TUTORIAL COMPILACIÓN Y EJECUCIÓN DE UN PROGRAMA EN C – LINUX

Este tutorial se divide en dos partes, la primera describe la compilación y ejecución de un programa en c para Linux utilizando el compilador gcc. En la segunda parte se describe la creación y uso de un archivo makefile para determinar automáticamente que piezas de un programa necesitan ser recompiladas de acuerdo a un conjunto de reglas.

GCC es un compilador integrado del proyecto GNU para C; es capaz de recibir un programa fuente y generar un programa ejecutable binario acorde a la arquitectura de la máquina donde ha de correr. La sigla GCC significa "GNU Compiler Collection".

Etapas de compilación

El proceso de compilación involucra cuatro etapas sucesivas: pre procesamiento, compilación, ensamblado y enlazado. Para pasar del código fuente a un archivo ejecutable es necesario realizar estas cuatro etapas en forma sucesiva. Los comandos gcc (para c) y g++ (para c++) son capaces de realizar todo el proceso de una sola vez. A continuación se describen de forma general estas 4 etapas:

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.

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

Ensamblado: El ensamblado transforma el programa escrito en lenguaje ensamblador a código objeto, el cual es un archivo binario en lenguaje máquina ejecutable por el procesador.

Enlazado: Las funciones de C/C++ incluidas en el código fuente, tales como printf() o cout, se encuentran ya compiladas y ensambladas en bibliotecas existentes en el sistema. En esta etapa se incorpora el código binario de estas funciones al ejecutable.

Compilación y ejecución de un programa en c – Linux

1. **Crear un directorio de trabajo:**

La shell se abre por defecto en el directorio raíz asignado a nuestro usuario. Creamos un subdirectorio para nuestro programa. El subdirectorio se crea con mkdir e ingresamos a él con cd.

```
mkdir Prueba_C
cd Prueba_C
```

2. Escribir un programa en el editor:

Podemos escribir nuestro programa con cualquier editor que conozcamos.
La extensión de un programa en c es ".c". Editamos un fichero nuevo ***HolaMundo.c*** para escribir en él.

El código de nuestro programa inicial será:

```
#include <stdio.h>
int main()
{
    printf ("Hola mundo.\n");
    return 0;
}
```

3. Compilar el programa HolaMundo.c:

Para compilar nuestro programa usamos el comando (compilador) gcc.

```
gcc HolaMundo.c
```

Esto compila el programa y produce un ejecutable, si no hay errores, que se llama **a.out**

Ese nombre no es el mejor para nuestro programa. Por ello gcc tiene opciones para poder darle al ejecutable el nombre que queramos. La opción es **-o NombreDeseado**. Borramos el **a.out** con el comando **rm** y volvemos a compilar con la opción **-o**.

```
rm a.out
gcc HolaMundo.c -o EjecutableHolaMundo
```

Aquí ya tenemos el ejecutable *EjecutableHolaMundo*.

4. Ejecutar el programa HolaMundo:

Para ejecutar el programa no solo basta con poner su nombre, linux sólo busca ejecutables en determinados directorios. Esos directorios son los que se indican en la variable de entorno PATH. Ni el directorio de nuestro proyecto ni el directorio actual de trabajo están en esa variable. Para ejecutar nuestro programa hay que poner delante el path donde se encuentra. Este path puede ser absoluto o relativo al directorio actual. Vale cualquiera de los siguientes:

```
./home/tu_cuenta/Prueba_C/HolaMundo
./EjecutableHolaMundo
```

Aquí el programa *EjecutableHolaMundo* se ejecutaría satisfactoriamente.

5. Compilación de dos o más ficheros:

Si tenemos dos ficheros de código como los siguientes:

holamain.c: Programa principal con llamadas a otro módulo.

```c
/*holamain.c*/

void Saluda();

int main(){
    Saluda();
    return 0;
}
```

saluda.c: Módulo con una función C.

```c
/*saluda.c*/

#include <stdio.h>

void Saluda(){
    printf("Hola mundo\n");
}
```

Podemos compilar ambos ficheros con el comando:

```
gcc holamain.c saluda.c -c
```

Esto genera los ficheros de código objeto *holamain.o* y *saluda.o.* El código objeto es un archivo binario ejecutable por el Sistema Operativo.

Ahora podemos enlazar los ficheros de código objeto con:

```
gcc -o ejecutableHolaMundo holamain.o saluda.o
```

También podemos realizar ambas operaciones, compilar y enlazar a la vez, con el comando:

```
gcc -o ejecutableHolaMundo holamain.c saluda.c
```

En los dos casos `ejecutableHolaMundo` es el nombre del ejecutable que generamos.

En este caso también se ejecutaría dicho archivo `ejecutableHolaMundo` con algunos de los siguientes comandos:

```
./home/tu_cuenta/<directorio>/ejecutableHolaMundo
./ejecutableHolaMundo
```

Comando make y archivo makefile

1. Introducción

El comando make se utiliza para determinar automáticamente que piezas de un programa necesitan ser recompiladas de acuerdo a un conjunto de reglas. Para proyectos con varios cientos de líneas de código, permite agilizar el proceso de construcción de los programas, y en general, facilita el trabajo de compilación para uno o más archivos.

De esta forma y con los archivos adecuados, make compila todos los programas fuentes. Si alguno de ellos sufre alguna modificación, sólo será recompilado aquel que fue modificado, más todos los programas que dependan de él. Por supuesto, es necesario indicarle a make la dependencia de programas, lo cual se realiza en el archivo Makefile.

2. Archivo Makefile

Un archivo 'makefile' es un archivo de texto en el cual se distinguen cuatro tipos básicos de declaraciones:

- Comentarios.
- Variables.
- Reglas explícitas.
- Reglas implícitas.

2.1 Comentarios

Al igual que en los programas, contribuyen a un mejor entendimiento de las reglas definidas en el archivo. Los comentarios se inician con el carácter #, y se ignora todo lo que continúe después de ella, hasta el final de línea.

```
# Este es un comentario
```

2.2 Variables

Se definen utilizando el siguiente formato:

```
nombre = dato
```

De esta forma, se simplifica el uso de los archivos Makefile. Para obtener el valor se emplea la variable encerrada entre paréntesis y con el carácter $ al inicio, en este caso todas las instancias de $(nombre) serán reemplazadas por dato.

Ejemplo:

```
SRC = main.c
```

Origina que la siguiente línea:

```
gcc $(SRC)
```

Será interpretada como:

```
gcc main.c
```

Sin embargo, pueden contener más de un elemento dato.

Ejemplo:

```
objects = programa_1.o programa_2.o programa_3.o programa_4.o programa_5.o

programa: $(objects)
    gcc -o programa $(objects)
```

Hay que notar que make hace distinción entre mayúsculas y minúsculas.

Existe un conjunto de variables que se emplean para las reglas explicitas e implícitas, clasificadas en dos categorías: aquellas que son nombres de programas (como CC) y aquellas que tienen los argumentos para los programas (como CFLAGS). Estas variables son provistas y contienen valores predeterminados, sin embargo, pueden ser modificadas, como se muestra a continuación:

```
CC = gcc
CFLAGS = -Wall -O2
```

En el primer caso, se ha indicado que el compilador que se empleará es gcc y sus parámetros son -Wall -O2. El argumento -Wall muestra todos los mensajes de error y advertencia del compilador, el argumento -O2 controla el nivel de optimización de tiempo y memoria de compilación. Para mayor información sobre las banderas de compilación puede visitar el siguiente link: https://wiki.gentoo.org/wiki/GCC_optimization

2.3 Reglas explícitas

Estas le indican a make que archivos dependen de otros archivos, así como los comandos requeridos para compilar un archivo en particular. Su formato es:

```
archivoDestino: archivosOrigen
    comandos  # Existe un carácter TAB (tabulador) antes de cada comando.
```

Esta regla indica que, para crear **archivoDestino**, make debe ejecutar comandos sobre los archivos **archivosOrigen**.

Ejemplo:

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

Esta regla establece que, para crear el archivo de destino **main**, deben existir los archivos **main.c** y **funciones.h** y que, para crearlo, debe ejecutar el comando:
gcc -o main main.c funciones.h

2.4 Reglas implícitas

Son similares a las reglas explícitas, pero no indican los comandos a ejecutar, sino que make utiliza los sufijos (extensiones de los archivos) y las variables como CC Y CFLAGS para determinar que comandos ejecutar.

Ejemplo:
funciones.o: funciones.c funciones.h

Origina que se active automáticamente el siguiente comando:
$(CC) $(CFLAGS) -c funciones.c funciones.h

El cual crea el código objeto de funciones.o

3. EJEMPLO DE MAKEFILE:

Creamos el archivo llamado ***holamain.c***, el cual contiene el siguiente código en C.

```c
#include "saluda.h"
int main(){
    Saluda();
    return 0;
}
```

Se crea un archivo llamado ***saluda.c,*** el cual contiene el siguiente código:

```c
#include <stdio.h>
void Saluda (){
    printf("Hola Mundo\n");
}
```

Además de los anteriores archivos, se crea otro fichero llamado ***saluda.h***, que contiene, la declaración de las funciones:

```
/*Declaracion de Funcion externa*/

void Saluda();
```

Teniendo los tres archivos anteriores se crea el ***makefile*** correspondiente con el nombre de ***mimake***, el cual contiene las siguientes instrucciones:

```
# VARIABLES
CC = gcc
CFLAGS = -g -Wall -O2
SRC = holamain.c saluda.c saluda.h
OBJ = holamain.o saluda.o

# Reglas explicitas
all: $(OBJ)
	$(CC) $(CFLAGS) -o miejecutable $(OBJ)
clean:
	$(RM) $(OBJ) miejecutable
# Reglas implicitas
holamain.o: holamain.c saluda.h
saluda.o: saluda.c
```

: Indica que la separación se realiza mediante **un carácter TAB (tabulador) antes de cada comando**

1. A la variable CC se asigna el valor gcc, lo cual indica que se utilizara el compilador gcc
2. A la variable CFLAGS se asignan banderas que permiten controlar la optimización y salida en pantalla de los errores de compilación.
3. Se crea la variable SRC y se le asignan los archivos fuente que se van a compilar.
4. Se crea la variable OBJ y se establece el nombre de los archivos código objeto, que serán resultado del ensamblado.
5. La regla indica que cuando se ejecute el comando ***make*** sobre el archivo ***mimake*** automáticamente se generan los siguientes pasos:
   a) se ejecuta la regla: ***$(CC) $(CFLAGS) -o miejecutable $(OBJ)***
   b) lo cual produce que se ejecute el siguiente comando:
   gcc -g -Wall -O2 -o miejecutable holamain.o saluda.o
   c) lo cual a su vez produce que se activen las reglas implícitas (2.4 ver definición de las reglas implícitas).
6. La regla indica que cuando se ejecuta el comando **make –f mimake clean** se borran todos los código objeto y el ejecutable creado.
7. Reglas implícitas que permiten automáticamente crear los códigos objeto.

Para ejecutar el anterior makefile es necesario ingresar al directorio donde se ha creado, y ejecutar el siguiente comando en la consola:

make -f mimake

Este comando nos generara el ejecutable del programa ubicado en el directorio correspondiente con el nombre de ***miejecutable***.

Si queremos ejecutar las funciones que están en el **clean**, que para este caso es eliminar los archivos ***holamain.o***, ***saluda.o*** y el ejecutable llamado ***miejecutable,*** es necesario el siguiente comando:

make –f mimake clean

Para poner a correr dicho ejecutable, es necesario el siguiente comando:

./miejecutable

El cual nos mostrara en la consola el resultado de nuestro programa realizado en C:

Hola Mundo