Introducción al entorno de programación UNIX.

Extraido del libro "Curso de programación en C bajo UNIX", de Diego Rafael Llanos Ferraris.
Reproducido por Rubén Díez Lázaro con permiso de la editorial Paraninfo.

Mayo 2002

Índice

1. El compilador de C: cc

1.1. Introducción

En esta sección se explicará cómo se utiliza el compilador de línea, llamado cc, presente en la inmensa mayoría de los sistemas Unix. Se dice que cc es un compilador de línea porque no dispone de un entorno de desarrollo integrado: en su lugar, el programador debe escribir el código en C utilizando un editor de texto (en nuestro caso, el vi), y luego compilarlo. Para ello debe invocarse al compilador, indicándole, entre otras cosas, el nombre del fichero con el código fuente, el nombre donde se almacenará el código ejecutable, etcétera. En la siguiente sección veremos su utilización básica, y luego hablaremos de algunas de las posibilidades avanzadas que ofrece.

utilizan funciones

1.2. Compilación básica con cc

Supongamos que tenemos un fichero denominado *miprog.c*. Para compilarlo, basta con hacer

```
cc miprog.c
```

Esta instrucción genera un fichero ejecutable, de nombre *a.out*. Para ejecutar este programa, basta con invocarlo directamente:[1]

```
a.out
```

Si queremos dar un nombre distinto de *a.out* al fichero ejecutable, podemos indicarlo con la opción -o:

```
cc miprog.c –o miprog
```

De esta forma, el fichero generado a la salida se llamará *miprog*. Cabe añadir otra opción que es imprescindible en algunos casos. Si nuestro programa utiliza la biblioteca de funciones matemáticas (es decir, si hemos utilizado funciones que aparecen declaradas en el fichero de cabecera *math.h*), debemos indicar al compilador que enlace nuestro programa con dicha biblioteca. Esto se consigue con la opción *-lm*:

```
cc miprog.c –o miprog –lm
```

En la sección 1.6 volveremos sobre el tema del enlace con bibliotecas de funciones.

Para acabar con el conjunto de opciones básicas de compilación, veremos una instrucción del entorno Unix que resulta muy útil si aparecen muchos mensajes de error al compilar nuestro programa. Si el número de mensajes de error es mayor que el número de líneas que tiene nuestro monitor, los mensajes que aparecieron en primer lugar se salen de la pantalla. Para almacenarlos en un fichero y poder luego leerlos con detenimiento, debemos redirigir la salida de error estándar. Esto se consigue incluyendo al final de la orden de compilación la orden *2> fichero.txt*, donde fichero.txt es el fichero donde queremos almacenar los errores. Así, la compilación de nuestro programa quedará como

```
cc miprog.c –o miprog 2> errores.txt
```

pudiéndose luego ver los errores con *more errores.txt* o editar el fichero con *vi errores.txt*.

[1] ***NOTA:*** *En muchos sistemas Unix, por motivos de seguridad, no se incluye el directorio actual dentro de los directorios que pueden contener ejecutables: en ese caso, hay que indicar que el fichero a ejecutar se encuentra en el directorio actual, precediendo su nombre de los símbolos "./". Así la orden de ejecución será ./a.out.*

1.3. Compilación según el estándar ANSI con cc

Como ya hemos comentado a lo largo del libro, existen diferentes versiones del lenguaje C, cada una con diferentes características. Esto se debe a que cada fabricante posee una versión propia del lenguaje. De todas las versiones del lenguaje, el estándar **ANSI C** es el más ampliamente utilizado, y es el que venimos usando a lo largo del libro. Para asegurarnos de que el compilador sepa que nuestro programa está escrito en **ANSI C**, debemos indicarlo explícitamente a través de una opción de la línea de comandos.

Sucede, sin embargo, que la opción que permite compilar en **ANSI C** cambia de un compilador a otro. Por lo tanto, se hace necesario saber en qué versión de Unix estamos trabajando para utilizar la opción correcta. La opción de compilación *ANSI* de los principales compiladores es la siguiente:

- En el compilador de GNU (Linux): *-ansi*.
- En el compilador de Hewlett-Packard (HP-UX): *-Aa*.
- En el compilador de Sun (Sun OS, Solaris): *-Xa*.

1.4. Compilación por etapas con cc

Como dijimos en el capítulo anterior, las etapas de compilación de un programa en C incluyen tres partes fundamentales

1. El preproceso.
2. La compilación.
3. El enlace.

El compilador cc, por defecto, ejecuta las tres etapas sobre nuestro fichero, generando un fichero de código ejecutable a la salida. Sin embargo, es posible indicarle que sólo deseamos que realice una de ellas.

El preproceso

El preproceso es la etapa en la que se interpretan y expanden las directivas y macros que aparecen en el programa, además de quitarse los comentarios. Si querernos aplicar únicamente el preproceso a nuestro programa, podemos conseguirlo utilizando La opción *-E*. Si ejecutamos

```
cc -E miprog.c
```

el resultado de la etapa de preproceso saldrá por pantalla, no generándose ningún fichero de salida. Si queremos ver el resultado de esta etapa, podemos redirigir la salida a un fichero:

```
cc -E miprog.c > salida.txt
```

La compilación

La etapa de compilación, propiamente dicha, recibe como entrada un fichero fuente (con extensión .c) y devuelve a su salida un fichero objeto (con el mismo nombre, pero extensión .o). Estos ficheros objetos se unirán en el enlace para formar el fichero con el código ejecutable.

Para generar un fichero objeto, sin generar el fichero ejecutable, debemos utilizar la opción *-c*:

```
cc -c miprog.c
```

Esta orden se encargará de preprocesar el fichero y compilarlo. Si no se han producido errores, se habrá generado un fichero objeto *miprog.o*. No tiene demasiado sentido utilizar la opción *-c* cuando nuestro programa se compone únicamente de un fichero fuente, pero como veremos en las próximas secciones, esta opción resulta muy útil cuando nuestro programa se compone de varios ficheros en código fuente. En este caso, compilarlos separadamente nos permite ir buscando errores en cada uno de los ficheros, sin necesidad de compilar todos los demás.

El enlace

Para enlazar un conjunto de ficheros objeto para formar un único ejecutable, basta con indicar a cc el nombre de los ficheros objeto y el nombre que se desea que reciba el ejecutable, con la opción *-o*:

```
cc miprog.o -o miprog
```

El fichero *miprog.o* deberá haber sido creado anteriormente con una llamada al compilador con la opción -c (ver apartado anterior). Si no especifica mos el nombre del ejecutable a crear, se utilizará por defecto el nombre *a.out*.

1.5. Compilación de un programa con varios módulos

Es conveniente conveniencia de dividir un programa de gran tamaño en varios módulos más pequeños, al objeto de agrupar funciones que realicen tareas similares. Cuando el código fuente de un programa en C está dividido en varios ficheros con extensión *.c*, tendremos que compilarlos y enlazarlos todos juntos para crear el programa ejecutable.

Supongamos que tenemos el código fuente de nuestro programa, de nombre *prog*, en tres ficheros, de nombre *prog_1.c*, *prog_2.c* y *prog_3.c*. Para compilar todos ellos y enlazarlos en un único ejecutable tenemos que ejecutar las órdenes

```
cc -c prog_.c prog_2.c prog_3.c -ansi
cc prog_1.o prog_2.o prog_3.o -o prog
```

La primera orden se encarga de preprocesar y compilar por separado cada uno de los ficheros de código C indicados (la opción *-ansi* asegura que se interpretarán según el estándar **ANSI**: ver sección 1.3). La segunda orden enlaza todos los ficheros objeto, generando el programa *prog*.

La compilación manual de un programa que consta de más de tres módulos es una tarea pesada, ya que permanentemente hay que recordar qué módulos son los que hay que recompilar, al haber sido modificados, y cuáles no han sufrido ninguna modificación y pueden enlazarse directamente. Por ejemplo, si en el caso anterior modificamos sólo *prog_2.c*, las órdenes para volver a generar el fichero ejecutable serían:

```
cc -c prog_2.c -ansi
cc prog_1.o prog_2.o prog_3.o -o prog
```

Si recompilamos un módulo ya compilado con anterioridad, no pasa nada: el problema aparece cuando nos olvidamos de recompilar un módulo antes de la fase de enlace. La herramienta *make* (ver sección 2) automatiza esta tarea.

1.6. Utilización de bibliotecas de funciones con cc

Muchas veces es necesario enlazar nuestro programa con bibliotecas de funciones. Dichas bibliotecas pueden ser estándar –como la biblioteca de funciones matemáticas comentada en la sección 1.2– o bien creadas por nosotros mismos con la herramienta *ar* (descrita en la sección 3).

Para incluir esas bibliotecas en el enlace de nuestro programa, debemos indicarle al compilador dos cosas: cómo se llaman y dónde están. Los nombres de las bibliotecas de funciones son siempre de la forma *libnombre.a*, donde *nombre* es el nombre de la biblioteca. Por ejemplo, la biblioteca estándar de funciones matemáticas tiene el nombre *m*, por lo que el fichero de biblioteca será *libm.a*.

Para indicarle al compilador cómo se llama la biblioteca a utilizar, debemos utilizar la opción *-l* seguida inmediatamente del nombre, según hemos indicado más arriba. Así, para incluir la biblioteca de funciones matemáticas, la opción a utilizar será *-lm*.

Si la biblioteca a incluir es estándar, no hace falta indicarle al compilador dónde debe buscarla. Sin embargo, si la biblioteca a incluir es nuestra, tenemos que indicarle al compilador, además del nombre, dónde se encuentra. Para ello disponemos de la opción *-L*, que debemos utilizar seguida del directorio donde el compilador debe buscar las bibliotecas (además de buscarlas en los sitios habituales). Si queremos indicar que busque en dos directorios adicionales en lugar de en uno, basta con utilizar la opción *-L* dos veces, indicando ambos directorios.

Por ejemplo, supongamos que hemos hecho una biblioteca de funciones para cálculo estadístico, y la denominamos *libstat.a*. Supongamos además que dicha biblioteca se encuentra en el directorio *lib*, que cuelga de nuestro directorio de trabajo (es decir, es un subdirectorio del directorio donde se encuentra nuestro código fuente). De esta forma, para compilar nuestro programa *prog.c* y enlazarlo con esta biblioteca la orden será

```
cc prog.c –o prog –L./lib –lstat
```

Nótese que *-L* contiene el directorio */lib* precedido de los símbolos *./*, que indican el directorio actual. Si sólo hubiéramos puesto *-Llib/*, el compilador habría buscado en el directorio */lib*, que cuelga del directorio raíz.

1.7. utilización de ficheros de cabecera propios

La utilización de ficheros de cabecera propios es similar a la utilización de bibliotecas, aunque tiene algunas diferencias. De nuevo tenemos que indicarle al compilador dos cosas: dónde están y cómo se llaman.

El primer problema se resuelve con el parámetro *-I*, seguido del nombre del directorio donde se encuentran. Así, *-I./include* hace que el compilador busque ficheros de cabecera en el directorio *include* (que será un subdirectorio del directorio actual), además de buscarlos en los lugares habituales.

Para indicarle al compilador cómo se llaman los ficheros de cabecera a incluir en la compilación, tenemos la directiva *#include*. Esta directiva admite dos posibilidades: encerrar el nombre del fichero de cabecera entre los símbolos $<$ y $>$, como

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

que indica que ese fichero de cabecera está en los directorios estándar (o en un directorio adicional especificado con la opción *-I*, comentada más arriba). La segunda posibilidad es encerrar el nombre entre comillas dobles:

```
#include "./include/micabecera.h"
```

que indica que el fichero *micabecera.h* se encuentra en el subdirectorio *include*, que cuelga del directorio actual (por lo que no hace falta indicar dicho subdirectorio con la opción *-I*).

En la sección 5.2 se trata con detalle el uso de la directiva *include*.

1.8. Otras opciones de cc

El compilador *cc*, pese a tener un funcionamiento equivalente en cualquier sistema Unix, no siempre posee las mismas opciones en todos ellos. Entre las opciones adicionales que ofrece figura la inclusión en los ficheros de salida de información de depuración (para ser utilizada con un depurador de código), creación de ejecutables optimizados para una máquina concreta, posibilidad de inclusión de código ensamblador, etcétera. Remitimos al lector a la consulta de las páginas *man* de su compilador concreto para más detalles.

2. La herramienta make

Es una práctica común dividir grandes programas en pequeñas partes más manejables. Muchas veces, se desarrolla un programa en módulos distintos, agrupando en un mismo módulo el código que necesita un tratamiento concreto [2] Cada uno de estos módulos suele compilarse con opciones especiales y con determinadas definiciones y declaraciones. Para generar el fichero ejecutable, el código objeto resultante se enlaza con ciertas bibliotecas bajo el control de opciones especiales. Desgraciadamente, cuando un programa consta de más de dos o tres módulos, es muy fácil que un programador olvide qué archivos dependen de cuáles otros, cuáles han sido modificados recientemente, y la secuencia exacta de operaciones necesarias para generar una nueva versión del sistema. Es fácil olvidar, a la hora de compilar, qué ficheros objetos han sufrido cambios en sus dependencias y cuáles siguen siendo válidos, ya que un cambio en una declaración de un fichero puede invalidar una docena de archivos. Esto genera errores muy difíciles de detectar. Por otra parte, recompilar todo cada vez que se modifica algo sólo por asegurarse es una pérdida de tiempo.

La herramienta *make* permite automatizar muchas de las actividades realizadas en el desarrollo y mantenimiento de un programa. Si se almacena en un archivo la información de dependencias y secuencias de comando necesarias, cuando se desee generar una nueva versión ejecutable del programa, *make* se encargará de recompilar sólo los ficheros necesarios, independientemente de cuántos hayan sido modificados. Además, el fichero de descripción es fácil de escribir y no cambia frecuentemente.

make aparece en los sistemas Unix y en muchos compiladores comerciales de C para otros entornos, como MS-DOS o Windows. La mayor parte de las implementaciones de *make* utilizan los mismos formatos básicos. De todos modos, recomendamos consultar el manual de la versión concreta que se utilice.

2.1. ¿Cómo trabaja make?

Como hemos dicho, *make* proporciona un mecanismo sencillo para mantener versiones actualizadas de programas generados a partir de un cierta número de ficheros. Es posible indicarle a *make* la secuencia de comandos necesaria para crear ciertos archivos, así como la dependencia de algunos archivos respecto de otros. Cuando se realice un cambio en alguna parte del código fuente, *make* reprocesará sólo los archivos que lo necesiten.

[2] ***NOTA:*** *Además, cada módulo puede estar escrito en un lenguaje diferente, no necesariamente en C. Por ejemplo, uno puede necesitar escribir el código que genere un analizador sintáctico en el lenguaje utilizado por Yacc, y agrupar el resto de código en ficheros en C y Fortran.*

Lo que hace *make* es buscar el nombre del fichero final en la descripción del proyecto que se le suministra, asegurándose que todos los archivos de los que depende existan y están actualizados. Si estos archivos han sido modificados y el fichero final no, se encarga de crear este último.[3]

2.2. utilización básica de make

Consideremos el siguiente ejemplo. Supongamos que tenemos que compilar un programa denominado *prog*, cuyo código fuente tiene las siguientes características:

- El programa *prog* se construye compilando y enlazando tres ficheros fuente en C, de nombres *prog_1.c*, *prog_2.c* y *prog_3.c* (por convención, la salida obtenida al compilar estos programas bajo Unix será *prog_1.o*, *prog_2.o* y *prog_3.o*).
- Para crear el programa ejecutable tenemos que enlazar los tres ficheros objeto con la biblioteca *lS*.
- Los ficheros *prog_1.c* y *prog_2.c* incluyen el fichero *defin.h*, pero *prog_3.c* no. Por lo tanto, sólo *prog_1.c* y *prog_2.c* contendrán la línea

  ```
  #include defin.h
  ```

El fichero que describe las relaciones y operaciones entre los ficheros se denomina *makefile* o *Makefile* (se recomienda escribirlo con la M mayúscula para que aparezca al principio del directorio en el entorno Unix). El *Makefile* para nuestra caso sería el siguiente:

```
prog: prog_1.o prog_2.o prog_3.o
cc prog_1.o prog_2.o prog_3.o -ls -o prog
prog_1.o prog_2.o:      defin.h
```

Si este archivo se almacena en el directorio de trabajo, bajo el nombre de *Makefile*, el comando

```
make
```

realizará las operaciones necesarias para recompilar *prog* después de que realicemos cualquier cambio sobre los archivos *prog_1.c*, *prog_2.c*, *prog_3.c* o *defin.h*. Si el fichero *Makefile* se almacena con otro nombre, debe invocarse a *make* con la forma

```
make -f nombre_fichero
```

make trabaja con tres fuentes de información: un fichero de descripción suministrado por el usuario (coma el de arriba), los nombres de los archivos y las horas de sus últimas modificaciones (suministrados par el sistema operativo), y ciertas reglas ya incorporadas para realizar las acciones necesarias.

Por ejemplo, la primera línea de nuestro *Makefile* indica que *prog* depende de tres ficheros con extensión *.o*. Una vez que esos ficheros están disponibles, la segunda línea indica cómo enlazarlos para crear *prog*. La tercera línea dice que *prog_1.o* y *prog_2.o* dependen del fichero *defin.h*. *make* dispone de un conjunto de reglas internas que le permiten deducir que los tres ficheros con extensión

[3] ***NOTA:*** *Lo que en realidad define el fichero de descripción es el grafo de dependencias: make hace una búsqueda en profundidad de este grafo para determinar el trabajo que es necesario realizar.*

.o deberán generarse a partir de tres ficheros con el mismo nombre y extensión *.c* que se encuentran en el directorio actual. Además, *make* sabe cómo crear un fichero objeto a partir de un fichero fuente en C (a través de un *cc -c*).

Si, por ejemplo, el fichero *defin.h* fuera modificado, *make* se encargará de recompilar los ficheros de código fuente en C *prog_1.c* y *prog_2.c* (pero no *prog_3.c*), y *prog* se creará a partir de los tres ficheros con extensión *.o*.

Si no se indica el fichero que se desea generar, el primer fichero mencionado en la descripción es el que se crea; sin embargo, la orden

```
make prog_1.o
```

recompilaría *prog_1.o* si *prog_1.o* o *defin.h* hubieran cambiado. A veces, es útil incluir reglas con nombres mnemotécnicos que realicen ciertas tareas. Por ejemplo, una entrada *limpiar* podría utilizarse para eliminar ficheros intermedios:

```
limpiar: rm -f *.bak
```

Para ejecutarla, bastará con hacer

```
make limpiar
```

2.3. Utilización de macros con make

make tiene un mecanismo de macros sencillo que se utiliza para substituir elementos en líneas de dependencia y comandos. Una macro (o variable) se define indicando el nombre de la macro, seguida de un signo igual y, opcionalmente, el valor que adopta. Definiciones de macros válidas son las siguientes:

```
2 = xyz
LIBS = -lS -ll
FLAGS =
```

En el último caso, se asigna a *FLAGS* la cadena nula.

Una macro se invoca precediendo el nombre por un signo *$*; los nombres de las macros mayor de un carácter deben encerrarse entre paréntesis. Las siguientes son invocaciones de macros válidas:

```
$ (FLAGS)
$2
$(cc)
```

¿Cómo se representa literalmente el signo *$*? Como *$$*. Todas las macros reciben valores de entrada, como puede verse en el siguiente *Makefile*:

```
OBJECTS = prog_1.o prog_2.o prog_3.o
LIBS    = -lS
prog:   $(OBJECTS)
     cc $(OBJECTS) $(LIBS) -o prog
...
```

Por otro lado, el comando

```
make "LIBS = -ll -lS"
```

enlaza los tres objetos con las bibliotecas de lex (*-ll*) y la estándar (*-lS*), ya que las definiciones de macros de la línea de comandos sobreescriben las del *Makefile*.

Hay un par de macros especiales, que son definidas por *make* antes de comenzar. Estas marros son las siguientes:

- "$@": Almacena el nombre del archivo a construir.
- "$?": Almacena la cadena de nombres de fichero que son más jóvenes que el fichero a construir.

Si *make* encuentra un fichero que debe ser reconstruido (porque alguno de los ficheros de los que depende ha sido modificado), se ejecuta la secuencia de comandos indicada. Normalmente, cada línea de comandos se muestra por pantalla y, una vez sustituidas las macros que aparezcan en ella, se invoca al intérprete de comandos para que la ejecute. La impresión de la línea a ejecutar puede evitarse añadiendo la opción *-s* al invocar a *make* o bien si la línea de comandos comienza por un signo @.

2.4. Finalización de la ejecución de make

Una vez que finaliza la ejecución del comando, *make* comprueba el código de retorno (el valor que devuelve el programa una vez ejecutado: véase la función exit(). Si el comando se ejecutó sin problemas, se ejecuta entonces la siguiente línea en un nuevo shell, y así sucesivamente hasta que se hayan realizado todas las acciones. Si aparece un error (cuando algún programa devuelve un código de error distinto de cero), *make* se detiene. Lo habitual es que si se ha intentado compilar un programa con errores, el compilador devuelva un código de error.

Sin embargo, hay ocasiones en las que un código de retomo distinto de cero no necesariamente indica un problema. Por ejemplo, en Unix se puede utilizar el comando mkdir (que crea un directorio) para asegurarse de que existe: si existe, el comando devolverá un código de error, pero desearemos que make continúe de todos modos. Para ignorar errores en la línea de comandos, puede precederse la línea que contiene la orden por un guión -. Este guión se descarta antes de pasarle la línea al intérprete de comandos.

Por ejemplo,

```
clean:
    -rm -f *.o
```

Esto permite que se continúe aunque no se pueda borrar ningún archivo. Además, si se ejecuta make con el flag *-i*, se ignoran los errores en todos los comandos de todas las reglas. Tanto con este flag como con el guión, el comportamiento de *make* es similar, en el sentido de que ignora el error

pero muestra un mensaje que indica el código de retorno con el que acabó el comando, y especifica que ese error ha sido ignorado.

En caso contrario, *make* finaliza devolviendo un código de error distinto de cero. Sin embargo, si se especifica en línea de comandos la opción *-k* ("keep going"), *make* continúa reconstruyendo todos los ficheros que pueda antes de acabar con el código de error. Por ejemplo, si no puede obtenerse un fichero objeto a partir de un fichero en C, debido a un error en la compilación, *make* continuará compilando otros ficheros, aún sabiendo que el enlace final será imposible. La opción *-k* se utiliza normalmente para detectar el mayor número posible antes de volver a intentar la compilación.

Sufijo	Descripción
.o	Fichero objeto.
.c	Fichero fuente en C.
.f	Fichero fuente en Fortran.
.s	Fichero fuente en Assembler.
.y	Gramática fuente en Yacc para C.
.l	Gramática fuente en Lex.

Cuadro 1: Sufijos por defecto reconocidos por *make*.

2.5. Reglas implícitas

make utiliza una tabla de sufijos por defecto para suplir la falta de información sobre cómo tratar determinados ficheros. Por ejemplo, si tiene el fichero *pepe.c* y debe construir el *pepe.o*, supone que *pepe.c* es un fichero en C. Sin embargo, si *pepe.c* no aparece, pero aparece *pepe.y*, *make* invoca a Yacc para obtener *pepe.c*, y luego lo compila. La lista de sufijos utilizada por defecto es la que aparece en la tabla 1.

Para modificar estos compiladores utilizados por defecto, existen las macros **AS**, **CC**, **YACC**, y **LEX**, entre otras. Por ejemplo, para que los ficheros en C los compile utilizando el compilador *nuevocc* en lugar de *cc*, basta indicarlo con la línea

```
make CC=nuevocc
```

Las macros **CFLAGS**, **YFLAGS** y **LFLAGS** pueden utilizarse para pasarle a esos programas determinados flags. Por ejemplo, la línea

```
make "CFLAGS=-O"
```

hace que se utilice el compilador de C con la opción de optimización.

2.6. Conclusiones

Actualmente, casi todo paquete de software distribuido en el mundo en código fuente incorpora un *Makefile* para compilarlo. Esto incluye tanto pequeños programas de dos o tres módulos como compiladores de C (el compilador de C desarrollado por GNU, *gcc*, consiste en aproximadamente 25 Mb de código fuente en C, y para instalarlo en una máquina es preciso compilarlo con un compilador de ANSI C a través del *Makefile* que incorpora). Otro ejemplo de utilidad del *make* es el kernel de Linux (el núcleo del sistema operativo), que también hay que compilarlo a través del *Makefile* que viene en la distribución.

Hemos intentado dar una visión global de la utilidad *make*, explicando los comandos básicos. Para más información sobre *make* (aún quedaría mucho por decir), deberá consultarse la documentación de la versión concreta que se esté utilizando.

3. Creación de bibliotecas con ar

3.1. Introducción

Como hemos dicho en el capitulo anterior, la creación de bibliotecas permite una mayor modularidad y portabilidad en el programa. Agrupar un conjunto de funciones afines en una biblioteca presenta

varias ventajas. En primer lugar, no es necesario compilarlas repetidamente al ser utilizadas en diferentes programas. Por otra parte, las funciones archivadas pueden ser tratadas como "cajas negras", con unas especificaciones concretas de entrada y salida, con lo que el programador que las utiliza se desentiende de sus detalles de diseño.

En el entorno Unix, las bibliotecas se construyen con la ayuda del compilador de C y la utilidad *ar*. Esta utilidad no es solamente una herramienta utilizada para generar y mantener bibliotecas: es en rigor una herramienta para crear, modificar y extraer miembros de archivos. Un archivo, en este contexto, es un fichero compuesto por una colección de otros ficheros en una estructura que permite recuperar los ficheros originales, o miembros del archivo.

3.2. Utilización de ar: un ejemplo

Como ejemplo, vamos a crear una biblioteca que contenga las funciones para pasar de coordenadas rectangulares a polares y viceversa. A continuación aparece el código de estas funciones, tal como aparecerían en una típica sesión de trabajo Unix. Los dos ficheros que aparecen son *polarec.c*, que contiene la función del mismo nombre que pasa las coordenadas de polares a rectangulares, y *recapol.c*, que realiza la tarea inversa. *polarec.c* contendrá el código que aparece en el listado 3.1. Por su parte, *recapol.c* contendrá el código del listado 3.2.

Listado 3.1 Fichero *polarec.c*.

```
#include <math.h>

typedef struct {
float modulo;
float argumento;
}       polares;

typedef struct {
float x;
float y;
}       rectangulares;

rectangulares PolARec (polares pol_inicial) {

rectangulares rect_aux;

rect_aux.x = pol_inicial.modulo*cos(pol_inicial.argumento);
rect_aux.y = pol_inicial.modulo*sin(pol_inicial.argumento);

return(rect_aux);
}
```

Como puede verse, las dos funciones que aparecen en los listados 3.1 y 3.2 utilizan funciones de biblioteca estándar: las funciones *sin()*, *cos()* y *sqrt()*. Estas funciones aparecen declaradas en el fichero de cabecera *math.h*.

Listado 3.2 Fichero *recapol.c*.

```
#include <math.h>

typedef struct {
float modulo;
float argumento;
}       polares;

typedef struct {
float x;
float y;
}       rectangulares;

polares RecAPol(rectangulares rect_inicial) {

polares pol_aux;
float aux_x, aux_y;

aux_x=rect_inicial.x*rect_inicial.x; .
aux_y=rect_inicial.y*rect_inicial.y);

pol_aux.modulo=sqrt(aux_x+aux_y);
        pol_aux.argumento=rect_inicial.x/pol_aux.modulo;

return (pol_aux);
}
```

Lo primero es compilar estos dos ficheros para generar el código objeto que luego se almacenará en la biblioteca. Para ello se utiliza la opción *-c* del compilador de C:

```
$ cc -c polarec.c recapol.c
```

Si no han aparecido errores, procederemos a la creación de la biblioteca:

```
ar r libnueva.a recapol.o polarec.o
```

a lo cual el programa ar nos responderá:

```
ar: creating libnueva.a
```

A la herramienta *ar* se le pasa ante todo la opción *r*, que crea un nuevo archivo con el nombre indicado, seguido de los nombres de los ficheros que se deben añadir. Podemos utilizar el comando de Unix *file*, que nos indica de qué clase es un fichero en concreto, para ver qué nos dice de *libnueva.a*.

```
$ file libnueva.a
libnueva.a: current ar archive random library
```

Ya tenemos la biblioteca creada: sólo falta usarla. Escribiremos un programa llamado *coord*, que pase unas coordenadas polares a rectangulares, y que las vuelva a pasar a polares (para ver si se producen errores por redondeo). El programa es el del listado 3.3.

Listado 3.3 Programa que utiliza las funciones *PolARec ()* y *RecAPol ()*.

```
#include "coord.h" main {

polares p;
rectangulares r;

printf ("Introduce el modulo :"); scanf ("%f,", &p.modulo);
printf ("Introduce el argumento :"); scanf ("%f", &p.argumento);

printf ("Las coordenadas polares son mod=%f, arg=%f\n", p.modulo, p.argumento);

r=PolARec(p);

printf ("Las coordenadas rectangulares son x=%f, y=%f\n" r.x, r.y);

p=RecAPol(r);

printf ("Y las nuevas polares, mod=%f, arg=%f\n", p.modulo, p.argumento);
}
```

Para utilizar las funciones creadas, necesitamos declarar su prototipo al principio del programa. Además, tenemos que declarar las estructuras polares y rectangulares. De todo eso se encarga el fichero de cabecera *coord.h*, que aparece en el listado 3.4. Como ese fichero no se encuentra en el

Listado 3.4 Fichero de cabecera *coord.h*.

```
#include <math.h>

typedef struct {
float modulo;
float argumento;
)polares;

typedef struct {
float x;
float y;
}rectangulares;

rectangulares PolARec (polares pol_inicial);
polares RecAPol (rectangulares rect_inicial);
```

directorio por defecto, es necesario indicarle al compilador dónde encontrarlo: por eso, al incluirlo en el listado anterior, no se lo encierra entre los símbolos <>, sino que se pone el nombre entre comillas (véase la sección 1.7).

Para compilar esto, es necesario tener varias cosas en cuenta. En primer lugar, que debemos indicarle al compilador que vamos a utilizar una nueva biblioteca. Esta biblioteca no se encuentra en el directorio por defecto en el que el compilador almacena todas las bibliotecas, sino en nuestro directorio. Además, tenemos que pedirle que incluya también la biblioteca que contiene las funciones matemáticas. Esa biblioteca es *libm.a*, y sí se encuentra en uno de los directorios por defecto para bibliotecas estándar. Veamos cuál será el formato de la llamada al compilador:

```
cc -L. coord.c -lnueva -lm -o coord
```

La opción *-L* indica al compilador el directorio en el que debe buscar bibliotecas además de los directorios por defecto. En nuestro caso, es el directorio actual (.). La opción *-l*, por su parte, indica el nombre de la biblioteca que debe enlazarse junto con nuestro programa. Véase la sección 1.6, para saber más sobre el uso de bibliotecas de funciones.

Una vez hecho todo esto, el directorio actual contendrá los siguientes archivos:

```
total 16
-rwxr-xr-x      1 diego infor   13312   Oct     20      13:12   coord
-rw-r--r--      1 diego infor   508     Oct     20      13:03   coord.c
-rw-r--r--      1 diego infor   234     Oct     20      11:37   coord.h
-rw-r--r--      1 diego infor   686     Oct     20      13:06   libnueva.a
-rw-r--r--      1 diego infor   354     Oct     20      11:26   polarec.c
-rw-r--r--      1 diego infor   248     Oct     20      13:11   polarec.o
-rw-r--r--      1 diego infor   417     Oct     20      11:28   recapol.c
-rw-r--r--      1 diego infor   208     Oct     20      13:11   recapol.o
```

Como puede verse, el fichero *coord* es un ejecutable (tiene permisos de ejecución). Ejecutándolo, obtenemos lo siguiente:

```
$ coord
Introduce el modulo     :1
Introduce el argumento :0.7854
Las coordenadas polares son mod=1.000000, arg=0.785400
Las coordenadas rectangulares son x=0.707106, y=0.707108
Y las nuevas polares, mod=1.000000, arg=0.785396
```

El argumento debe darse en radianes. Se le ha pasado aproximadamente PI/4, es decir, 45 grados. Por lo tanto, las coordenadas rectangulares x e y son aproximadamente iguales. Al volver a pasar las coordenadas rectangulares a polares se produce un error debido al redondeo, que habría sido menor si en lugar de utilizar el tipo de dato *float* se hubiera optado por un *double*.

4. Indentador de código: indent

Para terminar con nuestro repaso a las herramientas que facilitan la programación en C en entorno Unix, vamos a comentar una que puede ser de gran utilidad en algunos casos. Hablamos de *indent*, un indentador de código. *indent* se encarga de procesar un fichero con código fuente en C, y reescribirlo siguiendo criterios de legibilidad estándar. Para utilizarlo, basta con ejecutarlo seguido del nombre del fichero en C a procesar, e indicando además dónde queremos almacenar la nueva versión, tras la opción *-o*:

```
indent miprog.c -o nuevoprog.c
```

Si queremos que la nueva versión reemplace a la original, basta con no indicar fichero de salida:

```
indent miprog.c
```

De todos modos, el fichero original se almacena con el nombre *miprog.c~*.
Existen tres formatos básicos para el código fuente en C:

- El formato de codificación GNU.
- El formato de codificación utilizado por Kernighan y Ritchie en su libro *El lenguaje de programación C*.
- El formato Berkeley, utilizado en las primeras versiones de indent.

Si no especificamos ningún parámetro adicional, el fichero *miprog.c* se reemplaza por una versión con el formato de codificación GNU. Este formato es el utilizado en la programación de aplicaciones de dominio público dentro del proyecto GNU. También puede utilizarse indicando la opción *-gnu*.

Invocando a *indent* con la opción *-kr* se utiliza el formato de Kernighan y Ritchie en su libro *El lenguaje de programación C* Este libro marcó en su día un estándar en el C, por lo que su estilo de indentación es ampliamente utilizado.

Finalmente, con la opción *-orig* se utiliza el estándar de indentación de Berkeley. *indent* admite muchos más parámetros, que permiten especificar comportamientos concretos ante determinadas líneas de código: dónde y cómo poner los comentarios, cuántos espacios dejar en cada nivel de indentación, etcétera. Para conocerlos basta con consultar la página man:

```
man indent
```

El funcionamiento de *indent* depende del aspecto del código original. Por supuesto, dicho código debe ser sintácticamente correcto, ya que de lo contrario *indent* indicará un error.

indent resulta de utilidad para facilitar la lectura cuando el código fuente está pésimamente indentado, o bien para homogeneizar el aspecto de ficheros de código fuente desarrollado por diferentes programadores. Con respecto al estilo a utilizar, los tres estilos comentados son igualmente válidos: elegir uno en concreto es cuestión de gustos. Lo mejor es probar con los tres y quedarnos con el estilo que más nos guste.

5. Directivas del preprocesador

A continuación aparece un resumen de las directivas ANSI del preprocesador de C. Su conocimiento resulta muy útil porque es el principal mecanismo para establecer opciones dependientes del compilador al transportar código de una máquina a otra.

5.1. El preprocesado

La primera etapa del proceso que realiza el compilador sobre un programa fuente en C es el preprocesado. El resultado de esta etapa se denomina "unidad de traducción", y sirve como entrada al compilador propiamente dicho. El preprocesado se encarga de realizar, entre otras, las siguientes tareas:

- Reemplazar los comentarios por un único espacio en blanco.
- Reemplazar los valores de las constantes por su equivalente.
- Reconocer y ejecutar las directivas de preprocesado que aparecen en el programa.
- Reemplaza las secuencias de escape que aparecen dentro de caracteres y cadenas por sus caracteres individuales equivalentes.

Existe un conjunto de directivas que el preprocesador se encarga de ejecutar. Cada directiva comienza por el símbolo de "numeral" o "almohadilla", #. [4]

Veremos a continuación las directivas reconocidas por el ANSI C.

5.2. Directiva *include*

Permite incluir los contenidos de un fichero de cabecera u otro fichero fuente en la unidad de traducción. El preprocesador reemplaza la directiva por el fichero indicado en ella. Su formato es

```
#include <fichero.h>
```

para incluir el fichero de cabecera estándar *fichero.h*, que debe estar en el directorio del compilador utilizado para almacenar los ficheros de cabecera estándar. Si se desea incluir un fichero de cabecera propio, debe indicarse al compilador que lo busque en el directorio por defecto, encerrando al nombre de fichero entre comillas dobles:

[4] **Importante:** En el estándar ANSI se indica que el símbolo # puede aparecer rodeado de cualquier número de espacios en blanco o caracteres de tabulación. Otros dialectos de C, como el que utiliza el compilador del sistema operativo HP-UX, obliga a colocarlo en la columna 1 de la línea que contiene la directiva.

```
#include "mifichero.h"
```

Opcionahnente, puede suministrársele al compilador la ruta de búsqueda dentro del árbol de directorios:

```
#include "/usr/users/include/mifichero.h" .
```

En casi todos los compiladores existen opciones que permiten añadir directorios a la lista que se utiliza por defecto para buscar ficheros cacabecera. Esa opción suele ser *-l*. Consúltese la sección 1.7, o el manual del compilador para más detalles.

5.3. Directivas *define* y *undef*

Esta directiva permite definir un nombre como una macro. Siguiendo al nombre de la directiva, debe indicarse el nombre a definir, seguido del objeto que se define, como en los siguientes ejemplos:

```
#define PI            3.1416
#define producto(x,y) ((x) * (y))
```

En el primer caso, se define una constante con el nombre PI. En el segundo, se define una macro con dos parámetros. El preprocesador se encarga de expandir las macros en todas las líneas que no sean directivas. Para eliminar una definición de macro, se utiliza la directiva *undef*:

```
#undef nombre
```

donde *nombre* es el nombre de la macro o constante definida con *define*.

5.4. Directivas condicionales

Estas directivas permiten que el preprocesador seleccione un grupo de líneas de un fichero y elimine otras. Las directivas condicionales forman "grupos *if*". Un grupo *if* comienza con una de las siguientes directivas:

- La directiva *if* condicion, que será cierta si condicion lo es.
- La directiva *ifdef* nombre, que será cierta si nombre está definido a través de una directiva *define*.
- La directiva *ifndef* nombre, que será cierta si nombre no está definido a través de una directiva *define*.

A continuación, en las líneas siguientes, aparece el primer grupo de líneas que se quiere saltar selectivamente. Si la condición es verdadera, el preprocesador retiene este primer grupo de líneas, eliminándolo en caso contrario. El bloque de líneas a tratar de esta manera finaliza con una directiva *endif*.

Por ejemplo, puede definirse la constante PI sólo en el caso de que no se haya definido con anterioridad:

```
#ifndef PI
#define PI 3.1416
#endif
```

Opcionalmente, pueden utilizarse las siguientes directivas:

- La directiva *elif* condición, que hace que, si la anterior condición verificada es falsa, se evalúe la que acompaña al *elif*. Si ésta resulta ser cierta, las instrucciones que siguen a la directiva *elif* se conservan. En caso contrario, se eliminan.
- La directiva *else* indica la acción a realizarse si todas las condiciones anteriores resultan falsas.

Ambas directivas deben utlizarse dentro del grupo *if* correspondiente (es decir, antes del *endif* correspondiente a ese grupo).

Veamos un ejemplo:

```
#ifdef __linux__
int x;
#elif SISTEMA == MSDOS
short int x;
#else
#error SISTEMA OPERATIVO NO RECONOCIDO
#endif
```

En la expresión condicional que forma parte de una directiva *if* se escriben sólo expresiones constantes enteras, teniendo en cuenta que no pueden utilizarse los operadores *sizeof* ni la conversión forzada de tipos (cast). Por lo tanto, en el ejemplo anterior, SISTEMA y MSDOS deberán ser constantes enteras.

Debe tenerse en cuenta, además, que no puede incluirse un fichero fuente que contenga la directiva *if* sin la correspondiente directiva *endif* dentro del mismo fichero.

5.5. Directiva *error*

En el ejemplo anterior se utilizó esta directiva. Proporciona un mensaje de diagnóstico en tiempo de compilación. A continuación de ella debe aparecer texto que el preprocesador deba mostrar como mensaje de error. El proceso de compilación se detiene al aparecer dicho mensaje.

5.6. Directiva *pragma*

Según el estándar, la directiva *pragma* permite pasar al preprocesador información no estándar. Siguiendo al nombre de la directiva, debe aparecer cualquier texto que el traductor pueda analizar.