GNU make

Juan Bertoni, Santiago Calligari, Leandro Spagnolo, Julián Cabrera

3 de agosto de 2020

Índice

1. ¿Qué es GNU make?

Es una herramienta de código abierto para gestionar dependencias y organizar la compilación de los archivos de un programa. Cuando se desea obtener un archivo ejecutable a partir de código fuente escrito en un único archivo, puede utilizarse para compilar directamente. Pero si el programa está constituido por múltiples archivos, make representa una manera automática de comenzar la compilación. En resumen, es una herramienta para automatizar procesos, también denominada "task runner". Para llevar a cabo estos procesos de manera automática, sigue una serie de instrucciones especificadas por el usuario a través de un archivo de texto denominado makefile. Make determina qué es lo que necesita ser recompilado en el caso de haber hecho cambios en el programa. Esta herramienta no está limitada a un único lenguaje de programación, ya que puede ser utilizada en cualquiera en el que las órdenes sean dadas mediante comandos.

2. Introducción

Crearemos un pequeño programa. Dada su sencillez, podría compilarse en un archivo único, pero con fines instructivos, lo realizamos de esta manera.

Usaremos los siguientes códigos, que pueden encontrarse en el apéndice:

- encabezado.h
- saludo.c

En lugar de utilizar gcc, usaremos make. Con el comando make saludo, podemos generar el archivo. En efecto, al ingresar make saludo, make detecta que existe un archivo llamado "saludo.c", y por ello lo compila. La extensión.c le indica a make que se encuentra escrito en lenguaje C. Una extensión .cpp hubiera indicado que era C++, y lo mismo para las demás extensiones.

3. Las ventajas de make

Supongamos que el archivo "saludo" no estuviera creado. Nuestro objetivo es generar el archivo ejecutable "saludo". Para ello, bastaría simplemente con ingresar:

gcc -o saludo saludo.c

En principio, esto parecería mostrar que aprender a utilizar make no tiene utilidad, ya que su función sería completamente reemplazable. *Pero no lo es.*

Ingresemos ese comando. Nuevamente, se compila el archivo. Y si ingresamos el comando de nuevo, y de nuevo, lo mismo sucederá.

Tipeemos ahora el comando make saludo. *No hace nada.* Al contrario. *Nos dice que el archivo ya está hecho.* Entonces, tiene sentido pensar qué sucedería si fuera eliminado. Para ello, ingresamos el comando rm saludo, y nuevamente

utilizamos make saludo. *Esta vez sí se compila.* Esto significa que make tiene la ventaja de no compilar cuando el archivo ya está creado.

Sin embargo, surge un nuevo problema. Supongamos que, una vez terminada la compilación, deseamos modificar el archivo saludo.c. Luego, debemos compilar nuevamente el programa, y obtendremos una versión actualizada del archivo ejecutable saludo... pero... como ya existe dicho archivo ejecutable, en principio, make no funcionaría (porque ya vimos que no compila si el archivo ya estaba creado). En otras palabras, el comando make saludo no funcionaría dado que make consideraría que el archivo saludo ya fue creado, e ignoraría todos los cambios que hubiéremos realizado en saludo.c.

Intentemos cambiarlo, a ver qué sucede. Escribamos saludo() dos veces en lugar de una (o sea que ahora el programa dirá "Buenos días" dos veces). Una vez hecho esto, ingresemos el comando make saludo. *Se compila nuevamente.*

¿Qué está sucediendo? Parece que make detectó la modificación que realizamos al archivo saludo.c. Cabe preguntarse... ¿cómo hizo eso?

Para ello, retomaremos un concepto anteriormente estudiado. Recordemos que los archivos tienen una marca de tiempo, que indica la última vez que fueron editados. Una manera de verla es con el comando ls. Por ejemplo, si ingresamos el comando ls -l saludo.c, este se listará en formato largo, y de la salida de dicho comando podemos obtener la marca de tiempo.

El truco es que *make compara las marcas de tiempo.* Cuando nosotros modificamos el archivo saludo.c, cambiamos su marca de tiempo, y esto fue lo que make detectó. Lo que hace make es *verificar si coinciden las marcas de tiempo del código fuente y el archivo ejecutable*; es decir, (en nuestro caso) comparar las marcas de tiempo de saludo.c (el código fuente a partir del cual generamos el programa) y de saludo (el archivo ejecutable en sí). Cuando nosotros habíamos compilado inicialmente el código (antes de agregar saludo() una vez más), ambos archivos tenían la misma marca de tiempo porque fueron modificados al mismo tiempo. Pero, una vez que modificamos saludo.c, las marcas de tiempo pasaron a diferir dado que la de dicho archivo se actualizó. Y es esta diferencia la que le indica a make que debe recompilar, porque el archivo ha sido editado.

En resumen, mediante la comparación de marcas de tiempo, make tiene la siguiente habilidad: *cuando nosotros editamos solo algunos archivos del código fuente, make solamente recompila lo necesario, pero no lo que permanece igual.* Esto representa un gran ahorro de tiempo. Para ello, supongamos que contamos con un programa inmenso que tarda varios minutos en compilarse en total. Si por cada pequeña modificación que quisiéramos realizar al código nos viéramos obligados a recompilar todo, el proceso se volvería sumamente tedioso, extenso e ineficiente.

Esta habilidad especial de make, combinada con el hecho de que (como ya veremos) provee modos prácticos para compilar programas de alta complejidad, hacen de make una herramienta completamente útil.

Nota: cabe preguntarse qué sucedería si editamos el archivo encabezado.h. El comando make saludo, en este caso, no recompila el programa. En efecto, en la forma en la que estamos usándolo, make solo compara las marcas de tiempo del código fuente (saludo.c) y el archivo ejecutable (saludo). Sin embargo,

modificar encabezado.h no hará que make recompile. Entonces, por ejemplo, si reemplazamos “Buenos días\n” por “Muy buenos días\n” en este archivo .h, al ingresar el comando make saludo, nos devolverá un mensaje diciendo que saludo está actualizado, y no se dará cuenta de los cambios hechos a encabezado.h. El ejecutable saludo sigue imprimiendo el mensaje “Buenos días” (sin la palabra “muy”). Más adelante veremos cómo arreglar este inconveniente.

4. Programa ejemplo

Usaremos como ejemplo un programa escrito en C que implementar una calculadora en notación polaca tomado del libro clásico de Kernighan y Ritchie, “El lenguaje de programación C”. Notación polaca. La notación polaca es una forma de indicar operandos y operaciones apta para ser implementada mediante un “stack” o pila. En las operaciones con dos operandos, como 5 + 3, se ingresan primero los operandos 5, 3 y luego la operación; el ingreso de la operación ya implica la ejecución y presentación del resultado. Como puede verse en los ejemplos, no necesita signo de igual ni paréntesis. Con un poco de práctica, esta notación es tan efectiva como la tradicional, requiere menos símbolos y es más fácil de implementar.

- Ejemplo 1: para calcular la expresión 10 - 8 se digita 10 8 - 4 5 + *
- Ejemplo 2: para calcular la expresión (10 - 8) * (4 + 5) se digita 10 8 - 4 5 + *

Este programa es simple; puede implementarse en un archivo único, pero lo usaremos de ejemplo de un programa grande repartiendo el código en varios archivos de código c (extensión .c) y archivos de encabezados para las declaraciones (extensión .h). Todos estos archivos se encuentran en el apéndice

El código está integrado por los archivos:

- calc.h
- getch.c
- getop.c
- main.c
- stack.c

5. Compilación con make

A contiuación se muestra un makefile elemental, seguido de la instrucciones a ejecutar desde a línea de comandos para poder compilar. Luego, se explicará cómo funciona el proceso.

Cree un archivo makefile con el siguiente contenido:

```
# makefile para calculadora polaca, versión 1
# usar tabulador (no espacios) en la línea de comando
polaca : main.o stack.o getop.o getch.o
	gcc -o polaca main.o stack.o getop.o getch.o
main.o: main.c calc.h
	gcc -c main.c
stack.o: stack.c calc.h
	gcc -c stack.c
getop.o: getop.c calc.h
	gcc -c getop.c
getch.o: getch.c
	gcc -c getch.c
clean:
	rm polaca \
		main.o stack.o getop.o getch.o
```

Compile el programa dando el comando
make
Observe los mensajes indicativos de las tareas realizadas. Digite luego
./polaca
para verificar su funcionamiento. Digitando
make clean
se borra el ejecutable y los archivos objeto.

6. Reglas de make

Las reglas que definimos en el makefile deben tener un formato determinado:

```
destino : requisito
	comando a ejecutar
	...
```

- Destino: puede ser el nombre de un archivo a crear o el de una tarea a realizar.
- Requisito: es el nombre de un archivo del cual depende el destino.
- Comando: es la acción que el usuario quiere realizar.

En cuanto a la sintaxis, se puede organizar mejor la lectura del código colocando el caracter de continuación "\" para que los comandos estén en la misma línea. Por ejemplo, en el makefile utilizado, es como si rm polaca main.o stack.o getop.o getch.o estuviera en una única línea. Los renglones iniciados con # son comentarios, no se ejecutan. El resto de las líneas son reglas. Cada regla empieza con un nombre seguido de ":". Las líneas iniciadas con tabulador son continuación de la misma regla. Volvemos a remarcar que cada línea que sea

una continuación de otras (aquellas donde el usuario escribe los comandos, en nuestro ejemplo) debe comenzar con un tabulador (no con espacios).

Un destino suele depender de varios archivos requisito. Cuando el destino es el nombre de una tarea no hay requisitos. La creación de un destino puede requerir varios comandos. Nótese también que cada comando es un comando normal, que puede darse en la línea de comandos del shell, y debe ejecutar correctamente, pues make no sabe nada de él.

Vemos ahora un ejemplo. Considerese esta regla del archivo makefile antes mostrado:

```
getop.o: getop.c calc.h
        gcc -c getop.c
```

Esto significa que el destino es el archivo getop.o, los requisitos son getop.c y calc.h, y el comando a ingresar es gcc -c getop.c. En otras palabras, make ejecuta el comando gcc -c getop.o, y getop.c y calc.h son los requisitos porque ambos son requeridos para realizar la compilación (getop.c es el código fuente, y calc.h es un archivo .h necesario, que está declarado con include en getop.c)

En la regla comenzada por polaca:, se utilizan los archivos ejecutables .o para generar un único ejecutable (polaca) mediante el comando gcc -o. En este caso, polaca es el destino, los archivos .o son los requisitos, y gcc -o es el comando.

7. Uso de variables

La siguiente versión de makefile muestra el uso de variables:

```
# makefile para calculadora polaca, versión 2
# uso de variables
# usar tabulador (no espacios) en la línea de comando
CC = gcc
OBJECTS = stack.o getop.o getch.o
polaca : main.o $(OBJECTS)
        $(CC) -o polaca main.o $(OBJECTS)

main.o: main.c calc.h
        $(CC) -c main.c
stack.o: stack.c calc.h
        $(CC) -c stack.c
getop.o: getop.c calc.h
        $(CC) -c getop.c
getch.o: getch.c
        $(CC) -c getch.c
clean:
        rm polaca
                main.o stack.o getop.o getch.o
```

En esta versión resulta fácil cambiar el compilador, o agregar un nuevo nombre de archivo objeto. Existen nombres de variables comprendidas por make, para usos específicos, como CC en el ejemplo; otras son costumbre, como OBJECTS; también pueden crearse otras a gusto del programador.

Entonces, por ejemplo, para cambiar el compilador a g++, basta con reemplazar la instrucción (CC)=gcc por (CC)=g++.

8. Reglas predefinidas

make dispone de algunas reglas predefinidas. En particular, conoce el comando para compilar objetos desde C. Esto hace innecesario escribir los comandos en las reglas que crean los objetos, en tanto respondan a la manera usual de hacerlo.

```
# makefile para calculadora polaca, versión 3
# uso de variables; reglas predefinidas
CC = gcc
OBJECTS = stack.o getop.o getch.o
polaca : main.o $(OBJECTS)

$(CC) -o polaca main.o $(OBJECTS)

main.o: calc.h
stack.o: calc.h
getop.o: calc.h
getch.o:
.PHONY : clean
clean :
	-rm polaca $(objects)
```

Escribir la regla para clean de esta forma evita confusiones con un posible archivo llamado “clean”, y también evita la detención de make ante errores del comando rm, por ejemplo si no se encuentran los archivos a borrar, situación normal si no fueron creados. Esta es la función que cumple declarar una regla como “.PHONY”, lo cual puede hacerse escribiendo .PHONY: (nombre del destino) antes de escribir la regla.

Es posible escribir reglas predefinidas específicas para cada makefile:

```
# CC = gcc
.SUFFIXES: .o .cpp
```

.cpp.o:
$(CC) -c $<

La línea .SUFFIXES declara los sufijos a reconocer. Al ingresar la regla .cpp .o, se explicita cómo obtener un archivo .o partir de uno .cpp. La macro $<representa un prerrequisito. Esto quiere decir que asumirá el valor que sea necesario para la compilación. Si ingresamos esta regla con .SUFFIXES, no hay necesidad de especificar comandos cuando ingresemos destino, comando y requisitos de una regla (siempre que esta regla esté referida a la obtención de un .o a partir de un .cpp). Por ejemplo: archivo.o: archivo.cpp gcc

-c archivo.cpp En este caso, se puede omitir gcc -c archivo.cpp (el comando),

porque ya le estamos indicando a make cómo obtener un archivo .o a partir de un .cpp con ./SUFFIXES. El comando a ejeutar es siempre $(CC) -c $<. Como la macro $<asumirá el valor archivo.cpp (y siendo que CC asume el valor gcc) entonces este comando es gcc -c archivo.cpp, con lo cual no es necesario indicarlo.

9. La variable CFLAGS

Supongamos que estamos escribiendo el código de un archivo .c. Lo primero que hacemos, en general, es escribir #include <stdio.h>, y también agregamos otras librerías de ser necesario.

Sin embargo, podría ser que nosotros deseáramos incluir otros archivos de extensión .h que sean de nuestra propia creación. En ese caso, podemos apelar a algo "similar" a la opción -I de gcc.

Si nuestros archivos se encontraran dispersos, por ejemplo, en los directorios /home/usuario y /home/usuario/dir1, nosotros debemos guardar en la variable CFLAGS el contenido -I /home/usuario -I /home/usuario/dir1. Podemos hacer eso con las líneas de comandos:

CFLAGS="-I /home/usuario -I /home/usuario/dir1"; export CFLAGS

Entonces, en tal caso, cuando compilemos el programa con make, no se generarán errores por no encontrar los archivos .h dado que, por medio de la variable CFLAGS, le estamos indicando a make dónde buscarlos.

Otro ejemplo podría ser el caso en el que deseemos utilizar archivos .h que se ubiquen en el directorio en el que estamos compilando, y en un subdirectorio del mismo, llamado subdir (por ejemplo). En tal caso, si deseamos podemos usar una sola vez -I como ya se vio con gcc:

CFLAGS="-I ../subdir"; export CFLAGS

Nótese que, como ya se vio al estudiar gcc, para la búsqueda en el directorio del código fuente se puede colocar entre comillas el nombre del archivo .h. Entonces, si nosotros escribimos #include "header.h" en las primeras líneas

de nuestro programa, la diferencia con #include <header.h>es que, si header.h se encuentra en el mismo directorio que el código fuente, este será encontrado incluso aunque no esté especificado en CFLAGS.

Lo mismo puede hacerse con la opción -L, para indicar los directorios de búsqueda de librerías (como con gcc). Entonces, por ejemplo, tomando CFLAGS="-L /usr/lib", le indicamos al compilador que busque las librerías en este directorio. También es posible escribir -L múltiples veces para buscar en múltiples directorios; al igual que con -I.

Como es posible ver, la opción -I en CFLAGS funciona de igual manera que en gcc. De hecho, lo mismo sucede con otras opciones mostradas para gcc. Podemos incorporar -g a la variable CFLAGS para poder depurar nuestro programa. Esto puede hacerse definiendo:

CFLAGS=-g

Podemos incorporar múltiples opciones; por ejemplo:

CFLAGS="-I. -g"

Esta opción no solo nos permite debugguear, sino que también añade e directorio en el que estamos compilando a la lista de directorios de búsqueda de nuestros archivos .h. Sin embargo, cuando usamos make con un makefile,

CFLAGS no tiene efecto, porque le indicamos a make qué comandos seguir (por ejemplo gcc -c saludo.c ignora CFLAGS porque es un comando con gcc). Pero sí es posible indicar explícitaente que se sigan las instrucciones dictadas por CFLAGS:

gcc -c $(CFLAGS) saludo.c

De este modo, se ejecuta el comando gcc con las opciones que se encuentren escritas en el contenido de CFLAGS. Por ejemplo, si CFLAGS=-I, se lee:

gcc -c -I saludo.c

Por ello, el valor de CFLAGS sí puede influir en la compilación con make, si se indica explícitamente que lo haga.

10. Opciones de make

El comando make acepta entre otras estas opciones:

-n

muestra comandos a ejecutar, sin ejecutarlos

-f archivo

indica el nombre del archivo donde se encuentran las reglas, si no se llama "makefile".

11. Resumen

La compilación de un ejecutable a partir de programas en C y sus archivos de encabezado puede simplificarse creando un archivo makefile siguiendo los

modelos mostrados. Para compilar usando un archivo llamado makefile basta dar el comando

make

Si sólo quieren verse los comandos a ejecutar, digitar

make -n

Los comandos a ejecutar dependen del estado de actualización de los archivos; si no han habido cambios desde la última creación del archivo objeto la regla no se ejecuta.

make -f reglas.make

ejecuta make tomando las reglas del archivo reglas.make.

12. Apéndice

12.1. encabezado.h

```
#include<stdio.h>
void saludo(void){
printf("Buenos días\n");
}
```

12.2. saludo.c

```
#include<stdio.h>
#include"encabezado.h"
int main(){
saludo();
return 0;
}
```

12.3. calc.h

```
/* calculadora polaca inversa: calc.h */

#define NUMERO '0'/* señal para detección de número */
void push(double); /* extrae de la pila */
double pop(void); /* coloca en la pila */
int getop(char[]);
int getch(void);
void ungetch(int);
```

12.4. getch.c

```
/* calculadora polaca inversa: getch.c */
```

```
/* para reunir los caracteres que forman un número se leen dígitos
hasta el punto decimal o caracter no numérico, pero entonces ya
se ha leído un caracter de más.
getch() lee un caracter,
ungetch() lo devuelve a la entrada para ser releído.*/

#include <stdio.h>
#define BUFFER 100 /* largo del arreglo para buffer */

static char buf[BUFFER]; /* buffer de caracteres */
static int pbuf = 0; /* posición en el buffer */

/* getch: obtiene un caracter, acaso devuelto al buffer */
int getch(void)
{
if (pbuf >0) /* si el buffer no está vacío... */
return buf[–pbuf]; /* devuelve caracter del buffer. */
else /* si el buffer está vacío... */
return getchar(); /* lee nuevo caracter de la entrada */
}

/* ungetch: devuelve caracter a la entrada */
void ungetch(int c)
{
if (pbuf >= BUFFER)
printf("ungetch: demasiados caracteres, se agotó el buffer\n");
else
buf[pbuf++] = c;
}
```

12.5. getop.c

```
/* calculadora polaca inversa: getop.c */

#include <stdio.h>
#include <ctype.h>
#include "calc.h"

int getch(void);
void ungetch(int);

/* getop: obtiene el siguiente operador u operando numérico */
```

```
int getop(char s[])
{
int i, c;

while ((s[0] = c = getch()) == ' '|| c == '\t')
; /* proposición vacía */

s[1] = '\0';
if (!isdigit(c) && c != '.')
return c; /* no es un número */
i = 0;
if (isdigit(c)) /* reúne la parte entera */
while(isdigit(s[++i] = c = getch()))
; /* proposición vacía */
if (c == '.') /* reúne la parte fraccionaria */
while (isdigit(s[++i] = c = getch()))
; /* proposición vacía */
s[i] = '\0';
if (c != EOF)
ungetch(c);
return NUMERO;
}
```

12.6. main.c

```
/* calculadora polaca inversa */

/* pide ingreso de expresiones en notación polaca inversa hasta
obtener un caracter EOF (fin de archivo, Ctrl-D).
Ejemplo de notación polaca:
la expresión
(10 - 8) * (4 + 5)
se escribe
10 8 - 4 5 + *
*/

#include <stdio.h>
#include <stdlib.h>         /* para atof() */
#include "calc.h"

#define MAXOP 100           /* máxima cantidad operandos y operadores */
#define NUMERO '0'          /* señal para detección de número */

int getop(char[]);
void push(double);
```

```
double pop(void);

/* programa principal */
main()
{
int tipo;
double op2;
char s[MAXOP];

printf("\nCALCULADORA POLACA.\n");
printf("Ingrese expresiones en notación polaca inversa.\n");
printf("Para finalizar, digite Ctrl-D.\n");

while((tipo = getop(s)) != EOF) {
switch (tipo) {
case NUMERO:
push(atof(s));
break;
case '+':
push(pop() + pop());
break;
case '*':
push(pop() * pop());
break;
case '-':
op2 = pop(); /* extrae el sustraendo */
push(pop() - op2); /* extrae el minuendo y
resta */
break;
case '/':
op2 = pop(); /* extrae el divisor */
if (op2 != 0.0)
push(pop() / op2); /* extrae el dividendo y divide */
else
printf("ERROR: divisor nulo\n");
break;
case '\n':
printf("Resultado:\t %.8g\n", pop());
break;
default:
printf("ERROR: comando desconocido %s\n", s);
break;
}
}
return 0;
```

```
}
```

12.7. stack.c

```
/* calculadora polaca inversa: stack.c */

#include <stdio.h>
#include “calc.h”
#define MAXVAL 100 /* máxima cantidad de valores en la pila */

static int pp = 0; /* posición en la pila */
static double val[MAXVAL];

/* push: coloca f en la pila */
void push(double f)
{
if (pp <MAXVAL)
val[pp++] = f;
else {
printf(“ERROR: pila llena, no se puede agregar %g\n”, f);
exit (1);
}
}

/* pop: extrae de la pila, devuelve el valor de tope */
double pop(void)
{
if(pp >0)
return val[–pp];
else {
printf(“ERROR: pila vacía, no se puede extraer\n”);
return 0.0;
}
}
```