Sesión 1: Introducción al entorno de desarrollo (Y a ciertos conceptos fundamentales de C)

José Luis Díaz

Curso 2011-2012

1. Introducción: el entorno de desarrollo

Las salas de prácticas constan de equipos PC con sistema operativo Windows. Además, fuera de estas salas, existen unas máquinas servidoras que son también PC, pero de mayor potencia y capacidad, y que corren el sistema operativo Linux. Todas estas máquinas están conectadas por una red, por lo que es posible construir con ellas sistemas distribuidos.

De hecho, ya está montado un sistema de archivos distribuído. Cuando un alumno entra en sesión en una máquina Windows, su clave es validada en un servidor central, de modo que el usuario pueda usar la misma clave en todas las máquinas de prácticas. Del mismo modo, la carpeta personal del usuario no está presente físicamente en el PC desde el cual ha entrado en sesión, sino en una de las máquinas servidoras. No obstante, como parte del proceso de arranque de la sesión, Windows conecta esta carpeta remota a una unidad de red local, de modo que el usuario pueda cambiar a la unidad "`Z:`" y trabajar en ella como si fuera un disco local.

En las prácticas de sistemas distribuidos trabajaremos a veces de forma local en la máquina Windows desde la que se entra en sesión, y a veces de forma remota en una de las máquinas Linux, a la que nos conectaremos mediante el protocolo `ssh`. En esta primera sesión introductoria realizaremos este tipo de conexiones y copiaremos ficheros entre las diferentes máquinas.

A la vez, practicaremos con los entornos de desarrollo que utilizaremos, tanto en Windows como en Linux, recorriendo el proceso de desarrollo típico: editar, compilar, ejecutar. Realizaremos estas tareas sobre programas sencillos escritos en lenguaje C, que servirán de paso para ejemplificar ciertos conceptos fundamentales de este lenguaje, y que no obstante no suelen ser bien conocidos.

2. Desarrollo en Windows

2.1. Preparación del entorno

Para el desarrollo en Windows usaremos el compilador Visual C de Microsoft, pero no usaremos su entorno de desarrollo integrado (IDE), porque es demasiado pesado y complejo para los sencillos programas que vamos a desarrollar. En su lugar, usaremos el compilador de línea de comandos (CL).

Los programas de ejemplo con los que trabajaremos en esta sesión se suministran desde el servidor web de la asignatura. Necesitas una copia local de estos ficheros. Para ello:

- Abre un navegador Web y ve a la página `http://www.atc.uniovi.es/telematica/3sod/material_practico.php`
- Pincha con el botón derecho sobre el enlace "Código fuente para Windows" y elige la opción "Guardar destino como...", guardando así el fichero `Windows.zip`.
- Descomprime ese fichero en tu carpeta de usuario.

2.2. Compilación y edición

Vamos a probar a compilar uno de los programas de ejemplo anteriores.

- Busca en el menú Inicio de Windows el programa llamado "Símbolo del sistema de Microsoft Visual Studio" y ejecútalo. Esto abrirá una "Interfaz de comandos" preparada para la compilación de programas en C.

- En el interfaz de comandos, escribiendo `Z:` se cambia a la unidad del usuario, y escribiendo repetidamente el comando "`cd carpeta`" se puede entrar en las carpetas que se van encontrando. El comando "`cd ..`" permite "subir" a la carpeta superior, y el comando `dir` averiguar los contenidos de una carpeta.

  Usando estos comandos, situarse en la carpeta en la que se han descomprimido todos los ejemplos (programas en C)

- Mediante el siguiente comando, intentaremos compilar el primer ejemplo:

```
Z:\SOD> CL /W3 parametros.c
```

  Encontraremos que contiene un error. Véase la sección 4.1 en este mismo documento para comprender la naturaleza de este error. Para poder compilar el programa, será necesario editar el fuente y corregir dicho error.

- La edición del código fuente debe hacerse con un editor de "texto plano" (es decir, que no introduzca códigos especiales para marcar el tipo de letra o su tamaño). Se recomienda usar un editor específico para programadores, como LopEdit, o en su defecto el programa `wordpad` de windows. El `notepad` también es válido en principio, pero no funcionará correctamente si el código fuente que tratamos de editar tiene los retornos de carro al "estilo Unix", como será el caso si el fuente fue creado en una máquina Unix. En cambio `wordpad` reconoce también este formato.

  Existen otros editores gratuítos mucho más adecuados para la edición de código fuente. Uno de ellos es `notepad++`.

- Una vez corregido el error, se reintenta la compilación y se comprueba que no hay más errores. El programa puede ejecutarse simplemente escribiendo su nombre desde la interfaz de comandos[1]:

```
Z:\SOD> PARAMETROS
Se han recibido 1 parámetros.
Parámetro 0: |PARAMETROS|
```

  La misión de programa es obtener los parámetros que se le pasen desde la línea de comandos, y mostrarlos (al imprimir, se añaden unas barras verticales alrededor del texto de cada parámetro, para que sea visible si el parámetro contiene espacios al principio o al final)

- Experimenta con el programa. Prueba por ejemplo las siguientes:

```
Z:\SOD> PARAMETROS UNO DOS TRES
Z:\SOD> PARAMETROS ' UNO DOS TRES '
Z:\SOD> PARAMETROS " UNO DOS TRES "
Z:\SOD> PARAMETROS *.c
```

3. Desarrollo en linux

3.1. Programas necesarios para conectar con las máquinas de prácticas

Para poder transmitir los archivos a las máquinas linux de prácticas, y para poder "entrar en sesión" en estas máquinas y trabajar en ellas, son necesarios dos programas diferentes:

- Un programa para transferir ficheros entre la máquina local y la máquina de prácticas.
- Un programa para iniciar sesión en línea de comandos en la máquina de prácticas.

Las máquinas de prácticas (`bilbo`, `frodo`, `gollum`) son PC's con sistema operativo linux, y por motivos de seguridad sólo soportan el protocolo *SSH*, tanto para transferir ficheros como para iniciar sesión. Por tanto necesitaremos herramientas que soporten estos protocolos. Si bien hay muchas posibilidades, sugiero las dos siguientes por ser gratuitas y de gran calidad:

[1] Si en lugar de "`parámetros`" tu terminal muestra algo como `parßmetros`, se debe a que la terminal no usa la misma codificación que el fuente del programa. Puedes arreglarlo seleccionando en el menú de preferencias de la terminal el tipo de letra `Lucida Console`, y escribiendo seguidamente en la terminal el comando `CHCP 1252`

- Para transferencia de ficheros: WinSCP (`http://winscp.net/eng/download.php`). Presenta una interfaz gráfica similar a la del explorador de windows, en la que se pueden ver los ficheros de la máquina local y los de la máquina remota, así como copiar ficheros de una máquina a otra.
- Para "iniciar sesión" y trabajar en la máquina remota: PuTTY (`http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`. Presenta un extenso menú de opciones, pero basta introducir el nombre de la máquina y asegurarse de que está seleccionado el protocolo *ssh*. Una vez establecida la conexión, tendremos una terminal en la que ejecutar comandos en la máquina remota (son necesarios conocimientos básicos de linux para saber qué comandos tenemos disponibles).

3.2. Preparación del entorno

En linux el compilador a usar se llama `gcc`, y no necesita ninguna preparación especial antes de ser invocado.

Naturalmente lo que sí necesitaremos serán los códigos fuente de los programas de ejemplo, y éstos de momento los tenemos en la máquina Windows. Basta usar alguna de las utilidades descritas en el punto anterior para transferir estos ficheros a la máquina linux de prácticas, que será `gollum.edv.uniovi.es`.

Una vez transferido el fichero a la máquina linux, debemos abrir una sesión interactiva (*shell*) para trabajar. Usa para ello el programa `PuTTY`, y conéctate a la máquina `gollum.edv.uniovi.es`.

3.3. Compilación y edición

El comando para compilar un programa en C en linux es el siguiente:

```
$ gcc -Wall -o \emph{ejecutable} \emph{código-fuente.c}
```

La opción `-Wall` es para solicitar el máximo nivel de *warnings* (avisos), y la opción `-o` es para darle un nombre al ejecutable. Sin esta opción, el ejecutable se llama siempre por defecto `a.out`. Por ejemplo, para compilar el programa `parametros.c` la orden sería:

```
$ gcc -Wall -o parametros parametros.c
```

lo que generará un ejecutable llamado `parametros` (sin extensión), que podremos lanzar escribiendo:

```
$ ./parametros
```

El `./` delante del nombre del programa es necesario para indicar al sistema que el programa se halla en la "carpeta actual" (en Unix, el carácter "." representa a la carpeta actual). A diferencia de Windows, el sistema no busca el ejecutable en la carpeta actual, sino sólo en la lista de carpetas especificada en una variable de entorno llamada `PATH`. Puedes averiguar qué valor tiene esta variable escribiendo: `echo $PATH`. Verás una lista de nombres de carpeta separados por el carácter `:`. Cada vez que le das al sistema un comando, el sistema buscará en estas carpetas por orden, hasta encontrar un programa que se llame así en una de ellas. Si no lo encuentra, dará un error. La carpeta "." no forma parte del `PATH`, y por eso es necesario especificarla delante del nombre del ejecutable[2]

Si el programa contiene errores, el compilador te indicará en qué líneas aparecen. Deberás usar un editor para corregirlos. Los editores más habituales en el mundo linux son `vi` y `emacs` (en las máquinas de prácticas `emacs` se llama `xemacs`). Estos editores no son muy cómodos, de acuerdo con los estándares modernos de usabilidad, pero al menos están disponibles en todas las máquinas, por lo que es conveniente saber defenderse mínimamente con ellos.

Explicar a fondo estos editores se sale de los objetivos de este documento, pero mencionaremos seguidamente cuatro comandos básicos para poder usarlos en una emergencia.

[2] Otra solución es añadir la carpeta "." al `PATH`. Para ello debes escribir el comando `export PATH=$PATH:.`

3.3.1. El editor vi

Este editor tiene dos modos de funcionamiento: el modo comando y el modo escritura. Cuando se está en modo comando, no se puede añadir texto al documento, ya que cada letra que pulsemos se interpreta como un comando para hacer algo (guardar documento, borrar carácter, etc.). Para poder escribir hay que pasar al modo escritura. En el modo escritura, cada letra que pulsemos se añade al documento, como en un editor normal, pero a cambio no podremos realizar muchas acciones como guardar fichero, salir del editor, etc, que sólo están disponibles en el modo comando.

Por tanto, editar con vi supone cambiar continuamente entre el modo escritura (para añadir texto) y el modo comando (para moverse por el fichero, guardarlo, etc.)

- **Modo comando**. En este modo cada tecla se interpreta como un comando. Estos son los comandos más útiles:

  `:0` Va al principio del fichero

  `:$` Va al final del fichero

  `:`*`n`* Va a la línea *n* del fichero

  `0` Va al principio de la línea

  `$` Va al final de la línea

  `x` Borra el carácter bajo el cursor

  `D` Borra desde el cursor hasta el fin de línea

  `:w` Guarda el fichero

  `:q` Sale del editor

  `ZZ` Atajo: guarda y sale

  Los siguientes comandos pasan al modo de escritura y nos permiten añadir texto al fichero:

  `i` Inserta texto en la línea, en el lugar donde esté el cursor

  `A` Añade texto al final de la línea

  `o` Abre una línea bajo la actual para escribir en ella

  `O` Abre una línea sobre la actual para escribir en ella

- **Modo escritura**. En este modo se puede escribir, y borrar con la tecla de retroceso si nos equivocamos. Pulsando *ESC* se sale del modo escritura y se vuelve al modo comando.

3.3.2. El editor emacs

Este editor, si detecta que el usuario tiene disponible un servidor X-window[3], lo usará y abrirá una ventana nueva en la que realizar la edición. En caso contrario, el editor se abrirá en la misma terminal desde la que fue lanzado el comando. Cuando *emacs* detecta un servidor X-window y se ejecuta en *modo ventana*, el usuario puede utilizar el ratón para seleccionar texto, y para acceder a un menú desde el cual guardar fichero, copiar, pegar, etc. En cambio, cuando se abre en la propia terminal (modo *no-ventana*), el ratón es inútil, y todo el control de *emacs* se hace a base de combinaciones de teclas. Resulta conveniente por tanto conocer algunas de las combinaciones más importantes.

En la siguiente lista de teclas, el prefijo `C-` representa la tecla "Ctrl". Así, si pone `C-e`, representa la pulsación simultánea de las teclas "Ctrl" y "e". El prefijo `M-` representa lo que *emacs* llama "la tecla Meta". Esta tecla no existe en la mayoría de los teclados, pero dependiendo de la instalación normalmente estará asignada a la tecla Alt izquierda, de modo que si ponemos `M-w` queremos indicar la pulsación simultánea de las teclas "Alt" y "w". No obstante, en algunas terminales la tecla "Meta" no está asignada a la tecla "Alt", ni a ninguna otra, por lo que parecería imposible teclear la combinación `M-w`. Para este caso, emacs permite que se pulse en secuencia la tecla "*ESC*" y después "w". En general, la secuencia ESC x representa la pulsación de `M-x`, siendo `x` cualquier tecla.

En particular la combinación `M-x` sirve para ejecutar cualquier comando emacs. Cuando pulsamos esta combinación, emacs espera a que escribamos el nombre de un comando en la última

[3]Las salas de prácticas tienen disponible un servidor X-window si se ha arrancado previamente el programa `Xwin`, y se ha iniciado sesión en la máquina linux usando el comando `ssh` con la opción `-X`.

línea de la pantalla[4]. Los comandos emacs tienen nombres largos separados por guiones, como por ejemplo "number-line-mode". Podemos teclear sólo las primeras letras del comando y pulsar *TAB* para que emacs escriba el resto, siempre que no haya ambigüedad. Una vez aparece el nombre del comando, pulsando *Enter*, lanzaremos su ejecución. Muchos comandos nos pedirán seguidamente algunos datos necesarios para ejecutar ese comando.

M-< Ir al principio del fichero

M-> Ir al final del fichero

M-x goto-line Ir a una línea dada del fichero

M-x line-number-mode Activa el modo de numeración, en el cual se puede leer en la línea de estado (penúltima línea de la pantalla) en qué número de línea está el cursor

C-a Ir al principio de la línea

C-e Ir al final de la línea

C-d Borrar el carácter bajo el cursor

C-k Borrar desde el cursor hasta el fin de línea (y lo almacena en el portapapeles)

C-x C-f Guardar fichero

C-x C-c Salir del editor

C-x C-u Deshacer (*undo*) el último cambio

C-g Cancelar. Esta tecla sirve para salirse de una opción en la que hayamos entrado sin querer. Por ejemplo, al pulsar una combinación de teclas nos encontramos con que emacs nos está preguntando si estamos seguros, pero no sabemos dónde nos hemos metido. Pulsando repetidamente `C-g` acabaremos por salir de cualquier atolladero sin causar males mayores.

En algunas terminales la tecla de retroceso no borra, sino que invoca la ayuda de emacs (esto se debe a que esta tecla envía el código `C-h`). Si se da este caso, puedes acceder a la tecla de borrado pulsando Ctrl simultáneamente con la tecla de retroceso.

Para copiar y pegar bloques de texto, debes saber cómo seleccionar en emacs. Para emacs, el texto seleccionado es todo aquel comprendido entre "*la marca*", y "*el cursor*". La marca es un punto invisible que el usuario puede fijar pulsando `C-espacio`. El cursor puede moverse con las teclas normales de movimiento. Algunas versiones de emacs pintan de otro color la selección a medida que crece. En otras versiones no se pinta nada por lo que la selección es invisible. Una vez hemos seleccionado un texto, tenemos las siguientes teclas para manipularlo:

M-w Copiarlo al portapapeles (sin borrarlo del fichero)

C-w Cortarlo (lo borra del fichero además de copiarlo al portapapeles)

C-y Pegar el último texto que habiamos copiado al portapapeles.

C-y M-y Pegar el penúltimo texto que habíamos copiado al portapapeles.

C-y M-y M-y Pegar el antepenúltimo texto que habíamos copiado al portapapeles, etc. Cada pulsación adicional de `M-y` retrocede en el portapapeles.

[4]En realidad, las demás combinaciones de teclas no son sino "atajos" para acceder rápidamente a los comandos más comunmente usados. La ejecución mediante `M-x` nos permite acceder también a los comandos que no tienen asignado un atajo de teclado

4. Programas de ejemplo

Los siguientes programas pueden servirte para probar el entorno de desarrollo de Windows y de las máquinas linux. Además, cada uno de estos programas ilustra algún concepto fundamental, pero no muy bien conocido, de programación en lenguaje C.

El código fuente de estos programas puedes encontrarlo en la unidad "Public" compartida por la máquina Frodo.

4.1. parametros.c

Este programa ilustra cómo desde un programa en C se puede acceder a los parámetros que el usuario escribió al lanzar el programa.

La idea es que la función `main()` siempre recibe dos argumentos. El primero es un entero indicando cuántas palabras ha escrito el usuario en la línea de comandos (el propio nombre del programa cuenta, por lo que este argumento valdrá uno como mínimo). El segundo es un array de punteros, que contendrá tantos elementos como indique el primer argumento. Cada elemento de este array es un puntero a una cadena de caracteres, y cada una de estas cadenas son cada uno de los parámetros que el usuario tecleó en la línea de comandos. Así, el elemento 0 de este array siempre apunterá a una cadena que será la primera palabra tecleada en la línea de comandos, esto es, el propio nombre del programa.

Con este conocimiento no es muy difícil escribir un programa que vuelque por pantalla la lista de parámetros recibidos. Este programa se muestra en el siguiente listado:

```
1  // Acceso a los parametros suministrados desde linea de comandos
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6    int i;
7
8    printf("Se han recibido %d parámetros.\n", argc);
9    for (i=0; i<argc; i++)
10     printf("  Parámetro %d: |%s|\n", i, argv[i]);
11   return 1;
12 }
```

4.2. error.c

Este fichero muestra el uso de la salida de error en un programa en C. Todo programa en C puede generar mensajes que irán a parar a la *salida estándar* o a la *salida de error*, y en un buen diseño la salida estándar debería usarse sólo para los mensajes que el programa genera mientras todo va bien, y la salida de error para los mensajes de error.

Habitualmente ambas salidas están conectadas con la pantalla desde la cual se lanzó el programa, por lo que examinando lo que aparece en pantalla es imposible saber si el mensaje proviene de la salida estándar o la de error. Comprobemos esto ejecutando el programa *error*, una vez compilado. Este programa emite algunos mensajes de progreso a la salida estándar, y otros de error a la salida de error. Pero si lo ejecutamos así:

```
./error probando
Ejecutando programa...
Abriendo fichero..
Al abrir fichero: No such file or directory
Error número 29
```

vemos que aparecen juntos (ambos en la pantalla) los mensajes de progreso (terminados en puntos suspensivos) y los de error.

Sin embargo, si en el momento de lanzar el programa usamos un operador de *redirección*, entonces es posible separar ambas salidas. Usando el operador `1>`*`fichero`* podemos enviar a un *fichero* todo lo que va a la salida estándar, y con el operador `2>`*`fichero`* podemos enviar la salida de error. El operador `1>` también puede abreviarse simplemente como ">". Usando estos operadores podemos dejar visible sólo una de las salidas. Por ejemplo:

```
./error probando  > fichero
Al abrir fichero: No such file or directory
Error número 29
```

Los mensajes de progreso ya no son visibles en pantalla, sino que quedan almacenados en el fichero. En pantalla sólo son visibles los mensajes de error, ya que la salida de error no ha sido redireccionada. Análogamente podemos redireccionar sólo la salida de error, dejando la salida estándar conectada con la pantalla:

```
./error probando  2> errores
Ejecutando programa...
Abriendo fichero..
```

En este caso los mensajes de error ya no son visibles, sino que quedan almacenados en el fichero *errores*.

Finalmente podemos enviar la salida estándar a un fichero y la de error a otro, con lo que ya nada será visible en pantalla:

```
./error probando  > fichero 2> errores
```

Muchos programadores noveles desconocen la existencia de la salida de error y vuelcan los mensajes de error también en la salida estándar (puesto que ésta es la salida que resulta utilizada cuando se llama a `printf()`). Sin embargo esta forma de programar no es correcta, ya que después resultará imposible separar ambos tipos de mensaje usando operadores de redirección.

El programa `error.c` muestra cómo se pueden enviar mensajes a cada una de estas salidas:

- Para enviar algo a la salida estándar basta usar `printf()`.
- Para enviar algo a la salida de error, en cambio, debe usarse `fprintf()`, especificando como identificador de fichero la variable `stderr`. Esta variable está ya declarada e inicializada al incluir `stdio.h`
- La función `perror()` también vuelca mensajes de error en la salida de errores. Pero esta función, además del texto que le pasemos como parámetro, mostrará un mensaje (normalmente en inglés) con detalles acerca del error ocurrido. Estos detalles los obtiene de la variable global `errno`, y para tener acceso a ella necesitamos incluir el fichero de cabecera `errno.h`

Se incluye a continuación el listado de este programa

```
1  // Uso de la salida estandar de error
2  // Uso de la variable errno y de la funcion perror
3  // Este programa intenta abrir el fichero que se le pasa como argumento,
4  // y si no lo consigue informa del error
5  #include <errno.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  int main(int argc, char *argv[])
10 {
11   FILE *f;
12
13   printf("Ejecutando programa...\n");
14   if (argc<2) { // Si no se especifica argumento
15     fprintf(stderr, "Uso: %s fichero\n", argv[0]);
16     exit(-1);
17   }
18   printf("Abriendo fichero..\n");
19   f=fopen(argv[1], "r");
20   if (f==NULL) {
21     perror("Al abrir fichero");
22     fprintf(stderr, "Error número %d\n", errno);
23     exit(-1);
24   }
25   printf("Fichero abierto con éxito\n");
26   fclose(f);
27   return 0;
28 }
```

4.3. ficheros.c

Este programa muestra cómo abrir ficheros utilizando el tipo de datos `FILE*`, y cómo escribir en estos ficheros información, ya sea en forma ASCII legible (creando ficheros de texto), o en forma binaria, legible sólo para otros programas.

Los ficheros se abren con la función `fopen()`. Para escribir información legible, se usa la función `fprintf()`. Esta puede convertir cualquier tipo de dato simple (entero, real, cadena) a una secuencia de caracteres que quedará almacenada en el fichero. Si el fichero así generado se intenta abrir con un editor, se podrá leer el texto allí almacenado. Esta información puede recuperarse de nuevo desde programa haciendo uso de `fscanf`, que intentará "decodificar" las cadenas que va leyendo del fichero, convertirlas a un formato interno (entero, real,...) y asignar el resultado a variables del programa.

Esta forma de volcar datos a fichero hace que el fichero resultante no dependa de la *endianity* de la máquina que lo generó.

Sin embargo resulta más eficiente crear ficheros binarios. Este tipo de ficheros se abren de la misma forma, pero a la hora de volcar información en ellos, en lugar de `fprintf()` debe usarse `fwrite()`. Esta función se limita a volcar al fichero una serie de bytes, tal cual los encuentra en la memoria del computador. Por tanto el fichero así generado contiene los datos codificados según el esquema que haya usado la máquina que los generó. Para leer de nuevo estos datos debe usarse `fread()`. Los datos así recuperados sólo tendrán sentido si la máquina en que se leen tiene la misma arquitectura que la máquina en la que se generaron.

El programa `ficheros.c` se puede invocar de dos formas:

- **`./ficheros`** (sin argumentos). En este caso creará dos nuevos ficheros llamados `Datos.txt` y `Datos.dat`. Ambos contienen la misma información, si bien el primero en formato ASCII (legible) y el segundo en formato binario.
- **`./ficheros leer`**. En este caso intentará abrir los ficheros `Datos.txt` y `Datos.dat` (si no los encontrara daría un error), y leer la información allí almacenada para mostrarla por pantalla.

Resulta interesante generar los ficheros de datos llamando a "`ficheros`" sin parámetros, y seguidamente tratar de examinar los contenidos de los ficheros generados.

Con la utilidad `cat` se vuelca por pantalla el contenido de un fichero. Prueba a usar esta utilidad con `Datos.txt` y con `Datos.dat`, y observa la diferencia en el resultado en cada caso.

Con la utilidad `hexdump -C` *`fichero`* se puede examinar el contenido de un fichero binario. La salida de este comando muestra a la izquierda el valor de cada uno de los bytes, en hexadecimal, y a la derecha la interpretación ASCII de estos bytes, siempre que sea posible (hay bytes que no pueden ser intepretados como ASCII). Resulta instructivo usar esta herramienta para examinar qué contiene el fichero `Datos.dat`, y tratar de casar lo que estamos viendo con lo que el programa escribió (lo cual podemos averiguar mirando su código fuente). También resulta instructivo usar `hexdump` para examinar lo que contiene `Datos.txt`, visto como si fueran datos binarios.

Este programa también ejemplifica cómo crear ficheros y cómo detectar errores durante la creación o apertura. El listado se muestra a continuación:

```
1  // Ejemplo de escritura de fichero usando el tipo FILE*
2  //
3  // Cuando el programa se invoca sin argumentos, este crea
4  // el fichero "Datos.txt" con salida formateada en ASCII
5  // y el fichero "Datos.dat" con datos binarios.
6  //
7  // Cuando se le llama con un argumento (su valor se ignora), los
8  // ficheros anteriores son leidos (en modo texto o binario) y los
9  // valores obtenidos son mostrados.
10 //
11 #include <stdio.h>
12 #include <stdlib.h>
13 int main(int argc, char *argv[])
14 {
15   FILE *f_texto;  // Para el fichero formateado ASCII
16   FILE *f_data;   // Para el fichero binario
17   int x=10;
18   int y=47;
19   float g=1.0;
20   char txt[5]="Hola"; // 5 = 4 letras mas terminador
21
22   if (argc<2) { // No hay argumento, modo "crear ficheros"
23     // Abrir fichero de texto
24     f_texto=fopen("Datos.txt", "w");
25     if (f_texto==NULL) {
26       perror("Creando Datos.txt"); exit(-1);
27     }
28
29     // Abrir fichero binario. La letra "b" en el modo de apertura se
```

```
30      // ignora bajo UNIX, pero tiene su importancia bajo DOS/Windows
31      f_data=fopen("Datos.dat", "wb");
32      if (f_data==NULL) {
33        perror("Creando Datos.dat"); exit(-1);
34      }
35
36      // Escribimos los datos como ASCII en Datos.txt
37      fprintf(f_texto, "%s\n", txt);
38      fprintf(f_texto, "%d\n", x);
39      fprintf(f_texto, "%d\n", y);
40      fprintf(f_texto, "%f\n", g);
41      fclose(f_texto);
42
43      // Los escribimos como datos binarios en Datos.dat
44      fwrite(txt, sizeof(txt), 1, f_data);
45      fwrite(&x, sizeof(x), 1, f_data);
46      fwrite(&y, sizeof(y), 1, f_data);
47      fwrite(&g, sizeof(g), 1, f_data);
48      fclose(f_data);
49
50      printf("Ficheros Datos.txt y Datos.dat creados\n");
51      return 0;
52    }
53    else { // Si hay argumento, modo "leer ficheros"
54      f_texto=fopen("Datos.txt", "r");
55      if (f_texto==NULL) {
56        perror("Al abrir Datos.txt");  exit(-1);
57      }
58      f_data=fopen("Datos.dat", "rb");
59      if (f_texto==NULL) {
60        perror("Al abrir Datos.dat"); exit(-1);
61      }
62      // Leer datos ASCII de Datos.txt
63      fscanf(f_texto, "%s\n", txt);
64      fscanf(f_texto, "%d\n", &x);
65      fscanf(f_texto, "%d\n", &y);
66      fscanf(f_texto, "%f\n", &g);
67      fclose(f_texto);
68
69      printf("Datos leidos de Datos.txt\n");
70      printf(" txt=%s\n", txt);
71      printf(" x=%d\n", x);
72      printf(" y=%d\n", y);
73      printf(" g=%f\n", g);
74
75      // Leemos los datos binarios de Datos.dat
76      fread(txt, 5, 1, f_data);
77      fread(&x, sizeof(int), 1, f_data);
78      fread(&y, sizeof(int), 1, f_data);
79      fread(&g, sizeof(float), 1, f_data);
80      fclose(f_data);
81
82      printf("Datos leidos de Datos.dat\n");
83      printf(" txt=%s\n", txt);
84      printf(" x=%d\n", x);
85      printf(" y=%d\n", y);
86      printf(" g=%f\n", g);
87      return 0;
88    }
89  }
```

4.4. readwrite.c

Este programa muestra otra forma de crear ficheros y acceder a sus contenidos. Las funciones estudiadas en el ejemplo anterior (`ficheros.c`), son en realidad un *envoltorio* de alto nivel para hacer más sencillo el manejo de los ficheros. Pero estas funciones-envoltorio en realidad se limitan a llamar a otras de más bajo nivel que son las que realmente hacen el trabajo de leer y escribir en el disco. En este ejemplo se muestran estas segundas funciones.

Usando las funciones de bajo nivel para acceso a ficheros, los ficheros no son ya de tipo `FILE*`, sino de tipo `int`, llamado *descriptor del fichero*[5]

[5]En realidad este entero es un índice dentro de una tabla, y es esta tabla la que contiene la información necesaria para el acceso al disco. Hay una de estas tablas por cada proceso.

La función para abrir o crear ficheros usando esta API es `socket()`, la cual, además del nombre del fichero, requiere que se le pasen una serie de constantes especificando el "modo de apertura" (si vamos a leer o a escribir, y si debe crearse el fichero en caso de que no exista o no), y otra constante especificando los "permisos" con los que el archivo será creado, en caso de que haya que crearlo. Estos permisos indican qué usuarios podrán leer o modificar el archivo crado. Si no ponemos adecuadamente estos permisos puede darse el caso de que no podamos volver a crear ese fichero, por estar protegido contra escritura. Los valores que hay que poner a estas constantes pueden verse en el código fuente de ejemplo.

Para leer y escribir ficheros usando esta API, las únicas funciones disponibles son respectivamente `read()` y `write()`. Estas funciones sólo sirven para volcar al fichero o leer del mismo datos directamente en binario. En esta API no hay funciones para escribir o leer datos ASCII (es decir, no hay el equivalente a `fprintf()` o `fscanf()`).

Por ello, la creación de ficheros de texto usando esta API resulta muy incómoda. No obstante puede hacerse, como muestra el código de ejemplo, a base de llamar a `sprintf()` para crear previamente una cadena que contenga los datos ya formateados en ASCII, y seguidamente llamar a `write()` para volcar esta cadena al fichero. De forma análoga, podemos leer ficheros de texto cargando todos los bytes existententes en el fichero a un array, en una sola operación `read()`, para seguidamente "decodificar" el texto que hemos leído usando `sscanf()`, almacenando así el resultado en las variables del programa.

Pero donde esta API de bajo nivel es realmente útil es cuando queremos escribir o leer ficheros binarios. En este caso, las funciones `write()` y `read()` pueden usarse directamente como muestra el código fuente.

Otra ventaja de utilizar la API de bajo nivel, es que trata a los ficheros igual que a los sockets. Ya que tanto un fichero como un socket es referenciado mediante un simple `int` usando esta API, las funciones `read()`m `write()` y `close()` funcionan indistintamente sobre ficheros y sobre sockets[6].

```
1  // Ejemplo de acceso a ficheros usando las funciones de bajo nivel
2  // (open/read/write)
3  //
4  // La funcion es la misma que la del programa ficheros.c, de hecho,
5  // los ficheros creados con uno de ellos deberian ser legibles por el
6  // otro.
7  //
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <fcntl.h>
13 #include <unistd.h> // Para open/read/write en unix
14 #include <string.h>
15 int main(int argc, char *argv[])
16 {
17   int f_texto; // Ahora el tipo fichero es "int" !
18   int f_data;
19   int x=10;
20   int y=47;
21   float g=1.0;
22   char txt[5]="Hola"; // 5 = 4 letras mas terminador
23   char linea[80];     // Para convertir los datos a ASCII
24
25   if (argc<2) { // No hay argumento, modo "crear ficheros"
26     // Observar que en Unix, ademas de un modo de apertura se debe
27     // indicar los permisos de acceso con que se creara el fichero
28     // S_WRITE indica permiso de escritura para el propietario (el
29     // creador)
30     f_texto=open("Datos.txt", O_WRONLY|O_CREAT, S_IWRITE);
31     if (f_texto==-1) {
32       perror("Creando Datos.txt"); exit(-1);
33     }
34     // Al crear el fichero para datos binarios, bajo Windows/DOS,
35     // seria necesario añadir |O_BINARY al modo de apertura.
36     f_data=open("Datos.dat", O_WRONLY|O_CREAT, S_IWRITE);
37     if (f_data==-1) {
38       perror("Creando Datos.dat"); exit(-1);
39     }
40
41     // Escribimos los datos como ASCII en Datos.txt
```

[6] Sólo en linux. En Windows éstas funciones son sólo válidas para ficheros, pero no para sockets, ya que éstos son de otro tipo

```
42      // sprintf lo convierte en cadena de texto, y write lo escribe en
43      // el fichero. Observar el uso de strlen en lugar de sizeof.
44      sprintf(linea, "%s\n", txt); write(f_texto, linea, strlen(linea));
45      sprintf(linea, "%d\n", x);   write(f_texto, linea, strlen(linea));
46      sprintf(linea, "%d\n", y);   write(f_texto, linea, strlen(linea));
47      sprintf(linea, "%f\n", g);   write(f_texto, linea, strlen(linea));
48      close(f_texto);
49
50      // Ahora escribimos los mismos datos directamente en binario
51      write(f_data, txt, 5); // La cadena tiene 4 letras mas terminador
52      write(f_data, &x, sizeof(int));
53      write(f_data, &y, sizeof(int));
54      write(f_data, &g, sizeof(float));
55      close(f_data);
56
57      printf("Ficheros Datos.txt y Datos.dat creados\n");
58      return 0;
59    }
60    else { // Si hay argumento, modo "leer ficheros"
61      char buffer[200]; // Para leer el fichero .txt
62      // Nota: segun esta implementacion, el fichero no puede tener mas
63      // de 200 bytes
64      int n_bytes_leidos;
65
66      f_texto=open("Datos.txt", O_RDONLY);
67      if (f_texto==-1) {
68        perror("Al abrir Datos.txt");  exit(-1);
69      }
70      f_data=open("Datos.dat", O_RDONLY);
71      if (f_texto==-1) {
72        perror("Al abrir Datos.dat"); exit(-1);
73      }
74
75      // Leemos los datos ASCII de Datos.txt
76      // Leemos todo el fichero en un buffer, y extraemos la informacion
77      // de ese buffer usando sscanf.
78      n_bytes_leidos=read(f_texto, buffer, 200);
79      printf("Se han leido %d bytes del fichero Datos.txt\n",
80             n_bytes_leidos);
81      close(f_texto);
82      // Extraer datos del buffer, usando los tipos apropiados
83      sscanf(buffer, "%s\n%d\n%d\n%f\n", txt, &x, &y, &g);
84
85      printf("Datos leidos de Datos.txt\n");
86      printf(" txt=%s\n", txt);
87      printf(" x=%d\n", x);
88      printf(" y=%d\n", y);
89      printf(" g=%f\n", g);
90
91      // Leer los datos binarios de Datos.dat es mas sencillo
92      read(f_data, txt, 5);
93      read(f_data, &x, sizeof(int));
94      read(f_data, &y, sizeof(int));
95      read(f_data, &g, sizeof(float));
96      close(f_data);
97
98      printf("Datos leidos de Datos.dat\n");
99      printf(" txt=%s\n", txt);
100     printf(" x=%d\n", x);
101     printf(" y=%d\n", y);
102     printf(" g=%f\n", g);
103     return 0;
104   }
105 }
```

4.5. cadenas.c

Este programa muestra el uso de cadenas en C. El tipo de datos “cadena de caracteres”, o *string*, no es un tipo básico del C, sino que se implementa utilizando un *array* de caracteres. Es decir, una cadena es una zona de la memoria en la que se almacena una secuencia de bytes, cada uno de los cuales es el código ASCII de un caracter. La cadena debe estar terminada por un byte de valor cero, ya que las funciones de manejo de cadenas del C esperan este byte para saber dónde termina la cadena.

En el código de ejemplo se muestran tres posibles formas de inicializar una cadena:

```
char diez[10]="Holas";
char fija[]="Holas";
char *puntero="Holas";
```

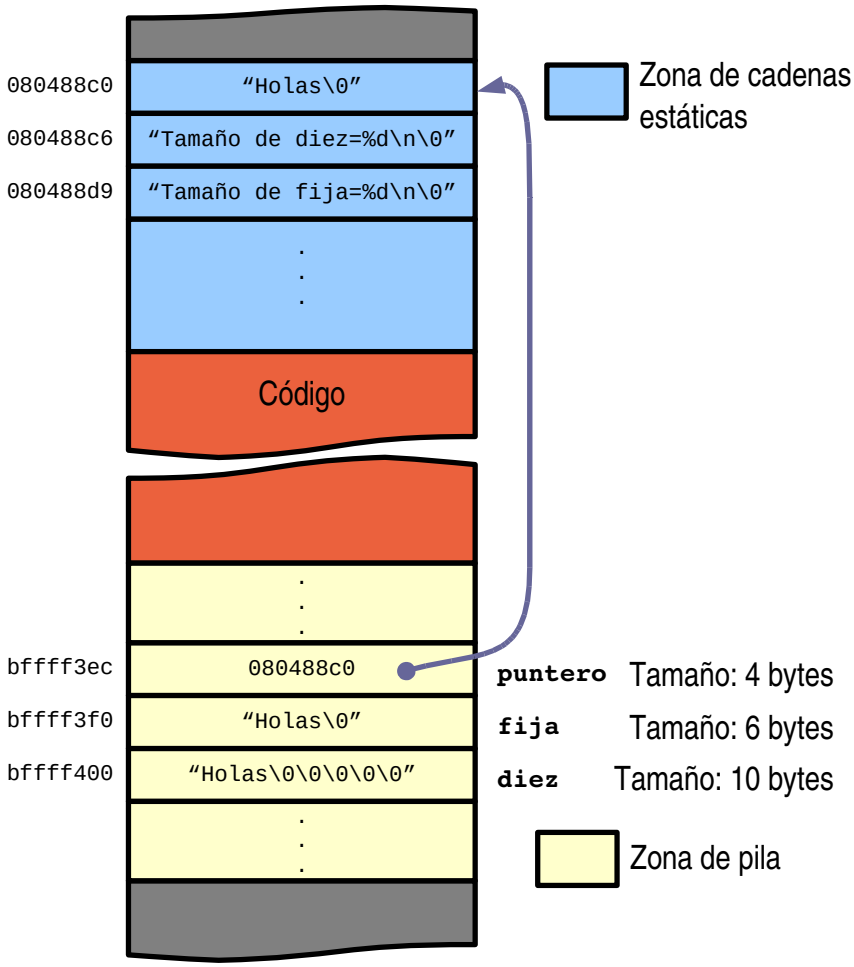
En la primera declaración se crea un array de 10 caracteres, y a la vez se inicializa con el texto "Holas". Este texto requiere 6 bytes (cinco para las letras que componen el texto y uno más para el byte de valor cero que actúa como terminador). Los restantes 4 caracteres del array quedan sin inicializar (probablemente contendrán ceros, pero no puede garantizarse).

La segunda declaración crea un array, pero no especificamos su tamaño sino que dejamos que lo calcule el compilador. Este caso sería equivalente a haber declarado un array de tamaño 6.

La última declaración es la más compleja. En este caso la cadena se declara como un puntero a `char`. Esto no reserva espacio para la cadena en sí, sino sólo para el puntero. Este puntero debe hacerse apuntar a una zona de memoria donde estará almacenada la cadena. Y esto es lo que se logra precisamente con la inicialización que acompaña a la declaración.

La cadena "Holas" en este caso no se almacena en la variable `puntero`, sino en una zona especial de la memoria denominada "la zona de las cadenas estáticas". En esta zona se almacenan todas las cadenas literales que usa nuestro programa, que son todas aquellas que en el código fuente aparecen entre comillas dobles. La variable `puntero` es inicializada de forma que apunta a esa zona de la memoria, en concreto a la posición donde está almacenada la cadena "Holas".

Es importante resaltar que en este tercer caso la variable `puntero`, al ser local, está almacenada en la pila, pero sin embargo el texto "Hola" no está almacenado en la pila, sino en la zona de cadenas estáticas. Esto se muestra en la figura siguiente:



El código fuente muestra que para averiguar cuántas letras tiene una cadena debe usarse la función `strlen()`, y no `sizeof()`, ya que ésta última nos da el tamaño de la variable, lo que no siempre coincide con la longitud del texto[7]. También se muestra que para acceder a una letra

[7] En particular, si la cadena se accede mediante un `char *`, el tamaño dado por `sizeof()` será siempre 4, ya que este es el tamaño de un puntero, que no depende de los datos a los que apunta

individual de la cadena se puede usar la sintaxis de array, incluso para cadenas declaradas como `char *`

Seguidamente el programa intenta añadir texto a estas cadenas. La función `strcat()` sirve para este cometido: concatenar cadenas. Pero hay que tener en cuenta que esta función no comprueba si el resultado de la concatenación cabe en el lugar destinado para ello. En concreto, se limita a recorrer la primera cadena que recibe como parámetro hasta encontrar su final (el byte nulo), y seguidamente copiar ahí mismo la cadena que recibe como segundo parámetro.

Como consecuencia, al intentar añadir texto a la variable `fija`, que tiene espacio tan solo para seis letras, el texto añadido sobreescribirá otras variables del programa (en concreto sobreescribe la variable `diez`, como se comprende observando la figura anterior). Por eso el programa produce un resultado inesperado.

Igualmente es incorrecto usar `strcat()` para añadir texto a una cadena estática, como se intenta hacer a continuación con la variable `puntero`. En este caso, el texto añadido que tampoco cabe en la cadena original, sobreescribirá otras cadenas estáticas. El resultado puede ser aún más chocante, ya que esto significa que otras cadenas del programa pueden resultar alteradas, incluso aunque no estén apuntadas por ninguna variable (por ejemplo, los textos utilizados en los `printf()`). Por esto, muchos operativos protegen la zona de memoria donde están almacenadas las cadenas estáticas, y por tanto un intento de modificarlas producirá un error de "Violación de segmento"[8]

El programa muestra seguidamente la forma correcta de añadir texto a una cadena. Ésta consiste en reservar espacio suficiente para la nueva cadena (la que resultará de ampliar la cadena original), usando la función `malloc()`. En el espacio reservado copiamos la cadena original usando `strcpy()`, y seguidamente "aumentamos" esta cadena usando la función `strcat()`.

A continuación se incluye el código fuente de este programa.

```
1  // Experimentos con cadenas (strings) en C
2  #include <stdio.h>
3  #include <string.h>
4  #include <malloc.h>
5
6  int main()
7  {
8    // Una cadena se puede guardar en un array de caracteres, o en una
9    // zona de memoria a la que se hace apuntar un puntero a caracteres.
10
11   char diez[10]="Holas"; // Reservamos 10 caracteres, aunque usamos
12                          // solo 6 (letras mas terminador)
13   char fija[]="Holas";   // El compliador calcula aqui cuantos bytes
14                          // reservar (seran 6)
15   char *puntero="Holas"; // Aqui la cadena se almacena en el area de
16                          // "strings estaticos", y se hace que el
17                          // puntero apunte a ella
18
19   // Sorpresa?
20   printf("Tamaño de diez=%d\n", sizeof(diez));
21   printf("Tamaño de fija=%d\n", sizeof(fija));
22   printf("Tamaño de puntero=%d\n", sizeof(puntero)); // !Eepa!
23
24   // Aqui no hay sorpresa
25   printf("Cadena diez.    Longitud=%d, contenido=|%s|\n",
26          strlen(diez), diez);
27   printf("Cadena fija.    Longitud=%d, contenido=|%s|\n",
28          strlen(fija), fija);
29   printf("Cadena puntero. Longitud=%d, contenido=|%s|\n",
30          strlen(puntero), puntero);
31
32   // La sintaxis para accedera a una letra cualquiera es la misma en
33   // los tres casos.
34   printf("Tercera letra: diez=%c, fija=%c, puntero=%c\n",
35          diez[2], fija[2], puntero[2]);
36
37   // La zona de memoria donde se almacenan los arrays es la misma (la
38   // pila de programa durante la ejecucion de main). En cambio la
39   // cadena estatica va a otra zona (la zona de strings estaticos)
40   printf("Direcciones de memoria donde se almacena el texto:\n");
41   printf("  diez   =%p\n", diez);
42   printf("  fija   =%p\n", fija);
43   printf("  puntero=%p\n", puntero);
44   printf(" &puntero=%p\n", &puntero); // Observar que el puntero se
```

[8] Si pese a todo queremos modificarlas, podemos compilar con la opción `-fwritable-strings`, para solicitar que esa zona de la memoria no esté protegida

```
45    // almacena en la pila, al ser una variable de main, aunque la
46    // direccion a que apunta no esta en la pila.
47
48    // Modificaciones de las cadenas.
49    printf("Añadir texto a la cadena fija [error grave]\n");
50    strcat(fija, ", caracolas");
51    printf("Resultado:\n");  // Inesperado?
52    printf(" Cadena fija.    Longitud=%d, contenido=|%s|\n",
53           strlen(fija), fija);
54    printf(" Cadena diez.    Longitud=%d, contenido=|%s|\n",
55           strlen(diez), diez);
56
57    printf("Añadir texto a la cadena puntero [mal hecho]\n");
58    strcat(puntero, ", caracolas");
59    printf("Resultado:\n");
60    printf(" Cadena puntero. Longitud=%d, contenido=|%s|\n",
61           strlen(puntero), puntero);
62    // Lo anterior puede dar un core o no, depende de la maquina
63
64    printf("Añadir texto a la cadena puntero [bien hecho]\n");
65    puntero=(char *) malloc(strlen(puntero)+strlen(", caracolas")+1);
66    strcpy(puntero, "Holas");
67    strcat(puntero, ", caracolas");
68    printf("Resultado:\n");
69    printf(" Cadena puntero. Longitud=%d, contenido=|%s|\n",
70           strlen(puntero), puntero);
71    // Aqui no hay problema. Es la forma correcta de hacerlo. Pero ahora
72    // el puntero ya no apunta a la zona de strings estaticos, sino al
73    // "heap" (la zona de memoria sobre la que opera malloc/free). La
74    // cadena original "Holas", sigue intacta en la zona de stings
75    // estaticos. El puntero a ella se ha perdido para siempre.
76
77    return 0;
78  }
```

4.6. Compilación separada

El último ejemplo muestra cómo dividir el código fuente de un programa en varios ficheros, cómo compilar éstos por separado, y cómo juntarlos finalmente para crear un único ejecutable.

La ventaja de realizar esta compilación separada se hace más patente cuando el fuente es muy extenso. Al separarlo en varios ficheros fuente se hace más cómoda su edición y la localización de las diferentes funciones que lo componen, y al compilarlos por separado se hace más corto el tiempo de compilación, ya que cuando realicemos un cambio sólo será necesario recompilar el fuente modificado y no todos de nuevo.

El ejemplo que se presenta, sin embargo, es muy corto y estas ventajas no se aprecian. En realidad, se usa sólo como un ejemplo mínimo que el alumno puede fácilmente extrapolar a casos de mayor envergadura.

El programa propuesto simplemente pide un par de números reales al usuario y sgeuidamente llama a dos funciones para realizar ciertas operaciones con ellos, tras lo que muestra el resultado y termina. El código de las funciones en cuestión está separado en otros dos ficheros. El total de ficheros fuente es tres:

`principal.c` Contiene el código fuente de la función `main()`.

`multiplica.c` Contiene el código fuente de la función `multiplica()`

`divide.c` Contiene el código fuente de la función `divide()`.

Ya que el programa principal no contiene el código de las funciones a las que invoca (porque están en otros ficheros), cuando compilemos este programa el compilador podrá quejarse de que esas funciones no han sido declaradas (si compilamos con la opción `-Wall`). Es una queja justa. Las funciones deberían declararse antes de usarse[9]. Es por esto que escribimos la declaración de las funciones en otros ficheros, para incluirlos en el programa principal mediante la directiva `#include`.

[9] Si se usa una función que no ha sido declarada, el compilador crea una declaración "sobre la marcha", intentando adivinar cómo sería la declaración que no hemos puesto. Para ello se basa en los tipos de los parámetros que hemos usado al llamarla. Muchas veces la intuición del compilador es correcta, pero otras no, por lo que siempre conviene tener la función correctamente declarada antes de llamarla por primera vez. De hecho, en este ejemplo, si faltaran los `#include`, el resultado de llamar a las funciones saldría mal. Puedes probarlo.

multiplica.h Contiene la declaración de la función `multiplica()`, aunque no su código (que está en `multiplica.c`).

divide.h Contiene la declaración de la función `divide()`, aunque no su código (que está en `divide.c`).

Si incluimos estos ficheros en el programa `principal.c`, haciendo uso de la directiva `#include`, el compilador encontrará las declaraciones de las funciones antes de ser llamadas, por lo que ya no protestará. Uno podría preguntarse por qué no incluir directamente los ficheros `multiplica.c` y `divide.c` dentro de `principal.c`, haciendo uso también de la directiva `#include`. Esto podría hacerse, pero entonces se perdería la posibilidad de la compilación separada. El incluir con `#include` un fichero `.c` es completamente equivalente a teclear allí mismo el código. Al compilar el programa principal se compilaría también todo el código de estas funciones. Sin embargo al incluir sólo el `.h`, el único código compilado es el de la función `main()`, y no el de las otras funciones. Esto puede tener su importancia si las otras funciones fuesen muy largas y se tardase mucho en compilarlas.

Para compilar por separado cada uno de los ficheros en `.c`, basta darle al compilador la opción `-c`. De este modo:

```
$ gcc -Wall -c principal.c
$ gcc -Wall -c divide.c
$ gcc -Wall -c multiplica.c
```

Cada uno de los comandos anteriores compila sólo uno de los fuentes, y produce como resultado un "codigo objeto", que es un fichero de igual nombre que el fuente, pero de extensión `.o`, que contiene la traducción a código máquina del fuente.

Un código objeto no puede ejecutarse, porque aún está incompleto. Falta añadir a él el código de todas las funciones que son usadas en el programa, y las cabeceras necesarias para que Linux lo reconozca como un fichero ejecutable. Este último paso se denomina *enlazado* o *linking* y se realiza con el comando `gcc`, pero pasándole los nombres de todos los ficheros objeto que hay que juntar, en lugar de los fuentes en C. Es decir:

```
$ gcc -o resultado principal.o divide.o multiplica.o
```

En este caso hemos decidido llamar "`resultado`" al ejecutable. Si pruebas a omitir alguno de los `.o` de la línea de comandos, el *linker* se quejará de que no encuentra alguna función (la que estaba en ese `.o`).

La ventaja de la compilación separada es que si ahora modificamos, por ejemplo, la función `main()`, pero no modificamos ninguna de las otras, entonces no es necesario volver a compilar los fuentes que no hayamos modificado. Bastaría hacer tan solo:

```
$ gcc -c principal.c
$ gcc -o resultado principal.o divide.o multiplica.o
```

para tener el ejecutable actualizado. Si los fuentes son de gran tamaño esto es mucho más rápido que volver a compilarlos todos, lo que ocurriría si no los tuvieramos separados en diferentes ficheros `.c` o si, teniéndolos separados, cometiéramos el error de juntarlos todos otra vez usando `#include` con los `.c` en lugar de los `.h`.

Seguidamente se muestra el código de estos ficheros:

4.6.1. principal.c

```
1  // Ejemplo de compilación separada. Programa principal.
2  #include <stdio.h>
3  // Los includes siguientes son necesarios para que el compilador sepa
4  // los prototipos de las funciones, y por tanto las conversiones de
5  // tipo que debe hacer (si fueran necesarias).
6  // ¡Prueba a quitar estas líneas!
7  #include "multiplica.h"
8  #include "divide.h"
9
10 int main()
11 {
12   double x, y;
13   double resultado;
14
```

```
15    printf("Introduce un par de números reales:");
16    scanf("%lf %lf", &x, &y);  // %lf es para doubles
17    resultado=multiplica(x,y);
18    printf("Su producto es %lf\n", resultado);
19    resultado=divide(x,y);
20    printf("Su cociente es %lf\n", resultado);
21    return 0;
22  }
```

4.6.2. multiplica.h

```
1  // Fichero de cabecera que proporciona el prototipo de la función
2  // multiplica()
3  #ifndef _MULTIPLICA_H_
4  #define _MULTIPLICA_H_
5
6  double multiplica(double, double);
7  #endif
```

4.6.3. multiplica.c

```
1  // Implementación de la función multiplica()
2  double multiplica(double a, double b)
3  {
4    return a*b;
5  }
```

4.6.4. divide.h

```
1  // Fichero de cabecera que proporciona el prototipo de la función
2  // divide()
3  #ifndef _DIVIDE_H_
4  #define _DIVIDE_H_
5
6  double divide(double, double);
7  #endif
```

4.6.5. divide.c

```
1  // Implementación de la función divide()
2  double divide(double a, double b)
3  {
4    return a/b;
5  }
```

Índice