Programmation C et Système

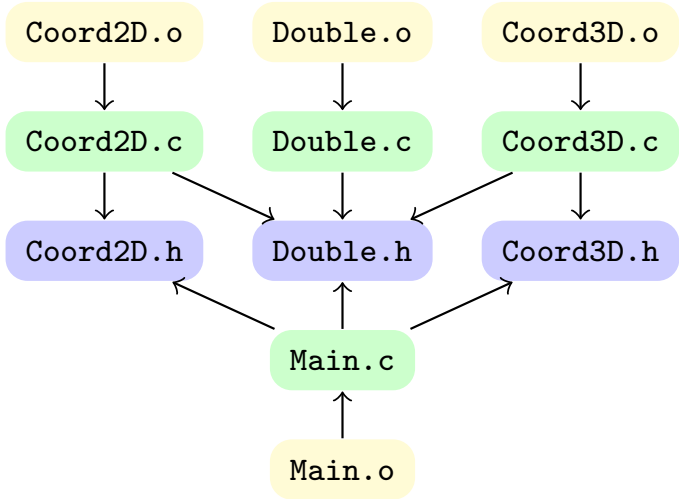
Le programme `make` et son fichier `Makefile`

Régis Barbanchon

L2 Informatique

Modification d'un source et recompilation

Reprenons notre programme fil rouge sur la compilation séparée :



Jusqu'ici, lorsqu'un fichier source est modifié,
on recompile systématiquement la totalité les sources :

```
$ ls *.c
Coord2D.c  Coord3D.c  Double.c  Main.c
$ clang -W -Wall -std=c99   -c *.c
$ ls *.o
Coord2D.o  Coord3D.o  Double.o  Main.o
```

avant de lier leur modules et `libm` en l'exécutable `prog` :

```
$ clang   -lm   *.o -o prog
```

Fichier `Makefile` et commande `make` (1/3)

Lorsqu'un fichier source est modifié, il suffit de :

- recompiler uniquement les modules qui en dépendent
- refaire l'édition de lien.

Un fichier `Makefile` permet de spécifier :

- les dépendances `.c/.h` de chaque fichier `.o` à compiler,
- les dépendances `.o` de l'exécutable lors de l'édition de liens,
- la commande à exécuter pour construire chacune de ces cibles.

Ces spécifications prennent la forme de règles, dont la syntaxe est :

```
target_file: dependancy_file_1 dependancy_file_2 ... dependancy_file_n
        command_to_create_target_file_from_dependancies
```

IMPORTANT : la commande de la règle doit obligatoirement être précédée d'une **tabulation**, sans autre forme d'espace.

Fichier `Makefile` et commande `make` (2/3)

Notre fichier `Makefile` contient donc 5 règles :

- 1 règle de d'édition de lien fabriquant `prog` à partir des `.o`,
- 4 règles de compilation fabriquant des `.o` à partir des `.c/.h`.

```
# file: Makefile

prog: Coord2D.o Coord3D.o Double.o Main.o
	clang Coord2D.o Coord3D.o Double.o Main.o -lm -o prog

Coord2D.o: Coord2D.c Coord2D.h Double.h
	clang -std=c99 -W -Wall -pedantic Coord2D.c -c

Coord3D.o: Coord3D.c Coord3D.h Double.h
	clang -std=c99 -W -Wall -pedantic Coord3D.c -c

Double.o: Double.c Double.h
	clang -std=c99 -W -Wall -pedantic Double.c -c

Main.o: Main.c Double.h Coord2D.h Coord3D.h
	clang -std=c99 -W -Wall -pedantic Main.c -c
```

RAPPEL : la commande de la règle doit obligatoirement être précédée d'une **tabulation**, sans autre forme d'espace.

Fichier `Makefile` et commande `make` (3/3)

La commande `make [-f makefile] [target]`
permet de construire la cible `target` à l'aide du fichier `file`.
Par défaut, il utilise le fichier `Makefile` du répertoire courant.
Par défaut, il construit la première cible du fichier (ici, `prog`).

```
$ make
clang -std=c99 -W -Wall -pedantic Coord2D.c -c
clang -std=c99 -W -Wall -pedantic Coord3D.c -c
clang -std=c99 -W -Wall -pedantic Double.c -c
clang -std=c99 -W -Wall -pedantic Main.c -c
clang Coord2D.o Coord3D.o Double.o Main.o -lm -o prog
```

`make` se fie aux dates de dernière modification des fichiers.
Si la date d'une cible est plus ancienne que celle d'une de ses dépendances, alors cette cible doit être reconstruite.

```
$ make
make: 'prog' is up to date.
```

Par exemple, si l'on touche au header `Coord2D.h`,
alors `Coord2D.o` et `Main.o` doivent être reconstruits, puis `prog` :

```
$ touch Coord2D.h
$ make
clang -std=c99 -W -Wall -pedantic Coord2D.c -c
clang -std=c99 -W -Wall -pedantic Main.c -c
clang Coord2D.o Coord3D.o Double.o Main.o -lm -o prog
```

Vers un `Makefile` générique (1/6)

On utilise en général pour toutes les cibles :

- le même compilateur `CC`,
- le même jeu d'options de compilation `CFLAGS`.

On élimine donc les redites de notation avec des variables.
Le bloc d'initialisation se trouve avant le bloc de règles.
Une variable `VAR` est ensuite évaluée avec la syntaxe `$(VAR)` :

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

prog: Coord2D.o Coord3D.o Double.o Main.o
	$(CC) Coord2D.o Coord3D.o Double.o Main.o $(LIBS) -o prog

Coord2D.o: Coord2D.c Coord2D.h Double.h
	$(CC) $(CFLAGS) Coord2D.c -c

Coord3D.o: Coord3D.c Coord3D.h Double.h
	$(CC) $(CFLAGS) Coord3D.c -c

Double.o: Double.c Double.h
	$(CC) $(CFLAGS) Double.c -c

Main.o: Main.c Double.h Coord2D.h Coord3D.h
	$(CC) $(CFLAGS) Main.c -c
```

Vers un `Makefile` générique (2/6)

Trois variables spéciales permettent d'éliminer des redites :

- `$@` indique la cible d'une règle,
- `$<` indique sa dépendance la plus à gauche,
- `$^` indique toutes ses dépendances.

On peut donc réécrire le `Makefile` comme :

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

prog: Coord2D.o Coord3D.o Double.o Main.o
	$(CC) $^ $(LIBS) -o $@

Coord2D.o: Coord2D.c Coord2D.h Double.h
	$(CC) $(CFLAGS) $< -c

Coord3D.o: Coord3D.c Coord3D.h Double.h
	$(CC) $(CFLAGS) $< -c

Double.o: Double.c Double.h
	$(CC) $(CFLAGS) $< -c

Main.o: Main.c Double.h Coord2D.h Coord3D.h
	$(CC) $(CFLAGS) $< -c
```

Vers un `Makefile` générique (3/6)

Toutes les commandes de compilation étant maintenant identiques, on peut centraliser cette commande dans la cible spéciale `.c.o`, et éliminer la commande de toutes les règles de compilation :

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

prog: Coord2D.o Coord3D.o Double.o Main.o
	$(CC) $^ $(LIBS) -o $@

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

Coord2D.o: Coord2D.c Coord2D.h Double.h
Coord3D.o: Coord3D.c Coord3D.h Double.h
Double.o: Double.c Double.h
Main.o: Main.c Double.h Coord2D.h Coord3D.h
```

Or, les dernières lignes sont ce que `clang -MM *.c` affiche :

```
$ clang -MM *.c
Coord2D.o: Coord2D.c Coord2D.h Double.h
Coord3D.o: Coord3D.c Coord3D.h Double.h
Double.o: Double.c Double.h
Main.o: Main.c Double.h Coord2D.h Coord3D.h
```

On doit donc pouvoir générer ces lignes automatiquement.

Vers un `Makefile` générique (4/6)

On crée une cible `depend` qui dépend des fichiers sources `.c` et `.h`, et dont le contenu et la sortie de `clang -MM` pour ces sources.

On inclut ensuite le contenu de `depend` en fin de fichier, grâce à la directive `-include depend` :

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

prog: Coord2D.o Coord3D.o Double.o Main.o
        $(CC) $^ $(LIBS) -o $@

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

depend: Coord2D.c Coord3D.c Double.c Main.c
        $(CC) -MM $^ | tee $@

-include depend
```

Il y a une redite de noms entre les fichiers `.o` et `.c` des modules.

Vers un `Makefile` générique (5/6)

On crée une variable `MODULES` contenant la liste des modules, et on génère la listes des fichiers `.o` et `.c` à partir de cette liste via la fonction `$(addsuffix suffix , list)` :

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

EXEC = prog
MODULES = Coord2D Coord3D Double Main

SOURCES = $(addsuffix .c , $(MODULES))
OBJECTS = $(addsuffix .o , $(MODULES))

$(EXEC): $(OBJECTS)
        $(CC) $^ $(LIBS) -o $@

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

depend: $(SOURCES)
        $(CC) -MM $^ | tee $@

-include depend
```

Vers un `Makefile` générique (6/6)

Si l'on ne veut pas écrire la liste des modules dans le `Makefile`, on peut les générer en listant les `.c` via `$(wildcard *.c)`, puis en supprimant les suffixes des `.c` via `$(basename list)` :

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

EXEC = prog
HEADERS = $(wildcard *.h)
SOURCES = $(wildcard *.c)
MODULES = $(basename $(SOURCES))
OBJECTS = $(addsuffix .o , $(MODULES))

$(EXEC): $(OBJECTS)
        $(CC) $^ $(LIBS) -o $@

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

depend: $(HEADERS) $(SOURCES)
        $(CC) -MM $(SOURCES) | tee $@

-include depend
```

On en profite pour rajouter les headers en dépendance de `depend`.

Suppression de l'écho de la commande exécutée

Avant qu'une commande `cmd` soit exécutée, elle est affichée écho, sauf lorqu'elle est lancée avec la syntaxe `@cmd`.
Cette syntaxe est surtout utile avec la commande `echo` :
en général, on affiche un message `msg` avec `@echo msg`.

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

EXEC = prog
HEADERS = $(wildcard *.h)
SOURCES = $(wildcard *.c)
MODULES = $(basename $(SOURCES))
OBJECTS = $(addsuffix .o , $(MODULES))

$(EXEC): $(OBJECTS)
        @echo === LINKING $@ ===
        $(CC) $^ $(LIBS) -o $@

.c.o:
        @echo === COMPILING $@ ===
        $(CC) $< $(CFLAGS) -c

depend: $(HEADERS) $(SOURCES)
        @echo === COMPUTING DEPENDANCIES $@ ===
        $(CC) -MM $(SOURCES) | tee $@

-include depend
```

Règle bidons (phony rules)

Les règles de la forme `target::` sont des règles bidons.
On les utilise quand `target` n'est pas le nom d'un fichier.
Par ex, si on veut que `make clean` supprime les fichiers `.o` :

```
# file: Makefile
CC = clang
CFLAGS = -std=c99 -W -Wall -pedantic
LIBS = -lm

EXEC = prog
HEADERS = $(wildcard *.h)
SOURCES = $(wildcard *.c)
MODULES = $(basename $(SOURCES))
OBJECTS = $(addsuffix .o , $(MODULES))

$(EXEC): $(OBJECTS)
        $(CC) $^ $(LIBS) -o $@

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

depend: $(HEADERS) $(SOURCES)
        @echo === COMPUTING DEPENDANCIES $@ ===
        $(CC) -MM $(SOURCES) | tee $@

clean::
        rm -f *.o

-include depend
```

Link avec les librairies utilisant `pkg-config` (1/5)

Supposons par exemple qu'on veuille écrire un programme qui affiche `"Hello World!"` dans une image.

```
$ ./hello
```

On peut le faire avec la libraire de développement MagickWand, qui s'installe sous Ubuntu comme suit :

```
$ sudo apt install libmagickwand-dev
```

Link avec les librairies utilisant pkg-config (2/5)

Le code de hello.c utilisant MagickWand s'écrit alors :

```c
// File: hello.c
#include <stdlib.h>
#include <stdbool.h>
#include <wand/magick-wand.h>

int main (void) {
  MagickWandGenesis();

  MagickWand * image= NewMagickWand();
  PixelWand * color= NewPixelWand();
  PixelSetColor (color, "#FFFF00");
  MagickNewImage (image, 200, 200, color);
  DestroyPixelWand (color);

  DrawingWand * drawing= NewDrawingWand();
  DrawSetTextAlignment (drawing, CenterAlign);
  DrawSetFont (drawing, "Times-Bold");
  DrawSetFontSize (drawing, 25);
  DrawAnnotation (drawing, 100, 100, (unsigned char *) "Hello World!");
  MagickDrawImage (image, drawing);
  DestroyDrawingWand (drawing);

  MagickDisplayImage(image, getenv ("DISPLAY"));
  MagickWriteImage (image, "./hello.png");
  DestroyMagickWand (image);

  MagickWandTerminus();
  return 0;
}
```

Link avec les librairies utilisant `pkg-config` (3/5)

Lors de la compilation, il faut rajouter les options affichées par `pkg-config --cflags MagickWand` :

```
$ pkg-config --cflags MagickWand
-fopenmp -DMAGICKCORE_HDRI_ENABLE=0 -DMAGICKCORE_QUANTUM_DEPTH=16
-I/usr/include/x86_64-linux-gnu/ImageMagick-6
-I/usr/include/ImageMagick-6
```

On y trouve l'option `-I path` qui rajoute le chemin `path` d'un répertoire où trouver les headers `.h` de la libiaire.

De même, lors de l'édition de lien, il faut rajouter les options affichées par `pkg-config --libs MagickWand`

```
$ pkg-config --libs MagickWand
-lMagickWand-6.Q16 -lMagickCore-6.Q16
```

On y trouve souvent l'option `-L path` qui rajoute le chemin `path` d'un répertoire où trouver les binaires de la librairie.
(Mais dans le cas présent, aucun chemin n'est rajouté.)

Link avec les librairies utilisant `pkg-config` (4/5)

En bash, `$(cmd)` est substitué par ce qu'affiche la commande `cmd`.
On peut donc compiler `hello.c` en `hello.o` comme suit :

```
$ gcc $(pkg-config --cflags MagickWand) -std=c99 hello.c -c
```

Et on peut lier `hello.o` et MagickWand en `hello` comme suit :

```
$ gcc $(pkg-config --libs MagickWand) hello.o -o hello
```

Ou encore, en un seul temps (compilation + édition de lien) :

```
$ gcc $(pkg-config --cflags --libs MagickWand) -std=c99 hello.c -o hello
```

La dernière commande est alors équivalente à :

```
$ gcc -fopenmp -DMAGICKCORE_HDRI_ENABLE=0 -DMAGICKCORE_QUANTUM_DEPTH=16 \
      -I/usr/include/x86_64-linux-gnu/ImageMagick-6                      \
      -I/usr/include/ImageMagick-6                                       \
      -lMagickWand-6.Q16 -lMagickCore-6.Q16                              \
      -std=c99 hello.c -o hello
```

Link avec les librairies utilisant `pkg-config` (5/5)

Dans un `Makefile`, la syntaxe `$(cmd)` du bash devient `$$(cmd)`, car la première est déjà prise pour l'expansion des variables.

```
# File: Makefile
CC      = gcc # because clang has problems with MagickWand
CFLAGS  = -std=c99 -W -Wall -pedantic
CFLAGS += $$(pkg-config --cflags MagickWand)
LIBS    = $$(pkg-config --libs   MagickWand)

EXEC    = hello
HEADERS = $(wildcard *.h)
SOURCES = $(wildcard *.c)
MODULES = $(basename $(SOURCES))
OBJECTS = $(addsuffix .o , $(MODULES))

$(EXEC): $(OBJECTS)
        @echo === LINKING $@ ===
        $(CC) $^ $(LIBS) -o $@

.c.o:
        @echo === COMPILING $@ ===
        $(CC) $< $(CFLAGS) -c

depend: $(HEADERS) $(SOURCES)
        @echo === COMPUTING DEPENDANCIES $@ ===
        $(CC) -MM $(SOURCES) | tee $@

-include depend
```

Remarque : On a augmenté la var `CFLAGS` avec l'opérateur `+=`.