Compilar y depurar aplicaciones con las herramientas de programación de GNU

Acerca de este documento

En este tutorial pretendemos mostrar el manejo de las herramientas de programación que GNU ha puesto a disposición de los usuarios en multitud de sistemas, y que Apple ha elegido como base para sus herramientas de programación. Estas herramientas incluyen los conocidos comandos `gcc` y `gdb`. Sin embargo, las herramientas de programación de GNU son mucho más completas, e incluyen multitud de comandos que vamos a ir comentando a lo largo de este tutorial.

Este documento se centra en la forma de trabajar, y las opciones particulares en Mac OS X pero, debido a la interoperatividad de estas aplicaciones, lectores acostumbrados a trabajar en otros entornos (como por ejemplo Linux, FreeBSD, o incluso Windows) pueden encontrar útil este tutorial, ya que en su mayoría las forma de trabajar es la misma. Además, hemos procurado hacer hincapié en las peculiaridades de las herramientas de GNU en Mac OS X, con el fin de facilitar el poder usar este documento para trabajar en otros entornos.

El tutorial asume que el lector conoce al menos el lenguaje C, aunque en concreto en tres temas se explica también el uso de las GCC para compilar aplicaciones C++ y Java. Si el lector no conoce alguno de estos dos últimos lenguajes puede saltarse el correspondiente tema sin perjuicios para poder seguir el resto del documento.

Al acabar este tutorial el lector debería de haber aprendido a compilar y depurar sus aplicaciones, crear librerías, medir el rendimiento, e incluso a combinar aplicaciones escritas en distintos lenguajes.

Nota legal

Madrid, Abril del 2006
Para cualquier aclaración contacte con:
`fernando@macprogramadores.org`

Tabla de contenido

TEMA 1: Introducción a las herramientas de programación de GNU

TEMA 2: Compilando en C

TEMA 3: Crear y usar librerías

TEMA 4: Compilando en C++

TEMA 5: Compilando en Java

TEMA 6: Combinar distintos lenguajes

TEMA 7: Depuración, optimización y perfilado

Tema 1

Introducción a las herramientas de programación de GNU

Sinopsis:

En este primer tema vamos a introducir que son las herramientas de programación de GNU, su evolución histórica, y como debemos usarlas.

Esta información es de naturaleza introductoria para el resto del documento, con lo que los usuarios más avanzados, si lo desean, pueden saltarse este primer tema y empezar su lectura en el Tema 2. Aunque antes de hacerlo les recomendamos echar un vistazo a los conceptos que este tema introduce.

1. Las GNU Compiler Collection (GCC)

En 1984 Richard Stallman inició el desarrollo de un compilador C al que llamó `gcc` **(GNU C compiler)**. El compilador fue desarrollado con la intención de que los programadores que quisieran participar en el proyecto GNU pudieran desarrollar software con vistas a desarrollar un UNIX libre. En aquel entonces todos los compiladores que existían eran propietarios, con lo que se tuvo que desarrollar un compilador desde el principio.

Con el tiempo este compilador evolucionó y paso a soportar varios lenguajes (en concreto actualmente soporta C, C++, Objective-C, Java, Fortran y Ada), con lo que el nombre (que originariamente indicaba que se trataba de un compilador C) también evolucionó y pasó a escribirse en mayúsculas como **GCC (GNU Compiler Collection)**. El nuevo nombre pretende indicar que se trata, no de un comando, sino de un conjunto de comandos usados para poder programar en distintos lenguajes.

En español suele encontrarse el nombre **Herramientas de Programación de GNU** para referirse a las GCC (la traducción literal "Colección de Desarrollo GNU" no parece una traducción adecuada), y este nombre es el que hemos elegido para este tutorial.

Las GCC son posiblemente el proyecto de software libre más importante que se ha emprendido en el mundo. Principalmente porque todo el software libre que se está empezando a desarrollar se apoya en él. Desde su creación GNU expuso el objetivo claro de ayudar a crear un UNIX libre, y la mejor prueba de que este objetivo se ha cumplido es Linux, un sistema operativo creado por miles de desarrolladores los cuales siempre han tenido como denominador común las GCC. Además las GCC no sólo se han usado para crear Linux, sino que otros muchos sistemas operativos como las distribuciones BSD, Mac OS X o BeOS también las han elegido como sus herramientas de desarrollo.

Actualmente el desarrollo de las GCC está siendo coordinado por un amplio grupo de comunidades procedentes de la industria, la investigación, y la universidad. El grupo que coordina los desarrollos es el **GCC steering comittee**, el cual trata de velar por los principios de software libre bajo los que fue creada inicialmente la herramienta.

Apple eligió las GCC como base de sus herramientas de desarrollo, y aunque han creado herramientas de interfaz gráfica como Xcode o Interface Builder, estas herramientas lo único que hacen es apoyarse en las GCC para realizar su trabajo. Es más, los empleados de Apple han contribuido a mejorar muchos aspectos de las GCC, especialmente en lo que a las optimizaciones para PowerPC y AltiVec se refieren.

2. Arquitectura del compilador de GNU

Una característica de las GCC, que además es muy común de encontrar en el software libre, es que el proyecto no ha seguido un proceso formal de ingeniería con fases de análisis, diseño, implementación y pruebas. GCC está siendo creado por cientos de programadores de distintas nacionalidades trabajando sobre un repositorio donde tanto la información de análisis como la de diseño está entremezclada con el código fuente a base de comentarios. Aun así han conseguido crear una herramienta de calidad, estable, rápida y con un bajo consumo de memoria.

2.1. Frontend y backend

A groso modo podemos dividir el proceso de compilado en dos partes: Frontend y backend. El **frontend** (donde se realiza el análisis léxico y gramatical) lee un fichero de código fuente y crea una estructura en forma de árbol en memoria de acuerdo a la gramática del lenguaje. El **backend** lee este árbol y lo convierte en una secuencia de instrucciones ensamblador para la máquina destino. Dentro del backend podemos incluir tanto el análisis semántico, como la generación de código y la optimización.

Aunque los lenguajes que GCC puede compilar son muy distintos, a medida que avanzan las fases del frontend se va generando una estructura común a todos los lenguajes, de forma que un mismo backend puede generar código para distintos lenguajes. Otra característica de GCC es que usando distintos backend podemos generar código para distintas plataformas sin necesidad de disponer de un frontend para cada plataforma. Esto ha permitido que actualmente las GCC sean capaces de generar código para casi todos los tipos de hardware existentes.

2.2. Fases del compilador

Antes de empezar a describir como usar el comando `gcc`, conviene recordar al lector las tres fases que tiene la generación de un ejecutable, por parte un compilador:

1. **Preprocesado**. Expande las directivas que empiezan por `#` como `#define`, `#include` o `#ifdef`.
2. **Compilación**. Traduce el código fuente a código objeto, un código cercano al código máquina donde las llamadas externas están sin resolver y las direcciones de memoria son relativas.

3. **Enlazado**. Combina los diferentes ficheros de código objeto resolviendo las llamadas que han quedado pendientes y asignando direcciones definitivas al ejecutable final.

En el caso de `gcc`, aunque llamaremos compilación al proceso global, la etapa de compilación consta de dos partes:

1. **Compilación.** Donde se traduce el código fuente a código ensamblador.
2. **Ensamblado.** Donde se traduce el código ensamblador en código objeto.

El proceso de compilación lo realiza en parte el frontend (generando un árbol gramatical), y en parte el backend (generando el código ensamblador para la máquina que corresponda). Tanto el proceso de ensamblado como el de enlazado son realizados por el backend.

3. Comandos disponibles

Como ya hemos comentado, actualmente las GCC constan de un gran número de comandos que iremos viendo a lo largo del tutorial, y que hemos resumido en la Tabla 1.1. En la tabla también indicamos el tema donde se explica su funcionamiento.

Comando	Descripción	Tema
`gcc`	Compilador de las GCC.	1
`cc1`	El compilador actual de C a ensamblador.	2
`as`	Ensamblador que traduce de lenguaje ensamblador a código objeto.	2
`ld`	El enlazador de GCC.	2
`gcc_select`	Cambia la versión del compilador que estemos usando	2
`c++filt`	Elimina los nombres con name mangling de la entrada estándar e imprime el resultado en la salida estándar.	2
`nm`	Muestra la tabla de símbolos de un fichero de código objeto o de una librería.	2,3
`otool`	Muestra información sobre un fichero Mach-O.	3
`libtool`	Permite generar librerías tanto de enlace estático como de enlace dinámico.	3
`cmpdylib`	Compara la compatibilidad entre dos librerías de enlace dinámico.	3
`strip`	Elimina símbolos innecesarios de un fichero de código objeto.	3
`g++`	Una versión de `gcc` que fija el lenguaje por defecto a C++, e incluye las librerías estándar de C++.	4
`c++`	Equivalente al anterior.	4
`collect2`	Genera las inicializaciones de los objetos globales.	4
`gcj`	Compilador Java de las GCC.	5
`gij`	Interprete Java de las GCC.	5
`jcf-dump`	Nos permite obtener información sobre el contenido de un fichero `.class`.	5
`jv-scan`	Recibe un fichero de código fuente Java y produce distintas informaciones sobre éste.	5
`grepjar`	Busca una expresión regular en un fichero `.jar`.	5
`fastjar`	Una implementación del comando `jar` de Sun que ejecuta considerablemente más rápido.	5
`gcjh`	Permite generar los prototipos CNI (o JNI si usamos la opción `-jni`) de los métodos a implementar.	6
`gdb`	Es el depurador de GNU. Permite ejecutar paso a	7

	paso un programa e identificar errores.	
`leaks`	Permite identificar fugas de memoria, es decir, memoria reservada dinámicamente que nunca se libera.	7
`malloc_history`	Permite inspeccionar un log con toda la memoria dinámica que ha sido reservada y liberada durante la ejecución de un programa.	7
`gprof`	Permite perfilar un programa, es decir, detectar partes de programa que si se optimizan podríamos conseguir una considerable mejora global en su rendimiento.	7
`gcov`	Permite realizar test de cobertura a un programa, es decir, saber cuantas veces se ejecuta cada línea del programa.	7

Tabla 1.1: Comandos de GCC

Actualmente no todos los comandos que se describen en este tutorial están disponibles por defecto en Mac OS X (y en general tampoco en ningún otro sistema operativo), con lo que si no encuentra alguno de estos comandos en su terminal, dejamos como ejercicio para el lector el buscarlo e instalarlo. En el caso de Mac OS X, actualmente no está disponible el compilador de Java de GNU, aunque existe una implementación de la maquina virtual de Sun realizada por Apple, es decir, no encontrará comandos como `gcj`. Le recomendamos instalar el paquete `mingw-gcc` del proyecto Fink, ya que este paquete si que incluye muchos de los comandos de las GCC. Si sigue sin encontrar algún otro comando de los comentados en este tutorial, posiblemente lo encuentre ya portado a Mac OS X dentro del gestor de paquetes Fink.

3.1. Opciones de la línea de comandos

Las opciones de la línea de comandos siempre empiezan por uno o dos guiones. En general, cuando la opción tiene más de una letra se ponen dos guiones, y cuando sólo tiene una letra se pone un guión. Por ejemplo, para compilar el fichero `hola.c` y generar el fichero de código objeto `hola.o`, usando ANSI C estándar se usa el comando:

```
$ gcc hola.c --ansi -c -o hola.o
```

Vemos que se indican primero los ficheros de entrada (`hola.c` en este caso), y luego las opciones, algunas de las cuales tienen parámetros y otras no. En general `gcc` es bastante flexible y suele aceptar que le pasemos las opciones en cualquier orden.

Las opciones de una sola letra, si reciben parámetros estos pueden ir separados por espacio o ir juntos, es decir, en el ejemplo anterior podríamos haber usado `-ohola.o` en vez de `-o hola.o`.

Las opciones de la línea de comandos se pueden clasificar entres categorías:

1. Específicas del lenguaje. Son opciones que sólo se pueden usar junto con determinados lenguajes de programación. Por ejemplo, `-C89` sólo se usa en C para indicar que queremos usar el estándar ISO de 1989.

2. Específicas de la plataforma. Son opciones que sólo se usan en determinada plataforma. Por ejemplo `-f-ret-in-387` se usa sólo en Intel para indicar que el retorno en punto flotante de las funciones se haga en un registro de punto flotante.

3. Generales. Son opciones que se pueden usar en todos los lenguajes y en todas las plataformas, como por ejemplo `-O` usada para indicar que queremos optimizar el código objeto resultante.

Como veremos más adelante, hay opciones que no van dirigidas al compilador sino al enlazador. Cuando `gcc` recibe una opción que no entiende simplemente la pasa al enlazador esperando que éste la sepa interpretar.

3.2. Variables de entorno

Además de usando opciones, el comportamiento de `gcc` puede ser personalizado usando variables de entorno. Las principales variables de entorno que afectan el comportamiento de `gcc` se resumen en la Tabla 1.2.

Var. de entorno	Descripción
`LIBRARY_PATH`	Lista de directorios, separados por dos puntos, en la que se indica en que directorios buscar librerías. Si se indica la opción `-L` primero se busca en la ruta dada por la opción, y luego en la dada por la variable de entorno.
`CPATH`	Directorio donde buscar ficheros de cabecera tanto para C, C++ y Objective-C.
`C_INCLUDE_PATH`	Directorio donde buscar ficheros de cabecera. Variable de entorno usada sólo por C.
`CPLUS_INCLUDE_PATH`	Directorio donde buscar ficheros de cabecera. Variable de entorno usada sólo por C++.
`OBJC_INCLUDE_PATH`	Directorio donde buscar ficheros de cabecera. Variable de entorno usada sólo por Objective-C.

Tabla 1.2: Principales variables de entorno de `gcc`

Con las variables `CPATH`, `C_INCLUDE_PATH`, `CPLUS_INCLUDE_PATH` y `OBJC_INCLUDE_PATH`, si se indica como un elemento de la lista el directorio vacío, se busca en el directorio actual. Por ejemplo si `CPATH` vale `/sw/lib::`, es equivalente a usar `-I. -I/sw/lib`. Además el primer elemento no debe ser dos puntos o se interpreta como un elemento vacío. Por ejemplo en el ejemplo anterior sería equivalente que `CPATH` valiese `:/sw/lib`.

3.3. Empezar a manejar `gcc`

Si nosotros pasamos a `gcc` un programa como el del Listado 1.1, ejecutando:

```
$ gcc hola.c
```

Éste lo compilará y enlazará generando como salida el fichero ejecutable `a.out`:

```
$ a.out
Hola mundo
```

```
/* hola.c */

#include <stdio.h>

main()
{
 printf("Hola mundo\n");
 return 0;
}
```

Listado 1.1: Programa C mínimo

Si preferimos que el fichero de salida tenga otro nombre podemos indicarlo con la opción `-o`. Por ejemplo, el siguiente comando genera el fichero ejecutable `hola`.

```
$ gcc hola.c -o hola
```

Tema 2

Compilando en C

Sinopsis:

En este segundo tema pretendemos detallar el funcionamiento del compilador en lo que al lenguaje C se refiere.

Empezaremos viendo el funcionamiento del preprocesador, para luego detallar las opciones de línea de comandos propias del lenguaje C. Por último veremos que extensiones al C estándar introducen las GCC.

1. El preprocesador

El concepto de preprocesador fue introducido inicialmente por C, pero después otros lenguajes como C++ o Objective-C lo han heredado.

El programa encargado de realizar el preproceso es un comando llamado `cpp`[1].

1.1. Opciones relacionadas con el preprocesador

Existen una serie de opciones relacionadas con el preprocesador que se resumen en la Tabla 2.1.

Opción	Descripción
`-D`	Define un identificador del preprocesador (que puede ser un macro). Para ello usamos `-D` *`nombre`*`=[`*`valor`*`]`, donde si no asignamos *`valor`*, por defecto *`nombre`* vale 1.
`-U`	Usar la opción `-U` *`nombre`* cancela el identificador del preprocesador previamente definido. El identificador podría haber sido definido en un fichero, o con la opción `-D`.
`-I`	`-I` *`directorio`* incluye *`directorio`* en la lista de directorios donde buscar ficheros de cabecera. El directorio aquí pasado se usa antes que los directorios de inclusión estándar, con lo que esta opción nos permite sobrescribir ficheros de cabecera estándar.
`-W`	Existen algunos warnings relacionados con el preprocesador como por ejemplo `-Wunused-macros` que indica que queremos generar un warning si un identificador del preprocesador definido en un `.c` no es usado en todo el fichero. Esta opción no afecta a los identificadores definidos, y no usados, en los ficheros de cabecera, o en las opciones del preprocesador.

Tabla 2.1: Principales opciones relacionadas con el preprocesador

Como ya comentamos en el apartado 3.2 del Tema 1, las variables de entorno `CPATH`, `C_INCLUDE_PATH`, `CPLUS_INCLUDE_PATH` y `OBJC_INCLUDE_PATH` también permiten indicar directorios donde buscar ficheros de cabecera.

[1] Razón por la que al compilador de C++ se le llamó `c++` (o `g++`) y no `cpp`.

1.2. Identificadores del preprocesador desde la línea de comandos

Los identificadores del preprocesador siempre se pueden crear con la sentencia del preprocesador `#define`, pero además existe una serie de identificadores predefinidos que podemos obtener ejecutando el comando `cpp` con la opción `-dM` sobre un fichero vacío.

```
$ cpp -dM /dev/null
#define __APPLE__ 1
#define __DECIMAL_DIG__ 17
#define __DYNAMIC__ 1
#define __GNUC__ 3
```

Por otro lado siempre podemos definir identificadores desde la línea de comandos usando la opción `-D`, lo cual es útil muchas veces para controlar la forma en que compila nuestro programa. Por ejemplo, el programa del Listado 2.1 usa el identificador `DEPURABLE` para decidir si imprimir información de depuración. Para definir el identificador del preprocesador desde la línea de comandos podemos hacer:

```
$ gcc -DDEPURANDO depurable.c
```

```
/* depurable.c */

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

int main()
{
 double w=exp(M_PI/2);
 #ifdef DEPURANDO
  printf("w vale %f",w);
 #endif
 return 0;
}
```

Listado 2.1: Programa que usa un identificador del preprocesador

Si no indicamos valor para el identificador, por defecto vale `1`, podemos indicar un valor para el identificador de la forma:

```
$ gcc -DDEPURANDO=si depurable.c
```

En caso de que el valor tenga espacios u otros símbolos especiales debemos de entrecomillar el valor de la forma `-DDEPURANDO="Imprime mensaje"`.

1.3. Rutas de los ficheros de cabecera

Las sentencias del preprocesador de la forma:

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

Buscan ficheros de cabecera en los **directorios de inclusión estándar**, que en la mayoría de los SO (incluido Mac OS X) son, por este orden:

```
/usr/local/include
/usr/include
```

Si usamos la sentencia del preprocesador con comillas:

```
#include "fichero.h"
```

Además también se buscan los ficheros de cabecera en el directorio actual.

Si queremos que se busquen ficheros de cabecera en otros directorios debemos usar la opción `-I`, la cual debe de usarse una vez por cada nuevo directorio a añadir a la lista de directorios de ficheros de cabecera. Por ejemplo, para que busque ficheros de cabecera en `/sw/include` y `/usr/X11/include` debemos de usar:

```
$ gcc -I/sw/include -I/usr/X11/include fichero.cpp
```

2. Usando el compilador de C

La Tabla 2.2 presenta las extensiones de fichero utilizadas por las GCC en el caso del lenguaje C.

Extensión	Descripción
`.a`	Librería de enlace estático.
`.c`	Código fuente C que debe ser preprocesado.
`.h`	Fichero de cabecera.
`.i`	Código fuente C que no debe ser preprocesado. Este tipo de fichero se puede producir usando el preprocesador.
`.o`	Fichero objeto en formato adecuado para ser usado por el enlazador `ld`.
`.s`	Código ensamblador. Este fichero puede ser ensamblado con el comando `as`.
`.so`	Librería de enlace dinámico.

Tabla 2.2: Extensiones usadas por las GCC para el lenguaje C

2.1. Compilar y enlazar código fuente C

Para realizar la compilación en sentido global (compilación y ensamblado) de un fichero (pero no su enlazado) se usa la opción `-c`. Por ejemplo, para realizar la compilación del programa del Listado 1.1, que repetimos por comodidad en el Listado 2.2, podemos ejecutar:

```
$ gcc hola.c -c
```

Esto genera el fichero objeto `hola.o`, donde si queremos cambiar el nombre por defecto del fichero de salida también podríamos usar la opción `-o`.

```
/* hola.c */

#include <stdio.h>

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

Listado 2.2: Programa C en un sólo fichero

Una vez que tenemos el código objeto podemos generar el fichero ejecutable usando:

```
$ gcc hola.o -o hola
```

Si lo que tenemos son varios ficheros, como por ejemplo los del Listado 2.3 y Listado 2.4, podemos compilar ambos ficheros con el comando:

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

Lo cual genera los ficheros de código objeto `holamain.o` y `saluda.o`.

```
/* holamain.c */

void Saluda();

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

Listado 2.3: Programa principal con llamadas a otro módulo

```
/* saluda.c */

#include <stdio.h>

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

Listado 2.4: Módulo con una función C

Ahora podemos enlazar los ficheros de código objeto con:

```
$ gcc holamain.o saluda.o -o hola2
```

Y lógicamente también podemos realizar ambas operaciones a la vez con el comando:

```
$ gcc holamain.c saluda.c -o hola2
```

2.2. Preprocesar y generar código ensamblador

Como hemos dicho en el apartado 2.2 del Tema 1, la compilación consta de dos partes, una de compilación en sí donde se genera un fichero en lenguaje ensamblador, y otra de ensamblado donde a partir de un fichero en lenguaje ensamblador se genera otro de código objeto.

Podemos pedir que se realice sólo la fase de compilación, generando un fichero ensamblador a partir del programa C del Listado 2.2, usando la opción `-S` de la siguiente forma:

```
$ gcc hola.c -S
```

Esto genera el fichero `hola.s` con el programa C traducido a ensamblador. Lógicamente ahora podemos ensamblar este fichero generando uno de código objeto con el comando:

```
$ gcc hola.s -c
```

O generar directamente el ejecutable con:

```
$ gcc hola.s -o hola
```

También podemos indicar a `gcc` que queremos realizar sólo la etapa de preprocesado con la opción `-E`:

```
$ gcc hola.c -E -o hola.i
```

Este comando preprocesa las directivas del fichero `hola.c`, y genera el fichero `hola.i` con el código C preprocesado. Como se indica en la Tabla 2.2, la extensión `.i` la usa `gcc` para referirse a los ficheros preprocesados, con lo que si ahora ejecutamos:

```
$ gcc hola.i -c
```

Lo compila, pero ya no volvería a ejecutar el preprocesador sobre el fichero.

2.3. El comando `gcc` es un driver

Para ejecutar la fase de enlazado lo que hace `gcc` es ejecutar el comando `ld`. Este programa también lo podemos ejecutar nosotros de forma individual si tenemos un fichero de código objeto. Por ejemplo en el caso anterior podemos hacer:

```
$ ld hola.o -lcrt1.o -lSystem -o hola
```

La opción `-l` sirve para indicar librerías contra las que queremos enlazar. Con la opción `-lctr1.o` indicamos a `ld` que queremos enlazar el runtime de arranque de C para consola, es decir, el código que se ejecuta antes de empezar a ejecutar la función `main()`. Con la opción `-lSystem` indicamos que queremos enlazar funciones básicas del lenguaje C, como por ejemplo `printf()`.

De hecho `gcc` es lo que se llama un **driver**, porque es un comando que se encarga de ejecutar otros comandos. En concreto primero `gcc` ejecuta el comando `cc1` que es el compilador actual de C, es decir, el que traduce de C a ensamblador[1], luego ejecuta el comando `as` que es el ensamblador actual, y por último ejecuta el comando `ld` que enlaza el programa con las librerías C necesarias.

Podemos preguntar a `gcc` que comandos está ejecutando al actuar como driver con la opción `-###`. Por ejemplo:

```
$ gcc hola.c -###
Reading specs from /usr/libexec/gcc/darwin/ppc/3.3/specs
Thread model: posix
gcc version 3.3 20030304 (Apple Inc. build 1809)
 "/usr/libexec/gcc/darwin/ppc/3.3/cc1" "-quiet" "-D__GNUC
__=3" "-D__GNUC_MINOR__=3" "-D__GNUC_PATCHLEVEL__=0" "-D_
_APPLE_CC__=1809" "-D__DYNAMIC__" "hola.c" "-fPIC" "-quie
t" "-dumpbase" "hola.c" "-auxbase" "hola" "-o" "/var/tmp/
/ccQnqpw8.s"
 "/usr/libexec/gcc/darwin/ppc/as" "-arch" "ppc" "-o" "/va
r/tmp//ccEntkj5.o" "/var/tmp//ccQnqpw8.s"
 "ld" "-arch" "ppc" "-dynamic" "-o" "a.out" "-lcrt1.o" "-
lcrt2.o" "-L/usr/lib/gcc/darwin/3.3" "-L/usr/lib/gcc/darw
in" "-L/usr/libexec/gcc/darwin/ppc/3.3/../../.." "/var/tm
p//ccEntkj5.o" "-lgcc" "-lSystem" |
 "c++filt"
```

En negrita hemos marcado los comandos que ejecuta el driver. El entrecomillado de los argumentos de los comandos es opcional si estos no tienen espacios, aun así el driver los entrecomilla todos.

El último comando `c++filt`, al que `ld` pasa su salida, elimina el name mangling de los símbolos del texto que recibe como entrada y imprime el texto recibido en la salida. Esto se hace para facilitar su legibilidad por parte del usuario, si el enlazador informa de nombres de símbolos sin resolver.

Si las opciones de compilación que recibe `gcc` que no son válidas para el compilador `gcc` se las pasa al siguiente elemento de la cadena que es el ensamblador, y si éste tampoco las puede interpretar se pasan al enlazador. Por ejemplo las opciones `-l` y `-L` son opciones del enlazador, y cuando las recibe `gcc`, éste sabe que son propias del enlazador con lo que no se las pasa al compilador.

Hay algunas opciones, como por ejemplo `-x`, que son válidas tanto para el compilador como para el enlazador. En el caso del compilador sirve para indicar el lenguaje en que están hechos los ficheros de entrada, y en el caso del enlazador sirve para indicar que elimine los símbolos no globales que no

[1] El comando `cc1` a su vez llama al comando `cpp` para que realice el preprocesado.

se estén usando. Por defecto `gcc` enviaría la opción al compilador, con lo que si queremos que se la envíe al enlazador debemos de precederla por la opción `-Xlinker`. Por ejemplo, para indicar que el programa está hecho en C++ (aunque tenga extensión `.c`), y pedir al enlazador que elimine los símbolos no globales que no se estén usando, podemos hacer:

```
$ gcc hola.c -x cpp -Xlinker -x
```

2.4. Estándares

Por defecto `gcc` compila un programa C con todas las sintaxis extendidas del lenguaje habilitadas. Si queremos utilizar sólo determinado estándar, la Tabla 2.3 muestra un resumen de las opciones que permiten indicar el estándar a seguir.

Opción	Descripción
`-traditional`	Compila con la sintaxis del C original.
`-std=c89`	Acepta el estándar ISO C89.
`-ansi`	Igual que `-iso=c89`.
`-std=c99`	Acepta el estándar ISO C99.
`-std=gnu89`	Acepta el estándar ISO C89 más las extensiones de GNU. Es la opción por defecto si no se indica.
`-std=gnu99`	Acepta el estándar ISO C99 más las extensiones de GNU.
`-pedantic`	Emite todos los warnings que exigen los estándares ISO, y obliga al cumplimiento del estándar.

Tabla 2.3: Opciones para indicar el estándar C a seguir

El uso de opciones como `-ansi` o `-std=c99` no hace que se rechacen las extensiones de GNU. Si queremos garantizar que nuestro programa cumple estrictamente los estándares ISO (y en consecuencia compila con otros compiladores) debemos de añadir la opción `-pedantic`, en cuyo caso tampoco produce un error el usar las extensiones de GNU, pero si que produce warnings avisando de la circunstancia. Si, por contra, queremos que una violación del estándar ISO produzca un error, en vez de un warning, podemos usar la opción `-pedantic-error`.

2.5. Indicar el lenguaje a utilizar

El driver `gcc` utiliza la extensión de los ficheros para determinar el lenguaje utilizado. Por ejemplo si el fichero tiene la extensión `.c` utiliza el programa `cc1` para compilarlo. En ocasiones podemos tener el código fuente en un fichero con distinta extensión. En este caso podemos usar la opción `-x` para indicar el lenguaje en que está hecho el programa. Por ejemplo:

```
$ gcc -x c++ hola.c
```

Compila el programa `hola.c` con el compilador de C++, en vez de con el del lenguaje C.

3. Extensiones al lenguaje C

Como ya comentamos en el Acerca de, en este tutorial estamos suponiendo que el lector conoce el lenguaje C, y no vamos a explicar cómo se usa el lenguaje. Lo que vamos a ver en este apartado es qué extensiones al lenguaje C han introducido las GCC. Muchas extensiones al lenguaje que ha introducido inicialmente las GCC han sido adoptadas luego por el estándar ISO, aunque en este apartado vamos a exponer sólo las extensiones que a día de hoy no forman parte del estándar.

Sin no desea usar estas extensiones de las GCC siempre puede usar la opción `-ansi`, `-std=c89` o `-std=c99` para evitar el funcionamiento por defecto que, como se indica en la Tabla 2.3 es `-std=gnu89`. Si además usa la opción `-pedantic` obtendrá un warning cada vez que use una extensión (a no ser que preceda la extensión con la palabra clave `__extension__`).

Debido a que C es la base de otros lenguajes como C++ o Objective-C, muchas de las extensiones que aquí se explican son válidas también en estos otros lenguajes.

3.1. Arrays de longitud variable

En C estándar la longitud de un array debe conocerse en tiempo de compilación, con esta extensión vamos a poder dar la longitud de un array de forma que sea calculada en tiempo de ejecución. El Listado 2.5 muestra un ejemplo de como se hace esto. Observe que la longitud del array `combi` no se conoce hasta el tiempo de ejecución.

```
void Combina(const char* str1, const char* str2)
{
 char combi[strlen(str1)+strlen(str2)+1];
 strcpy(combi,str1);
 strcat(combi,str2);
 printf(combi);
}
```

Listado 2.5: Ejemplo de uso de arrays de longitud variable

También podemos pasar como parámetros de una función arrays de longitud variable, tal como muestra el ejemplo del Listado 2.6.

```
void RellenaArray(int longitud, char letras[longitud])
{
 int i;
 for (i=0;i<longitud;i++)
     letras[i] = 'A';
}
```

Listado 2.6: Array de longitud variable pasado como parámetro

Podemos invertir el orden de los parámetros haciendo una declaración adelantada de la longitud:

```
void RellenaArray(int longitud; char letras[longitud]
                , int longitud)
```

En este caso las declaraciones adelantadas se ponen delante de los parámetros. Puede haber tantas declaraciones adelantadas como queramos, separadas por coma o punto y coma, y acabadas en punto y coma.

3.2. Arrays de longitud cero

Las GCC permiten crear dentro de estructuras arrays de longitud cero. De esta forma podemos crear zonas de memoria de longitud variable a las que accedemos con el array. Aunque no es necesario, estos arrays suelen ser el último campo de la estructura. El programa del Listado 2.7 ilustra el uso de esta técnica. En el ejemplo `sizeof(CadenaVariable)` devuelve 4 porque el array mide 0 bytes para `sizeof()`, pero si luego indireccionamos con `pc->letras[i]` podemos acceder a los bytes siguientes a la memoria ocupada por la estructura.

```
/* arraycero.c */

#include <stdio.h>
#include <string.h>

typedef struct
{
 int tamano;
 char letras[0];
} CadenaVariable;

main()
{
 int i;
 char* cadena = "Hola que tal";
 int longitud = strlen(cadena);
 CadenaVariable* pc = (CadenaVariable*)
                 malloc(sizeof(CadenaVariable)+longitud);
 pc->tamano = longitud;
```

```
 strcpy(pc->letras,cadena);
 for (i=0;i<pc->tamano;i++)
     printf("%c ",pc->letras[i]);
 printf("\n");
 return 0;
}
```

Listado 2.7: Ejemplo de array de longitud cero

El mismo objetivo se puede conseguir definiendo el array sin indicar longitud. Esto no sólo tiene la ventana de ser C estándar, sino que además nos permite indicar los elementos del array en la inicialización. El Listado 2.8 muestra como usar array de longitud indefinida.

```
* arrayindefinido */

#include <stdio.h>
#include <string.h>

typedef struct
{
 int tamano;
 char letras[];
} CadenaVariable;

CadenaVariable c = {12,{'H','o','l','a',' ','q','u','e'
                       ,' ','t','a','l','\0'};

main()
{
 printf("%s ocupa %i bytes para sizeof()\n"
       ,c.letras,sizeof(c));
 return 0;
}
```

Listado 2.8: Ejemplo de array de longitud indefinida

3.3. Rangos en la sentencia `case`

En C estándar podremos definir varios casos en una sentencia `switch` de la forma:

```
case 8:
case 9:
case 10:
case 11:
```

En C de GNU podemos usar la forma alternativa:

```
case 8 ... 11:
```

Es importante encerar la elipsis (. . .) entre espacios, para evitar que el parser confunda el rango con un número en punto flotante. El uso de rango en sentencias case es muy típico encontrarlo en el caso de rangos de caracteres constantes de la forma:

```
case 'a' .. 'm':
```

3.4. Declarar variables en cualquier punto del programa

Si en vez de usar el lenguaje C por defecto (que recuerde que es `-std=gnu89`) usa `-std=c99` o `-std=gnu99` (es decir, si usa es el estándar ISO C99) podrá declarar variables en cualquier punto del programa (incluido dentro de un bucle `for`), y no sólo al principio de un ámbito. Esta forma de declarar es válida siempre en C++, pero en C sólo lo es cuando use ISO C99. El Listado 2.9 muestra un ejemplo de declaración de variables.

```
/* declaraciones.c */

#include <stdio.h>

int main()
{
 printf ("Empezamos:\n");
 char letra = 'A';
 for (int i=0;i<10;i++)
     printf("%c",letra);
 printf("\n");
 return 0;
}
```

Listado 2.9: Ejemplo de declaración de variables en cualquier punto del programa

Recuerde que para compilar el Listado 2.9 deberá usar el comando:

```
$ gcc -std=c99 declaraciones.c
```

3.5. Números largos

El estándar ISO C99 ha definido la forma de crear enteros de dos palabras, es decir, enteros de 64 bits en máquinas de 32 bits, y enteros de 128 bits en máquinas de 64 bits. Para ello añadimos `long long` al tipo, por ejemplo, en una máquina de 32 bits podemos declarar:

```
long long int i; // Entero con signo de 64 bits
unsigned long long int ui; // Entero sin signo de 64 bits
```

Las constantes para estos tipos lo que hacen es llevar el sufijo `LL`. Por ejemplo:

```
i = -7423242050344LL;
ui = 484525254345349 43LL;
```

3.6. Atributos

La palabra clave `__attribute__` se puede poner al final de la declaración de una función, una variable, o un tipo de dato con el fin de personalizar el comportamiento del compilador.

En el caso de las funciones, como vemos en el Listado 2.10, el atributo se pone al final del prototipo (no de la implementación). El Listado 2.10 también muestra como se ponen atributos a las variables de tipos de datos. Como vemos, los atributos también se pueden poner a los campos de una estructura.

```
struct PaqueteDatos
{
 char tipo;
 int padre __attribute__ ((aligned(4)));
};

int TipoError __attribute__ ((deprecated)) = 0;

void ErrorFatal(void) __attribute__ ((noreturn));

void ErrorFatal(void)
{
 printf("Houston, tenemos un problema");
 exit(1);
}

int getLimite() __attribute__((pure,noinline));
```

Listado 2.10: Ejemplo de uso de atributos

Una declaración puede tener varios atributos, en cuyo caso se ponen separados por coma, como en el caso de la función `getLimite()` del Listado 2.10.

La Tabla 2.4 muestra los atributos aplicables a las funciones, la Tabla 2.5 los atributos aplicables a la declaración de variables y la Tabla 2.6 los atributos aplicables a las declaraciones de tipos de datos.

Atributo	Descripción
`deprecated`	Hace que siempre que el compilador detecte una llamada a una función marcada con este atributo

	produzca un mensaje donde se le indique que la función está obsoleta.
`section`	Permite indicar una sección alternativa para la función. Por ejemplo `void foobar(void) __attribute__ ((section("bar")));` indica que la función `foobar()` debe colocarse en la sección `bar`.
`constructor`	Una función marcada con este atributo es ejecutada automáticamente antes de entrar en la función `main()`. Este atributo es usado por los constructores de objetos C++ globales. Véase el apartado 5 del Tema 4.
`desctructor`	Una función marcada con este atributo es ejecutada después de salir de la función `main()`, o después de ejecutar `exit()`.Véase el apartado 5 del Tema 4.
`nonnull`	Permite indicar parámetros de la función de tipo puntero que no pueden ser `NULL` en la llamada. Véase más adelante la explicación.
`format`	Permite que el compilador compruebe los atributos que recibe una función que tiene una cadena de formato. Ver explicación más abajo.
`weak_import`	Permite declarar referencias weak. Su uso lo veremos en el apartado 2.8.3 del Tema 3.
`cdecl`	Estilo de llamada a funciones por defecto de C. Atributo válido sólo en máquinas Intel.
`stdcall`	Estilo de llamada a funciones al estilo Pascal. Atributo válido sólo en máquinas Intel.
`fastcall`	Una forma de llamar a funciones más rápida que la anterior. En concreto los dos primeros parámetros, en vez de pasarse por la pila, se pasan en los registros `ECX` y `EDX`. Atributo válido sólo en máquinas Intel.
`always_inline`	Una función que ha sido declarada con el modificador `inline` no suele expandirse inline a no ser que este tipo de optimización esté activada. Con este atributo la función siempre se expande inline, incluso aunque esté activada la depuración. Véase el apartado 2.1 del Tema 7.
`noinline`	Impide que una función se implemente inline, aunque esté activado este tipo de optimización. Véase el apartado 2.1 del Tema 7.
`pure`	Indica al compilador que esta función no tiene efectos laterales, es decir, que no modifica variables globales, zonas cuya dirección de memoria recibe como parámetro, ni ficheros. A diferencia de con el atributo `const`, esta función si que puede leer estos valores. Este atributo le sirve al optimizador para optimizar subexpresiones. Esta opción permite que la función sea llamada menos veces de lo que dice el programa. Por ejemplo para calcular una raíz cúbica en una expresión

	puede no ser necesario llamar varias veces a la función si el parámetro de la función no cambia.
`const`	Este atributo es parecido a `pure`, pero además de indicar que no modifica, indica que tampoco lee los valores indicados.
`noreturn`	Indica que la función no retorna. Esto permite que el compilador optimice el código de la función al no poner preámbulo de retorno. El Listado 2.10 muestra un ejemplo de uso de este atributo en una función que termina el programa llamando a `exit()`.
`visibility`	Permite indicar la visibilidad de una función a la hora de exportarla en una librería de enlace dinámico. Véase el apartado 2.6.2 del Tema 3.

Tabla 2.4: Atributos aplicables a las funciones

El atributo `nonnull` permite indicar que un parámetro de una función no debería de ser `NULL`. Si pasamos `NULL` en este atributo el compilador lo detecta y emite un warning. Para que el compilador emita el warning es necesario haber pasado la opción `-Wnonnull`, o bien `-Wall` (que es la que se usa para avisar de todo tipo de warnings). En el atributo podemos indicar los parámetros sometidos a este control como muestra el siguiente ejemplo.

```
void* mi_memcpy (void *dest, const void *src, size_t len)
                            __attribute__((nonnull (1, 2)));
```

Si no se indican números de parámetros, todos los parámetros punteros deben de no ser `NULL`. Por ejemplo, en este otro prototipo todos los parámetros punteros deben de no ser `NULL`:

```
void* mi_memcpy (void *dest, const void *src, size_t len)
                                  __attribute__((nonnull));
```

El atributo `format` permite construir funciones tipo `printf()`, es decir, funciones que reciben como parámetro una cadena con el formato, y después un número variable de parámetros. Esto permite que el compilador compruebe que los tipos de la lista variable de parámetros coincidan con la cadena de formato. Por ejemplo, la siguiente función recibe como segundo parámetro una cadena de formato tipo `printf`, y el compilador comprueba que los parámetros a partir del tercero sigan este formato.

```
void MensajeLog(void* log,char* formato,...)
                           __attribute__ ((format(printf,2,3)));
```

Existen distintos tipos de formatos: En concreto los formatos que acepta este atributo son `printf`, `scanf`, `strftime` y `strfmon`.

Atributo	Descripción
`deprecated`	Una variable con este atributo hace que el compilador emita un warning, siempre que el programa intente usarla, indicando que la variable está obsoleta.
`aligned`	Indica que la variable debe de estar en una posición de memoria múltiplo del valor indicado.
`section`	Nos permite indicar que una variable debe crearse en su propio segmento y sección en vez del segmento estándar `DATA` y las secciones estándar `data` y `bbs`. Por ejemplo para crear una variable en el segmento `DATA` y en la sección `eventos` podemos hacer: `int raton __attribute__ ((section("DATA,eventos"))) = 0;`
`visibility`	Permite indicar la visibilidad de una variable a la hora de exportarla en una librería de enlace dinámico. Véase el apartado 2.6.2 del Tema 3.

Tabla 2.5: Atributos aplicables a la declaración de variables

Atributo	Descripción
`deprecated`	Este atributo puesto en un tipo de dato hace que el compilador emita un warning, indicando que la variable está obsoleta, siempre que intente instanciar una variable de este tipo.
`aligned`	Una variable de un tipo con este modificador es una variable que el compilador debe colocar en memoria en una posición múltiplo del número pasado como argumento. Véase el Listado 2.10 para ver un ejemplo de su uso.

Tabla 2.6: Atributos aplicables a las declaraciones de tipos de datos

3.7. Valor de retorno de sentencias compuestas

Un sentencia compuesta es un bloque de sentencias encerradas entre llaves. Cada sentencia compuesta tiene su propio ámbito y puede declarar sus propias variables locales, por ejemplo:

```
{
 int a = 5;
 int b;
 b = a+5;
}
```

En el C de GNU, si encerramos una sentencia compuesta entre paréntesis podemos obtener como valor de retorno el valor de retorno de la última

sentencia, y el tipo del retorno será el mismo que el de la última sentencia del bloque, por ejemplo, el valor de retorno de la siguiente sentencia será `8`:

```
ret = ({
        int a = 5;
        int b;
        b = a+5;
       });
```

Esta construcción es útil para escribir macros. Un conocido problema con los macros se produce cuando un parámetro del macro se usa en más de un lugar de su implementación, por ejemplo, en el siguiente macro:

```
#define siguiente_par(x) ( (x%2==0) ? x : x+1 )
```

El macro funciona bien, hasta que lo ejecutemos de la forma:

```
int par = siguiente_par(n++);
```

Donde se produce el efecto lateras de que `n` se incrementa 2 veces. Una forma de solucionar este efecto lateral sería la siguiente:

```
#define siguiente_par(x) \
     ({ \
       int aux = x; \
       ( (aux%2==0) ? aux : aux+1); \
      })
```

3.8. Operador condicional con operandos omitidos

En el operador condicional se evalúa la condición, y dependiendo de si está se cumple o no se devuelve uno de los operandos que aparecen detrás. Al usar este operador es muy típico usar construcciones donde se compruebe si una variable es cero, por ejemplo:

```
int x = y ? y : z;
```

Comprueba si `y` es distinta de cero, en cuyo caso se asigna a `y` a `x`, y sino se asigna a `x` el valor de `z`.

Para evitar problemas debidos a evaluar dos veces `y`, como el descrito en el apartado anterior, el C de GNU también permite omitir el operando y escribir la expresión anterior de la forma:

```
int x = y ? : z;
```

3.9. Nombre de función como cadena

Además de los identificadores del preprocesador especiales `__FILE__` y `__LINE__`, el C de GNU tiene la palabra reservada `__FUNCTION__` que nos devuelve el nombre de la función en la que nos encontramos.

Esto es útil para hacer cosas como:

```
char* msg = "Es la funcion " __FUNCTION__ " del fichero "
            __FILE__;
```

Actualmente la palabra reservada `__FUNCTION__` esta obsoleta en favor de la palabra reservada `__func__` propuesta por el estándar ISO C99.

3.10. Macros con número variable de argumentos

Existen dos formas de definir macros con número variable de argumentos. Esto se debe a que primero GCC hizo una extensión, y luego el estándar ISO C99 propuso otra distinta.

La forma propuesta por el estándar ISO C99 es:

```
#define msg_error(fmt,...) fprintf(stderr,fmt,__VA_ARGS__);
```

Donde la lista de parámetros del macro que vayan en la elipsis (`...`) se sustituyen en `__VA_ARGS__`.

La sintaxis de la extensión de GNU es esta otra:

```
#define msg_error(fmt,args...) fprintf(stderr,fmt,args);
```

3.11. El operador `typeof`

El C de GNU añade el operador `typeof`, el cual nos devuelve el tipo de una expresión. Su uso es parecido al del operador `sizeof`, pero en vez de devolver una longitud devuelve un tipo. A continuación se muestran algunos ejemplos de su uso:

```
char* str;
typeof(str) str2;   // Un char*
typeof(*str) ch;    // Un char
typeof(str) A1[10]; // Equivale a char* A1[10]
```

El operador `typeof` también puede recibir nombres de tipos:

```
typeof(char*) str;
```

Se estará preguntando para que sirve este operador. Su principal aplicación son los macros, por ejemplo el siguiente macro evita el efecto lateral que explicamos en el apartado 3.7:

```
#define max(a,b) \
       ({ typeof (a) _a = (a); \
          typeof (b) _b = (b); \
          _a > _b ? _a : _b;  })
```

El operador `typeof` también facilita la creación de macros que sirven para definir tipos. Por ejemplo:

```
#define vector(tipo,tamano) typeof(tipo[tamano])
vector(double,10) muestras;
```

Téngase en cuenta que la declaración de variable anterior después de pasar por el preprocesador se convierte en:

```
typeof(double[10]) muestras;
```

Esta forma de declarar un array si que es válida, a pesar de que:

```
double[10] muestras;
```

No sea una forma válida de declarar el array.

El estándar ISO ha introducido también este operador con el nombre `__typeof__`, con lo que actualmente ambas formas son válidas en las GCC.

3.12. Uniones anónimas en estructuras

Esta es una opción definida por el estándar para C++, pero no para C, aunque las GCC la hacen disponible desde el lenguaje C.

Las **uniones anónimas en estructuras** nos permiten definir dos campos dentro de una estructura que ocupan posiciones solapadas de memoria, sin necesidad de dar un nombre a la unión. El Listado 2.11 muestra la forma de definir campos solapados dentro de una estructura de acuerdo al C estándar. Las GCC además permiten usar uniones anónimas dentro de estructuras como muestra el Listado 2.12. En el C estándar para acceder al campo `descriptor` de la unión debemos usar `registro.datos.descriptor`, con la unión anónima usamos la forma `registro.descriptor`.

```
struct
{
 int codigo;
 union
      {
       int descriptor;
       char campos[4];
      } datos;
} registro;
```

Listado 2.11: Ejemplo de unión dentro de estructura de acuerdo al C estándar

```
struct
{
 int codigo;
 union
      {
       int descriptor;
       char campos[4];
      };
} registro;
```

Listado 2.12: Ejemplo de unión anónima dentro de estructura

3.13. Casting a un tipo unión

Otra extensión del C de GNU permite que un tipo de dato que es del mismo tipo que un miembro de una unión, puede hacerse casting explícito al tipo de la unión. El Listado 2.13 muestra un ejemplo en el que un tipo `double` es convertido en un tipo unión, asignando sus 8 bytes a la unión.

```
/* castunion.c */

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

union Partes
{
 unsigned char byte[8];
 double dbl;
};

int main(int argc, char* argv[])
{
 double valor = M_PI;
 union Partes p;
 p = (union Partes) valor;
 return 0;
}
```

Listado 2.13: Ejemplo de casting a un tipo unión

Téngase en cuenta que la conversión sólo se permite de tipo fundamental a unión, no en el sentido contrarío, es decir, la siguiente sentencia fallará:

```
valor = (double) p;
```

4. Warnings

En general siempre se recomienda prestar atención a los warnings y eliminarlos del programa. Esto se debe a que todo programa que produce un warning puede ser reescrito de forma que el warning no se produzca y se consiga el mismo o mayor rendimiento.

El compilador de GCC por defecto no emite todos los mensajes de warning que es capaz de detectar. Esto se debe a que por defecto las GCC compilan C siguiendo el estándar `gnu89` el cual es bastante tolerante con los warnings. Si pasamos al modo `gnu99` (o `c99`), usando la opción `-std=gnu99` (o `-std=c99`), veremos que el número de warnings generado aumenta apareciendo nuevos warnings que la versión `gnu89` había dejado pasar. En cualquier caso no es necesario cambiar de estándar, basta con usar la recomendable opción de línea de comandos `-Wall`, la cual hace que se produzcan los warnings más importantes que las GCC son capaces de detectar.

Como ejemplo de la idoneidad de usar esta opción, el Listado 2.14 muestra un programa que aparentemente está bien hecho.

```
/* programaseguro.c */

#include <stdio.h>

int main(int argc, char* argv[])
{
 printf("2.0 y 2.0 son %d\n",4.0);
 return 0;
}
```

Listado 2.14: Programa con bug

Cuando compilamos el programa anterior no obtenemos mensajes de error, pero al irlo a ejecutar obtenemos un mensaje inesperado:

```
$ gcc programaseguro.c -o programaseguro
$ programaseguro
2.0 y 2.0 son 1074790400
```

Si el programa anterior lo hubiéramos compilado con la opción `-Wall` el bug se hubiera detectado:

```
$ gcc programaseguro.c -Wall -o programaseguro
programaseguro.c: In function `main':
programaseguro.c:8: warning: int format, double arg (arg 2)
```

Es decir, en la cadena de formato deberíamos haber usado la opción `%lf` (numero en punto flotante largo) en vez de `%d` (entero decimal). Este pequeño ejemplo muestra la importancia de usar siempre en nuestros programas la opción `-Wall` (que a partir de ahora vamos a empezar a usar).

Si queremos ser estrictos, podemos usar la opción `-Werror` para que cualquier warning sea considerado un error y no permita compilar el programa.

La opción `-Wall` abarca muchos, pero no todos los posibles warnings que pueden detectar las GCC, aunque sí que abarca los warnings que es más recomendable siempre controlar. La Tabla 2.7 muestra los mensajes de warning que incluye la opción `-Wall`, y la Tabla 2.8 muestra otros mensajes de warning que no incluye la opción `-Wall`.

Warning	Descripción

`-Wcomment`	Esta opción avisa de comentarios tipo C anidados, los cuales son causa frecuente de error. Por ejemplo:
	`/* Probando a comentar esta instrucción`
	`double x = 1.23; /* Coordenada x */`
	`*/`
	Una forma más segura de comentar secciones con comentarios tipo C es usar `#ifdef 0` ... `#endif` de la forma:
	`#ifdef 0`
	`double x = 1.23; /* Coordenada x */`
	`#endif`

`-Wformat`	Esta opción hace que se avise de errores en el formato de funciones como `printf()`, usada en el Listado 2.14.
`-Wunused`	Avisa cuando hay variables declaradas que no se usan.
`-Wimplicit`	Avisa si llamamos a funciones que no tienen prototipo declarado.
`-Wreturn-type`	Avisa cuando una función no tiene declarado en su prototipo un tipo de retorno (en cuyo caso por defecto es `int`), o cuando hemos olvidado retornar con `return` el valor de una función con retorno declarado distinto de `void`.

Tabla 2.7: Warnings incluidos en `-Wall`

Warning	Descripción
`-Wconversion`	Avisa cuando se realiza una conversión de tipos, por ejemplo: `unsigned int x=-1;` No es considerado un warning por `-Wall`, pero si usándo la opción `-Wconversion`.

`-Wsign-compare`	Avisa si se están comparando valores con signo con valores sin signo. Por ejemplo:
	
	`int a = 3;`
	`unsigned int b = 3;`
	`if (a==b)`
	`  printf("Son iguales\n");`
	
	La comparación no produciría un warning usando `-Wall`, pero si usando `-Wsign-compare`.

`-Wshadow`	Esta opción avisa sobre la redeclaración de variables con el mismo nombre en dos ámbitos, por ejemplo, en el siguiente ejemplo `y` está declarada en dos ámbitos, lo cual puede resultar lioso y conducir a errores.
	
	`double Prueba(double x)`
	`{`
	`  double y = 1.0;`
	`  {`
	`  double y;`
	`  y = x;`
	`  }`
	`  return y;`
	`}`

`-Wwrite-string`	Esta opción avisa si intentamos escribir en una cadena. El estándar ISO no define cuál debe ser el resultado de modificar una cadena, y escribir sobre estas está desaconsejado.
`-Wmissing-prototypes`	Avisa si una función no estática es implementada sin haber declarado previamente su prototipo. A diferencia de `-Wimplicit` el warning se produce aunque la función declare su prototipo en la implementación. Esto ayuda a detectar funciones que no están declaradas en ficheros de cabecera.
`-Wtraditional`	Avisa sobre usos de C anteriores al estándar ANSI/ISO 89 que están desaconsejados (por ejemplo funciones sin parámetros).

Tabla 2.8: Warnings no incluidos en `-Wall`

5.Cambiar la versión de las GCC

Actualmente podemos tener instaladas varias versiones de las GCC. Esto permite compilar aplicaciones que compilaban bien con una versión de las GCC, pero debido a cambios en el lenguaje o en las librerías de las GCC, con una versión más reciente el código fuente de la aplicación deja de compilar correctamente.

El comando `gcc_select` nos permite cambiar la versión que estemos usando. Si queremos saber cuál es la versión que estamos usando actualmente, podemos ejecutar este comando sin parámetros:

```
$ gcc_select
Current default compiler:
gcc version 4.0.1 (Apple Computer, Inc. build 5247)
```

Para saber que versiones tenemos instaladas podemos usar la opción `-l` (o `--list`):

```
$ gcc_select -l
Available compiler versions:
3.3             3.3-fast        4.0
```

Para cambiar a otra versión podemos indicar al comando a versión a usar:

```
$ sudo gcc_select 3.3
Default compiler has been set to:
gcc version 3.3 20030304 (Apple Computer, Inc. build 1819)
$ gcc_select
Current default compiler:
gcc version 3.3 20030304 (Apple Computer, Inc. build 1819)
```

Actualmente tanto `gcc` como los demás comandos de las GCC son enlaces simbólicos a ficheros donde se implementa la versión actual del compilador. Por ejemplo `gcc` es un enlace simbólico a `gcc-3.3` o `gcc-4.0` dependiendo de la versión que tengamos actualmente elegida. Podemos preguntar a `gcc_select` qué ficheros modifica al cambiar a una versión con la opción `-n`. Esta opción no modifica la versión actual de las GCC, tan sólo indica los ficheros que se modificarían:

```
$ gcc_select -n 3.3
Commands that would be executed if "-n" were not specified:
  rm -f /usr/bin/gcc
  ln -sf gcc-3.3 /usr/bin/gcc
  rm -f /usr/share/man/man1/gcc.1
  ln -sf gcc-3.3.1 /usr/share/man/man1/gcc.1
  .......
```

6. Resolución de problemas

Podemos obtener una ayuda en línea sobre las opciones soportadas por el comando `gcc` usando el comando:

```
$ gcc --help
```

Este comando sólo nos muestra las principales opciones, si queremos una descripción detallada de todas las opciones del comando `gcc` podemos usar:

```
$ gcc -v --help
```

La opción `-v` también resulta muy útil para seguir los pasos que sigue el driver para generar un programa. Muchas veces esto nos ayuda a identificar el punto exacto donde falla la compilación. A continuación se muestra el resultado de ejecutar este comando en la máquina del autor:

```
$ gcc -v hola.c
Using built-in specs.
Target: powerpc-apple-darwin8
Configured with: /private/var/tmp/gcc/gcc-
5026.obj~19/src/configure --disable-checking --prefix=/usr --
mandir=/share/man --enable-language s=c,objc,c++,obj-c++ --
program-transform-name=/^[cg][^+.-]*$/s/$/-4.0 / --with-gxx-
include-dir=/include/gcc/darwin/4.0/c++ --build=powerpc-apple-
darwin8 --host=powerpc-apple-darwin8 --target=powerpc-apple-
dar win8
Thread model: posix
gcc version 4.0.0 (Apple Computer, Inc. build 5026)
 /usr/libexec/gcc/powerpc-apple-darwin8/4.0.0/cc1 -quiet -v -
D__DYNAM IC__ hola.c -fPIC -quiet -dumpbase hola.c -auxbase
hola -version -o / var/tmp//ccFHi4To.s
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/lib/gcc/powerpc-apple-darwin8/4.0.0/include
 /usr/include
 /System/Library/Frameworks
 /Library/Frameworks
End of search list.
GNU C version 4.0.0 (Apple Computer, Inc. build 5026)
(powerpc-apple-darwin8)
   compiled by GNU C version 4.0.0 (Apple Computer, Inc. build
5026).
as -arch ppc -o /var/tmp//ccozRXSa.o /var/tmp//ccFHi4To.s
 /usr/libexec/gcc/powerpc-apple-darwin8/4.0.0/collect2 -
dynamic -arch ppc -weak_reference_mismatches non-weak -o a.out
-lcrt1.o /usr/lib/gcc/powerpc-apple-darwin8/4.0.0/crt2.o -
var/tmp//ccozRXSa.o -lgcc -lgcc_eh -lSystemStubs -lmx -lSystem
```

Tema 3

Crear y usar librerías

Sinopsis:

Las librerías nos permiten agrupar grandes cantidades de código en un sólo fichero reutilizable desde otros programas. En este tema vamos a estudiar tanto la creación como el uso de librerías.

El tema empieza detallando como crear y usar librerías de enlace estático, después haremos el mismo recorrido sobre las librerías de enlace dinámico, y por último veremos los frameworks, un tipo de librerías de enlace dinámico en las que junto a la librería de enlace dinámico encontramos otros recursos.

1. Librerías de enlace estático

Una **librería de enlace estático**, también llamada **librería estática** o **archivo**, es un conjunto de ficheros `.o` generados por el compilador como normalmente. Enlazar un programa con un fichero de código objeto de la libraría es equivalente a enlazarlo con un fichero de código objeto en el directorio.

A las librerías de enlace estático también se las llama archivos porque el comando usado para generarlas es `ar`, como vamos a ver a continuación.

1.1. Creación y uso de librerías de enlace estático

Para construir una librería primero necesitamos compilar los módulos en ficheros de código objeto. Como ejemplo vamos a usar los ficheros de código fuente del Listado 3.1 y Listado 3.2.

```
/* saludo1.c */

#include <stdio.h>

void Saludo1()
{
 printf("Hola por primera vez\n");
}
```

Listado 3.1: Función que saluda una vez

```
/* saludo2.c */

#include <stdio.h>

void Saludo2()
{
 printf("Hola de nuevo\n");
}
```

Listado 3.2: Función que saluda de nuevo

Para obtener los ficheros de código objeto correspondientes usamos el comando:

```
$ gcc saludo1.c saludo2.c -c
```

Ahora podemos usar el comando `ar` con la opción `-r` (replace and add) para crear un archivo con los ficheros de código objeto dentro de él. El comando, crea el archivo si no existe, y añade (o reemplaza si existen) los ficheros

indicados. Por ejemplo, para crear el fichero `libsaludos.a` usamos el comando:

```
$ ar -r libsaludos.a saludo1.o saludo2.o
ar: creating archive libsaludos.a
```

La librería ya está lista para ser usada. El Listado 3.3 muestra un programa que usa la librería.

```
/* saludos.c */

void Saludo1();
void Saludo2();

main()
{
 Saludo1();
 Saludo2();
 return 0;
}
```

Listado 3.3: Programa que ejecuta las funciones de la librería

Para compilar el programa del Listado 3.3 junto con la librería podemos usar el comando:

```
$ gcc saludos.c libsaludos.a -o saludos
```

Si la librería hubiera estado en otro directorio, hubiera bastado con indicar la ruta completa del fichero de librería.

Aunque no es estrictamente obligatorio, si queremos usar la opción del enlazador `-l` para indicar librerías con las que enlazar el programa, los ficheros de las librerías estática deben tener un nombre que empiece por `lib` y acabar en `.a`. Si los ficheros siguen esta convención podemos acceder a ellos de la forma:

```
$ gcc saludos.c -lsaludos -o saludos
ld: can't locate file for: -lsaludos
```

En este caso el comando enlazador `ld` ha buscado un fichero llamado `libsaludos.a`, pero no lo ha encontrado porque no se encuentra en los directorios reservados para almacenar librerías, ni en la variable de entorno `LIBRARY_PATH`.

En la mayoría de los SO (incluido Mac OS X), los directorios predefinidos para librerías son, por este orden de búsqueda:

```
/usr/local/lib
/usr/lib
```

Para copiar la librería a estos directorios debe tener permiso de administrador, pero puede usar la opción del enlazador `-L` (mayúscula) para indicar directorios adicionales donde el enlazador debe buscar librerías:

```
$ gcc saludos.c -L. -lsaludos -o saludos
```

La opción `-L` (al igual que la opción `-I`) debe usarse una vez por cada directorio que se quiera añadir a la lista de directorios de librerías.

Quizá le resulte más cómodo incluir el directorio actual en la variable de entorno `LIBRARY_PATH`, con el fin de no tener que usar la opción `-L` siempre que quiera enlazar con la librería:

```
$ export LIBRARY_PATH=$LIBRARY_PATH:.
$ gcc saludos.c -lsaludos -o saludos
```

Por último comentar que en Mac OS X (y en otros muchos sistemas UNIX) existe el comando `libtool`[1] el cual nos permite crear una librería de enlace estático (opción `-static`) o de enlace dinámico (opción `-dynamic`). La librería que antes generamos con el comando `ar` también la podemos generar con el comando `libtool` de la forma:

```
$ libtool -static saludo1.o saludo2.o -o libsaludos.a
```

Si no pasamos opción a `libtool` por defecto asume `-static`, aun así es recomendable indicar la opción por claridad.

1.2. La tabla de símbolos

En un fichero de librería estática, delante de los ficheros de código objeto se almacena una **tabla de símbolos**, que sirve para indexar todos los símbolos globales (variables globales y funciones) de la librería. Antiguamente esta tabla de símbolos había que crearla con el comando `ranlib`, pero actualmente el comando `ar` crea automáticamente la tabla de símbolos, con lo que el comando `ranlib` ha pasado a marcarse como un comando obsoleto que se mantiene por compatibilidad.

Cuando el enlazador elige un símbolo de la tabla de símbolos para introducir su código en el programa, el enlazador incluye todo el fichero de código objeto donde esté (aunque el programa no use todo su contenido). Si ningún símbolo de un fichero de código objeto es usado por el programa, el enlazador no incluye este fichero de código objeto en el ejecutable. Esta última regla es importante tenerla en cuenta cuando se crear librerías, ya que

[1] El comando `libtool` que encontrará en Mac OS X lo crearon los ingenieros de Apple, y no es exactamente igual al comando `libtool` de GNU, aunque su finalidad es semejante

si un fichero de código objeto hace llamadas a otro fichero de código objeto de la librería (pero el programa no hace ninguna referencia a este último), el usuario de nuestra librería obtendrá mensajes de error indicando que hay símbolos no resueltos durante el enlazado.

Ya que los ficheros de código objeto no se enlazan si no se usan, un programa que use librerías estáticas puede ocupar menos espacio que el correspondiente fichero enlazado directamente con los ficheros de código objeto sin empaquetar en librería (que siempre se enlazan aunque no se usen).

Podemos usar el comando `nm` para ver la tabla de símbolos de un fichero `.o` o `.a` de la forma:

```
$ nm saludo1.o
00000000 T _Saludo1
         U _printf
         U dyld_stub_binding_helper
```

En el caso de que lo hagamos sobre una librería (fichero `.a`) nos muestra los símbolos de cada fichero:

```
$ nm libsaludos.a
libsaludos.a(saludo1.o):
00000000 T _Saludo1
         U _printf
         U dyld_stub_binding_helper

libsaludos.a(saludo2.o):
00000000 T _Saludo2
         U _printf
         U dyld_stub_binding_helper
```

1.3. Modificar y borrar elementos de una librería

Podemos ver que ficheros `.o` contiene una librería con el comando `-t`:

```
$ ar -t libsaludos.a
__.SYMDEF SORTED
saludo1.o
saludo2.o
```

`__.SYMDEF SORTED` es la tabla de símbolos que crea `ar` para evitar que se introduzcan símbolos duplicados (en varios ficheros `.o`), y para acelerar el acceso a los símbolos por parte del enlazador.

También podemos preguntar por todos los atributos de los ficheros con la opción `-v` (verbose):

```
$ ar -tv libsaludos.a
rw-r--r-- 503/503   48 Dec  4 21:03 2005 __.SYMDEF SORTED
rw-r--r-- 503/80   816 Dec  4 17:59 2005 saludo1.o
rw-r--r-- 503/80   808 Dec  4 17:59 2005 saludo2.o
```

O bien podemos extraer los ficheros de una librería con el comando `-x` (eXtract):

```
$ ar -x libsaludos.a
```

Que extrae todos los ficheros `.o` al directorio actual. O bien indicar el fichero a extraer:

```
$ ar -x libsaludos.a saludo1.o
```

La operación de extraer no borra los ficheros de la librería, sino que si queremos borrar unos determinados ficheros debemos usar la opción `-d` de la forma:

```
$ ar -d libsaludos.a saludo1.o
$ ar -t libsaludos.a
__.SYMDEF SORTED
saludo2.o
```

Donde hemos eliminado el fichero `saludo1.o` de la librería.

1.4. Juntar módulos

Aunque una librería está formada por módulos, a veces conviene combinar varios módulos de código objeto en uso sólo, ya que la llamadas entre funciones de distintos módulos es un poco más lenta que entre funciones del mismo módulo. Para realizar esta tarea el comando `ld` puede recibir varios ficheros de código objeto y generar otro con los módulos agrupados en uno sólo. Si hacemos esto es importante acordarse de usar la opción `-r` para que se conserve la información del módulo objeto necesaria para que éste puede luego volver a ser enlazado, ya que `ld` elimina por defecto esta información de su salida, a la que se llama **información de reasignación**.

En el ejemplo anterior vemos que si preguntamos por la composición del fichero usando el comando `nm` obtenemos:

```
$ nm libsaludos.a
libsaludos.a(saludo1.o):
00000000 T _Saludo1
         U _printf$LDBLStub
         U dyld_stub_binding_helper
```

```
libsaludos.a(saludo2.o):
00000000 T _Saludo2
         U _printf$LDBLStub
         U dyld_stub_binding_helper
```

Si ahora combinamos los ficheros de código objeto `saludo1.o` y `saludo2.o` en uno sólo:

```
$ ld -r saludo1.o saludo2.o -o saludo12.o
```

Y volvemos a generar la librería, la nueva librería estará formada por un sólo módulo:

```
$ $libtool -static saludo12.o -o libsaludos.o
$ $nm libsaludos.o
libsaludos.o(saludo12.o):
00000000 T _Saludo1
0000003c T _Saludo2
         U _printf$LDBLStub
         U dyld_stub_binding_helper
```

2. Librerías de enlace dinámico

Una **librería de enlace dinámico**, también llamada **librería compartida**, contiene ficheros de código objeto que se cargan en memoria, y se enlazan con el programa, cuando el programa que usa sus funciones accede a una de ellas por primera vez.

La forma de funcionar de las librerías de enlace dinámico tiene dos ventajas sobre las librerías de enlace estático: Por un lado la misma librería puede estar cargada en memoria una vez y compartida por varias aplicaciones, lo cual reduce el consumo de memoria[1]. Esta es la razón por la que también se llaman librerías compartidas. Por ejemplo, el Foundation y el Application framework son librerías de enlace dinámico compartidas por todas las aplicaciones Cocoa. La segunda ventaja es que las aplicaciones se actualizan automáticamente cuando se actualizan las librerías. Por ejemplo, si una librería tenia un bug por el que una función fallaba o tenia grandes retrasos, si la función se arregla, o optimiza, la aplicación se aprovecha de esta mejora.

También esta forma de funcionar tiene dos inconvenientes: El primero es que como la librería tiene que cargarse, y enlazarse al programa, la primera vez que se usa un símbolo de la librería, el programa sufre pequeñas paradas (latencias) en mitad de su ejecución, y el segundo inconveniente es que cuando se actualizan las librerías de enlace dinámico, si no se hace con cuidado[2] pueden producirse errores en los programas antiguos que las usen.

2.1. Como funcionan las librerías de enlace dinámico

Cuando se ejecuta una aplicación, el núcleo de Mac OS X carga el código y datos de la aplicación en el espacio de memoria del nuevo proceso. El núcleo además carga el llamado **cargador dinámico (dynamic loader)** (`/usr/lib/dyld`) en la memoria del proceso, y pasa el control a éste. En este momento el cargador dinámico carga las librerías de enlace dinámico que el programa utilice. Durante el proceso de enlazado de la aplicación el enlazador estático, es decir el comando `ld`, guardó dentro de la aplicación las rutas de las librerías de enlace dinámico que utilizaba. Si el cargador dinámico detecta que alguna de las librerías de las que depende el ejecutable falta, el proceso de arranque de la aplicación falla.

[1] Las librerías de enlace dinámico se mapean en memoria en el modo copy-on-write de forma que el segmento de código siempre es compartido por todas las aplicaciones que usen la librería. Si una librería escribe en el segmento de datos, se crea otra copia de este segmento para la aplicación, y todas las demás aplicaciones siguen usando el anterior segmento de datos.

[2] En el apartado 2.2 veremos que siguiendo unas sencillas reglas estos problemas nunca se produciran.

Como veremos, el programador también puede decidir no usar el cargador dinámico para cargar las librerías de enlace dinámico automáticamente durante el arranque de la aplicación, sino cargarlas sólo cuando pasemos por el trozo del programa que las usa. Para ello, veremos que se usan funciones de la familia `dlopen()`. Esto permite acelerar el proceso de arranque de las aplicaciones, y evita cargar en memoria librerías que no vamos a usar, aunque requiere más trabajo por parte del programador.

2.2. Compatibilidad entre versiones

2.2.1. Compatibilidad hacia atrás y hacia adelante

Desde el punto de vista de la aplicación que usa una librería de enlace dinámico existen dos tipos de compatibilidad que conviene concretar.

Se dice que una librería es **compatible hacia atrás** (o que suplanta a una versión anterior) cuando podemos cambiar la librería y las aplicaciones enlazadas con la antigua librería siguen funcionando sin modificación. Para conseguir compatibilidad hacía atrás debemos de mantener tanto las funciones antiguas como las nuevas.

Se dice que una librería es **compatible hacia adelante** (o que suplanta a una versión posterior) cuando es capaz de ser usada en lugar de versiones posteriores de la librería, y las aplicaciones enlazadas con librerías futuras siguen pudiendo enlazar con la librería actual. Lógicamente la compatibilidad hacia adelante es más difícil de conseguir porque la librería tiene que predecir que cambios se harán en el futuro en la librería.

2.2.2. Versiones mayores y menores

Existen dos tipos de revisiones que se pueden hacer a una librería: Revisiones que son compatibles con las aplicaciones clientes, es decir, que no requieren cambios en las aplicaciones clientes, y revisiones que requieren que las aplicaciones cliente se vuelvan a enlazar con la nueva librería de enlace dinámico. Las primera son las llamadas **versiones menores (minor versions)**, y las segundas las **versiones mayores (mayor versions)**.

Las versiones mayores se producen cuando necesitamos modificar algún aspecto de la librería que impliquen eliminar símbolos públicos de la librería o cambiar los parámetros o el funcionamiento de alguna función de librería.

Las versiones menores se producen cuando arreglamos algún bug que hacía fallar a la aplicación, mejoramos el rendimiento de las funciones, o añadimos nuevos símbolos que la versión anterior no tenía.

Los nombres de las librerías de enlace dinámico suelen empezar por `lib` y acabar en `.dylib`[1], es decir, tienen la forma `lib`*`xxx`*`.dylib` donde sólo *`xxx`* es variable. Cuando realizamos una versión mayor debemos cambiar el nombre de la librería incluyendo el nuevo número de versión mayor en el nombre de la librería. Por ejemplo si tenemos el fichero `libSaludos.A.dylib`, y hacemos una versión mayor el siguiente fichero podría ser `libSaludos.B.dylib`. La forma de numerar las librerías no está estandarizada, podrían por ejemplo haberse llamado `libSaludos.1.dylib` y `libSaludos.2.dylib`.

Para facilitar el cambio de versión mayor es muy típico crear un enlace simbólico de la forma `libSaludos.dylib` que apunte a la última versión mayor (p.e. `libSaludos.B.dylib`).

Lógicamente crear versiones mayores consume más espacio en disco, y es algo que debe de intentar evitarse en la medida de lo posible.

Cuando se enlaza un programa con una librería, el enlazador estático guarda dentro del ejecutable el path de la librería al que apunta el enlace simbólico. El cargador dinámico usa este path durante el arranque de la aplicación para encontrar la versión mayor de la librería. Luego las versiones mayores sirven para que la librería pueda tener compatibilidad hacia atrás, de forma que al actualizar la librería los programas que enlazaban con el anterior fichero de librería sigan funcionando.

Dentro de la versión mayor de una librería podemos designar también números de versión, que son números incrementales de la forma *`x`*`[.`*`y`*`[.`*`z`*`]]`, conde *`x`* es un número entre 0 y 65535, y *`y`*, *`z`* son números entre 0 y 255. Cada fichero de versión mayor tiene dos números de versión menor: El **número de versión actual** y el **número de versión de compatibilidad**. La unión de estos dos números es lo que llamaremos la **versión menor**. Dentro de una versión mayor sólo puede haber una versión menor, una nueva versión menor simplemente sobrescribe un número de versión anterior. Esto difiere del esquema de las versiones mayores, donde muchas versiones mayores pueden coexistir (en distintos ficheros).

Cada vez que actualizamos cualquier aspecto de la librería debemos de cambiar el número de versión actual. Por contra, el número de versión de compatibilidad sólo se cambia cuando cambiamos la interfaz pública de la librería (p.e. añadiendo una clase o una función). En este caso el número de versión de compatibilidad debe ponerse al mismo valor que el número de versión actual. En caso de que el cambio sea sólo arreglar un bug o mejorar el rendimiento de la librería, pero no modifiquemos la interfaz pública, sólo

[1] Esto es una peculiardad de Mac OS X, en la mayoría de los sístemas UNIX las librerías de enlace dinámico tienen la extensión `.so` y funcionan de forma ligeramente diferente. Mac OS X también soporta la extensión `.so` por compatibilidad.

debemos actualizar el número de versión actual. Luego el número de versión de compatibilidad es un número que indica el número de versión actual más antigua con la que la aplicación cliente que enlaza con nuestra librería puede ser enlazada al ser ejecutada la aplicación.

Para indicar a `gcc` el número de versión actual se usa la opción del compilador `-current_version` cuando enlazamos la librería. Para indicar el número de versión de compatibilidad se usa la opción `-compatibility _versión` cuando enlazamos la librería. Por ejemplo, si nuestro número de versión actual de la librería es `-current_version 1.2`, y el número de versión de compatibilidad es `-compatibility_version 1.1`, indicamos que las aplicaciones clientes no pueden ser enlazadas con versiones anteriores a la 1.1.

Hay que tener en cuenta que las versiones menores siempre son compatibles hacia atrás, o de lo contrario deberíamos de haber actualizado el número de versión mayor. Además utilizando inteligentemente las versiones menores podemos conseguir además la compatibilidad hacia adelante.

En principio el número de versión actual no es suficiente para garantizar la compatibilidad hacía adelante, es decir, una aplicación que está enlazada por el enlazador estático con la versión actual 1.1, va a funcionar si al ejecutarla el cargador dinámico la enlaza con la versión actual 1.2, pero al revés no tiene porque ser necesariamente cierto, ya que la versión actual 1.2 puede tener nuevas funciones que la versión actual 1.1 no tenga.

Para conseguir compatibilidad hacía adelante se usa la versión de compatibilidad. Cuando el enlazador estático asocia una librería con la aplicación, almacena en la aplicación el número de versión de compatibilidad de la librería. Antes de cargar la librería de enlace dinámico el cargador dinámico compara la versión menor actual de la librería existente en el sistema de ficheros del usuario, con la versión de compatibilidad del fichero de librería con la que fue enlazada la aplicación. Si la versión actual de la librería es menor a la versión de compatibilidad guardada en la aplicación, la carga de la librería falla. Los casos en los que una librería es demasiado antigua no son comunes pero tampoco imposibles. Ocurren cuando el usuario intenta ejecutar la aplicación en un entorno con una versión antigua de la librería. Cuando esta condición ocurre, el cargador dinámico detiene la carga de la aplicación y da un mensaje de error en consola.

El comando `cmpdylib` se puede usar para comprobar si dos librerías de enlace dinámico son compatibles entre sí.

Es importante recordar que el cargador dinámico compara la versión de compatibilidad de la aplicación con la versión actual de la librería, pero esto no lo hacen las funciones de la familia `dlopen()`, que se usan para cargar librerías en tiempo de ejecución.

2.3. Creación y uso de librerías

El código objeto de las librerías de enlace dinámico debe ser código **PIC (Position Independent Code)**, que es un código preparado para poder ser cargado en cualquier posición de memoria, lo cual significa que todas las direcciones de todas sus símbolos globales (variables y funciones) debe de ser relativas, y no absolutas, con el fin de que la librería pueda ser cargada y sus símbolos accedidos en tiempo de ejecución. En el apartado 2.10 detallaremos mejor este proceso.

Como muestra la Figura 3.1, el programa que se encarga de cargar en memoria un ejecutable y ponerlo en ejecución es el cargador dinámico, su nombre proviene de que no sólo es capaz de cargar en memoria ejecutables, sino que también sabe cargar librerías dinámicas.

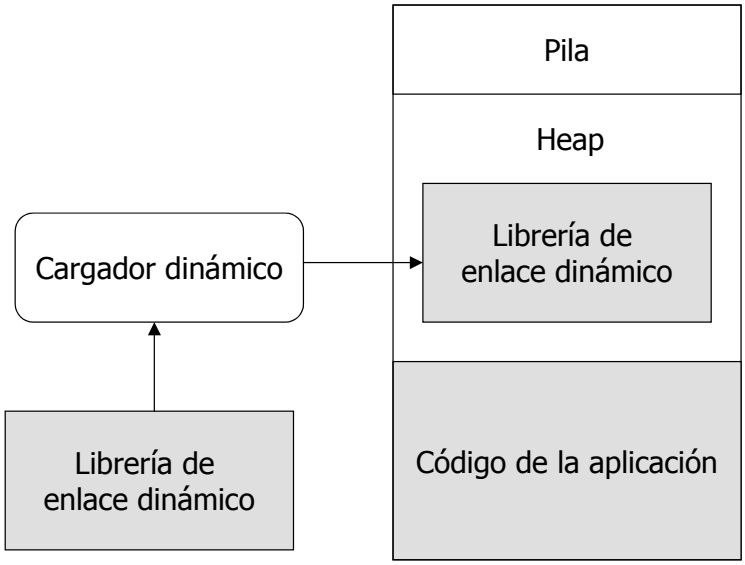


Figura 3.1: Carga de una librería de enlace dinámico en la memoria del proceso

Para generar este tipo de código se usa la opción `-fPIC`[1]. Explicados estos conceptos, vamos a ver como se crear librerías de enlace dinámico, para lo cual vamos a volver a usar los ficheros del Listado 3.1 y del Listado 3.2. Empezamos compilando el código fuente en ficheros de código objeto PIC con el comando:

```
$ gcc -c -fPIC saludo1.c saludo2.c
```

Para ahora enlazar la librería compartida debemos usar la opción `-dynamiclib` de la forma:

[1] En el caso de Mac OS X la opción `-fPIC` no es necesaria ya que por defecto el driver pone esta opción en todos los ficheros que compila.

```
$ gcc -dynamiclib saludo1.o saludo2.o -o libSaludos.dylib
```

Ya sólo nos queda llamar a la librería desde un programa como el del Listado 3.3. Para ello podemos simplemente enlazar el ejecutable con la librería usando:

```
$ gcc saludos.c libSaludos.dylib -o saludos
```

Podemos ver las librerías de enlace dinámico de que depende nuestro ejecutable con el comando `otool` y la opción `-L`[1]:

```
$ otool -L saludos
saludos:
    libSaludos.dylib (compat ver 0.0.0, current ver 0.0.0)
    /usr/lib/libmx.A.dylib (compat ver 1.0.0, current ver
92.0.0)
    /usr/lib/libSystem.B.dylib (compat ver 1.0.0, current ver
88.1.2)
```

Como ya sabemos, `gcc` es un driver que se encarga de llamar a los comandos adecuados para generar el ejecutable o librería deseados. La opción `-dynamiclib` de `gcc` lo que produce es, que en vez de ejecutarse el enlazador estático (comando `ld`), se ejecute el comando `libtool -dynamic`, el cual produce la librería de enlace dinámico. Es decir la librería anterior también la podríamos haber generado con el comando:

```
$ libtool -dynamic saludo1.o saludo2.o -lSystemStubs -lSystem
-o libSaludos.dylib
```

Aun así resulta más conveniente usar el comando `gcc`, ya que éste se encarga de pasar a `libtool` las opciones adecuadas. Por ejemplo, `gcc` sabe que al llamar a `libtool` le debe pasar las opciones `-lSystemStubs` y `-lSystem` para que enlace correctamente con las librerías que usa nuestro programa, con lo que `gcc` nos resuelve el problema de forma transparente.

2.3.1. Dependencias entre librerías

Cuando creamos una librería de enlace dinámico que usa otras librerías de enlace dinámico, los nombres de las librerías de las que depende nuestra librería se guardan como nombres de instalación dentro de nuestra librería. Cuando una aplicación usa nuestra librería, no sólo se carga nuestra librería, sino también las librerías de las que nuestra librería depende. Al igual que con la aplicación, podemos ver las librerías de enlace dinámico de las que depende una librería con el comando `otool -L`.

[1] En los sistemas UNIX tradicionales se usa `ldd` para encontrar las dependencias. Por contra en Mac OS X no existe este comando sino que en su lugar se usa `otool -L`.

Tenga en cuenta que lógicamente, la carga de las librerías de enlace dinámico por parte del cargador dinámico durante el arranque de la aplicación enlentece el arranque. Si estamos creando una librería de enlace dinámico (o aplicación) que usa una librería de enlace dinámico en escasas ocasiones, quizá sería bueno que se plantease el uso de las funciones de la familia `dlopen()` que se explican en el apartado 2.5.

Aunque la imagen de las librerías es cargada en memoria (por parte del cargador dinámico) durante el arranque de la aplicación, como muestra la Figura 3.1, las referencias a los símbolos exportados utilizan **enlace tardio (lazy binding)**, que significa que la dirección del símbolo no es resuelta hasta que se vaya a usar por primera vez. En el apartado 2.8 se describe con más detalle este proceso.

2.4. Instalar la librería

En este apartado vamos a ver cómo hace el cargador dinámico para buscar las librerías de enlace dinámico.

2.4.1. El nombre de instalación

Las librerías de enlace dinámico tienen un **nombre de instalación**, que indica en que ruta va a estar instalado el fichero de la librería cuando lo vayan a usar las aplicaciones.

Al ejecutar `otool` en el apartado anterior podemos ver que la librería `libSaludos.dylib` no tiene ruta de instalación, lo cual significa que debe de estar instalada en el mismo directorio que la aplicación. Debido a que la finalidad de las librerías es ser usadas por varias aplicaciones lo normal es que la librería se encuentre en un directorio de librería (típicamente `/usr/lib`, `/usr/local/lib` o `~/lib`), y que varias aplicaciones enlacen con esa misma ruta.

Para indicar el nombre de instalación se usa la opción `-install_name` como muestra el siguiente ejemplo:

```
$ gcc -dynamiclib saludo1.o saludo2.o -install_name /usr/local
/lib/ libSaludos.dylib -o libSaludos.dylib
```

El nombre de instalación se almacena dentro del fichero de librería, y también se almacena dentro del fichero de la aplicación cliente con el fin de que el cargador dinámico pueda acceder a este fichero cuando la aplicación se ejecute.

Podemos obtener el nombre de instalación de una librería con la opción `-D` de `otool`:

```
$ otool -D libSaludos.dylib
/usr/local/lib/libSaludos.dylib
```

Si ahora volvemos a compilar la aplicación, y volvemos a preguntar a `otool` por las librerías de enlace dinámico de la aplicación:

```
$ gcc saludos.c libSaludos.dylib -o saludos
$ otool -L saludos
saludos:
     /usr/local/lib/libSaludos.dylib (compat ver 0.0.0,
current ver 0.0.0)
     /usr/lib/libmx.A.dylib (compat ver 1.0.0, current ver
92.0.0)
     /usr/lib/libSystem.B.dylib (compa ver 1.0.0, current ver
88.1.2)
```

Obtenemos el nombre de instalación como una ruta absoluta. Pero si ahora intentamos ejecutar la aplicación:

```
$ saludos
dyld: Library not loaded: /usr/local/lib/libSaludos.dylib
Reason: image not found
Trace/BPT trap
```

Su ejecución falla ya que ahora busca la librería en la ruta de su nombre de instalación, y no en el directorio actual. Lógicamente siempre podemos copiar la librería en la ruta de su nombre de instalación, y poder así ejecutar la aplicación:

```
$ sudo cp libSaludos.dylib /usr/local/lib
$ saludos
Hola por primera vez
Hola de nuevo
```

2.4.2. Proceso de búsqueda de librerías

El proceso que sigue el cargador dinámico para buscar librerías de enlace dinámico es el siguiente:

Cuando el nombre de instalación de la librería es una ruta relativa (p.e. `libSaludos.dylib`), el cargador dinámico busca las librerías a cargar en este orden:

1. En los directorios indicados por la variable de entorno `$LD_LIBRARY_PATH`.
2. En los directorios indicados por la variable de entorno `$DYLD_LIBRARY_PATH`.
3. En el directorio de trabajo de la aplicación.

4. En los directorios indicados por la variable de entorno `$DYLD_FALLBACK_LIBRARY_PATH`.

Por contra, cuando el nombre de instalación de la librería es una ruta absoluta (p.e. `/usr/local/lib/libSaludos.dylib`), el cargador dinámico busca las librerías a cargar en este otro orden:

1. En los directorios indicados por la variable de entorno `$DYLD_LIBRARY_PATH`.
2. En la ruta dada por el nombre de instalación.
3. En los directorios indicados por la variable de entorno `$DYLD_FALLBACK_LIBRARY_PATH`.

Como habitualmente, las variables de entorno pueden tener la ruta de uno o más directorios de búsqueda separados por dos puntos. Por defecto estas variables no tienen valor asignado, aunque el administrador puede fijar estas variables en scripts de arranque para buscar librerías en directorios no estándar.

La variable de entorno `$LD_LIBRARY_PATH` es la recomendada para las librerías con extensión `.so`, que es la extensión usada por la mayoría de los sistemas UNIX. Por contra `$DYLD_LIBRARY_PATH` es la recomendada para las librerías con extensión `.dylib`.

Apple recomienda usar librerías con la extensión `.dylib`, aunque esto no es obligatorio sino opcional. En Mac OS X existe la opción de `gcc` no estándar `-install_name` para poder guardar el nombre de instalación con la ruta relativa o absoluta de la librería. El uso de esta opción acelera el proceso de carga de la librería, ya que el cargador dinámico no tiene que buscar la librería, sabe exactamente donde se encuentra el fichero.

En los sistemas UNIX tradicionales no existe el nombre de instalación y se buscan librerías de enlace dinámico en los directorios por defecto (que son `/usr/lib` y `/usr/local/lib`), y después en los directorios indicados por la variable de entorno `$LD_LIBRARY_PATH`. Para mantener compatibilidad con estos casos, y con los casos en los que el programador de la librería no indica un nombre de instalación (usando la opción `-install_name`) existen las reglas de búsqueda en ruta relativa.

El orden en el proceso de búsqueda de librerías es importante porque si un mismo símbolo está definido en dos librerías, el cargador dinámico cargaría la primera librería que usase este símbolo. Por ello es recomendable al crear librerías poner a todos los símbolos públicos un prefijo que evite conflictos con símbolos de otras librerías. En el caso de las funciones de la familia `dlopen()`, existen opciones para poder buscar todas las librerías que definen un símbolo.

Otra información importante que se almacena en las librerías es la versión requerida de las librerías de que dependa. En tiempo de ejecución el cargador dinámico comprobará que las versiones de las librerías cargadas sean correctas.

2.5. Carga de librerías en tiempo de ejecución

Normalmente cada sistema UNIX implementa la carga dinámica de librerías de enlace dinámico de forma diferente. Esto ha dificultado la portabilidad de aplicaciones entre sistemas UNIX. Para facilitar la portabilidad de librerías de enlace dinámico Jorge Acereda y Peter O'Gorman desarrollaron un conjunto de funciones llamadas **Dynamic Loader Compatibility (DLC)**. En concreto se trata de cinco funciones definidas en el fichero `/usr/include/dlfcn.h`, y que se resumen en la Tabla 3.1.

Función	Descripción
`dlopen()`	Abre una librería de enlace dinámico. Una aplicación debe llamar a esta función antes de usar cualquier símbolo exportado de la librería. Si la librería no está abierta por el proceso llamante, la librería se carga en la memoria del proceso. La función retorna un handle que usarán `dlsym()` y `dlclose()`. La función mantiene la cuenta del número de veces que se ha pedido abrir cada librería.
`dlclose()`	Usado por el proceso para cerrar el handle. Si `dlopen()` se ha llamado varias veces, esta función deberá ser llamada el mismo número de veces. En caso de que la librería fuera cargada en memoria por `dlopen()` (y no por el cargador dinámico), la última llamada a `dlclose()` descarga la librería de enlace dinámico de la memoria del proceso.
`dlsym()`	Retorna la dirección de un símbolo exportado por una librería de enlace dinámico.
`dladdr()`	Recibe una dirección de memoria y, si ésta corresponde con la dirección de memoria de una variable o función de la librería, devuelve información sobre este símbolo en una variable de tipo `Dl_info` con información sobre la variable o función.
`dlerror()`	Devuelve una cadena con una descripción del error producido en la última llamada a `dlopen()`, `dlsym()` o `dladdr()`.

Tabla 3.1: Funciones de DLC

El Listado 3.4 muestra como podemos cargar la librería `libSaludos.dylib`, y ejecutar una función de ésta, usando las funciones de la familia `dlopen()`.

```
/* saludosdl.c */

#include <stdio.h>
#include <dlfcn.h>

void Saludo1();
void Saludo2();

main()
{
 // Abre la libreria
 void* l = dlopen("libSaludos.dylib",0);
 if (l==NULL)
    {
     printf("Error abriendo la libreria: %s\n",dlerror());
     return 1;
    }
 // Obtiene un puntero al simbolo de la funcion
 void (*pfn)() = dlsym(l,"Saludo1");
 if (pfn==NULL)
    {
     printf("Error buscando la funcion: %s\n",dlerror());
     return 1;
    }
 // Ejecuta la funcion de la libreria
 pfn();
 // Cierra la libreria
 dlclose(l);
 return 0;
}
```

Listado 3.4: Ejemplo de acceso a una librería con las funciones de la familia `dlopen()`

2.6. Encapsular la funcionalidad

Reducir el número de símbolos que una librería exporta acelera los accesos a la tabla de símbolos, y hace que la librería sea más fácil de usar y mantener.

Cómo regla general, aunque es totalmente posible hacerlo, no se recomienda exportar variables globales, ya que esto permitiría al usuario asignarlas valores inapropiados, en vez de esto, se recomienda crear funciones get/set que accedan a estas variables, y que controlen los valores que se asignan.

Otra regla de buen diseño es que si una función va a ser ejecutada tanto por los clientes de la librería, como por la propia librería, debemos de crear una función wrapper, destinada al cliente, que controle los parámetros que recibe, y que llame a la función real. Internamente la librería puede llamar a la función real para evitar que se comprueben los valores de los argumentos, ya que la librería sabrá pasar sólo valores correctos a la función.

2.6.1. Nombres en dos niveles

Como hemos visto en el apartado 2.4.2, el cargador dinámico no detecta conflictos de nombres cuando el mismo símbolo es exportado por dos librerías distintas, sino que carga el símbolo de la primera librería que encuentra. Para evitar este tipo de errores difíciles de detectar por el usuario existen dos soluciones: La primera es poner a todos los símbolos exportados un prefijo que identifique a la librería, la segunda es usar nombres de dos niveles.

Los **nombres en dos niveles** fueron introducidos en Mac OS X 10.1 y consisten en añadir el nombre de la librería como parte del nombre del símbolo. Esto evita que dos símbolos con el mismo nombre en dos librerías distintas interfieran entre ellos. En el apartado 2.8.4 se tratará este tema con más detalle.

2.6.2. Control de la visibilidad de un símbolo

Por defecto todos los símbolos de una librería se exportan (son públicos), es decir, son accesibles desde los clientes de la librería. En general, conviene exportar sólo los símbolos de interfaz que va a usar el usuario, no necesariamente todos los símbolos que definimos durante la programación de la librería. Esto se debe a que cuantos más símbolos exportados haya, más símbolos tiene que cargar el cargador dinámico al cargar la librería.

En Mac OS X existen varias formas de controlar la visibilidad de los símbolos. En los siguientes subapartados vamos a comentar con detalle cada una de ellas.

1. Usar el modificador `static`

La primera forma de evitar que se exporte una variable o función global es usar el modificador `static`. Los símbolos marcados con `static` no son exportados. El Listado 3.5 muestra un ejemplo de cómo podemos implementar una librería donde los símbolos `SetNombre` y `GetNombre` son exportados, pero los símbolos `Nombre` y `_SetNombre` no.

```
/* usuario.c */

#include <string.h>

static char Nombre[255] = "";

static int _SetNombre(const char* n)
{
 strcpy(Nombre,n);
 return 0;
}
```

```
// Función wrapper
int SetNombre(const char* n)
{
 if ( n!=NULL && strlen(n)<255 )
    return _SetNombre(n);
 else
    return 1; // Parametro no valido
}

const char* GetNombre()
{
 return Nombre;
}
```

Listado 3.5: Ocultación de símbolos con `static`

Si ahora compilamos esta librería:

```
$ gcc -dynamiclib usuario.c -o libUsuario.dylib
```

Podemos usar el comando `nm` para ver los símbolos de la librería:

```
$ nm libUsuario.dylib
libUsuario.dylib(dylib1.o):
00000e18 t ___initialize_Cplusplus
00000df8 t __dyld_func_lookup
         u __mh_dylib_header
00000df0 t cfm_stub_binding_helper
00001000 d dyld__mh_dylib_header
00001108 s dyld_func_lookup_pointer
00001104 s dyld_lazy_symbol_binding_entry_point
00000dc0 t dyld_stub_binding_helper

libUsuario.dylib(ccLGTGxY.o):
00000f4c T _GetNombre
00001004 d _Nombre
00000edc T _SetNombre
00000e90 t __SetNombre
         U _strcpy
         U _strlen
         u dyld_stub_binding_helper
```

El comando muestra tanto los símbolos exportados como los no exportados. La Tabla 3.2 muestra los tipos de símbolos que puede identificar `nm` en un fichero de código objeto. Los símbolos que aparecen en mayúsculas son símbolos exportados, y los que aparecen en minúsculas son símbolos privados (o no exportados).

En caso de que sólo nos interesen los símbolos públicos podemos usar la opción `-g`:

```
$ nm -g libUsuario.dylib
libUsuario.dylib(dylib1.o):

libUsuario.dylib(ccsNU7NW.o):
00000f4c T _GetNombre
00000edc T _SetNombre
         U _strcpy
         U _strlen
```

Obsérvese que símbolos como `_strcpy` o `_strlen` aparecen como públicos, esto se debe a que estos símbolos han sido cogidos de una librería de enlace estático donde eran símbolos públicos.

Tipo	Descripción
U	Símbolo externo indefinido (Undefined), es decir, un símbolo que ha sido declarado pero no definido por los módulos objeto.
T	Sección de código (Text).
D	Sección de datos (Data).
B	Sección BSS.
C	Símbolo común. Véase apartado 2.11.2
I	Símbolo indirecto.
S	Símbolo en otra sección distinta a las anteriores.
-	Símbolo de depuración. Para que se muestren debemos usar `-a`

Tabla 3.2: Tipos de símbolos de acuerdo al comando `nm`

El inconveniente que a veces presenta el uso del modificador `static` es que ocultamos las funciones marcadas con `static`, como por ejemplo `_SetNombre`, a otros módulos de nuestra librería. Esto hace que el uso del modificador `static` no sea adecuado cuando queremos ocultar un símbolo a los clientes de la librería, pero permitir que sea accesible desde todos los módulos de nuestra librería.

2. Uso de listas de exportación

Un segunda opción es indicar en un fichero los únicos símbolos que queremos exportar (incluido el prefijo "_" ya que son símbolos destinados a ser usados por el enlazador, no al compilador).

Para ello empezamos creando un fichero con los símbolos a exportar como el siguiente:

```
$ cat usuario.export
_SetNombre
_GetNombre
```

Y compilamos la librería con la opción `-exported_symbols_list`, la cual sirve para indicar este fichero:

```
$ gcc usuario.c -dynamiclib -exported_symbols_list
usuario.export -o libUsuario.dylib
```

3. Indicar la visibilidad con atributos

Esta es posiblemente la forma más recomendable para indicar la visibilidad de los símbolos de la librería. Consiste en usar el atributo `visibility` (véase la Tabla 2.4 y Tabla 2.5 del Tema 2). Este atributo puede tomar uno de los valores que aparecen en la Tabla 3.3.

Visibilidad	Descripción
`default`	Visibilidad por defecto, que es pública.
`hidden`	El símbolo no debe de ser exportado en una librería de enlace dinámico.
`internal`	Igual que la anterior, pero con el añadido de que el símbolo tampoco va a ser llamado desde otro modulo de la librería. Esto permite al compilador no generar el registro PIC para el símbolo.

Tabla 3.3: Niveles de visibilidad de un símbolo para el atributo `visibility`

Normalmente los únicos valores que vamos a usar son `default`, para exportar el símbolo, y `hidden` para ocultar el símbolo. En concreto, `default` se lo pondremos a los símbolos a exportar, y `hidden` no lo vamos a poner en los símbolos a ocultar, sino que en su lugar usaremos la opción `-fvisibility=hidden` que oculta todos los símbolos que no tengan puesto el atributo de visibilidad. El Listado 3.6 muestra como implementar la librería `libUsuario.dylib` usando esta técnica. Hemos usado el identificador del preprocesador `EXPORT` para marcar como exportables sólo a los símbolos que nos interesa. Además, ahora los símbolos que no queremos exportar no los hemos marcado como `static`.

Para compilarlo podemos usar el comando:

```
$ gcc -dynamiclib -fvisibility=hidden usuario.c -o
libUsuario.dylib
```

```
/* usuario.c */

#include <string.h>

#define EXPORT __attribute__((visibility("default")))

char Nombre[255] = "";

int _SetNombre(const char* n)
{
 strcpy(Nombre,n);
 return 0;
}
```

```
EXPORT int SetNombre(const char* n)
{
 if ( n!=NULL && strlen(n)<255 )
    return _SetNombre(n);
 else
    return 1; // Parametro no valido
}

EXPORT const char* GetNombre()
{
 return Nombre;
}
```

Listado 3.6: Indicar la exportabilidad de los símbolos con atributos

2.7. Inicialización de una librería

Cuando se carga una librería de enlace dinámico, ésta puede necesitar preparar recursos o realizar tareas de inicialización antes de hacer cualquier otra cosa. Estas tareas son realizadas por las llamadas **funciones de inicialicación de librería** y **funciones de finalización de librería**.

Tradicionalmente estas tareas han sido realizadas por las funciones `_init()` y `_fini()`. Mac OS X tiene otra forma de implementar estas funciones que es la que vamos a comentar ahora.

Las aplicaciones también pueden definir sus propias funciones de inicialización y finalización, aunque en este apartado nos vamos a centrar en ver como se crean estas funciones para las librerías de enlace dinámico.

Para indicar que una función es de inicialización/finalización se usan los atributos `constructor` y `destructor` respectivamente. Es recomendable que estas funciones no estén exportadas, para lo cual es buena idea el marcarlas con el atributo `visibility="hidden"`[1].

Si una librería tiene varias funciones de inicialización, estas se ejecutan en el orden en que las encuentra el compilador. Por contra las funciones de finalización se ejecutan en el orden inverso a el que las encuentra el compilador.

[1] En `gcc` existe también la opción `-init` *`nombrefn`* que recibe el nombre de una función sin parámetros y sin retorno, y la ejecuta como función de inicialización de librería (es decir, la marca como `constructor`). Pero en este caso sólo podemos dar una función de inicialización de libería. Además, no existe la correspondiente opción de `gcc` para indicar una función de finalización de librería.

Como ejemplo de creación y uso de estas funciones vamos a crear una librería con varias funciones de inicizalización/finalización como muestra el Listado 3.7.

```
/* inicializable.c */

#include <stdio.h>

#define INIT __attribute__((constructor,visibility("hidden")))
#define FINI __attribute__((destructor,visibility("hidden")))


INIT void Inicializador1()
{
 printf("[%s] %s\n",__FILE__,__func__);
}

INIT void Inicializador2()
{
 printf("[%s] %s\n",__FILE__,__func__);
}

FINI void Finalizador1()
{
 printf("[%s] %s\n",__FILE__,__func__);
}

FINI void Finalizador2()
{
 printf("[%s] %s\n",__FILE__,__func__);
}
```

Listado 3.7: Ejemplo de funciones de inicialización/finalización de librería

Ahora podemos generar la librería con el comando:

```
$ gcc inicializable.c -dynamiclib -o libInicializable.dylib
```

```
/* trial.c */

#include <stdio.h>

int main()
{
 printf("[%s] %s\n",__FILE__,__func__);
 return 0;
}
```

Listado 3.8: Programa que usa la librería con inicializadores/finalizadores

Y enlazar el programa del Listado 3.8 con el comando:

```
$ gcc trial.c libInicializable.dylib -o trial
```

Y al ejecutarlo obtenemos:

```
$ ./trial
[inicializable.c] Inicializador1
[inicializable.c] Inicializador2
[trial.c] main
[inicializable.c] Finalizador2
[inicializable.c] Finalizador1
```

Es importante tener en cuenta que las funciones de inicialización/finalización de una librería siempre pueden llamar a funciones de otra librería. Esto se debe a que el cargador dinámico siempre ejecuta las funciones de inicialización/finalización de las librerías de que se depende antes de ejecutar las funciones de inicialización/finalización de nuestra librería.

Por último, comentar que a partir de Mac OS X 10.4 las funciones de inicialización pueden acceder a los argumentos de la línea de comandos y a las variables de entorno igual que lo hace la función `main()`: recibiéndolos como parámetros. El Listado 3.9 muestra cómo se implementaría una función de este tipo.

```
INIT void Inicializador1(int argc, char* argv[], char* envp[])
{
 printf("[%s] %s\n",__FILE__,__func__);
}
```

Listado 3.9: Función de inicialización que lee los argumentos y variables de entorno

2.8. Encontrar símbolos importados

Cuando en el programa cliente el cargador dinámico carga un fichero de librería, éste tiene que conectar los símbolos importados por el programa cliente, con los símbolos exportados por la librería. En este apartado detallaremos como ser realiza este enlace.

2.8.1. Enlazar símbolos

El **enlace de símbolos** (symbols binding), también llamado **resolución de símbolos**, es el proceso por el que las referencias que hace una imagen (ejecutable o librería de enlace dinámico) a datos y funciones de otras imágenes son inicializadas para apuntar al símbolo que las corresponde.

Cuando una aplicación se carga, el cargador dinámico mapea las librerías de enlace dinámico que usa la aplicación en la memoria del proceso, pero no resuelve las referencias, sino que estas suelen usar **enlace tardío** (lazy binding), que consiste en que las referencias no se resuelven hasta que se usan por primera vez. Lógicamente este enlace se realiza de forma recursiva,

es decir si una librería usa a su vez otra librería, esta segunda librería también se tiene que mapear en memoria.

Téngase en cuenta que mientras que mapear un fichero en memoria (p.e. usando la función `mmap()`) es poco costoso, ya que el fichero no se carga realmente en memoria hasta que se produce un fallo de página, enlazar todos los símbolos de una librería implica recorrer todas las direcciones de memoria de la aplicación que hacen referencia a símbolos externos asignándolas la dirección de memoria donde está cargado el símbolo. Para evitar el coste de enlazar todos los símbolos de una librería (muchos de los cuales posiblemente luego no sean usados) se suele usar enlace tardío.

De hecho, el cargador dinámico puede enlazar símbolos de varias formas dependiendo de la opción que pasemos, o bien al comando `ld` cuando enlazamos una aplicación con una librería de enlace dinámico, o bien al comando `libtool` cuando le pedimos enlazar una librería de enlace dinámico con otra librería de enlace dinámico:

- Por defecto se usa el **enlace tardio** (también llamado lazy binding o just-in-time binding), donde el cargador dinámico enlaza un símbolo sólo la primera vez que se va a utilizar el símbolo. En este modo, si un símbolo nunca se utiliza, nunca se enlaza.

- Otra forma de enlazar los símbolos es el **enlace adelantado** (también llamado load-time binding), donde los símbolos de una librería son resueltos cuando el cargador dinámico mapea la librería durante el arranque del programa. En este caso debemos pasar a `ld` durante el enlazado de la aplicación (o a `libtool` durante el enlazado del fichero librería) la opción `-bind_at_load`. Esta opción resulta útil, por ejemplo, en las aplicaciones en tiempo real donde la resolución de símbolos puede hacer que la aplicación sufra latencias intolerables.

- El **preenlazado** (prebinding) es una forma de enlace adelantado que realiza el enlazado de símbolos en tiempo de compilación para evitar hacerlo en tiempo de ejecución. En el siguiente apartado se describe con más detalle esta técnica.

- **Referencias weak**. Este tipo de referencias fueron introducidas en Mac OS X 10.2, y son útiles para acceder a características que no están disponibles en versiones antiguas del sistema operativo, pero sí en nuevas versiones. En el apartado 2.8.3 detallaremos como se usan este tipo de referencias.

2.8.2. Preenlazado

En principio el enlace de símbolos se tiene que hacer en tiempo de ejecución porque, hasta que la librería no se carga, no se conoce las zonas de memoria que están libres en el proceso, es decir, las zonas de memoria que están libres en el espacio de memoria de la aplicación. Las librerías del sistema operativo son librerías muy usadas por muchas aplicaciones, con lo que una posible optimización consiste en pedir al cargador dinámico que, si es posible, las mapee siempre en determinada región de memoria del proceso. Lógicamente, si parte de esta región está ya ocupada por otra librería el preenlazado no es posible. En caso de que esta región no esté ocupada, la librería se cargará en esa dirección y las referencias de la aplicación que hagan referencia a los símbolos de la librería no necesitarían ser enlazadas, lo cual supone una considerable mejora del rendimiento. En caso de que el cargador dinámico no pueda cargar la librería en la dirección de memoria preasignada, el cargador dinámico deberá a continuación modificar todas las referencias de la aplicación que hacen referencia a los símbolos de la librería para que el enlace sea correcto.

El preenlazado requiere que las librerías especifiquen su dirección de memoria base preferida, y que estas regiones no solapen. Apple no recomienda crear este tipo de librerías a los desarrolladores externos a Apple, ya que sino se vuelve muy difícil conseguir que no haya solape entre las librerías, pero las librerías del sistema operativo sí suelen estar creadas con esta opción, y las aplicaciones pueden pedir usar las ventaja del preenlazado con la opción `-prebind` de `ld`.

Podemos ver si un ejecutable está preenlazado con alguna librería con el comando `otool` y la opción `-h` (que muestra las cabeceras de Mach-O del fichero) y la opción `-v`. Por ejemplo, para ver el tipo de enlace de símbolos que utiliza iMovie podemos ejecutar el comando:

```
$ otool -h -v /Applications/iMovie.app/Contents/MacOS/iMovie
Mach header
magic    cputype filetype flags
MH_MAGIC PPC     EXECUTE  NOUNDEFS DYLDLINK PREBOUND TWOLEVEL
```

Entre los flags de la cabecera Mach-O encontramos el flag `PREBOUND`, que indica que se está usando preenlazado.

Podemos consultar la dirección preferida de carga de una librería con el comando `otool` y la opción `-l`, que muestra los comandos del cargador dinámico. Por ejemplo, para ver la dirección preferida de carga de la librería `/usr/lib/libz.dylib` usamos el comando:

```
$ otool -l /usr/lib/libz.dylib
libz.dylib:
Load command 0
      cmd LC_SEGMENT
  cmdsize 396
  segname __TEXT
   vmaddr 0x9108e000
   vmsize 0x0000f000
  fileoff 0
 filesize 61440
  maxprot 0x00000007
 initprot 0x00000005
   nsects 5
    flags 0x0
```

Donde `vmaddr` nos muestra la dirección preferida de carga, y `vmsize` nos indica el tamaño del segmento `__TEXT` de la librería.

2.8.3. Símbolos weak

Existen dos tipos de símbolos weak: Las referencias weak y definiciones weak. En este apartado veremos qué son y para qué sirve cada tipo.

Referencias y definiciones weak

Las **referencias weak**[1] son símbolos externos que importa una aplicación donde, a diferencia de los demás tipos de símbolos externos, si el cargador dinámico no encuentra el símbolo lo deja a `NULL` en vez de fallar. Antes de que una aplicación use estos símbolos, debe comprobar que el símbolo no sea nulo de la forma:

```
if (FuncionWeak!=NULL)
   FuncionWeak(); // Ejecuta la función weak
```

Para que una aplicación indique que una función debe ser tratada como una referencia weak, debe poner el atributo `weak_import` en el prototipo de la función de la forma:

```
void FuncionWeak(void) __attribute__((weak_import));
```

Téngase en cuenta que tiene que ser la aplicación, y no la librería de enlace dinámico, la que declare a un símbolo que va a importar como referencia weak[2].

[1] Si está familiarizado con el formato de los ejecutables ELF puede pensar que una referencia weak es un símbolo weak de ELF, no los confunda, no es lo mismo. Los símbolos weak de ELF son símbolos que pueden ser sobreescritos por símbolos no weak. El equivalente a los símbolos weak de ELF en Mac OS X son las definiciones weak.
[2] Razón por la que se dio el nombre `weak_import` al atributo.

Podemos hacer que todos los símbolos de una librería a importar sean tratados como referencias weak usando la opción `-weak_library`[1] *libreria* de `ld`. Esto permite que al ejecutar el programa no falle si no existe esta librería de enlace dinámico, pero el programa debe de comprobar que los símbolos no sean `NULL` antes de usar cualquier símbolo de la librería así importada.

Las **definiciones weak** son símbolos que son ignorados por el enlazador si existe otra definición no weak del símbolo. Para declarar un símbolo como definición weak se usa el atributo `weak` de la siguiente forma:

```
void MiFuncion(void) __attribute__((weak));
```

Esto permite que, por ejemplo, una aplicación redefina, en la imagen de la aplicación, una función, y sólo si la aplicación no implementa esta función se usa la definición weak de una librería.

Este tipo de símbolos los usan los compiladores de C++ para implementar la instanciación de plantillas. El compilador de C++ a las instanciaciones implícitas de plantillas las marca como definiciones weak, y a las instanciaciones explícitas las marca como definiciones normales (no weak). De esta forma el enlazador estático siempre elige una instanciación explícita frente a una implícita.

Ejemplo de referencias weak

El Listado 3.10 muestra un programa que usa la librería del Listado 3.6.

```
/* usausuario.c */

#include <stdio.h>

int SetNombre(const char* n);
const char* GetNombre();

int main()
{
 SetNombre("Fernando");
 printf("El nombre es %s\n",GetNombre());
 return 0;
}
```

Listado 3.10: Programa que usa la librería `libUsuario.dylib`

Podemos compilarlo y ejecutarlo así:

```
$ gcc usausuario.c libUsuario.dylib -o usausuario
```

[1] En caso de tratarse de un framework, en vez de una librería de enlace dinámico, se usa la opción `-weak_framework` *libreria*.

```
$ ./usausuario
El nombre es Fernando
```

Si ahora borrásemos el fichero `libUsuario.dylib` e intentáramos ejecutar de nuevo el programa:

```
$ rm libUsuario.dylib
$ ./usausuario
dyld: Library not loaded: libUsuario.dylib
  Referenced from: ./usausuario
  Reason: image not found
Trace/BPT trap
```

El cargador dinámico falla porque no encuentra la librería de enlace dinámico.

Para poder detectar la ausencia de la librería no necesitamos modificar el fichero del Listado 3.6, lo que necesitamos es modificar el fichero `usausuario.c` para indicarle que enlace como referencias weak los símbolos de la librería de enlace dinámico a importar. El Listado 3.11 muestra como quedaría ahora el fichero `usausuario.c`.

```
/* usausuario.c */

#include <stdio.h>

#define WEAK_IMPORT __attribute__((weak_import))

WEAK_IMPORT int SetNombre(const char* n);
WEAK_IMPORT const char* GetNombre();

int main()
{
 if (SetNombre && GetNombre)
    {
     SetNombre("Fernando");
     printf("El nombre es %s\n",GetNombre());
    }
 else
    printf("Falta la libreria con las funciones weak\n");
 return 0;
}
```

Listado 3.11: Programa que enlace los símbolos de una librería como referencias weak

Ahora compilamos el Listado 3.11 y obtenemos:

```
$ gcc usausuario.c libUsuario.dylib -o usausuario
/usr/bin/ld: warning weak symbol references not set in output
with MACOSX_DEPLOYMENT_TARGET environment variable set to:
10.1
/usr/bin/ld: warning weak referenced symbols:
_GetNombre
_SetNombre
```

```
/usr/bin/ld: warning dynamic shared library: libUsuario.dylib
not made a weak library in output with
MACOSX_DEPLOYMENT_TARGET environment variable set to: 10.1
```

Los warning avisan de que la variable de entorno `MACOSX_DEPLOYMENT_TARGET` no está fijada, y cuando no se fija se asume como plataforma destino del ejecutable Mac OS X 10.1. Por desgracia las referencias weak aparecieron en Mac OS X 10.2 con lo que el compilador no enlaza con los símbolos como referencias weak.

Podemos solucionar el problema fijando la variable de entorno `MACOSX_DEPLOYMENT_TARGET` y volviendo a compilar:

```
$ export MACOSX_DEPLOYMENT_TARGET=10.2
$ gcc usausuario.c libUsuario.dylib -o usausuario
```

Ahora el programa ejecuta correctamente:

```
$ ./usausuario
El nombre es Fernando
```

Y si borramos la librería, ahora `dyld` detecta que el símbolo weak no se puede cargar, lo pone a `NULL`, y la aplicación no falla.

```
$ rm libUsuario.dylib
$ ./usausuario
Falta la libreria con las funciones weak
```

Puede usar el comando `nm` con la opción `-m` para ver que símbolos están siendo importados como weak:

```
$ nm -m usausuario|grep weak
  (undefined) weak external _GetNombre (from libUsuario)
  (undefined) weak external _SetNombre (from libUsuario)
```

En la cabecera del fichero de la aplicación se definen una serie de **comandos de carga** (load commands), que son comandos que se ejecutan al cargar la aplicación, y que por ejemplo ejecutan `dyld` para cargar librería. Podemos ver estos comandos de carga ejecutando `otool` con la opción `-l`.

Obsérvese que basta con que un símbolo (`_GetNombre` o `_SetNombre`) de la librería de enlace dinámico a usar no esté declarado como referencia weak para que `dyld` falle al ir a cargar la librería. Por ello existe la regla de que si una aplicación importa todos los símbolos de una librería de enlace dinámico como referencias weak, el comando de carga de la librería es weak (de tipo `LC_LOAD_WEAK_DYLIB`). Basta con que un sólo símbolo de la librería de enlace dinámico no sea importado como weak para que el comando de carga no sea weak. Por ejemplo si ejecutamos:

```
$ otool -l usausuario
......
Load command 5
          cmd LC_LOAD_WEAK_DYLIB
      cmdsize 44
         name libUsuario.dylib (offset 24)
   time stamp 1136411437 Wed Jan  4 22:50:37 2006
      current version 0.0.0
compatibility version 0.0.0
......
```

Vemos que nos informa que el comando de carga de `libUsuario.dylib` es de tipo `LC_LOAD_WEAK_DYLIB`.

Si quitamos el prefijo `WEAK_IMPORT` a alguna de las funciones del Listado 3.11 el comando `otool` nos devolverá ahora que no está cargándose la librería como weak.

```
$ otool -l usausuario
......
Load command 5
          cmd LC_LOAD_DYLIB
      cmdsize 44
         name libUsuario.dylib (offset 24)
   time stamp 1136411629 Wed Jan  4 22:53:49 2006
      current version 0.0.0
compatibility version 0.0.0
......
```

Macros para control de compatibilidad

El fichero de cabecera `/usr/include/AvailabilityMacros.h` contiene una serie de macros que nos ayudan a controlar la compatibilidad de una aplicación a la hora de ejecutarla en distintas versiones de Mac OS X. Es decir, cuando una aplicación, compilada en una versión de sistema operativo más moderna, se ejecuta en una versión antigua de sistema operativo, podrían no estar disponibles librerías de enlace dinámico que sólo existen en la versión moderna. Entonces vamos a describir que macros nos permiten saber si nuestro programa se está ejecutando en una versión antigua del SO.

Podemos fijar el macro `MAC_OS_X_VERSION_MIN_REQUIRED` a una versión de sistema operativo, y de esta forma indicar la versión mínima de sistema operativo que nuestra aplicación requiere para ejecutar. En caso de que no esté definido este macro, pero si `MACOSX_DEPLOYMENT_TARGET`, este sería el valor que cogería, sino por defecto vale `1010`[1]. Todas las APIs por debajo o

[1] Obsérvese que `MACOSX_DEPLOYMENT_TARGET` recibe valores de la forma `10.1` mientras que `MAC_OS_X_VERSION_MIN_REQUIRED` recibe valores de la forma `1010`, que es lo que vale el identificador `MAC_OS_X_VERSION_10_1` que encontramos en el fichero de cabecera. El sistema de macros se encarga de realizar las transformaciones oportunas.

iguales a está versión serán enlazadas como referencias normales (no weak). Las APIs definidas después de la versión indicada en `MAC_OS_X_VERSION_MIN_REQUIRED`, serán enlazadas como referencias weak.

A este macro le podemos asignar valores, por ejemplo, con la opción `-D` de la forma:

```
-DMAC_OS_X_VERSION_MIN_REQUIRED=MAC_OS_X_VERSION_10_X
```

Donde *X* es la versión de sistema operativo de que se trate.

Por otro lado, el macro `MAC_OS_X_VERSION_MAX_ALLOWED` permite especificar el número máximo de versión de las APIs del sistema operativo que nuestra aplicación puede usar. Las APIs que sean superiores a esta versión no serán visibles durante la compilación de nuestra aplicación. En caso de que este macro no se defina estará fijado a la mayor versión actual de nuestro sistema operativo, es decir, la más alta que soporte nuestro compilador. En el momento de escribir este tutorial esta sería la 10.4.

Un uso común de estas macros consiste en definir el valor de `MAC_OS_X_VERSION_MIN_ALLOWED` al mismo valor que `MAC_OS_X_VERSION_MAX_ALLOWED`, y los errores que se produzcan no indican APIs que no están definidas para esa versión. Por ejemplo, primero los fijamos a `MAC_OS_X_VERSION_10_1` y si se producen errores sabemos que nuestro software requiere una versión más moderna del sistema operativo, luego podemos probar a fijarlos a por ejemplo `MAC_OS_X_VERSION_10_2`.

2.8.4. Nombres en dos niveles

Para referirnos a un símbolo de una librería de enlace dinámico normalmente damos un nombre de símbolo. Este nombre se suele dar siguiendo el convenio C, con lo que la función `SetNombre` tendrá el símbolo `_SetNombre`.

Las aplicaciones creadas con las herramientas de desarrollo de Mac OS X 10.0 cargaban todos los símbolos de todas las librerías de enlace dinámico usando una sola lista global de símbolos. Cualquier símbolo al que la aplicación se refería podía ser encontrado en cualquier librería de enlace dinámico, con la única restricción de que la librería formase parte de la lista de librerías de las que dependía la aplicación (o una de las librerías de las que dependa otra librería).

Mac OS X 10.1 introdujo los nombres de símbolos en dos niveles. El primer nivel da el nombre de la librería que contiene el símbolo, y el segundo es el nombre del símbolo. Es decir, cuando se usan nombres en dos niveles, el enlazador estático, al guardar referencias a los símbolos importados, lo que

hace es guardar referencias tanto al nombre de la librería que contiene el símbolo, como al nombre del símbolo.

Los nombres en dos niveles presentan dos ventajas respecto a los nombres en un nivel:

- La primera ventaja es que mejoran el rendimiento ya que se acelera el proceso de búsqueda de símbolos en librerías. Esto se debe a que con los nombres en dos niveles el cargador dinámico sabe exactamente en que librería buscar el símbolo. Por contra, usando nombres en un nivel el cargador dinámico debe de buscar por todas las librerías, a ver cuál de ellas contiene el símbolo.

- La segunda ventaja es que mejoran la compatibilidad hacia adelante. En el esquema de nombres en un nivel, dos o más librerías (que use una misma aplicación) no pueden tener símbolos con el mismo nombre ya que sino el enlazador estático producirá un error de enlazado. Sin embargo, si más adelante una librería incorpora un símbolo que ya existía en otra librería con la que enlazaba la aplicación, el cargador dinámico podría enlazar equivocadamente con otra librería.

Por defecto, a partir de Mac OS X 10.1 se usan nombres en dos niveles. Si por alguna razón desea usar nombres en un nivel debe pasar la opción `-flat_namespace` al comando `ld`.

2.9. Librerías C++

El hecho de que C++ use name mangling para representar los símbolos tiene una serie de implicaciones que debemos tener en cuenta a la hora de desarrollar una librería C++, y que vamos a estudiar en este apartado.

Aunque tanto las funciones como las clases C++ puede exportarse, Apple recomienda seguir las siguientes dos reglas a la hora de crear una librería de enlace dinámico C++:

1. La clase debe exportar tanto los constructores públicos como los métodos públicos.

2. Las funciones, siempre que sea posible, deben exportarse al estilo C (usando `export "C"`).

3. Las clases deben declarar todos los métodos públicos virtuales, y crear funciones factory para instanciar los objetos.

La segunda regla se propuso para poder acceder a las funciones de librerías cargadas dinámicamente con las funciones de la familia `dlopen()`. Recuérdese que en estas funciones nosotros pasábamos a `dlsym()` la cadena `"NuevaPersona"` y `dlsym()` buscaba en la librería la cadena `"_NuevaPersona"`. Al tener las funciones C++ una técnica de name mangling que no se garantiza que se conserve en el futuro, las funciones C no deberían de confiar en componer el nombre de la función con name mangling (p.e. `__Z4Sumaii`). El exportar, siempre que no exista sobrecarga, los nombres de las funciones al estilo C evita posibles problemas de compatibilidad hacia adelante.

La razón de la tercera regla es que si los métodos públicos (los que va a llamar la aplicación) son virtuales pueden ejecutarse sin necesidad de conocer el nombre del símbolo en la librería, aunque eso sí, sólo a través de punteros, que es cuando el enlace dinámico actúa.

La primera regla se pone para poder acceder a la librería desde C++, ya que cuando un objeto de la librería es instanciado en la pila o como objeto global, la llamada a los métodos sería estática (y deberán de estar exportados).

El Listado 3.12 muestra un ejemplo de como debería de declararse el prototipo de una clase, y el Listado 3.13 muestra como debería de implementarse los métodos de la clase.

```
/* CPersona.h */

#ifndef _CPERSONA_H_
#define _CPERSONA_H_

class CPersona
{
 private:
  char nombre[100];
 public:
  CPersona(const char* nombre);
  virtual void setNombre(const char* nombre);
  virtual const char* getNombre() const;
};

extern "C" CPersona* NuevaPersona(const char* nombre);
extern "C" void BorraPersona(CPersona* p);

#endif
```

Listado 3.12: Prototipo de una librería de enlace dinámico C++

```
/* CPersona.cpp */

#include "CPersona.h"

#define EXPORT __attribute__((visibility("default")))

////////////////////////////////////////////////////////////
EXPORT CPersona* NuevaPersona(const char* nombre)
{
 return new CPersona(nombre);
}
////////////////////////////////////////////////////////////
EXPORT void BorraPersona(CPersona* p)
{
 delete p;
}
////////////////////////////////////////////////////////////
EXPORT CPersona::CPersona(const char* nombre)
{
 strcpy(this->nombre,nombre);
}
////////////////////////////////////////////////////////////
EXPORT void CPersona::setNombre(const char* nombre)
{
 strcpy(this->nombre,nombre);
}
////////////////////////////////////////////////////////////
const char* CPersona::getNombre() const
{
 return this->nombre;
}
////////////////////////////////////////////////////////////
```

Listado 3.13: Implementación de una librería de enlace dinámico C++

Podemos generar la librería con el comando:

```
$ g++ -fvisibility=hidden CPersona.cpp -dynamiclib -o
libPersona.dylib
```

Lógicamente, cuando la librería de enlace dinámico es una librería dependiente de la aplicación (cargada por `dyld`) se pueden usar los operadores `new` y `delete` para crear objetos, o bien crear instancias globales o locales de los objetos. Por contra, cuando estemos enlazando con la librería en tiempo de ejecución desde las funciones de la familia `dlopen()` necesitamos usar las funciones factory que hemos declarado como `extern "C"` para que resulte más sencillo encontrar su símbolo.

El Listado 3.14 muestra como podemos usar los objetos de la librería de enlace dinámico accediendo desde las funciones de la familia `dlopen()`.

```
/* usapersona */

#include "CPersona.h"

using namespace std;

int main()
{
 // Abre la libreria
 void* lib = dlopen("./libPersona.dylib",0);
 if (!lib)
    {
     cerr << "Fallo abriendo la libreria" << endl;
     return 1;
    }
 // Crea un objeto
 CPersona*(*pfn1)(const char*) =
                  (typeof(pfn1)) dlsym(lib,"NuevaPersona");
 if (pfn1==NULL)
    {
     cerr << "Fallo accediendo a NuevaPersona()" << endl;
     return 1;
    }
 CPersona* p = pfn1("Fernandito");
 // Usa el objeto
 p->setNombre("Don Fernando");
 cout << p->getNombre() << endl;
 // Libera el objeto
 void(*pfn2)(CPersona* p) =
                  (typeof(pfn2)) dlsym(lib,"BorraPersona");
 if (!pfn2)
    {
     cerr << "Fallo accediendo a BorraPersona()" << endl;
     return 1;
    }
 pfn2(p);
 // Cierra la libreria
 dlclose(lib);
 if (!lib)
    {
     cerr << "Fallo cerrando la libreria" << endl;
     return 1;
    }
 return 0;
}
```

Listado 3.14: Uso de objetos de librería de enlace dinámico con `dlopen()`

Ahora podríamos compilar y ejecutar este programa con:

```
$ g++ usapersona.cpp -o usapersona
$ ./usapersona
```

2.10. Código dinámico

Para poder disponer de librerías de enlace dinámico, las aplicaciones, las herramientas de desarrollo, las librerías de enlace dinámico, y el cargador dinámico deben dar soporte para dos tecnologías: Position Independent Code (PIC) y direccionamiento indirecto. En este apartado se describen estas dos tecnologías.

2.10.1. Position Independent Code

Como se introdujo en el apartado 2.3, PIC es el nombre que recibe la técnica que permite al cargador dinámico cargar el código de una librería de enlace dinámico en una región no fija de memoria. Si el código generado no es PIC, el cargador dinámico necesitaría cargar el código de la librería de enlace dinámico siempre en la misma dirección de memoria. Esto sería así con el fin de que las referencias relativas a variables de la librería (situadas normalmente en el segmento `__DATA`) que haga el código de la librería (situado normalmente en el segmento `__TEXT`) sean direccionamientos correctos. Lógicamente, esta limitación haría el mantenimiento de las librerías de enlace dinámico instaladas en el sistema operativo extremadamente difícil.

Mach-O, el formato de los binarios de Mac OS X, por defecto es PIC, y se basa en la observación de que el segmento `__DATA` se encuentra siempre a un offset constante del segmento `__TEXT`. Es decir, el cargador dinámico, cuando carga una librería siempre carga el segmento `__TEXT` a la misma distancia del segmento `__DATA`. De esta forma una función puede sumar a su dirección un offset fijo para determinar la dirección del dato al que quiere acceder. Esta forma de direccionamiento, llamada direccionamiento relativo, actualmente está implementado en el ensamblador tanto de PowerPC como de Intel con lo que un direccionamiento con código PIC no es más lento que su correspondiente direccionamiento con código no PIC. De hecho, todos los segmentos del formato Mach-O, y no sólo los segmentos `__TEXT` y `__DATA`, se cargan a distancias fijas de los demás segmentos[1].

El código PIC es requerido sólo por las librerías de enlace dinámico, no por las aplicaciones, que normalmente siempre se cargan en la misma dirección de memoria virtual. Aunque el código PIC no es más lento, en cuanto a lo que a direccionamientos se refiere, si que introduce restricciones que aumentan el tamaño del código generado y reducen su rendimiento. Aunque las

[1] Si está familiarizado con el formato ELF (Executable and Linking Format), muy utilizado por sistemas como Linux, obsérve que Mach-O no tiene una GOT (Global Offset Table) ya que en el formato Mach-O no se consulta esta tabla para acceder a otros segmentos, sino que se utilizan direcciones relativas, lo cual es más rápido debido a que el direccionamiento se hace directamente usando instrucciones máquina de direccionamiento relativo. El formato ELF se diseñó cuando no todas las máquinas soportaban (ni siguen soportando) direccionamiento relativo.

aplicaciones no requieren código PIC, si que necesitan la característica de direccionamiento indirecto que veremos en el siguiente apartado. Por esta razón las GCC 3.1 introdujeron la opción `-mdynamic-no-pic`, la cual deshabilita el código PIC, pero mantiene los direccionamientos indirectos, necesarios para poder acceder a símbolos de una librería de enlace dinámico[1].

2.10.2. Direccionamiento indirecto

El **direccionamiento indirecto** es el nombre que recibe una técnica de generación de código (separada del PIC) que permite acceder a los símbolos de una librería de enlace dinámico desde otra imagen (que puede ser una aplicación o otra librería). Con esta técnica una librería puede ser actualizada sin necesidad de actualizar la imagen cliente que la utiliza.

Cuando se genera una llamada a una función definida en una librería de enlace dinámico el compilador, por cada función de librería que usa la imagen cliente, crea un stub de símbolo y un puntero a símbolo tardío. El **stub de símbolo** es un trozo de código ensamblador que se encarga de ejecutar la función de librería. El **puntero a símbolo tardio** no es más que un trozo de memoria donde se almacena un puntero a la función a ejecutar. La primera vez que se ejecuta la función el puntero a símbolo tardío contiene la dirección de memoria de una función llamada `dyld_stub_binding_helper()` la cual se encarga de enlazar el símbolo. Las demás veces el puntero a símbolo tardío tendrá la dirección real del símbolo a ejecutar.

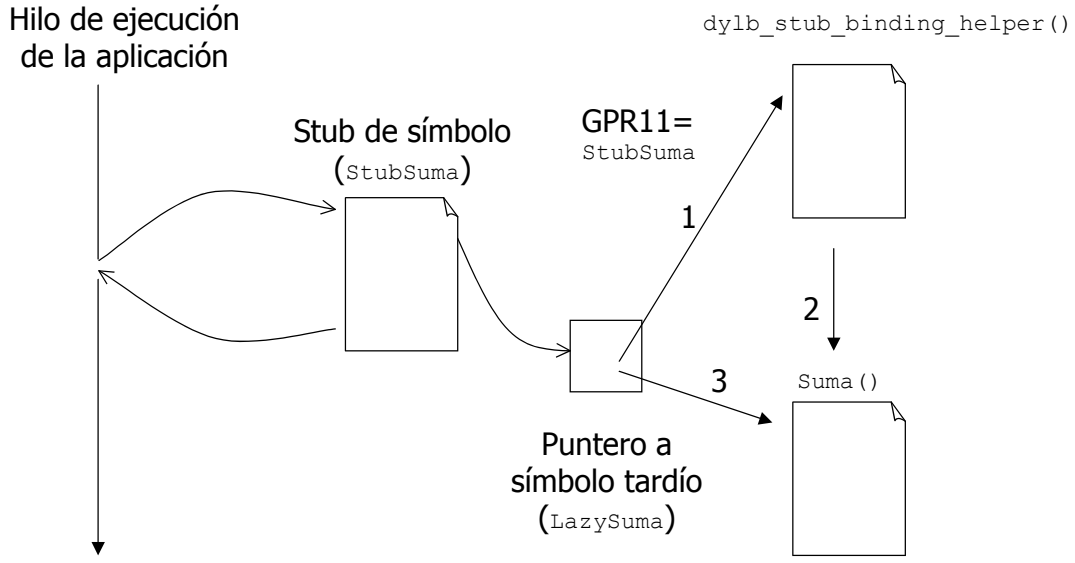


Figura 3.2: Direccionamiento indirecto

La Figura 3.2 muestra el proceso de direccionamiento indirecto. Cuando el hilo de la aplicación llama al stub de símbolo (`StubSuma` en este ejemplo),

[1] Actualmente Xcode usa la opción `-mdynamic-no-pic` por defecto al generar aplicaciones

éste coloca en el registro máquina GPR11 su propia dirección y ejecuta la función cuya dirección esté almacenada en el puntero a símbolo tardío (`LazySuma` en este ejemplo). La primera vez este puntero tendrá la dirección de memoria de la función `dyld_stub_binding_helper()` con lo que ésta se ejecutará recibiendo en el registro GPR11 la dirección del stub de símbolo, la cual le sirve para determinar que símbolo de librería enlazar. Después la función `dyld_stub_binding_helper()` ejecuta la función `Suma()` y almacena la dirección de memoria de `Suma()` en el puntero a símbolo tardío. De esta forma las demás veces que se indireccione este puntero se ejecutará directamente la función `Suma()`.

2.11. Herramientas para ficheros binarios

2.11.1. Herramientas para analizar ficheros Mach-O

Ya hemos visto que para analizar un fichero Mach-O disponemos de dos comandos `nm` y `otool`.

El comando `nm` nos muestra la tabla de símbolos de un fichero de código objeto, sea este un `.o`, una librería o un ejecutable. Normalmente la salida de este comando tendrá la forma:

```
$ nm usuario.o
000000bc T _GetNombre
00000140 D _Nombre
0000004c T _SetNombre
00000000 T __SetNombre
         U _strcpy
         U _strlen
         U dyld_stub_binding_helper
```

Opción	Descripción
`-a`	Muestra todos los símbolos, incluidos los de depuración.
`-g`	Muestra sólo los símbolos globales (externos).
`-u`	Muestra sólo los símbolos indefinidos (undefined), es decir, los símbolos que han sido declarados pero no definidos por el módulo objeto.
`-m`	Muestra el segmento y sección de cada símbolo, así como información sobre si el símbolo es un símbolo externo o no externo.
`-s`	Muestra sólo el segmento y sección indicados. Para ello se usa esta opción de la forma `-s` *`segmento seccion`*.

Tabla 3.4: Principales opciones del comando `nm`

Donde cada símbolo está precedido por un valor (o blanco si es un símbolo indefinido), después irá una letra cuyo significado se detalla en la Tabla 3.2. Si el símbolo es privado aparecerá en minúsculas y si es un símbolo exportado

aparecerá en mayúsculas. La Tabla 3.4 resume las principales opciones del comando `nm`.

El otro comando que tiene Mac OS X para analizar un fichero Mach-O es el comando `otool`[1]. Las distintas opciones de este comando se resumen en la Tabla 3.5. Todas estas opciones tienen dos modos. El modo numérico resumido que se da cuando sólo indicamos la opción, y el modo simbólico extendido que se da cuando también indicamos la opción `-v` o `-V`. Cada opción indica la información que queremos obtener. El comando `otool`, a diferencia del comando `nm`, no puede ejecutarse sin opciones.

Opción	Descripción
`-a`	Usado para mostrar la cabecera de un archivo. Supone que el fichero es un archivo (fichero `.a`).
`-S`	Muestra el contenido del fichero `__.SYMDEF`, suponiendo que el fichero pasado sea un archivo (fichero `.a`).
`-f`	Muestra las cabeceras fat, suponiendo que el archivo sea un binario universal.
`-h`	Muestra la cabecera Mach del fichero.
`-L`	Muestra los nombres y números de versión de las librerías de enlace dinámico que usa el fichero objeto. Véase el apartado 2.3.
`-D`	Muestra simplemente el nombre de instalación de una librería de enlace dinámico. Véase apartado 2.4.1.
`-s`	Muestra sólo el contenido de un segmento y sección. Para ello usamos la opción `-s` *`segmento seccion`*.
`-t`	Muestra el contenido de la sección `(__TEXT,__text)`. Junto con la opción `-v` desensambla su contenido, y con la opción `-V` también desensambla simbólicamente los operandos.
`-d`	Muestra el contenido de la sección `(__DATA,__data)`.
`-o`	Muestra el contenido del segmento `__OBJC`, usado por el runtime de Objective-C.
`-r`	Muestra el contenido de las entradas reasignables. Lógicamente sólo funciona con ficheros de código objeto (`.o`), y librerías estáticas y dinámicas, pero no con ejecutables.
`-I`	Muestra la tabla de símbolos indirectos.
`-T`	Muestra la tabla de contenido de una librería de enlace dinámico.
`-R`	Muestra la tabla de referencias de una librería de enlace dinámico.

Tabla 3.5: Principales opciones del comando `otool`

[1] Este comando, al igual que el formato Mach-O, es propio del sistema Mac OS X, la mayoría de los sistemas UNIX disponen en su lugar del comando `objdump`.

2.11.2. Los símbolos comunes

Los símbolos comunes son más típicos en le lenguaje Fortran, pero también se usan en C para solucionar vicios comunes de los programadores.

En C cuando se crea una variable global que va a ser usada por varios módulos, esta debe de declararse (preferiblemente en un fichero de cabecera) con el modificador `extern` en todos los módulos excepto en el que se define.

En C también se permite que varios ficheros `.c` definan una variable global siempre que no la asignen un valor, tal como muestra el Listado 3.15 y Listado 3.16.

```
/* modulo1.c */

int global;

int main()
{
 return global;
}
```

Listado 3.15: Módulo con variable global no inicializada

```
/* modulo2.c */

int global;
```

Listado 3.16: Otro módulo con variable global no inicializada

En caso de que las variables globales no estén inicializadas varios módulos pueden definir el mismo símbolo (que es lo que llamamos una variable común) y durante el enlazado la variables comunes se fundirán en una sola:

```
$ gcc -c modulo1.c modulo2.c
$ nm modulo1.o modulo2.o
modulo1.o:
00000004 C _global
00000000 T _main

modulo2.o:
00000004 C _global
```

Vemos que las variables comunes aparecen marcadas con el tipo `C` (véase Tabla 3.2). Durante el enlazado las variables globales se fundirán en una sola.

```
$ gcc modulo1.o modulo2.o -o modulo1
```

2.11.3. Eliminar símbolos innecesarios

El comando `strip` permite eliminar símbolos innecesarios de un fichero de código objeto. Esto puede resultar útil para reducir el tamaño de un programa o librería una vez que éste ha sido depurado.

Este comando no elimina todos los símbolos: los símbolos usados para el enlace dinámico nunca se eliminan, o sino el programa o librería dejarían de funcionar. Pero si que es útil eliminar los símbolos privados y los de depuración.

Por ejemplo si compilamos el Listado 1.1 y ejecutamos sobre él `nm` obtendríamos gran cantidad de símbolos:

```
$ gcc hola.c -o hola
$ nm hola
00003000 D _NXArgc
00003004 D _NXArgv
00002a30 T ___darwin_gcc3_preregister_frame_info
0000300c D ___progname
00002bfc t ___stub_getrealaddr
00002488 t __call_mod_init_funcs
00002584 t __call_objcInit
         U __cthread_init_routine
00002798 t __dyld_func_lookup
00002740 t __dyld_init_check
00001000 A __mh_execute_header
00002210 t __start
         U _abort
```

Muchos de los cuales son innecesarios y se pueden eliminar con `strip`:

```
$ strip hola
$ nm hola
00003000 D _NXArgc
00003004 D _NXArgv
0000300c D ___progname
         U __cthread_init_routine
00001000 A __mh_execute_header
         U _abort
```

El comando sobrescribe el fichero dado como argumento. Podemos enviar el resultado a otro fichero con la opción `-o`:

```
$ strip hola -o hola-release
```

También podemos ejecutar `strip` sobre una librería, e incluso sobre un fichero de código objeto metido dentro de una librería de enlace estático de la forma:

```
$ strip saludo1.o libsaludos.a
```

3. Frameworks

Un **framework** es una estructura jerárquica de directorios donde se almacenan recursos compartidos, tales como librerías de enlace dinámico, ficheros `.nib`, ficheros de cabecera, cadenas localizadas, y documentación de referencia sobre el funcionamiento del framework, todo ello encapsulado en un sólo paquete. Varias aplicaciones pueden usar todos estos recursos simultáneamente. Para ello el sistema va cargando en memoria todos estos recursos, según los van necesitando las aplicaciones, y al ser recursos compartidos todos ellos ocupan la misma zona de memoria física, aunque no necesariamente la misma dirección de memoria virtual de cada proceso.

Un framework es un tipo de **bundle**[1], que son conjuntos de ficheros que se agrupan de forma que desde el Finder se ven como un icono. Otros ejemplos de bundles son las aplicaciones o los plug-in de aplicación. Los framework, a diferencia de los otros tipos de bundles no tienen el atributo que hace que Finder los muestre como un icono, sino que es posible usar el Finder para inspeccionar su contenido. Esto se hizo así para facilitar al desarrollador moverse dentro del contenido del framework.

La ventanas que introducen los framework respecto a las librerías de enlace dinámico convencionales son:

- El framework encapsula junto con la librería de enlace dinámico sus ficheros de cabecera y su documentación
- El framework puede tener otro tipo de recursos como iconos, ficheros `.nib`, o cadenas localizadas a cada lengua y país.
- Todos los recursos del framework, y no sólo la librería de enlace dinámico, se pueden compartir en memoria entre las aplicaciones.

La capa Darwin de Mac OS X está implementada mediante librerías de enlace estático y enlace dinámico independientes, pero otros interfaces de programación con Mac OS X están implementadas como frameworks, como por ejemplo Carbon, Cocoa, o los Core Services. Estos interfaces están agrupados bajo **umbrella frameworks**, que son frameworks que no contienen funcionalidad, sino que agrupan a otros frameworks.

Además de usar los frameworks del sistema, el programador puede crear sus propios frameworks. En caso de que el framework esté destinado a apoyar a una determinada aplicación, y no a compartirlo con otros desarrolladores, sería un **framework privado**. Por contra un **framework público** sería un framework destinado a ser compartido con otras aplicaciones, y en este caso se suele publicar su interfaz, para que otros desarrolladores lo puedan usar.

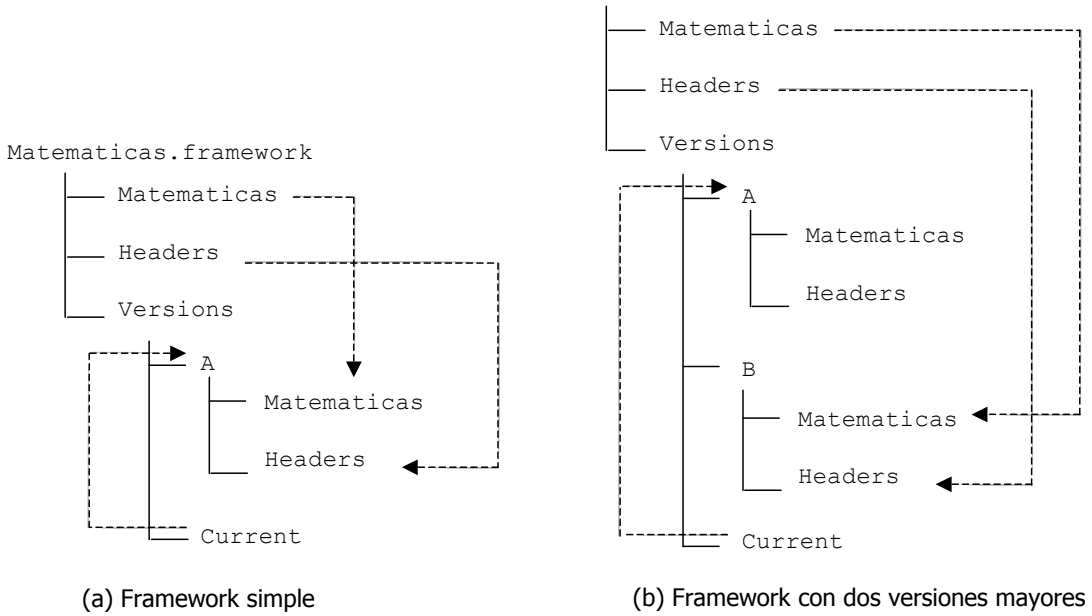
[1] Su contenido puede ser gestionado usando los Core Foundation Bundle Services, o la clase `NSBundle` de Cocoa.

3.1. Anatomía

En Mac OS X, los recursos compartidos se empaquetan usando frameworks estándar y umbrella frameworks. Ambos frameworks tienen la misma estructura, aunque los umbrella frameworks añaden pequeños refinamientos a la estructura del framework estándar para permitir incorporar otros frameworks.

Como ya hemos comentado, un framework es un tipo de bundle y se caracterizan por ser una carpeta con la extensión `.framework`. En este apartado vamos a estudiar la estructura de ficheros y directorios de un framework estándar, y en el apartado 3.5 estudiaremos la estructura de los ficheros de los umbrella frameworks.

Los framework estándar tienen la estructura de ficheros de la Figura 3.3 (a) donde las líneas discontinuas representan enlaces simbólicos. En el primer nivel encontramos dos enlaces a ficheros almacenados en subdirectorios. El primero es `Matematicas` que apunta a la librería de enlace dinámico más reciente de nuestra librería (obsérvese que no tienen la extensión `.dylib`). El segundo es `Headers` que apunta a la carpeta de ficheros de cabecera más reciente de la librería. También en el primer nivel encontramos la carpeta `Versions` donde se almacenan las versiones mayores de nuestra librería. Cuando una herramienta de desarrollo intenta acceder a un recurso del framework siempre accede a un recurso del primer nivel de directorios, pero si estos son un enlace, la llevarán a usar la ruta del recurso enlazado.



(a) Framework simple

(b) Framework con dos versiones mayores

Figura 3.3: Estructura de ficheros de un framework estándar

Dentro de la carpeta `Versions` existirá una carpeta por cada versión mayor, etiquetada cada versión sucesiva con los nombres `A`, `B`, etc[1]. Además estará la carpeta `Current` que apunta a la versión mayor más moderna. Cada versión contendrá todo lo que deba encontrarse en el nivel superior del framework, por ejemplo, si el framework tiene iconos, estos deberán estar duplicados en cada versión mayor, aunque hayan ido cambiando en las sucesivas versiones. Al existir enlaces a los recursos del primer nivel del framework (que son los únicos que existen para la herramienta de desarrollo) la librería puede ir evolucionando en el tiempo sin que la herramienta de desarrollo que la usa sea consciente. Sólo las aplicaciones que usan el framework conocerán la verdadera ruta de los recursos, y de esta forma se mantiene compatibilidad hacia atrás.

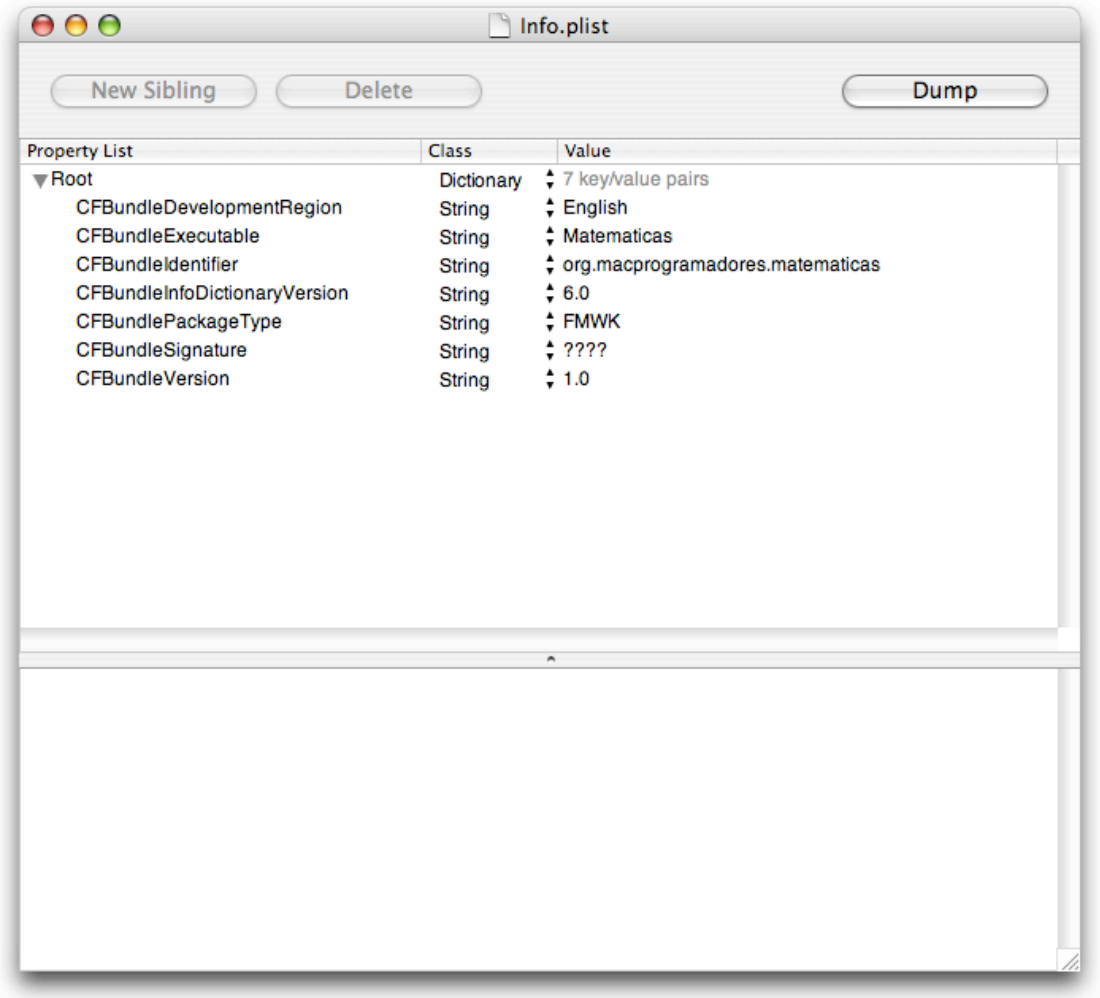


Figura 3.4: Contenido del fichero `Info.plist`

[1] El nombre que se asigna a las versiones mayores no tiene porque ser necesariamente letras mayúsculas sucesivas, pueden ser por ejemplo números, pero esta forma de versionado es la más utilizada en el caso de los frameworks.

El recurso más importante de cada versión mayor es la librería de enlace dinámico, y es el único obligatorio. La carpeta `Headers` es opcional, aunque muy usada cuando se crea un framework público.

Al igual que pasa con la carpeta `Headers`, en caso de que el framework tenga otros recursos, deberá crearse una carpeta con los recursos en la correspondiente subcarpeta de versión mayor dentro de `Versions`, y un enlace simbólico en el primer nivel de la jerarquía de directorios que apunte a la carpeta más reciente con los recursos. Por ejemplo, cuando a Xcode le pedimos crear un framework, por defecto crea la carpeta `Resources` donde mete recursos como ficheros con texto localizado o un fichero llamado `Info.plist` con información de localización del recurso como la que se muestra en la Figura 3.4.

Otra carpeta que es común encontrar en un framework es la carpeta `Documentation`, la cual contiene ficheros HTML o PDF que describen las funciones de la librería.

Lógicamente el programador puede crear carpetas con otros nombres en los que mete los recursos que considere necesario.

3.2. Versionado de los frameworks

En el apartado 2.2 vimos que las librerías de enlace dinámico tenían versiones mayores y versiones menores. Las versiones mayores permitían la compatibilidad hacia atrás, de forma que si actualizábamos la librería, aplicaciones enlazadas con una versión mayor antigua seguían funcionando. Por contra las versiones menores, conseguían la compatibilidad hacia adelante, de forma que una aplicación enlazada con una librería más moderna podría seguir funcionando cuando la ejecutamos en un entrono donde exista una versión más antigua de la librería. Las versiones menores, además de ser compatibles hacia adelante, obligatoriamente deberán de ser compatibles hacia atrás, o de lo contrarío deberíamos de haber creado una nueva versión mayor.

Recuérdese que en el apartado 2.2.2 vimos que la versión menor tenía dos números: El número de versión actual que se fijaba con la opción `-current_version`, que permitía numerar cada cambio que hacíamos a la librería, y la versión de compatibilidad, que se fijaba con la opción `-compatibility_version`, y que permitía conseguir la compatibilidad hacía adelante, es decir, si entre dos librerías no cambia el número de versión de compatibilidad (sólo el número de versión actual), una aplicación enlazada con la versión más moderna de la librería seguiría funcionando si la ejecutamos en un entrono donde esté instalada la versión antigua de la librería. Luego, el tener la misma versión de compatibilidad entre dos versiones de una misma librería indica

que la versión más antigua tiene exactamente la misma interfaz (y que como mucho, tiene errores o código menos optimizado).

Las versiones menores en los frameworks se fijan con las mismas opciones de `gcc` que en las librerías de enlace dinámico, pero las versiones mayores siguen el esquema de la Figura 3.3 (b). En la carpeta `Versions` puede haber más de un directorio, en cuyo caso nuestro framework tiene varias versiones mayores. Aunque la herramienta de desarrollo siempre enlaza con la última versión mayor, las aplicaciones enlazan con el fichero de librería de enlace dinámico cuya ruta es el directorio apuntado por el enlace simbólico en el momento en el que se compiló, con lo que una actualización de versión mayor no afecta a las aplicaciones enlazadas con versiones mayores anteriores.

3.3. Un framework de ejemplo

En el aparado 3.3.1 vamos a hacer un ejemplo detallado donde se describen los pasos necesarios para crear un framework. En el apartado 3.3.2 veremos como se instala el framework. En el apartado 3.3.3 detallaremos como usar un framework desde una aplicación. Por último en el apartado 3.3.4 veremos como crear múltiples versiones menores dentro de un framework.

3.3.1. Crear un framework

Lo primero que tenemos que hacer para crear un framework es crear la estructura de directorios del framework. En nuestro ejemplo vamos a crear la estructura de directorios de la Figura 3.3 (a):

```
$ mkdir Matematicas.framework
$ cd Matematicas.framework
$ mkdir Versions
$ cd Versions
$ mkdir A
$ cd A
$ mkdir Headers
$ cd ..
$ ln -s A Current
$ cd ..
$ ln -s Versions/A/Headers/ Headers
$ cd
```

Lo siguiente que vamos a hacer es crear una librería donde sólo implementamos la función de ejemplo `Suma()`. El Listado 3.17 y Listado 3.18 muestran como implementar esta librería.

```
/* Matematicas.h */
```

```
#ifndef _SUMA_H_
#define _SUMA_H_

int Suma(int a, int b);

#endif
```

Listado 3.17: Cabecera de la librería del framework

```
/* Matematicas.c */

#include "Matematicas.h"

int Suma(int a, int b)
{
 int c = a+b;
 return c;
}
```

Listado 3.18: Implementación de la librería del framework

Ahora podemos generar la librería de enlace dinámico del framework:

```
$ gcc Matematicas.c -dynamiclib -install_name /Library/Framework/
Matematicas.framework/Versions/A/Matematicas -current_version 1.0.0
-compatibility_version 1.0.0 -o Matematicas
```

Obsérvese que hemos dado como nombre de instalación la ruta `/Library/Framework/Matematicas.framework/Versions/A/Matematicas`. Esto es necesario hacerlo porque, como se explica en el apartado 3.3.2, los frameworks normalmente se instalan en una subcarpeta de la carpeta `/Library/Frameworks`. Como el nombre de instalación se registra dentro de la aplicación para que el cargador dinámico pueda encontrar la librería, durante la creación de la librería es necesario indicar este nombre de instalación.

También obsérvese que el nombre de instalación da la ruta física de la librería dentro de su carpeta de versión mayor, y no la ruta del enlace simbólico a la librería. Esto se hace para que al cambiar la versión mayor del framework no cambie la librería con lo que estaban enlazadas las aplicaciones.

Al crear la librería también hemos usado las opciones `-current_version` y `-compatibility_version` para indicar que la versión menor, tanto actual como de compatibilidad, es la 1.0.0.

Una vez tenemos creada la librería podemos copiarla a la carpeta del framework y crear el enlace a este fichero en el primer nivel:

```
$ cp Matematicas Matematicas.framework/Versions/A/
$ ln -s Matematicas.framework/Versions/A/Matematicas
Matematicas.framework/
```

Por último copiamos el fichero de cabecera de nuestra librería a la carpeta `Headers`:

```
$ cp Matematicas.h Matematicas.framework/Versions/A/Headers/
```

3.3.2. Instalar un framework

Los frameworks creados por desarrolladores deben de ir en una de estas carpetas:

- La mayoría de los frameworks públicos se colocan en la carpeta `/Library/Frameworks`.
- Si un framework va a ser usado por un sólo usuario, podemos instalarlo en la carpeta `~/Library/Frameworks` del usuario.
- Si un framework va a ser usado por muchos usuarios de una red, podemos ponerlo en la carpeta `/Network/Library`.

Apple recomienda usar siempre la carpeta `/Library/Frameworks` y ha comenzado a desaconsejar el uso de las carpetas `~/Library/Frameworks` y `/Network/Library`. La primera porque si se pasa una aplicación de la cuenta de un usuario a la de otro ésta deja de funcionar, la segunda porque produce retrasos en el acceso a red, especialmente cuando no se puede acceder a ésta.

Lo que nunca debemos hacer es colocar nuestras librerías en la carpeta `/System/Library/Framework`. Esta carpeta se la reserva Apple para sus propias librerías.

En nuestro ejemplo vamos a copiar el framework al directorio `/Library/Frameworks`:

```
$ mkdir /Library/Frameworks/Matematicas.framework
$ cp -R Matematicas.framework /Library/Frameworks/Matematicas.framework
```

3.3.3. Usar un framework

Cuando enlazamos con un framework al enlazador estático le tenemos que dar el nombre del framework con la opción `-framework` seguida del nombre del framework sin extensión. Por ejemplo para indicar que se enlace con nuestro framework llamado `Matematicas.framework` debemos de dar al comando `ld` (o al comando `gcc` si preferimos usar el driver) la opción `-framework Matematicas`.

Para que `ld` encuentre el framework, éste debe encontrarse en la carpeta `/Library/Framework` o `/System/Library/Framework`. Por defecto `ld` no busca en ninguna otra carpeta, incluidas `~/Library/Frameworks` ni `/Network/Library`.

Si queremos que el framework se busque en otras carpetas tenemos dos opciones: Indicar a `ld` donde buscarlo con la opción `-F`, o indicar una lista de directorios a buscar (separados por dos puntos) en la variable de entorno `DYLD_FRAMEWORK_PATH`.

Imaginemos que queremos usar el framework que hemos creado antes, y para ello creamos el programa del Listado 3.19.

```
/* usaframework */

#include <stdio.h>
#include <Matematicas/Matematicas.h>

int main()
{
 printf("4+7 son %i\n",Suma(4,7));
 return 0;
}
```

Listado 3.19: Programa que usa el framework `Matematicas`

Todos los ficheros de cabecera puestos en la carpeta `Headers` de un framework con nombre *`Framework`* pueden ser accedidos de la forma:

```
#include <Framework/Cabecera.h>
```

Donde *`Framework`* es el nombre del framework y *`Cabecera`* es el fichero de cabecera (puesto en la carpeta `Headers`) a incluir.

Por ejemplo, para acceder a nuestra librería usamos:

```
#include <Matematicas/Matematicas.h>
```

Observe que en ningún sitio vamos a usar la opción `-I` para indicar donde está instalado el fichero de cabecera, ya que basta con que el fichero de cabecera esté en la carpeta `Headers` de un framework, para que este tipo de inclusión funcione.

Además al ir a usar los ficheros de cabecera de un framework es tradicional incluir sólo el **fichero de cabecera maestro**, que es el fichero de cabecera cuyo nombre coincide con el del framework, por ejemplo:

```
#include <AddressBook/AddressBook.h>
```

`AddressBook.h` es el fichero maestro e incluirá todos los ficheros de cabecera del framework que necesite.

Luego la forma de poder hacer nuestra aplicación que accede al framework será:

```
$ gcc usaframework.c -framework Matematicas -o usaframework
$ ./usaframework
4+7 son 11
```

El comando:

```
$ gcc usaframework.c -M
usaframework.o: usaframework.c /usr/include/stdio.h
  /usr/include/_types.h \
  /usr/include/sys/_types.h /usr/include/sys/cdefs.h \
  /usr/include/machine/_types.h /usr/include/ppc/_types.h \
  /Library/Frameworks/Matematicas.framework/Headers/
Matematicas.h
```

Nos ayuda a ver que la inclusión de la ruta `Matematicas/Matematicas.h` ha sido sustituida por el preprocesador por `/Library/Frameworks/Matematicas.framework/Headers/Matematicas.h`.

3.3.4. Crear múltiples versiones de un framework

Si ejecutamos el comando:

```
$ otool -L
/Library/Frameworks/Matematicas.framework/Versions/A/Matematic
as
/Library/Frameworks/Matematicas.framework/Versions/A/Matematic
as:
   /Library/Framework/Matematicas.framework/Matematicas
   (compatibility version 1.0.0, current version 1.0.0)
   /usr/lib/libgcc_s.1.dylib
   (compatibility version 1.0.0, current version 1.0.0)
   /usr/lib/libmx.A.dylib
   (compatibility version 1.0.0, current version 92.0.0)
   /usr/lib/libSystem.B.dylib
   (compatibility version 1.0.0, current version 88.1.2)
```

Vemos que el binario que hemos generado tiene como números de versión menor (tanto actual como de compatibilidad) el valor `1.0.0`.

En el apartado 3.2 dijimos que siempre que íbamos a realizar una actualización de nuestro software deberíamos de actualizar la versión actual, y sólo cuando hacíamos cambios en la interfaz (p.e. añadir una función) deberíamos de actualizar la versión de compatibilidad, ya que las versiones

menores eran siempre compatibles hacia atrás, pero no necesariamente hacia adelante, es decir, una aplicación que enlazase con una versión superior no tenia garantizado el poder ejecutarse en un entorno donde está instalada una versión anterior, a no ser que las versión de compatibilidad fuesen las mismas.

Vamos a modificar ahora el programa del Listado 3.18 para optimizar la función `Suma()` de forma que no use una variable temporal[1] de la forma:

```
int Suma(int a, int b)
{
 return a+b;
}
```

Como este cambio es sólo una mejora de rendimiento, y no afecta a la interfaz, podemos recompilar la aplicación cambiando sólo la versión actual y manteniendo la versión de compatibilidad. Luego podemos recompilar la librería de la forma:

```
$ gcc Matematicas.c -dynamiclib -install_name /Library/Framewo
rk/Matematicas.framework/Versions/A/Matematicas –current_versi
on 1.0.1 -compatibility_version 1.0.0 -o Matematicas
```

Y la aplicación `usaframework` seguirá funcionando sin necesidad de recompilarla.

Si ahora decidiésemos añadir al programa del Listado 3.18 una función `Resta()` de la forma:

```
int Resta(int a, int b)
{
 return a-b;
}
```

Entonces no sólo deberíamos de aumentar la versión actual, sino que aumentamos la versión de compatibilidad de la forma:

```
$ gcc Matematicas.c -dynamiclib -install_name /Library/Framewo
rk/Matematicas.framework/Versions/A/Matematicas -current_versi
on 1.1.0 -compatibility_version 1.1.0 -o Matematicas
```

Aunque no es norma, si que se acostumbra a aumentar el tercer dígito de la versión cuando se hacen mejoras compatibles hacia adelante consistentes en mejoras del rendimiento o se arreglan bugs, mientras que cuando se añade funcionalidad no compatible hacia adelante se modifica el segundo dígito. Por último el primer dígito se suele incrementar cuando se cambia la versión mayor.

[1] Esta optimización la hubiera realizado automáticamente `gcc` si hubiéramos activado la optimización con la opción `-O`.

3.4. Frameworks privados

Además de los frameworks públicos, las aplicaciones tienen frameworks privados, que son frameworks que en vez estar destinados a ser compartidos entre varias aplicaciones, están destinados a ser usados sólo por una aplicación.

La ventaja de los frameworks privados es que la aplicación siempre tiene la versión correcta del framework. El inconveniente está en que no son compartidos entre varias aplicaciones.

Normalmente los frameworks privados se usan para que dentro de una aplicación grande se puedan desarrollar módulos más pequeños que no están destinados a ser compartidos con otras aplicaciones, sino a uso exclusivo por parte de nuestra aplicación.

Estos frameworks deben de almacenarse en un subdirectorio con el nombre `Frameworks` dentro del bundle de la aplicación. La Figura 3.5 muestra como organiza la aplicación Keynote los ficheros de su bundle. En la subcarpeta `MacOS` encontramos el fichero `Keynote`, que es el ejecutable de la aplicación. En la subcarpeta `Frameworks` encontramos dos frameworks privados.

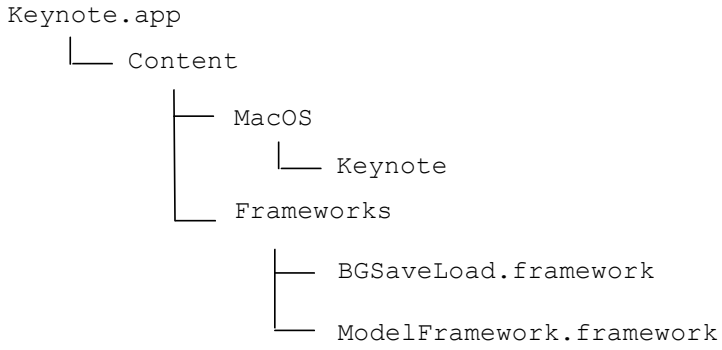


Figura 3.5: Ejemplo de frameworks privados

Los frameworks privados tienen siempre como comienzo del nombre de instalación el valor `@executable_path/../Frameworks`. Por ejemplo el nombre de instalación de la librería de enlace dinámico del framework `BGSaveLoad.framework` de la Figura 3.5 sería `@executable_path/../Frameworks/BGSaveLoad.framework/Versions/A/BGSaveLoad`, que es la ruta donde se encuentra alojada la librería de enlace dinámico del framework respecto a la aplicación.

3.5. Los umbrella frameworks

Los **umbrella frameworks** son una forma de agrupar varios frameworks en un sólo framework. Esto permite ocultar al programador complejas dependencias entre los subframeworks. Además de esta forma el programador no tiene porque saber que frameworks debe de importar y que dependencias existen entre los frameworks.

La estructura de un umbrella framework es similar a la de un framework normal, excepto en dos aspectos: El primero es que, como muestra la Figura 3.6, disponen de una carpeta con el nombre `Frameworks` donde se almacenan los subframeworks que lo componen. El segundo es que la carpeta `Headers` no contiene todos los ficheros de cabecera de cada subframework, sino que sólo contiene un fichero de cabecera maestro (`Quartz.h` en la Figura 3.6) el cual incluye todos los ficheros de cabecera de los subframeworks que hagan falta. Nunca debemos de incluir en nuestro programa los ficheros de cabecera de los subframeworks directamente, de hecho los ficheros de cabecera de los subframeworks tienen protegida su inclusión directa. Para ello los propios ficheros de cabecera detectan que falta por definir un identificador del fichero maestro y usan la directiva `#error` para avisar al programador de que lo correcto es incluir el fichero maestro.

En la Figura 3.6 también vemos la librería de enlace dinámico `Quartz`, que lo que hace es depender de las librerías de enlace dinámico de los subframeworks de forma que al cargar la librería de enlace dinámico del umbrella frameworks se cargan las librerías de enlace dinámico de los subframeworks.

Lógicamente un umbrella framework puede a su vez incluir otros umbrella frameworks creando estructuras más complejas.

Apple no recomienda a los desarrolladores externos crear umbrella frameworks, aunque permite hacerlo y proporciona herramientas para ello. La razón que da es que los umbrella frameworks son agrupaciones demasiado grandes y normalmente un framework no debería de tener tanta complejidad. Según Apple los únicos frameworks lo suficientemente complicados como para necesitar crear umbrella frameworks son los frameworks que implementan la funcionalidad del sistema operativo, y que se encuentran en la carpeta `/System/Library/Frameworks`.

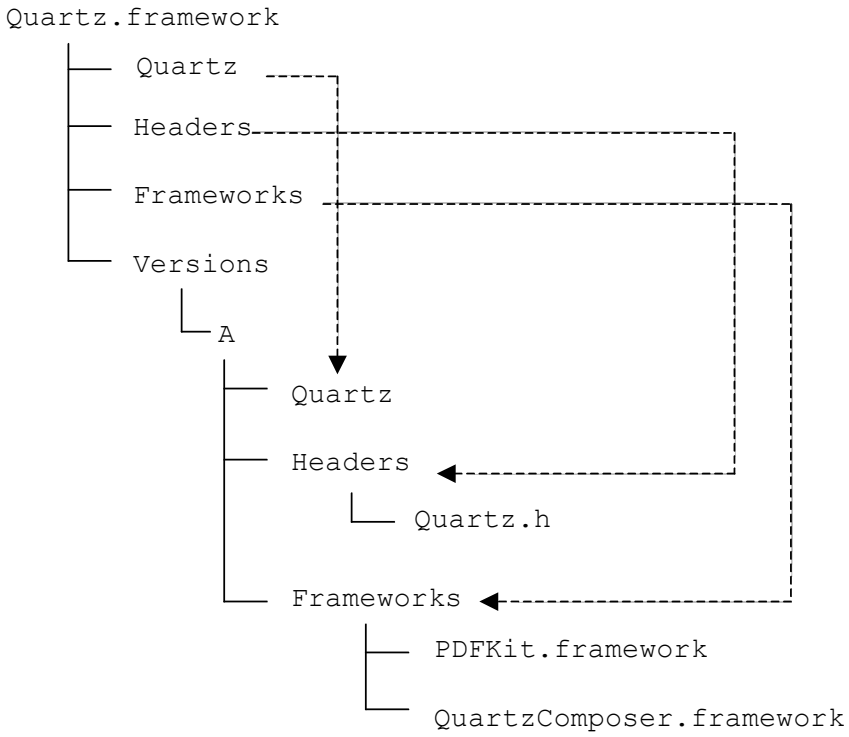


Figura 3.6: ejemplo de umbrella framework

Tema 4

Compilando en C++

Sinopsis:

En este tema pretendemos detallar el funcionamiento del compilador en lo que al lenguaje C++ concierne. En concreto estudiaremos las opciones particulares del compilador en lo que a C++ se refiere y las extensiones no estándar introducidas para el lenguaje.

En este tema también veremos aspectos referentes al name mangling de C++, y a cómo crear cabeceras precompiladas, las cuales mejoran considerablemente el tiempo de compilación de un programa.

1. El compilador de C++

Muchas de las opciones del compilador de C son heredadas por el compilador de C++, con lo que en este tema sólo trataremos las opciones específicas de C++. Lógicamente, antes de leer este tema conviene haber leído el Tema 2.

Los ficheros C++ se caracterizan por tener la extensión `.cc` o `.cpp`. Si suponemos que tenemos un fichero de código fuente como el del Listado 4.1, podemos compilarlo con el comando:

```
$ g++ hola.cpp -o hola
```

```
/* hola.cpp */

#include <iostream>
using namespace std;

int main()
{
 cout << "Hola mundo" << endl;
 return 0;
}
```

Listado 4.1: Programa mínimo C++

El comando `g++` es un comando que se limita a llamar a `gcc` añadiendo la opción de enlazado `-lstdc++` la cual hace que se use la librería de enlace estático `libstdc++.a` durante el enlazado. Esta librería tiene las funciones y clases de C++.

Observe que si ejecuta:

```
$ gcc hola.cpp -o hola
/usr/bin/ld: Undefined symbols:
std::basic_ostream<char,    std::char_traits<char>
>::operator<<(std::basic_ostream<char, std::char_traits<char>
>& (*)(std::basic_ostream<char, std::char_traits<char> >&))
std::ios_base::Init::Init()
std::ios_base::Init::~Init()
std::cout
collect2: ld returned 1 exit status
```

El enlazador fallará por no encontrar los símbolos de la librería de C++. Lógicamente siempre puede ejecutar el comando de la forma:

```
$ gcc hola.cpp -lstdc++ -o hola
```

En muchos sistemas (incluido Mac OS X) también existe el comando `c++` que es idéntico a `g++`:

```
$ c++ hola.cpp -o hola
```

De hecho, actualmente tanto `c++` como `g++` son enlaces al compilador actual de C++ (p.e. `g++-4.0`). Igualmente tanto el comando `cc` como `gcc` son enlaces al compilador de C actual (p.e. `gcc-4.0`). Como se explicó en el apartado 5 del Tema 2, estos enlaces se pueden cambiar a otra versión del compilador usando el comando `gcc_select`.

```
/* nombrefuncion.cpp */

#include <iostream>
using namespace std;

int Funcion1(int a, int b)
{
 cout << "Dentro de " << __func__ << endl;
 cout << "Dentro de " <<  __FUNCTION__ << endl;
 cout << "Dentro de " <<  __PRETTY_FUNCTION__ << endl;
 return 0;
}

class X
{
 public:
 int metodo1(double d)
      {
       cout << "Dentro de " << __func__ << endl;
       cout << "Dentro de " << __FUNCTION__ << endl;
       cout << "Dentro de " << __PRETTY_FUNCTION__ << endl;
      }
};

int main()
{
 Funcion1(2,4);
 X x;
 x.metodo1(3.7);
 return 0;
}
```

Listado 4.2: Programa que usa `__PRETTY_FUNCTION__`

2. Extensiones al lenguaje C++

En esta segunda parte del tema comentaremos algunas extensiones al lenguaje C++ introducidas por las GCC. Al igual que en el caso de C es posible que muchas de estas opciones sean aceptadas por los organismos de estandarización ANSI y ISO en futuras revisiones del lenguaje C++.

2.1. Obtener el nombre de una función y método

En el apartado 3.9 vimos que la palabra reservada `__FUNCTION__`, o su versión estándar ISO 99 `__func__`, devolvían el nombre de la función C dentro de la que se encontraban.

Si estamos programando en C++, junto con las anteriores palabras reservadas, tenemos la palabra reservada `__PRETTY_FUNCTION__`, que además de devolver el nombre de la función o método donde se encuentre situada, devuelve los parámetros y retorno de la función.

Por ejemplo, si ejecutamos el programa del Listado 4.2 obtendríamos la siguiente salida:

```
$ ./nombrefuncion
Dentro de Funcion1
Dentro de Funcion1
Dentro de int Funcion1(int, int)
Dentro de metodo1
Dentro de metodo1
Dentro de int X::metodo1(double)
```

2.2. Los operadores ?> y ?<

Estos operadores también son una extensión de las GCC al lenguaje C++ que permiten obtener el máximo o mínimo de sus operandos.

Por ejemplo, para obtener el mínimo o máximo de dos números haríamos respectivamente:

```
int min = a <? b;
int max = a >? b;
```

Estos operadores también se pueden sobrecargar. El Listado 4.3 muestra un ejemplo con el operador `>?` sobrecargado.

```
/* Numero.cpp */

#include <iostream>
using namespace std;

class Numero
{
 friend Numero operator>?(const Numero& n, const Numero& n2);
 private:
  int n;
 public:
  Numero(int n):n(n){}
  int getNumero() const {return n;};
};

Numero operator>?(const Numero& n, Numero& n2)
{
 return Numero(n.getNumero() >? n2.getNumero());
}

int main()
{
 Numero n1(4);
 Numero n2(3);
 Numero n3 = n1 >? n2;
 cout << "El maximo es " << n3.getNumero() << endl;
 return 0;
}
```

Listado 4.3: Ejemplo de sobrecarga del operador `>?`

3. Name mangling

Como ya sabemos, C++ usa name mangling para representar los símbolos, luego si ejecutamos `nm` sobre un fichero de código objeto:

```
$ nm Numero.o
00000000 W _ZN6NumeroC1Ei
00000000 W _ZNK6Numero9getNumeroEv
         U _ZNKSs4sizeEv
         U _ZNKSsixEj
0000014c T main
```

Obtenemos los símbolos con name mangling. Podemos usar la opción `--demangle` de `nm` para eliminar el name mangling.

```
$ nm --demangle Numero.o
00000000 W Numero::Numero(int)
00000000 W Numero::getNumero() const
         U std::string::size() const
         U std::string::operator[](unsigned int) const
0000014c T main
```

4. Cabeceras precompiladas

Los compiladores de C y C++ actualmente pasan la mayoría de su tiempo compilando una y otra vez cabeceras mucho más grandes que el código fuente del fichero `.c` o `.cpp` en sí. El compilador de GNU a partir de la versión 3.3 incluye la posibilidad de usar cabeceras precompiladas, la cual acelera mucho la compilación de programas C y C++. En este apartado se explica como aprovechar esta nueva característica

Para facilitar la explicación de las cabeceras precompiladas, supondremos que nuestros ficheros fuente incluyen en su mayoría el fichero `<iostream>` con las operaciones típicas de entrada/salida C++, y el fichero `<math.h>` con las operaciones matemáticas más normales.

En consecuencia hemos decidido crear un fichero precompilado con estos ficheros de cabecera, con el fin de acelerar la compilación de nuestro proyecto. El fichero de cabecera, en general, deberá incluir sólo ficheros de cabecera que cambien poco, preferiblemente sólo los que vienen con el compilador (los que tienen nombres `#include <...>`), y no los ficheros de cabecera de nuestro programa (que tienen nombres `#include "..."`), ya que sino tendría que recompilarse el fichero de cabecera cada vez que modificáramos uno de nuestros ficheros `.h`.

Los ficheros de cabecera precompilados tienen el mismo nombre que el fichero `.h` correspondiente, pero con el sufijo `.gch`. Por ejemplo si creamos el fichero `precompiled.h` con las cabeceras comunes que incluye nuestro proyecto, el fichero de cabecera precompilado se deberá llamar `precompiled.h.gch`.

El fichero que nosotros hemos creado como ejemplo es:

```
$ cat precompiled.h
#include <iostream>
#include <math.h>
using namespace std;
```

Lo siguiente que tenemos que hacer es precompilar este fichero, para lo cual ejecutamos:

```
$ g++ -c precompiled.h -o precompiled.h.gch
```

Lo cual nos crea el fichero `precompiled.h.gch` con todos los ficheros incluidos por `precompiled.h` precompilados.

Ahora si, por ejemplo, el fichero `encuadre.cpp` incluye el fichero `precompiled.h`, cuando ejecutemos:

```
$ g++ -c encuadre.cpp -o encuadre.o
```

El compilador de GNU buscará primero en el disco un fichero llamado `precompiled.h.gch`, si lo encuentra lo usará, y sólo si no lo encuentra incluirá `precompiled.h`.

Aquí conviene hacer varias apreciaciones: La primera es que para que el compilador de GNU use el fichero `.gch` que encuentre, éste deberá estar compilando exactamente con las mismas opciones con las que hemos lanzado la compilación de `encuadre.cpp`, es decir basta con que hubiéramos usado:

```
$ g++ -O1 -c encuadre.cpp -o encuadre.o
```

Para que no nos acepte el fichero precompilado y use el sin precompilar.

La segunda apreciación es que el compilador de GNU requiere un fichero precompilado para cada dialecto de C (C, C++, Objective-C). Si quisiésemos que un fichero precompilado se usase por distintos dialectos de C, necesitaríamos crear uno distinto para cada dialecto (usando la opción `-x c-header`, `-x c++-header` y `-x objective-c-header`). En este caso la forma de proceder es crear un directorio con el nombre `precompiled.h.gch` y dentro de él meter todos los ficheros precompilados para todos los dialectos que estemos usando (da lo mismo el nombre que demos a estos ficheros, `gcc` examina todos los ficheros del directorio en busca del que necesite).

La tercera apreciación es que si vamos a usar ficheros precompilados en un fichero `Makefile` conviene crear una regla que actualice los ficheros de cabecera precompilados como la siguiente:

```
encuadre.o: encuadre.cpp encuadre.h Matriz.h Punto.h
precompiled.h.gch
precompiled.h.gch: precompiled.h
	$(CXX) $^ $(CXXFLAGS) -o $@
```

Desgraciadamente actualmente `make` no soporta el uso de reglas implícitas para los ficheros precompilados.

5.Un parche para el enlazador

En muchos lenguajes orientados a objeto, como C++, es necesario que el programa ejecute constructores estáticos antes de que el hilo principal del programa empiece a ejecutar. Al enlazador le resulta extremadamente difícil modificar el código objeto para añadir estas inicializaciones con lo que las GCC optaron por solucionar este problema con el comando `collect2`.

El driver `gcc` actualmente en vez de enlazar código C++ ejecutando el comando `ld`, enlaza ejecutando el comando `collect2`. Este comando detecta constructores estáticos que deban de ser ejecutados antes de que empiece la ejecución de la función `main()` (es decir, símbolos marcados con el atributo `constructor`), genera un fichero `.c` temporal con una función `__main()` que llama a estos constructores, compila el fichero temporal y altera `main()` para que llame a `__main()`. Una vez hecho esto `collect2` ya puede llamar a `ld` para que realice el enlazado real.

El comando `collect2` recibe las mismas opciones que `ld`, y transmite a `ld` las opciones recibidas. El comando `collect2` no sólo llama a `ld`, sino que después llama a `strip` para eliminar símbolos innecesarios del binario generado.

Tema 5

Compilando en Java

Sinopsis:

Java es diferente a los otros lenguajes soportados por las GCC respecto a que en los demás lenguajes el compilador genera un binario ejecutable, mientras que Java genera bytecodes ejecutables en una máquina virtual.

El compilador Java de las GCC se diferencia de otros compiladores Java en que, no sólo es capaz de generar bytecodes, sino que además es capaz de convertir estos bytecodes en código binario directamente ejecutable en la plataforma destino. Esta peculiaridad permite generar código Java que ejecuta considerablemente más rápido que su correspondiente versión de bytecodes. En este tema estudiaremos como llevar esto a cabo.

1. Compilando Java con las GCC

Para que una clase Java sea ejecutable debe de tener un método estático `main()` como el del Listado 5.1.

```
public class HolaMundo
{
 public static void main(String[] args)
        {
         System.out.println("Hola mundo");
        }
}
```

Listado 5.1: Programa mínimo Java

Como sabrá, este programa se puede compilar con el compilador `javac` y ejecutar así:

```
$ javac HolaMundo.java
$ ls -l HolaMundo.class
-rw-r--r--  1 fernando admin 422 Feb 5 14:29 HolaMundo.class
$ time java HolaMundo
Hola mundo
real    0m1.126s
user    0m0.310s
sys     0m0.198s
```

Si lo que queremos es generar un binario, debemos usar el comando `gcj`[1] con la opción `--main` , la cual usamos para indicar la clase donde está el método `main()` de la forma:

```
$ gcj --main=HolaMundo HolaMundo.java -o HolaMundo
$ time HolaMundo
Hola mundo
real    0m0.286s
user    0m0.097s
sys     0m0.088s
```

En este ejemplo puede apreciar la mejora en rendimiento temporal que conseguimos con la compilación binaria.

[1] Tenga en cuenta que, como indicamos en el apartado 3 del Tema 1, el comando `gcj`, al igual que otros muchos comandos que vamos a ver en este tema, no viene con las Developer Tools de Apple, sino que deberá instalárselo de otra distribución, como por ejemplo Fink. Además, al menos en el momento de escribir este tutorial, las `gcj` tool estaban marcadas como inestables en Fink, con lo que nos ha sido necesario habilitar el branch `unstable/main` para que Fink sí que las instalase.

Tenga en cuenta que, como los binarios ejecutables están libres de la convención de nombres Java, el nombre de fichero de salida que demos a la opción `-o` no tiene porque llamarse igual que la clase.

La diferencia de rendimiento que hemos mostrado más arriba es un poco engañosa ya que las GCC, al menos en el momento de escribir este documento, generan bytecodes más optimizados que la herramienta `javac` de Apple.

Es decir, podemos pasar el programa del Listado 5.1 a bytecodes con el comando `gcj` y la opción `-C`:

```
$ gcj -C HolaMundo.java
$ ls -l HolaMundo.class
-rw-r--r--  1 fernando  admin  406 Feb 5 14:28 HolaMundo.class
```

Donde puede observar que el fichero de bytecodes tiene menos bytes que cuando lo compilamos con `javac` de Apple.

Podemos ejecutar ahora estos bytecodes tanto con el comando `java` de Apple, como con el comando `gij` de las GCC:

```
$ time java HolaMundo
Hola mundo
real    0m0.699s
user    0m0.306s
sys     0m0.187s
$ time gij HolaMundo
Hola mundo
real    0m0.358s
user    0m0.124s
sys     0m0.093s
```

Donde vemos que las GCC no sólo generan bytecodes más optimizados, sino que la máquina virtual de las GCC ejecuta los bytecodes más rápido que la máquina virtual de Apple.

Obsérvese que para compilar a bytecodes hemos usado la opción `-C`, ya que `-c` sirve, al igual que normalmente, para generar un fichero de código objeto:

```
$ gcj HolaMundo.java -c
$ nm HolaMundo.o
0000016c d __CD_HolaMundo
00000174 d __CT_HolaMundo
000000c8 s __Utf1
00000000 T __ZN9HolaMundo4mainEP6JArrayIPN4java4lang6StringEE
```

O si preferimos generar código ensamblador también podemos usar la opción `-S`:

```
$ gcj -S HolaMundo.java
```

Conviene comentar que además de aceptar código fuente Java, `gcj` también acepta bytecodes:

```
$ gcj HolaMundo.class --main=HolaMundo -o HolaMundo
```

O incluso podemos darle como entada un fichero `.jar`, del cual `gcj` saca la clase indicada y la compila para generar el ejecutable:

```
$ jar cf libsaludos.jar HolaMundo.class
$ gcj --main=HolaMundo libsaludos.jar -o HolaMundo
```

Ahora para ejecutar el programa no necesitaríamos adjuntar el fichero `.jar`:

```
$ rm libsaludos.jar
$ ./HolaMundo
Hola mundo
```

2. Utilidades Java de las GCC

Las GCC, además de los comandos `gcj` y `gij` traen otros muchos comandos de utilidad para Java que se resumen en la Tabla 5.1.

Comando	Descripción
`gcj`	Compilador Java de las GCC.
`gij`	Interprete Java de las GCC.
`gcjh`	Los métodos nativos Java se pueden escribir tanto en el estándar JNI (Java Native Interface) de Sun, como en CNI (Compiled Native Interface), una interfaz C++ para métodos nativos desarrollada por las GCC. El comando `gcjh` nos permite generar los prototipos CNI (o JNI si usamos la opción `-jni`) de los métodos a implementar.
`jcf-dump`	Nos permite obtener información sobre el contenido de un fichero `.class`. Usando la opción `-javap` podemos obtener la salida en el mismo formato que el comando `javap` de Sun.
`jv-scan`	Recibe un fichero de código fuente Java y produce distintas informaciones sobre éste. Por ejemplo, con la opción `--complexity` nos devuelve la complejidad ciclomática de cada clase. Consulte `man` para una información más detallada.
`grepjar`	Busca una expresión regular en un fichero `.jar`. Por ejemplo `grepjar "main" libsaludos.jar` busca la palabra `main` en los ficheros de `libsaludos.jar`.
`fastjar`	Una implementación del comando `jar` de Sun que ejecuta considerablemente más rápido.

Tabla 5.1: Comandos Java de las GCC

Tema 6

Combinar distintos lenguajes

Sinopsis:

En este tema vamos a comentar los detalles necesarios para combinar programas escritos en distintos lenguajes.

Debido a que las GCC utilizan el mismo backend para producir código objeto para los distintos lenguajes, resulta relativamente sencillo combinar programas escritos en distintos lenguajes.

En cualquier caso el programador deberá tener en cuenta una serie de aspectos relativos al nombrado de símbolos en los distintos lenguajes, name mangling, paso de argumentos, conversión entre tipos de datos, manejo de errores y librerías de runtime de los distintos lenguajes.

1.Combinar C y C++

El lenguaje C y C++ combinan de forma natural debido a que C++ se diseñó como una extensión a C. Una consecuencia es que el mecanismo de paso de parámetros de ambos lenguajes es el mismo.

Una diferencia la encontramos en el mecanismo de nombrado de símbolos. Mientras que en C los símbolos usados para referirse a los nombres de las funciones no usan los parámetros, en C++ estos símbolos siempre incluyen información sobre los parámetros de la función. Sin embargo vamos a ver que C++ tiene un mecanismo para poder referirse a las funciones usando símbolos con la convención de nombrado de C.

1.1. Llamar a C desde C++

El Listado 6.1 muestra un programa C++ que ejecuta la función C definida en el Listado 2.4. Para ello el programa C++ declara el prototipo de la función C con el modificador `extern "C"`.

```
/* pidesaludo.cpp */

extern "C" void Saluda();

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

Listado 6.1: Programa C++ que ejecuta una función C

Para compilar los dos módulos y generar el ejecutable podemos usar:

```
$ g++ -c pidesaludo.cpp
$ gcc -c saluda.c
$ gcc saluda.o pidesaludo.o -lstdc++ -o pidesaludo
```

Obsérvese que hemos necesitado incluir la librería de C++ durante el enlazado.

Como normalmente un programa C++ necesita acceder a más de una función C, es más común declarar estas funciones de la forma:

```
extern "C" {
 void Saluda();
 int Suma(int a, int b);
 void Despide(); }
```

Como las declaración de las funciones se suele hacer en un fichero de cabecera, es muy común encontrar una declaración que sirva tanto para C como para C++ de la siguiente forma:

```
#ifdef __cplusplus
extern "C"{
#endif

 void Saluda();
 int Suma(int a, int b);
 void Despide();

#ifdef __cplusplus
}
#endif
```

Donde el identificador `__cplusplus` es un identificador que sólo está definido cuando compilamos con el compilador de C++.

1.2. Llamar a C++ desde C

Para que un programa C pueda llamar a una función C++ es necesario que la función C++ sea declarada con el modificador `extern "C"`, de esta forma el símbolo que genera el compilador de C++ para esta función será compatible con el nombrado de símbolos de C.

El Listado 6.2 muestra una función C++ declarada con un nombrado de símbolo al estilo C. Obsérvese que aunque el prototipo es de tipo C, la implementación de la función puede tener instrucciones propias del lenguaje C++.

```
/* saludocpp.cpp */

#include <iostream>

extern "C" void SaludoCPP()
{
 std::cout << "Hola desde C++" << std::endl;
}
```

Listado 6.2: Función C++ llamable desde C

Si compilamos este módulo y usamos `nm` para obtener información sobre sus símbolos veremos que el símbolo `_SaludoCPP` sigue la convención de nombrado de C:

```
$ g++ -c saludocpp.cpp
$ nm saludocpp.o
00000000 T _SaludoCPP
         U __ZSt4cout
```

Podemos hacer un programa C como el del Listado 6.3 que llama a la función C++ (definida con prototipo al estilo de C).

```
/* pidesaludo.c */

void SaludoCPP();

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

Listado 6.3: Programa C que llama a una función implementada en C++

Y podemos compilar este módulo C y enlazarlo con el módulo C++ de la forma:

```
$ gcc -c pidesaludo.c
$ gcc pidesaludo.o saludocpp.o -lstdc++ -o pidesaludo
```

2. Acceso a C desde Java

Podemos usar **JNI (Java Native Interface)** para comunicarnos entre clases Java ejecutando en una maquina virtual y librerías de enlace dinámico nativas escritas en C, C++ o ensamblador[1].

JNI fue diseñado, y es más útil, cuando un programa que se está ejecutando en una máquina virtual Java quiere acceder a funcionalidad nativa del sistema donde se está ejecutando. En el apartado 2.4 veremos que también se puede usar JNI cuando un programa C quiere acceder a funcionalidad de una clase Java.

Hay que tener en cuenta que el acceso desde Java a funcionalidad nativa elimina el principio de independencia de la plataforma que caracteriza a los programas Java. Aunque esta limitación se puede paliar en parte escribiendo la librería nativa para varios sistemas.

2.1. Una clase Java con un método nativo

La forma más usada por JNI para combinar C y Java es crear una clase Java con un **método nativo**, es decir un método que ejecuta código C.

Lógicamente el método nativo también podría estar implementado en C++ o incluso en ensamblador, pero vamos a suponer que lo vamos a implementar en C.

```
/* HolaNativo.java */

public class HolaNativo
{
 static {
         System.loadLibrary("HolaNativo.dylib");
        }
 public static void main(String[] args)
        {
         HolaNativo h = new HolaNativo();
         h.saluda();
        }
  public native void saluda();
 }
```

Listado 6.4: Clase Java con un método nativo

[1] En este apartado vamos a resumir como funciona JNI. Si desea profundizar en esta tecnología puede consultar el "Tutorial de JNI" que tenemos publicado en la web de macprogramadores.org.

Como ejemplo, el Listado 6.4 muestra una clase con un método nativo `saluda()`, y un método `main()` que ejecuta el método nativo. El método nativo se declara como parte de la clase, pero no se implementa, ya que su implementación se encuentra en una librería de enlace dinámico. La clase también tiene un inicializador estático que llama al método estático `System.loadLibrary()` para cargar la librería de enlace dinámico que tiene el método nativo implementado.

Podemos compilar la clase Java con el comando:

```
$ gcj -C HolaNativo.java
```

Una vez tengamos el fichero `HolaNativo.class`, podemos ejecutar el comando `gcjh` con la opción `-jni` de la forma:

```
$ gcjh -jni HolaNativo
```

Este comando genera un fichero con el nombre `HolaNativo.h` que contiene el prototipo de la función que debemos implementar para responder a las llamadas al método nativo.

```
$ cat HolaNativo.h
/* DO NOT EDIT THIS FILE - it is machine generated */

#ifndef __HolaNativo__
#define __HolaNativo__

#include <jni.h>

#ifdef __cplusplus
extern "C"
{
#endif

JNIEXPORT void JNICALL Java_HolaNativo_saluda (JNIEnv *env,
jobject);

#ifdef __cplusplus
}
#endif

#endif /* __HolaNativo__ */
```

Vemos que el nombre de la función a implementar está formado por el símbolo `Java_` seguido del nombre de la clase y del nombre del método a implementar. En el ejemplo este nombre será `Java_HolaNativo_saluda()`. Además las funciones que implementan el método nativo siempre tienen al menos dos parámetros (a pesar de que el método Java no tenga parámetros). El primer parámetro es un puntero a información de entorno de

JNI, y el segundo es un puntero al objeto sobre el que se ejecuta el método (es decir, el puntero `this` del objeto Java).

Podemos implementar la función del método nativo como muestra el Listado 6.5.

```
/* HolaNativo.c */

#include <jni.h>
#include "HolaNativo.h"

void JNICALL Java_HolaNativo_saluda (JNIEnv *env
                                     ,jobject this)
{
 printf("Hola desde C\n");
}
```

Listado 6.5: Implementación de un método nativo

Ahora podemos compilar la librería de enlace dinámico:

```
$ gcc  HolaNativo.c -dynamiclib -o libHolaNativo.dylib
```

Observe que el nombre de la librería debe ser el mismo que dimos a `System.loadLibrary()`, pero precedido por `lib`.

Por último, suponiendo que la librería de enlace dinámico esté en un directorio accesible por la librería (por ejemplo el mismo directorio que la clase) podemos ejecutar el programa Java en una máquina virtual así:

```
$ gij HolaNativo
Hola desde C
```

Tipo Java	Tipo C	Descripción
`boolean`	`jboolean`	8 bits sin signo.
`byte`	`jbyte`	8 bits con signo.
`char`	`jchar`	16 bits sin signo.
`short`	`jshort`	16 bits con signo.
`int`	`jint`	32 bits con signo.
`long`	`jlong`	64 bits con signo.
`float`	`jfloat`	32 bits formato IEEE.
`double`	`jdouble`	64 bits formato IEEE.
`boolean`	`jboolean`	8 bits sin signo.

Tabla 6.1: Tipos de datos fundamentales Java y su declaración correspondiente en C

2.2. Tipos de datos en JNI

Los tipos de datos de C y Java son parecidos pero no son exactamente los mismos. En el fichero `jni.h` encontramos definidos los tipos de datos Java mediante `typedef`. La Tabla 6.1 resume estos tipos de datos.

2.3. Pasar parámetros a un método nativo

Como con cualquier otro método Java, es posible pasar argumentos y retornar valores de métodos nativos.

Por ejemplo, si tenemos una clase como la del Listado 6.6 con el método nativo `suma()` con parámetros y retorno, al igual que antes, podemos obtener el prototipo del método nativo en C con sólo compilar la clase Java y usar `gcjh` para generar el prototipo de la función C del método nativo:

```
$ gcj -C SumaNativa.java
$ gcjh -jni SumaNativa
$ cat SumaNativa.h
.................
JNIEXPORT jint JNICALL Java_SumaNativa_suma (JNIEnv *env,
jobject, jint, jint);
```

Obsérvese que además de los dos parámetros que siempre tiene la función del método nativo, tiene otros dos parámetros de tipo `jint`, y también retorna un `jint`.

```
/* SumaNativa.java */

public class SumaNativa
{
 static{
 System.loadLibrary("SumaNativa.dylib");
       }
 public static void main(String[] args)
       {
        SumaNativa s = new SumaNativa();
        System.out.println("3+4=" + s.suma(3,4));
       }
 public native int suma(int a, int b);
}
```

Listado 6.6: Clase Java con parámetros en un método nativo

Ahora podemos implementar el método nativo tal como muestra el Listado 6.7, y generar la librería de enlace dinámico correspondiente con el comando:

```
$ gcc SumaNativa.c -dynamiclib -o libSumaNativa.dylib
```

Por último, al igual que antes, podemos ejecutar el programa Java con método nativo:

```
$ gij SumaNativa
3+4=7
```

```
/* SumaNativa.h */
#include <jni.h>
#include "SumaNativa.h"

jint JNICALL Java_SumaNativa_suma (JNIEnv *env, jobject this,
jint a, jint b)
{
 jint total = a+b;
 return total;
}
```

Listado 6.7: Implementación de un método nativo con parámetros

2.4. Acceso a clases Java desde un método nativo

Aunque un programa Java puede instanciar una máquina virtual y ejecutar desde ella clases Java, en este tutorial no vamos a explicar como se hace[1]. Lo que sí vamos a explicar es cómo, un método nativo, que ha sido ejecutado desde un programa Java, puede volver a acceder a la clase Java que le llamó.

```
/* Teclado.Java */

public class Teclado
{
 static{
        System.loadLibrary("Teclado.dylib");
       }
 public static void main(String[] args)
        {
         Teclado t = new Teclado();
         System.out.print("Escriba un numero:");
         t.leeNumero();
        }
 // Metodo nativo
 public native void leeNumero();
 // Metodo callback
 public void leidoNumero(int n)
        {
         System.out.println("El numero leido es " + n);
        }
}
```

Listado 6.8: Clase Java con un método nativo que llama a un método callback de la clase

[1] Para saber como se hace esto puede consultar el "Tutorial de JNI" publicado en macprogramadores.org

Como ejemplo usaremos la clase Java del Listado 6.8 la cual dispone del método nativo `leeNumero()`, el cual, cuando lee un número, llama al método Java `leidoNumero()`.

El Listado 6.9 muestra la implementación del método nativo. El método nativo llama a la función JNI `GetObjectClass()` para obtener un puntero a la clase del objeto al que pertenece el método (apuntado por `this`). Después, usando la función JNI `GetMethodID()` obtiene el método callback `leidoNumero()`, que es el método Java al que queremos llamar desde C. Por último utiliza la función JNI `CallVoidMethod()` para llamar al método Java.

La función `GetMethodID()`, además del nombre del método, necesita conocer sus parámetros, ya que en Java existe la sobrecarga. Para ello utiliza una signatura inventada por JNI consistente en indicar entre paréntesis los tipos de los parámetros, y después indicar el tipo de retorno. `"(I)V"` significa que el método recibe un `int` y que devuelve `void`. Puede consultar la documentación de referencia de está función para aprender más sobre las signaturas.

```
/* Teclado.h */

#include <jni.h>
#include "Teclado.h"

JNIEXPORT void JNICALL Java_Teclado_leeNumero (JNIEnv *env
                                               ,jobject this)
{
 jclass clase = (*env)->GetObjectClass(env,this);
 jmethodID id =
      (*env)->GetMethodID(env,clase,"leidoNumero","(I)V");
 if (id==0)
    {
     printf("Metodo leidoNumero() no encontrado");
     return;
    }
 int numero;
 int ret = scanf("%i",&numero);
 if (ret!=0)
    (*env)->CallVoidMethod(env,this,id,numero);
 else
    printf("Numero no valido\n");
}
```

Listado 6.9: Método nativo que llama a un método Java

Por último, podemos compilar el programa Java y la librería nativa con:

```
$ gcj -C Teclado.java
$ gcjh -jni Teclado
$ gcc Teclado.c -dynamiclib -o libTeclado.dylib
```

Y ejecutar el programa que hemos hecho con:

```
$ gij Teclado
Escriba un numero:4
El numero leido es 4
```

Tema 7

Depuración, optimización y perfilado

Sinopsis:

A estas alturas el lector ya sabrá manejar las herramientas de GNU para generar programas.

Para terminar este tutorial pretendemos estudiar las habilidades del título de este último tema: Cómo se hace para depurar las aplicaciones cuando no se comportan de acuerdo a nuestros objetivos. Cómo pedir a las GCC que optimicen el código generado. Cómo detectar la corrupción y pérdida de memoria. Y cómo perfilar el código, es decir, cómo detectar cuellos de botella en trozos del programa que si optimizamos podemos conseguir un programa mucho menos ávido de recursos, y en consecuencia más agradable de usar.

1. Depurar aplicaciones

En este primer apartado aprenderemos a usar el comando `gdb` (GNU Debugger) para depurar una aplicación generada con las GCC.

1.1. Generar código depurable

Para generar código depurable debemos usar la opción `-g` tanto durante la compilación como durante el enlazado de la aplicación. A esta opción se la puede preceder por un nivel de información de depuración de acuerdo a la Tabla 7.1. Si no se indica nivel, por defecto se usa `-g2`.

Nivel	**Descripción**
1	Este nivel incluye la mínima cantidad de información de depuración en el fichero de código objeto. La información es suficiente para trazar las llamadas a funciones y examinar el valor de las variables globales, pero no hay información que relacione el código ejecutable con el fuente, ni información que nos permita evaluar las variables locales.
2	Este es el nivel por defecto. Incluye toda la información del nivel 1, junto con información que relaciona el código objeto con el fuente, y información sobre los nombres y posiciones de las variables locales.
3	Este nivel incluye toda la información del nivel anterior, y además añade información sobre la definición de los macros del preprocesador.

Tabla 7.1: Niveles de información de depuración

1.2. Cargar un programa en el depurador

Supongamos que tenemos el programa del Listado 7.1, el cual al irlo a ejecutar curiosamente falla:

```
$ gcc fibonacci.c -o fibonacci
$ ./fibonacci 4
Bus error
```

Si ojeando el programa no encontramos ninguna razón para que falle, vamos a necesitar depurarlo. Para ello introducimos información de depuración sobre el programa en el código ejecutable (usando la opción `-g`) y lo cargamos en el depurador con los comandos:

```
$ gcc fibonacci.c -g -o fibonacci
$ gdb fibonacci 4
(gdb)
```

Tras ejecutar el depurador obtenemos el prompt `(gdb)` donde para ejecutar el programa podemos escribir el comando `run`, el cual iniciaza la ejecución del programa:

```
(gdb) run
Starting program: ./fibonacci
Reading symbols for shared libraries . done
Indique un numero como argumento
Program exited normally.
```

Nuestro programa está diseñado para recibir como argumento el número de Fibonacci a calcular (véase Listado 7.1). Debido a que no hemos suministrado este argumento, nuestro programa acaba con un mensaje donde se indica que no se ha recibido este argumento.

Puede abandonar el depurador con el comando `quit`:

```
(gdb) quit
```

```
/* buclefor.c */

#include <stdio.h>

int Fibonacci(int n)
{
 if (n<=1)
    return 1;
 return Fibonacci(n-1)+Fibonacci(n-2);
}

int main(int argc, char* argv[])
{
 if (argc!=2)
    {
     printf("Indique un numero como argumento");
     return 0;
    }
 int n;
 sscanf(argv[1],"%n",&n);
 int sol = Fibonacci(n);
 printf("El fibonacci de %n es %n\n",n,sol);
 return 0;
}
```

Listado 7.1: Programa C defectuoso

Es importante recordar que para pasar argumentos a un programa no podemos hacer:

```
$ gdb fibonacci 4
./4: No such file or directory.
Unable to access task for process-id 4: (os/kern) failure.
(gdb)
```

Ya que en este caso `gdb` interpreta el argumento del programa a depurar como un argumento de `gdb`.

Para pasar argumentos al programa a depurar debemos usar la opción `--args` de `gdb`. Esta opción se debe poner siempre en último lugar, ya que detiene la interpretación de opciones por parte de `gdb`, y todo lo que pongamos detrás de esta opción se considera el nombre del programa a depurar y los argumentos del éste.

```
$ gdb --args fibonacci 4
(gdb)
```

Ahora podemos ejecutar el programa `fibonacci` con argumentos:

```
(gdb) run
./fibonacci 4
Reading symbols for shared libraries . done
Program received signal EXC_BAD_ACCESS, Could not access
memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x00000000
0x9000d3e4 in __vfprintf$LDBL128 ()
```

La ejecución con `run` nos da algo más de información. Parece ser que se ha producido un acceso a la dirección de memoria 0, el cual ha activado la señal `EXC_BAD_ACCESS`. Además `gdb` nos informa de que el problema se ha producido en la función `printf()`. De hecho el problema se debe al común error de los programadores de usar `%n` (el parámetro es un puntero a entero) en vez de `%i` (el parámetro es un entero). De hecho si hubiéramos usado la recomendable opción `-Wall` hubiéramos descubierto antes este error:

```
$ gcc fibonacci.c -g -o fibonacci -Wall
fibonacci.c: In function 'main':
fibonacci.c:27: warning: format '%n' expects type 'int *', but
argument 2 has type 'int'
fibonacci.c:27: warning: format '%n' expects type 'int *', but
argument 3 has type 'int'
```

Corregido este error descubrimos que el programa anterior todavía no funciona correctamente:

```
$ gcc fibonacci.c -g -o fibonacci -Wall
$ ./fibonacci 4
El Fibonacci de 0 es 1
```

Vemos que el programa responde que la posición 0 de la sucesión de Fibonacci es 1, cuando lo que le hemos preguntado es cuál es la cuarta posición. Luego vamos a necesitar volver a ejecutar el depurador.

En este caso vamos a ejecutar el programa a depurar paso a paso para ver como se llega a este extraño resultado.

```
$ gdb --args fibonacci 4
(gdb)
```

Lo primero que vamos a hacer es ejecutar el comando `list` el cual nos muestra 10 líneas del código fuente asociado al programa desde la posición donde estemos, o desde el principio del fichero si no ha comenzado su ejecución:

```
(gdb) list
5       int Fibonacci(int n)
6       {
7        if (n<=1)
8           return 1;
9        return Fibonacci(n-1)+Fibonacci(n-2);
10      }
11
12      int main(int argc, char* argv[])
13      {
14       if (argc!=2)
(gdb)
```

Al comando `list` también le podemos dar como argumento un número de línea a partir de la cual queremos listar. En este caso también listaría 10 líneas. Otra opción es darle al comando `list` dos números separados por coma como argumento para indicar la primera y última línea a listar.

```
(gdb) list 5,20
5       int Fibonacci(int n)
6       {
7        if (n<=1)
8           return 1;
9        return Fibonacci(n-1)+Fibonacci(n-2);
10      }
11
12      int main(int argc, char* argv[])
13      {
14       if (argc!=2)
15          {
16           printf("Indique un numero como argumento");
17           return 0;
18          }
19       int n;
20       sscanf(argv[1],"%n",&n);
```

1.2.1. Breakpoints y depuración paso a paso

Debido a que `run` ejecuta el programa todo seguido hasta el final, lo que vamos a hacer es poner un breakpoint al principio de la función `main()`. Para ello usaremos el comando `break`, el cual nos permite poner un breakpoint en un determinado número de línea o nombre de función. En este caso vamos a poner un breakpoint en la función `main()` y otro en la función `Fibonacci()`:

```
(gdb) break main
Breakpoint 1 at 0x29bc: file fibonacci.c, line 14.
(gdb) break Fibonacci
Breakpoint 2 at 0x2934: file fibonacci.c, line 7.
```

Podemos consultar los breakpoints fijados con el comando `info breakpoints`:

```
(gdb) info breakpoint
N  Type       Disp En Address
1  breakpoint keep y  0x000029bc in main at fibonacci.c:14
2  breakpoint keep y  0x00002934 in Fibonacci at fibonacci.c:7
```

O bien eliminar un breakpoint con el comando `clear` que borra el breakpoint cuyo número de línea o nombre de función especifiquemos. Por ejemplo, para borrar el breakpoint que hemos puesto en la función `Fibonacci()` podemos usar:

```
(gdb) clear 7
Deleted breakpoint 2
```

Una vez puesto el breakpoint en `main()` podemos iniciar la ejecución de la función hasta el breakpoint de la función `main()` ejecutando el comando `run`:

```
(gdb) run
Starting program: ./fibonacci 4
Reading symbols for shared libraries . done
Breakpoint 1, main (argc=2, argv=0xbffff994) at fibonacci.c:14
14       if (argc!=2)
```

Y luego podemos inspeccionar el valor de la variable `argc` con el comando `print`:

```
(gdb) print argc
$1 = 2
```

Claramente el valor recibido es el esperado, luego podemos continuar la ejecución paso a paso con el comando `next`, el cual por defecto avanza un paso, aunque también le podemos pasar como argumento el número de pasos a avanzar como argumento:

```
(gdb) next
20        sscanf(argv[1],"%n",&n);
```

El comando `next` ejecuta la línea 14, y debido a que no se cumple la condición de la sentencia `if`, nos informa que la siguiente línea que se ejecutará será la 20.

Ejecutamos un paso más, y comprobamos si la variable `n` se ha inicializado correctamente con el valor recibido como argumento:

```
(gdb) next 1
21        int sol = Fibonacci(n);
(gdb) print argv[1]
$2 = 0xbffffa46 "4"
(gdb) print n
$3 = 0
```

Parece ser que `n` no se ha inicializado con el 4 que había en `argv[1]`. De nuevo el problema se debe a que `scanf()` ha usado `%n` en la cadena de formato, en vez de `%i`.

Abandonamos la ejecución del depurador con el comando `quit`, corregimos el error, y volvemos a intentarlo:

```
$ gcc fibonacci.c -g -o fibonacci -Wall
$ gdb --args fibonacci 4
(gdb)
```

Esta vez podemos fijar directamente el breakpoint en la línea 21, y comprobamos que ahora el argumento se lee correctamente:

```
(gdb) break 21
Breakpoint 1 at 0x2a04: file fibonacci.c, line 21.
(gdb) run
Starting program: ./fibonacci 4
Reading symbols for shared libraries . done
Breakpoint 1, main (argc=2, argv=0xbffff994) at fibonacci.c:21
21        int sol = Fibonacci(n);
(gdb) print n
$1 = 4
```

El comando `next` ejecutaría la función `Fibonacci()` completamente sin entrar en ella paso a paso. Si lo que queremos es meternos dentro de una función podemos usar el comando `step` de la siguiente forma:

```
(gdb) step
Fibonacci (n=4) at fibonacci.c:7
7         if (n<=1)
```

La función `Fibonacci()` es recursiva y si continuamos usando `step` vamos a irnos metiendo en las sucesivas llamadas recursivas:

```
(gdb) step
9         return Fibonacci(n-1)+Fibonacci(n-2);
(gdb) step
Fibonacci (n=3) at fibonacci.c:7
7         if (n<=1)
(gdb) step
9         return Fibonacci(n-1)+Fibonacci(n-2);
(gdb) step
Fibonacci (n=2) at fibonacci.c:7
7         if (n<=1)
```

Podemos ver el estado de la pila junto con el valor de los parámetros en cada llamada con el comando `where`:

```
(gdb) where
#0  Fibonacci (n=2) at fibonacci.c:7
#1  0x0000295c in Fibonacci (n=3) at fibonacci.c:9
#2  0x0000295c in Fibonacci (n=4) at fibonacci.c:9
#3  0x00002a10 in main (argc=2, argv=0xf994) at fibonacci.c:21
```

También podemos retornar de una llamada a una función con el comando `finish`:

```
(gdb) finish
Run till exit from #0  Fibonacci (n=2) at fibonacci.c:7
0x0000295c in Fibonacci (n=3) at fibonacci.c:9
9         return Fibonacci(n-1)+Fibonacci(n-2);
Value returned is $1 = 2
(gdb) where
#0  0x0000295c in Fibonacci (n=3) at fibonacci.c:9
#1  0x0000295c in Fibonacci (n=4) at fibonacci.c:9
#2    0x00002a10  in  main  (argc=2,  argv=0xbffff994)  at
fibonacci.c:21
```

Vemos que hemos retornado de una llamada recursiva. Usando `finish` podemos retornar del resto de llamadas recursivas, pero si se han producido muchas quizá sería mejor idea fijar un breakpoint temporal, que es un breakpoint que una vez que se detiene una vez el depurador en él se borra. En nuestro caso vamos a fijar el breakpoint temporal al acabar la primera llamada a `Fibnonacci()`, y para ello usaremos el comando `tbreak`:

```
(gdb) list
16           printf("Indique un numero como argumento");
17           return 0;
18          }
19        int n;
20        sscanf(argv[1],"%i",&n);
21        int sol = Fibonacci(n);
22        printf("El Fibonacci de %i es %i\n",n,sol);
```

```
23       return 0;
24      }
(gdb) tbreak 22
Breakpoint 2 at 0x2a18: file fibonacci.c, line 22.
```

Para continuar la ejecución no debemos de ejecutar el comando `run`, ya que éste inicia la ejecución del programa desde el principio, sino que debemos usar `continue`:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) continue
Continuing.
Breakpoint 2, main (argc=2, argv=0xbffff994) at fibonacci.c:22
22       printf("El Fibonacci de %i es %i\n",n,sol);
```

Lógicamente, tras alcanzar el breakpoint éste se elimina automáticamente:

```
(gdb) info break
N  Type           Disp En Address
1  breakpoint     keep y  0x00002a04 in main at fibonacci.c:21
   breakpoint already hit 1 time
```

Finalmente el programa parece correctamente depurado:

```
(gdb) print sol
$1 = 5
```

Una facilidad de `gdb` que todavía no hemos comentado es que podemos ejecutar una función de la imagen que estamos depurando usando el comandos `call`:

```
(gdb) call (int) Fibonacci(6)
$2 = 13
```

Para terminar podemos abandonar la ejecución:

```
(gdb) quit
```

El Listado 7.2 muestra como quedaría el programa del ejemplo después de corregir los errores encontrados.

```
/* fibonacci.c */

#include <stdio.h>

int Fibonacci(int n)
{
 if (n<=1)
```

```
    return 1;
 return Fibonacci(n-1)+Fibonacci(n-2);
}

int main(int argc, char* argv[])
{
 if (argc!=2)
    {
     printf("Indique un numero como argumento");
     return 0;
    }
 int n;
 sscanf(argv[1],"%i",&n);
 int sol = Fibonacci(n);
 printf("El Fibonacci de %i es %i\n",n,sol);
 return 0;
}
```

Listado 7.2: Programa a depurar después de corregir los errores encontrados.

1.2.2. Otros comandos de depuración

Podemos usar el comando `help` de `gdb` para obtener ayuda. Si emitimos el comando `help` sin argumentos nos indica como podemos obtener globalmente ayuda:

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping
the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in
that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

Vemos que a `help` le podemos pasar un conjunto de clases en las que se agrupan los comandos por funcionalidad (p.e. la clase `stack` agrupa comandos relacionados con inspeccionar la pila), o bien un nombre de comando, en cuyo caso nos describe su funcionalidad.

Un facilidad bastante usada de `gdb` es que no es necesario escribir el nombre completo del comando mientras que no exista ambigüedad, por ejemplo, en vez de escribir el comando `run`, podemos escribir sólo `r`, o en vez de escribir `info breakpoints` podemos escribir sólo `i b`.

Otra facilidad es que si pulsamos intro sin escribir comando se ejecuta el último comando ejecutado. Lo cual es bastante útil, por ejemplo, cuando estamos ejecutando un programa paso a paso, ya que sólo es necesario escribir una vez el comando `next` (o su abreviatura `n`) para que las demás veces avance paso a paso con sólo volver a pulsar intro.

Actualmente `gdb` tiene cientos de comandos, la Tabla 7.2 resume los principales comandos de `gdb`.

Comando	Descripción
`awatch`	Pone un watchpoint a una variable de forma que siempre que se lea o escriba esta variable se detiene la ejecución. Véase también los comandos `rwatch` y `watch`.
`break`	Fija un breakpoint en el número de línea o función dada.
`clear`	Borra el breakpoint en el número de línea o nombre de función dados.
`continue`	Continua la ejecución después de parar debido a un breakpoint o a un watchpoint.
`Ctrl+C`	Detiene el programa como si hubiera un breakpoint en el punto por el que está ejecutándose.
`disable`	Deshabilita un breakpoint cuyo número se da como argumento.
`display`	Muestra el valor de una expresión cada vez que la ejecución del programa se detiene en un breakpoint o watchpoint.
`enable`	Habilita el breakpoint cuyo número pasamos como argumento.
`finish`	Continua la ejecución del programa que estamos depurando hasta acabar la función en la que nos encontremos.
`ignore`	Permite que un breakpoint se ignore un determinado número de veces. Por ejemplo `ignore 4 23` pide que el breakpoint número 4 sea ignorado 23 veces antes de volver a detenerse en él.
`info breakpoints`	Muestra todos los breakpoints, así como su estado (habilitado/deshabilitado/ignorado).
`info watchpoint`	Muestra los watchpoint, las variables a las que están asignados, y su estado.
`kill`	Mata el proceso actual.
`list`	Muestra 10 líneas de código. Si no se dan más argumentos se muestran 10 líneas cercanas a la posición actual. Si se indica un argumento se muestran líneas cercanas al número dado. Si se indican dos argumentos, separados por coma,

	estamos indicando el número de línea desde el que listar, y el número de líneas a listar.
`next`	Ejecuta el número de líneas indicadas como argumento sin meterse dentro de las funciones. Si no se indica argumento, por defecto se avanza una línea.
`print`	Imprime el valor de una variable o expresión.
`ptype`	Imprime el tipo de una variable.
`return`	Fuerza el retornar inmediatamente de una función sin terminar de ejecutarla.
`run`	Empieza la ejecución del programa desde el principio. Si el programa ya se está ejecutando detiene su ejecución para volverla a comenzar.
`rwatch`	Pone un watchpoint a una variable de forma que el programa sólo se detiene cuando la variable es leída (no cuando es escrita). Véase también los comandos `awatch` y `watch`.
`set`	Permite modificar el valor de una variable durante la ejecución de un programa. Por ejemplo `set sol=12` cambiaría el valor de la variable `sol`.
`step`	Ejecuta el número de líneas indicadas como argumento metiéndose dentro de las funciones que tengan información de depuración. Si no se indica argumento, por defecto se avanza una línea.
`tbreak`	Pone un breakpoint temporal en la línea o nombre de función indicados como argumento.
`watch`	Pone un watchpoint a una variable de forma que sólo se detiene cuando la variable es escrita (no cuando es leída). Véase también los comandos `awatch` y `watch`.
`whatis`	Imprime el tipo y valor de una variable.

Tabla 7.2: Comandos más comunes de `gdb`

1.3. Análisis postmortem de un programa

En los sistemas UNIX, cuando un programa casca se ejecuta una función del sistema operativo que hace un volcado a disco (**dump**) del contenido de la memoria en el momento de fallar el programa, produciendo lo que también se llama un **fichero de core**.

Los ficheros de core son muy útiles cuando creamos aplicaciones beta que entregamos a los usuarios, ya que si la aplicación inexplicablemente falla podemos pedir al usuario que nos reporta el problema, que nos envíe el fichero de core para reconstruir la escena.

Esta opción no siempre está activada por defecto, por ejemplo en Mac OS X por defecto está desactivada. Para comprobar si la opción de dump está activada podemos usar el comando:

```
$ ulimit -c
0
```

Esta opción nos indica el tamaño máximo del fichero de core. Si es 0 significa que está desactivada la opción. Podemos activarla pasando un valor distinto de 0, o el valor `unlimited` para indicar que el fichero de core puede medir todo lo que queramos[1].

```
$ ulimit -c unlimited
$ ulimit -c
unlimited
```

```
/* defectuoso.c */

#include <stdio.h>

char** Nada;

void FuncionDefectuosa()
{
 printf("%n\n",Nada);
}

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

Listado 7.3: Programa defectuoso

Una vez activamos esta opción lo que necesitamos es un programa que falle, como por ejemplo el del Listado 7.3. Cuando ejecutamos este programa obtenemos la salida:

```
$ gcc defectuoso.c -g -o defectuoso
$ ./defectuoso
Bus error
```

Si activamos la opción de generar dump y volvemos a ejecutar el programa:

```
$ ulimit -c unlimited
$ ./defectuoso
Bus error (core dumped)
```

[1] En Mac OS X, con el fin de poder hacer dump de aplicaciones que no sean de consola, también podemos poner la opción `COREDUMPS=-NO-` del fichero `/etc/hostconfig` al valor `COREDUMPS=-YES-`.

Ahora Mac OS X genera un fichero de core en la carpeta `/cores`:

```
$ ls -l /cores
total 96048
-r--------   1 fernando  admin  49176576 Feb 26 22:04 core.645
```

Supongamos que un usuario nos ha reportado este fichero de core, todo lo que tenemos que hacer para depurarlo es habernos acordado de compilar el programa con la opción `-g` con el fin de que tenga símbolos de depuración y podamos cargar ahora el fichero de core en `gdb`. Para ello debemos de pasar a `gdb` tanto el nombre del ejecutable como el del fichero de core. Una rápida inspección con los comandos `where` y `list` de `gdb` nos permite identificar el punto donde la aplicación falló.

```
$ gdb defectuoso /cores/core.645
Core was generated by `./defectuoso'.
#0  0x9000d3e4 in __vfprintf$LDBL128 ()
(gdb) where
#0  0x9000d3e4 in __vfprintf$LDBL128 ()
#1  0x900ff038 in vfprintf_l$LDBL128 ()
#2  0x90101674 in printf$LDBL128 ()
#3  0x00002af4 in FuncionDefectuosa () at defectuoso.c:9
#4  0x00002b20 in main () at defectuoso.c:14
(gdb) list
5       char** Nada;
6
7       void FuncionDefectuosa()
8       {
9        printf("%n\n",Nada);
10      }
11
12      int main()
13      {
14       FuncionDefectuosa();
(gdb)
```

1.4. Enlazar el depurador con una aplicación en ejecución

La herramienta `gdb` dispone de la posibilidad de enlazar con una aplicación en ejecución. Esto puede resultar muy útil para depurar aplicaciones que transcurrido un tiempo fallan inesperadamente, o donde sólo conseguimos reproducir una situación anómala cuando ejecutamos la aplicación sin depuración[1].

[1] Este tipo de problemas se plantean muy a menudo en aplicaciones de tiempo real, donde la decisión de tomar un camino o otro depende del tiempo que tarde la aplicación en ir de un punto del programa a otro.

El Listado 7.4 muestra un programa que sólo se bloquea cuando escribimos el comando `bloquearse`. Para demostrar como `gdb` puede enlazar con este programa vamos a ejecutar primero el programa:

```
$ ./bloqueable
Escriba un comando (bloquearse/salir/otro):bloquearse
```

Y ahora desde otra consola vamos a enlazar con él desde el depurador. Para ello `gdb`, además del nombre del programa, necesita recibir el PID del programa con el que enlazar (que obtendremos con el comando `ps`). A partir de este momento podemos depurar el programa como normalmente, e identificar la causa del bloqueo:

```
$ ps
  PID  TT  STAT      TIME COMMAND
  687  p1  Ss     0:00.32 -bash
  901  p1  R+     1:49.13 ./bloqueable
  907  p2  Ss     0:00.09 -bash
$ gdb bloqueable 901
Attaching to program: 'bloqueable', process 901.
Reading symbols for shared libraries . done
Bloqueate () at bloqueable.c:6
6        while (1);
(gdb)
```

```
/* bloqueable.c */

#include <stdio.h>

void Bloqueate()
{
 while (1);
}

int main()
{
char cmd[255] = "";
 while (strcmp(cmd,"salir"))
       {
        printf("Escriba un comando (bloquearse/salir/otro):");
        scanf("%s",cmd);
        if (!strcmp(cmd,"bloquearse"))
           Bloqueate();
       }
 return 0;
}
```

Listado 7.4: Programa que se mete en un bucle infinito

2. Optimización

Aunque podemos generar código optimizado (opción `-O`) y con depuración (opción `-g`), normalmente el combinar ambas opciones da lugar a un programa que se comporta de forma extraña cuando lo depuramos debido a las optimizaciones. En general la opción de depuración `-g` se debería usar sólo mientras estemos implementando el programa, mientras que la opción de optimización `-O` se aconseja usar sólo cuando hayamos acabado el programa y queramos distribuirlo.

2.1. Opciones de optimización

En este apartado vamos a empezar comentando cuales son las opciones de `gcc` que activan los diferentes niveles de optimización. En los siguientes apartados describiremos con más detalle dos de estas técnicas: Scheduling y deshacer bucles.

El comando `gcc` proporciona cuatro niveles de optimización (del 0 al 3) que permiten indicar aspectos tales como las preferencias entre un código más optimizado a cambio de mayor tiempo de compilación, o el balanceo entre velocidad de ejecución y tamaño de programa[1]. Además, veremos que existen otras opciones que nos permiten controlar otros aspectos de la optimización.

Las opciones dedicadas a la optimización del comando `gcc` son las siguientes:

`-O0`

Esta es la opción por defecto cuando no se indica ninguna opción de optimización, y lo que hace la opción es indicar que no queremos optimización. Además ésta es la opción que se recomienda cuando vamos a depurar un programa, ya que se evitan efectos extraños en la depuración al ir avanzando por el programa paso a paso.

`-O1` (ó `-O`)

Esta es la forma más común de optimización, y también se puede activar con `-O` sin indicar nivel de optimización. La opción activa todas las optimizaciones excepto el scheduling, y las optimizaciones que requieran aumentar el tamaño del programa generado. Con esta opción normalmente el programa copilado es menor al generado con `-O0`, e incluso el programa puede compilar más rápido debido a que esta opción hace que el backend del compilador necesite procesar menos datos.

[1] Debido a que es muy común que un codigo más rápido también necesite consumir más memoria RAM (pero no necesariamente ocupar más espacio su imagen en disco).

`-O2`

Esta opción además de activar las optimizaciones de `-O1` activa la optimización por scheduling, aunque sigue sin realizar optimizaciones que aumenten el tamaño del ejecutable. El introducir la optimización por scheduling hace que el tiempo necesario para generar el ejecutable aumente considerablemente. En general, esta es la mejor opción para las versiones release, ya que proporciona máxima optimización sin aumentar el tamaño del ejecutable. De hecho, este es el nivel de optimización que suelen usar los paquetes GNU.

`-O3`

Esta opción activa optimizaciones que pueden aumentar el tamaño del ejecutable generado (p.e. expandir funciones `inline`). Normalmente estas optimizaciones aumentan la velocidad de ejecución si se dispone de memoria RAM suficiente, pero también pueden hacer al programa más lento. En general esta optimización se recomienda sólo para funciones muy frecuentemente usadas.

`-Os`

Esta opción activa todas las opciones de `-O2` más aquellas tendientes a reducir el tamaño del ejecutable (aunque el tiempo de ejecución pudiera amentar). El uso de esta opción puede hacer que se produzca código más lento, pero también puede resultar más rápido debido a que carga más rápido y se utiliza mejor la cache.

`-funroll-loops`

Esta opción activa el loop unrolling (deshacer bucles) con lo que produce ejecutables más grandes. En general está opción no es recomendable, sino que su uso dependerá de cada caso.

2.2. Scheduling

La **segmentación (pipelines)** es un técnica usada por los microprocesadores más avanzados (la mayoría de los procesadores actuales: Pentium, SPARC, PowerPC, Alpha, ...) por la cual se solapa la ejecución de varias instrucciones contiguas en el programa. Para poder ejecutar varias instrucciones en paralelo es necesario que las instrucciones máquina del programa estén colocadas de forma que una instrucción no tenga que esperar a que acaben las anteriores, para lo cual muchas veces es necesario que el compilador reordene las instrucciones maquinas convenientemente. El **scheduling** es la técnica usada para reordenar estas instrucciones por parte del compilador.

2.3. Deshacer bucles

Cuando un bucle es corto y y tiene un número de iteraciones fijas, como por ejemplo:

```
for (int i=0;i<8;i++)
    y[i] = i;
```

El bucle gasta más tiempo en comprobar la condición de repetición y en actualizar el contador que en realizar la operación del cuerpo del bucle.

En situaciones como ésta conviene, desde el punto de vista de la optimización, deshacer el bucle para obtener:

```
y[0] = 0;
y[1] = 1;
y[2] = 2;
y[3] = 3;
y[4] = 4;
y[5] = 5;
y[6] = 6;
y[7] = 7;
```

3. Control de corrupción y pérdida de memoria

Este apartado explica cómo podemos detectar los problemas de corrupción y pérdida de memoria usando una serie de herramientas que existen en OS X para tal fin.

3.1. Corrupción de memoria

La corrupción de memoria ocurre cuando el programa escribe datos en una zona de memoria distinta a la esperada. En este caso lo mejor que puede pasar es que el sistema operativo lo detecte y el programa sea terminado. Si el sistema operativo no lo detecta, este cambio acabará afectando a otro trozo de programa que tarde o temprano fallará.
El error de corrupción de memoria más común en C es el del **desbordamiento de buffer**, el cual ocurre cuando un programa escribe más allá del trozo de memoria que teníamos reservada para él. Por ejemplo:

```
char* mi_copia = malloc (strlen(cadena));
strcpy (mi_copia, cadena);
```

Aquí estamos escribiendo un byte más allá del bloque de memoria reservado, para corregirlo deberíamos de haber reservado un byte más para que pudiéramos almacenar el carácter de final de línea:

```
char* mi_copia = malloc (strlen(cadena)+1);
strcpy (mi_copia, cadena);
```

Este tipo de errores también es muy típico que se produzca por bucles que dan una vuelta más de las que deberían:

```
char digitos[10];
bool encontrado=false;
for (int i=0;i<=10;i++)
    digitos[i] = '0'+i;
```

Obsérvese que el bucle se repite 11 veces, y no 10. Lo peor de todo es que el bucle no falla, pero modificará la variable `encontrado` haciendo que luego el programa no se comporte como esperábamos.

Un desbordamiento en memoria dinámica reservada con `malloc()` puede provocar que el fallo se detecte mucho más tarde, ya que `malloc()` almacena información de contabilidad de los bloques asignados justo delante del puntero que nos devuelve, con lo que si sobrescribimos esta memoria el problema se producirá mucho más tarde, cuando ejecutemos los `free()` de

esos bloques. Estos problemas pueden resultar extremadamente difíciles de detectar.

Otro efecto lateral del desbordamiento de búferes es que podemos enviar mucha información a un programa, que no controla el tamaño de sus búferes, para que se desborden, y escribir así en la pila del programa, de forma que consigamos retornar a una determinada dirección de memoria. Esta técnica la han utilizado en muchas ocasiones los hackers para burlar la seguridad de un sistema.

Otra forma de corrupción de memoria es la conocida como **puntero loco**, la cual se produce cuando la dirección de memoria a la que apunta un puntero no tiene relación con la dirección de memoria a la que realmente debería de apuntar. Un ejemplo se produce cuando a un trozo de memoria lo apuntamos con dos punteros y no actualizamos uno de ellos. Por ejemplo:

```
char* nombre_usuario;
const char* getNombreUsuario()
{
 return nombre_usuario;
}

void setNombreUsuario(const char* nombre)
{
 free(nombre_usuario);
 nombre_usuario = strdup(nombre); // Realiza un malloc()
}
```

Ahora considérese el siguiente escenario:

```
nombre = getNombreUsuario();
setNombreUsuario("Luis");
printf(nombre); //Aquí se está usando un puntero loco
```

3.2. Pérdida de memoria

La pérdida de memoria se produce cuando un programador hace un programa que reserva memoria, con `malloc()`, y olvida liberar esta memoria, con `free()`, cuando el programa ya no la va a usar más.

Este despiste normalmente no da problemas cuando el programa que vamos a ejecutar tiene una vida corta, ya que el sistema operativo libera toda la memoria reservada por un programa una vez que este termina (o falla). En programas que pueden permanecer ejecutando durante semanas, meses o

años (p.e. servidores) el problema se vuelve acumulativo, ya que cada vez que el programa pasa por un punto hace una reserva que nunca liberará[1].

3.3. Las malloc tools

Para detectar estos problemas, y otros muchos problemas de memoria, las librerías de reserva de memoria de Mac OS X pueden ayudarnos como vamos a ver a continuación.

Aunque a estas librerías se las llama las malloc tools, en referencia a que es la función `malloc()` la que proporciona estas ayudas de depuración, en realizad se pueden usar con todos los sistemas de reserva de memoria de Mac OS X, como por ejemplo el operador `new` de C++.

Para que estas librerías nos ayuden lo único que tenemos que hacer es fijar determinadas variables de entorno antes de ejecutar la aplicación. Estas variables no es necesario fijarlas antes de compilar ya que es el propio runtime de la función `malloc()`, y no el compilador el que comprueba si estas variables de entorno están fijadas, y de hecho podemos depurar cualquier comando o programa de Mac OS X fijando estas variables.

Variable entorno	Descripción
`MallocHelp`	Muestra ayuda sobre como funcionan las malloc tools.
`MallocGuardEdges`	Añade dos páginas de guarda a los lados de cada bloque.
`MallocDoNotProtectPrelude`	Quita la página de guarda anterior a los bloques (sólo es útil cuando usamos `MallocGuardEdges`).
`MallocDoNotProtectPostlude`	Quita la página de guarda posterior a los bloques (sólo es útil cuando usamos `MallocGuardEdges`).
`MallocScribble`	Ayuda a detectar lecturas de bloques liberados escribiendo 0x55 en todos los bytes al liberarlos.
`MallocStackLogging`	Almacenan información de llamada a rutinas para el comando `malloc_history`.
`MallocStackLoggingNoCompact`	Almacenan información ampliada de llamada a rutinas para el comando `malloc_history`.

Tabla 7.3: Variables de entorno de las malloc tools

[1] Esto explica porque Windows NT 4.0 sufría un proceso degenerativo tan rápido que le hacía ejecutar cada vez más lento y que transcurrida una semana había que reiniciarlo para que volviera a funcionar a un ritmo normal. Los servicios de Windows NT 4.0 padecían de este mal, y poco a poco se iban comiendo toda la memoria.

En la Tabla 7.3 se resumen cuales son las variables de entorno que usan las malloc tools. Vamos a comentar más detalladamente como funcionan estas variables de entorno.

3.3.1. Obtener ayuda

Podemos usar la variable de entorno `MallocHelp` para que las malloc tools nos muestren un mensaje de ayuda cada vez que vayamos a ejecutar un programa indicando que variables de entorno reconoce.

Para ver como funciona simplemente fije esta variable de entorno a cualquier valor. Por ejemplo, en bash haríamos:

```
$ export MallocHelp=si
```

Cualquier programa que ejecute ahora mostrará ayuda sobre las malloc tools:

```
$ ls
malloc[513]: environment variables that can be set for debug:
- MallocGuardEdges to add 2 guard pages for each large block
- MallocDoNotProtectPrelude to disable protection (when
previous flag set)
- MallocDoNotProtectPostlude to disable protection (when
previous flag set)
- MallocStackLogging to record all stacks. Tools like leaks
can then be applied
- MallocStackLoggingNoCompact to record all stacks. Needed for
malloc_history
- MallocScribble to detect writing on free blocks: 0x55 is
written upon free
- MallocCheckHeapStart <n> to check the heap from time to time
after <n> operations
- MallocHelp - this help!
Compiladores IC Music Public Desktop Library bin Documents
Logica Pictures Sites tmp
```

Para poder depurar las aplicaciones gráficas con las malloc tools, éstas deberán de poder leer estas variables de entorno, para lo cual deberá ejecutarlas desde la consola, por ejemplo así:

```
$ open /Applications/Mozilla.app
```

Como ve, esta variable de entorno sólo sirve para dar ayuda, vamos a comentar otras variables de entorno más interesantes.

3.3.2. Detectar accesos fuera de la memoria reservada

La variable de entorno `MallocGuardEdges` pone una página de memoria de 4KB sin permisos de acceso justo delante y detrás de cada bloque asignado. Esto permite capturar desbordamientos de buffer. La documentación indica que esto sólo se hace cuando se trata de bloques "grandes". Aunque no se especifica el valor de "grande", experimentalmente se puede comprobar que "grande" es cuando el bloque de memoria supera los 12KB.

El Listado 7.5 muestra un programa Objective-C, al que hemos llamado `mallocguard.m` que prueba esta opción.

```
/* mallocguard.m */

int main()
{
 char* memoria = (char*) malloc(1024*16);
 // Escribe fuera de la memoria reservada
 memoria[(1024*16)+1] = 'x';
 return 0;
}
```

Listado 7.5: Programa que escribe fuera de la memoria reservada

Una vez compilado:

```
$ gcc mallocguard.m -o mallocguard
```

Si ejecutamos el programa sin fijar `MallocGuardEdges` obtenemos:

```
$ ./mallocguard
```

Pero si fijamos esta variable de entorno:

```
$ export MallocGuardEdges=1
$ ./mallocguard
malloc[548]: protecting edges
Bus error
```

Nos detecta que nuestro programa escribe fuera del bloque permitido produciéndose un fallo de página.

Para acabar comentaremos que podemos quitar las páginas de guarda anterior o posterior al bloque de memoria reservado usando las variables de entorno `MallocDoNotProtectPrelude` y `MallocDoNotProtectPostlude`.

3.3.3. Detectar accesos a memoria liberada

La variable de entorno `MallocScribble` hace que cuando se libere un bloque de memoria se escriba sobre todos sus bytes el valor 0x55, lo cual sirve para capturar intentos de utilizar ese bloque después de haber sido liberado. Obsérvese que 0x55 se ha elegido porque es un valor impar y cualquier intento de dereferenciar un puntero que apunte a un valor impar produce una excepción (odd address), con lo que si en la estructura de memoria liberada hubiera alguna variable de tipo puntero que luego quisiéramos utilizar se va a producir esta excepción. Por desgracia, experimentalmente hemos comprobado que `free()` siempre deja los 8 primeros bytes a 0, lo cual puede dificultar la captura de errores.

El Listado 7.6 muestra un ejemplo llamado `mallocscribble.m` que usa esta opción.

```
/* mallocscribble.m */

#import <stdlib.h>
typedef struct
{
 char nombre[20];
 char apellidos[30];
} Persona;

int main()
{
 Persona* p = malloc(sizeof(Persona));
 strcpy(p->nombre,"Fernando");
 strcpy(p->apellidos,"Lopez");
 printf("Nombre:%s Apellidos:%s\n",p->nombre,p->apellidos);
 free(p);
 printf("Nombre:%s Apellidos:%s\n",p->nombre,p->apellidos);
 return 0;
}
```

Listado 7.6: Programa que detecta accesos a memoria liberada

Al ejecutarlo obtenemos:

```
$ ./mallocscribble
Nombre:Fernando Apellidos:Lopez
Nombre: Apellidos:Lopez
```

Obsérvese que `free()` ha sobrescrito los 8 primeros bytes con ceros, por eso el nombre no aparece.

Si ahora activamos la opción `MallocScribble` de las malloc tools:

```
$ export MallocScribble=1
```

```
$ ./mallocscribble
malloc[344]: enabling scribbling to detect mods to free blocks
Nombre:Fernando Apellidos:Lopez
Nombre: Apellidos:UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
```

Vemos que aunque `free()` sigue dejando los primeros 8 bytes a 0, todos los demás quedan con un 0x55, con lo que nos ayuda a identificar que estamos leyendo de memoria liberada.

3.3.4. Detectar la corrupción de la memoria dinámica

Las variables de entorno `MallocCheckHeapStart` y `MallocCheckHeapEach` nos permiten chequear una posible corrupción de las estructuras internas de `malloc()` (debido a que el programa ha escrito donde no debía), ejecutando rutinas de chequeo del heap periódicamente.

En `MallocCheckHeapStart` indicamos el número de reservas de memoria necesarias antes de que se empiece a chequear el heap, y en `MallocCheckHeapEach` indicamos los intervalos tras los que realizar los chequeos.

A medida que el programa se ejecuta se van imprimiendo mensajes como el siguiente, indicando que el chequeo a sido exitoso:

```
MallocCheckHeap: PASSED check at 37800th operation
```

Cuando se nos avisa de que las estructuras de memoria de `malloc()` han sido alteradas podemos ajustar las variables `MallocCheckHeapStart` y `MallocCheckHeapEach` para ir buscando en que zona ocurre la modificación.

3.4. El comando `leaks`

`leaks` es un comando que examina a un proceso en ejecución para detectar búferes asignados por `malloc()` que no están referenciados por ningún puntero.

Para poder comprobar las perdidas de memoria de un proceso, éste debe de estar en ejecución, con lo que si es necesario podemos poner un bucle infinito al final del programa:

```
main()
{
 ..............
 ..............
 while(1);
 return 0;
}
```

Por ejemplo, imaginemos que tenemos el programa del Listado 7.7:

```
/* pierde.c */

#include <stdio.h>

main()
{
 char* bloque1 = (char*)malloc(1024);
 printf("Perder memoria (s/n)?");
 char c;
 while( (c=getchar()) == 's' )
      {
       bloque1= (char*)malloc(1024);
       getchar(); // Quita el '\n'
       printf("Perder memoria (s/n)?");
      }
 return 0;
}
```

Listado 7.7: Programa con fugas de memoria

Este programa pierde memoria cada vez que asignamos un nuevo valor a `bloque1`.

Para comprobar como el comando `leaks` detecta las pérdida de memoria, primero debemos de ejecutar este programa en una consola. Suponiendo que el programa esté en un fichero llamado `pierde.c` haríamos:

```
$ gcc pierde.c -o pierde
$ pierde
Perder memoria (s/n)?
```

Ahora desde otra consola miramos que número de proceso tiene el comando y usamos `leaks` para ver si ha perdido memoria:

```
$ ps
PID TT STAT TIME COMMAND
631 p1 Ss 0:00.08 -bash (bash)
721 p1 S+ 0:00.01 pierde
725 std Ss 0:00.03 -bash (bash)
$ leaks -cycles 721
Locating all 11 malloced pointers in use ...
Computing connected groups ...
Computing non-malloced pages ...
Enumerating all pages non-malloced ...
Doing the transitive closure of all reachable groups ...
Gathering leaks ...
==== 0 simple leaks
==== 0 non circular group leaks
==== 0 cyclic leaks
```

Vemos que inicialmente no hay ninguna pérdida de memoria detectada, pero si ahora hacemos que `pierde` pierda memoria:

```
Perder memoria (s/n)?s
Perder memoria (s/n)?
```

Y volvemos a ejecutar `leaks`:

```
$ leaks -cycles 721
Locating all 12 malloced pointers in use ...
Computing connected groups ...
Computing non-malloced pages ...
Enumerating all pages non-malloced ...
Doing the transitive closure of all reachable groups ...
Gathering leaks ...
==== 1 simple leaks
Leak: 0x000442b0 size=1038
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
0x00000000 0x00000000 0x00000000 0x00000000
...
==== 0 non circular group leaks
==== 0 cyclic leaks
```

Vemos que acaba de detectar la pérdida de memoria.

3.5. El comando `malloc_history`

`malloc_history` nos permite inspeccionar un proceso en ejecución y ver las asignaciones de memoria que ha hecho. Para que este comando funcione necesitamos exportar la variable de entorno `MallocStackLogging`. Por ejemplo, podemos depurar nuestro anterior programa así:

```
$ export MallocStackLogging=1
$ pierde
malloc[768]: recording stacks using standard recorder
Perder memoria (s/n)?s
Perder memoria (s/n)?
```

Ahora desde otra consola ejecutamos:

```
$ malloc_history 768 -all_by_size
1 calls for 131072 bytes: thread_800013b8 |0x1000 | start |
_start | main | __srget | __srefill | __smakebuf | malloc |
malloc_zone_malloc
```

```
1 calls for 131072 bytes: thread_800013b8 |0x1000 | start |
_start | main | printf | vfprintf | __swsetup | __smakebuf |
malloc | malloc_zone_malloc
1 calls for 1024 bytes: thread_800013b8 |0x1000 | start |
_start | main | malloc | malloc_zone_malloc
1 calls for 1024 bytes: thread_800013b8 |0x1000 | start |
_start | main | malloc | malloc_zone_malloc
1 calls for 72 bytes: thread_800013b8 |0x1000 | start | _start
| __keymgr_dwarf2_register_sections |
_dyld_register_func_for_add_image |
register_func_for_add_image |
dwarf2_unwind_dyld_add_image_hook |
_keymgr_get_and_lock_processwide_ptr |
_keymgr_get_and_lock_key | _keymgr_get_or_create_key_element |
_keymgr_create_key_element | malloc | malloc_zone_malloc
1 calls for 36 bytes: thread_800013b8 |0x1000 | start | _start
| __keymgr_dwarf2_register_sections |
_dyld_register_func_for_add_image |
register_func_for_add_image |
dwarf2_unwind_dyld_add_image_hook | calloc |
malloc_zone_calloc
1 calls for 18 bytes: thread_800013b8 |0x1000 | start | _start
| __keymgr_dwarf2_register_sections |
_dyld_register_func_for_add_image |
register_func_for_add_image |
dwarf2_unwind_dyld_add_image_hook | calloc |
_malloc_initialize | malloc_set_zone_name | malloc_zone_malloc
1 calls for 8 bytes: thread_800013b8 |0x1000 | start | _start
| __keymgr_dwarf2_register_sections |
_dyld_register_func_for_add_image |
register_func_for_add_image |
dwarf2_unwind_dyld_add_image_hook |
_keymgr_get_and_lock_processwide_ptr | _init_keymgr | malloc |
malloc_zone_malloc
```

Aquí aparecen todas las llamadas a `malloc()` que ha hecho nuestro proceso (directamente o a través de subrutinas).

Una utilidad de este comando es detectar accesos a bloques de memoria liberada, combinándolo con la variable de entorno `MallocScribling`, la cual escribía 0x55 en los búferes liberados. Si hemos intentado escribir en un buffer liberado, `malloc_history` mostrará un mensaje de advertencia avisándonos de tal hecho.

3.6. MallocDebug

Mac OS X dispone de una herramienta llamada `MallocDebug.app` que podemos encontrar en el directorio `/Developer/Applications` que nos permite depurar la memoria dinámica reservada por `malloc()` de forma gráfica. Recomendamos al lector que la eche un vistazo.

4. Perfilado

De todos es conocida la vieja frase: "Un programa pasa el 90% de su tiempo ejecutando el 10% del código". El objetivo del **perfilado (profiling)** es identificar este 10% del programa y optimizarlo.

4.1. Usar el profiler `gprof`

El perfilador de GNU lo implementa el comando `gprof`, un comando que resulta útil a la hora de medir el rendimiento de un programa, para ello este programa registra el número de llamadas que se producen a cada función del programa, y el tiempo gastado en cada una de ellas. Esto nos ayuda a identificar las funciones que más tiempo gastan. Si conseguimos optimizar las funciones que dominan el tiempo de ejecución, estaremos optimizando de forma más inteligente nuestro programa.

Collatz propuso la conjetura (actualmente sin demostración conocida) de que cualquier número entero positivo x_n operado repetidamente por dos operaciones tiende a 1. Estas dos operaciones son: Si x_n es impar, se convierte en $3x_n+1$. Si x_n es par se convierte en $x_n/2$, es decir[1]:

$$x_{n+1} = \begin{cases} \text{Si } x_n \text{ es impar} \text{ ---> } 3x_n+1 \\ \text{Si } x_n \text{ es impar} \text{ ---> } x_n/2 \end{cases}$$

El Listado 7.8 muestra un programa que comprueba la conjetura de Collatz con los 5.000.000 de primeros números naturales. Para ello llama a la función `convergencia()` que calcula el número de pasos necesarios para que convenga cada número. La función `convergencia()` a su vez llama a la función `siguiente()` que devuelve el siguiente número de la serie de Collatz.

Para usar el perfilador debemos de crear un **ejecutable instrumentalizado para perfilación**, el cual contiene instrucciones adicionales para registrar las funciones llamadas y el tiempo dedicado a cada una. Para crear un ejecutable instrumentalizado para perfilación debemos compilar y enlazar el programa con la opción `-pg`:

```
$ gcc -Wall -c -pg collatz.c
$ gcc -Wall -pg collatz.o -o collatz
```

Ahora ejecutaríamos el programa como normalmente:

```
$ ./collatz
```

[1] Observe que x_n=1 es condición de parada y si x_n=2, x_{n+1}=1, la curiosidad está en que salen más pares que impares, ya que sino $x_{n+1} = 3x_n+1$ haría que la serie tendiera a infinito.

```
#include <stdlib.h>

unsigned int siguiente(unsigned int n)
{
 if (n%2==1)
    n = 3*n+1;
 else
    n= n/2;
 return n;
}

unsigned int convergencia(unsigned int n)
{
 unsigned int pasos=0;
 while (n!=1)
       {
        if (n>500000000)
           {
            fprintf(stderr,"Error %u no converge\n",n);
            exit(1);
           }
        n = siguiente(n);
        pasos++;
       }
 return pasos;
}

int main()
{
 // Estudia la convergencia de los numeros de 1 a 5000000
 unsigned int i;
 for (i=1;i<5000000;i++)
     {
      unsigned int pasos = convergencia(i);
      printf("%u converge a 1 en %u pasos\n",i,pasos);
     }

 return 0;
}
```

Listado 7.8: Programa que comprueba la conjetura de Collatz

El programa instrumentalizado según se ejecuta va silenciosamente creando en el directorio actual un fichero llamado `gmon.out`, con información sobre las funciones ejecutadas y su tiempo de ejecución. Podemos analizar el contenido de este fichero con el comando `gprof`, al que le debemos dar como argumento el nombre del programa que se estaba perfilando:

```
$ gprof collatz
       cumul   self              self    total
%time   secs   secs     calls  ms/call  ms/call  name
68.59   2.14   2.14 740343580     0.03     0.03  _siguiente
31.09   3.11   0.97   4999999     1.94     6.22  _convergencia
 0.32   3.12   0.01         1                    _main
```

Obsérvese que en principio podríamos pensar que la función que más tiempo está ejecutando es `main()` porque desde que empieza la ejecución del programa hasta que acaba siempre estamos dentro de ella, pero esto no es cierto, y de hecho aunque el campo cumulative secs así lo muestra, el campo self secs y % time muestran sólo el tiempo pasado dentro de cada función.

4.2. Test de cobertura con `gcov`

Los **test de cobertura** nos dicen cuantas veces ha sido ejecutada cada línea del programa durante su ejecución. Normalmente durante la etapa de pruebas de un software es recomendable comprobar que todas las líneas del programa son ejecutadas al menos una vez. Si se detecta que una línea nunca se ejecuta, esta línea puede ser eliminada del programa. Además, el conocer las líneas más ejecutadas es útil para poder perfilar el programa.

Las GCC proporcionan el comando `gcov` para poder realizar test de cobertura. Suponiendo que partimos de nuevo del programa del Listado 7.8, para realizar un test de cobertura podemos compilar el programa de la forma:

```
$ gcc -fprofile-arcs -ftest-coverage collatz.c -o collatz
```

Esto hace que el ejecutable se instrumentalice para registrar el número de veces que es ejecutada cada función. En concreto la opción `-ftest-coverage` hace que se cuente el número de veces que se ejecuta cada línea, mientras que la opción `-fprofile-arcs` hace que se guarde información sobre la frecuencia con la que se toma cada rama de una instrucción de bifurcación.

Ahora debemos ejecutar el programa instrumentalizado para que se cree esta información de instrumentalización:

```
$ ./collatz > /dev/null
```

Esto hace que se generen varios ficheros en el directorio actual con información del test de cobertura. Esta información puede ser analizada por `gcov`, al que además debemos de pasar los nombres de los ficheros de código fuente:

```
$ gcov collatz.c
File 'collatz.c'
Lines executed:89.47% of 19
collatz.c:creating 'collatz.c.gcov'
```

El comando nos informa de que se ha creado el fichero `collatz.c.gcov` con esta información:

```
$ cat collatz.c.gcov
        -:    0:Source:collatz.c
        -:    0:Graph:collatz.gcno
        -:    0:Data:collatz.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:/* collatz.c */
        -:    2:
        -:    3:#include <stdio.h>
        -:    4:#include <stdlib.h>
        -:    5:
        -:    6:unsigned int siguiente(unsigned int n)
  5024987:    7:{
  5024987:    8: if (n%2==1)
  1665647:    9:    n = 3*n+1;
        -:   10: else
  3359340:   11:    n= n/2;
  5024987:   12: return n;
        -:   13:}
        -:   14:
        -:   15:unsigned int convergencia(unsigned int n)
    49999:   16:{
    49999:   17: unsigned int pasos=0;
  5124985:   18: while (n!=1)
        -:   19:       {
  5024987:   20:        if (n>500000000)
        -:   21:           {
    #####:   22:            fprintf(stderr,"Error %u no con...
    #####:   23:            exit(1);
        -:   24:           }
  5024987:   25:        n = siguiente(n);
  5024987:   26:        pasos++;
        -:   27:       }
    49999:   28: return pasos;
        -:   29:}
        -:   30:
        -:   31:int main()
        1:   32:{
        -:   33: // Estudia la convergencia de los numeros ...
        -:   34: unsigned int i;
    50000:   35: for (i=1;i<50000;i++)
        -:   36:     {
    49999:   37:      unsigned int pasos = convergencia(i);
    49999:   38:      printf("%u converge a 1 en %u pasos\n...
        -:   39:     }
        -:   40:
        1:   41: return 0;
        -:   42:}
```

Observe que el comando `gcov` nos informó de que sólo se han ejecutado el 89.47% de las líneas. Además nos marca con `#####` las líneas que nunca se han ejecutado.