

Construcción de programas con `make`

Acerca de este documento

Este documento pretende recoger toda la información necesaria para poder escribir ficheros `Makefile` usados para controlar la construcción de un programa con el comando `make`.

El documento empieza explicando las opciones más básicas para ir profundizando en ellas poco a poco. En los apartados 7 y 8 se acaban dando una serie de reglas prácticas que simplifican mucho el desarrollo de estos ficheros, con lo que aunque el lector no desee leer todo el documento le recomendamos que sí que consulte estos apartados.

Nota Legal

Este tutorial ha sido escrito por Fernando López Hernández para MacProgramadores, y de acuerdo a los derechos que le concede la legislación española e internacional el autor prohíbe la publicación de este documento en cualquier otro servidor web, así como su venta, o difusión en cualquier otro medio sin autorización previa.

Sin embargo el autor anima a todos los servidores web a colocar enlaces a este documento. El autor también anima a cualquier persona interesada en conocer el funcionamiento del comando `make` a bajarse o imprimirse este tutorial.

Madrid, Abril 2004

Para cualquier aclaración contacte con:
`fernando@DELITmacprogramadores.org`

Tabla de contenido

1	Los ficheros <code>Makefile</code>	4
1.1	Dependencias y reglas	4
1.2	El target final.....	7
1.3	Falsos targets.....	8
1.4	Targets de seguimiento.....	10
1.5	Uso de comodines	11
2	Principales argumentos del comando <code>make</code>	12
3	Macros.....	15
3.1	Introducción.....	15
3.2	Macros predefinidas.....	18
3.3	Fijar los valores de las macros desde fuera del <code>Makefile</code>	18
3.4	Macros de expansión única y macros de expansión recursiva.....	19
3.5	Macros automáticas	20
3.6	Comodines en macros.....	22
4	Compilación condicional.....	23
5	Acceso al shell.....	24
6	Reglas implícitas.....	25
6.1	Como usar las reglas implícitas.....	25
6.2	Crear reglas implícitas.....	26
7	Tener un proyecto repartido en varios subdirectorios.....	29
7.1	Buscar dependencias en varios subdirectorios con la directiva <code>vpath</code>	29
7.2	La variable de entorno <code>VPATH</code>	30
7.2.1	Incluir otros ficheros	30
7.3	Tener librerías organizadas en subdirectorios	31
7.4	Paso de variables a un sub-make.....	31
8	Obtener las reglas con <code>gcc</code>	33

1 Los ficheros `Makefile`

1.1 Dependencias y reglas

Cuando tenemos que compilar un proyecto pequeño simplemente usamos el comando `gcc` de la forma:

```
$ gcc -o miprograma fuente1.c fuente2.c
```

Pero si el proyecto empieza a tener muchos ficheros `.c` y `.h` al final se pierde demasiado tiempo recompilando todos los ficheros del programa cada vez que hacemos un cambio, o bien, eligiendo qué ficheros necesitan recompilarse. Para solucionar este problema podemos crear un fichero `Makefile` con las reglas de recompilación de los diferentes ficheros del proyecto.

Una vez escritas las reglas en el fichero `Makefile` ejecutamos el comando `make` desde el directorio donde esté el fichero `Makefile`, y `make` busca un fichero con ese nombre y si lo encuentra ejecuta sus reglas.

Imaginemos que tenemos los siguientes ficheros:

```
// main.c
#include "primero.h"
.....
```

```
// primero.c
#include "primero.h"
#include "segundo.h"
.....
```

```
// segundo.c
#include "segundo.h"
.....
```

Ahora según los `#include` que tenemos cuando modifiquemos `segundo.h` debemos de recompilar `primero.c`, `segundo.c`, y cuando modifiquemos `primero.h` debemos de recompilar `main.c`, `primero.c`. Por otro lado imaginemos que cuando recompilemos `segundo.c` hay que volver a recompilar `primero.c` y `main.c`, cuando recompilemos `primero.c` hay que recompilar `main.c`, por último sabemos que cuando recompilemos algún `.c` tendremos que volver a enlazar los ficheros para generar un nuevo ejecutable. La Figura 1 muestra estas dependencias de forma gráfica.

Estas reglas escritas en el fichero Makefile quedarían:

```

miprograma : main.o primero.o segundo.o
    gcc -o miprograma main.o primero.o segundo.o
main.o : main.c primero.h primero.o segundo.o
    gcc -c main.c
primero.o : primero.c primero.h segundo.h segundo.o
    gcc -c primero.c
segundo.o : segundo.c segundo.h
    gcc -c segundo.c
    
```

Listado 1: Fichero Makefile de las dependencias

Si ahora lo ejecutamos obtenemos:

```

$ make
gcc -c primero.c
gcc -c main.c
gcc -o miprograma main.o
primero.o segundo.o
    
```

En su forma general cada regla tiene la forma:

```

target : dependencias
    comando1
    comando2
    .....
    
```

Target. Es el nombre del fichero que es generado al cumplirse la regla, y que tiene que ser recompilado cuando cambian las dependencias.

Dependencias. Son dependencias temporales en las que indicamos a make que la fecha de un fichero llamado *target* siempre debe de ser posterior a la fecha de los ficheros de los que depende, llamados dependencias.

P.e. para la dependencia:

```

miprograma : main.o primero.o segundo.o
    
```

Si make encuentra que la fecha de cualquiera de los ficheros *main.o*, *primero.o*, *segundo.o* es posterior a la de *miprograma*, ejecuta la compilación de *miprograma*. También se ejecuta la regla si *miprograma* no existe, que es equivalente a que su fecha sea anterior.

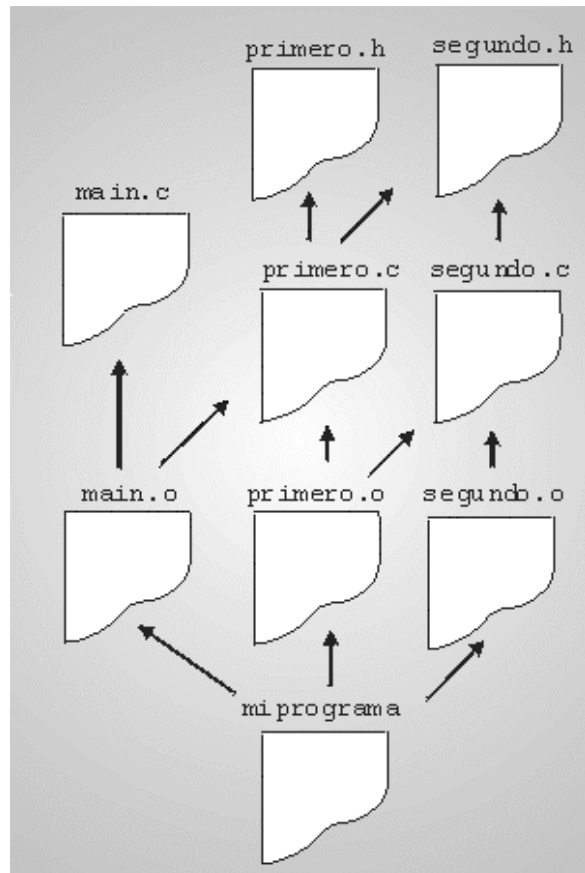


Figura 1: Ejemplo de dependencias

Para ello en general, como hemos visto antes, se escribe una línea en la que se separa el `target` de las dependencias por ":", de la forma:

```
target : dependencial dependencia2 dependencia3
```

Comandos. Se ejecutan cuando se cumple la regla. En el Listado 1 un comando sería:

```
gcc -o miprograma main.o primero.o segundo.o
```

Cada regla puede tener uno o más comandos, y los comandos deben obligatoriamente de empezar por tabulador. El olvidarse el tabulador es una de las principales causas de error al empezar a hacer ficheros `Makefile`.

Obsérvese que una regla puede tener ficheros en las dependencias que a su vez sean `target` de otras reglas, en cuyo caso se ejecuta la regla donde aparece como `target`, y luego la regla donde aparece como dependencia. P.e. en el Listado 1 `main.o` es `target` y dependencia, luego primero se ejecuta la regla en la que `main.o` aparece como `target` y luego la regla en la que aparece como dependencia.

Conviene aclarar que `make` no sólo vale para ejecutar el `gcc`, sino que se utiliza en todo tipo de proyectos en los que hay dependencias entre los ficheros.

P.e si estamos escribiendo un libro para el que hemos creado un fichero por tema, podemos concatenar todos los temas para crear un libro, y cada vez que modifiquemos un tema volver a generar el libro así:

```
libro.txt : tema1.txt tema2.txt tema3.txt
    cat tema1.txt tema2.txt tema3.txt libro.txt
```

Los programadores no sólo usan los ficheros `Makefile` para recompilar su programa, sino que también es muy común usarlos para generar los ficheros de `man` y para instalar la aplicación

En el caso de los proyectos de programas C, normalmente el ejecutable depende de los `.o` del proyecto, y los `.o` dependen de su correspondiente `.c` y de los ficheros `.h` que incluye el `.c`

Si alguna línea del `Makefile` es demasiado larga, podemos partir una línea en dos usando una `\` de la forma:

```
main.o : main.c primero.h segundo.h tercero.h cuarto.h \
quinto.h sexto.h septimo.h octavo.h noveno.h decimo.h
    gcc -c main.c
```

Respecto a los comandos, cada comando se ejecuta en un shell distinto, lo cual puede producir problemas en casos como este:

```
miprograma : visor/capa.cpp
    cd visor
    gcc -c capa.cpp
```

Ya que el cambio de directorio sólo tiene lugar para el primer comando, pero no para el segundo que se ejecuta en un shell distinto.

Para ejecutar ambos en un mismo shell podemos hacer:

```
miprograma : visor/capa.cpp
    cd visor; gcc -c capa.cpp
```

O si queremos que quede más ordenado ponemos:

```
miprograma : visor/capa.cpp
    cd visor;\
    gcc -c capa.cpp
```

1.2 El target final

El fichero `Makefile` del Listado 1 genera un árbol de dependencias como el de la Figura 1 donde para generar `miprograma`, a lo mejor antes puede tener que haber creado otros ficheros `.o` y ejecuta las reglas de generación de los `.o`, primero y por último la de generación de `miprograma`.

Se llama **target final** al último target que se ejecuta, para el que antes se pueden haber ejecutado otros target intermedios. Generalmente el target final será el programa ejecutable.

El target final debe ser la primera regla que aparezca en el `Makefile`, aunque si queremos que el target final sea otro distinto al del principio del `Makefile` podemos ejecutar `make` así:

```
$ make primero.o
```

Esto fijaría como target final al target intermedio `primero.o`

Pero puede pasar que necesitemos crear varios programas al ejecutar el `make`, y por defecto `make` sólo ejecuta las reglas necesarias para crear el primer target que aparece en el `Makefile`. Es decir, si nuestro proyecto consta de varios ejecutables como p.e:

```
miprograma1 : main.o primero.o segundo.o
             gcc -o miprograma1 main.o primero.o segundo.o

miprograma2: principal.o segundo.o
             gcc -o miprograma principal.o segundo.o
```

Al hacer `make` sólo se ejecutan las reglas necesarias para generar `miprograma1`, y `miprograma2` no se genera.

Podemos pedirle que ejecute las reglas necesarias para generar `miprograma2` pasando este fichero como target final de la forma:

```
$ make miprograma2
```

Pero cuando un proyecto está formado por varios ejecutables, se suele meter un falso target final llamado `all` de la forma:

```
all : miprograma1 miprograma2

miprograma1 : main.o primero.o segundo.o
             gcc -o miprograma1 main.o primero.o segundo.o

miprograma2: principal.o segundo.o
             gcc -o miprograma principal.o segundo.o
```

Ahora al hacer:

```
$ make
```

Se ejecuta la primera regla que es `all` que produce que se generen tanto `miprograma1` como `miprograma2`

1.3 Falsos targets

Un **falso target** es un target que sabemos que no existe, y que lo ponemos a posta para que la regla se ejecute siempre, ya que `make` piensa que el target es un fichero y al ver que no existe intenta ejecutar los comandos de la regla que lo generan.

Por ejemplo es muy típico encontrar en los ficheros `Makefile` la regla:

```
clean:
    rm *.o
```

Que borra todos los ficheros temporales de la compilación, y que se ejecutaría de la forma:

\$ `make clean`

A los falsos targets no se les suele poner dependencias, pero esto no suele dar problemas ya que la regla se ejecuta si el fichero del target no existe.

Esto sólo causaría problemas si en el directorio existe un fichero llamado `clean`, ya que al no existir dependencias inevitablemente se consideraría que el fichero `clean` está actualizado y no se ejecutarían los comandos de la regla. Para evitar ese problema se indica que el target es un fichero ficticio (phony en inglés) de la forma:

```
.PHONY: clean
clean:
    rm *.o
```

Otro sitio donde se deben usar los falsos targets es en la regla que usábamos para generar varios ejecutables:

```
.PHONY: all
all : miprograma1, miprograma2

miprograma1 : main.o primero.o segundo.o
    gcc -o miprograma1 main.o primero.o segundo.o

miprograma2: principal.o segundo.o
    gcc -o miprograma principal.o segundo.o
```

También conviene decir que otro target ficticio muy típico de encontrar en los Makefile es:

```
.PHONY: install
install : miprograma
    # Proceso de copia de los ficheros al
    # directorio destino
    .....
```

Y que en caso de no existir `miprograma` ejecuta las reglas que generan `miprograma` antes de instalarlo.

Nosotros nos podemos crear nuestras propias reglas como por ejemplo:

```
.PHONY: imprime
imprime:
    print main.c primero.c segundo.c
```

Cuando un falso target le ponemos como dependencia de otro falso target, la regla siempre se cumple, luego podemos usar falsos targets como subrutinas de otros de la siguiente forma:

```
.PHONY: cleanall cleanobj cleantxt
cleanall : cleanobj cleantxt
        rm miprograma

cleanobj :
        rm *.o

cleantxt :
        rm *.txt
```

Ahora al hacer:

```
$ make cleanall
```

Borramos los `.o`, los `.txt` y `miprograma`

1.4 Targets de seguimiento

Los **target de seguimiento**, son target que a diferencia de los falsos targets representan ficheros reales, pero que sólo se crean con la finalidad de que almacenen una fecha, la cual la guardan en el atributo de fichero destinado a almacenar la última fecha de modificación del fichero.

Los target de seguimiento se pueden usar de esta forma:

```
imprime_actualizados : main.c primero.c segundo.c tercero.c
primero.h segundo.h tercero.h
        lpr $?
        touch imprime_actualizados
```

Ahora si ejecutamos:

```
$ make imprime_actualizados
```

Sólo se imprimen los ficheros con fecha de modificación posterior al fichero `imprime`, y con `touch` actualizamos la fecha del fichero `imprime`.

En este caso `?$` (como veremos en el apartado 3.5) se expande por los ficheros con fecha posterior a `imprime_actualizados`.

1.5 Uso de comodines

En el `Makefile` podemos poner los comodines `*`, `?` y `[...]`, los mismos que en el shell. Los comodines se expanden cuando `make` parsea el fichero `Makefile` en el target, las dependencias y los comandos (en este último caso la expansión no la realiza `make`, sino el shell), pero no se expanden cuando aparecen en cualquier otro sitio del `Makefile`.

Por ejemplo, podemos imprimir los ficheros `.c` que han cambiado desde la última vez que los imprimimos con:

```
imprime_actualizados : *.c
    print $?
    touch imprime_actualizados
```

En este caso creamos el fichero de seguimiento `imprime_actualizados` y `*.c` se expande por todos los ficheros con esa extensión separados por espacio.

Recuérdese que `$?` se expande por los ficheros más nuevos que `imprime_actualizados`

2 Principales argumentos del comando `make`

`make` siempre busca un fichero llamado `makefile` y si no lo encuentra busca uno llamado `Makefile`. Tradicionalmente la mayoría de los programadores crean el fichero con el nombre `Makefile`, porque así aparece al principio del listado cuando hacemos un `ls`

Aun así podría darse el caso de que necesitemos crear varios ficheros `Makefile` para nuestro proyecto, y como sólo puede haber un fichero con el nombre `Makefile` en nuestro directorio ponemos usar el argumento de línea de comandos `-f fichero` o `--file=fichero` para indicar a `make` que ejecute un fichero con nombre distinto. P.e:

```
make -f Makefile2
```

Cuando alguno de los comandos de las reglas de `make` termina devolviendo un código de error, `make` se detiene y no sigue ejecutando. Nosotros podemos forzar a `make` a que siga ejecutando el fichero `Makefile` aunque encuentre un error con el argumento `-i` (`ignore`).

A veces lo que nos interesa es que continúe ejecutando el `make` aunque un determinado comando falle, en ese caso podemos preceder el comando con un `"-"`. P.e.: reacuérdese que en el ejemplo anterior borrábamos los ficheros `*.o` usando la regla:

```
.PHONY: clean
clean:
rm *.o
```

Pero si `rm` no encuentra ningún fichero `.o` devuelve un error. Sin embargo si ponemos el `"-"` delante de `rm`:

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

Ahora al ejecutarlo ignora el código de error que devuelve `rm`

Otra cosa a comentar es que pasa si falla la ejecución de un programa, o se detiene bruscamente (p.e. con `Ctrl+C`), en este caso el fichero que genera ese comando puede quedar creado en un estado defectuoso, lo cual causa un problema: El fichero mal creado tiene una fecha de actualización mayor a la de sus dependencias, pero sin embargo al ir a usar ese fichero por otro comando, el otro comando falla. Para solucionar este problema podemos poner al principio de nuestro `Makefile` la directiva:

```
.DELETE_ON_ERROR:
```

Que borra el target de la regla en caso de producirse un error en alguno de sus comandos.

Como hemos visto en los ejemplo cuando se ejecuta `make`, por defecto nos muestra las líneas de comandos que va ejecutando, podemos pedirle que no las muestre con el argumento `-s` (`silent`)

También podemos decir que no queremos que se saquen los mensajes con la directiva:

```
.SILENT:
```

Otras veces lo que queremos es no provocar el eco de un determinado comando. P.e. un caso típico es:

```
echo "Fase de compilación terminada con éxito"
```

Mostraría al ejecutarse el `make`:

```
echo "Fase de compilación terminada con éxito"  
Fase de compilación terminada con éxito
```

En este caso seguramente nos interese que salga el mensaje pero no el comando `echo`, y para ello precedemos al comando por una `@` de la forma:

```
@echo "Fase de compilación terminada con éxito"
```

También podríamos decir a `make` que nos imprima por consola los comandos que ejecutaría sin que los ejecute realmente, para ello usamos el argumento `-n`. P.e:

```
$ make -n  
gcc -c primero.c  
gcc -c main.c  
gcc -o miprograma main.o primero.o segundo.o
```

Sin embargo no nos ha ejecutado los `gcc`

Podemos indicar que determinado comando se ejecute aunque se use el argumento `-n` precediéndolo por un `+`. Por ejemplo podemos poner:

```
@+echo "Copyright 2004"
```

De forma que imprime el mensaje de copyright aunque no se ejecuten los comandos.

Por último también podemos poner comentarios en el fichero `Makefile` empezando la línea con una `#`, o bien poniendo la `#` al final de la línea que queremos comentar y después de la `#` poner el comentario. P.e:

```
#Esta es una línea de comentario  
rm *.o # Borro los ficheros .o antes de volver a generarlos
```

3 Macros

3.1 Introducción

Las **macros**, también conocidos como **variables** son identificadores que se sustituyen por su valor al parsear `make` el fichero `Makefile`.

Las macros se declaran en el fichero `Makefile` de la forma:

```
macro=valor
```

Y se expanden (sustituyen por su valor) después usando `$macro`, `$(macro)` o `${macro}`. La forma `$macro` no suele gustar a los que crean ficheros `Makefile` ya que dificulta la comprensión del `Makefile`, con lo que se aconseja usar `$(macro)` o `${macro}`.

Las macros se suelen utilizar en los `Makefile` para poner opciones que luego se cambiarán, p.e. durante el desarrollo se usa las opciones de depuración y para generar la versión final se usan opciones de optimización.

Las macros más usados en los `Makefile` son:

```
#Compilador a usar  
CC=gcc
```

Reacuérdesse que que en Mac OS X 10.1 se usaba el comando `cc` mientras que a partir de Mac OS X 10.2 se empezó a usar el comando `gcc`, a pesar de que `cc` se mantuvo por compatibilidad..

```
#Path de los ficheros de cabecera  
INCLUDE=.
```

Indica una lista (separada por comas) de directorios donde buscar los ficheros de cabecera

```
#Opciones usadas durante el desarrollo  
CFLAGS=-g -Wall -ansi
```

Indica los argumentos que se pasarán al compilador

Cuando vayamos a compilar la versión definitiva podemos cambiarlos por algo así:

```
#Opciones de la versión definitiva  
CFLAGS=-O -Wall -ansi
```

Para evitar escribir en varios sitios del `Makefile` la lista de ficheros `.o`, se suele crear la macro `OBJS` con los ficheros `.o` y cuando actualizamos el `Makefile` sólo actualizamos esta lista.

Es decir en vez de escribir:

```
miprograma : main.o primero.o segundo.o
    $(CC) $(CFLAGS) -o miprograma main.o primero.o
segundo.o
```

Se suele escribir:

```
OBJS=main.o primero.o segundo.o

miprograma : $(OBJS)
    $(CC) $(CFLAGS) -o miprograma $(OBJS)
```

Por último, también es muy frecuente encontrar el macro:

```
#Donde instalar la aplicacion
INSTALLDIR=/usr/local/bin
```

Usada para indicar donde instalar la aplicación al ejecutar:

```
$ make install
```

Después de lo que hemos aprendido podemos cambiar nuestro `Makefile` como muestra el Listado 2:

```
#Compilador a usar
CC=gcc

#Path de los ficheros de cabecera
INCLUDE=.

#Opciones usadas durante el desarrollo
CFLAGS=-g -Wall -ansi

#Lista de ficheros .o
OBJS=main.o primero.o segundo.o

#Donde instalar la aplicacion
INSTALLDIR=/boot/home/config/bin

miprograma : $(OBJS)
    $(CC) $(CFLAGS) -o $(OBJS)
    @+echo "Programa compilado correctamente. Copyright
MacProgramadores 2004"
main.o : main.c primero.h primero.o segundo.o
```

```

    $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
primero.o : primero.c primero.h segundo.h segundo.o
    $(CC) -I$(INCLUDE) $(CFLAGS) -c primero.c
segundo.o : segundo.c segundo.h
    $(CC) -I$(INCLUDE) $(CFLAGS) -c segundo.c

.PHONY: clean
clean:
    -rm *.o

.PHONY: install
install : miprograma
    @if [ -d $(INSTALLDIR) ]; \
    then \
        cp miprograma $(INSTALLDIR); \
        chmod a+x $(INSTALLDIR)/miprograma; \
        chmod og-w $(INSTALLDIR)/miprograma; \
        echo "Aplicacion instalada correctamente en \
$(INSTALLDIR)"; \
    else \
        echo "No se ha instalado la aplicacion porque \
$(INSTALLDIR) no existe"; \
    fi

```

Listado 2: fichero Makefile con macros

Por último saber que las macros también se pueden pasar como argumentos a make, en cuyo caso sobrescribe las macros que encuentra make en el fichero Makefile. P.e. cuando vayamos a sacar la versión definitiva podemos hacer:

```

$ make CFLAGS=-O
gcc -I. -O -c segundo.c
gcc -I. -O -c primero.c
gcc -I. -O -c main.c
gcc -O -o miprograma main.o primero.o segundo.o
Programa compilado correctamente. Copyright
MacProgramadores 2004

```

3.2 Macros predefinidas

`make` tiene una serie de macros predefinidas, que se enumeran en la Tabla 1, los cuales nosotros podemos sobrescribir volviéndolas a declarar en el fichero `Makefile`.

Otra forma de sobrescribir estas macros es declararlas en variables de entorno y ejecutar `make` con el argumento `-e`, como veremos en le siguiente apartado.

Nombre	Expansión	Descripción
MAKE	<code>make</code>	Programa que gestiona los <code>Makefile</code>
AR	<code>ar</code>	Programa para generar librerías estáticas, es decir ficheros <code>.a</code>
ARFLAGS	<code>-rv</code>	Flags de <code>ar</code>
YACC	<code>yacc</code>	Analizador sintáctico
YFLAGS		Flags de <code>YACC</code>
LEX	<code>lex</code>	Analizador léxico
LFLAGS		Flags del analizador léxico
LDFLAGS		Flags del enlazador <code>ld</code>
CC	<code>gcc</code>	Compilador predefinido de C
CXX	<code>g++</code>	Compilador predefinido de C++
CPP	<code>gcc -E</code>	Programa para realizar el preprocesado
CFLAGS		Flags del compilador de C
CXXFLAGS		Flags del compilador de C++
FC	<code>cc -O 1</code>	Compilador de Fortran

Tabla 1: Macros predefinidas

3.3 Fijar los valores de las macros desde fuera del `Makefile`

Podemos fijar el valor de una macro en una variable de entorno, de forma que si `make` no encuentra una macro que tiene que expandir declarada dentro del `Makefile` comprueba a ver si existe una variable de entorno con ese nombre y la utiliza.

En principio tienen prioridad las macros declaradas dentro del `Makefile` sobre las variables de entorno, pero podemos hacer que las variables de entorno tengan prioridad usando el argumento `-e` al ejecutar `make`. Este argumento se usa cuando queremos sobrescribir el valor de una macro del fichero `Makefile`.

Como hemos dicho antes, otra forma de sobrescribir una macro que tenemos declarada dentro del `Makefile` desde fuera es pasándola como argumento en la línea de comandos de la forma:

```
make CFLAGS="-O -Wall"
```

También existe una forma de que tenga prioridad la macro del `Makefile` sobre el comando, y para ello usamos la directiva `override` de la siguiente forma:

```
override CFLAGS += -g
```

Aunque esta directiva la debemos usar con precaución ya que sino no podrá luego nadie modificar una macro que quiera modificar sin modificar el `Makefile`. En el ejemplo anterior lo que estamos haciendo es que alguien puede modificar los flags de `CFLAGS`, pero pongan los que pongan en la opción nosotros al final añadimos la opción `-g`.

3.4 Macros de expansión única y macros de expansión recursiva

Las macros no se expanden (sustituyen por su valor) durante la asignación, sino durante su evaluación en alguna de las reglas. Esto nos permite hacer cosas como:

```
UNO := $(OTRO)
OTRO := $(AQUEL)
AQUEL := Fernando

culpable:
    @echo $(UNO)
```

Ahora al ejecutar:

```
$ make culpable
Fernando
```

Obsérvese que a la variable `UNO` le estamos asignando el valor `$(OTRO)` cuando todavía el parser de `make` no conoce `OTRO`, pero no hay problema porque `UNO` queda valiendo `$(OTRO)` y el valor de `UNO` se expande en la regla cuando ya el valor de `OTRO` ya si se conoce.

Llamamos **macro de expansión recursiva** a una macro que se expande tantas veces como haga falta hasta que no queden más expansiones pendientes. Estas macros se caracterizan porque se crean con el operador `=`, y son las únicas que hasta ahora conocemos.

Un problema que encontramos al expandir la macro recursivamente es que intentemos concatenar un valor a la macro de la forma:

```
CFLAGS = $(CFLAGS) -O
```

Resultaría en una expansión infinita y el `make` quedaría colgado (afortunadamente `make` lo detecta y da un warning).

Las **macros de expansión única** son macros que se expanden sólo una vez y que se crean con el operador `:=`

Las macros de expansión única evitan este problema ya que ahora podemos poner:

```
CFLAGS := $(CFLAGS) -O
```

Y la macro sólo se expande una vez.

Incluso podemos poner:

```
CFLAGS := $(CFLAGS) -O
CFLAGS := $(CFLAGS) -Wall
```

Y `CFLAGS` acabaría valiendo `-O -Wall`, ya que la macros se expande una vez por cada asignación.

En general se recomienda usar `:=` en vez de `=` ya que evita el problema anterior y no suele tener contraindicación. El único problema es que hay `make` antiguos que no reconocen `:=`

Otra forma alternativa de resolver el problema de la recursividad infinita es usando el operador `+=` que sirve para concatenar de la forma:

```
CFLAGS = -O
CFLAGS += -Wall
```

3.5 Macros automáticas

También hay unas macros internas cuyo valor depende de la regla que estemos ejecutando, y que se muestran en la Tabla 2.

Macro	Expansión
<code>\$\$</code>	Target de la regla que se está ejecutando
<code>\$\$*</code>	Target con el sufijo eliminado
<code>\$\$<</code>	Primer fichero de dependencia que permitió que la regla se ejecutase
<code>\$\$?</code>	Lista de ficheros de dependencias más recientes que el target
<code>\$\$^</code>	Lista de todas las dependencias

Tabla 2: Macros automáticas

`$$` nos permite saber el nombre del fichero target, luego p.e. la regla que genera el ejecutable `miprograma` la podríamos haber escrito así:

```
miprograma : $(OBJ)
             $(CC) $(CFLAGS) -o $$ $(OBJ)
```

`$(<` Indica el fichero dependencia que produjo que se ejecutara la regla. Es decir que era más nuevo que el target. En caso de que la regla tenga varios ficheros dependencias más modernos que el target, aquí se nos pasa el primero de los que cumplió con esta condición.

Debido a que en la lista de dependencias puede haber más de uno más modernos que el target, muchas veces en vez de `$(<` se usa `$(?`

`$(?` Es una lista de los ficheros de dependencias de la regla más modernos que el target, separados por espacio. P.e. podemos usar este macro para que `ar` archive los fichero `.o` más modernos que el target en una regla de la forma:

```
milib.a : main.o primero.o segundo.o
ar -rv $$ $(?)
```

Conviene destacar que un error muy típico al escribir una regla es intentar escribir:

```
main.o : main.c primero.h primero.o segundo.o
        $(CC) $? -I$(INCLUDE) $(CFLAGS) -c $$
```

Donde se usa `$(?` para referirse a las dependencias y `$$` para referirse al target. Aunque usar `$$` para referirse al target es correcto, no lo es el usar `$(?` para referirnos a las dependencias ya que se expande no por todas las dependencias, sino sólo por las que se han modificado (su fecha de actualización es posterior a la del target).

Si quisiéramos hacer esto deberíamos de utilizar `$(^` de la siguiente forma:

```
main.o : main.c primero.h primero.o segundo.o
        $(CC) $(^ -I$(INCLUDE) $(CFLAGS) -c $$
```

Si una variable automática está formada por un nombre de directorio más un fichero del directorio podemos sacar cada una de las partes con las macros automáticas de la Tabla 3:

Macro	Descripción
<code>\$(@D)</code>	El directorio de target
<code>\$(@F)</code>	El fichero del target

Tabla 3: Macros automáticas para fichero y directorio

P.e. si `$(@)` vale `fuentes/main.c` `$(@D)` valdría `fuentes` y `$(@F)` valdría `main.c`

En caso de que `$(@)` no tuviera directorio `$(@D)` valdría `."`

Análogamente podríamos sacar el directorio y fichero de las otras variables automáticas con: `$(*D)`, `$(*F)`, `$(<D)`, `$(<F)`, `$(?D)`, `$(?F)`, respectivamente.

3.6 Comodines en macros

Recuérdese que el comodín `*` sólo se expandía cuando lo poníamos en el target o en las dependencias, luego si creamos la variable:

```
OBJS := *.o
```

El comodín no se expande en la variable y si queremos que se expanda tenemos que usar la función `wildcard` de la siguiente forma:

```
OBJS:=$(wildcard *.o)
```

Obsérvese que el uso de comodines hubiera sido válido en el `Makefile` del Listado 2, donde podríamos haber puesto:

```
OBJS:=$(wildcard *.o)
```

En vez de:

```
OBJS=main.o primero.o segundo.o
```

de esta forma, cuando se añada un nuevo módulo al programa no tendremos que modificar esta variable, aunque tiene el inconveniente de que si borramos los `.o`, `OBJS` se expandirá por una cadena varia y el `Makefile` no funcionará como esperamos.

4 Compilación condicional

Podemos hacer que `make` lea un trozo de fichero `Makefile` o no en función de una condición. Téngase en cuenta que las sentencias de control que vamos a ver sirven para que `make` vea un trozo de fichero o no, pero no para controlar los comandos del shell en tiempo de ejecución.

Por ejemplo podemos hacer que si `CFLAGS` no tiene opciones se use la opción `-O -Wall` de la forma:

```
ifeq ($(CFLAGS),)
    CFLAGS = -O -Wall
endif
```

Las posibles condicionales que podemos usar son:

```
ifeq (valor1,valor2)
    Leido si son iguales
else
    Leido si son distintos
endif
```

También podemos comprobar si dos valores son distintos con:

```
ifneq (valor1,valor2)
    Leido si son distintos
else
    Leido si son iguales
endif
```

Por último también podemos comprobar si una macro está definida de la forma:

```
ifdef CFLAGS
    CFLAGS+= -Wall
else
    CFLAGS = -g -Wall
endif
```

O bien si un macro no está definido con:

```
ifndef CFLAGS
    CFLAGS = -g -Wall
else
    CFLAGS += -Wall
endif
```

5 Acceso al shell

Muchas veces necesitamos ejecutar un programa del shell para usar ese resultado con el fin de tomar una decisión. Un ejemplo típico es decidir sobre que plataforma se está ejecutando el `Makefile`. en este caso podemos usar el comando `arch` que nos dice la plataforma donde estamos (p.e. `ppc`, `i386`, `i486`, `i586`, `i686`, `alpha`, `sparc`, `arm`, `mips`, etc):

```
$ arch
ppc
```

Podemos obtener el resultado de ejecutar un comando en una variable de entorno de la forma:

```
VAR:= $(shell comando)
```

Por ejemplo si queremos usar un el flag `-Wno-long-double` sólo cuando estemos en una arquitectura PowerPC podemos usar:

```
ifeq ($(shell arch),ppc)
    CXXFLAGS += -Wno-long-double
endif
```

`shell` no sólo nos permite ejecutar un comando, sino todo un script del shell donde estemos trabajando. Por ejemplo, para obtener el shell que tiene instalado el usuario podemos usar:

```
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then \
    echo $$BASH; \
    else if [ -x /bin/bash ]; then echo /bin/bash; \
    else echo sh; fi ; fi)
```

6 Reglas implícitas

6.1 Como usar las reglas implícitas

Aunque nosotros hasta ahora hemos tenido que decir a `make` como obtener un fichero `target` (normalmente ejecutando el comando `gcc`), `make` tiene ya una serie de reglas implícitas para regenerar los tipos de fichero más conocidos.

Podemos saber que reglas trae `make` ya predefinidas con el comando:

```
$ make -p
```

Gracias a las reglas implícitas podemos cambiar nuestro `Makefile` así:

```
miprograma : $(OBJS)
    $(CC) $(CFLAGS) -o miprograma $(OBJS)
    @+echo "Programa compilado correctamente. Copyright
MacProgramadores 2004"
main.o : main.c primero.h primero.o segundo.o
primero.o : primero.c primero.h segundo.h segundo.o
segundo.o : segundo.c segundo.h
```

Y aunque no pongamos reglas para generar los ficheros `.o`, `make` sabe como generarlos y al ejecutar `make` obtenemos:

```
$ make
gcc -g -Wall -ansi -c segundo.c -o segundo.o
gcc -g -Wall -ansi -c primero.c -o primero.o
gcc -g -Wall -ansi -c main.c -o main.o
gcc -g -Wall -ansi -o miprograma main.o primero.o segundo.o
Programa      compilado      correctamente.      Copyright
MacProgramadores 2004
```

Incluso `make` expande, al aplicar las reglas implícitas las macros `CC` y `CFLAGS`, aunque la macro `INCLUDE` no la expande. Eso es porque las macros predefinidas que comentábamos en el apartado anterior, como `CC` o `CFLAGS`, son usadas por las reglas implícitas.

Además cada vez que ponemos como `target` un fichero `.o`, `make` pone el mismo nombre de fichero pero con el `.c` en las dependencias, es decir si tenemos el `target` `main.o` se pone en las dependencias `main.c` sin que nosotros se lo digamos, luego podemos reescribir las reglas anteriores así:

```
main.o : primero.h primero.o segundo.o
primero.o : primero.h segundo.h segundo.o
segundo.o : segundo.h
```

Otra cosa que podemos hacer es generar un fichero ejecutable a partir de un único `.c` sin usar un fichero `Makefile` de la forma:

```
$ make programita  
gcc programita.c -o programita
```

Como vemos `make` busca un fichero llamado `programita.c` que dará lugar al ejecutable `programita`

Y si el fichero se llamara `programinta.cpp`, `make` usaría el compilador de C++ para este programa:

```
$ make programinta  
g++ programita.cpp -o programita
```

También existe otra regla que transforma los ficheros `.cpp` en ficheros `.o`

P.e. si cambiamos el `Makefile` así:

```
main.o : main.cpp primero.h primero.o segundo.o
```

Y ejecutamos de nuevo el `make` obtenemos:

```
$ make  
gcc -g -Wall -ansi -c segundo.c -o segundo.o  
gcc -g -Wall -ansi -c primero.c -o primero.o  
g++ -c main.cpp -o main.o  
gcc -g -Wall -ansi -o miprograma main.o primero.o segundo.o  
Programa compilado correctamente. Copyright  
MacProgramadores 2004
```

En este caso utiliza el comando `g++` que es el que dice su regla para los ficheros de C++.

También existe regla implícita par los fichero `.cc` (Ficheros C++ de UNIX), que también lanza el `g++` así:

```
g++ -c main.cc -o main.o
```

6.2 Crear reglas implícitas

Un problema que tiene `make` es que no tiene regla implícita para los ficheros `.CPP` que traemos de Windows (donde no se distingue mayúsculas de minúsculas).

Le podemos decir a `make` que convierta la extensión a minúsculas a la vez que compilamos el fichero. Para ello tenemos que enseñar a `make` como compilarlo creando nuevas reglas implícitas en el fichero `Makefile`, lo que se llama **reglas patrón implícitas** (implicit pattern rules).

Las reglas patrón implícitas son como las reglas normales, sólo que utilizan el comodín `%` para representar uno o más caracteres.

Por ejemplo la regla patrón implícita `%.o : %.c` nos permite indicar como crear un fichero `.o` a partir de un `.c`

Para solucionar el problema de que `make` compile los ficheros `.CPP` con una regla implícita escribiríamos lo siguiente:

```
%.o : %.CPP
    mv $< $*.cpp
    g++ $(CFLAGS) -I$(INCLUDE) -c $<
```

El orden en que aparecen las reglas patrón implícitas en el `Makefile` es importante porque este es el orden en que son consideradas por `make`, y si se pueden aplicar dos reglas, `make` aplica sólo la primera que cumple con el patrón.

Luego si creamos reglas patrón de la forma:

```
%1.o : %1.c
    comandos para ficheros de tipo %1

%2.o : %2.c
    comandos para ficheros de tipo %2

%.o : %.c
    comandos genéricos para cualquier nombre de fichero
```

Podemos tener una forma de crear los ficheros de tipo `%1` y otra para los de tipo `%2`. Y como última regla ponemos una forma general de crear los ficheros independientemente del nombre que tengan, que sería como un comodín para cuando no se han cumplido las dos reglas anteriores.

Si el patrón del target tiene una `/` indicando que el target está metido en un subdirectorio, `make` elimina el directorio para realizar la comparación con el patrón, y luego lo vuelve a añadir. Luego si creamos la regla:

```
miprograma : fuente/main.o
    $(CC) $? -o miprograma
```

Y en el directorio fuente hemos metido el fichero `main.cpp`, `make` encontrará que este fichero cumple con la regla patrón implícita que definimos antes y ejecutará:

```
g++ -I -c fuente/main.cpp
```

Ahora, según esta regla que hemos aprendido podemos colocar todos nuestros ficheros fuente en el directorio fuente y crear la regla:

```
objs/%.o : fuente/%.cpp
    $(CXX) $(CXXFLAGS) -I$(INCLUDE) -c $< -o $@

objs/%.o : fuente/%.c
    $(CC) $(CFLAGS) -I$(INCLUDE) -c $< -o $@

miprograma : objs/main.o objs/primero.o objs/segundo.o
    $(CC) $? -o $@
```

Al ejecutar `make` tendríamos:

\$ **make**

```
g++ -I -c fuente/main.cpp -o objs/main.o
cc -I -c fuente/primero.c -o objs/primero.o
cc -I -c fuente/segundo.c -o objs/segundo.o
cc objs/main.o objs/primero.o objs/segundo.o -o miprograma
```

Por último decir que podemos redefinir reglas patrón implícitas ya existentes o cancelarlas con sólo volver a escribir la regla patrón de otra forma o bien escribirla vacía.

P.e. si queremos que no genere ficheros `.o` a partir de un `.c` implícitamente hacemos:

```
%.o : %.c
```

7 Tener un proyecto repartido en varios subdirectorios

Cuando el proyecto es grande, se suelen crear varios directorios destinados a almacenar los distintos módulos de nuestro programa. Vamos a ver una serie de trucos respecto a como podemos gestionar en este tipo de proyectos.

7.1 Buscar dependencias en varios subdirectorios con la directiva `vpath`

Podemos indicar a `make` que además de en el directorio actual busque los ficheros que actúan como dependencias de las reglas en otros subdirectorios con la directiva `vpath`

Esta directiva nos permite indicar donde buscar los ficheros que cumplan con un determinado patrón que se suelen colocar en un directorio distinto (p.e. los `.h`).

La directiva tiene tres formas:

```
vpath patron directorio
```

Indica donde buscar los ficheros que cumplan con el patrón `patron`

```
vpath patron
```

Borrar el patrón de búsqueda `patron`

```
vpath
```

Borrar todos los patrones de búsqueda previamente definidos

Los patrones usan el comodín `%`, ya que `*` es un comodín definido para expandirse en los ficheros que cumplen ese comodín tal como explicamos antes.

Un ejemplo de patrón de búsqueda muy usado es:

```
vpath %.h ../cabeceras
```

De forma que si en el `Makefile` aparece como dependencia un fichero `.h` que no se encuentra en el directorio, se busca en este otro directorio.

Sin embargo aunque `make` encuentra los ficheros `.h` del directorio `../cabeceras`, al ejecutar `gcc`, éste fallará si no hemos indicado el directorio `../cabeceras` en el argumento `-I` de `gcc`.

7.2 La variable de entorno `VPATH`

Otra forma de indicar los directorios donde buscar ficheros que no aparezcan en el directorio actual es con la variable de entorno `VPATH`

El orden de búsqueda que sigue `make` es el siguiente:

1. Busca en el directorio actual
2. Busca en los directorios que diga `vpath`, si el nombre del fichero cumple con el patrón.
3. Busca en los directorios que diga la variable de entorno `VPATH`

7.2.1 Incluir otros ficheros

Un fichero `Makefile` puede incluir otros ficheros con:

```
include fichero
```

Esto se suele hacer cuando hay opciones o macros que queremos tengan todos nuestros `Makefile`, en cuyo caso se incluye el fichero donde hemos puesto las opciones comunes.

El problema es que para que `include` encuentre el fichero tiene que estar en nuestro directorio, con lo que dentro de nuestro directorio se suele crear un enlace al fichero de las opciones.

Otra posibilidad es usar las opciones `-I camino` o `--include-dir camino` para indicar donde buscar el fichero a incluir.

La otra forma de incluir ficheros es, en vez de usar la directiva `include`, poner uno o más ficheros en la variable de entorno `MAKEFILES`, (separados por espacio si hay más de uno), de forma que si esta variable de entorno existe `make` incluye los ficheros puestos aquí antes de procesar el fichero `Makefile`.

7.3 Tener librerías organizadas en subdirectorios

A veces se crean subdirectorios destinados a almacenar una librería en cada uno de ellos, donde cada librería está formada por varios ficheros `.o`

En este caso se suele crear un `Makefile` en el subdirectorio con las reglas que compilan esa librería y en el directorio raíz se coloca un `Makefile` con reglas que ejecutan los ficheros `Makefile` de los subdirectorios.

Para ejecutarlo se crean reglas de la forma:

```
milib.a :  
    cd dirmilib;$(MAKE)
```

Recuérdese que las reglas sin dependencia se ejecutan siempre, luego esta regla se ejecutara siempre y lo que hará será llamar a `make` con el `Makefile` del subdirectorio. Ya vimos en el apartado 3.2 que la macro predefinida `MAKE` representaba el `make` que estábamos usando y por defecto valía `make`.

Otra consideración nueva que tenemos que hacer en este caso es que cada comando de la regla se ejecuta en un shell distinto, luego el cambio de directorio con `cd dirmilib` de un shell no afecta al siguiente shell que ejecuta `$(MAKE)`. Para que esto no ocurra vimos que se podían poner las instrucciones del `make` separadas por punto y coma.

7.4 Paso de variables a un sub-make

A veces cuando llamamos a un `make` para un subdirectorio queremos pasarle opciones en variables, una técnica muy usada es pasarlas en variables de entorno.

Recuérdese que si creamos variables de entorno `make` primero busca una macro definida en el fichero `Makefile` y si no la encuentra busca una variable de entorno con ese nombre.

Nosotros podemos fijar el valor del parámetro a pasar al sub-make en una variable de entorno, pero la variable de entorno tenemos que exportarla de la forma:

```
$ export OPCION=valor
```

Ya que como `make` ejecuta el sub-make en un shell distinto, si no lo exportamos no recibe la nueva variable de entorno.

También podemos hacer que el sub-make no reciba una variable de entorno con:

```
$ unexport OPCION=valor
```

Otra forma de pasar parámetros a un sub-make es fijando una variable en la línea de comandos de la forma:

```
milib.a :  
    (cd dirmilib;$(MAKE) ARFLAGS=-rv)
```

8 Obtener las reglas con `gcc`

La herramienta `gcc` tiene el argumento `-MM` que analiza los ficheros de entrada y nos devuelve las reglas que deberíamos poner en el fichero `Makefile`.

Esta opción es muy recomendable usarla ya que de esta forma estamos seguros de no olvidarnos poner ninguna dependencia ni poner dependencias de más.

```
$ gcc -MM main.c primero.c segundo.c
main.o : main.c primero.h
primero.o : primero.c primero.h segundo.h
segundo.o : segundo.c segundo.h
```

Estas reglas devueltas las copiamos en el `Makefile` y ya está listo.

También podemos usar el argumento `-M`, que nos mete también las dependencias respecto a los ficheros `.h` estándar de C, con lo que es menos usada.