Make

Juan José Moreno Moll

1. Introducción 3
2. Fichero *makefile* 3
2.1. Ejemplo de *makefile* sencillo 4
2.2. Variables de un *makefile* 6
2.3. Reglas implícitas 7
2.4. Otras características 8
3. Escribir un fichero *makefile* 9
3.1. Contenidos de un *makefile* 9
3.2. Nombres de *makefile* 9
3.3. Incluir otros *makefiles* 10
4. Reglas en un *makefile* 10
4.1. Sintaxis de las reglas 10
4.2 Caracteres Comodín 10
4.3. Reglas con varias etiquetas. Reglas múltiples 11
4.4. Reglas de patrones estáticos 12

1. Introducción

Un programa escrito en C normalmente está compuesto por muchos archivos. Estos archivos se van modificando según se va completando o cambiando el programa. Cada vez que se modifica algún archivo del programa, éste debe recompilarse para generar un ejecutable del programa actualizado. La modificación de un archivo del programa no implica que se deban recompilar todos los archivos del programa. Sólo se debe recompilar el fichero modificado así como otros ficheros que puedan depender del fichero modificado.

La utilidad **make** determina automáticamente qué archivos del programa deben ser recompilados y las órdenes que se deben utilizar para recompilarlos. En programas muy grandes nos ayuda a mantener el ejecutable totalmente actualizado. No debemos recordar las dependencias entre archivos del programa ni las órdenes necesarios para compilar cada parte del programa.

La utilidad **make** no sólo sirve para compilar programas en C. También se puede utilizar:

1) con otros lenguajes cuyo compilador se pueda ejecutar en una 'shell' (en la línea de ordenes).
2) para tareas de instalación y actualización de aplicaciones.

Para usar el **make** debemos escribir un fichero llamado *makefile*. Este fichero describe las relaciones entre ficheros y las ordenes que se deben ejecutar con cada archivo.

Si en un directorio existe el fichero *makefile* para ejecutar la utilidad **make** simplemente debemos escribir en la línea de órdenes:

c:\..> make

Con esta orden **make** busca el fichero *makefile* y se encarga de ejecutar las ordenes contenidas en el fichero *makefile*.

2. Fichero *makefile*

La utilidad **make** normalmente ejecuta las reglas contenidas en un archivo llamado *makefile*. Es posible indicar a **make** que lea reglas de ficheros con un nombre diferente a éste, como se verá más adelante.

Un *makefile* simple está compuesto por reglas que tienen la siguiente forma:

```
etiqueta ... : dependencias...
        orden
        orden
        ...
```

Una *etiqueta* es un rótulo que generalmente se refiere a un nombre de fichero. Este fichero es aquel que queremos actualizar con esta regla. También hay etiquetas que indican la acción que la regla realiza. Por ejemplo *limpiar* (ver en el ejemplo siguiente).

Una *dependencia* es un fichero que se usa en la regla. Si este fichero se modifica la regla se active. Una regla puede depender de varios ficheros.

Una *orden* es la acción que make ejecuta si la regla se activa. Una regla puede ejecutar más de una orden. **Nota muy importante**: Una línea con una **orden empieza** siempre con un **tabulador** (no por varios caracteres espacio).

Una regla tiene un nombre (*etiqueta*) y dice como rehacer un fichero (mediante ordenes) si algún fichero del que depende se modifica (*dependencias*).

2.1. Ejemplo de *makefile* sencillo

Éste es un ejemplo de un *makefile* que permite generar un ejecutable llamado *prog.exe* a partir de un programa en C que tiene 3 ficheros: el programa principal *prog.c*, un archivo fuente *vector.c*, y su archivo de cabecera *vector.h.*

```
todo: prog.exe

vector.o:   vector.h   vector.c
	gcc   -c  vector.c   -Wall

prog.o:  prog.c    vector.h
	gcc   -c    prog.c   -Wall

prog.exe:   prog.o   vector.o
	gcc   prog.o   vector.o   -o prog.exe

limpiar:
	del   prog.o
	del   vector.o
	del   prog.exe
```

Este *makefile* permite hacer dos cosas: crear un ejecutable actualizado del programa, y borrar el fichero ejecutable y todos ficheros objetos del programa.

Para crear el ejecutable actualizado del programa hay que escribir en la línea de órdenes:

```
c:\..>  make
```

Veamos que ocurre si ejecutamos la línea anterior.

En el ejemplo anterior tenemos una primera regla llamada 'todo'. Esta regla no tiene órdenes y sí tiene una dependencia 'prog.exe'. Esta regla nos indica sólo que hay que ver si el fichero *prog.exe* está actualizado. La regla que se encarga de actualizar el fichero *prog.exe* está más abajo y tiene como etiqueta 'prog.exe'.

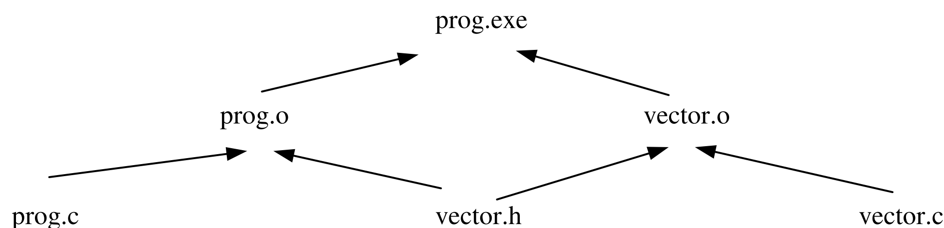
La segunda regla tiene como etiqueta 'vector.o'. La orden que contienen la regla es: *gcc -c vector.c -Wall* que hace que el compilador construya el fichero objeto *vector.o*. Y tiene las dependencias *vector.c* y *vector.h*. Esta orden se ejecutará si el fichero *vector.c* o el fichero *vector.h* han sido modificados.

La tercera regla es semejante a la anterior.

La cuarta regla tiene como etiqueta a 'prog.exe'. La orden de la regla hace que se construya el fichero ejecutable 'prog.exe'. Y sus dependencias son *prog.o* y *vector.o*. Esta regla se lanzara si los ficheros indicados en las dependencias han sido modificados.

El funcionamiento de este *makefile* es el siguiente. Al ejecutar la utilidad **make** sin argumentos se activa la primera regla del *makefile*. Esta regla es 'todo'. Esta regla no tiene órdenes que ejecutar, pero sí tiene una dependencia, 'prog.exe'. La regla 'todo' lo único que hace es activar la regla 'prog.exe'. La regla 'prog.exe', a su vez, se activa si los ficheros *prog.o* y *vector.o* se han modificado. Antes de ejecutar la orden asociada a la regla 'prog.exe' se comprueban si es necesario activar las reglas asociadas a los ficheros 'vector.o' y ' prog.o'. La regla 'prog.exe' llama a las reglas 'vector.o' y 'prog.o'. La regla 'vector.o' tiene dependencias de los ficheros *vector.c* y *vector.h*. Como se ve ya no hay reglas asociadas a estos ficheros por lo que la orden asociado a la regla 'vector.o' se ejecutará si alguno de los ficheros indicados en sus dependencias han sido modificado. La regla 'prog.o' funciona exactamente igual que la regla 'vector.o'.

Para generar el fichero *prog.exe* en primer lugar se observa si se ha modificado alguno de los siguientes ficheros: *vector.c, vector.h y prog.c*. En caso afirmativo se actualizan los ficheros objetos *vector.o* y *prog.o*. Y en último lugar se construye el fichero ejecutable *prog.exe* a partir de los ficheros objetos previamente actualizados. Un esquema de como el *makefile* indica las dependencias entre los ficheros que forman el programa y el proceso de actualización que sigue hasta construir un ejecutable *prog.exe* acorde con los ficheros fuente (**.c y *.h*) que se están manejando en ese momento puede ser:



Como se ve al ejecutar la utilidad **make** se activan todas las reglas del *makefile* excepto la regla 'limpiar'. Esta regla se utiliza para eliminar los ficheros objeto y el fichero ejecutable y puede tener sentido para forzar una compilación completa del programa. La línea de órdenes necesaria para borrar el ejecutable y los objetos es:

c:\..> make limpiar

La utilidad **make** sin argumentos ejecuta la primera regla del *makefile*. Para ejecutar una regla distinta hay que pasarle como argumento el nombre de la regla que queremos activar. Si por ejemplo sólo quisiéramos obtener el fichero *vector.o* podríamos ejecutar:

c:\..> make vector.o

2.2. Variables de un *makefile*

En el *makefile* anterior hemos usado en varias líneas la orden 'gcc' y la lista de objetos 'vector.o prog.o'. Estas palabras se repiten muchas veces por dentro del texto del *makefile*. Estas cadenas se pueden sustituir por variables que se definen dentro del *makefile.*

Una variable se define de la forma siguiente:

nombre_variable= lista palabras a las que sustituye

Y se hace referencia a una variable dentro del *makefile* de la forma siguiente:

$(nombre_variable)

Dentro de un *makefile* cada vez que se encuentre la referencia a una variable ésta equivale a la lista de palabras indicadas en el momento en que se definió la variable. El *makefile* anterior se puede sustituir por uno equivalente como el siguiente:

```
obj= vector.o   prog.o
CC= gcc
WARNING = -Wall
LIB=


todo: prog.exe

vector.o:   vector.h   vector.c
	$(CC)   -c  vector.c   $(WARNING)

prog.o:  prog.c    vector.h
	$(CC)   -c    prog.c   $(WARNING)

prog.exe:   $(obj)
	$(CC)    $(obj)   -o prog.exe  $(LIB)
```

```
limpiar:
      del   prog.exe
      del  $(obj)
```

Las variables facilitan la escritura y evitan repetir lista de palabras iguales en líneas distintas (con el peligro de que no todas las líneas tengan la misma lista de palabras). También facilitan los cambios de un *makefile*. Obsérvense las variables CC, WARNING y LIB. Si quisiéramos utilizar otro compilador (por ejemplo 'turboc') sólo deberemos modificar el valor de la variable CC. Lo mismo ocurriría si quisiéramos modificar el tipo de 'warnings' que queremos conocer al compilar. Sólo habría que modificar la variable WARNING. O si necesitáramos añadir alguna librería para enlazar ('linkar'), sólo deberíamos añadir a la definición de la variable LIB las librerías necesarias.

2.3. Reglas implícitas

Cuando se compila en C no es necesario añadir una orden a una regla. Si la regla tiene una etiqueta del tipo **.o* **make** entiende que se debe ejecutarse una orden del tipo *gcc -c *.c*. Esto es lo que se llama una regla implícita. Por ejemplo, el *makefile* anterior se puede sustituir por otro con reglas implícitas:

```
obj= vector.o   prog.o
CC= gcc
WARNING = -Wall
LIB=


todo: prog.exe

vector.o:   vector.h

prog.o:  vector.h

prog.exe:   $(obj)
      $(CC)    $(obj)   -o prog.exe  $(LIB)

limpiar:
      del   prog.exe
      del  $(obj)
```

En este caso el *makefile* no es totalmente equivalente al anterior. Cuando se construyen los ficheros **.o* no se utiliza la variable WARNING.

Cuando los ficheros objeto son creados por reglas implícitas es posible agrupar reglas que tengan una misma dependencia. Por ejemplo:

```
obj= vector.o   prog.o
CC= gcc
WARNING = -Wall
LIB=


todo: prog.exe

prog.o vector.o:   vector.h

prog.exe:   $(obj)
        $(CC)    $(obj)   -o prog.exe  $(LIB)

limpiar:
        del   prog.exe
        del  $(obj)
```

O también:

```
obj= vector.o   prog.o
CC= gcc
WARNING = -Wall
LIB=


todo: prog.exe

$(obj):   vector.h

prog.exe:   $(obj)
        $(CC)    $(obj)   -o prog.exe  $(LIB)

limpiar:
        del   prog.exe
        del  $(obj)
```

2.4. Otras características

Reglas para borrar

Una de las reglas del *makefile* anterior era 'limpiar' que se encarga de borrar ficheros. Esta regla puede causar problemas si apareciera un fichero llamado *limpiar* en el directorio. Para evitar este problema las reglas para borrar ficheros se deben escribir:

```
.PHONY:   limpiar
limpiar:
```

```
del   prog.exe
del   $(obj)
```

El porqué de esta forma de escribir las reglas para borrar ficheros se puede encontrar en el manual de la utilidad **make**.

Cómo utilizar make con ficheros cuyo nombre no es *makefile*

La utilidad **make** puede utilizar cualquier fichero que contenga reglas como las anteriormente descritas. Por defecto **make** siempre busca ficheros con el nombre *makefile*. Si queremos que ejecute reglas de un fichero con otro nombre se debe utilizar la opción **-f**. Por ejemplo si tengo un fichero llamado 'regla.txt' y quiero ejecutar las reglas contenidas en él, la línea de orden que se utilizaría es:

```
c:\..> make   -f    reglas.txt
```

3. Escribir un fichero *makefile*

La información que dice a **make** como recompilar o realizar otras tareas con un programa está escrita en un fichero llamado *makefile*.

3.1. Contenidos de un *makefile*

Un *makefile* contiene cinco tipo de entidades: reglas explícitas, reglas implícitas, definición de variables, directivas y comentarios:

* Reglas explícitas: Son reglas que dicen cómo y cuando aplicar ciertos órdenes a ficheros. Las órdenes que deben ejecutarse se encuentran escritos en la regla.
* Reglas implícitas: Son reglas que dicen cómo y cuándo aplicar ciertos órdenes a ciertos ficheros en función del nombre y/o la extensión de esos ficheros. Las órdenes que deben ejecutarse dependen del nombre del fichero sobre el que se aplican. La orden no aparece en la regla.
* Definición de variables: es una línea de texto que indica que una cierta variable equivale a una cadena de texto. En el texto del *makefile* siguiente a esta definición escribir esa variable equivale a escribir la cadena de texto a la que se ha asociado en la definición.
* Directivas: Realizan tareas especiales como incluir otros ficheros, decidir cuando ciertas partes del *makefile* deben o no ignorarse, etc.
* Comentarios: Son líneas que comienzan por el carácter '#'.

3.2. Nombres de *makefile*

Cuando se ejecuta la orden **make,** éste busca los ficheros con los nombres: *makefile, Makefile,* por este orden.

Si queremos utilizar un nombre distinto de los anteriores para referirnos a un fichero de tipo *makefile* debemos utilizar la opción '-f' de la orden **make**.

c:\...> make -f reglas.txt

3.3. Incluir otros *makefiles*

La forma de incluir otros *makefiles* (u otros ficheros) es utilizar la directiva *include*. La forma de utilizar esta directiva es la siguiente:

include fichero1 fichero2 ...

4. Reglas en un *makefile*

4.1. Sintaxis de las reglas

La síntaxis de una regla es:

```
etiqueta ... : dependencias...
	orden
	orden
	...
```

Una regla dice dos cosas: cuándo un fichero está fuera de fecha y por tanto si hay que actualizarlo (si hay que activar o no la regla) y como debe actualizarse este fichero.

La *etiqueta* es usualmente uno o varios nombres de ficheros separados por espacios. Puede haber *etiquetas* que no sean nombres de ficheros. Generalmente se utiliza un solo nombre por regla.

Las órdenes son líneas que comienzan con un carácter '\tab' e indican que hacer cuando la regla se activa.

Los criterios para saber si un fichero está fuera de fecha (o si hay que activar una regla) está especificado en las *dependencias*. Las *dependencias* consisten en una serie de nombres de ficheros separados por espacios. Una *etiqueta* está fuera de fecha si es más antiguo que cualquiera de los ficheros indicados en las *dependencias*.

4.2 Caracteres Comodín

Es posible utilizar los caracteres comodín '*' y '?'. Tienen el mismo significado que en una 'shell'. Por ejemplo *.c indica la lista de ficheros del directorio de trabajo que terminan con '.c'.

Ejemplos:

```
limpiar:
	delete   *.o   *.exe
```

También es posible utilizarlos en definición de variables:

```
obj = *.o

...

$(obj):   vector.h

prog.exe:   $(obj)
	$(CC)    $(obj)   -o prog.exe  $(LIB)
```

Aunque en este caso es mejor utilizarlos a través de la función ‘wildcard’:

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

...

$(obj):   vector.h

prog.exe:   $(obj)
	$(CC)    $(obj)   -o prog.exe  $(LIB)
```

4.3. Reglas con varias etiquetas. Reglas múltiples

Una regla que tiene varias etiquetas equivale a escribir varias reglas, cada una de ellas con una sola etiqueta. Las reglas múltiples se suelen hacer cuando tenemos varias reglas simples que tienen un formato equivalente. Así por ejemplo la regla:

```
vector.o   prog.o:    vector.h
	...
```

Equivale a las dos reglas:

```
vector.o:     vector.h
	...
prog.o:       vector.h
	...
```

Las reglas múltiples se ejecutan tantas veces como etiquetas tiene la regla. Cada vez que se ejecuta la regla múltiple se refiere a una etiqueta de la lista. Para saber en cada momento de la ejecución que etiqueta es la que se está activando tenemos la variable: '$@'. Esta variable sustituye al nombre de la etiqueta actual en la línea de órdenes. Así la regla múltiple anterior se puede escribir:

```
vector.o   prog.o:    vector.h
        $(CC)  -c   $(subst  .o,,$ @).c  -o  $@   $(WARNING)
```

La función $(subst cad1, cad2, texto) lo que hace es sustituir en la cadena 'texto' cada aparición de la subcadena 'cad1' por la subcadena 'cad2'. Así por ejemplo: $(subst ee, EE, peepe no eesta) da como resultado la cadena : 'pEEpe no EEsta'.

En este caso la función $(subst .o, , $@) significa: sustituir en la cadena $@ (que puede ser 'vector.o' ò 'prog.o') la subcadena '.o' por la cadena vacía. Es decir a los valores de $@ les eliminamos la terminación '.o'. En la línea de órdenes después de hacer esto se la añade al resultado la terminación '.c': $(subst .o, , $@)**.c.**

4.4. Reglas de patrones estáticos

Existen reglas múltiples en las que las dependencias no son comunes a todas las etiquetas. Cada etiqueta tiene una dependencia que está asociada a su nombre.

En el caso anterior (Ver 4.3) se observa que la regla múltiple tiene una única dependencia 'vector.h'. Esto no es totalmente correcto. La dependencia debería ser 'vector.h' tanto para 'vector.o' como para 'prog.o' y además cada fichero '*.o' debería depender de su fichero fuente correspondiente '*.c'. Si quisiéramos hacer una regla en la que intentamos hacer que haya una dependencia de los ficheros '*.c' tendríamos que escribir:

```
vector.o   prog.o:   vector.h   vector.c   prog.c
        $(CC)  -c   $(subst  .o,,$ @).c  -o  $@   $(WARNING)
```

Si escribimos la regla de esta forma ocurre que se actualiza 'vector.o' si el fichero 'prog.c' se ha modificado y que se actualiza 'prog.o' si el fichero 'vector.c' se ha modificado, lo cual no es correcto.

Las reglas de patrones estáticos resuelven este problema. La sintaxis de este tipo de reglas es:

```
etiquetas ...:  etiqueta-patrón:    dependencias-patrón
                orden
                ....
```

etiquetas es una lista de ficheros a la que se va aplicar la regla. Las *etiquetas-patrón* y las *dependencias-patrón* indica el tipo de dependencia que se da para cada etiqueta. Cada *etiqueta* se compara con la *etiqueta-patrón*. De esta comparación se obtiene una cadena que es una parte de la *etiqueta* y que llamamos *raíz*. Esta cadena se sustituye en las *dependencias-patrón* para construir las dependencias de cada *etiqueta*. Tanto los *etiquetas-patrón* y las *dependencias-patrón* contienen al carácter '%' que sustituye a cualquier parte de la *etiqueta* y como hemos dicho la subcadena que es sustituida por '%' se llama *raíz*. Por ejemplo la comparación de 'vector.o' y de '%.o' da como *raíz* 'vector'. Las dependencias se obtienen sustituyendo el valor obtenido para '%' (la *raíz*) comparando las *etiquetas* con las *etiquetas-patrón* en cada *dependencia-patrón*. Por ejemplo si tenemos una *etiqueta* 'vector.o', una *etiqueta-patrón* '%.o' y una *dependencia-patrón* '%.c' obtenemos un valor de '%' 'vector' (*raíz*) y por lo tanto la dependencia es 'vector.c'.

Para referirnos a las etiquetas y a las dependencias que se obtienen en cada ejecución de la regla múltiple tenemos las variables '$@' que se refiere a la etiqueta y '$<' que se refiere a la dependencia. Por ejemplo el programa del punto **2.** lo podemos escribir:

```
obj= vector.o   prog.o
CC= gcc
WARNING = -Wall
LIB=


todo: prog.exe

$(obj):  %.o:  %.c  vector.h
	$(CC)  -c  $<  -o  $@     $(WARNING)

prog.exe:   $(obj)
	$(CC)    $(obj)   -o prog.exe  $(LIB)

limpiar:
	del   prog.exe
	del  $(obj)
```

En este caso cuando '$(obj)' vale 'vector.o', la *raíz* vale 'vector', '$<' vale 'vector.c' y '$@' vale 'vector.o'.

Y cuando '$(obj)' vale 'prog.o', la *raíz* vale 'prog', '$<' vale 'prog.c' y '$@' vale 'prog.o'