18-3-2020

Lisbeth Martínez Velázquez

ing.mecatrónica 8-at/m prof:CARLOS gARABITO



Comandos gcc y gdb

Programación de sistemas de embebidos

**Comando gcc**

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, gener un archivo ejecutable a.out.

*gcc -o hola hola.c*

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

*g++ -o hola hola.cpp*

compila el programa en C++ hola.c, gener 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:

Opciones.

**- c**

realiza preprocesamiento y compilación, obteniento 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.

**-Iruta**

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 sustituídas en el código por su valor en todos los lugares donde aparece su nombre.

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

/\* 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 victor victor 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 victor victor 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*

**Comando gdb**

Gdb es un depurador desarrollado por la FSF (Free Software Foundation) que se usa principalmente en Linux y bajo la licencia publica gratis GPL (GNU Public Licence). Este documento no tiene como objetivo el remplazar el manual oficial de gdb sino servir como una pequeña introducción en Español a su uso básico.

**1.El objetivo de un depurador**

Un depurador como gdb tiene como objetivo el permitirle al usuario ver lo que esta ocurriendo en su programa durante su ejecución o lo que ocurrió antes de que el programa parara de ejecutar a razón de un error (bug).

gdb puede hacer varios tipos de cosas para lograr estos objetivos:

● Empezar a correr el programa, especificar cualquier opción requerida por este.

● Detener la ejecución del programa basado en una linea (breakpoint), condiciones especificas (watchpoint) o excepciones (catchpoint) .

● Mostrar los valores de variables o estructuras durante la ejecución del programa (tracepoint).

● Examinar que ha pasado cuando el programa se ha detenido.

● Cambiar cosas en el programa para determinar por ejemplo si arreglan las fallas determinadas con el depurador .

Como se puede ver el depurador (debugger) es una herramienta fundamental en el desarrollo y corrección de fallas. El gdb puede ser utilizado para depurar (debug) fallas en varios lenguajes incluyendo: C, C++, y Fortran.

El gdb es un depurador de texto, todos sus comandos son instrucciones textuales. Un frontend gráfico que funciona con gdb es ddd. Aprender a utilizar ddd es simple una vez conocidos los comandos para el funcionamiento de gdb ya que todas la capacidades ddd están basadas directamente en comandos y las capacidades de gdb.

**2.Comandos esenciales de gdb**

Hay varios comandos esenciales para poder utilizar gdb, los comandos incluidos aquí forman parte de un set minimo para poder utilizarlo.

**2.1Empezar la ejecución de nuestro programa bajo el contro de gdb**

Para invocar a un programa que se ejecute bajo gdb:

*gdb programa*

También se puede invocar para estudiar un core file del programa:

*gdb programa core*

Otra forma de invocar gdb para depurar un proceso actualmente corriendo:

*gdb programa PID*

PID es el process ID del programa que esta ejecutando actualmente y que se quiere depurar. Un PID es un numero en Linux y Unix que se obtiene típicamente con el comando: ps -ef.

También se le pueden pasar argumentos al programa a depurar:

*gdb –args programa argumento1 argumento2 ... argumento N*

Para obtener ayuda sobre los comandos a pasarle a gdb:

*gdb -help*

**2.2Para terminar la ejecución de gdb**

Para terminar la ejecución de gdb se puede usar:

*quit o q*

**2.3Para compilar un programa para ser depurado**

Es necesario incluir la '-g' bandera para que el compilador incluya la información referente a los símbolos y el código fuente en el archivo objeto de el programa. Después esta información es incluida en el programa ejecutable. El siguiente es un ejemplo de un make file que incluye esta bandera:

*# Un makefile para crear un ejecutable con símbolos*

*OBJECTS = array.o*

*CFLAGS = -g*

*EXE = array*

*all: $(EXE)*

*$(EXE): $(OBJECTS)*

*g++ $(CFLAGS) -o $(EXE) $(OBJECTS)*

*array.o: array.cc*

*g++ $(CFLAGS) -c -o array.o array.cc*

*clean:*

*rm $(EXE) $(OBJECTS)*

Para hacer un make clean y borrar el ejecutable y los objetos indeseados invocar usando:

*make -f array.mak clean*

Para hacer un make y crear el archivo objeto (array.o) y el ejecutable (array):

*make -f array.mak*

**2.4Para empezar a correr el programa a depurar**

Para empezar a correr el programa deseado simplemente usar:

*run o r*

Se le pueden pasar argumentos al programa a ejecutar:

*r argumento1 argumento2 ... argumento N*

También se puede modificar el ambiente de ejecución, el directorio de trabajo y el standard input y output a usar. Para estas funciones favor de revisar la documentación oficial de gdb.

Para cambiar y ver los argumentos usados al ejecutar el programa a depurar:

*set args*

*show args*

**2.5El ambiente (environment) del programa a depurar**

El ambiente del programa a depurar es heredado directamente del ambiente en el cual se ejecuto gdb. Esto incluye las variables de Unix/Linux PATH, LIBPATH etc.

Para ver y cambiar estas variables del ambiente se puede usar:

*show environment [varname]*

*set environment varname [= value]*

Para eliminar una variable del ambiente

*unset environment varname*

**2.6Para pegarse a un programa ya ejecutando**

Si se desea depurar un programa (que tiene un process ID o PID) que ya esta ejecutando entonces dentro de gdb se puede usar el comando attach de la siguiente forma:

*attach PID*

Para desconectar gdb del programa se usa el comando detach:

*detach*

**2.7Para instalar y ver un breakpoint**

Hay muchas maneras diferentes para instalar un breakpoint una vez que se esta depurando un programa. Las usadas mas frequentemente son dadas aquí.

Para parar al comienzo de una función se usa el nombre de la función:

*break función*

Para parar en una linea especifica de una función:

*break función:linea*

Para ver los breakpoints, catchpoints o watchpoints instalados:

*info breakpoints*

**2.8Para controlar el flujo de ejecución**

Hay variados comandos para controlar el flujo de ejecución del programa siendo examinado.

Para ejecutar una o mas lineas de código fuente y entrar a cualquier función que sea llamada:

*step [cuenta]*

Para ejecutar una o mas lineas de código fuente y no entrar a cualquier función que sea llamada:

*next [cuenta]*

**2.9Para examinar memoria y variables**

Se usa el comando examine (x) para examinar la memoria. Hay muchos formatos para ver la memoria a examinar. El comando es el siguiente:

*x /nfu addr*

* n es la cuenta de unidades de memoria a mostrar.
* f es el formato a usar para mostrar la memoria: s (string), x (hex), i (machine).
* u es el tamaño a usar: b (bytes), h (halfwords = 2 bytes), w (words = 4 bytes), g (giant words = 8 bytes).

**Bibliografía:**

http://mmc.geofisica.unam.mx/femp/CyC++/DebuggingAndOptimization/CompilaryDepurarConHerramientasGNU.pdf

https://www.fing.edu.uy/tecnoinf/mvd/cursos/eda/material/otros/compilador\_GCC.pdf

http://profesores.elo.utfsm.cl/~tarredondo/info/datos-algoritmos/ELO-320%20gdb%20Info.pdf