Ce fichier doit contenir toutes les informations de dépendances entre les sources du programme, et toutes les informations nécessaires pour compiler ces fichiers (compilateurs à utiliser, options de compilation, etc.)

Pour automatiser l'étape de compilation avec un Makefile, il est nécessaire de décomposer les inclusions présentent dans chacun des fichiers.

Pour commencer, déclarons au sein de variables les instructions et options qui seront utilisées :

```
//nous utilisons le compilateur gcc, gcc sera accessible dans le makefile par la variable $(CC)
CC=gcc

//on peut définir une variable pour les options de compilation. Ici on utilisera les options –Wall et -g
CFLAGS=-Wall -g

//on peut définir une variable pour les options de l'édition de liens. Ici on utilisera les options –lm
LDFLAGS=-lm

//on peut aussi définir dans une variable le nom du fichier executable. Ici il s'appellera MonAppli
EXEC= MonAppli
```

Ensuite il faut définir les instructions de compilation. Pour notre exemple, il est nécessaire de réaliser les actions suivantes (dans l'ordre) :

- Création du fichier objet (.o) pour element
  ```
  element.o: element.c
      $(CC) -o element.o -c element.c $(CFLAGS)
  ```

- Création du fichier objet (.o) pour liste
  ```
  liste.o: liste.c
      $(CC) -o liste.o -c liste.c $(CFLAGS)
  ```

- Création du fichier objet (.o) pour main
  ```
  main.o: main.c
      $(CC) -o main.o -c main.c $(CFLAGS)
  ```

- Edition des liens (MonAppli est dans la variable $(EXEC))
  ```
  $(EXEC): main.o liste.o element.o
      $(CC)  main.o liste.o element.o -o $(EXEC) $(LDFLAGS)
  ```

- Création de l'exécutable
  ```
  all : $(EXEC) $(LDFLAGS)
  ```

Enfin on peut ajouter des options à notre fichier Makefile. Par exemple, une option de nettoyage des fichiers générés (les .exe, les .o, les .stackdump, etc.). On défini les options que l'on veut créer de la sorte :

```
//option de nettoyage clean. Elle appelle la commande du shell « rm » qui permet la suppression de fichiers. Ici on supprime tous les fichiers se terminant par « *.o », « *.exe » et « *.stackdump »
    rm -rf  *.o *.exe  $(EXEC)
```

Notre fichier Makefile réassemblé avec tous ces éléments est le suivant :

```
CC=gcc
```

```
CFLAGS=-Wall-g
#si on ne souhaite pas d'option lors de l'édition de lien, on ne renseigne pas cette variable
LDFLAGS=
EXEC= MonAppli

all : $(EXEC)

#compile element
element.o: element.c
	$(CC) -o element.o -c element.c $(CFLAGS)

#compile liste
liste.o: liste.c
	$(CC) -o liste.o -c liste.c $(CFLAGS)

#compile main
main.o: main.c
	$(CC) -o main.o -c main.c $(CFLAGS)

#link main avec liste et element
$(EXEC): main.o liste.o element.o
	$(CC) main.o liste.o element.o -o $(EXEC) $(LDFLAGS)


//options du makefile
clean:
	rm -rf *.o *.exe *.stackdump
```

## *Attention !*

- prenez soin de bien respecter les tabulations au début de certaines lignes !
- le fichier doit se nommer makefile (sans extension) et être placé dans le répertoire courant (contenant les .h et .c)

## *Utilisation du makefile*

par la suite on tapera dans l'invite de commande :

```
//compilation
utilisateur@machine ~/home/programmation
$make

//nettoyage
utilisateur@machine ~/home/programmation
$make clean
```