ENTORNO DE DESARROLLO DE GNU (gcc y make)

1. COMPILACIÓN C EN LINUX DESDE LÍNEA DE COMANDOS: GCC Y MAKE

Las herramientas de este apartado se utilizan desde la línea de comandos. Por ello, todas se ejecutarán desde una terminal de Linux. En este apartado veremos cuáles son los comandos que se utilizan para compilar un programa desde una terminal o desde la línea de comandos y para automatizar tareas relacionadas con la creación de programas. Básicamente son dos los programas que se introducirán: el compilador de GNU, *gcc*, y el programa de *scripting* *make*.

1.1. El Compilador de C Creado por GNU: gcc

El entorno de compilación de GNU, GCC, consiste en un conjunto de herramientas que permiten la compilación de programas escritos en diversos lenguajes. En particular, nos centraremos en el compilador/enlazador de programas C: **gcc**.

El programa gcc funciona principalmente como compilador/enlazador. Esto es, compila y enlaza los archivos de código fuente que se les pase para dar lugar a un determinado ejecutable. Por ejemplo, para compilar y enlazar el código de un supuesto proyecto *basico1*, bastaría con escribir, en el directorio en el que se encuentra *main.c*:

```
$  gcc main.c –o basico1
```

que compilaría *main.c* y generaría el ejecutable *basico1*. La opción –o indica cuál queremos que sea el nombre del ejecutable generado por gcc, y éste se puede poner junto al parámetro –o o aparte. De no indicar ningún nombre, el archivo ejecutable generado por defecto se denominará *a.out*.

Si quisiéramos ahora ejecutar *basico1*, habría que escribir:

```
$  ./basico1
```

Ahora bien, ¿se puede separar el proceso de compilación del de enlazado? Sí. Para ello, cuando se utilice gcc se debe añadir el parámetro –c, que indica que gcc solo actúe como compilador:

```
$  gcc –c main.c
$  gcc –c funciones.c
```

Estos dos comandos generarían respectivamente *main.o* y *funciones.o* que son los archivos objeto resultado del proceso de compilación.

Ahora para enlazarlos podemos utilizar nuevamente gcc, ahora sin parámetros (supongamos que queremos llamar calculadora a nuestro ejecutable):

```
$  gcc –ocalculadora  main.o funciones.o
```

O utilizar directamente el enlazador de gnu, ld:

```
$  ld –ocalculadora  main.o funciones.o
```

Estas son algunas opciones destacables de gcc:

`-Wall:` (warning all) muestra por pantalla todos los warning o avisos especificados por el compilador

`-ansi:` No permite el uso de funciones no pertenecientes al estándar ANSI C

`-c:` compilación

`-o:` necesaria para establecer el nombre del ejecutable salida, por defecto a.out

`-g:` incluye en el ejecutable binario información necesaria para la depuración

`-help:` muestra la ayuda

`-l:` especifica librerías adicionales que van a ser enlazadas

`-O:` especifica niveles de optimización de código

`-I:` especifica directorios adicionales para búsqueda de archivos de cabeceras

`-L:` especifica directorios adicionales para búsqueda de librerías

1.2. Uso de *make* para Automatizar Tareas de Compilación

Make es una utilidad usada para automatizar la ejecución de tareas por medio del uso de un script, llamado *makefile*. En éste se indican los comandos usados, en nuestro caso para compilación, así como otras posibles tareas de mantenimiento (borrado o movimiento de ficheros, etc.).

La estructura más general de un fichero de script *makefile* es como sigue:

```
# Declaraciones de macros

MACRO1=valor

#Declaraciones de reglas.

objetivo1:  dependencia1 dependencia2 dependenciaN

       comando1 dependencia1 dependencia2

       comando2 dependencia2 dependenciaN  $(MACRO1)

dependencia1: archivo1 archivo2

       comando3  archivo1 archivo2
```

Básicamente un fichero makefile se compone de dos partes: una parte de declaración de macros globales (“constantes globales”) y una parte de definición de reglas u objetivos.

Parte de definición de reglas

La parte de definición de reglas consiste en una secuencia de entradas formadas por un objetivo, una serie de dependencias (que pueden ser referencias a otros objetivos), y un conjunto de comandos que “dicen” cómo conseguir el objetivo, y que son comandos de Linux (tal como gcc, cp, ls, echo, etc.). Usualmente, cada objetivo tiene correspondencia con el nombre de un archivo a producir (aunque, excepcionalmente puede existir alguna regla que no se refiera a la producción de un fichero sino a la realización de una acción). En general, el objetivo será un archivo intermedio, como puede ser un archivo objeto (.o),

resultado de una compilación antes del enlazado; o ser el archivo ejecutable que resulta de la traducción a código máquina del proyecto.

Para ilustrar estos conceptos, veamos un ejemplo de archivo *makefile*:

```
# Objetivo para enlazar los archivos del proyecto Basico1
basico1: main.o
      gcc -obasico1 main.o

# Objetivo para compilar el archivo main.c del proyecto Basico1
main.o: main.c
      gcc -c main.c

# Objetivo para eliminar todos los ficheros generados
clean:
      rm -f main.o
```

En este ejemplo se han incluido tres reglas. Las dos primeras tienen el propósito de enlazar y compilar el código del proyecto basico1, respectivamente. La última de eliminar los archivos generados por la compilación ("clean" el proyecto).

Para que la primera regla se ejecute, debe existir el fichero *main.o*, o existir alguna otra regla que indique cómo construirlo. Asimismo, esta regla asume que se generará un fichero llamado *basico1*, el ejecutable, pero solo en el caso de que éste no exista y que ninguna de sus dependencias tenga una fecha de modificación posterior a la fecha de creación del objetivo, en este caso basico1. Para ello, ***make* buscará un fichero en el directorio actual que se llame igual que el objetivo correspondiente, si lo encuentra, comprobará la fecha y la existencia de las dependencias para saber si debe o no recompilar**, o compilar por primera vez el objetivo (esto es, ejecutar el conjunto de comandos, situado en las líneas siguientes a la cabecera de la regla, que estén indentadas con un tabulador).

Veamos cuál sería el orden de ejecución, si tuviésemos el archivo con el código de *main.c*, junto al *makefile* anterior. Para ejecutar el script tenemos que ejecutar el siguiente comando:

```
$ make basico1
```

Donde estamos indicando que se ejecute *make*, que éste busque un fichero que se llame *makefile* o *Makefile* o *MAKEFILE*, y ejecute los comandos necesarios, solo si es preciso (el fichero no está o está desactualizado), para conseguir el objetivo *basico1*. Si no se indicara el objetivo, se ejecutaría el primer objetivo que se encuentre presente en el archivo *Makefile*.

En este caso, y teniendo en cuenta la configuración previa, *make* buscaría basico1, como no estaría (solo tenemos dos ficheros: *main.c* y *makefile* en el directorio actual), comprobaría si todas las dependencias de ejecución de la regla están. En este caso, buscaría el fichero *main.o*. Como tampoco se encuentra en el directorio, buscaría una regla cuyo objetivo sea *main.o*. Esta es la segunda regla. Entonces comprobaría las dependencias de esta regla. En este caso, *main.c*. Como este fichero existe, *make* ejecutaría el comando asociado a dicha regla, esto es, *gcc –c main.c*, que produce la compilación de *main.c* en *main.o*, el fichero con código objeto. Una vez producido se

retorna a la primera regla, y se ejecutan los comandos asociados, ya que las dependencias están satisfechas (solo hay una, *main.o* y se acaba de generar). Entonces, se ejecutaría el comando que enlaza los archivos del proyecto (en este caso solo uno).

La última regla, *clean*, no tiene dependencias, lo que significa, que se ejecutará incondicionalmente cuando se le indique desde la línea de comandos (siempre que no haya un fichero que se denomine *clean*). De hecho, es típico incluir esta regla de ejecución incondicional de forma que se encargue de la eliminación de los archivos intermedios producidos durante el proceso de compilación del ejecutable final.

Parte de definición de macros o variables globales

Por otra parte, se ha mencionado que en los archivos makefile también se puede incluir una parte relativa a definición de macros o variables globales. El uso más normal de macros con makefile es de tres tipos:

- Macros globales definidas previamente a la utilización de una regla. Estas variables tienen más un carácter de alias definidos al principio de un script. Esto es, asociaciones entre un nombre (la variable) y un valor de tipo textual. Estas variables se pueden expandir (obtener su contenido) poniendo entre paréntesis el nombre y anteponiendo un símbolo $. Por ejemplo: se puede definir una variable llamada CC al principio del programa que contenga el nombre del compilador, *gcc*, y luego usar esta variable para llamar al compilador en cualquier parte del script. Así, si en el futuro deseamos cambiar el compilador que estamos usando, no tendremos que cambiar la referencia a *gcc*, si no solo el valor de la variable.

  Ejemplo de uso de una macro global:

```
CC = gcc
ejecutable: modulo1.o modulo2.o modulo3.o
      $(CC) -oejecutable modulo1.o modulo2.o modulo3.o
```

- Macros locales haciendo referencia a partes de una regla. Estas variables se usan en el ámbito de los comandos de una regla. Son atajos para referenciar dependencias o el nombre del objetivo de forma genérica. Algunas de las variables más útiles son:

Macro local	Valor o Acción
$^	Se sustituye por todas las dependencias de una regla
$<	Se sustituye por la primera dependencia de una regla
$@	Se sustituye por el objetivo de una regla

  La utilidad de estas macros estriba en poder definir reglas tales como:

```
ejecutable: modulo1.o modulo2.o modulo3.o
       gcc -o$@ $^
```

  cuyo comando se expandiría a:

```
gcc -oejecutable modulo1.o modulo2.o modulo3.o
```

- Macros pasadas por el usuario que llama a *make*. Estas variables se referencian igual que las variables locales (símbolo $ delante del nombre de la variable entre

paréntesis). Tienen prioridad sobre las macros definidas en (pueden sobreescribir el valor de) un archivo makefile. Tienen la peculiaridad de que no se definen en el script, si no que se definen al llamar a *make*. Un ejemplo de uso puede ser:

```
$  make basico1 CC=cc "CFLAGS= –Wall –ansi"
```

que sobreescribe o define dos macros: CC y CFLAGS.

El siguiente ejemplo de fichero *makefile* compilaría un proyecto calculadora, el cual consta de un programa principal: *calculadora.c* y las bibliotecas func_n_reales y fun_n_complejos donde se definen e implementan funciones matemáticas con números reales y complejos, respectivamente. Suponiendo que las librerías anteriores están completamente definidas a partir de los ficheros *func_n_reales.c*, *func_n_reales.h* , *fun_n_complejos.c*, y *fun_n_complejos.h*, y que ambas bibliotecas son necesarias para la ejecución del programa principal *calculadora.c*. Este *makefile* automatiza el proceso de compilación generando un ejecutable de nombre *'calculadora'* e incluye una regla 'clean' que borra tanto el ejecutable como los archivos objeto generados en el proceso.

```
calculadora: calculadora.o func_n_reales.o fun_n_complejos.o
        gcc -ocalculadora calculadora.o func_n_reales.o fun_n_complejos.o

calculadora.o: calculadora.c func_n_reales.h fun_n_complejos.h
        gcc -c calculadora.c

funciones.o: func_n_reales.c func_n_reales.h
        gcc -c func_n_reales.c

complejo.o: fun_n_complejos.c fun_n_complejos.h
        gcc -c fun_n_complejos.c

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

Enlaces de interés sobre *make*

http://www.gnu.org/software/make/manual/make.html **en GNU page**.