GNU cc (I)

GNU cc (*gcc*) es el compilador del Proyecto GNU. Compila programas escritos en C, C++, Objective C, e incluso fortran (a través de ***g77***). Próximamente estarán disponibles Pascal, Modula-2, Ada 9x y otros muchos más lenguajes de desarrollo.

gcc permite al programador el control total del proceso de compilación, compuesto de cuatro etapas:

- Preprocesado
- Compilación
- Ensamblado
- Enlazado

Se puede parar el proceso después de cada etapa para ir examinando los resultados. Acepta los dialectos del lenguaje C (ANSI-C y K&R), así como C++ y Objective C. Puede incorporar en el código binario información para el depurado, así como realizar diversas optimizaciones de código. Además es un *compilador cruzado*, o sea, se puede desarrollar código en un procesador para se ejecutado en otro con distinta arquitectura. Por último, **gcc** posee extensiones de los lenguajes C y C++ que mejoran la optimización y facilitan el trabajo, a cambio de perder portabilidad (cosa poco recomendable).

Un tutorial corto

Para coger familiaridad, empecemos con el típico ejemplo en C:

```
1   /*
2   hello.c - Mi primer programa de C
3    */
4   #include <stdio.h>
5
6   int main(void)
7   {
8         fprintf(stdout, "¡Hola, Mad Hawk!\n");
9         return 0;
10  }
```

Para compilar y ejecutar este programa:

```
$ gcc hello.c –o Hello
$ ./hello
¡Hola, Mad Hawk!
```

La primera orden le dice al **gcc** que compile y enlace el fichero fuente hello.c, generando el ejecutable indicado por el argumento -o hello. La segunda orden ejecuta el programa, obteniéndose como resultado la tercera línea.

Varias cosas han pasado sin darnos cuenta. Lo primero que ha hecho **gcc** es manda al preprocesador (**cpp**) el fichero hello.c para expandir macros e incluir el contenido de los ficheros #include. Seguido, se compila el código preprocesado en código objeto, para finalmente el linkador (**ld**) crea el hello binario.

Veamos este proceso paso a paso. Si usamos la opción -E paramos el proceso justo después de realizarse la precompilación:

```
$ gcc –E hello.c –o hello.i
```

Podemos observar en el fichero hello.i el resultado del preprocesado, con el fichero stdio.h incrustado dentro de él. Continuando con el proceso, usaremos la opción -c para parar despúes de realizar la compilación.

```
$ gcc –c hello.i –o hello.o
```


madhawk@linuxmail.org

Ya tan sólo queda enlazar el fichero objeto con las librerías del sistema para generar el ejecutable:
$ gcc hello.o –o hello
Para saber el punto de arranqu **gcc** se fija en la extensión del fichero de entrada.. Las extensiones más comunes reconocidas por **gcc** son:

Extensión	Tipo
.C	Código fuente Lenguaje C
.C,.cc	Código fuente Lenguaje C++
.i	Código fuente C Preprocesado
.ii	Código fuente C++ Preprocesado
.S,.s	Código fuente Lenguaje Ensamblador
.o	Código objeto compilado
.a,.so	Código de librería compilado

Aunque lo normal es crear un ejecutable a partir del código fuente, a veces resulta conveniente parar el proceso en un punto determinado. Un caso habitual es cuando queremos obtener el código objeto para integrarlo en una librería, por lo que no es necesario la etapa de enlace. Otro caso cuando queremos obtener el código del precompilado para chequear algún conflicto que tengamos entre los ficheros includes y nuestro código.
La mayor parte de los programas de C se generan partiendo de varios fuentes, códigos objeto y librerías. g**cc** tomará los ficheros que le damos de entrada, los procesará convenientemente, y los enlaza para generar un único ejecutable. Dar todos y cada uno de los ficheros que componen una aplicación puede resultar demasiado tedioso. Para facilitar esta tarea se suele utilizar la utilidad make que veremos más adelante.

Opciones comunes de la línea de comandos
La lista de opciones ocuparía varias páginas, por lo que sólo veremos las opciones más comunes:

Opción	Descripción
-o <file>	Nombre del fichero de salida. Si no se especifica al generar el ejecutable, el nombre por defecto será a.out
-c	Compilar sin enlazar
-DMAC=VAL	Define el macro MAC y le asigna el valor VAL
-IDIR	Sitúa el directorio DIR delante de la lista de directorios dónde buscar los ficheros includes.
-LDIR	Sitúa el directorio DIR delante de la lista de directorios dónde buscar las librerías.
-static	Enlaza con librerías estáticas. Por defecto enlaza las librerías dinámicamente.
-lFICH	Enlaza con la librería libFICH
-g	Incluye información estándar para depurado
-ggdb	Incluye información específica para el depurador gdb
-O	Optimiza el código compilado
-ON	Especifica un nivel de optimación del código entre 0 y 3
-ansi	Soporta el C ANSI/ISO estándar
-pedantic	Emite todos los avisos requeridos por el C ANSI/ISO
-pedantic-errors	Emite todos los errores requeridos por el C ANSI/ISO
-traditional	Soporta la sintáxis Kernighan & Ritchie del lenguaje C
-w	Suprime todos los mensajes de aviso (una mala idea)
-Wall	Emite todos los avisos que el gcc pueda generar.
-werror	Convierte todos los avisos en errores, lo que parará la compilación
-MM	Crea una lista de dependencias compatible con make
-v	Muestra los comandos utilizados en cada etapa de la compilación

GNU cc (II)

GCC es un compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr.
La sigla GCC significa "GNU Compiler Collection". Originalmente significaba "GNU C Compiler"; todavía se usa GCC para designar una compilación en C. G++ refiere a una compilación en C++.
Sintaxis.

```
gcc [ opción | archivo ] ...
g++ [ opción | archivo ] ...
```

Las opciones van precedidas de un guión, como es habitual en UNIX, pero las opciones en sí pueden tener varias letras; no pueden agruparse varias opciones tras un mismo guión. Algunas opciones requieren después un nombre de archivo o directorio, otras no. Finalmente, pueden darse varios nombres de archivo a incluir en el proceso de compilación.
Ejemplos.

```
gcc hola.c
```

compila el programa en C hola.c, genera un archivo ejecutable a.out.

```
gcc -o hola hola.c
```

compila el programa en C hola.c, genera un archivo ejecutable hola.

```
g++ -o hola hola.cpp
```

compila el programa en C++ hola.c, genera un archivo ejecutable hola.
gcc -c hola.c
no genera el ejecutable, sino el código objeto, en el archivo hola.o. Si no s indica un nombre para el archivo objeto, usa el nombre del archivo en C y le cambia la extensión por .o.

```
gcc -c -o objeto.o hola.c
```

genera el código objeto indicando el nombre de archivo.

```
g++ -c hola.cpp
```

igual para un programa en C++.

```
g++ -o ~/bin/hola hola.cpp
```

genera el ejecutable hola en el subdirectorio bin del directorio propio del usuario.

```
g++ -L/lib -L/usr/lib hola.cpp
```

indica dos directorios donde han de buscarse bibliotecas. La opción -L debe repetirse para cada directorio de búsqueda de bibliotecas.

```
g++ -I/usr/include hola.cpp
```

indica un directorio para buscar archivos de encabezado (de extensión .h).
Sufijos en nombres de archivo.
Son habituales las siguientes extensiones o sufijos de los nombres de archivo:

.c	fuente en C
.C .cc .cpp .c++ .cp .cxx	fuente en C++; se recomienda .cpp
.m	fuente en Objective-C
.i	C preprocesado
.ii	C++ preprocesado
.s	fuente en lenguaje ensamblador
.o	código objeto
.h	archivo para preprocesador (encabezados), no suele figurar en la linea de comando de gcc

Opciones.

- c	realiza preprocesamiento y compilación, obteniendo el archivo en código objeto; no realiza el enlazado.

- E	realiza solamente el preprocesamiento, enviando el resultado a la salida estándar.
-o *archivo*	indica el nombre del archivo de salida, cualesquiera sean las etapas cumplidas.
-I*ruta*	especifica la ruta hacia el directorio donde se encuentran los archivos marcados para incluir en el programa fuente. No lleva espacio entre la I y la ruta, así: -I/usr/include
-L	especifica la ruta hacia el directorio donde se encuentran los archivos de biblioteca con el código objeto de las funciones referenciadas en el programa fuente. No lleva espacio entre la L y la ruta, así: -L/usr/lib
-Wall	muestra todos los mensajes de error y advertencia del compilador, incluso algunos cuestionables pero en definitiva fáciles de evitar escribiendo el código con cuidado.
-G	incluye en el ejecutable generado la información necesaria para poder rastrear los errores usando un depurador, tal como GDB (GNU Debugger).
-V	muestra los comandos ejecutados en cada etapa de compilación y la versión del compilador. Es un informe muy detallado.

Etapas de compilación.

El proceso de compilación involucra cuatro etapas sucesivas: preprocesamiento, compilación, ensamblado y enlazado. Para pasar de un programa fuente escrito por un humano a un archivo ejecutable es necesario realizar estas cuatro etapas en forma sucesiva. Los comandos gcc y g++ son capaces de realizar todo el proceso de una sola vez.

1. Preprocesado.

En esta etapa se interpretan las directivas al preprocesador. Entre otras cosas, las variables inicializadas con #define son sustituidas en el código por su valor en todos los lugares donde aparece su nombre.

Usaremos como ejemplo este sencillo programa de prueba, circulo.cpp:

```
/* Circulo.c: calcula el área de un círculo.
   Ejemplo para mostrar etapas de compilación.
*/
#define PI 3.1416
main()
{
  float area, radio;
  radio = 10;
  area = PI * (radio * radio);
  printf("Circulo.\n");
  printf("%s%f\n\n", "Area de circulo radio 10: ", area);
}
```

El preprocesado puede pedirse con cualquiera de los siguientes comandos; cpp alude específicamente al preprocesador.

```
 $ gcc -E circulo.c > circulo.pp
 $ cpp circulo.c > circulo.pp
```

Examinando circulo.pp

```
 $ more circulo.pp
```

puede verse que la variable PI ha sido sustituída por su valor, 3.1416, tal como había sido fijado en la sentencia #define.

2. Compilación.
La compilación transforma el código C en el lenguaje ensamblador propio del procesador de nuestra máquina.

```
$ gcc -S circulo.c
```

realiza las dos primeras etapas creando el archivo circulo.s; examinándolo con

```
$ more circulo.s
```

puede verse el programa en lenguaje ensamblador.

3. Ensamblado.
El ensamblado transforma el programa escrito en lenguaje ensamblador a código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador.
El ensamblador se denomina as:

```
$ as -o circulo.o circulo.s
```

crea el archivo en código objeto circulo.o a partir del archivo en lenguaje ensamblador circulo.s. No es frecuente realizar sólo el ensamblado; lo usual es realizar todas las etapas anteriores hasta obtener el código objeto así:

```
$ gcc -c circulo.c
```

donde se crea el archivo circulo.o a partir de circulo.c. Puede verificarse el tipo de archivo usando el comando

```
$ file circulo.o
circulo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1, not stripped
```

En los programas extensos, donde se escriben muchos archivos fuente en código C, es muy frecuente usar gcc o g++ con la opción -c para compilar cada archivo fuente por separado, y luego enlazar todos los módulos objeto creados. Estas operaciones se automatizan colocándolas en un archivo llamado makefile, interpretable por el comando make, quien se ocupa de realizar las actualizaciones mínimas necesarias toda vez que se modifica alguna porción de código en cualquiera de los archivos fuente.

4. Enlazado
Las funciones de C/C++ incluídas en nuestro código, tal como printf() en el ejemplo, se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. Es preciso incorporar de algún modo el código binario de estas funciones a nuestro ejecutable. En esto consiste la etapa de enlace, donde se reúnen uno o más módulos en código objeto con el código existente en las bibliotecas.
El enlazador se denomina ld. El comando para enlazar

```
$ ld -o circulo circulo.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 08048184
```

da este error por falta de referencias. Es necesario escribir algo como

```
$ ld -o circulo /usr/lib/gcc-lib/i386-linux/2.95.2/collect2 -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o circulo /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc-lib/i386-linux/2.95.2/crtbegin.o -L/usr/lib/gcc-lib/i386-linux/2.95.2 circulo.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-linux/2.95.2/crtend.o /usr/lib/crtn.o
```

para obtener un ejecutable.
El uso directo del enlazador ld es muy poco frecuente. En su lugar suele proveerse a gcc los códigos objeto directamente:

```
$ gcc -o circulo circulo.o
```

crea el ejecutable circulo, que invocado por su nombre

```
$ ./circulo
Circulo.
Area de circulo radio 10: 314.160004
```

da el resultado mostrado.
Todo en un solo paso.

En programa con un único archivo fuente todo el proceso anterior puede hacerse en un solo paso:

```
$ gcc -o circulo circulo.c
```

No se crea el archivo circulo.o; el código objeto intermedio se crea y destruye sin verlo el operador, pero el programa ejecutable aparece allí y funciona.
Es instructivo usar la opción -v de gcc para obtener un informe detallado de todos los pasos de compilación:

```
$ gcc -v -o circulo circulo.c
```

Enlace dinámico y estático.
Existen dos modos de realizar el enlace:
- estático: los binarios de las funciones se incorporan al código binario de nuestro ejecutable.
- dinámico: el código de las funciones permanece en la biblioteca; nuestro ejecutable cargará en memoria la biblioteca y ejecutará la parte de código correspondiente en el momento de correr el programa.

El enlazado dinámico permite crear un ejecutable más chico, pero requiere disponible el acceso a las bibliotecas en el momento de correr el programa. El enlazado estático crea un programa autónomo, pero al precio de agrandar el tamaño del ejecutable binario.
Ejemplo de enlazado estático:

```
$ gcc -static -o circulo circulo.c
$ ls -l circulo
-rwxr-xr-x    1 hacker   hacker     237321 ago  4 11:24 circulo
```

Si no se especifica -static el enlazado es dinámico por defecto.
Ejemplo de enlazado dinámico:

```
$ gcc -o circulo circulo.c
$ ls -l circulo
-rwxr-xr-x    1 hacker   hacker      4828 ago  4 11:26 circulo
```

Notar la diferencia en tamaño del ejecutable compilado estática o dinámicamente. Los valores pueden diferir en algo de los mostrados; dependen de la plataforma y la versión del compilador.
El comando ldd muestra las dependencias de bibliotecas compartidas que tiene un ejecutable:

```
$ gcc -o circulo circulo.c
$ ldd circulo
      libc.so.6 => /lib/libc.so.6 (0x40017000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ gcc -static -o circulo circulo.c
$ ldd circulo
      statically linked (ELF)
```

La compilación estática no muestra ninguna dependencia de biblioteca.

Resumen.
Para producir un ejecutable con fuente de un solo archivo:

```
$ gcc -o circulo circulo.c
```

Para crear un módulo objeto, con el mismo nombre del fuente y extensión .o:

```
$ gcc -c circulo.c
```

Para enlazar un módulos objeto:

```
$ gcc -o circulo circulo.o
```

Para enlazar los módulos objeto verde.o, azul.o, rojo.o, ya compilados separadamente, en el archivo ejecutable colores:

```
$ gcc -o colores verde.o azul.o rojo.o
```