Introduction to Objective-C
Software Systems Final Project Spring 2016

Samantha Kumarasena and Deborah Hellen

May 2014

Introduction

In this project, we aimed to learn about Objective-C's main features, syntax, compilation, and memory management and implement what we learned to build a simple iOS application. We first researched the language and learned about its characteristics and then used CodeSchool's "Try Objective C" tutorial[1] to learn syntax and implementation. After completing this tutorial we also completed CodeSchool's "Try iOS" tutorial[2] and did some additional research into iOS development. Through these tutorials, we learned the basics of Objective-C and Apple's Cocoa/Cocoa Touch libraries.

About Objective-C

Objective-C is a class-based, object-oriented language based on the C language. It first appeared in 1983, making it one of the oldest object-oriented languages. It is the main programming language used by Apple for OSX and iOS applications. Objective-C is a perfect superset of C meaning that an Objective-C program is capable of using C without modification and allows programmers to use C libraries. Despite this, Objective-C uses a very distinct, Smalltalk-like syntax that is unusual within the context of the C language.

Structure and Syntax

Objective-C's syntax is very similar to that of Smalltalk. Smalltalk is an object-oriented language invented in the 1970s, created largely for educational uses.[3] It features a "message-sending" syntax, in which messages are sent to objects. These messages are much like methods – they can return results and take parameters. However, their syntax differs significantly. An example of Objective-C message-sending:

```
[object method:argument];
```

This is quite different from C's typical "dot" notation.

Additionally, Objective-C method and variable names are usually camel-cased and are quite verbose, particularly the Cocoa and Cocoa Touch libraries. For example:

```
stringByReplacingOccurrencesOfString
```

[1]Tutorial can be found at: http://tryobjectivec.codeschool.com/
[2]Tutorial can be found at: https://www.codeschool.com/courses/try-ios
[3]Smalltalk: http://www.smalltalk.org/main/

This method does precisely what it says (creates a new string by replacing occurrences of a substring within an existing string).[4] Function names typically serve as a full description of the function, which can make Objective-C difficult to decipher. This, combined with an unusual Smalltalk-based message structure, can be daunting to a programmer new to Objective-C.

However, Objective-C also implements many potentially useful features not present in C. Objective-C enables programmers to choose between static and dynamic typing. Variable types can be specified, just as in C. However, Objective-C allows for dynamic typing when an object's type may not be known at runtime. A catch-all type, id, is used to declare a variable of indeterminate type.[5] This is similar in concept to a void pointer in C; however, Objective-C is much more supportive of dynamic typing than C. Class protocols determine whether and how objects of various types interact, and so dynamic typing is not as dangerous because runtime exceptions will alert a programmer to a type mismatch. (Protocols will be discussed in more depth in the next section, titled "Object-Oriented Development Features".)

As far as types go, Objective-C is capable of using traditional C types. However, Apple includes its own libraries of classes for app-creation (Cocoa for OSX and Cocoa Touch for iOS). Since Objective-C is mainly used for OSX/iOS development, Cocoa and Cocoa Touch classes are frequently used (in the tutorial we followed, we used Cocoa classes almost exclusively). These classes are generally prefixed with "NS": e.g. NSNumber, or NSArray. Some are basic types, such as strings and numbers. Others are interface elements, such as table views and buttons. Still others are data structures, such as dictionaries. NSNumbers and NSArrays have analogues in C (namely ints and arrays).

However, some of these analogue classes are mutable in C but immutable in Objective-C – NSNumber is one example. As a result, if an Objective-C programmer wishes to perform numerical operations, the NSNumber must be casted to an int. Operations can be performed on the int, and the resulting int would then be casted back to an NSNumber.[6] This can be inconvenient. However, the founders of Objective-C decided to make NSNumber immutable because the values of numbers are by nature immutable. Furthermore, the immutability of the NSNumber class – along with other classes such as NSArray and NSString – is due to the fact that immutable objects are more reliable when being passed to and shared by many different objects.

Additionally, Objective-C implements blocks. Blocks are a syntactical feature similar to that of C functions. However, they are often connected to segments of either the stack or the heap, enabling them to save state in the segments of memory they are connected to (which can change the outcome of the block; in other words, executing the same block twice may not yield the same results). Blocks can be nameless, and are literal-inlined where required in the code. Additionally, they have access to all local variables in their scope. As a result, they're a simple way to encapsulate small amounts of code. They can serve as an alternative to traditional callback functions; blocks can access to local variables, meaning that callback functions can access all context information directly.[7]

[4]More information on the Cocoa/Cocoa Touch libraries: https://developer.apple.com/library/ios/documentation/miscellaneous/conceptual/iphoneostechoverview /iPhoneOSTechnologies/iPhoneOSTechnologies.html

[5]Static versus dynamic typing: https://developer.apple.com/library/ios/documentation/general/conceptual/DevPedia-CocoaCore/DynamicTyping.html

[6]Mutability in Objective-C: https://developer.apple.com/library/ios/documentation/general/conceptual/ CocoaEncyclopedia/ObjectMutability/ObjectMutability.html

[7]More on blocks: https://developer.apple.com/library/mac/documentation/cocoa/Conceptual/Blocks/Articles/ bxGettingStarted.html

Object Oriented Development Features

Objective-C allows programmers to define categories, another concept borrowed from Smalltalk. Categories enable programmers to write methods for classes for which they do not have the source code. With categories, methods are added to class definitions at run-time, ensuring that the class definition does not need to be recompiled when modifying a class (and so the programmer does not need access to the source) – methods within the category are indistinguishable from the methods defined within the source.[8] This allows code to be written in small pieces and combined later, which is often considered good practice for developing and debugging.

Protocols are an important part of defining interfaces between classes in Objective-C. They define the procedures classes use to interact with each other by defining the methods and/or variables both classes must have to do so. If these conditions are not met, a runtime exception will occur. Protocols can also include optional messages, which must be explicitly checked before they can be used. Cocoa and Cocoa Touch classes define many protocols, and programmers often write their own protocols that inherit from these default protocols. Furthermore, protocols help ensure class anonymity – a programmer does not need to know the class an object belongs to (which is particularly important if the object's class needs to remain hidden).[9] If the programmer wants to interact with that object, the protocol will suffice.

Compilation and Runtime

For the most part, the compilation process for Objective-C is the same as that of C because Objective-C is a superset of C and takes advantage of all of C's syntax and features. Like in C, the compiler first goes through a series of checks to ensure that the syntax is valid, the statically-allocated objects have the correct types, and the code structure is correct and then it produces object code. For each source file, the compiler makes a corresponding object file which has a .o or .obj extension. If a project has multiple source files, after compilation the linker checks that all of the correct object files and libraries exist and are accessible and then it combines all of the object files into a single executable. [10]

Many C compilers (such as gcc) now compile Objective-C when given the proper flags, but most commonly Objective-C is compiled using the built-in compiler provided in Xcode. Xcode is the standard development environment for iOS and OSX. In Xcode, the compiling and linking are combined into the "build" feature, which removes much of the linking responsibility from the programmer.

Finding errors during compilation in Objective-C is a bit different than in C because Objective-C uses both Smalltalk-style messaging and C syntax and therefore supports both dynamic and static typing. This means that the compiler checks only the statically allocated objects at compile-time. For example, an object might be sent a message that is not specified in its interface. At compile time, there will be no warning or error message to alert the programmer if the argument does not match the specified type. Though the program would successfully compile, at runtime it would raise a runtime exception.

Objective-C also provides a late-binding, dynamic runtime in that the language connects methods names with their implementations immediately before they're used during runtime. In C, this is done at compile-time. This allows for "method swizzling", switching two method names and their

[8] https://developer.apple.com/library/mac/documentation/cocoa/conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html

[9] Protocols and anonymity: https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/WorkingwithProtocols/WorkingwithProtocols.html

[10] Objective-C compilation process http://iosdevelopmenttutorials.com/the-four-phases-of-implementing-an-objective-c-application/

implementations. This is useful when overriding methods (a programmer does not have to subclass to override, and can access the original implementation). Furthermore, if an object is not prepared to handle a message – if that method is not defined within its protocol – a dynamic runtime can sometimes help a programmer to handle this issue through "dynamic method resolution". Instead of issuing a runtime exception, the program can attempt to handle this by supplying an alternate function, if one is provided by the programmer. If this fails, the runtime can move on to "message forwarding" (forwarding a message to an object that is prepared to handle it, if such an object exists).[11] This helps the programmer deal with typing issues that arise from using dynamic typing (type mismatch, etc.)

Memory Management

Like in C, memory management primarily deals with the process of allocating memory during runtime, using it, and then freeing it when the program is done with it. When writing Objective-C it is common to consider memory management in terms of the application's "object graph", which is a representation of the relationships between groups of objects. This is important because deallocating an object before its dependents can cause major program crashes that are difficult to debug.

There are two main types of memory management available in Objective-C. The first is Manual Retain-Release (MRR) in which the programmer explicitly manages memory by keeping track of objects using reference counting provided by NSObject. The programmer manually inserts the memory management calls and deallocation methods. The second memory management option is Automatic Reference Counting (ARC).[12] It relies upon the same reference counting system as MRR, but the compiler automatically inserts the memory management calls at compile-time by evaluating the lifetime requirements of objects and blocks. ARC is newer than MRR and is the more widely encouraged method because it lessens the tedious and error-prone task of memory management for the programmer. It is important to note that in this system, the programmer still needs to call "alloc" when creating an object, but does not need to call "dealloc".[13]

One of the common challenges in Objective-C is managing memory for multiple related objects. Objects can be related by "strong" references, meaning that your application takes ownership of the value that it references, or "weak" references, meaning that your application does not take ownership of the value. An object cannot be deallocated until all of its strong references are released, which can cause a retain cycle if two objects have strong references to each other. ARC does not necessarily protect against creating a retain cycle, so it is the programmer's responsibility to allow all objects to be released. One way to avoid creating a retain cycle is to specify a "parent" object that has strong references to its "children," but the children only have weak references to the parent.[14] For example, a node in a linked list may have a weak reference to the previous node to avoid causing cyclical references.

[11]An interesting description of Objective-C's dynamic runtime: Martin Pilkington, http://pilky.me/21/

[12]About Memory Management on the Mac Developer Library - https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html

[13]Transitioning to Automatic Reference Counting https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html

[14]Practical Memory Management Tips https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmPractical.html

Writing an App

We finished the "Try Objective-C" and the "Try iOS" tutorials. Both were based on Objective-C, but "Try Objective-C" dealt only with basic syntax and features while "Try iOS" involved learning simple app development. In these tutorials, we experienced a few of the features described above. The syntax of the language was difficult to understand at first because it is heavily based on Smalltalk, which neither of us were familiar with. We also found the verbose naming scheme tiresome and found that it caused us to repeatedly check documentation. Code became convoluted and difficult to read very quickly because of the (in our opinion, excessively) long method names.

The majority of the tutorial levels focused on app development with the Cocoa and Cocoa Touch libraries, and we were able to see how objects typically interact in an app setting. Inheritance featured strongly in the tutorial; the inheritance paths between Cocoa Touch classes are fairly straightforward. We were not able to examine many of the other properties of the language through the tutorials – they were mainly an introduction to the Cocoa/Cocoa Touch libraries. If this project were to continue, we would have tried to experiment more with some of the other features, such as categories and method swizzling, that we mentioned above.