Universidad Tecnológica Nacional
Facultad Regional Buenos Aires

Departamento de Electrónica

Asignatura: Informática I

Uso de la utilidad make

Armando proyectos

Autor: Ing. Alejandro Furfaro. Profesor Titular

Índice de contenido

Introducción

Cuando programamos proyectos mas o menos significativos normalmente utilizamos varios archivos fuentes, headers, referencias a librerías, y demás componentes.

Esto hace que en un punto se vuelva tediosa la tarea de compilar, linkear, y demás actividades.

Claro. Si alguna vez el lector ya se enfrentó con una bonita herramienta de programación con una amigable y simpática interfaz gráfica, posiblemente esté pensando que el autor no conoce de su existencia explicando cosas tan elementales. El autor ha trabajado con esas interfaces bonitas, simplistas y muy potentes en lo que hace a simplificarnos la vida.

¿Entonces para que leer esta guía?. Como en el caso de las anteriores: para aprender.

Un *IDE* (*Integrated Development Environmet*) que trabaja en modo gráfico (a veces llamados entornos visuales) es la octava maravilla del mundo para los programadores de aplicaciones, ya que nos quita de encima numerosas tareas. Pero esto no es una razón para que dejemos de entender como funcionan las cosas. Finalmente una de las rezones de ser de los ingenieros es entender el funcionamiento de las cosas. Y esto no es posible cuando desde el principio utilizamos herramientas que nos simplifican el trabajo ya que lo hacen a costa de esconder ciertas complejidades. El conocer estos detalles es una tarea ardua pero hace la diferencia en algunas ocasiones, aunque le doy la derecha al lector: suena muy antipático.

Luego de trabajar con estas herramientas puede continuar con el bonito IDE lleno de facilidades. Pero la diferencia fundamental será que entonces sabrá como las hace. Y cuando tenga un problema no se sentirá tan perdido como en otras ocasiones. (Al menos eso espero).

Make

make es una herramienta que permite ejecutar una secuencia de procesos. Utiliza un *script*, llamada comúnmente *makefile*.

Es capaz de determinar automáticamente cuales pasos de una secuencia deben repetirse debido al cambio en algunos de los archivos involucrados en la construcción de un objeto, o en una operación, y cuales no han registrado cambios desde la última vez de modo que no es necesario repetirlos.

Sus uso mas común es recompilar programas que residen en diversos archivos, pero también podemos automatizar el testing de estos programas, así como su empaquetado y distribución.

De manera sencilla lo que tenemos que hacer es definir un archivo llamado

Makefile en el directorio raíz de nuestro proyecto (en realidad lo podemos poner en otro lado y hacérselo saber a *make*) y dentro de ese archivo escribimos las reglas necesarias para construir nuestro proyecto.

Luego, alcanza con ejecutar

```
make <regla>
```

en el directorio en el que hemos definido el *Makefile*.

Parece simple.

La lógica de make

make funciona teniendo siempre presente las fases de desarrollo de un programa, cuyo proceso completo podemos ver en la Fig. 1.

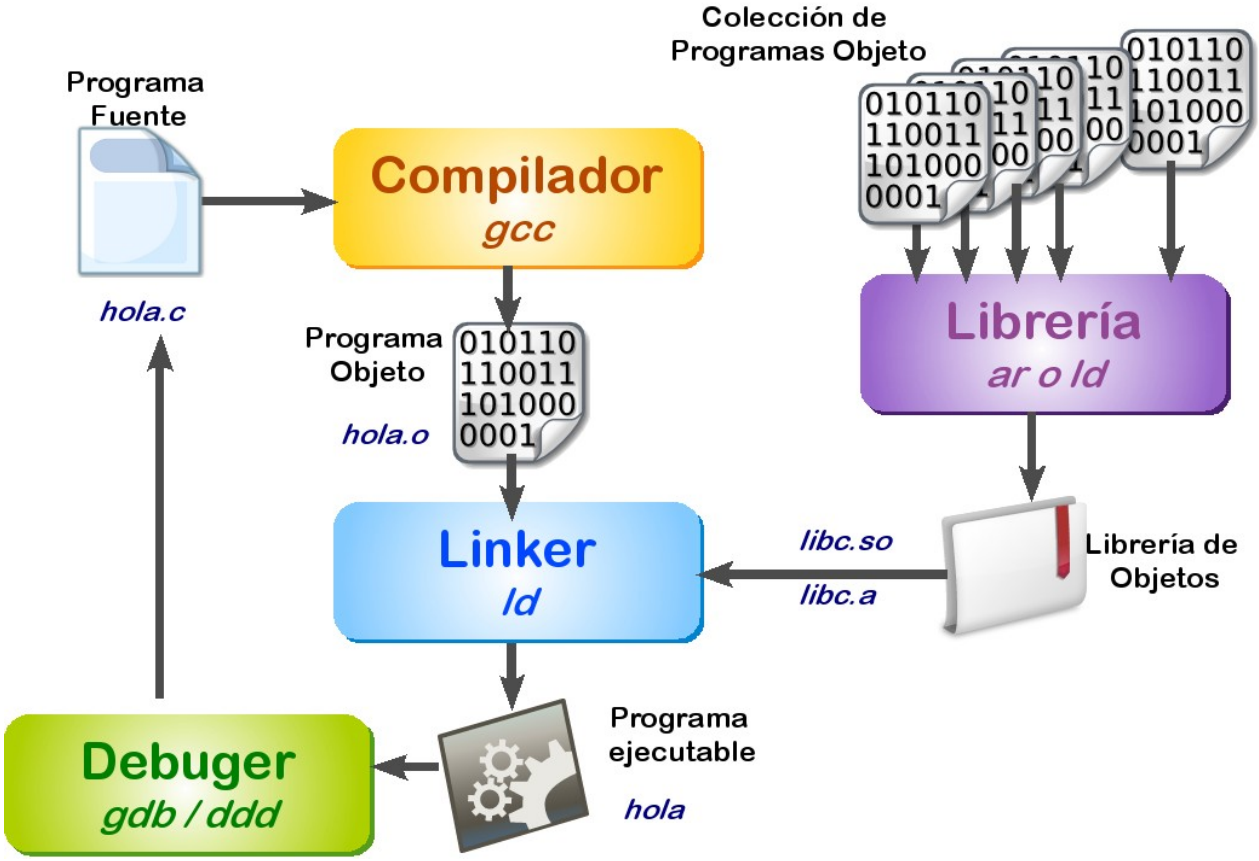

Fig. 1. Proceso de desarrollo

Este proceso genera una serie de dependencias. Cada programa objeto depende de su correspondiente programa fuente, y headers si los hubiere. Un ejecutable dependerá de los programas objeto involucrados en su proyecto, mas las librerías de uso en el mismo.

Esta situación se observa en la Figura 2.

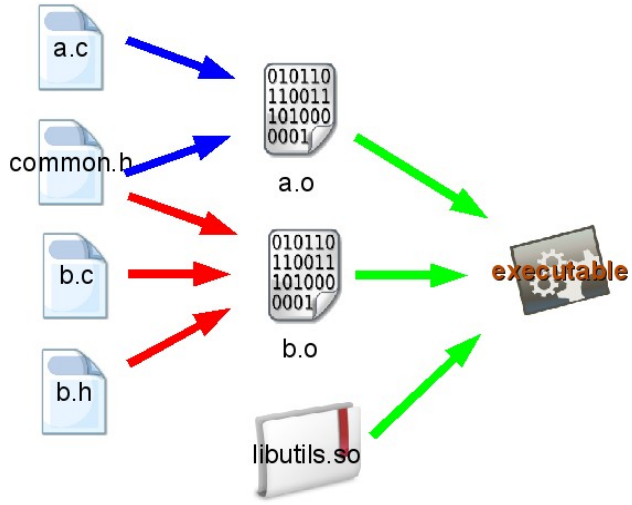

Fig.2. Dependencias en un proyecto.

En esta figura podemos ver las dependencias generadas en el desarrollo de un proyecto.

Tenemos tres grupos claramente diferenciados por los colores de las flechas.

A la hora de armar un *makefile* necesitamos tener claro el mapa de dependencias. Es como tener esta figura (adaptada a la realidad de nuestro proyecto) en la cabeza mientras escribimos los diferentes comandos en el *makefile*.

Es necesario respetar estas dependencias ya que *make* asume que el *script* escrito en el *makefile* ha tenido en cuenta este criterio.

Una vez armado y en funcionamiento, si modificamos el archivo *b.c*, no importa el motivo, *make* detectará la actualización pero en lugar de repetir todas las compilaciones y linkeos, solo recompilará al archivo *b.c*, lo cual modificará el objeto *b.o*, y por lo tanto deberá relinkear la aplicación para llegar a la versión de *executable* que contenga los cambios hechos en *b.c*. Lo interesante es que no tocó el resto. Es decir lo que no cambia se deja tal como está.

Pero veamos como es una primer versión de *makefile* para el caso de la Figura 2.

```
excecutable: a.o b.o
     gcc a.o b.o -o executable -lutils
a.o: a.c
     gcc -c -o a.o a.c
b.o: b.c
     gcc -c -o b.o b.c
clean:
     rm -f ./*.o
     rm -f executable
```

Lo que vemos al inicio de la línea como un nombre terminado con ':' se denomina *regla*.

Si no le especificamos nada y solo tipeamos *make* a secas, *make* asumirá que la regla a ejecutar es solo la de la primer línea con esta característica. En nuestro primer ejemplo *executable*.

Si la regla a continuación del carácter ':' tiene dependencias, éstas deberán corresponder dentro del *makefile* a otras reglas que se escriban a continuación de la regla dependiente.

En nuestro caso *excecutable* es dependiente de *a.o* y *b.o.*

Por lo tanto debe necesariamente existir una regla para *a.o* y otra para *b.o* escritas luego de la regla dependiente. Estas líneas están a continuación con sus respectivas dependencias.

Cada regla tiene a continuación el comando que necesita para su implementación.

Finalmente hemos escrito una regla que permita limpiar todos los archivos generados a partir de los fuentes. Esto puede resultar útil para hacer una recompilación general, especialmente al cierre del proyecto donde podemos eliminar ciertas opciones de compilación (como por ejemplo la opción -g) y por lo tanto se necesita reducir los tamaños de los ejecutables.

Si ejecutamos

```
make clean
```

se ejecuta esa regla pasando por alto las restantes, hasta resolver sus dependencias.

El formato general de cada regla es:

dependiente: dependencia

comando para generar el dependiente a partir de la dependencia.

En tutoriales varios y bibliografía que pueda consultar, puede encontrarse la siguiente terminología para lo mismo

target: fuentes (o sources)

comando (o command)

En cualquier caso es solo a los efectos de comprender la lógica de generación de cada línea.

Un detalle nada menor:

Las líneas que contienen comandos empiezan con un tabulador. De otro modo no funciona.

Variables

La versión anterior es la mas primitiva. El uso de variables permite por un lado resumir opciones, nombres y demás, y en especial facilitar los cambios cada vez que hay que retocar parámetros que se repiten a lo largo del *makefile*.

Comencemos preparando nuestro *makefile* para poder modificarlo de manera simple en caso de usar otro compilador (puede ser el *cc*, o el *gpp*)

```
CC=gcc
excecutable: a.o b.o
     $(CC) a.o b.o -o executable -lutils
a.o: a.c
     $(CC) -c -o a.o a.c
b.o: b.c
     $(CC) -c -o b.o b.c
clean:
     rm -f ./*.o
     rm -f executable
```

Vemos que la forma de usar variables se parece a la que usa el *shell*, salvo por los paréntesis que aquí se utilizan adicionalmente al carácter '$' para acceder al contenido.

Con este criterio, podemos afianzar mas el uso de variables

```
CC=gcc
CFLAGS=-c -g
LDFLAGS=-g -lutils
OBJS=a.o b.o
excecutable: $(OBJS)
     $(CC) $(OBJS) -o executable $(LDFLAGS)
a.o: a.c
     $(CC) $(CFLAGS) -o a.o a.c
b.o: b.c
     $(CC) $(CFLAGS) -o b.o b.c
clean:
     rm -f ./*.o
     rm -f executable
```

Macros

make posee algunas reglas sintácticas adicionales que pueden facilitar, aún mas la generalización de un *makefile*, aunque a costa de volverlo mas críptico.

```
CC=gcc
CFLAGS=-c -g
```

```
LDFLAGS=-g -lutils
OBJS=a.o b.o
excecutable: $(OBJS)
     $(CC) $(OBJS) -o $@ $(LDFLAGS)
a.o: a.c
     $(CC) $(CFLAGS) -o $@ $<
b.o: b.c
     $(CC) $(CFLAGS) -o $@ $<
clean:
     rm -f ./*.o
     rm -f executable
```

$@ es una macro que contiene para cada regla el valor del target (o dependiente). En el caso de la regla principal de nuestro *makefile $@=executable*. En la regla siguiente: *$@=a.o*, y en la siguiente *$@=b.o.*

$< por su parte se refiere en cada regla al fuente de la misma, siempre que haya un único fuente.

En este punto observemos el aspecto del *makefile*.

Es muy diferente del primero que habíamos escrito, es mas inteligible, pero mas flexible para su modificación.

Observemos las reglas de formación de *a.o* y *b.o*. Realmente son muy similares. De hecho con las macros que usamos hasta aquí la línea correspondiente al comando es la misma en ambos casos.

```
CC=gcc
CFLAGS=-c -g
LDFLAGS=-g -lutils
OBJS=a.o b.o
excecutable: $(OBJS)
     $(CC) $(OBJS) -o $@ $(LDFLAGS)
%.o: %.c Makefile
     $(CC) $(CFLAGS) -o $@ $<
clean:
     rm -f ./*.o
     rm -f executable
```

Aparece el carácter '%' que trabaja de *wildcard*, indicando el equivalente a "cualquier cosa". Entonces, la línea de la segunda regla se lee:

"Cualquier cosa" que termine en "*.o*" que necesite para construir *executable*, se construye a partir de la dependencia del mismo nombre terminada en "*.c*".

Además al incluir *Makefile* en esa línea, este *Makefile* hará que *make* detecte incluso si cambió el *Makefile* en cuyo caso repite todo el proceso completo.

Vamos logrando un *makefile* mas general. Ya no dependemos de los nombres de un fuente determinado. Esto significa que podemos agregar fuentes a nuestro proyecto, y los va a compilar. Solo deberíamos agregar el objeto en la variable *OBJS* para que se incluyan como dependencias en *executable*.

Hay una forma mas sofisticada que está en este listado. La opción *-MM* del *gcc*, hace que no se compilen los fuentes, pero el compilador lista las dependencias por *stdout*.

Lo podemos verificar en nuestro directorio de trabajo de la guía anterior de librerías. Recordemos primero su contenido

```
alejandro@notebook:~/InfoI/second$ ls -las
total 48
4 drwxr-xr-x 4 alejandro alejandro 4096 feb 19 18:53 .
4 drwxr-xr-x 6 alejandro alejandro 4096 feb 17 01:16 ..
8 -rwxr-xr-x 1 alejandro alejandro 6367 feb 16 18:02 demo
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 20 18:19 dynamic
4 -rw-r--r-- 1 alejandro alejandro  210 feb 16 11:30 holalib.c
4 -rw-r--r-- 1 alejandro alejandro  106 feb 16 07:58 holalib.h
4 -rw-r--r-- 1 alejandro alejandro  836 feb 16 15:47 holalib.o
4 -rw-r--r-- 1 alejandro alejandro  978 feb 16 15:47 libhola.a
4 drwxr-xr-x 2 alejandro alejandro 4096 feb 19 17:08 shared
4 -rw-r--r-- 1 alejandro alejandro  121 feb 14 12:32 test_holalib.c
4 -rw-r--r-- 1 alejandro alejandro  772 feb 16 15:47 test_holalib.o
alejandro@notebook:~/InfoI/second$
```

Utilicemos ahora la opción *-MM* para ver la salida.

```
alejandro@notebook:~/InfoI/second$ gcc -MM *.c
holalib.o: holalib.c holalib.h
test_holalib.o: test_holalib.c holalib.h
alejandro@notebook:~/InfoI/second$
```

Aplicamos esta facilidad en nuestro *makefile*

```
CC=gcc
CFLAGS=-c -g
LDFLAGS=-g -lutils
OBJS=a.o b.o
SOURCES=$(OBJS:.o=.c)
excecutable: $(OBJS)
     $(CC) $(OBJS) -o $@ $(LDFLAGS)
dependencias:
     $(CC) $(CFLAGS) -MM $(SOURCES) > $@
-include dependencias
clean:
     rm -f ./*.o
     rm -f executable
```

En primer lugar hemos definido una variable *SOURCES*. Para inicializarla hemos simplemente tomado la lista de valores incluidos en *OBJS*, a los que les hemos reemplazado el sufijo "*.o*" por "*.c*", utilizando los operadores "*:*" e "=”

Luego creamos una regla llamada *dependencias* que se ocupa de ejecutar al compilador con la opción *-MM*, tomando uno a uno los archivos que estén en la variable *SOURCES*, y envía línea por línea su salida a la variable *$@*, que corresponde a la regla dependencias.

De este modo a la salida del comando, la regla dependencia contiene la salida del compilador (que es del tipo de la listada anteriormente en nuestro directorio */InfoI/second*).

Inmediatamente debajo de esta regla se incluye dependencias, mediante la macro -include.

Esto pondrá la lista de reglas y dependencias listadas como salida.

Podemos ir en el makefile mas allá y por ejemplo crear una regla para la entrega del proyecto.

```
CC=gcc
CFLAGS=-c -g
LDFLAGS=-g -lutils
OBJS=a.o b.o
SOURCES=$(OBJS:.o=.c)
HEADERS=*.h
```

```
excecutable: $(OBJS)
     $(CC) $(OBJS) -o $@ $(LDFLAGS)
%.o: %.c Makefile
     $(CC) $(CFLAGS) -o $@ $<
clean:
     rm -f ./*.o
     rm -f executable
enviar:
     tar zcvf proyecto.tar.gz $(SOURCES) $(HEADERS) Makefile
```

Para ello se agregó tan solo una variable llamada *HEADERS* para poder incluir en el archivo de entrega a los *headers* sin los cuales no es compilable.

Al final se incluye una regla llamada entrega. Al terminar nuestro proyecto, conviene ejecutar

```
make clean
```

Luego quitar las opciones de compilación de debuging e incluir tal vez alguna opción de optimización (*-O0*, *-O1*, *-O2*, o *-O3*, consultar *man gcc* para ver detalles). Una vez testeado por última vez, se ejecuta

```
make enviar
```

El resultado es un archivo *proyecto.tar.gz* con todo el proyecto listo para enviar.

Conclusiones

Hemos aprendido a construir nuestro proyecto utilizando una herramienta sumamente potente. Tan potente que esta guía es solo una pequeñez comparada con las posibilidades y construcciones que permite hacer.

La experiencia de uso de *make* indica que una vez que tenemos un *makefile* mas o menos apto para nuestro proyecto, todo lo que hacemos es simplemente cambiar el nombre de las dependencias en la variable *OBJS*, y agregar o quitar *flags* para luego reutilizarlo.

Si en cambio nos topamos con un entorno gráfico de programación, ahora sabremos mas acerca de como personalizarlo y sacarle provecho. No se lo cuenten a nadie. Pero esos entornos llaman finalmente a *gcc* y *make*. ;)