Make

Un programa para controlar la recompilación

Gerardo Aburruzaga García
gerardo.aburruzaga@uca.es

Índice

1. Introducción a make **2**
1.1. Cómo leer este manual . . . 3

2. Un ejemplo sencillo de *makefile* **3**
2.1. Cómo make procesa este *makefile* . . . 5

3. Las macros simplifican el *makefile* **5**
3.1. Macros predefinidas . . . 6
3.2. Dejemos que make deduzca las órdenes . . . 6
3.3. Otro estilo de *makefile* . . . 7
3.4. Reglas para limpiar el directorio . . . 8

4. Cómo escribir un *makefile* **9**
4.1. Contenido de un *makefile* . . . 9
4.2. Qué nombre dar al *makefile* . . . 9

5. Cómo escribir las reglas **10**
5.1. Sintaxis de una regla . . . 10
5.2. Objetivos falsos . . . 11
5.3. Ficheros vacíos como objetivos . . . 12
5.4. Objetivos especiales incorporados . . . 13
5.5. Objetivos múltiples en una regla . . . 13
5.6. Reglas múltiples para un objetivo . . . 14

6. Escritura de las órdenes en la regla **14**
6.1. Visión de las órdenes . . . 14
6.2. Ejecución de las órdenes . . . 15
6.3. Errores en las órdenes . . . 15
6.4. Interrupción de make . . . 16

7. Macros o variables **16**
7.1. Macros predefinidas . . . 17

8. Reglas implícitas **18**
8.1. Uso de las reglas implícitas . . . 18
8.2. Escritura de reglas implícitas (método de sufijos) . . . 19
8.3. Cancelación de una regla implícita . . . 19
8.4. Definición de una regla predeterminada . . . 19

9. Cómo ejecutar make **20**
9.1. Parámetros para especificar el *makefile* . . . 20
9.2. Parámetros para especificar el objetivo . . . 20
9.3. Parámetros para no ejecutar las órdenes . . . 20
9.4. Parámetros para cambiar macros . . . 21
9.5. Resumen de parámetros y opciones para make . . . 21

10. Un ejemplo real **22**
10.1. Comentarios iniciales 22
10.2. Macros configurables 23
10.3. Macros no configurables 23
10.4. Reglas principales de construcción e instalación 24
10.5. Reglas intermedias o secundarias 25
10.6. Reglas adicionales 25
10.7. Reglas de borrado 26
10.8. Regla de distribución 27
10.9. Utilización 27

Lista de *makefiles*

1. *Makefile* para un editor — primera versión 4
2. *Makefile* para un editor — segunda versión 7
3. *Makefile* para un editor — tercera versión 7
4. *Makefile* simplificado para el programa que saluda 18

1. Introducción a make

El propósito de la utilidad make es determinar automáticamente qué piezas o módulos de un programa necesitan ser recompilados, y ejecutar las órdenes apropiadas para esta tarea.

Los ejemplos que se verán mostrarán programas en C++, pero make puede usarse con cualquier otro lenguaje de programación cuyo compilador pueda ejecutarse mediante una orden. De hecho, make no está limitado a compilar programas. Puede usarse para describir cualquier tarea donde algunos ficheros deban ser actualizados de alguna forma automáticamente a partir de otros cada vez que estos últimos cambien. Un ejemplo de esto sería la confección de documentos usando algún procesador de textos como `groff` o LaTeX.

Antes de usar la orden `make` hay que escribir con el editor de textos favorito un fichero llamado «el *makefile*» que describe las relaciones entre los ficheros que componen el programa, así como las órdenes necesarias para actualizar cada uno de ellos. Tratándose de un programa, normalmente el fichero ejecutable se actualiza a partir de módulos objeto, que a su vez se construyen a partir de la compilación de código fuente.

Una vez que existe el fichero *makefile*, cada vez que se modifique algún fuente, o varios, la simple orden

```
make
```

bastará normalmente para la recompilación del programa. Pero lo bueno de make es que no lo compila todo de nuevo, sino que basándose en los datos del *makefile* y las fechas de última modificación que el sistema operativo guarda para cada fichero, decide cuáles son los que deben ser actualizados, y para cada uno de ellos ejecuta las órdenes apropiadas según se especifica en el *makefile*.

Por otra parte, mediante el uso de parámetros y opciones que se le pasan a make, se puede controlar qué ficheros recompilar o cómo. Esto se verá más adelante en §9.

1.1. Cómo leer este manual

Las líneas marcadas como ésta expresan conceptos fundamentales que debe memorizar. ✔

Las secciones marcadas con este signo expresan características no tan fundamentales, que pueden dejarse para una lectura posterior, cuando ya se conozca mejor el programa. ✘

2. Un ejemplo sencillo de *makefile*

Supongamos que hemos hecho un editor de textos que consiste en 8 ficheros fuente en C++ y 3 ficheros de cabecera propios. Necesitamos un fichero *makefile* que le diga a make cómo compilar y enlazar para obtener el editor. Supondremos en nuestro ejemplo que todos los fuentes C++ incluyen la cabecera `"defs.h"` pero que sólo aquéllos donde se definen las funciones de edición incluyen `çommands.h"` y que solamente los que manejan las funciones de bajo nivel, como por ejemplo los cambios en el *búfer* del editor, incluyen `"buffer.h"`.

El programa make obtiene todos los datos del fichero *makefile*, que habrá que escribir con un editor. ✔

Para reconstruir el editor debe recompilarse cada fichero fuente C++ que haya sido modificado. Si un fichero de cabecera ha cambiado, para asegurarnos, debe recompilarse cada fichero fuente que incluya esa cabecera. Finalmente, si cualquier fuente ha sido recompilado, deben enlazarse todos los módulos objeto, ya recién hechos o ya provenientes de anteriores compilaciones, para producir el nuevo ejecutable.

El *makefile* 1 presenta la primera versión de un fichero *makefile* que nos permitirá compilar todos los módulos y enlazarlos cuando llegue el momento.

Las líneas largas las dividimos en dos mediante el signo \ seguido inmediatamente de un salto de línea; esto las hace más fáciles de leer.

Cada fichero que es generado por un programa (esto es, todos los ficheros excepto los fuentes y cabeceras) es el *objetivo* ("target" en inglés) de una *regla*. En este ejemplo los objetivos son los módulos objeto, como `main.o`, `kbd.o`, etc., y el ejecutable, `edit`.

Cada objetivo debe aparecer al principio de una línea seguido por el signo de dos puntos.

Tras los dos puntos vienen las *dependencias* del objetivo; esto es, todos los ficheros que se usan para actualizarlo.

Un objetivo necesita recompilación o enlace (*actualización*) si y sólo si cualquiera de sus dependencias cambia. Además, cualquier dependencia debe ser actualizada antes si lo necesita. En nuestro ejemplo, `edit` depende de cada uno de los ocho módulos objeto; el fichero objeto `main.o` depende del fichero fuente `main.cpp` y del fichero de cabecera `defs.h`; si cambia uno de éstos, `edit` debe ser enlazado de nuevo, pero antes, `main.o` debe ser recompilado.

Tras el objetivo y los ':' vienen las dependencias, separadas por blancos. ✔

Por omisión, make empieza con la primera regla que encuentra en el *makefile* (el *default goal* en inglés), sin contar aquellas cuyo objetivo empieza con un

```
edit: main.o kbd.o commands.o display.o \
      insert.o search.o files.o utils.o
        c++ -o edit  main.o kbd.o commands.o display.o \
                     insert.o search.o files.o utils.o

main.o: main.cpp defs.h
        c++ -c main.cpp

kbd.o: kbd.cpp defs.h commands.h
        c++ -c kbd.cpp

commands.o: commands.cpp defs.h commands.h
        c++ -c commands.cpp

display.o: display.cpp defs.h buffer.h
        c++ -c display.cpp

insert.o: insert.cpp defs.h buffer.h
        c++ -c insert.cpp

search.o: search.cpp defs.h buffer.h
        c++ -c search.cpp

files.o: files.cpp defs.h buffer.h commands.h
        c++ -c files.cpp

utils.o: utils.cpp defs.h
        c++ -c utils.cpp
```

Makefile 1: *Makefile* para un editor — primera versión

punto. Por lo tanto pondremos en primer lugar la regla para el ejecutable, `edit`. Las otras se procesan porque aparecen como dependencias de esta.

> *Make* hace la primera regla que encuentra en el *makefile*, si no se indica otra cosa. ✔

Tras cada línea que contiene la regla, es decir, el objetivo y sus dependencias, vienen una o más líneas de órdenes para el intérprete de órdenes del sistema operativo (*shell*), que dicen cómo actualizar el fichero objetivo. Estas líneas empiezan con un tabulador, lo cual indica a make que son líneas de órdenes que debe mandar a ejecutar. Observe que make no sabe nada acerca de cómo funcionan las órdenes; es responsabilidad del programador el que éstas sean correctas; todo lo que make hace es mandarlas a ejecución cada vez que el objetivo necesite ser actualizado.

> Tras la línea de '*objetivo* : *dependencias*' vienen las líneas de órdenes precedidas de un TAB. ✔

2.1. Cómo make procesa este *makefile*

Tras leer el *makefile*, make empieza el trabajo procesando la primera regla; esto es, la que se refiere a enlazar `edit`; pero antes, debe procesar las reglas para las dependencias de `edit`: todos los módulos objeto. Cada uno de éstos tiene su propia regla, que dice que debe actualizar el fichero '`.o`' mediante compilación de su fichero fuente. La recompilación debe efectuarse si el fichero fuente o cualquiera de los de cabecera nombrados como dependencias es más reciente que el fichero objeto, o si éste no existe.

Aun antes de recompilar cualquier fichero objeto que lo necesite, make considera cómo actualizar sus dependencias: el fichero fuente y los ficheros de cabecera. Pero este *makefile* no especifica cómo hacerlo, pues dichos ficheros no son el objetivo de ninguna regla, así que make no hace nada a este respecto. Estos ficheros los crea o modifica el programador no automáticamente sino a voluntad, con un editor de textos como `vi` o `emacs`; pero si estos ficheros fuente hubieran sido generados automáticamente por algún programa como `bison` o `yacc`, éste sería el momento en el que se actualizarían mediante su propia regla.

Después de todas las compilaciones precisas, make debe decidir si enlazar `edit`. Esto debe hacerse si este fichero no existe o si alguno de los módulos objeto es más reciente; evidentemente, si un módulo objeto acaba de ser recompilado es más reciente que `edit`, luego debe enlazarse.

> *Make* actualiza un objetivo si no existe o si alguna de sus dependencias es más reciente. ✔

Así por ejemplo, si nosotros modificamos el fichero `insert.cpp` y a continuación ejecutamos `make`, make compilará dicho fichero para obtener `insert.o`, y luego enlazará todos los módulos para formar el ejecutable `edit`.

Si cambiamos el fichero `commands.h` y ejecutamos `make`, se recompilarán `kbd.cpp`, `commands.cpp` y `files.cpp` para formarse respectivamente los ficheros `kbd.o`, `commands.o` y `files.o`, y luego se enlazaría `edit`.

3. Las macros simplifican el *makefile*

En nuestro ejemplo, teníamos que listar todos los ficheros objeto dos veces en la regla para `edit`, como se recuerda aquí:

```
edit: main.o kbd.o commands.o display.o \
      insert.o search.o files.o utils.o
        c++ -o edit  main.o kbd.o commands.o display.o \
                     insert.o search.o files.o utils.o
```

Tales duplicaciones son propensas a error: si se añadiera un nuevo módulo objeto al sistema, podríamos agregarlo a una lista y olvidarnos de la otra. Pero podemos eliminar este riesgo, simplificar el *makefile* y ahorrarnos el escribir más de la cuenta usando *macros* o *variables*. Éstas permiten definir una cadena de caracteres textual con un nombre, y sustituir esta cadena por el nombre las veces que sean necesarias.

De hecho, es práctica habitual para cada *makefile* el que haya una macro llamada `OBJECTS` u `OBJS` que sea una lista de todos los módulos objeto. Por lo tanto, para definir una macro, pondríamos al principio del *makefile* una línea como

```
OBJECTS = main.o kbd.o commands.o display.o \
          insert.o search.o files.o utils.o
```

Para definir una *variable* o *macro* se pone su nombre, el signo igual y el texto al que sustituirá. ✔

Luego, en cada sitio donde debería aparecer la lista de módulos objeto, podríamos sustituir tal lista por el valor de la macro escribiendo `${OBJECTS}` u `$(OBJECTS)`. Por lo tanto, nuestra regla para `edit` quedaría así:

```
edit: $(OBJECTS)
        c++ -o edit  $(OBJECTS)
```

Para obtener el valor de una macro se pone el signo dólar y su nombre entre paréntesis o llaves. ✔

Se pueden usar las llaves o los paréntesis indistintamente. Además, si la variable o macro sólo tiene una letra, no hacen falta. Por ejemplo, serían equivalentes `$A`, `$(A)` y `${A}`.

3.1. Macros predefinidas

Además de las que podamos escribir, make tiene unas macros predefinidas, que no hace falta que sean definidas por nosotros, pero que podemos redefinir si queremos. Entre ellas están `CXX`, que representa al compilador de C++ y vale `c++`[1], si no la cambiamos, y `CXXFLAGS`, que representa las opciones del compilador de C++ y vale la cadena vacía. Estas macros son usadas por make internamente, como se verá en la sección siguiente.

3.2. Dejemos que make deduzca las órdenes

La mayoría de las veces no es necesario describir las órdenes para compilar los ficheros fuente individuales en C++ porque make puede figurarse cuáles son: tiene incorporada una regla implícita mediante la cual sabe cómo actualizar un fichero '`.o`' a partir de otro '`.cpp`' con el mismo nombre usando una orden de compilación adecuada. Por ejemplo, para obtener el módulo objeto `main.o` compilaría `main.cpp` mediante una orden como la siguiente.

```
${CXX} -c ${CXXFLAGS} main.cpp
```

Por lo tanto podemos omitir del *makefile* las órdenes de compilación en las reglas de los ficheros objeto.

Make tiene una regla implícita que le dice cómo compilar un fichero fuente en C++ para obtener un módulo objeto. ✔

Del mismo modo, los ficheros fuente en C++ usados de esta forma se añaden automáticamente a la lista de dependencias, por lo que si omitimos las órdenes, podemos también quitarlos de la lista.

El *makefile* 2 ha sido reescrito con todo lo que sabemos hasta ahora. Así es como se escribiría en la práctica real (más o menos).

[1] En realidad, su valor concreto depende del sistema y de la versión de make.

```
# Esto es un comentario. make no lo tiene en cuenta, así
# como las líneas en blanco. Un comentario es lo que va
# desde el signo # hasta el final de la línea.

OBJECTS = main.o kbd.o commands.o display.o \
          insert.o search.o files.o utils.o

edit: $(OBJECTS)
        ${CXX} -o edit  $(OBJECTS)

main.o: defs.h
kbd.o: defs.h commands.h
commands.o: defs.h commands.h
display.o: defs.h buffer.h
insert.o: defs.h buffer.h
search.o: defs.h buffer.h
files.o: defs.h buffer.h commands.h
utils.o: defs.h
```

Makefile 2: *Makefile* para un editor — segunda versión

3.3. Otro estilo de *makefile*

Como en el *makefile* anterior las reglas para los módulos objeto especifican solamente dependencias, no órdenes, podemos agruparlas por dependencias en vez de por objetivos. El *makefile* alternativo que resulta es el 3.

```
OBJECTS = main.o kbd.o commands.o display.o \
          insert.o search.o files.o utils.o
CXX = g++           # Emplear el compilador de GNU
CXXFLAGS = -g -Wall # -g para poder depurar. -Wall: avisos extra

edit: $(OBJECTS)
        ${CXX} -o edit  $(OBJECTS)

$(OBJECTS): defs.h
kbd.o commands.o files.o: command.h
display.o insert.o search.o files.o: buffer.h
```

Makefile 3: *Makefile* para un editor — tercera versión

Aquí, `defs.h` se da como una dependencia de todos los módulos objeto, mientras que las cabeceras `commands.h` y `buffer.h` son dependencias de los módulos objeto específicos listados para cada uno.

Si esta forma es mejor que la anterior es una cuestión de gusto personal. Ésta es más compacta sin duda, pero mucha gente prefiere la otra porque encuentra más claro el tener toda la información sobre cada objetivo en un solo sitio. Observe que aquí los módulos objeto están repetidos como objetivos.

3.4. Reglas para limpiar el directorio

Lo normal al trabajar en un programa medianamente complicado es crear un directorio y poner todos los ficheros en él: los fuentes, las cabeceras, la documentación quizá, y el *makefile*. Conforme se prueba el programa deben mantenerse los módulos objeto que se forman al compilar los fuentes; esto es lo que hace que al modificar algún fichero fuente la recompilación sea mínima, gracias a make: si cada vez que compiláramos borrásemos los módulos objeto, make tendría que recompilarlo todo a la siguiente vez y no valdría para casi nada. Sin embargo, una vez que hemos comprobado el programa y estamos satisfechos con él, una vez que vemos que no tiene errores y que lo instalamos o copiamos a otro sitio y ya nos disponemos a trabajar en otro proyecto en otro directorio, debemos borrar los módulos objeto y los inservibles para ahorrar espacio de disco. Indudablemente lo podemos hacer «a mano», pero make también nos puede ayudar en esto.

En efecto, compilar un programa no es la única cosa que make puede hacer; es decir, no es la única cosa para la que podemos escribir reglas en un *makefile*. Hay en él normalmente una serie de reglas para hacer una determinada serie de tareas rutinarias como imprimir los fuentes, generar o imprimir la documentación, embellecer el código de los programas fuente, etc. Y por supuesto limpiar el directorio, que es de lo que se ha hablado en el párrafo anterior. Veamos la regla para ello; es costumbre llamarla `clean` («limpiar», en inglés):

```
clean:
     rm -f edit $(OBJECTS) *~ \#* core
```

Esta regla se añadiría al final del *makefile*: evidentemente no deseamos que sea la regla predeterminada (la primera), que debe ser la de `edit`, para recompilar el editor.

Esta regla no tiene dependencias, como se ve. Por tanto no depende de `edit` y no se ejecutará nunca si damos la orden `make` sin argumentos. Para que esta regla se ejecute debemos dar la orden:

```
make clean
```

Algunas notas más:

- ✶ El nombre '`clean`' es arbitrario; podría ser '`limpiar`' por ejemplo. No obstante, es costumbre llamar así a la regla que limpia el directorio; y uno espera encontrarla cuando ve que hay un *makefile* por ahí.
- ✶ Ya debe saberse que antes de la orden, `rm` en este caso, debe haber un carácter TAB. La opción `-f` indica a `rm` que no haga preguntas ni proteste en ningún caso, para que el proceso sea lo más automático posible.
- ✶ Los ficheros a borrar serán evidentemente los módulos objeto (he aquí otro uso interesante de la variable `OBJECTS`), pero además se puede borrar el ejecutable `edit` —supuesto que se ha salvado, copiado o instalado en otro sitio—, el fichero `core`, que es una imagen de la memoria en el caso de que hubiera ocurrido un error en tiempo de ejecución, y los ficheros de respaldo del editor `emacs`, supuesto que se ha usado este editor. Cada uno pondrá lo que más le convenga. En general, todos los ficheros que pueda haber que no sirvan o que puedan volver a ser generados. ¡¡Pero nunca los fuentes!!

✶ Existen dos macros incorporadas en make, llamadas `RM` y `RMFLAGS`, que contienen respectivamente la orden de borrado y sus opciones apropiadas para cada sistema.

4. Cómo escribir un *makefile*

La información que necesita make para recompilar un sistema la obtiene mediante la lectura y análisis de una especie de base de datos que se encuentra en un fichero llamado el *makefile*.

4.1. Contenido de un *makefile*

Un fichero *makefile* contiene básicamente tres clases de cosas: reglas, definiciones de macros y comentarios. Algunas versiones de make soportan además una cuarta clase: directivas.

reglas Una regla dice cuándo y cómo reconstruir uno o más ficheros, llamados los *objetivos* de la regla. En ella se listan, tras los objetivos y separados por el signo dos puntos, los otros ficheros de los cuales depende el objetivo, llamados las *dependencias*; y en otras líneas, después de un carácter TAB (tabulador), se puede dar una serie de *órdenes* para efectuar la tarea. A veces los objetivos y las dependencias no son ficheros reales.

macros Una definición de variable o macro es una línea que especifica un valor textual para una variable o macro; este valor podrá ser sustituido más tarde las veces que haga falta. En el ejemplo de la sección anterior se definía una variable llamada `OBJECTS` cuyo valor textual era la lista de los módulos objeto.

comentarios Un comentario es aquel texto que va desde el signo '`#`' hasta el final de la línea. Como es habitual, los comentarios son muy útiles para poner *copyrights*, nombre de autor, lista de modificaciones, versión, etc., y para aclarar cosas o dividir secciones. En las líneas de órdenes, es el *shell* o intérprete de órdenes el que decide qué es un comentario, pero suele coincidir con este signo. Por supuesto, make no los tiene en cuenta, así como las líneas en blanco.

directivas Una directiva o directriz es una orden para make que le insta a hacer algo especial mientras está leyendo el *makefile*, como por ejemplo, incluir otro en éste. Las directivas dependen mucho de la versión de make que estemos usando, y no vamos a verlas aquí.

4.2. Qué nombre dar al *makefile*

En UNIX, por omisión, cuando make busca el *makefile*, busca primero un fichero con el nombre `makefile`; si no existe, prueba con `Makefile`.

Normalmente es recomendable llamar al fichero *makefile* con el nombre `Makefile`, para que en un listado del directorio ordenado por nombres, como

por ejemplo al dar la orden `ls`, aparezca al principio de él, quizá junto a otros ficheros importantes como `LEAME`[2].

Si make no encuentra un fichero con ninguno de esos dos nombres, en algunas versiones no usa ningún *makefile*, con lo cual dará un error; en otras, emplea sus reglas implícitas como si se le hubiera proporcionado un *makefile* vacío.

Si, por algún motivo, se quiere usar otro nombre para su *makefile*, por ejemplo `edit.mk`, se puede hacer, pero cuando se ejecute `make` habrá que especificar que debe leer de este fichero mediante la opción '`-f`', a la cual debe seguir el nombre del fichero *makefile* (vea §9.1 en la página 20). En este caso la orden sería:

```
make -f edit.mk
```

5. Cómo escribir las reglas

Una regla aparece en el *makefile* y describe cuándo y cómo reconstruir ciertos ficheros, llamados objetivos de la regla (normalmente uno por cada regla). A continuación lista las dependencias del objetivo, y las órdenes necesarias para crearlo o actualizarlo.

El orden de las reglas no tiene importancia salvo para determinar la predeterminada, es decir, el objetivo que make construirá si no se le especifica otro en la línea de órdenes. Esta regla predeterminada es la primera que aparece en el *makefile* si no empieza por un punto. Por lo tanto, se escribirá el *makefile* de forma que la primera regla que se ponga sea la encargada de compilar el programa entero, o todos los programas que se describan.

Por ejemplo, supongamos que se va a aprovechar parte del editor del ejemplo anterior para hacer un programa que nos permita ver el contenido de un fichero; sea `view` el nombre de tal programa. Para que al dar la orden `make` sin parámetros se recompilen los dos programas y no sólo el editor, se pondrá como primera regla una cuyo objetivo dependa de los dos. Es costumbre llamar a tal regla `all` («todo», en inglés). Así, la primera regla sería simplemente:

```
all: edit view
```

No tiene órdenes asociadas. Como `all` depende de `edit` y de `view`, si alguno de los dos, o ambos, necesitan reconstruirse, así se hará.

5.1. Sintaxis de una regla

En general, una regla aparece como:

objetivo(s) : *dependencia(s)*
TAB *orden*
TAB *orden*
. . .

o como:

[2] Esto es cierto si la orden antedicha clasifica por el código de caracteres en uso; si lo hace según el idioma pondrá las mayúsculas en el mismo orden que las minúsculas.

objetivo(s) : *dependencia(s)* ; *orden*
TAB *orden*
TAB *orden*
. . .

objetivos Son nombres de ficheros o palabras, separadas por espacios en blanco. Se pueden usar comodines. Normalmente sólo se pone un objetivo, pero a veces conviene poner más de uno.

dependencias Son nombres de ficheros u otros objetivos, de los que dependen, separados por espacios en blanco. Se pueden usar comodines.

órdenes Son las del *shell* de UNIX (normalmente el *shell* de Bourne, `sh`), o las del intérprete de órdenes del sistema operativo donde estemos, mediante las cuales se reconstruirá el objetivo. Las líneas de órdenes deben empezar obligatoriamente con el carácter TAB (*tabulador*), pero la primera orden puede ponerse también en la misma línea del objetivo, tras las dependencias, separado de ellas por un carácter punto y coma ('`;`').

5.2. Objetivos falsos

¿Se acuerda de este signo? ¿Está seguro de que quiere leer ahora este capítulo?, ¿no preferiría dejarlo para una segunda lectura, suponiendo que lo vaya a leer otra vez?

Un *objetivo falso* ("phony target" en inglés) es aquél que no es realmente un nombre de fichero, sino un nombre descriptivo de alguna acción, de algunas órdenes que se ejecutarán al requerirse explícitamente. Ya hemos visto un ejemplo. Vamos a repetirlo algo más simplificado:

```
clean: ; rm -f *.o core
```

Observe que faltan las dependencias y que hemos puesto la orden en la misma línea, separada por un punto y coma, para variar.

Como la orden de borrado claramente no crea ningún fichero que se llame `clean`, probablemente no existirá tal fichero, por lo que `rm` se ejecutará cada vez que se dé la orden '`make clean`'.

Ahora bien, este objetivo dejará de funcionar si alguien alguna vez crea un fichero que se llame `clean` en el directorio. Porque como no hay dependencias en esta regla, tal fichero sería considerado siempre como puesto al día y entonces no necesitaría reconstruirse, luego no se ejecutarían las órdenes asociadas y `make` respondería con algo como

```
make: 'clean' is up to date
```

Para evitar esto (ciertamente remoto en este caso, por otra parte, sobre todo si el idioma del usuario no es el inglés) se puede declarar explícitamente que el objetivo es falso (*phony*) usando el objetivo especial incorporado en `make` `.PHONY` (observe el punto inicial y que está en mayúsculas), así:

```
.PHONY: clean
```

Una vez hecho esto, '`make clean`' ejecutará la orden de borrado independientemente de que exista un fichero que se llame `clean`.

Un objetivo falso no debe ser dependencia de un objetivo que sea realmente un fichero, pero sí puede tener dependencias. Ya se ha visto anteriormente otro objetivo falso, que era el llamado `all`, el cual dependía de dos objetivos. Ahora, para hacer las cosas bien, se pondría:

```
all: edit view
.PHONY: all
```

El siguiente ejemplo de borrado selectivo ilustra el hecho de que cuando un objetivo falso es una dependencia de otro, sirve como una subrutina de él. Si se da la orden '`make cleanall`' se borrarían los módulos objeto, los ficheros de diferencias, y el fichero ejecutable, mientras que '`make cleanobj`' borraría sólo los módulos objeto.

```
.PHONY: cleanall cleanobj cleandiff

cleanall: cleanobj cleandiff
        rm -f edit

cleanobj:
        rm -f *.o

cleandiff:
        rm -f *.diff
```

5.3. Ficheros vacíos como objetivos

El *objetivo vacío* es una variante del falso; se usa para dar una orden para una acción que se realiza de tarde en tarde explícitamente. En oposición al objetivo falso, el objetivo vacío puede existir como fichero, pero sus contenidos no importan, con lo que usualmente es un fichero vacío.

El propósito de un objetivo vacío es registrar eventos: recordar, mediante su fecha de última modificación, cuándo se ejecutaron por última vez las órdenes de su regla. Para ello se pone como una de las órdenes el programa `touch`, que lo que hace es modificar esa fecha simplemente, creando de paso el fichero si no existía.

El objetivo vacío debe tener algunas dependencias. Cuando se pide reconstruirlo, las órdenes se ejecutan si alguna de las dependencias es más reciente que el objetivo; o sea, si una dependencia al menos ha sido modificada desde la última vez que se reconstruyó el objetivo. El siguiente ejemplo ilustrará todo esto. Es un objetivo para imprimir en papel los ficheros fuente que se han modificado desde la última impresión.

```
print: foo.cpp bar.cpp
        lpr -p $?
        touch print
```

Con esta regla, '`make print`' ejecutará la orden de imprimir `lpr -p` si algún fichero fuente de los especificados como dependencia ha cambiado desde el último

'`make print`'. Para ello se usa también una variable incorporada o especial de make, '`?`', que se sustituye por los ficheros de las dependencias que han cambiado.

Los ficheros vacíos correspondientes a estos objetivos no deberían borrarse, salvo por supuesto cuando ya no se va a trabajar más en el proyecto.

5.4. Objetivos especiales incorporados

Ya hemos visto que `.PHONY`, cuando se usaba como nombre de objetivo, tenía un significado especial. Vamos a ver ahora una lista de estos nombres incorporados en make:

`.PHONY` Las dependencias del objetivo especial `.PHONY` se consideran objetivos falsos. Cuando llegue la hora de considerar tales objetivos, make ejecutará sus órdenes independientemente de que exista o no un fichero con esos nombres, o de cuáles sean sus fechas de última modificación.

`.SUFFIXES` La dependencia de este objetivo especial será la lista de sufijos que se va a usar al comprobarla (vea §8.2).

`.PRECIOUS` Los objetivos que son a su vez dependencias de este objetivo especial reciben este *precioso* tratamiento especial: si make es interrumpido durante la ejecución de sus órdenes, no se borran.

`.IGNORE` Simplemente con mencionar este objetivo, sin dependencias ni órdenes, make pasará por alto cualquier error que pueda producirse en la ejecución de órdenes; es decir, si hay algún error, seguirá adelante como pueda, en lugar de pararse. Equivale a la opción `-k` (vea §9.5).

`.SILENT` Simplemente con mencionar este objetivo, sin dependencias ni órdenes, las que se ejecuten no se mostrarán antes en la salida estándar, como es lo normal. Equivale a la opción `-s` (vea §9.5).

Los dos últimos objetivos mencionados se mantienen por compatibilidad con versiones anteriores; hay otras formas mejores de conseguir lo mismo.

También pueden considerarse objetivos especiales los formados por concatenación de sufijos, que son una forma antigua de definición de regla implícita (vea más adelante §8.2).

5.5. Objetivos múltiples en una regla

Una regla con múltiples objetivos es equivalente a múltiples reglas con un objetivo, y todo lo demás idéntico; las mismas órdenes se aplicarán a todos los objetivos, pero sus efectos pueden variar porque se puede sustituir el nombre del objetivo en la línea de órdenes empleando la variable especial '`@`'. Hay dos casos donde es de utilidad usar múltiples objetivos:

① Queremos sólo dependencias, no órdenes. Ya hemos visto un caso de esto, en el *makefile* 3:

```
kbd.o commands.o files.o: command.h
```

En este ejemplo, se proporciona una dependencia adicional a cada uno de los tres módulos objeto.

② Órdenes similares sirven para todos los objetivos. Ejemplo: ✘

```
# Escoger para GDEV uno de:
# -Tlatin1 para terminal/impresora alfanumérica
# -Tps     para impresora/pantalla PostScript
GDEV = -Tps

edit.ps edit.txt: edit.ms
        groff -ms $(GDEV) edit.ms > $@
```

En este ejemplo, `groff` es un formateador de textos que produce salida para una terminal alfanumérica o impresora de matriz de puntos usando el código internacional de caracteres ISO-Latin1 cuando se usa la opción `-Tlatin1` o bien produce código PostScript® para una impresora láser o pantalla gráfica con visor PostScript cuando se usa la opción `-Tps`. Usamos una variable para distinguir las opciones.

En §9 se explica cómo al llamar a `make` se pueden dar valores a macros o decirle qué objetivo queremos hacer. Si damos la orden

```
make GDEV=-Tlatin1 edit.txt
```

se ejecutará lo siguiente:

```
groff -ms -Tlatin1 edit.ms > edit.txt
```

5.6. Reglas múltiples para un objetivo ✘

Un fichero puede ser el objetivo de múltiples reglas. Todas las dependencias mencionadas en todas las reglas se mezclan en una sola lista de dependencias para el objetivo. Si éste es más antiguo que cualquier dependencia de cualquiera de sus reglas, las órdenes (debe haber una sola serie) se ejecutan.

6. Escritura de las órdenes en la regla

Las órdenes de una regla consisten en líneas de órdenes del intérprete de órdenes (*caparazón* o *shell*, en UNIX) que se ejecutan una a una. Cada línea de órdenes debe empezar por un carácter TAB, salvo que la primera puede ir tras las dependencias, separada de éstas por un punto y coma. Las líneas en blanco no se tienen en cuenta. Aunque se puede utilizar en UNIX el *shell* que se prefiera, make emplea siempre el *shell* de Bourne, `sh`, a menos que se especifique otra cosa en el *makefile*. En cuanto a los comentarios y su sintaxis, en estas líneas no se puede usar el comentario de make, sino el del *shell*. Da la casualidad de que para `sh` el signo '`#`' indica también un comentario.

6.1. Visión de las órdenes ✘

Normalmente make muestra cada línea de órdenes antes de su ejecución, a menos que hayamos puesto el objetivo `.SILENT` (§5.4) o hayamos dado la opción `-s` (§ 9.5), pero se puede hacer que selectivamente las órdenes que se quieran no se muestren si el primer carácter es el signo '`@`', que no se le pasa al *shell*.

Normalmente esto se utiliza para órdenes cuyo único efecto sea mostrar algún mensaje informativo, para que no aparezca repetido:

```
@echo Compilando ...
```

Cuando damos a make la opción `-n` su único efecto es mostrar lo que haría, sin ejecutar nada realmente, con lo cual el signo `@` no tiene en este caso efecto ninguno, ni el objetivo `.SILENT`; si le damos en cambio la opción `-s` el efecto es el mismo que si hubiéramos usado el objetivo `.SILENT` o si todas las órdenes hubieran estado precedidas por `@`: no se muestra ninguna orden de las que se ejecutan. Todo esto se verá en §9.5.

6.2. Ejecución de las órdenes

Para cada nueva línea de órdenes, make crea un subproceso (*subshell*). Esto implica que las que cambien aspectos relativos al proceso, como por ejemplo `cd` (cambiar directorio), no afectarán a las siguientes. Para evitar esto habría que poner la siguiente orden en la misma línea, tras un punto y coma; entonces dicha línea será considerada como una sola, que make pasa al *shell*, el cual la divide en varias, pues para éste, el signo punto y coma es un terminador de orden. Ejemplo:

```
prog.1: doc/prog.man
        cd doc ; nroff -man prog.man > ../$@
```

Otra forma de escribirlo:

```
prog.1: doc/prog.man
        cd doc ; \
        nroff -man prog.man > ../$@
```

Puesto que la barra invertida \ seguida de nueva-línea indica que la siguiente es una continuación de ésta. Obsérvese que la variable especial `@` se sustituye por el objetivo; en este caso por `prog.1` (una página del Manual de UNIX).

El programa usado como caparazón se toma de la variable especial de make `SHELL`; si no se define se toma `/bin/sh`, el *shell* de Bourne. No se emplea en UNIX la variable de ambiente `SHELL` que el usuario tenga definida como su *shell* favorito, aunque sí en DOS/Windows. La variable `SHELL` pues deberá definirse en UNIX en el *makefile* o al llamar a `make`.

Note que todo esto depende mucho del sistema operativo y del intérprete de órdenes que emplee. Normalmente el *shell* de Bourne se basta y sobra para las simples órdenes que tiene que hacer make; además es el más rápido, puesto que es el más simple. No lo cambie a no ser que tenga fundadas razones para ello. No tiene nada que ver el que se use otro en interactivo o en los guiones (*shell-scripts*). (Además, para tranquilizarle, los *shells* de Korn, *Bash* y el Z son compatibles con el de Bourne casi totalmente —al revés no es cierto, evidentemente—).

6.3. Errores en las órdenes

Cuando una orden finaliza su ejecución, make mira su *status* de terminación. Si la orden acabó con éxito, se ejecuta la siguiente de la lista en un *subshell* o, si

era la última, se acaba con la regla que se estuviera haciendo. Pero si hubo un error y no acabó bien, lo normal es que make termine con la regla y con todas las demás que hubiera.

A veces el que una orden falle no indica un problema. Por ejemplo, se puede dar una orden `mkdir` para crear un directorio por si no existiera. Si ya existía, `mkdir` fallará al no poder crearlo, pero normalmente no se desea que esto haga que el proceso finalice.

Para que se soslayen los errores en una orden, póngase como primer carácter de ella el signo guion '`-`', como en el ejemplo:

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

Cuando se usa el objetivo especial `.IGNORE`, los errores se soslayan siempre, como si todas las órdenes hubieran empezado con el signo guion; lo mismo se obtiene con la opción `-i`. En cualquier caso make trata estos errores como éxito, salvo que muestra un mensaje de aviso y dice que el error no ha sido tenido en cuenta.

Como ya se ha dicho, si hay un error que no se debe soslayar, como uno de compilación, make deja de procesar la regla, y todas las demás, y se detiene devolviendo el control al sistema operativo. Pero si se llama con la opción `-k` sólo deja de procesar la regla donde se produjo el error, y sigue trabajando con otras que no dependan de ésta; o sea, sigue hasta donde puede. Vea §5.4 y §9.5 para más información.

6.4. Interrupción de make

Si make recibe una señal de interrupción, u otra fatal, mientras está ejecutando una orden, puede borrar el fichero objetivo que se estuviera actualizando. Esto se hace si la fecha de última modificación del fichero objetivo ha cambiado desde que make la comprobó por primera vez. El propósito de esto es asegurarse de que make empiece desde cero la próxima vez que se ejecute, para que no considere como actualizado un fichero que quizá quedó a medio hacer cuando se recibió la señal de interrupción, normalmente un CTRL/C.

Este comportamiento puede evitarse si tal fichero (*precioso* para nosotros) se hace depender del objetivo especial `.PRECIOUS`, ya visto en §5.4.

7. Macros o variables

Una variable o macro es un nombre definido para make que representa una cadena de texto llamada el *valor* de la macro. Este valor puede sustituirse cuando se quiera en cualquier parte de un *makefile*.

Una macro puede representar una lista de nombres de ficheros, opciones de un compilador, programas a ejecutar, directorios donde buscar ficheros fuente o donde instalar los ejecutables, o cualquier otra cosa imaginable.

Un nombre de macro puede ser cualquier secuencia de caracteres salvo '`#`', '`:`', '`=`' o espacios en los extremos. Pero no deben usarse más que letras, dígitos y el signo subrayado '`_`'. Es tradicional usar letras mayúsculas.

Para definir una macro se escribe el nombre seguido del signo igual (puede dejar espacios alrededor de él) y la cadena de caracteres textual que conforma su valor.

Para sustituir el valor de la macro en cualquier punto, se escribe el signo dólar `$` seguido del nombre de la variable entre paréntesis o llaves; puede omitirlos si el nombre de la variable sólo tiene un carácter, pero no es recomendable. Para conseguir el signo dólar como nombre de macro deberá escribir pues dos seguidos.

Todas las *variables de ambiente* o entorno que uno tenga definidas en su proceso, salvo `SHELL` en UNIX, son heredadas por make como si se hubieran definido en el *makefile*.

7.1. Macros predefinidas

El programa make usa internamente una serie de macros para sus reglas predefinidas y órdenes. Por ejemplo, en el caso de poner en un *makefile*

```
files.o: files.h
```

make sabe que `files.o` depende de `files.cpp` y de `files.h`, y sabe cómo construir el objeto a partir del fuente; así:

```
$(CXX) -c $(CXXFLAGS) $<
```

Aquí vemos una serie de macros predefinidas y especiales. `CXX` y `CXXFLAGS` representan al compilador de C++ y las opciones que se le han de pasar respectivamente.

> Make tiene unas macros predefinidas. Entre ellas, `CXX` representa al compilador de C++, y `CXXFLAGS` sus opciones. ✔

Si se redefinen estas macros en el *makefile* o en la línea de órdenes o en el ambiente, se usarán esas definiciones; si no, make utilizará sus propias definiciones incorporadas, que suelen ser:

```
CXX      = c++   (o g++ en el caso de GNU Make)
CXXFLAGS =       (ninguna opción)
```

Por otra parte, `<` es una macro especial incorporada que se sustituye por la dependencia, así como `@` se sustituye por el objetivo.

Por lo tanto, se podría modificar el compilador a emplear o sus opciones definiendo las macros `CXX` o `CXXFLAGS` de cualquiera de las formas: en el *makefile*, en la línea de órdenes o en el ambiente. Por ejemplo, la siguiente orden emplea en make el compilador de GNU con la opción de optimización:

```
make CXX=g++ CXXFLAGS=-O files.o
```

Aparte de estas, que se encuentran en todas las versiones de make, existen muchas otras, pero tanto sus nombres como sus valores predeterminados dependen del sistema operativo y de la versión de make que utilicemos. Por ejemplo, en el make nativo de Tru64 UNIX[3] no existe una macro para el compilador de

[3] Anteriormente, Digital UNIX, y aún antes, DEC OSF/1.

C++ ni por supuesto para sus opciones; sin embargo en la versión de make de GNU, sí existen.

La opción `-p` de make muestra en la salida estándar todas las reglas y macros que make traiga predefinidos (más todas las variables de ambiente, que make utiliza como macros). Las macros que se listan a continuación tienen nombres especiales de un solo carácter, y son usadas internamente por make para sus reglas implícitas (§8), o se pueden emplear a discreción.

- `@` El objetivo de la regla
- `<` La primera dependencia
- `?` Las deps. más nuevas que el objetivo, separadas por espacio
- `^` Todas las dependencias, separadas por un espacio
- `*` El fichero objetivo menos el sufijo.

Recuérdese que se tienen que hacer preceder del signo dólar `$` para emplearlas.

8. Reglas implícitas

Las reglas implícitas le dicen a make cómo construir un objetivo a partir de las dependencias, cuando tal cosa no se especifica.

Las reglas implícitas trabajan tomando como base las extensiones de los ficheros implicados, y usando una serie de macros predefinidas, como las vistas en la §7.1. Por ejemplo, la compilación de un programa en C++ parte de un fichero con extensión '`.cpp`' (código fuente) para formar uno con extensión '`.o`' (módulo objeto), mediante una orden como `${CXX} -c ${CXXFLAGS} .cpp`. Así que cuando no se especifica nada de esto, make hace uso de la regla implícita que tiene almacenada.

A veces son necesarios varios pasos y ocurre una cadena de reglas implícitas. Por ejemplo, para obtener un módulo objeto (`.o`) a partir de una gramática *yacc* (`.y`) primero hay que obtener el fuente C (`.c`) como paso intermedio[4].

Se puede hacer que make no use ninguna regla implícita, o se puede anular o redefinir una determinada o definir otra nueva.

> Make tiene incorporadas reglas implícitas que le dicen cómo hacer determinadas tareas cuando no se le especifican. ✔

8.1. Uso de las reglas implícitas

Para emplear una regla implícita lo único que hay que hacer es omitir las órdenes. O bien se escribirá una regla sin éstas, o incluso no se escribirá la regla en absoluto, como en el *makefile* 4 para el programa que saluda de una primera lección de C++, ampliado un poco. Recuérdese que la variable especial `^` se

```
hola: hola.o getopt.o getopt1.o
	$(CXX) $(LDFLAGS) -o hola $^
```

Makefile 4: *Makefile* simplificado para el programa que saluda

sustituye por todas las dependencias.

[4] Los ficheros intermedios se borran automáticamente.

Y eso es todo. Como no se menciona de qué dependen los módulos objeto ni cómo obtenerlos, make lo averigua a partir de sus reglas implícitas.

Si hubiera una dependencia adicional que make no conociera, bastaría con poner dicha dependencia. Ejemplo:

```
hola.o: hola.h
```

En cualquier caso la regla implícita le dice a make, en este ejemplo, lo siguiente:

```
hola.o: hola.cpp
        $(CXX) -c $(CXXFLAGS) hola.cpp
```

8.2. Escritura de reglas implícitas (método de sufijos)

Hay dos formas de escribir nuevas reglas implícitas: las reglas de patrones y las reglas de sufijos. Estas últimas son más viejas, pero todas las versiones de make las entienden y por eso se explican brevemente aquí.

Para definir una regla implícita se escribe un objetivo que son dos extensiones de fichero o sufijos concatenados: el primero es el sufijo fuente y el segundo el sufijo objetivo.

Por ejemplo, la regla implícita para compilar un programa en C++ se escribiría:

```
.cpp.o:
        $(CXX) -c $(CXXFLAGS) $<
```

Esto significa que un fichero cuya extensión es '`.o`' depende de otro con el mismo nombre cuya extensión es '`.cpp`' y se construye a partir de él mediante la orden que se muestra. Si las macros `CXX` y `CXXFLAGS` no están definidas automáticamente por make, lo cual depende de la versión, el usuario tendrá que hacerlo.

8.3. Cancelación de una regla implícita

Para reemplazar una regla implícita simplemente se escribe una con los mismos sufijos pero diferentes órdenes.

Los sufijos o extensiones de ficheros conocidos son los nombres de las dependencias del objetivo especial `.SUFFIXES` que inicialmente contiene todos los sufijos conocidos. Para variar esta lista o borrarla simplemente se escribe una regla cuyo objetivo sea el antes citado y cuyas dependencias sean los sufijos que se deseen, o ninguno. Ejemplos:

```
.SUFFIXES: .ps .z    # Se añaden estos sufijos a la lista
.SUFFIXES:           # Se borra por completo la lista
.SUFFIXES: .c .o .h # Una vez borrada, ésta es nuestra propia lista
```

8.4. Definición de una regla predeterminada

Podemos definir una regla implícita para aquellos objetivos y dependencias que no tengan órdenes propias o para los que no se aplique ninguna otra regla implícita. Para ello simplemente habrá que escribir una regla cuyo objetivo sea `.DEFAULT` y cuyas órdenes sean las deseadas para el caso antedicho.

9. Cómo ejecutar make

La forma más simple es llamar a make sin parámetros. Esto construye la primera regla encontrada en un fichero llamado `makefile` o, mejor, `Makefile`. Pero podemos querer usar otro fichero *makefile* u otro compilador, u otras opciones de compilación; o podemos querer ver simplemente qué ficheros están anticuados sin que se recompilen.

Mediante el uso de parámetros y opciones a make, podemos hacer estas cosas y muchas otras.

9.1. Parámetros para especificar el *makefile*

Mediante la opción '`-f`' seguida del nombre del fichero *makefile*. Si no se usa esta opción, make, en UNIX, busca el *makefile* en un fichero que se llame `makefile` y, si éste no existe, `Makefile`. Ejemplo:

```
make -f editor.mk
```

La extensión `mk` es arbitraria; podría ser cualquiera, o no haber.

9.2. Parámetros para especificar el objetivo

En un *makefile* pueden existir varios objetivos. A veces un paquete de programas se compone de varios, y el primer objetivo que se pone es uno que los agrupa a todos, de forma que al llamar a make sin parámetros se reconstruye todo; pero puede ser que se quiera por alguna razón construir solamente uno; o puede que haya un objetivo para borrar lo sobrante en el directorio, o para imprimir los fuentes, y no sea el primero del *makefile*.

Para ello, simplemente al llamar a make se le dice qué objetivo se desea. Por ejemplo, supóngase que el *makefile* contiene al principio:

```
.PHONY: all
all: size nm ld ar as
```

Si no se especifica ningún objetivo, se hará `all` y por tanto todas sus dependencias; pero si sólo se quiere construir el programa `size`, se hará:

```
make size
```

9.3. Parámetros para no ejecutar las órdenes

El *makefile* le dice a make si un objetivo está anticuado y cómo actualizarlo. Pero a veces no se desea esto. Para ello están las siguientes opciones:

`-t` Con *te* de «**t**ocar» (por el programa de UNIX `touch`). Se marcan los objetivos como actualizados pero sin cambiarlos. En otras palabras, make finge compilar los objetivos pero realmente no los modifica.

`-n` Con *ene* de «**n**o operar». Se ven todas las órdenes que make haría pero no se ejecutan realmente. Esta opción es muy útil para no «meter la pata» y para probar con otras opciones.

`-q` Con *cu* de ***q****uestion* («preguntar»). Silenciosamente se mira si los objetivos están ya actualizados, pero no se ejecutan las órdenes en ningún caso; o sea, no se produce compilación ni salida. El único efecto es devolver un código de *status* al *shell*.

La opción '`-n`' es realmente muy útil. Recuérdese y utilícese. ✔

9.4. Parámetros para cambiar macros

Un parámetro que contenga el signo igual especifica el valor de una macro. La orden

`make` *macro*=*valor*

cambia el valor de la macro *macro* a *valor*. Todas las definiciones de esa macro existentes en el *makefile* no se tienen en cuenta. Por ejemplo, se puede hacer que la compilación se efectúe con el compilador de GNU y las opciones de dar toda clase de avisos extra y generar una tabla de símbolos para el depurador mediante:

```
make CXX=g++ CXXFLAGS="-Wall -g"
```

9.5. Resumen de parámetros y opciones para make

`make` [`-f` *fichero*] [*opción*...] [*variable*=[*valor*] ...] [*objetivo* ...]

objetivo El objetivo que va a construirse. Por omisión, el primero del *makefile*.

macro= Anular la macro especificada.

macro=*valor* Dar el valor especificado a la macro nombrada; preferencia sobre definiciones que hubiera en el *makefile*.

`-f` *fichero* El fichero especificado se convierte en el *makefile*. Por omisión, en UNIX se buscan `makefile` o `Makefile`.

`-i` Idéntica a un objetivo `.IGNORE`; se soslayan todos los errores de órdenes.

`-k` Continúa tras un error tanto como sea posible, intentando otros objetivos que no dependen de aquél donde se ha producido el fallo.

`-n` Muestra las órdenes que se ejecutarían, pero no las ejecuta realmente.

`-p` Muestra la base de datos (reglas implícitas y macros predefinidas) que resulta de leer el *makefile*.

`-q` Sólo devuelve un *status*: cero (éxito) si el objetivo a construirse estaba actualizado; no-cero si no.

`-r` Elimina todas las reglas implícitas y la lista de sufijos. Equivale a escribir un objetivo `.SUFFIXES` sin dependencias.

`-s` Con *ese* de silencio. No muestra las órdenes al ejecutarse. Equivale a un objetivo `.SILENT`

`-t` Actualiza los ficheros pero sin modificarlos realmente.

10. Un ejemplo real

Se presenta aquí, como colofón, un *makefile* real y completo: el del programa GNU *hello*, el programa que saluda, basado en el primer programa que se presenta en el libro de Kernigham & Ritchie *El Lenguaje de Programación C*.

Como el listado es muy largo, se presenta poco a poco, numerado y comentado.

Esta sección es muy importante, pues aunque el programa parezca una tontería, ha sido hecho para mostrar cómo se escribe un paquete de *software* para el sistema GNU. Aquí se ven las características más importantes de un *makefile* completo y real.

10.1. Comentarios iniciales

```
# Generated automatically from Makefile.in by configure.
# Makefile for GNU hello.       -*- Indented-Text -*-
# Copyright (C) 1992 Free Software Foundation, Inc.

# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation; either version 2, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

#### Start of system configuration section. ####
```

- La primera línea dice que este *makefile* ha sido generado automáticamente; en efecto, lo ha sido a partir de uno genérico mediante un programa llamado `configure` que detecta varias características del sistema operativo y modifica ciertos ficheros en consecuencia.
- En la segunda línea vemos `-*-Indented-Text-*-`. Esto es para que el editor Emacs se ponga en ese modo; es decir, texto sangrado. Así es más fácil escribir. Emacs intenta adivinar el modo en el que debe ponerse a partir de la extensión del fichero; si éste no tiene, o no es reconocida, se queda en modo `Fundamental`. Pero si en las primeras líneas del fichero ponemos `-*-`*modo*`-*-` entonces cada vez que se edita ese fichero se cambia a ese modo. Obviamente, para que esto no interfiera con el procesamiento normal del fichero, debe ponerse entre comentarios, como aquí.
- El resto se refiere al *copyright*, licencia, (falta de) garantía, etc. Por último se nos anuncia que va a empezar la sección configurable. Aquí se definirán algunas macros que el usuario podría querer modificar.

10.2. Macros configurables

```
srcdir = .
VPATH = .

CC = gcc -O

INSTALL = /usr/bin/installbsd -c
INSTALL_PROGRAM = $(INSTALL)
INSTALL_DATA = $(INSTALL) -m 644

DEFS =  -DSTDC_HEADERS=1 -DHAVE_ALLOCA_H=1
LIBS =

CFLAGS = -g
LDFLAGS = -g

prefix = /usr/local
exec_prefix = $(prefix)

bindir = $(exec_prefix)/bin
infodir = $(prefix)/info

#### End of system configuration section. ####
```

- La macro `srcdir` es el directorio donde están los fuentes, y se iguala al directorio de trabajo actual; lo mismo que `VPATH`, que es una macro especial de GNU make.
- Al igual que la macro `CXX` contiene el nombre del compilador de C++, `CC` contiene el del compilador de C, y `CFLAGS` sus opciones. El compilador que se empleará de forma predeterminada en este programa será GNU C; obsérvese que se le da aquí una opción, `-O`, para aplicar el *optimizador* de código. ¿No debería ponerse esto en `CFLAGS`? En general sí, pero ésta es una forma de asegurarse de que siempre que se emplee `$CC` se va a aplicar esta opción, aun cuando no se haya puesto `$CFLAGS`.
- Las tres macros siguientes se refieren al programa de instalación. Usaremos `install` en la versión BSD: `installbsd`. Para instalar el programa ejecutable emplearemos `installbsd -c`, que copia el ejecutable con los permisos y propietarios adecuados. Para instalar la documentación usaremos `installbsd -c -m 644`, que copia los ficheros cambiando los permisos a 644 (`rw-r--r--`).
- La macro `DEFS` define dos macros para el preprocesador de C, que se pasarán como opciones a `gcc`. La macro `LIBS` está vacía; representa bibliotecas de funciones adicionales que hicieran falta; si para saludar hubiera que calcular una raíz cuadrada, habría que enlazar con la biblioteca matemática y debería valer `-lm`, por ejemplo.

10.3. Macros no configurables

```
SHELL = /bin/sh
```

```
SRCS = hello.c version.c getopt.c getopt1.c alloca.c
OBJS = hello.o version.o getopt.o getopt1.o
HDRS = getopt.h
DISTFILES = $(SRCS) $(HDRS) COPYING ChangeLog NEWS Makefile.in \
            README INSTALL hello.texi gpl.texinfo configure \
            configure.in texinfo.tex hello.info testdata TAGS
```

- Esta sección es de macros que ya no deberían ser modificadas. La primera no haría falta ponerla, le dice a make que use el *shell* de Bourne, pero eso es lo normal. Quizá los autores lo han puesto para que el lector sepa que quieren que se use ése y no otro.
- Las demás macros son, por orden, los ficheros fuente, los módulos objeto, las cabeceras (sólo una en este caso, pero en un futuro podrían añadirse más), y los ficheros de distribución; o sea, todos los que componen el paquete original: fuentes, documentación, etc.

10.4. Reglas principales de construcción e instalación

```
all: hello hello.info

doc: hello hello.info hello.dvi

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

install: all
        $(INSTALL_PROGRAM) hello $(bindir)/hello
        -$(INSTALL_DATA) $(srcdir)/hello.info $(infodir)/hello.info

hello: $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)
```

- Empezamos con las reglas. La regla predeterminada es `all` por ser la primera; como depende de `hello` y `hello.info`, se construirán estos ficheros si procede.
- Si queremos la documentación completa, además del programa, emplearemos el objetivo `doc`, que es como el anterior pero crea el fichero `hello.dvi`; este fichero es una representación de la documentación que puede verse o imprimirse mediante ciertos programas.
- La regla del objetivo `.c.o.` es una regla de sufijos (§8.2, pág. 19). Así alteramos la que make tiene predefinida para compilar programas en C, introduciendo las macros `CPPFLAGS` (incorporada también en make), que representa opciones para el preprocesador, y `DEFS`, que son definiciones de macros del preprocesador de C.
- La regla para la instalación responde al objetivo `install`. Observe que depende de `all`; esto es comprensible, pues si no está hecho el programa, ¿qué se puede instalar? En las dos líneas de órdenes se instala con la primera el ejecutable, y con la segunda la documentación interactiva; si esta instalación falla por algún motivo (como que en el sistema no se use

el sistema de información Info), entonces el error no se tiene en cuenta. Este es el significado del guion inicial.

➤ Con la regla para `hello` se construye el ejecutable. Los ejecutables dependen de los módulos objeto, no directamente de los fuentes. En la orden no se emplea la macro `CFLAGS`, pues no se está compilando, sino `LDFLAGS`, incorporada también en make, que representa opciones para el enlazador, que es quien está actuando. La macro especial `@` representa al propio objetivo, y `LIBS` eran bibliotecas adicionales, que aquí no se usan.

10.5. Reglas intermedias o secundarias

```
hello.o getopt.o getopt1.o: getopt.h

hello.info: hello.texi
        makeinfo $(srcdir)/hello.texi
hello.dvi: hello.texi
        texi2dvi $(srcdir)/hello.texi
```

➤ Las reglas cuyos objetivos son los módulos objeto aparecen simplificadas; como todos se compilan igual, se ponen juntos; sus dependencias son la cabecera y los fuentes de cada uno, pero éstos no tienen por qué ponerse, pues make los deduce. Las órdenes de compilación se omiten, pues se aplica la regla implícita modificada anteriormente por el objetivo `.c.o.`.

➤ Las dos reglas siguientes son para la documentación. Observe que el sistema es tal que a partir de un solo fichero fuente, `hello.texi`, escrito en un formato llamado TEXinfo, se obtiene la documentación interactiva en `hello.info` mediante el programa `makeinfo`, o la documentación lista para imprimirse en papel o verse en una pantalla, en el fichero `hello.dvi`, mediante el programa `texi2dvi`.

10.6. Reglas adicionales

```
check: hello
        @echo expect no output from diff
        ./hello > test.out
        diff -c $(srcdir)/testdata test.out
        rm -f test.out

Makefile: Makefile.in config.status
        ./config.status

config.status: configure
        $(srcdir)/configure --no-create

configure: configure.in
        autoconf

TAGS: $(SRCS)
        etags $(SRCS)
```

- Ahora vienen reglas adicionales, aunque también importantes. Aquélla cuyo objetivo es `check` se encarga de comprobar, de forma simple, si el programa funciona. Evidentemente su dependencia es el propio programa, pues si no existe no se puede comprobar si funciona. A través de varias órdenes se efectúa la tarea. Primero se nos muestra un mensaje. El signo `@` no forma parte de la orden, sino que hace que `make` no la muestre al ejecutarse. Se ejecuta el programa `hello` mandando la salida a un fichero, el cual se compara con uno incluido en la distribución. Estos dos ficheros deben ser iguales; como dice el mensaje, si hay alguna salida, habrá algún error. Por último se borra el fichero generado por el programa.

- Las tres reglas siguientes se refieren a la recreación del propio *makefile*, a partir del de la distribución, `Makefile.in`, y de un programa llamado `configure`, el cual también puede generarse a partir de otro llamado `autoconf`, que usa un fichero de datos general llamado `configure.in`; `configure` puede generar directamente el *makefile* o un guion del *shell* llamado `config.status`, que puede ejecutarse luego.

- El fichero `TAGS` se obtiene al aplicar el programa `etags` a los fuentes; `etags` es la versión adaptada a Emacs (de ahí la *e*) de la orden `tags`; ésta crea en el fichero antedicho una lista de todas las funciones del programa con un formato tal que es reconocido por el editor mediante ciertas órdenes que posee; con éstas puede el programador pasar cómoda y rápidamente en el editor de una función a otra, aunque estén en distintos ficheros.

10.7. Reglas de borrado

```
clean:
        rm -f hello *.o core test.out

mostlyclean: clean

distclean: clean
        rm -f Makefile config.status

realclean: distclean
        rm -f TAGS
```

- Reglas para borrar los ficheros sobrantes, una vez que se está satisfecho con la compilación y seguramente se ha probado e instalado el programa. El objetivo `clean` borra los módulos objeto, el ejecutable, un posible volcado de memoria si ocurrió algún fallo y la salida de la comprobación. La opción `-f` hace que si no existe alguno de estos ficheros o no se puede borrar, no se produzcan mensajes de error ni `rm` considere que ha fallado.

- El objetivo `mostlyclean` depende del anterior, por lo que borra lo mismo que éste. Normalmente puede borrar más ficheros, pero aquí no hay nada más que borrar.

- El objetivo `distclean` borra lo mismo que `clean` y además los ficheros que pueden ser regenerados mediante `configure`, quedándonos con la distribución original.

➤ Por último, el borrado más potente lo conseguimos con `realclean`, que deja el paquete en su estado original.

10.8. Regla de distribución

```
dist: $(DISTFILES)
        echo hello-‘sed -e ’/version/!d’ -e \
                ’s/[^0-9.]*\([0-9.]*\).*/\1/’ -e \
                 q version.c‘ > .fname
        rm -rf ‘cat .fname‘
        mkdir ‘cat .fname‘
        ln $(DISTFILES) ‘cat .fname‘
        tar chzf ‘cat .fname‘.tar.gz ‘cat .fname‘
        rm -rf ‘cat .fname‘ .fname

# Prevent GNU make v3 from overflowing arg limit on SysV.
.NOEXPORT:
```

➤ El objetivo `dist` prepara los ficheros para ser distribuidos. Depende por tanto de los ficheros originales de la distribución. La primera orden usa `sed` para extraer del fichero fuente `version.c` el número de versión del programa. Con esto se crea un directorio llamado `hello-`*versión*, se enlazan los ficheros de la distribución a él, se archiva y comprime el directorio y se borra. El resultado es un archivo comprimido `hello-`*versión*`.tar.gz`, que puede ponerse en un FTP anónimo para su distribución general, por ejemplo.

➤ La línea de comentario dice para qué sirve el objetivo especial `.NOEXPORT`, que sólo existe en GNU make.

10.9. Utilización

Y ahora un ejemplo de uso. Se supone que partimos de la distribución original y que hemos ejecutado `configure`.

Primero vamos a crear el ejecutable, vamos a compilar. De paso comprobamos que el programa funciona, que para eso hay un objetivo *check*. Además, como éste tiene como dependencia el ejecutable, se construye.

```
$  make check
g++ -O -c -g  -DSTDC_HEADERS=1 -DHAVE_ALLOCA_H=1 hello.c
g++ -O -c -g  -DSTDC_HEADERS=1 -DHAVE_ALLOCA_H=1 version.c
g++ -O -c -g  -DSTDC_HEADERS=1 -DHAVE_ALLOCA_H=1 getopt.c
g++ -O -c -g  -DSTDC_HEADERS=1 -DHAVE_ALLOCA_H=1 getopt1.c
g++ -O -g -o hello hello.o version.o getopt.o getopt1.o
expect no output from diff
./hello > test.out
diff -c ./testdata test.out
*** ./testdata  vi 12 may 13:30:15 1995
--- test.out    vi 12 may 13:32:05 1995
***************
*** 1 ****
! hello, world
--- 1 ----
```

```
! Hello, world!
*** Exit 1
```

Algo parece haber ido mal. La compilación no ha dado ningún aviso ni error, pero el mensaje decía que no se esperaba ninguna salida del programa `diff` y sí se ha producido. Se ve que ha habido una diferencia entre el fichero hecho por el programador con la salida que él pensaba que tenía que salir y la salida real del programa. Pero esta diferencia ha consistido en que el programa saluda por omisión con "hello" en mayúscula. Esto no puede considerarse un fallo; podríamos cambiar la forma de probar el programa, modificarlo para cambiar el comportamiento por omisión, modificar el fichero con la salida esperada, o dejarlo como está, que es lo que haremos. En resumen, parece un pequeño *gazapo* del programador.

Una vez hecho el programa, lo instalamos.

```
$  make install
makeinfo ./hello.texi
/usr/bin/installbsd -c ./hello /usr/local/bin/hello
/usr/bin/installbsd -c -m 644 ./hello.info /usr/local/info/hello.info
```

Si se tuvieran los suficientes permisos, el ejecutable `hello` se hubiera copiado (opción `-c`) en el directorio local para ejecutables; y la documentación en formato *Info*, con los permisos 644 (`rw-r-r-`), en el directorio adecuado. Si se desea probar, cámbiese en el *makefile* estos directorios por otros donde se tengan permisos de escritura.

Ahora vamos a generar la documentación para imprimirla posteriormente.

```
$  make doc
texi2dvi hello.texi
This is TeX . . . (mensajes suprimidos)
```

Esto nos ha generado el fichero `hello.dvi`, el cual podemos:

- Verlo:
  - En una terminal alfanumérica, con `dvi2tty`.
  - En una terminal X-Window, con `xdvi`.
  - En un PC con MS-Windows, con `dviwin`.
- Imprimirlo:
  - En una impresora PostScript, con `dvips`.
  - En cualquier impresora conectada a un PC con MS-Windows y con su controlador correspondiente, con `dviwin`.
- Analizarlo (`dvitype`), manipularlo (`dvicopy`, `dviselect`, ...) y borrarlo (`rm`).

Los programas mostrados no tienen que ser éstos; hay otros muchos para hacer lo mismo.

Por último, limpiamos el directorio para conservar espacio en disco.

```
$  make clean
rm -f hello *.o core test.out
```

Esto ha sido todo. No estaría mal copiar este programa y practicar, no sólo con el *makefile* sino con el programa en sí. Se puede obtener por FTP anónimo de

```
http://ftp.gnu.org/gnu/hello/hello-1.3.tar.gz
```

Este documento se escribió con Emacs, procesó con LaTeX 2_ε, y convirtió a PDF mediante `pdflatex`. La versión de make empleada para las pruebas ha sido GNU Make versión 3.79.1, en un sistema GNU/Debian Linux.

Se han usado los tipos EC a 10 pt generados por METAFONT.

Se terminó en Cádiz, en mayo de 1995, y fue revisado en noviembre de 2000. En noviembre de 2003 se actualizó la URL del programa *hello* y el colofón. En marzo de 2007 se ha vuelto a revisar. Ha sido inestimable, como siempre, la ayuda de mi amigo y compañero Francisco Palomo Lozano, del Departamento de Lenguajes y Sistemas Informáticos.