Universidad
Carlos III de Madrid

SISTEMAS INFORMÁTICOS EN TIEMPO REAL

PRÁCTICAS DE LABORATORIO

DPTO. DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

José María Armingol Moreno
Miguel Ángel de Miguel Paraíso

PRÁCTICA 1

Introducción a la programación en Linux
COMPILACIÓN Y DEPURACIÓN DE PROGRAMAS

Compilar programas es una tarea imprescindible que desempeña el programador en C antes de ver culminado su objetivo final. La idea de esta sección es explorar un poco más en detalle la compilación sistemática de múltiples programas que conforman un proyecto en C. Se puede compilar directamente con "cc" o "gcc" o con el comando *make*. Para hacerlo con *make*, se necesita de un archivo de macros llamado *makefile*, donde se incluyen los detalles de compilación de cada uno de los programas a compilar. El "*makefile*" indica la relación entre los archivos del proyecto, hasta el efecto de recompilar archivos prerequisitos si se ha modificado el principal o si alguno ha sido eliminado. Para la depuración de programas se utilizará GDB (GNU DeBugger) de GNU (Gnu is Not Unix) que permite, entre otras muchas funciones, observar las variables de un programa mientras este se está ejecutando.

Compilador C

Para compilar un programa C, se utiliza la orden *cc* (compilador C):

$ gcc programa.c

Si no hay errores en el programa, aparece una versión ejecutable del programa en un fichero llamado *a.out*.

Si queremos que el nombre del ejecutable sea diferente, deberemos indicárselo al compilador, para ello es preciso utilizar la opción *–o*:

$ gcc programa.c –o programa

ahora el compilador coloca el código de salida en un fichero llamado *programa*, y hace ejecutable ese fichero.

Montaje de múltiples ficheros objeto

Cualquier proyecto de tamaño razonable tendrá muchos módulos que deberán compilarse por separado, cada uno en un fichero distinto. Cada uno de los módulos al ser compilado generará un fichero objeto **.o*. Para obtener el programa final será necesario montar todos los ficheros juntos.

La orden *ld* denominada editor de enlaces o montador, permite enlazar múltiples ficheros de código objeto y formar un fichero ejecutable único.

Ej:
Supongamos que tenemos los ficheros prog.c, prog1.c y prog2.c. La compilación se realizará de la siguiente manera:

$ gcc –c main.c -I.
$ gcc –c func1.c -I.
$ gcc –c func2.c -I.

o bien:

$ gcc –c main.c func1.c func2.c -I.

La opción *–c* es una instrucción que le indica al compilador que cree el correspondiente fichero *.o*, y que termine en dicho punto la compilación en vez de seguir el trámite completo de intentar montar el programa.

A continuación será preciso producir una versión ejecutable, para ello será necesario realizar lo siguiente:

$ gcc –o programa main.o func1.o func2.o

Mantenimiento de programas con *make*

La labor de *make* es facilitar el paso de texto fuente de un programa a la forma ejecutable final. *make* construye un programa de acuerdo con un conjunto de criterios contenidos en un fichero de descripción al que nos referiremos como *Makefile* o *makefile*, dicho fichero contiene la información sobre lo que pretendemos construir y las ordenes del sistema que *make* deberá invocar para conseguirlo.

Para el ejemplo del apartado anterior tendremos que crearnos el siguiente fichero Makefile:

```
# Fichero de compilación

programa: main.o func1.o func2.o
	gcc –o programa main.o func1.o func2.o

main.o: main.c
	gcc –c main.c -I.

func1.o: func1.c func1.h
	gcc –c func1.c -I.

func2.o: func2.c func2.h
	gcc –c func2.c -I.
```

make ignora las líneas que comienzan con # considerándolas como líneas asociadas a comentarios. La línea siguiente es una definición de dependencia. La primera palabra indica el objetivo, en nuestro caso programa. Se separa de las cosas que depende, llamadas prerrequisitos, por dos puntos (:). Los prerrequisitos son todos los ficheros objeto individuales.

Las siguientes líneas son las reglas, es decir, las órdenes que deben ejecutarse para construir el objetivo a partir de sus prerrequisitos.

A continuación, hemos definido otros tres objetivos, cada uno de los ficheros objeto depende de su fichero fuente y puede ocurrir que también de algún fichero de inclusión común *.h*.

Si consideramos que *make* es un programa inteligente, que tiene un cierto conocimiento predefinido de las dependencias, por ejemplo, sabe que un fichero *.o* dependerá de un fichero fuente *.c* del mismo nombre, por lo que buscará el fichero fuente y verificará si el fichero objeto necesita ser rehecho. Esto es lo que se denomina una *regla implícita*, es decir, que no figura explícitamente en el fichero *makefile* y que el comando *make* aplica por defecto. Por ello, se puede abreviar el fichero Makefile anterior a:

```
# Fichero de compilación
CFLAGS = -I.
programa: main.o func1.o func2.o
	gcc –o  programa  main.o func1.o func2.o
```

El fichero anterior aunque es breve contiene un cierto número de repeticiones, que se pueden simplificar utilizando macros:

```
# Fichero de compilación

CFLAGS = -I.

OBJECTS = main.o func1.o func2.o
programa: $(OBJECTS)
        gcc –o programa  $(OBJECTS)
```

Una macro se referencia haciendo preceder su nombre entre paréntesis con un signo $.

Existen una serie de macros que afectan a las reglas implícitas. Veamos algunos ejemplos que servirían para generar una aplicación que utiliza gráficos:

```
# La macro CC indica el compilador que queremos utilizar.
# Esta línea indica que se utilizará el compilador gcc
CC = gcc

# La macro CFLAGS almacena las opciones de que usará el compilador para generar
# los objetos (los archivos con extensión .o).
# Esta línea indica la ubicación de los ficheros de cabecera correspondientes a las
# bibliotecas de gráficos (opción -I), y que queremos código que se pueda depurar
# (opción -g):
CFLAGS = -I/usr/X11R6/incluye/X11  -g

# La macro LDFLAGS almacena las opciones de que usará el enlazador para generar
# los ejecutables a partir de los objetos.
# Esta línea indica la ubicación de las bibliotecas de gráficos (opción -L):
LDFLAGS = -L/usr/X11R6/lib

# La macro LIBS especifica qué librerías, además de las de sistema, debe utilizar el
# enlazador para generar el ejecutable.
# Esta línea indica que se utilizarán las bibliotecas de gráficos y la matemática
# (opción -l):
LIBS = -lXt –lX11 –lm
```

Estas macros son utilizadas por las reglas implícitas. Si la regla es explícita, debe incluirse la referencia a estas macros:

```
OBJECTS = prog.o prog1.o prog2.o
prog: $( OBJECTS)
        $(CC) $(LDFLAGS) $(LIBS) –o prog  $(OBJECTS)
```

El comando make, por defecto, tratará de generar el primer objetivo indicado en el makefile. Por ejemplo, dado el siguiente makefile:

```
CC = gcc
CFLAGS = -g
LDFLAGS =
LIBS = –lm

padre: padre.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) padre.o funciones.o –o padre

hijo1: hijo1.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) hijo1.o funciones.o –o hijo1

hijo2: hijo2.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) hijo2.o funciones.o –o hijo2
```

El comando make solo generará el ejecutable padre. Para generar los tres ejecutables de manera automática, se suele incluir al principio un objetivo ficticio denominado *all*, que depende del resto de ejecutables de la aplicación. Siguiendo el ejemplo anterior, tendríamos:

```
CC = gcc
CFLAGS = -g
LDFLAGS =
LIBS = –lm

all: padre hijo1 hijo2

padre: padre.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) padre.o funciones.o –o padre

hijo1: hijo1.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) hijo1.o funciones.o –o hijo1

hijo2: hijo2.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) hijo2.o funciones.o –o hijo2
```

Otro objetivo ficticio que se emplea con frecuencia es el objetivo *clean*, cuya misión es eliminar todos los objetos y los ejecutables. La ejecución de *make clean* borrará todos estos archivos

```
CC = gcc
CFLAGS = -g
LDFLAGS =
LIBS = –lm

all: padre hijo1 hijo2
```

```
padre: padre.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) padre.o funciones.o –o padre

hijo1: hijo1.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) hijo1.o funciones.o –o hijo1

hijo2: hijo2.o funciones.o
	$(CC) $(LDFLAGS) $(LIBS) hijo2.o funciones.o –o hijo2

clean:
	rm –f padre.o hijo1.o hijo2.o funciones.o padre hijo1 hijo2
```

Depurador GDB (DNU DeBugger)

El propósito de un depurador (debugger) de programas no es sino permitir al programador conocer algunos detalles de qué está ocurriendo dentro del programa mientras éste se está ejecutando, o bien, qué estaba haciendo el programa en el momento en que falló. GDB puede realizar las siguientes operaciones para detectar y cambiar cosas en el programa en tiempo de ejecución y así depurar un fallo y experimentar los consiguientes efectos:

- Comenzar el programa con unos valores iniciales.
- Parar el programa en determinados puntos o líneas de programa, así como bajo ciertas condiciones.

Para poder utilizar *gdb* con un programa, previamente éste ha de haberse tenido que compilar con la opción *-g* Esta opción, introduce en el fichero binario ejecutable que es salida del compilador, una serie de índices necesarios para el depurador.

GDB permite los siguientes modos de ejecución:

1. Modo consola enlazado a un programa no ejecutado. En este modo, gdb abre una consola y queda a la espera de comandos por parte del usuario (ver lista de comandos, más adelante)
2. Modo consola enlazado a un programa que ya estaba en ejecución. Análogo al modo anterior, salvo que el programa que se va a depurar es un programa que ya estaba en ejecución.
3. Recibiendo comandos de un fichero externo mediante tuberías (ver práctica 2)

```
Sintaxis:  gdb    [-help] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev]
           [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir]
           [prog[core|procID]]
```

En esta práctica solo se utilizará:

```
>>gdb  [prog[core|procID]]
```

Comandos de GDB:

- *break [file:]function*, establece un "punto de ruptura" en una función o en una línea concreta del programa. Un punto de ruptura es un punto del programa en el que la ejecución del mismo se detiene.
- *run [arglist]*, comienza el programa con la lista de argumentos "arglist".
- *bt*, muestra la pila del programa.
- *print expr*, muestra el valor de una expresión.
- *c*, continúa después de un punto de ruptura.
- *next*, ejectua la siguiente línea del programa. Si ésta es una llamada a función, "step" salta dicha función.
- *edit [file:]function*, edita la línea de programa donde se encuentra parado.
- *list [file:]function*, lista el texto del programa en la vecindad de la línea donde se encuentra parado.
- *step*, ejecuta la siguiente línea de programa. Si ésta es una llamada a función, "step" penetra en dicha función.
- *help*, muestra un resumen de ayuda.
- *quit*, sale de la consola GDB.

En esta práctica se va a trabajar con dos programas:
1.- punteros.c
2.- misPunteros.c

0.- Abrir la herramienta de depuración GDB:
>> gdb
GNU gdb Red Hat Linux (6.1post-1.20040607.41rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)

Introducir el comando "help" y leer con atención lo que muestra.
Introducir el comando "quit".

1.- punteros.c
A) Tecleear el siguiente programa:

PROGRAMA 1

```
#include<stdlib.h>      /* cabecera de la librería estándar de  C*/
#include<stdio.h> /* cabecera de la librería de entrada/salida de datos
*/
#include<errno.h> /* cabecera de la librería de manejo de códigos de
error */
#include<string.h>

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

```
  // argc - numero de argumentos de entrada + 1
  // argv[] - array de argumentos de entrada
  //   argv[0] = RUTA/nombre-programa
  //   argv[1] = argumento 1
  //   argv[i] = argumento i
  //   argv[argc] = ultimo argumento introducido
  // argv[>argc] = datos locales (nombre maquina, etc)

  // punteros a entero
  int * pt_i;
  int * pt_j;
  // puntero a caracteres
  char * frase;
  int i;
  size_t n;

  // malloc - reserva de memoria dinámica
  pt_i = (int*)malloc( 1 * sizeof( int ) );
  pt_j = (int*)malloc( 1 * sizeof( int ) );
  frase = (char*)malloc( 15 * sizeof( char ));

  // Asignación del contenido de memoria del puntero a entero
  *pt_i = 29;
  printf( "(1) pt_i = %d\n",*pt_i );
  *pt_j = 102;
  printf( "(1) pt_j = %d\n",*pt_j );
  pt_j = pt_i;

  *pt_i = 209;

  printf( "(2) pt_i = %d\n",*pt_i );
  printf( "(2) pt_j = %d\n",*pt_j );

// Pregunta para el alumno: ¿Cuánto vale *pt_j?
// ¿Cómo es que ha cambiado su valor de 102 a 209?

  // Asignación a cadena de caracteres
  strcpy( frase, "Esto es una frase" );
  n = strlen( frase );
  printf( "(3) frase[%d] = %s\n",(int)n,frase );
  printf( "(4) primera letra es = %c\n", *frase );
  frase = frase + 1;
  printf( "(5) segunda letra es = %c\n", *frase );

  // free - libera memoria dinámica
  free( pt_i );

  return 0;

}
```

B) Compilar con la siguiente linea en el *shell*:
>>gcc -g -o puntero punteros.c
donde,

- *gcc*, es el compilador para C.
- *-g*, es la opción que añade los índices para el depurador GDB.
- *-o*, (output) salida del compilador. Fichero binario ejecutable

- *puntero*, fichero binario de salida.
- *punteros.c*, fichero de entrada (fuente).

C)Lanzar *gdb* en modo consola, indicando como argumento de entrada el ejecutable *puntero*
>>gdb puntero
GNU gdb Red Hat Linux (6.1post-1.20040607.41rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...(no debugging symbols found)...Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb)

GDB se queda a la espera de recibir comandos por parte del programador.

D)Colocar puntos de ruptura en las correspondientes líneas de programa (comando *break*):

- justo después de la asignación de memoria. (línea 30)
- justo después de la asignación de valores del contenido de memoria del puntero. (línea 34)
- justo después de la igualdad entre punteros. (línea 36)

E) Ejecutar el programa e ir observando el cambio de los valores tanto de los punteros como de su contenido de memoria (comandos *run, print, list,…*).

F) Explicar lo ocurrido gráficamente, esto es, ayudarse de un simple diagrama de memoria de los punteros y del contenido de memoria a donde apuntan.
Se deberá entender porqué el valor del contenido del puntero *pt_j* ha cambiado si no se ha cambiado explícitamente el contenido de memoria al que apunta (**pt_j*)
G) Termina la ejecución
I) Vuelve a lanzar la ejecución con un parámetro de entrada: *run 246810*
J) Observa los valores de *argc* y *argv* Podrás ver que el parámetro de entrada ha quedado grabado en la variable *argv* como una cadena de caracteres.

Ejercicios:

1. Modificar el código del programa para que la asignación a uno de los valores a los que apunta un puntero a entero (*pt_i) se realice mediante un parámetro de entrada al programa (argv)
2. Modificar el código del programa para que la asignación a la cadena de caracteres “frase” se realice mediante un parámetro de entrada al programa (argv)

3. Añadir un bucle al final del código del programa que vaya recorriendo el array "frase" y que en cada iteración vaya mostrando por pantalla cada carácter del array. (Sugerencia: utilizar *frase = frase + 1* o bien *frase ++* para ir recorriendo el array "frase")
4. Coloca un punto de ruptura en medio del bucle para visualizar la variable *frase ¿Qué carácter hay al final de cada cadena de caracteres?

Código para el ejemplo de makefile:

main.c

```
#include <func1.h>
#include <func2.h>

int main() {

  printFunc1();
  printFunc2();

  return(0);
}
```

func1.c

```
#include <stdio.h>
#include <func1.h>

void printFunc1(void) {

  printf("Function 1\n");

  return;
}
```

func1.h

```
void printFunc1(void);
```

func2.c

```
#include <stdio.h>
#include <func2.h>

void printFunc2(void) {

  printf("Function 2\n");

  return;
}
```

Func2.h

```
void printFunc2(void);
```