Linking Objective-C Code

Recently a conversation on came up on twitter as to the significance of the `-ObjC` linker flag and why this is important for us to use when compiling Objective-C code. Before I get into the specifics of those flags and why they are important to understand the difference between them, I am going to briefly explain what linking is and how it fits into the larger picture of building your app.

Building an Application

When you build an application you have to supply at least one source code file. The compiler will take this input and for each file it is given, will output another file called an object file. These object files contain machine executable code, but are not in a format that can be directly used. When using an IDE such as Xcode, they are considered to be build artifacts. This means they are an intermediate state that is used when converting the raw source files into a executable binary. These object files will typically use the same naming conventions as the original source files did; for example, `NSFoobar.m` will become `NSFoobar.o`. On Darwin-based systems, these are mach-o files that are given the type `MH_OBJECT`. This tells the kernel that while they contain runnable code, they are not setup to be executed on their own. After the compiler has finished generating all of these object files, it is the static linker's job to perform linkage to create an executable binary.

Linkers

Before we go any further it is important to make of note of the fact that Darwin-based platforms have two types of linkers:

1. Static Linker (`ld`)
2. Dynamic Linker (`dyld`)

The Static Linker is used to create applications and libraries from object files. This is the linker that is used when performing linkage when building an application.

The Dynamic Linker is what executes binaries that are on disk. This will dynamically load the application binary into memory, as well as any other frameworks and libraries that the application binary needs to run.

Libraries

Additionally, there are also two types of libraries:

1. Static Libraries (`.a`)
2. Dynamic Libraries (`.dylib` and `.framework`)

For background information as to the differences between these two types of libraries, please check out this blog post.

Binaries

When working on Apple's platforms there are three common types of targets to build:

1. Application Binaries
2. Dynamic Libraries
3. Static Libraries

Both Application Binaries and Dynamic Libraries are classified as "executable binaries". This means that they are setup in such a way that allows the Dynamic Linker to load them into memory and begin executing code.

Static Libraries are not executable binaries, they are infact archive files (typically `.ar` but uses `.a`) that are used to store the object files produced by the compiler when given source code. The files that get stored inside of these archive files are called "members".

Compiled Code

When source code is turned into machine code that can be run on a computer, the functions and methods that we create must be translated into callable subroutines. These callable subroutines are called "symbols". These symbols have names that are based on the original function or method name that was defined in the source code file then encoded in a particular manner based on the specification of the language that they are implemented in. This encoding is called "mangling" as the original human-readable names become less human-readable so that they can be uniquely defined in the application's global symbol table.

Much of the power and features of the Objective-C language comes from the fact that much of the information needed to execute code can only be determined during runtime. For example, even though an Objective-C class that you create has a method defined and implemented on that class; the code that

is run when calling that method can only be determined while the application is executing. This is because the language performs a lookup on the class to find a method based on the name you passed it to look up the selector. This selector has an associated subroutine of executable code that should be run in response to the method call. While your class has a particular method defined and implemented, at runtime the implemenation of that method can be swapped out to point to a different subroutine to be called instead. To support this behavior, when compiling Objective-C code, the compiler also generates data that defines the layout and structure of classes and the methods that are implemented on them as additional data sections that are embedded into the object file. These are parsed at runtime to register the class information with the Objective-C language runtime. It is important to note that while class defintions generate symbols that are part of the object file's symbol table, an Objective-C category does not cause symbols to be generated for it. This is because a category is only additional supplimental data for an existing defined class.

Linking

Once the compiler has generated all of the object files, these are passed to the static linker to create an executable binary. The way this works is that the linker will go through all of the object files and by using a process of static analysis it will determine which of the object files are needed to create the executable binary based on which symbols are defined on each object. This will create a graph of all of the code dependencies that exist and allow only the necessary objects be used to create the executable binary. If you are using a library that was already compiled as either a static or dynamic library, then this must also be passed to the linker using the `-l` flag (this says to link a library by name that can be found in the library search paths).

- When the linker is passed a dynamic library, then it adds it to a list of libraries that must be loaded by the dynamic linker at launch time to correctly resolve the symbols it uses. Symbol references are added to your executable that mark them as external references that will be resolvable at runtime.
- When the linker is passed a static library, it will unpack this library based on what architecture is currently being built and then will unpack the members of the archive file (this are object files). It will then add these object files to the pool of object files it already knows about to resolve symbols from.

As mentioned in the previous section, Objective-C relies heavily on the dynamic runtime it has. Part of this heavy reliance is on the data generated when compiling the Objective-C source code files. Because of this behavior that is

governed by the additional runtime data that is defined outside of the symbol table, there needs to be a way to tell the static linker that the certain object files should be loaded even if they don't seem to be necessary. The linker has three flags dedicated to controling this special type of behavior override.

`-all_load`

The `-all_load` flag tell the linker that it should link every object file (member) of every static library that is passed to the linker. This is a rather drastic option and can cause your executable binary to dramatically increase in size. This flag should be avoided if at all possible when using many static libraries.

`-ObjC`

The `-ObjC` flag controls behavior around Objective-C code. This will tell the linker that it should look through all the object files (members) of each static library to find the object files (members) that contain any additional Objective-C runtime data. This allows developers to link object files that only contain Objective-C catagories, or any other Objective-C code that the static analysis cannot resolve as being called directly (such as creating a class using `NSClassFromString()`). This is the flag that is typically passed when using static libraries that contain Objective-C code. Keep in mind, this flag means that **ALL** Objective-C code that is passed to the linker will be added to the executable binary regardless of if it gets used or not.

`-force_load`

The `-force_load` flag is very similar to the `-all_load` flag, except that it takes an argument of a path to a static library. When passed to the linker, this flag says that regardless of whatever other flags are passed to the linker, that it should link all of the object files (members) of the specific static library that is specified by the passed argument. This allows for a more controlled behavior of selectively loading all the code from one static library but not having to bloat up the executable binary with unnecessary code from other libraries.
