Objective-C

Curso práctico para programadores Mac OS X, iPhone y iPad

2ª edición actualizada a Mac OS X 10.8 y iOS 6

Fernando López Hernández

Objective-C. Curso práctico para programadores Mac OS X, iPhone y iPad, 2ª edición
Fernando López Hernández

ISBN: 978-84-939450-8-4
EAN: 9788493945084
BIC: UMQ; UMS

RC Libros
Calle Mar Mediterráneo, 2. Nave 6
28830 SAN FERNANDO DE HENARES, Madrid
Teléfono: +34 91 677 57 22
Fax: +34 91 677 57 22
Correo electrónico: info@rclibros.es
Internet: www.rclibros.es

Diseño de colección, cubierta y pre-impresión: Grupo RC
Impresión y encuadernación: Service Point
Depósito Legal: M-4798-2013
Impreso en España

17 16 15 14 13 (1 2 3 4 5 6 7 8 9 10 11 12)

PARTE I:

EL LENGUAJE OBJECTIVE-C

EMPEZANDO A PROGRAMAR CON OBJECTIVE-C

1

OBJETIVOS DEL CAPÍTULO

Este primer capítulo de naturaleza introductoria empieza describiendo los entornos donde podemos programar con Objective-C, y después describe las herramientas necesarias para crear aplicaciones.

Antes de que en el Capítulo 2 se adentre en los conceptos del lenguaje, creemos que será útil para el lector familiarizarse con el uso de las herramientas de programación. Con este fin, este capítulo empieza enseñando al lector cómo crear sencillos programas, y describe la forma de compilar y enlazar estas aplicaciones.

El capítulo empieza describiendo cómo compilar aplicaciones Objective-C desde la consola, utilizando Clang y las herramientas de programación de GNU. En la segunda parte de este capítulo se estudia cómo, usando la herramienta gráfica Xcode, podemos mejorar la productividad del programador.

ENTORNOS DE PROGRAMACIÓN

Objective-C procede de los tiempos de NeXTSTEP, el sistema operativo en el que se basa Mac OS X. A su vez iOS es una evolución de Mac OS X para adaptarlo a dispositivos móviles. iOS es el nombre que Apple ha dado al sistema operativo de dispositivos como iPhone, iPad, iPod Touch y Apple TV. Objective-C es el lenguaje principal y nativo para desarrollo de aplicaciones en todos estos sistemas operativos.

Una vez que el programador de aplicaciones Mac OS X o iOS aprende el lenguaje Objective-C, su siguiente paso será empezar a conocer la extensa librería de clases y funciones que proporcionan estos sistemas operativos. Estas librerías se pueden dividir, grosso modo, en tres grandes grupos:

- **Foundation Framework**. Son un conjunto de clases de utilidad que permiten representar estructuras de datos complejas (arrays, listas, diccionarios, etc.). Estas clases también incluyen otras funcionalidades como el acceso a red, la gestión de procesos e hilos, el runtime de configuración del sistema, la programación multihilo y las técnicas para sincronizar estos hilos. La principal característica de estas clases es que, en su mayoría, son comunes a Mac OS X y iOS. Esto se debe a que no incluyen las clases relacionadas con la parte que más cambia: la interfaz gráfica de usuario.
- **Cocoa**. Aquí se incluyen las clases propias de Mac OS X. Estas clases están relacionadas con la interfaz gráfica, la impresión, el acceso a audio y vídeo, y todos los demás servicios que proporciona Mac OS X. Tradicionalmente, Apple también ha utilizado el término **Aplicación Kit Framework** (AppKit Framework) para referirse a estas clases. Dentro del Cocoa encontramos otros kits de desarrollo especializados en determinadas tareas como puedan ser Image Kit, QuickTime Kit, Apple Scripting Kit, etc.
- **Cocoa Touch**. Aquí se incluyen las clases propias de la interfaz de usuario de iOS. Las clases de Cocoa están pensadas para manejar aplicaciones mediante un teclado y un ratón, con lo que Apple tuvo que crear un conjunto de librerías distintas, llamado Cocoa Touch, para poder manejar dispositivos táctiles. La principal librería que encontramos en Cocoa Touch es UI Kit, el kit que permite desarrollar los elementos de la interfaz gráfica de las aplicaciones iOS. Otros kits que encontramos son, por ejemplo, Map Kit para gestionar mapas Google, iAd Kit para gestionar los banners de publicidad, Game Kit para juegos o Audio Unit Kit para gestionar el sonido.

La Figura 1.1 muestra un resumen de las capas software de programación tanto de Mac OS X como de iOS. Observe que las partes de más alto nivel son programables en Objective-C, mientras que el bajo nivel (**Core Services**) se programa en C. Esta organización permite alcanzar mayor productividad a los programadores de alto nivel mediante el uso de la programación orientada a objetos. Como se explicará en el Capítulo 2, Objective-C es una extensión de C para aprovechar las ventajas de la programación orientada a objetos. Observe que hay librerías Objective-C (por ejemplo, Quick Time Kit o Foundation Framework) que tienen su correspondiente librería C (QuickTime y Core Foundation, en el ejemplo anterior). En este caso, las llamadas a las librerías Objective-C suelen estar mapeadas directamente; es decir, envuelven a su correspondiente llamada a librería C.

Otra API no mostrada en esa figura es **Carbon**, la antigua API de programación de Mac OS Classic (Mac OS 9 e inferiores). Carbon es un API programable en C y de transición, que Mac OS X integra para mantener compatibilidad con aplicaciones

antiguas. El fin último de Carbon es desaparecer, y de hecho, iOS ya no incluye esta API.

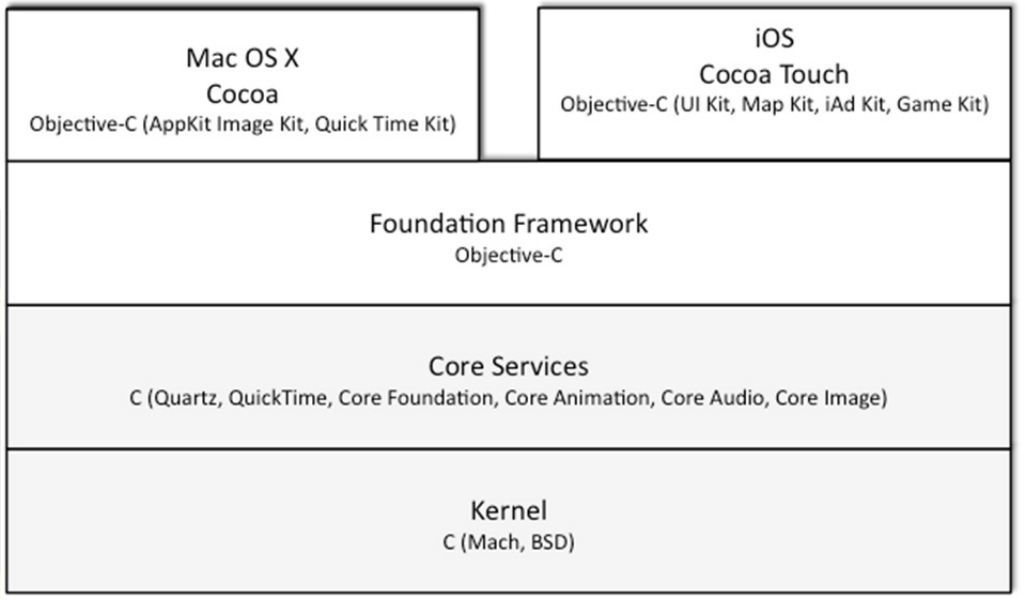

Figura 1.1: Capas software de programación en Mac OS X y iOS

En este libro vamos a centrarnos en estudiar el lenguaje Objective-C. También estudiaremos algunas clases de Foundation Framework. Las clases de Cocoa y Cocoa Touch se dejan para otros libros que usted podrá leer, una vez que domine el lenguaje.

COMPILANDO CON LAS GCC

Las **GCC** (GNU Compiler Collection) son un conjunto de herramientas que proporciona GNU para programar en varios lenguajes y plataformas. Mac OS X y iOS se apoyan en estas herramientas para realizar las tareas de compilación y enlazado de sus aplicaciones.

Para mejorar la productividad y la facilidad de uso de estas herramientas, Apple creó Xcode, un IDE (Integrated Development Environment) que permite realizar de forma gráfica las tareas comunes de programación. Xcode hace llamadas a las GCC para realizar las tareas de compilación y enlazado de forma transparente al programador.

En esta sección veremos cómo utilizar las herramientas de programación de GNU para compilar y enlazar aplicaciones Objective-C desde el terminal[1]. Al final del capítulo veremos cómo se puede utilizar Xcode para mejorar la productividad del programador.

En este capítulo vamos a pasar por encima los detalles relativos al lenguaje, con lo que si no conoce Objective-C puede que muchos aspectos de lenguaje le resulten confusos. En el Capítulo 2 introduciremos el lenguaje a nivel conceptual, y en el Capítulo 3 empezaremos a detallar todos los aspectos del lenguaje con el nivel de detalle que se requiere para su correcta comprensión.

Crear un ejecutable

Debido a que Objective-C es una extensión al lenguaje C, un programa C compila en Objective-C sin necesidad de cambios. El Listado 1.1 muestra un programa básico Objective-C que solo usa la sintaxis de C, excepto por dos aspectos:

```
/* holamundo.m */
#import <stdio.h>
#import <stdlib.h>

int main() {
    printf("Hola desde Objective-C\n");
    return EXIT_SUCCESS;
}
```

Listado 1.1: Programa Objective-C básico

El primer aspecto es que, en vez de usar la directiva del preprocesador `#include`, hemos usado la directiva del preprocesador `#import`. Ambas directivas incluyen un fichero dentro de otro fichero, pero a diferencia de `#include`, la directiva `#import` asegura que el fichero incluido no se incluya más de una vez. En consecuencia, en los ficheros de cabecera incluidos con `#import` no es necesario hacer un control de inclusiones con la directiva del preprocesador `#ifdef`, tal como acostumbra a hacerse en C y C++.

[1] Si, al final de este capítulo, cree que todavía no conoce lo suficiente estas herramientas, encontrará una descripción a fondo de estas herramientas en el tutorial "Compilar y depurar aplicaciones con las herramientas de programación GNU", publicado online en MacProgramadores.org.

La segunda diferencia está en que la extensión de los ficheros Objective-C es `.m`. Esta es la extensión que permite al comando `gcc` saber que se trata de un programa Objective-C.

Para compilar y ejecutar este programa, bastaría con ejecutar los siguientes comandos:

```
$ gcc holamundo.m -o holamundo
$ ./holamundo
Hola desde Objective-C
```

Framework y runtime de Objective-C

Para programar en Objective-C disponemos de dos frameworks distintos: el primer framework es el **framework de clases de GNU**, que son un conjunto de clases inicialmente desarrollado por NeXTSTEP para Objective-C, que fueron abiertas bajo licencia GNU, y cuya clase base es la clase `Object`. El segundo es el **framework de clases de NeXTSTEP**, que es el conjunto de clases que desarrolló NeXTSTEP en 1994, cuya clase base es `NSObject`, y que actualmente es el usado por Mac OS X y iOS para implementar Cocoa y Cocoa Touch. Apple no ha abierto el código fuente del framework de clases de NeXTSTEP.

Actualmente, el framework de clases de GNU ha dejado de actualizarse, y GNU también está haciendo una implementación de código fuente abierto del nuevo framework de clases de NeXTSTEP, llamado GNUStep. Esta implementación también utiliza la clase base `NSObject`, así como el resto de clases del framework de NeXTSTEP, pero aún no está terminada.

Para usar el framework de clases de GNU, debemos enlazar con el fichero `libobjc.a` usando la opción del enlazador `-lobjc`. Por el contrario, para enlazar con el framework de clases de NeXTSTEP debemos enlazar con el framework `Foundation.framework` usando la opción del enlazador `-framework Foundation`. Lógicamente, podemos enlazar con ambos frameworks de clases, proporcionando ambas opciones durante el enlazado.

Es importante no confundir los frameworks de Objective-C con los runtime de Objective-C. Los **frameworks de Objective-C** son las librerías de clases (cuya clase base es `Object` o `NSObject`, respectivamente). Los runtime de Objective-C son un conjunto de funciones de librería, escritas en C, en las que se apoyan las clases de Objective-C para alcanzar el potencial dinámico característico de este lenguaje. Por defecto, Mac OS X usa el **runtime de NeXTSTEP**, el cual actualmente da soporte tanto

al framework de clases de GNU como al framework de clases de NeXTSTEP. Podemos pedir usar el **runtime de GNU** usando la opción del enlazador `-fgnu-runtime`, pero en este caso solo tendremos acceso al framework de clases de GNU. En este último caso deberemos enlazar con la librería `libobjc-gnu.a`, usando la opción `-lobjc-gnu`.

Programar con el framework de clases de GNU

En esta sección, vamos a ver cómo programar usando el framework de clases de GNU. En la siguiente sección, veremos las diferencias cuando usamos el framework de clases de NeXTSTEP.

Normalmente, cada clase Objective-C consta de dos ficheros: uno con la extensión `.h` que contiene la interfaz, y otro con la extensión `.m` que contiene la implementación. El Listado 1.2 y el Listado 1.3 muestran, respectivamente, un ejemplo de interfaz e implementación de una clase Objective-C llamada `Saludador`. Observe que estamos usando como clase base la clase `Object` situada en el fichero de cabecera `<objc/Object.h>`.

```
/* Saludador.h */
#import <objc/Object.h>

@interface Saludador : Object {
    char* saludo;
}
- init;
- (void)setSaludo:(char*)unSaludo;
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla;
- (void)saluda;
@end
```

Listado 1.2: Interfaz de una clase Objective-C con el framework de clases de GNU

```
/* Saludador.m */
#import "Saludador.h"
#import <stdio.h>
#import <stdlib.h>
#import <string.h>

@implementation Saludador
- init {
    if ((self = [super init])) {
        saludo = "Hola mundo";
    }
    return self;
}
- (void)setSaludo:(char*)unSaludo {
```

```
    saludo = unSaludo;
}
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla {
    saludo = malloc(strlen(unSaludo)+strlen(unaColetilla)+1);
    strcpy(saludo,unSaludo);
    strcat(saludo,unaColetilla);
}
- (void)saluda {
    printf("%s\n",saludo);
}
@end
```

Listado 1.3: Implementación de una clase Objective-C con el framework de clases de GNU

Una vez tengamos definida la clase, para instanciar y usar un objeto de esta clase, necesitamos un programa principal como el del Listado 1.4.

```
/* pidesaludo.m */
#import <stdlib.h>
#import "Saludador.h"

int main() {
    Saludador* s = [[Saludador alloc] init];
    [s saluda];
    [s setSaludo: "Hola de nuevo"];
    [s saluda];
    [s setSaludo: "Hola buenos dias,"
               y: "encantado de verle"];
    [s saluda];
    [s free];
    return EXIT_SUCCESS;
}
```

Listado 1.4: Programa que usa un objeto Objective-C del framework de clases de GNU

Al estar usando el framework de clases de GNU, el programa puede ser compilado y enlazado con los siguientes comandos:

```
$ gcc -c Saludador.m
$ gcc -c pidesaludo.m
$ gcc Saludador.o pidesaludo.o –lobjc –arch i386 -o pidesaludo
```

O bien realizar los tres pasos a la vez con el siguiente comando:

```
$ gcc pidesaludo.m Saludador.m –lobjc –arch i386 -o pidesaludo
```

Observe que hemos ejecutado `gcc` con la opción `-arch i386`. Esto se debe a que Apple ha dejado de mantener este conjunto de clases, las cuales no compilan en 64 bits (`x86_64`), que es la arquitectura por defecto desde Mac OS X 10.6. Para usar el

framework de clases de GNU, tenemos que compilar para una plataforma en la que se soporte. En concreto, el framework de clases de GNU está soportado en `i386` (Intel de 32 bits) y `ppc970` (PowerPC). Las arquitecturas `armv6` y `armv7` de iOS y `ppc64` (PowerPC de 64 bits) tampoco soportan el framework de clases de GNU. Dado que Apple ha dejado de dar soporte al framework de clases de GNU, en el resto de este libro nos vamos a limitar a estudiar el framework de clases de NeXTSTEP.

Programar con el framework de clases de NeXTSTEP

El Listado 1.5, el Listado 1.6 y el Listado 1.7 muestran cómo implementar y usar la clase `Saludador` de la sección anterior, pero enlazando con el framework de clases de NeXTSTEP. La principal diferencia está en que ahora derivamos de `NSObject`, en vez de derivar de `Object`. Una segunda diferencia está en que importamos el fichero `<Foundation/NSObject.h>`, en vez de importar el fichero `<objc/Object.h>`. La tercera diferencia está en que la clase `NSObject` proporciona el método `release` para decrementar la cuenta de referencias y liberar la memoria dinámica de un objeto, mientras que la clase `Object` de GNU proporciona otro método equivalente llamado `free`. En el Listado 1.7 vemos como ahora se usa `release`.

```
/* Saludador.h */
#import <Foundation/NSObject.h>

@interface Saludador : NSObject {
    char* saludo;
}
- init;
- (void)setSaludo:(char*)unSaludo;
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla;
- (void)saluda;
@end
```

Listado 1.5: Interfaz de una clase Objective-C con el framework de clases de NeXTSTEP

```
/* Saludador.m */
#import "Saludador.h"
#import <stdio.h>
#import <stdlib.h>
#import <string.h>

@implementation Saludador
- init {
    if ((self = [super init])) {
        saludo = "Hola mundo";
    }
    return self;
}
- (void)setSaludo:(char*)unSaludo {
```

```
    saludo = unSaludo;
}
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla {
    saludo = malloc(strlen(unSaludo)+strlen(unaColetilla)+1);
    strcpy(saludo,unSaludo);
    strcat(saludo,unaColetilla);
}
- (void)saluda {
    printf("%s\n",saludo);
}
@end
```

Listado 1.6: Implementación de una clase Objective-C con el framework de clases de NeXTSTEP

```
/* pidesaludo.m */
#import <stdlib.h>
#import "Saludador.h"

int main() {
    Saludador* s = [[Saludador alloc] init];
    [s saluda];
    [s setSaludo: "Hola de nuevo"];
    [s saluda];
    [s setSaludo: "Hola buenos dias,"
               y: "encantado de verle"];
    [s saluda];
    [s release];
    return EXIT_SUCCESS;
}
```

Listado 1.7: Programa que usa un objeto Objective-C con el framework de clases de NeXTSTEP

Ahora, para compilar y ejecutar el programa usamos los siguientes comandos:

```
$ gcc Saludador.m pidesaludo.m -framework Foundation -o pidesaludo
$ ./pidesaludo
```

CLANG, LLVM Y LLDB

El compilador de GCC consta de un **front-end** y de varios **back-ends**. El back-end permite generar instrucciones en los distintos lenguajes y arquitecturas. El front-end de GCC es el comando `gcc`, y las distintas opciones de línea de comandos que soporta.

LLVM (Low-Level Virtual Machine) es un nuevo conjunto de back-ends para distintos lenguajes y arquitecturas que presenta varias ventajas técnicas respecto a los back-ends tradicionales de GCC.

En 2005, Apple contrató a miembros del equipo de desarrollo de LLVM para trabajar en mejorar estos back-ends. La idea inicial era mantener el front-end de GCC y reemplazar sus back-ends. **LLVM-GCC** es el nombre que recibió este proyecto.

El intento de combinar LLVM con GCC dio lugar a varios problemas:

- Apple usa principalmente Objective-C, pero el lenguaje Objective-C tiene poca prioridad para los desarrolladores de GCC.
- El front-end de GCC no se integra bien con las funcionalidades interactivas que los ingenieros de Apple querían meter en Xcode.
- Apple no está de acuerdo con algunas restricciones de patentes que introduce la licencia GPLv3, bajo la que se distribuye GCC 4.3.

Estos problemas hicieron que en 2007 Apple decidiera escribir un nuevo front-end al que llamaron Clang. **Clang** es un front-end que es en gran parte similar al front-end de GCC. El comando que lo implementa es `clang` y es una alternativa al comando `gcc` de las GCC. Algunas diferencias entre ellos son:

- La mayoría de las opciones del comando `clang` son similares a las del comando `gcc`. Con el tiempo, es posible que estas opciones diverjan cada vez más.
- Clang solo soporta los lenguajes C99, C++0x, C++ y Objective-C. GCC, además de estos lenguajes, soporta Java, Fortran y Ada.
- Clang y LLVM se distribuyen bajo la licencia abierta de la Universidad de Illinois. GCC se distribuye bajo licencia GNU. La licencia abierta de la Universidad de Illinois es parecida a la licencia BSD, en el sentido de que no exige distribuir el código fuente con las modificaciones que se hagan al compilador.

Aunque Apple sigue distribuyendo el comando `llvm-gcc`, el objetivo es que en el futuro solo se use el comando `clang`. GCC 4.2.1 es la última versión de GCC con licencia GPLv2, y posiblemente la última versión de GCC que incluyan las herramientas de desarrollo de Apple.

Las principales ventajas que indica Apple para el front-end Clang frente a GCC son:

- Durante el proceso de compilación, Clang mantiene más información sobre el código fuente original que GCC. Esto permite mapear más fácilmente los errores en el código fuente.
- Clang dispone de un **analizador de código estático** que permite encontrar errores comunes de programación que GCC no encontraba.
- Clang permite una compilación incremental más sencilla y eficiente, la cual se puede usar para reducir el tiempo de compilación de Xcode cuando se hacen pequeños cambios en el código fuente. Por su parte, aunque GCC soporta la

compilación incremental, está pensado para seguir un ciclo más tradicional, basado en compilar-enlazar-ejecutar.
- Clang permite indexar los símbolos del código fuente. Estos símbolos son usados por Xcode para facilitar al usuario la navegación por el código fuente.
- Clang permite compilar en varios hilos paralelos, reduciendo el tiempo de compilación en máquinas multi-core.

Las principales ventajas que indican los desarrolladores de GCC frente a Clang son:

- GCC soporta los lenguajes Java, Fortran y Ada, que Clang no soporta.
- En tests realizados en 2011, aunque Clang compila más rápido que GCC, GCC genera código más optimizado que Clang.

El proyecto LLVM fue iniciado en la Universidad de Illinois bajo la dirección de Vikram Adve y Chris Lattner. En 2005, Apple contrató a Chris Lattner como líder del equipo de desarrollo de LLVM.

La principal novedad de LLVM, frente a los back-ends de GCC, es que genera una representación intermedia llamada **IF** (Intermediate Form). Esta representación intermedia permite obtener errores y warnings más precisos, y optimizar más fácilmente el código fuente. Durante la fase de enlazado, esta representación intermedia se convierte en instrucciones del ensamblador para la máquina destino.

LLVM se distribuye con un nuevo depurador llamado `lldb`. El comando `lldb` es similar al comando `gdb` de GCC, pero permite aprovechar mejor la información que genera LLVM[2].

Compilando con Clang

Actualmente, Xcode permite elegir entre usar Clang o GCC para compilar aplicaciones Mac OS X, aunque el front-end por defecto es Clang. En esta sección vamos a ver cómo se usa Clang para compilar aplicaciones. También veremos que las opciones de línea de comando de `clang` son muy parecidas a las de `gcc`. En el resto de este libro usaremos `clang` para compilar nuestras aplicaciones.

Para compilar el ejemplo del Listado 1.1 con Clang, en vez de hacer:

[2] El comando `lldb` se encuentra en la ruta `/Developer/usr/bin/lldb`. Si quiere usar este comando desde el terminal, deberá añadir su directorio al `PATH`.

```
$ gcc holamundo.m -o holamundo
```

Hacemos:

```
$ clang holamundo.m -o holamundo
```

Análogamente, el ejemplo del Listado 1.2, del Listado 1.3 y del Listado 1.4 se compilaría con el siguiente comando:

```
$ clang pidesaludo.m Saludador.m -lobjc -arch i386 -o pidesaludo
```

O el ejemplo del Listado 1.5, del Listado 1.6 y del Listado 1.7 con el siguiente comando:

```
$ clang Saludador.m pidesaludo.m -framework Foundation -o pidesaludo
```

Crear una librería estática o dinámica

Al igual que con C o C++, con Objective-C también podemos crear una librería de enlace estático o dinámico, que luego usemos desde otros programas. Por ejemplo, para crear una **librería de enlace estático** que incluya la clase `Saludador` del Listado 1.6, podemos generar el fichero de librería con el siguiente comando:

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

Una vez creada la librería de enlace estático, podemos enlazar con ella desde el programa del Listado 1.7 ejecutando el siguiente comando:

```
$ clang pidesaludo.m libsaludos.a -framework Foundation -o pidesaludo
```

Si lo que queremos es crear una **librería de enlace dinámico**, podemos crear la librería de enlace dinámico, y enlazar el programa principal del Listado 1.7 con la librería, ejecutando los siguientes comandos:

```
$ clang -dynamiclib Saludador.o -framework Foundation -o
libSaludos.dylib
$ clang pidesaludo.m libSaludos.dylib -framework Foundation -o
pidesaludo
```

Compilación cruzada para iOS

Aunque los programas anteriores compilan directamente para Mac OS X, si queremos compilar este programa para iOS, necesitaremos conocer más a fondo Clang.

Ya hemos visto que la opción `-arch` permite indicar la plataforma para la que queremos generar el binario. Actualmente, iOS soporta dos opciones:

- `armv6` para iPhone 3G e inferiores.
- `armv7` para iPad y iPhone 3GS y superiores.

La opción `armv6` es la opción más segura, ya que todos los dispositivos iOS ejecutan los programas compilados con esta opción. Si nuestra aplicación es solo para iPad podemos usar la opción `armv7` para que el compilador genere nuevas instrucciones máquina que mejoran ligeramente el rendimiento.

Sin embargo, esta opción no es suficiente para que Clang genere una aplicación que pueda ejecutar en iOS. Las diferencias entre Mac OS X y iOS son suficientes como para que Apple haya decidido crear diferentes versiones de sus herramientas de desarrollo, una por cada plataforma. En concreto, estas plataformas de desarrollo (que se muestran en la Figura 1.2) son:

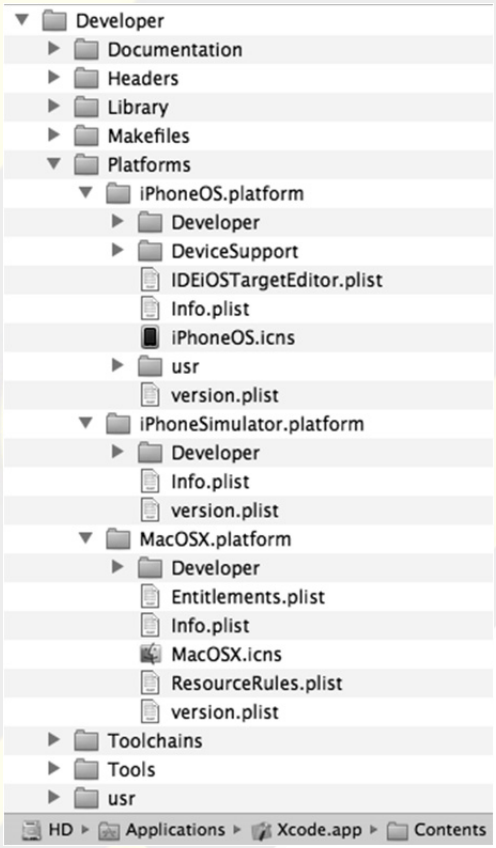

Figura 1.2: Plataformas de desarrollo

- **iPhoneOS.platform** es la plataforma de desarrollo para generar binarios para los procesadores ARM. Estos binarios se generan preparados para enlazar con las librerías de enlace dinámico del dispositivo.
- **iPhoneSimulator.platform** es la plataforma para desarrollar aplicaciones que ejecutan en iPhone Simulator. Se diferencia de la anterior en que el binario

contiene instrucciones Intel, y enlaza con las librerías de enlace dinámico del simulador.

- **MacOSX.platform** es la plataforma por defecto. Genera binarios Intel que enlazan con las librerías de enlace dinámico del sistema operativo Mac OS X.

Si queremos generar una aplicación iOS, debemos cambiar de **plataforma de desarrollo**, lo cual se suele hacer creando la variable de entorno `DEVROOT` de la siguiente forma:

```
$ export DEVROOT=/Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneOS.platform/Developer
```

Una plataforma de desarrollo puede tener más de un **SDK** (Software Development Kit). Luego, además de la plataforma de desarrollo, debemos indicar con qué SDK queremos trabajar, declarando la variable de entorno `SDKROOT` de la siguiente forma:

```
$ export SDKROOT=$DEVROOT/SDKs/iPhoneOS5.1.sdk
```

Dado que el comando `clang`, `ld`, etc., a utilizar son también distintos dependiendo de la plataforma de desarrollo, para compilar aplicaciones iOS también se suele modificar la variable de entorno `PATH` para que encuentre los comandos de la plataforma de iOS, antes que los de Mac OS X:

```
$ export PATH="$DEVROOT/usr/bin:$PATH"
```

Una vez declaradas estas variables de entorno, podemos compilar nuestra aplicación con el siguiente comando:

```
$ export CFLAGS="-isysroot $SDKROOT -miphoneos-version-min=4.0"
$ clang -arch armv6 $CFLAGS -c Saludador.m pidesaludo.m
```

La variable de entorno `CFLAGS` proporciona la opción de compilación `-isysroot`, que indica el directorio raíz donde `clang` debe buscar los ficheros de cabecera. De no proporcionar esta opción, `clang` buscaría ficheros de cabecera en `/usr/include` (el directorio de ficheros de cabecera por defecto). Esto significa que usaría los ficheros de cabecera de Mac OS X, en vez de usar los de iOS. Esto posiblemente produciría errores de compilación, ya que estos ficheros de cabecera son distintos. La opción `-miphoneos-version-min` indica la versión mínima de iOS para ejecutar la aplicación. En el ejemplo, `4.0`, indica que nuestra aplicación debe ejecutar en iOS 4.0 o superior. Conviene que este número sea lo más bajo posible para que nuestra aplicación ejecute en versiones antiguas de iOS. Solo si estamos usando funcionalidades nuevas, debemos elevar este número para poder compilar con esa nueva funcionalidad.