

Prácticas de Sistemas Operativos

Toñi Reina, David Ruiz, Juan Antonio Álvarez,
Antonio Tallón, Javier Gutiérrez, Pablo Neira, Paco Silveira,
José Ángel Bernal y Sergio Segura

Boletín #2: Introducción a C

Curso 2008/09

Índice

1. El compilador GNU de C.	2
2. Argumentos en la línea de comandos	3
3. Comando <code>make</code>	5
4. Ejercicios	7

1. El compilador GNU de C.

El programa gcc, es un compilador de programas escritos en lenguajes C, C++, Fortran y Objective C. Un compilador es un programa que traduce fuentes escritos en un lenguaje de programación (por ejemplo, en el lenguaje C) a programas que pueden ser ejecutados por el sistema operativo.

En esta sección no se estudiará el lenguaje de programación C, ni se verán todas las posibilidades de gcc. Sólo se dan unas indicaciones muy básicas para comenzar a compilar programas escritos en C con el compilador gcc.

En caso de que el programa a generar tenga un sólo archivo fuente. El proceso de compilación es sencillo, basta que escriba `gcc -o binario fuente.c` reemplazando *binario* por el nombre del ejecutable que desea generar y *fuentes.c* por el nombre del archivo fuente.

Ejemplo 1 *Empleando el Xwpe, escriba el programa que se presenta a continuación, genere un ejecutable con nombre holaMundo y ejecútelo.*

```
#include <stdio.h>

int main()
{
    printf("Hola Mundo!!\n");
    return 0;
}
```

En caso de que el proyecto conste de varios fuentes, debe compilar cada uno por separado para generar código objeto para cada uno y después enlazar todos los códigos objeto. Para compilar un fuente *fuentes.c* a código objeto *fuentes.o* emplee: `gcc -c fuentes.c`. Para enlazar varios códigos objeto (*fuentes1.o* y *fuentes2.o*) en un archivo ejecutable binario emplee: `gcc -o binario fuentes1.o fuentes2.o`

Ejemplo 2 *Utilizando el Xwpe escriba el siguiente código en un archivo salida_alt.h.*

```
#ifndef __SALIDA_ALT__
#define __SALIDA_ALT__

void muestra(char *);

#endif
```

Escriba el siguiente en un archivo salida_alt.c.

```
#include <stdio.h>

void muestra(char *msg)
{
    printf("Mensaje: %s\n",msg);
}
```

Escriba el siguiente texto en un archivo con nombre mensaje.c.

```
#include "salida_alt.h"

int main()
{
    muestra("Muestra este bonito mensaje por la salida estándar");
    return 0;
}
```

Finalmente genere código objeto para `salida_alt.c` y `mensaje.c` y enlázelos para crear un ejecutable `mensaje`.

En el caso de que en el proyecto haga llamada a funciones del sistema operativo tendremos que enlazar o no con las bibliotecas adecuadas del sistema operativo.

Por ejemplo si queremos realizar un programa que muestre por la salida estándar el nombre de la máquina y la dirección IP en la que se está ejecutando tendremos que hacer uso de las funciones `gethostname` y `gethostbyname` como se muestra a continuación:

```
/* maquina.c */
#include <unistd.h>
#include <netdb.h>
#include <sys/socket.h>

#define MAX 256
typedef char Cadena[MAX];

int main()
{
    Cadena nombre;
    struct hostent *hp;

    gethostname(nombre, sizeof(nombre));
    hp=gethostbyname(nombre);

    printf("La máquina se llama: %s\n", nombre);
    printf("La dirección IP es: %s\n", inet_ntoa(hp->h_addr));
}
```

Para saber qué ficheros de cabeceras tiene incluir en el fuente C y con qué bibliotecas tiene que enlazar a la hora de generar el ejecutable puede utilizar el comando `man`. Por ejemplo, si en el servidor de prácticas `murillo` ejecutamos el comando `man gethostbyname` obtenemos una salida con el aspecto que muestra la figura 1. En la sección de *synopsis* se indican las cabeceras necesarias (*netdb.h*) y las librerías con las que enlazar (*nsl*). De esta forma para generar el fichero ejecutable para *Solaris 8* tenemos que compilar de la siguiente forma: `gcc -o maquina -lnsl maquina.c`¹.

```
1:   Networking Services Library Functions           gethostbyname(3NSL)
2:
3:   NAME
4:   gethostbyname, gethostbyname_r, gethostbyaddr,
5:   gethostbyaddr_r, gethostent, gethostent_r, sethostent,
6:   endhostent - get network host entry
7:
8:   SYNOPSIS
9:   cc [ flag ... ] file ... -lnsl [ library ... ]
10:  #include <netdb.h>
11:
12:  struct hostent *gethostbyname(const char *name);
```

Figura 1: Salida del comando `man gethostbyname`

2. Argumentos en la línea de comandos

Muchos programas Unix obtienen información a través de argumentos introducidos por la línea de comandos, y si los programas están preparados para aceptarlos, podrán tenerlos en cuenta. Por ejemplo, en la Figura 2, el

¹Si intenta compilar este mismo programa en Linux observará que no es necesario enlazar nuestro código con ninguna biblioteca de sistema.

comando `ls` se va a ejecutar con dos argumentos, una opción `-l`, para indicarle que cambie su comportamiento por defecto, y un directorio `/tmp` para decirle cuál es el directorio que queremos listar.

```
$ ls -l /tmp
```

Figura 2: Ejemplo de línea de comandos con argumentos

Para que estos argumentos puedan ser tenidos en cuenta en nuestro programa C, tenemos que invocar a la función `main` con dos argumentos: `argc` y `argv`.

```
void main(int argc, char *argv[] ) {  
    . . .  
}
```

En este caso, el sistema operativo pasa dos parámetros a la función principal del programa. `argc` es un entero que indica el número de argumentos en la línea de comandos, y `argv` es un array de punteros a carácter, en donde cada puntero apunta a uno de los argumentos almacenados como cadena en algún lugar de la memoria. `argv[0]` apuntará a una cadena que contiene el nombre del programa. En la Figura 3 se muestra mediante una representación gráfica cuál sería el contenido de las variables `argc` y `argv` para el caso del ejemplo mostrado en la Figura 2.

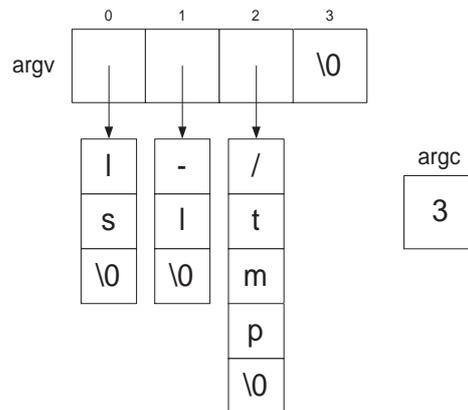


Figura 3: Contenido de `argc` y `argv`

Si el programa no ha sido escrito o preparado para recibir y tratar argumentos que se indiquen en la línea de comandos, no es necesario especificar nada en la declaración de la función `main()`, tal y como ocurre en los ejemplos 1 y 2 de este boletín.

Ejemplo 3 *Escriba el siguiente programa que se escribe a continuación, compílelo y ejecútelo. Este programa trabaja con la línea de argumentos, e imprime cada *n* segundos por la salida estándar el mensaje que se indica por la línea de argumentos. El formato de la línea de argumentos sería:*

```
$ imprime <segundos> <mensaje>
```

y un ejemplo de invocación del programa sería el siguiente:

```
$ imprime 5 "Estoy aquí"  
  
#include <stdio.h>  
#include <string.h>  
  
int main(int argc, char *argv[])  
{
```

```

int segundos;

if (argc != 3){
    fprintf (stderr, "Uso: %s <segundos> <mensaje>\n", argv[0]);
    return 1;
}

sscanf (argv[1], "%d", &segundos);

while (1){
    sleep (segundos);
    printf ("%s\n", argv[2]);
}

return 0;
}

```

¿Qué problemas podemos tener al tratar la línea de argumentos? Razone la respuesta.

3. Comando make

Resulta realmente útil utilizar el programa *make* a la hora de trabajar con proyectos de cierta envergadura, donde no tenemos uno o dos ficheros fuente, sino una amplia colección de ellos repartidos muchas veces en varios directorios.

Cuando trabajemos con múltiples archivos, puede ocurrir que olvidemos qué parte hemos actualizado (y por lo tanto debemos recompilar) y cuáles no. El programa *make* puede ocuparse por nosotros de llevar esa contabilidad y compilar sólo aquellos módulos del programa que sean necesarios.

El funcionamiento es sencillo, cuando llamamos a *make* éste busca un fichero de configuración en el directorio actual, que se debe llamar *Makefile* o *makefile*. Este fichero de configuración contiene lo que se conocen como dependencias, y le va a servir a *make* para comprobar que los ficheros de los que depende el proceso en cuestión están actualizados. Si no lo están, probablemente llevará a cabo alguna acción (como, p. ej., compilar un fuente para tener el objeto). Si alguna dependencia falla, esto es, algún fichero no existe y no tiene forma de crearlo, el programa dará un error y se detendrá. Si no, ejecutará los comandos indicados hasta el final.

Un fichero *Makefile* básico está formado por una lista de reglas. Las reglas constan de tres partes:

1. Una etiqueta, que usaremos en la llamada al programa *make*. Cada etiqueta distingue un grupo de acciones a realizar.
2. Un conjunto de dependencias, esto es, aquellos ficheros que son necesarios para llevar a cabo las acciones que corresponden a la etiqueta.
3. Los comandos a ejecutar.

La forma en la que se escriben las reglas es esta:

```

etiqueta : [dependencia1 ...] [comandos] [# comentarios]
[(tabulación) comandos] [# comentarios]
...

```

Un ejemplo de fichero *makefile* puede ser éste el que se muestra en el siguiente ejemplo:

Ejemplo 4 Para compilar los fuentes del caso práctico 2 escriba el siguiente texto en un archivo de nombre *Makefile* y después ejecute *make*.

```

mensaje: mensaje.o salida.o
        gcc -o mensaje mensaje.o salida.o

mensaje.o: mensaje.c salida_alt.h
        gcc -c -g mensaje.c

salida.o: salida.c salida_alt.h
        gcc -c -g salida.c

```

Para ver cómo funciona el comando *make* puede probar a modificar uno de los archivos fuente (por ejemplo cambie el módulo *mensaje.c* para que en lugar de imprimirse el mensaje por la salida estándar, se imprima en un archivo) y vuelva a ejecutar *make*. ¿Qué diferencias encuentra en las operaciones que *make* hace con respecto al punto anterior?

Como puede comprobar, *make* comprueba si se ha modificado alguno de los archivos de dependencias (en este caso, usted ha modificado *mensaje.c*) y ejecutará el conjunto de órdenes asociadas a la dependencia con etiqueta *mensaje.o* (el comando `gcc -c -g mensaje`), que hace que se genere de nuevo el archivo *mensaje.o*, que a su vez provoca la ejecución de los comandos asociados a la regla con etiqueta *mensaje*.

El ejemplo anterior es bastante simple. *make* también permite la definición de macros para ser utilizadas en las líneas de dependencias o en las de comandos. Un ejemplo más complejo con macros de fichero *makefile* puede ser éste el que se muestra en la figura 4

```

FILES= f1.c \
       f2.c \
       f3.c

OFILES= f1.o \
        f2.o \
        f3.o

LIBS= -lm

NOMBRE= proyecto

all: $(OFILES)
     gcc -g -o $(NOMBRE) $(OFILES) $(LIBS)

comp: $(FILES)
     gcc -g -c $(FILES)

save:
     tar cvfz /home/druiz/proyecto.tgz /home/druiz/proyecto

```

Figura 4: Ejemplo de fichero *makefile* con macros

Como se puede apreciar, las macros se definen al principio del archivo, y para ser referenciadas en la zona de dependencias se les antepone el carácter dólar y se encierran entre paréntesis. Note también que si una línea es demasiado larga, puede poner una contrabarra y continuar en la línea siguiente, tanto en la definición de macros como en la de comandos.

La etiqueta *all* se refiere a la creación del ejecutable final, mientras que *comp*² se encarga de compilar los fuentes y *save* sirve para hacer una copia de seguridad de todo el trabajo.

En el caso del ejemplo anterior, *make* también se puede utilizar con el siguiente formato:

```
make [etiqueta]
```

²Realmente, la etiqueta *comp* sobra, según las reglas implícitas del *make*, pero esto no lo vamos a ver en este curso introductorio

donde `etiqueta` es una de las definidas en el archivo `Makefile`. Así, si ejecutamos `make save` basándonos en el archivo del ejemplo anterior, comprimiremos y empaquetaremos todos los archivos del proyecto.

4. Ejercicios

1. Utilice Xwpe para editar un programa C que lea de la entrada estándar dos números, correspondientes al número de filas y al número de columnas de una matriz, e imprima por la salida estándar dicha matriz a la salida con el siguiente aspecto:

```
druiz@macarena:~/src/ssoo/boletin2/ej1 > ej1
Introduzca los valores de n y m separados por coma: 3,4
 1 2 3 4
 2 3 4 1
 3 4 1 2
druiz@macarena:~/src/ssoo/boletin2/ej1 >
```

Utilice el compilador en línea gcc para generar el ejecutable. No hace falta que almacene en memoria la matriz, simplemente imprima la estructura por la salida estándar.

2. Utilice Xwpe para editar un programa C que lea de la entrada estándar una cadena y la devuelva del revés:

```
druiz@macarena:~/src/ssoo/boletin2/ej2 > ej2
Introduzca la cadena: Hola Mariola.
.aloiraM aloH
druiz@macarena:~/src/ssoo/boletin2/ej2 >
```

Utilice el compilador en línea gcc para generar el ejecutable.

3. Realice las modificaciones necesarias para que los programas 1 y 2 tomen sus datos de entrada de la línea de comandos en lugar de la entrada estándar.

```
druiz@macarena:~/src/ssoo/boletin2/ej3 > ej3 3 4
 1 2 3 4
 2 3 4 1
 3 4 1 2

druiz@macarena:~/src/ssoo/boletin2/ej3 > ej3 1
ERROR!! Número incorrecto de parámetros.

druiz@macarena:~/src/ssoo/boletin2/ej4 > ej4 "Hola Mariola."
.aloiraM aloH

druiz@macarena:~/src/ssoo/boletin2/ej4 > ej4 Hola Mariola.
aloH .aloiraM
```

4. Escriba un programa al que se le pase como argumento un número mayor que uno, y devuelva el número primo inmediatamente superior.

```
druiz@macarena:~/src/ssoo/boletin2/ej5 > ej5 7
El siguiente número primo es: 11
```

5. Escriba un programa que reciba como parámetros las dos coordenadas cartesianas x e y de un punto del plano y devuelva como resultado un número del 1 al 4 que indique el cuadrante al cual pertenece al punto (no considere los ejes de coordenadas).

```
druiz@macarena:~/src/ssoo/boletin2/ej6 > ej6 3 -1
(1) | (2)
-----
(3) | (4)
El punto (3,-1) se encuentra en el cuadrante (4).
```

6. Realice un módulo matemático (biblioteca de funciones) que incluya un conjunto de funciones calculadas mediante el desarrollo en serie de *Taylor*. En concreto, las funciones (con sus correspondientes fórmulas) serán las siguientes:

- double seno (double x);

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

- double coseno (double x);

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

- double exponencial (double x);

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

En cada una de las series se irán calculando términos hasta que se obtenga un término cuyo valor sea menor que una constante infinitesimal $\varepsilon = 10^{-4}$. Una vez realizado el módulo, escriba el siguiente programa y ejecútelo:

```
#include <stdio.h>
#include "math.h"

#define PI 3.14159

int main (void)
{
    printf("El seno de pi/4 es: %g\n", seno(PI/4));
    printf("El coseno de pi/4 es: %g\n", coseno(PI/4));
    printf("El valor del número e es e^1: %g\n", exponencial(1.0));
    return 0;
}
```

7. Escriba un programa que lea una lista de números reales (hasta que lea un 0.0), los almacene en una tabla y calcule e imprima la suma, el valor máximo, el valor mínimo y el valor medio de todos ellos.

```
druiz@macarena:~/src/ssoo/ boletin2/ej7 > ej7
4.5
5.0
9.5
0.0
Suma: 19.000
Máximo: 9.500
Mínimo: 4.500
Media: 6.333
```

8. Compile ahora todos los ejercicios utilizando el entorno de desarrollo Xwpe.