USO DE MAKE CON CLIP

SACL a882sacl@yahoo.com.ar
Grupo Clip clip-castellano@gruposyahoo.com.ar
Versión 0.1 - 12/06/2005

Revisiones:

INTRODUCCION

Como ya sabemos, compilar un prg no presenta ningún problema desde la línea de comandos debemos hacer lo siguiente:

$ *clip -erM miprograma.prg -lmilib*

Pero que pasa si tenemos un proyecto con una multitud de *.prg y librerías. En teoría lo podríamos compilar "manualmente" desde el terminal, pero a un costo de tiempo y orden elevado. Para organizarnos mejor, podemos hacer uso de una utilidad muy antigüa en el mundo Unix, el "papá" de todas aquellas herramientas que hemos conocido en Windows para automatizar nuestras compilaciones y ensamblados. Estamos hablando del comando "make".

Los "impacientes" o ya versados en el tema, pueden saltarse toda esta "chachara" teórica" e ir directamente a la sección de su uso con Clip.

MAN MAKE (compendio del manual de make)

El propósito de la utilidad make es determinar automáticamente qué piezas de un programa necesitan ser recompiladas, y lanzar las órdenes para recompilarlas. Este manual describe la implementación de make del proyecto GNU, que ha sido escrita por el gran Richard Stallman y por Roland McGrath.

Nuestros ejemplos muestran programas en C, que es lo más común, pero se puede emplear make con cualquier lenguaje de programación cuyo compilador pueda ejecutarse con una orden del shell. De hecho, make no está limitado a compilar programas. Se puede usar para describir cualquier tarea donde algunos ficheros deban ser actualizados automáticamente a partir de otros en cualquier momento en que éstos cambien.

Para prepararnos a utilizar make, debemos escribir antes un fichero llamado el makefile que describe las relaciones entre los ficheros de nuestro programa, y las órdenes necesarias para actualizar cada fichero. En un programa, normalmente el fichero ejecutable se actualiza a partir de los ficheros o módulos objeto, los cuales a su vez se construyen mediante la compilación de los ficheros con el código fuente.

Una vez que exista un makefile apropiado, cada vez que cambiemos algún fichero fuente, esta simple orden:

...basta y sobra para que se realicen todas las recompilaciones necesarias. El programa make emplea los datos del makefile (y otros internos) y los tiempos de última modificación de los ficheros para decidir cuáles de ellos necesitan ser actualizados. Para cada uno de esos ficheros, lanza las órdenes que tiene grabadas en su base de datos.

Make ejecuta órdenes en el "makefile" para actualizar uno o más nombres de objetivo, donde nombre es típicamente un programa. Si no se ha dado la opción -f, make buscará los makefiles llamados GNUmakefile, makefile, y Makefile, en este orden, parando en el primero que encuentre y dando un error si no encuentra ninguno.

Normalmente deberíamos llamar a nuestro "makefile" o bien makefile o bien Makefile (Recomendamos Makefile porque normalmente aparece cerca del principio del listado de un directorio, al lado de otros ficheros importantes como LÉAME). El primer nombre que se busca, "GNUmakefile", no se recomienda para la mayoría de los makefiles. Solamente deberíamos emplear este nombre si tenemos un makefile que sea específico del make de GNU y no pueda ser leído y comprendido por otras versiones de make. Si makefile es `-', se lee de la entrada estándar.

El make actualiza un objetivo si éste depende de ficheros de prerrequisito (o dependencias) que hayan sido modificados desde la última modificación del objetivo, o si éste no existe.

FORMATO DEL ARCHIVO MAKEFILE

Un archivo Makefile es un archivo de texto en el cual se distinguen cuatro tipos básicos de declaraciones:

1) Comentarios.

2) Variables.

3) Reglas explícitas.

4) Reglas implícitas.

(1) Comentarios

Al igual que en los programas, contribuyen a un mejor entendimiento de las reglas definidas en el archivo. Los comentarios se inician con el caracter #, y se ignora todo lo que continúe después de ella, hasta el final de línea.

```
# Este es un comentario
```

(2) Variables

Se definen utilizando el siguiente formato:

```
nombre = dato
```

De esta forma, se simplifica el uso de los archivos Makefile. Para obtener el valor se emplea la variable encerrada entre paréntesis y con el caracter $ al inicio, en este caso todas las instancias de $(nombre) serán reemplazadas por dato. Por ejemplo, la siguiente definición

```
SRC = main.c
```

origina la siguiente línea:

```
gcc $(SRC)
```

y será interpretada como:

```
gcc main.c
```

Sin embargo, pueden contener más de un elemento dato. Por ejemplo:

```
objects = programa_1.o programa_2.o programa_3.o \
        programa_4.o programa_5.o

programa: $(objects)
        gcc -o programa $(objects)
```

Hay que notar que make hace distinción entre mayúsculas y minúsculas.

(3) Reglas explícitas

Estas le indican a make qué archivos dependen de otros archivos, así como los comandos requeridos para compilar un archivo en particular. Su formato es:

```
archivoDestino: archivosOrigen
     comandos  # Existe una caracter TAB (tabulador) antes de cada comando.
```

Esta regla indica que, para crear archivoDestino, make debe ejecutar comandos sobre los archivos archivosOrigen. Por ejemplo:

```
main: main.c funciones.h
    gcc -o main main.c funciones.h
```

significa que, para crear el archivo de destino main, deben existir los archivos main.c y funciones.h y que, para crearlo, debe ejecutar el comando:

```
gcc -o main main.c funciones.h
```

(4) Reglas implícitas

Son similares a las reglas explícitas, pero no indican los comandos a ejecutar, sino que make utiliza los sufijos (extensiones de los archivos) para determinar que comandos ejecutar. Por ejemplo:

funciones.o: funciones.c funciones.h

origina la siguiente línea:

$(CC) $(CFLAGS) -c funciones.c funciones.h

Existe un conjunto de variables que se emplean para las reglas implícitas, y existen dos categorías: aquellas que son nombres de programas (como CC) y aquellas que tienen los argumentos para los programas (como CFLAGS). Estas variables son provistas y contienen valores predeterminados, sin embargo, pueden ser modificadas, como se muestra a continuación:

```
CC = gcc
        CFLAGS = -Wall -O2
```

En el primer caso, se ha indicado que el compilador que se empleará es gcc y sus parámetros son -Wall -O2.

VARIABLES AUTOMATICAS

Se pueden utilizar algunos indicadores especiales que permiten hacer ficheros makefile más simples. Dentro de una regla, puedo utilizar cierto tipo de "variables especiales", que van a representar los elementos según su tipo y posición para esa regla determinada.

Sea :

Objetivo : requisito1 requisito 2 requisito3 requisitoN

Entonces:

$@
Representa al objetivo de la regla (lo que quiero obtener).

$^
Representa la lista completa de requisitos.

$?
Es la lista de requisitos "más recientes" que el objetivo.

$<
Es el primer requisito de la regla (requisito1).

MAKEFILE A USAR CON CLIP (Ejemplo general)

```
#---INICIO MAKE---
# Asumimos que tenemos un procedimiento PRINCIPAL de nombre principal.prg.
# En caso de que se llame de otra forma, reemplazar su denominación en la
```

```
# variable MAINPRG.

# Defino mis variables (macros). SOLO éstas debo retocar para cada caso particular.

MAINPRG = principal.prg
CLIP = $(CLIPROOT)/bin/clip        #Ubicación del compilador Clip
CLIPFLAGS_OBJ = -a -O              #Opciones para compilar los objs.
CLIPFLAGS_EXE = -e -r -M           #Opciones para compilar el ejecutable.
CLIPLIBS  = -lclip-mysql -lsup     #Las librerías que voy a ocupar (precedidas de una "l")

# Mis Objs, se excluye el obj del procedimiento principal. MUY IMPORTANTE, se generará al final.

OBJ = fuente1.o fuente2.o fuente3.o fuenteN.o


# Reglas explícitas. Por defecto se usa la etiqueta "all", que será
# la primera que se encuentra y la única que se ejecuta, junto a sus dependencias.
# MAINPRG contiene el procedimiento "principal".
# Para generar los obj, recurrirá a la regla ".prg.obj".


all: $(MAINPRG) $(OBJ)
        $(CLIP) $(CLIPFLAGS_EXE) $(MAINPRG) $(OBJ) $(CLIPLIBS)


# Defino las terminaciones a usar (sufijos) y... como se generan.

.SUFFIXES: .prg .o
.prg.o:
        $(CLIP) $(CLIPFLAGS_OBJ) $<


# Defino una regla "clean" que para su ejecución, desde la linea de comandos
# debo digitar : make clean
# En el fondo lo que hace es "eliminar" (rm) los archivos "basura" (temporales) que
# pudiesen haberse creado durante la compilación. Pueden crearse su(s) propia regla(s).
# -r : recursive   -> Borra todos los directorios dependientes si los hay.
# -f : force       -> Fuerza el borrado sin preguntar, aún si no existiesen.

clean:
      rm -rf  *.o *.c *.a *.so *.pa *.log  *.b  *.bak *~ core* *core

#----FIN MAKE----
```

Bueno, este fue sólo un resumen orientado al uso de "make" con Clip. Si necesitan más información pueden recurrir a los comandos "man make" e "info make".