Introducción al GCC y Makefile

Ing Florencia Ferrigno

2016

UTN - FRBA

Un poco de historia

- El creador del **GCC** (GNU C Compiler) fue **Richard Stallman**, fundador del proyecto **GNU**.
- El proyecto GNU se inicio en 1984 para crear un sistema operativo similar a UNIX pero libre, de código abierto.
- Cualquier sistema operativo basado en UNIX necesita de un compilador en C, sin embargo en ese tiempo no existían los compiladores de software libre.
- En 1987 se hace la primer entrega del **GCC**. Y compilaba únicamente en C. Con el paso del tiempo se fue extendiendo para darle soporte a otros lenguajes como por ejemplo C++. Y ahora su anacrónimo es GNU Compiler Collection.

Que es el GCC?

- Es compilador portable, es decir que se ejecuta en la mayoría de las plataformas que hereden de UNIX
- Está escrito en C y puede compilarse a sí mismo!
- El GCC es capaz de compilar cualquier programa en lenguaje C escrito en un archivo de texto, por ejemplo el **kate**, o versiones de consola como el **vim**

Qué es el proceso de compilar?

- Consiste en convertir los archivos de código fuente a código de máquina, una secuencia de unos y ceros.
- Esta secuencia será almacenada en un archivo **ejecutable**
- El proceso para llegar al ejecutable consta de cuatro pasos: **preprocesamiento, compilación, ensamblado y linkeo**.
- Vamos a tomar a modo de ejemplo el clásico "hola mundo" y aprovecharemos para ver algunas buenas prácticas de programación.

Archivo fuente

HolaMundo.c

```
/*
Todo lo que este entre estas barras y * sera
considerado un comentario
*/
#include <stdio.h>
int main()
{
   printf("Hello, World!\n");
   return 0;
}
```

Preprocesamiento

- Es la primer etapa de todo el proceso
- El preprocesador se encarga de:
  - Eliminar los comentarios
  - Interpretar y procesar las directivas de preprocesamiento. Las mismas son las que SIEMPRE están precedidas por #
- Cada vez que el preprocesador encuentra una directiva del tipo: **#include** reemplazará esa línea por el contenido de dicho archivo.
- Las directivas **#define** (que establecen constantes simbólicas) también serán sustituidas por el valor literal.

Preprocesamiento

- Para poder apreciar el trabajo del preprocesador vamos a llamar a la aplicación **gcc** de la siguiente manera:

gcc HolaMundo.c -E -oHolaMundo.i

- La opción -**E** que indica detener el proceso de compilación luego de realizado el preprocesamiento. Los archivos que no requieran de preprocesamiento serán ignorados
- La opción -**o** especifica el archivo de salida. Por convención la salida del preprocesador se la define con **.i** como parte del nombre del archivo.

IMPORTANTE: Recuerden que en Linux no existen las extensiones, todos los *"punto algo"* que se especifican son **sufijos** que se usan por convención

Compilación

- Una vez que tenemos el archivo preprocesado, podemos avanzar un paso más e intentar compilarlo.
- El compilador será quien analice la semántica y la sintaxis del código preprocesado y lo traducirá a un archivo objeto en lenguaje ensamblador
- Para realizar este paso:

gcc -S HolaMundo.i -oHolaMundo.s

- La opción -**S** es para detener la etapa de compilación y no realizar la etapa de ensamblado. La salida será en la forma de código assembler
- El nombre del archivo de salida, por convención será con el **.s** al final.

Ensamblado

- En esta etapa se traduce el archivo en lenguaje ensamblador a código binario

gcc -c HolaMundo.s -oHolaMundo.o

- La opción **-c** es para detener en la etapa de ensamblado y no realizar la etapa de linkeo.
- Con la opción **-o** indicamos como queremos que se llame el archivo de salida. Por convención se le agrega **.o**
- Las referencias o llamadas a funciones no se resuelven en esta etapa sino en la siguiente etapa. Por ejemplo no puede saber por que valor numérico reemplazar a *printf()*

Linkeo

- En esta etapa se resuelven las referencias a objetos externos ya sea en forma de archivos objetos o bibliotecas, como por ejemplo las llamadas a **printf** o **scanf**, los cuales se encuentran en la biblioteca standand del C.
- Nos permite incorporar de algún modo el código binario de estas funciones a nuestro ejecutable
- Quien se encarga del linkeo es el programa **ld**, a quien deberíamos decirle que objetos queremos enlazar. Su llamada sería algo así:

ld -o HolaMundo HolaMundo.o -lc

Al ejecutar dicho comando veremos que nos tira un error del siguiente tipo:

ld: warning: cannot find entry symbol _start

Da este error por falta de referencias.

Linkeo

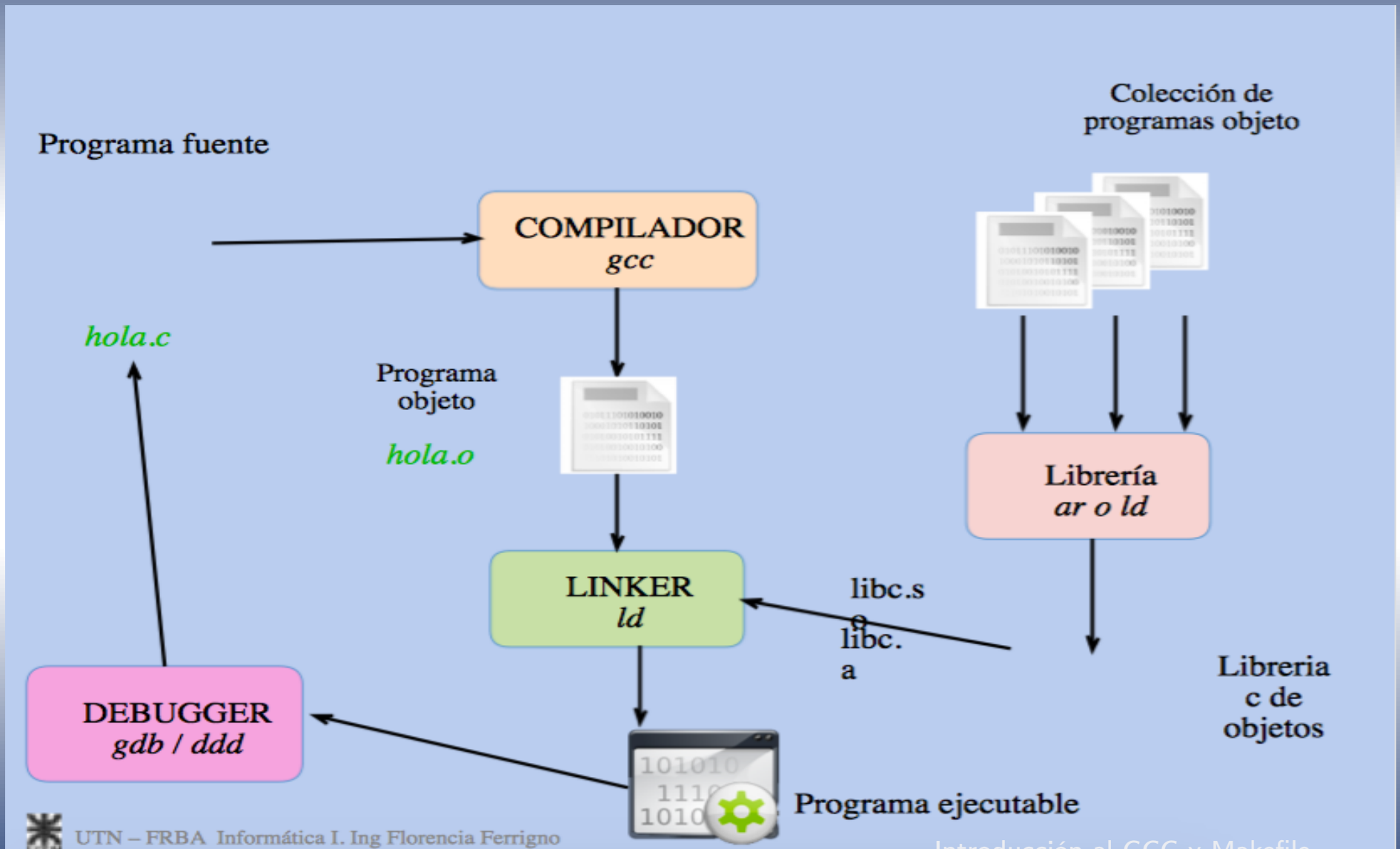
- Es necesario escribir lo siguiente:
  ld -o HolaMundo /usr/lib/gcc-lib/i386-linux/2.95.2/collect2 -m elf_i386 dynamic-linker /lib/ld-linux.so.2 -o HolaMundo /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc-lib/i386-linux/2.95.2/crtbegin.o -L/usr/lib/gcc-lib/i386-linux/2.95.2 HolaMundo.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-linux/2.95.2/crtend.o /usr/lib/crtn.o

Linkeo

- Es necesario escribir lo siguiente:
  ld -o HolaMundo /usr/lib/gcc-lib/i386-linux/2.95.2/collect2 -m elf_i386 dynamic-linker /lib/ld-linux.so.2 -o HolaMundo /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 HolaMundo.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-linux/2.95.2/crtend.o /usr/lib/crtn.o
- Para tranquilidad de todos, gcc resuelve estas llamadas por lo que no es usual llamarlo explícitamente.

  gcc -oHolaMundo HolaMundo.o
- Aquellas bibliotecas que no se linkean de manera automática se pueden linkear a través de la opción **-l**.

Proceso de compilación



Resumen de opciones del gcc

- **-E**: realiza solamente el preprocesamiento
- **-S**: realiza solamente la compilación
- **-c**: realiza preprocesamiento, compilación y ensamblado
- **-l**: especifica una librería adicional que se usará en el proceso de linkeo. Por convención el nombre de las librerías comienza con lib... Por ejemplo: libm.so.6, sin embargo cuando queremos hacer un link con esta librería lo que vamos a hacer será: **-lm**
- **-o**: especifica el nombre del archivo de salida. Recordar las convenciones para los nombres:

.h	Archivo header
.c	Archivo fuente
.i	Archivo preprocesado
.s	Archivo en lenguaje ensamblador
.o	Archivo objeto

Resumen de opciones del gcc

- **-Wall**: Esta opción viene de *Warning all*, nos sirve para poder ver todas las advertencias (warnings)
- **-g**: incluye en el ejecutable inforamción necesaria para poder rastrear los errores mediante el uso de un debugger como por ejemplo el KDBG.
- **-D**: permite hacer defines al momento de la compilación. Por ejemplo queremos definir la cantidad de vueltas que va a dar una iteración:

gcc -oejecutable -DITER=30 main.c

- **-I**: especifica la ruta adicional del directorio de headers. Por ejemplo: **-I/usr/include**
- **-v**: muestra los comandos ejecutados en cada etapa de compilación.
- **-L**: especifica la ruta adicional del directorio donde se encuentran las liberías. Por ejemplo: **-L/usr/lib**

Que es el make?

- El **make** es una utilidad o servicio que determina qué componentes de un proyecto necesitan ser recompilados y por lo tanto ejecuta los comandos necesarios para hacerlo. El **make** necesita un archivo **Makefile** que le indique qué hacer.
- A medida que se van haciendo modificaciones al programa, **make** permite reconstruir el programa con un único y breve comando.
- **make** acelera el proceso de compilación, minimizando los tiempos porque es lo suficientemente inteligente para determinar qué archivos han cambiado y sólo reconstruye dichos archivos.

Makefile

- Es un archivo de texto que contiene todos los comandos necesarios para desarrollar proyectos de software.
- Todos los **Makefiles** están ordenados en forma de reglas, especificando qué es lo que hay que hacer para obtener un módulo en concreto.
- El **make** sigue las reglas de este archivo y ejecuta cada uno de sus comandos en el orden estipulado chequeando que se cumplan cada una de sus dependencias.

Makefile

- Definir un archivo llamado Makefile en el directorio raíz de nuestro proyecto, donde está también todo nuestro codigo.
- Dentro de este archivo escribimos las reglas necesarias para reconstruir nuestro proyecto
- Para ejecutar simplemente escribimos en nuestra línea de comandos:

make <regla>

Dependencias

- Sabemos que para llegar a un ejectuable primero debemos llegar a un archivo **objeto**. Y para llegar al mismo debemos contar con los archivos **fuente o de código** y los **headers**.
- En otras palabras:
  - El ejecutable depende de los archivos objetos y/o librerías
  - El archivo objeto depende de los arvhivos de código
- A la hora de escribir un Makefile vamos a prestar especial atención a todas estas depdencias. Ya que make asumirá que se ha tenido en cuenta dicho criterio

Creación del Makefile

- Como dijimos anteriormente, el Makefile está conformado por una serie de reglas. Cada una de ellas deberá cumplir con el siguiente formato:

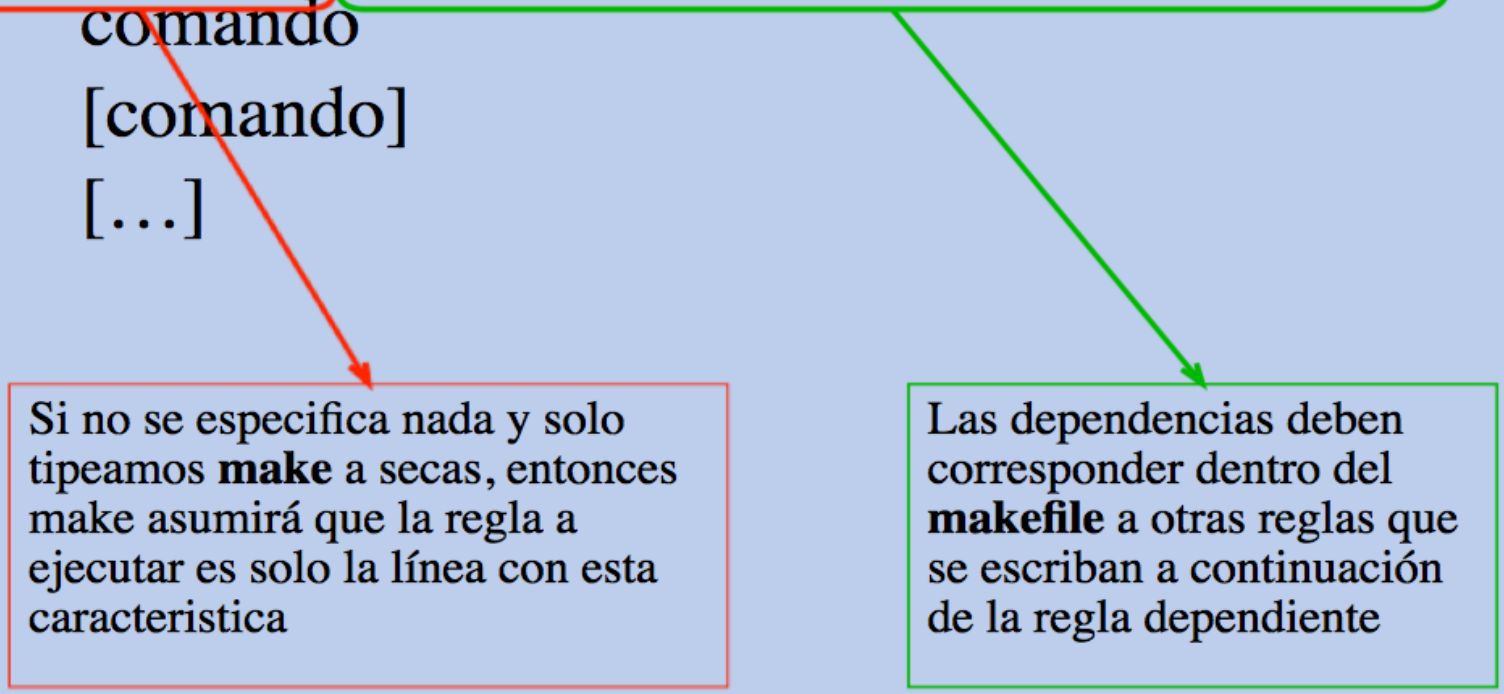
target : dependencia [dependencia][...]
comandos

Veamos de qué se trata cada una de las partes que conforman una regla:

- **target**: también se le dice regla u objetivo. el archivo objeto que make trata de crear
- **dependencia**: generalmente archivos requeridos para construir un target.
- **comando**: a ser ejecutados con el fin de crear el target a partir de las dependencias entre archivos especificados.

Creacion del makefile

objetivo : dependencia [dependencia][…]
comando
[comando]
[…]



Ejemplo de Makefile

```
ejecutable : a.o b.o
    gcc a.o b.o –o ejecutable –lutils
a.o : a.c d.h
    gcc –c –o a.o a.c
b.o : b.c d.h
    gcc –c –o b.o b.c
clean:
    rm –f ./*.o
    rm –f ejecutable
```

Esta regla limpia todos los archivos generados a partir de los fuentes.
Si ejecutamos ***make clean*** se ejecuta solo esta regla pasando por alto las demas

Para tener en cuenta

- **Importante**: Se deben respetar las tabulaciones. Luego de escribir el objetivo y la dependencia, los comandos deben contar con un **TAB**, que no es lo mismo que hacer 8 espacios!
- Si esto no se respeta, al querer ejecutar el make, recibiremos un mensaje de error: missing separator y se detendrá

Como trabaja el make

- Si no se le indica nada y simplemente se ejecuta:

  $make

  La aplicación busca el Makefile y comienza a ejecutar la primer regla (objetivo o target) que figura en el archivo.

- Si queremos ejecutar alguna regla en particular vamos a tener que especificárselo al make al momento de invocarlo:

  make <nombre de la regla>

  El make va a abrir el archivo Makefile y va a buscar el nombre de la regla que se le ha especificado e intetará ejecutar a partir de ahí, respetando las dependencias.

Cómo trabaja el make

- Si un objetivo no existe en el directorio, el make lo construye ejecuntando los comandos que figuran en el Makefile
- Si el objetivo existe, make hace un chequeo por cambios. Esto se hace mediante la lectura del timestamp que todos los archivos poseen. Cada vez que un archivo es modificado, el timestamp es modificado.
- make compara la fecha y hora del target con la fecha y hora de los cuales depende. Si al menos uno de dichos archivos es más nuevo que el target, entonces el make reconstruye el target interpretando que el hecho de que ese archivo sea más nuevo es señal de que debe incorporarse al mismo algún cambio de código.

Variables del make

- Una variable es un nombre simbólico que será substituido por su valor en el momento de la aplicación de las reglas posteriores. Para definir una variable se utiliza la siguiente sintaxis:

<IDENTIFICADOR>=valor

- Por convenio las variables tienen identificadores con todas las letras en mayúscula. Para "expandir" una variable, es decir, para obtener su valor se utiliza la siguiente sintaxis:

$(IDENTIFICADOR)

Variables del make

- **Make** define algunas variables que es necesario conocer y que vamos a utilizar asignandole los valores que nos sea mas conveniente:
  - **CC**: esta variable es para definir el compilador que vamos a usar. La sigla proviene de C Compiler.
  - **CFLAGS**: variable para definir las opciones de compilación.
  - **LDFLAGS**: variable para definir las opcioner de linkeo
  - **LDLIBS**: variable para definir las librerías del linkeo.
- Además de estas variables podemos definir todas las que querramos, como por ejemplo una variable para listar los objetos que intervienen en un proyecto:

$$\mathbf{OBJS = a.o\ b.o}$$

Ejemplo de Makefile con el uso de variables

```
CC = gcc
CFLAGS = -g -c
LDFLAGS = -g -lm
OBJS = main.o
ejecutable : $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) -o ejecutable
main.o : main.c
	$(CC) $(CFLAGS) main.c -o main.o
clean:
	rm -f ./*.o
	rm -r ejecutable
new : clean ejecutable
```

Macros

- **$@**: Nombre completo del target
- **$?**:Es la lista de requisitos más recientes que el objetivo.
- **$<**: Es el primer requisito de la regla.
- **%**: Trabaja de wildcard, indicando el equivalente a "cualquier" cosa.
- #: Comenta una linea. Es la contraparte del // en C

Conociendo estas macros podemos escribir un Makefile más críptico pero también más portable...

```
CC = gcc
CFLAGS = -g -c
LDFLAGS = -g -lm
OBJS = main.o
ejecutable : $(OBJS)
	$(CC) $(LDFLAGS) $(OBJS) -o $@
%.o : %.c
	$(CC) $(CFLAGS) $<.c -o $@
clean:
	rm -f ./*.o
	rm -r ejecutable
new : clean ejecutable
svn:	svn add *.c
	svn add *.h
	svn add Makefile
	svn add Doxygen
	svn ci -m "Commit de una nueva versión"
```