1. Introducción

La herramienta make es utilizada habitualmente en el desarrollo de proyectos con gran número de archivos interdependientes. Utiliza reglas definidas en un archivo llamado 'makefile' para construir un archivo 'destino' especificado por el usuario a partir de un cierto número de archivos 'fuente', de los que el destino depende. El archivo destino se crea sólo si no existe o está desactualizado, es decir, si alguno de los archivos fuente es más reciente que el archivo destino. Dicho de otra forma, el archivo destino es construido de nuevo solo si alguno de los archivos fuente ha sido modificado desde la última vez que el archivo destino fue creado. Esto implica que al construir un proyecto completo compuesto de muchos archivos, solo aquellos desactualizados se volverán a construir, ahorrando de este modo una cantidad considerable de tiempo y recursos.

Make es invocado de la siguiente forma:

make [opciones] [destino(s)]

Las opciones más habituales son:

- **-h**: Muestra ayuda sobre make.
- **-f archivo**: Indica que el archivo que contiene las reglas no se llama 'makefile' ni 'Makefile', sino 'archivo'.
- **-n**: Muestra los comandos que make ejecutaría, pero sin ejecutarlos realmente. Útil para testear makefiles.

A lo largo de este tutorial, veremos como definir destinos y las reglas para crearlos.

2. Dependencias

Como se puede imaginar, make es especialmente útil al programar aplicaciones relativamente grandes con cierta cantidad de archivos de distintos tipos. A partir de las reglas especificadas en el makefile, make crea un árbol de dependencias que es utilizado para construir el destino solicitado por el usuario. Veamos el siguiente ejemplo, en el que tenemos una aplicación compuesta de los siguientes archivos:

- data.c
- data.h
- main.c
- io.c
- io.h

El archivo de código fuente data.c incluye al archivo de cabecera data.h, mientras que io.c incluye a io.h. El archivo main.c incluye a ambos. Sin make, habría que utilizar los siguientes comandos para compilar la aplicación y crear un ejecutable llamado project1:

```
gcc -c data.c
gcc -c io.c
gcc -c main.c
gcc -o project1
data.o main.o
io.o
```

Ejemplo 1

Así creamos los archivos objeto a partir de los archivos de código y de cabecera apropiados en cada caso, para después utilizar los tres archivos objeto generados para construir un archivo ejecutable. make inferirá el árbol de dependencias a partir de las reglas incluidas en el makefile mostrado en la figura. La utilidad de make radica en el hecho de que construirá un archivo si y solo si cualquiera de sus fuentes (o fuentes de las fuentes, y así sucesivamente) es más reciente que el archivo.

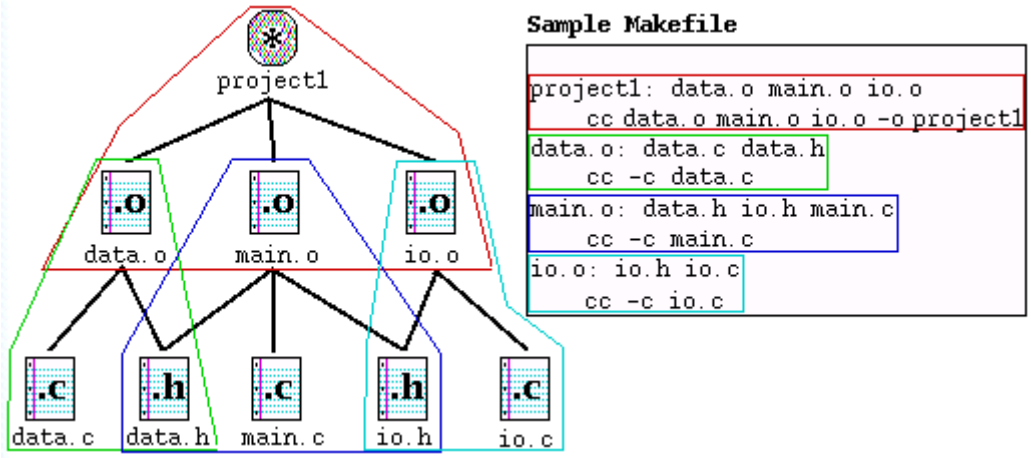

Figura 1: (de http://www.eng.hawaii.edu/Tutor/Make/)

Veamos algunos casos:

- Si el destino a construir es 'io.o' pero las horas y fecha de 'io.h' y 'io.h' son anteriores que las de 'io.o', no se realiza ninguna acción.
- Si el destino es 'data.o' y 'data.c' es más antiguo pero 'data.h' más reciente, 'data.o' será reconstruido a partir de ambos, utilizando la regla apropiada.
- Si el destino es el ejecutable 'project1' y el resto de archivos son más antiguos con la excepción de io.h: 'io.o' será reconstruido, y a continuación 'project1' será reconstruido. Hay que destacar que 'data.o' y 'main.o' no son reconstruidos.

3. Destinos y reglas

Como ya hemos mencionado, un makefile se compone de reglas que indican a make como construir un destino a partir de sus fuentes. Una regla tiene el siguiente formato genérico:

```
destino: [lista
de dependencias]
[comando 1
comando 2
....
comando N]
```

Estructura 1

Veamos el significado de cada elemento:

- **destino**: el archivo a construir, por ejemplo 'data.o'
- **lista de dependencias**: lista con todos los archivos fuentes de los que el destino depende *directamente*por ejemplo. 'data.c data.h'. El/los comando(s) se ejecutan solo si alguno de estos archivos fuente son más recientes que el destino. Si no se proporciona esta lista, los comandos se ejecutan siempre.
- **comandos**: comandos que serán interpretados por el sistema operativo, y que normalmento se utilizan para construir el destino, aunque se puede utilizar cualquier tipo de comando. Es *muy importante* recordar que *cada comando debe ir precedido de un tabulador,* de otro modo make produciría un error. Para dividir un comando en varias líneas, se puede utilizar el carácter '\' tras cada línea. Un ejemplo típico de comando sería 'gcc -c data.c'.

Un ejemplo de regla en un makefile sería, por tanto:

```
data.o: data.c
data.h
gcc -c data.c
```

Ejemplo 2

Ahora presentamos un ejemplo que utilizaremos a lo largo del tutorial: supongamos que tenemos una aplicación capaz de planificar un viaje calculando el tiempo que lleva conducir de una ciudad a otra a una velocidad dada. La aplicación se compone de los siguientes archivos:

- **cities.txt**: archivo en el que el usuario introduce las ciudades que quiere incluir en la aplicación, en unalista de texto.
- **dist_table.dat**: archivo que contiene una tabla con todas las ciudades y las distancias entre ellas.
- **create_table.c**: código fuente para un programa que comprueba que todas las ciudades incluidas en cities.txt están incluidas en la tabla de 'dist_table.dat', si alguna falta pide las distancias desde ella al resto de ciudades y las incluye en la tabla.
- **travel_time.c**: código fuente para un programa que calcula el tiempo necesario para viajar entre dos ciudades a una velocidad dada en km/h (por ejemplo 'travel_time Madrid Santander 120').
- **input.h**: archivo de cabecera con funciones para leer de un archivo, incluido por ambos archivos de código fuente.
- **foutput.h**: archivo de cabecera con funciones para escribir en un archivo, incluido por create_table.c.
- **scr_io.h**: archivo de cabecera con funciones para leer o escribir por pantalla en un archivo, incluido por ambos archivos de código fuente.

El árbol de dependencias es el siguiente:

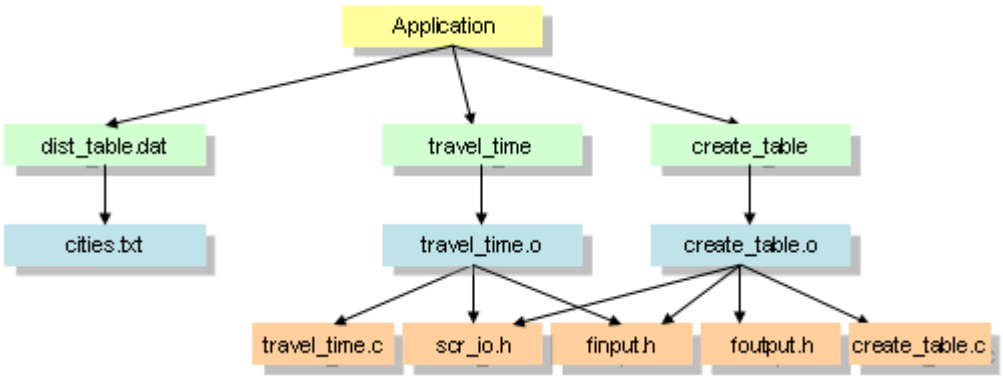

Figura 2

Utilizando lo que hemos aprendido hasta ahora, el makefile tendría la siguiente forma:

```
# Makefile para la aplicación de cálculo del tiempo de viaje

travel_time: travel_time.o
   gcc -o travel_time travel_time.o

create_table: create_table.o
   gcc -o create_table create_table.o

travel_time.o: travel_time.c scr_io.h finput.h
   gcc -c travel_time.c

create_table.o: create_table.c scr_io.h finput.h foutput.h
   gcc -c create_table.c

dist_table.dat: cities.txt
   create_table cities.txt
```

Ejemplo 3

Añadir comentarios al makefile suele ser muy útil, para hacerlo solo es necesario comenzar una línea cualquiera con el carácter '#'', como se muestra en el ejemplo. Observe que, en realidad, podríamos compilar los ejecutables directamente desde los archivos .c y .h, pero el paso adicional a través de los archivos objeto (.o) nos permitirá mostrar más características de make.

4. Destinos simbólicos

Utilizando el makefile en el Ejemplo 3, podemos crear nuestra aplicación mediante las siguientes llamadas a make:

```
> make
create_table
> make
travel_time
> make
dist_table.dat
```

Ejemplo 4

Hasta ahora todos los destinos son archivos, de modo que no podemos construir toda la apliación con una sola llamada a make. Para hacer esto, puede incluir *destinos simbólicos*, es decir, destinos con un nombre cualquiera que no se corresponden con archivos. Por ejemplo, podemos introducir un destino llamado 'all' para construir (actualizar) la aplicación completa con una sola llamada a make.

```
# Makefile para la aplicación de cálculo del tiempo de viaje con destinos
simbólicos
all: travel_time create_table dist_table.dat

travel_time: travel_time.o
   gcc -o travel_time travel_time.o

create_table: create_table.o
   gcc -o create_table create_table.o

travel_time.o: travel_time.c scr_io.h finput.h
   gcc -c travel_time.c

create_table.o: create_table.c scr_io.h finput.h foutput.h
   gcc -c create_table.c

dist_table.dat: cities.txt
   create_table cities.txt
```

Ejemplo 5

En el ejemplo hemos incluido un segundo destino simbólico llamado 'clean', que carece de lista de dependencias, con lo que los comandos se ejecutan siempre si el destino se invoca. Con este makefile, construir la aplicación completa se reduce a teclear 'make all'. Es más, si no se especifica ningún destino, make asume que solo se desea construir el primero, de modo que podríamos utilizar simplemente 'make'. Para borrar todos los archivos objeto, utilizaríamos 'make clean'.

5. Macros

Llegados a este punto, ya ha aprendido todo lo necesario para construir cualquier tipo de makefile para cualquier tipo de aplicación. A partir de ahora, veremos como reducir el tamaño de estos makefiles y hacerlos tan reutilizables como sea posible. Comenzaremos con las macros o variables. Las macros de make son similares a las variables utilizadas en la programación en shell script, puesto que almacenan cadenas que pueden ser referencias a través de los nombres de esas macros. Se inicializan mediante NOMBRE = valor, y se acceden mediante $(NOMBRE). Se suele utilizar la convención de escribir el nombre

de las macros en mayúsculas. Podemos modificar nuestro ejemplo para incluir una macro con el nombre del archivo que contiene la tabla de distancias de la siguiente manera:

```
# Makefile for travel time application with a macro

DATA = dist_table.dat

all: travel_time create_table $(DATA)

travel_time: travel_time.o
   gcc -o travel_time travel_time.o

create_table: create_table.o
   gcc -o create_table create_table.o

travel_time.o: travel_time.c scr_io.h finput.h
   gcc -c travel_time.c

create_table.o: create_table.c scr_io.h finput.h foutput.h
   gcc -c create_table.c

$(DATA): cities.txt
   create_table cities.txt

clean:
   rm *.o
```

Ejemplo 6

Ahora es más sencillo cambiar el nombre de este archivo, puesto que solo es necesario cambiar una línea del makefile. Por supuesto, esto se puede extender a los nombres de otros archivos, como muestra el siguiente ejemplo:

```
# Makefile para la aplicación de cálculo del tiempo de viaje con
# sustitución de texto en macros

PROG1 = travel_time.c
PROG2 = create_table.c

all: $(PROG1:.c= ) $(PROG2:.c= ) dist_table.dat

$(PROG1:.c= ): $(PROG1:.c=.o)
  gcc -o $(PROG1:.c= ) $(PROG1:.c=.o )

$(PROG2:.c= ): $(PROG2:.c=.o)
  gcc $(PROG2:.c= ) $(PROG2:.c=.o )

$(PROG1:.c=.o): $(PROG1) scr_io.h finput.h
  gcc -c $(PROG1)

$(PROG2:.c=.o): $(PROG2) scr_io.h finput.h foutput.h
  gcc -c $(PROG2)

dist_table.dat: cities.txt
  $(PROG2:.c= ) cities.txt

clean:
  rm *.o
```

Ejemplo 7

El siguiente ejemplo utiliza la posibilidad de sustituir texto en una macro: para sustituir una subcadena en la macro por otra, se puede hacer $(MACRO:antigua_subcadena=nueva_subcadena). Esta herramienta se aprovecha en el ejemplo para utilizar la misma macro para el nombre del ejecutable, el archivo de código fuente y el archivo objeto. También existe la posibilidad de definir una macro en la llamada a make, haciendo 'make NOMBRE_MACRO=valor'. Por ejemplo, en este ejemplo la macro DATA se define llamando a make con *'make DATA=dist_table.dat'*

```
# Makefile para la aplicación de cálculo del tiempo de viaje con
# sustitución de texto en macros

PROG1 = travel_time
PROG2 = create_table

all: $(PROG1) $(PROG2) $(DATA)

$(PROG1): $(PROG1).o
  gcc -o $(PROG1) $(PROG1).o

$(PROG2): $(PROG2).o
  gcc -o $(PROG2) $(PROG2).o

$(PROG1).o: $(PROG1).c scr_io.h finput.h
  gcc -c $(PROG1).c

$(PROG2).o: $(PROG2).c scr_io.h finput.h foutput.h
  gcc -c $(PROG2).c

$(DATA): cities.txt
  $(PROG2) cities.txt

clean:
  rm *.o
```

Ejemplo 8

No obstante, si no se define DATA al llamar a make, no se sustituirá nada.

Para acabar, presentamos una lista con algunas de las macros predefinidas por make que pueden ser muy útiles al escribir un makefile:

- **$(CC)**: devuelve el nombre del compilador de C por defecto (puede ser cc, gcc, etc)
- **$(CFLAGS)**: opciones para el compilador de C
- **$@**: nombre del destino de la regla actual
- **$?**: lista con los archivos más antiguos que el destino en la lista de dependencias de la regla actual
- **$^**: lista con todos los archivos en la lista de dependencias de la regla actual
- **$<**: archivo en la lista de dependencias compuesta de un solo archivo en la regla actual

En este último ejemplo, se muestra un uso intensivo de estas y otras macros:

```
# Makefile para la aplicación de cálculo del tiempo de viaje con macros
PROG1 = travel_time
PROG2 = create_table
INCLUDE1 = scr_io.h finput.h
INCLUDE2 = foutput.h
DATA = dist_table.dat
CITIES = cities.txt

all: $(PROG1) $(PROG2) $(DATA)

$(PROG1): $(PROG1).o
  $(CC) -o $@ $^

$(PROG2): $(PROG2).o
```

```
  $(CC) -o $@ $^

$(PROG1).o: $(PROG1).c $(INCLUDE1)
  $(CC) -c $(PROG1).c

$(PROG2).o: $(PROG2).c $(INCLUDE1) $(INCLUDE2)
  $(CC) -c $(PROG2).c

$(DATA): $(CITIES)
  $(PROG2) $^

clean:
  rm *.o
```

Ejemplo 9

6. Reglas implícitas

Algunas de las reglas en un makefile son bastante similares entre sí. Por ejemplo, las dos primeras reglas de nuestro ejemplo (las que construyen los ejecutables de los dos programas) son idénticas, con la única diferencia en los nombres de los archivos. Las *reglas implícitas* son aquellas que se utilizan para construir destinos sin incluir una regla específica para ellos en el makefile. El funcionamiento es el siguiente: si no se define una regla específica para el destino que se indica a make, se intenta utilizar una de estas reglas implícitas, primero una de las definidas por el usuario, y si ninguna es apropiada una de entre las predefinidas por make.

Comencemos presentando dos reglas implícitas predefinidas por make:

- **Construcción de archivos objeto**: cuando no se especifica una regla concreta para construir (actualizar) un archivo objeto, make intenta construirlo a partir de un archivo de código fuente con el mismo nombre pero extensión .c en lugar de .o, utilizando el comando `$(CC ) -c $(CPPFLAGS) $(CFLAGS)`. La macro CPPFLAGS incluye las opciones para archivo de código fuente de C++.
- **Construcción de ejecutables**: cuando no se especifica una regla concreta para construir (actualizar) un ejecutable, ,make intenta construirlo a partir de un archivo objeto con el mismo nombre y extensión .o, utilizando el comando `$(CC ) $(LDFLAGS) $(LOADLIBS)`. La macro LDFLAGS contiene por defecto '-o', mientras que LOADLIBS se puede utilizar para especificar archivos objeto adicionales a ser incluidos.

La aplicación de la segunda regla predefinida nos permite eliminar las dos primera reglas de nuestro makefile, puesto que make usará la regla implícita predefinida exactamente de la misma forma para crear los porgramas. Este makefile produce un resultado idéntico al anterior, pero es considerablemente más corto:

```
# Makefile para la aplicación de cálculo del tiempo de viaje con
# reglas predefinidas

PROG1 = travel_time
PROG2 = create_table
INCLUDE1 = scr_io.h finput.h
INCLUDE2 = foutput.h
DATA = dist_table.dat
CITIES = cities.txt

all: $(PROG1) $(PROG2) $(DATA)

$(PROG1).o: $(PROG1).c $(INCLUDE1)
  $(CC) -c $(PROG1).c

$(PROG2).o: $(PROG2).c $(INCLUDE1) $(INCLUDE2)
  $(CC) -c $(PROG2).c
```

```
$(DATA): $(CITIES)
  $(PROG2) $^

clean:
  rm *.o
```

Ejemplo 10

Observe que la primera regla implícita predefinida no se puede usar directamente en nuestro caso, puesto que no incluiría los archivos de cabecera (extensión .h) en la lista de dependencias. Una posible solución es modificar la macro CFLAGS para incluir estos archivos con la opción -i del compilador de C. La única desventaje es que las reglas para construir ambos archivos objeto incluirían los tres archivos de cabecera, pero esto no es muy grave en este caso:

```
# Makefile para la aplicación de cálculo del tiempo de viaje con
# reglas predefinidas

PROG1 = travel_time
PROG2 = create_table
INCLUDE1 = scr_io.h finput.h
INCLUDE2 = foutput.h
DATA = dist_table.dat
CITIES = cities.txt

CFLAGS = -i$(INCLUDE)

all: $(PROG1) $(PROG2) $(DATA)

$(DATA): $(CITIES)
  $(PROG2) $^

clean:
  rm *.o
```

Ejemplo 11

Si compara la longitud de este ejemplo con la de, por ejemplo, el Ejemplo 9, queda claro hasta que punto se puede llegar a simplificar un makefile sin perder funcionalidad.

Además de las reglas implícitas predefinidas por make, puede definir sus propias reglas implícitas, o modificar aquellas predefinidas, utilizando la siguiente sintaxis para especificar la regla implícita en el makefile:

```
%.o : %.c
  $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Ejemplo 12

De hecho, ésta es la regla predefinida utilizada por make para construir cualquier archivo objeto cuando no existe ninguna regla específica. Observe que la opción -o no es necesaria para la mayoría de compiladores, pero aun así se incluye por generalidad. Supongamos que desea que los nombre de los archivos involucrados se muestren en un mensaje cada vez que se aplica esta regla. En ese caso incluiría la siguiente regla implícita en el makefile:

```
%.o : %.c
  $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
  echo File $@ has been created from $^;
```

Ejemplo 13

Evidentemente, hay una gran cantidad de situaciones en la que el uso de reglas implícitas originales o modificadas puede ahorrar mucho tiempo, especialmente en proyectos complejos con gran número de

archivos. Un redefinición de esta regla más realista sería la siguiente, que utiliza una macro INCLUDEDIR que contiene el directorio en el que se almacenan los archivos de cabecera y una macro INCLUDE con los archivos de cabecera comunes a ser incluidos en cualquier caso. De esta forma, permite crear un archivo objecto a partir de los archivos de código fuente y de cabecera con el mismo nombre y los archivos de cabecera comunes:

```
%.o : %.c $(INCLUDEDIR)/%.h $INCLUDE
  $(CC) -c $(CFLAGS) $< -o $@
  echo File $@ has been created from $^;
```

Ejemplo 14

7. Destinos predefinidos y prefijos

Make proporciona algunos destinos predefinidos que aumentan su funcionalidad:

- **.PHONY nombre**: especifica que *nombre* no es el nombre de un archivo, sino el nombre de un destino simbólico. Esto evita un error cuando se llama a make para construir un destino simbólico con el mismo nombre que un archivo, puesto que make intentará construir el archivo por defecto.
- **.DEFAULT commands**: los comandos especificados aquí se ejecutarán para todos los destinos para los que no haya disponible ninguna regla explícita ni implicíta.

Por otra parte, hay dos prefijos para comandos con las siguientes características:

- **@comando**: la salida del comando no se mostrará en la pantalla
- **-comando**: make no mostrará un mensaje de error ni se detendrá si el comando produce un error

Podemos modificar nuestro ejemplo para incluir alguna de estas opciones:

```
# Makefile para la aplicación de cálculo del tiempo de
# viaje con destinos predefinidos y prefijos

PROG1 = travel_time
PROG2 = create_table
INCLUDE = scr_io.h finput.h foutput.h
DATA = dist_table.dat
CITIES = cities.txt

CFLAGS = -i$(INCLUDE)

all: $(PROG1) $(PROG2) $(DATA)

#Mostrar un mensaje en lugar de la salida del programa
$(DATA): $(CITIES)
  @$(PROG2) $^
  echo $@ has been created from $^
#No pasa nada si rm devuelve un error porque no hay archivos .o que borrar
clean:
  -rm *.o

.PHONY : all
.PHONY : clean
.DEFAULT echo 'No hay disponible ninguna regla para este destino'
```

Ejemplo 15

8. Condiciones

En el último apartado de este tutorial presentamos las condiciones (decisiones) que make permite definir dentro de una regla. Las principales son las siguientes:

- **ifdef M1**: Ejecuta los siguientes comandos si la macro M1 está definida, es decir, si se la ha

asignado un valor
- **ifndef M1**: Ejecuta los siguientes comandos si la macro M1 no está definida
- **ifeq (M1,M2)**: Ejecuta los siguientes comandos si las macros M1 y M2 tienen el mismo valor
- **ifneq (M1,M2)**: Ejecuta los siguientes comandos si las macros M1 y M2 tienen distintos valores
- **else**: Tras cualquiera de las cuatro anteriores, especifica los comandos a ejecutar en caso de que no se cumpla la condición
- **endif**: Finaliza una estructura condicional

Para ilustrar el uso de condiciones, modificamos el Ejemplo 9 de forma que podamos definir un directorio de destino para el primer programa, que puede ser especificado mediante una macro DESTDIR que contiene el directorio deseado al llamar a make (mediante '*make DESTDIR=valor*) o tomar el valor de un directorio por defecto especificado en la macro DEFAULTDESTDIR si DESTDIR no está definida:

```
# Makefile para la aplicación de cálculo del tiempo de viaje con macros

PROG1 = travel_time
PROG2 = create_table
INCLUDE1 = scr_io.h finput.h
INCLUDE2 = foutput.h
DATA = dist_table.dat
CITIES = cities.txt

DEFAULTDESTDIR = ./bin

all: $(PROG1) $(PROG2) $(DATA)

$(PROG1): $(PROG1).o
ifdef DESTDIR
  #No se produce un error si el directorio ya existe
  -mkdir $(DESTDIR)
  gcc -o $(DESTDIR)/$@ $^
else
  -mkdir $(DEFAULTDESTDIR)
  gcc -o $(DEFAULTDESTDIR)/$@ $^
endif

$(PROG2): $(PROG2).o
  gcc -o $@ $^

$(PROG1).o: $(PROG1).c $(INCLUDE1)
  gcc -c $(PROG1).c

$(PROG2).o: $(PROG2).c $(INCLUDE1) $(INCLUDE2)
  gcc -c $(PROG2).c
```

Ejemplo 16

9. Como aprender más

La descripción detallada de todas las posibilidades que ofrece make y de todos los trucos para escribir makefiles más cortos y efectivos escapa al propósito de este tutorial. No obstante, hay una cierta cantidad de tutoriales en Internet que puede utilizar para mejorar sus habilidades a la hora de utilizar make y los makefiles en proyectos realmente complejos. Aquí tiene un selección:

- En español:
  - www-etsi2.ugr.es/depar/ccia/mp2/old/apoyo/make/. Versión local aquí.
  - casa.ccp.servidores.net/curso-gnu-c/gnu_c/make.html. Versión local aquí.
- En inglés:
  - www.gnu.org/software/make/manual/html_node/make_toc.html. Versión local aquí.
  - www.eng.hawaii.edu/Tutor/Make/. Versión local aquí.
  - www.cs.indiana.edu/classes/c304/Makefiles.html. Versión local aquí.

Si prefiere un libro, dispone de una guía bastante extensa y útil, con multitud de ejemplos:

- **"Managing projects with make"**. Andrew Oram, Steve Talbot. O´Reilly & Associates Inc. 1991. **L/S 004.451.9 UNIX ORA**

Make se incluye en la mayoría de distribuciones de Linux, pero si desea utilizarlo sobre Windows, aquí tiene la versión de GNU para este sistema operativo.

10. Descripción del ejercicio

El ejercicio consiste en preparar un makefile para un sencillo proyecto (una calculadora que puede funcionar por línea de comandos o interactivamente) compuesto de los siguientes archivos

- calcular.c: funciones aritméticas comunes
- calcular.h: archivos de cabecera de las funciones comunes
- calc_interactive.c: programa para la calculadora interactiva
- calc_command.c: programa para la calculadora por línea de comandos

Revise el código fuente proporcionado para obtener más detalle. Piense en que pasaría si ejecutara el programa basado en comandos sin parámetros. La aplicación se compone de dos ejecutables, uno para cada tipo de calculadora. Debe preparar un makefile con las siguientes características:

- Los destinos incluirán al menos
  - **calc_interactive (o similar)**: ejecutable para la calculadora interactiva
  - **calc_command (o similar)**: ejecutable para la calculadora por línea de comandos
  - **clean**: eliminará archivos generados en la compilación
- Una sola llamada a make es suficiente para construir la aplicación al completo.
- No debe incluir ningún nombre de archivo en los destinos: use macros y sustituciones de texto. El nombre de cada archivo solo debe estar presente una vez en el makefile, de forma que modificarlo sea muy sencillo.
- Por defecto el compilador será 'gcc', pero se podrá definir una macro llamada SIMPLE al llamar a make para utilizar 'cc'.
- Por defecto se muestra la salida del compilador, pero si está definida SIMPLE se debe mostrar además un mensaje del tipo "El archivo 1 se ha creado a partir de los archivos: archivo 2, archivo 3, ..." .
- Por defecto los ejecutables se almacenan en el directorio actual, a no ser que se haya definido una macro llamada BINDIR con otra ruta relativa a éste.
- Make no termina en un error si se hace 'make clean' cuando no hay archivos que borrar (truco: el error de make está causado por un error en 'rm', puede buscar la opción adecuada de rm o modificar el makefile). No será necesario borrar el interior de los directorios que se hayan creado.
- Todo funciona perfectamente con un archivo de texto llamado 'clean' en el directorio.
- El makefile es lo más reducido posible

11. Ayuda para el ejercicio

Recuerde que:

- **gcc -c nombre.c** genera un archivo objeto con el mismo nombre que el archivo .c
- **gcc -o ejecutable f1.o f2.o** compila los archivos f1.o and f2.o en el ejecutable

Puede ser interesante utilizar una extensión (por ejemplo .x) para los ejecutables, a fin de reducir el tamaño del makefile definiendo reglas implícitas apropiadas.