Functions

Along with variables, conditionals, and loops, functions are one of the fundamental components of any modern programming language. They let you reuse an arbitrary block of code throughout your application, which is necessary for organizing and maintaining all but the most trivial code bases. You'll find many examples of functions throughout the iOS and OS X frameworks.

Just like its other basic constructs, Objective-C relies entirely on the C programming language for functions. This module introduces the most important aspects of C functions, including basic syntax, the separation of declaration and implementation, common scope issues, and function library considerations.

Basic Syntax

There are four components to a C function: its return value, name, parameters, and associated code block. After you've defined these, you can **call** the function to execute its associated code by passing any necessary parameters between parentheses.

For example, the following snippet defines a function called `getRandomInteger()` that accepts two `int` values as parameters and returns another `int` value. Inside of the function, we access the inputs through the `minimum` and `maximum` parameters, and we return a calculated value via the `return` keyword. Then in `main()`, we call this new function and pass -10 and 10 as arguments.

```
// main.m
#import <Foundation/Foundation.h>

int getRandomInteger(int minimum, int maximum) {
```

```
    return arc4random_uniform((maximum - minimum) + 1) + minimum;
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int randomNumber = getRandomInteger(-10, 10);
        NSLog(@"Selected a random number between -10 and 10: %d",
              randomNumber);
    }
    return 0;
}
```

The built-in `arc4random_uniform()` function returns a random number between 0 and whatever argument you pass. (This is preferred over the older `rand()` and `random()` algorithms.)

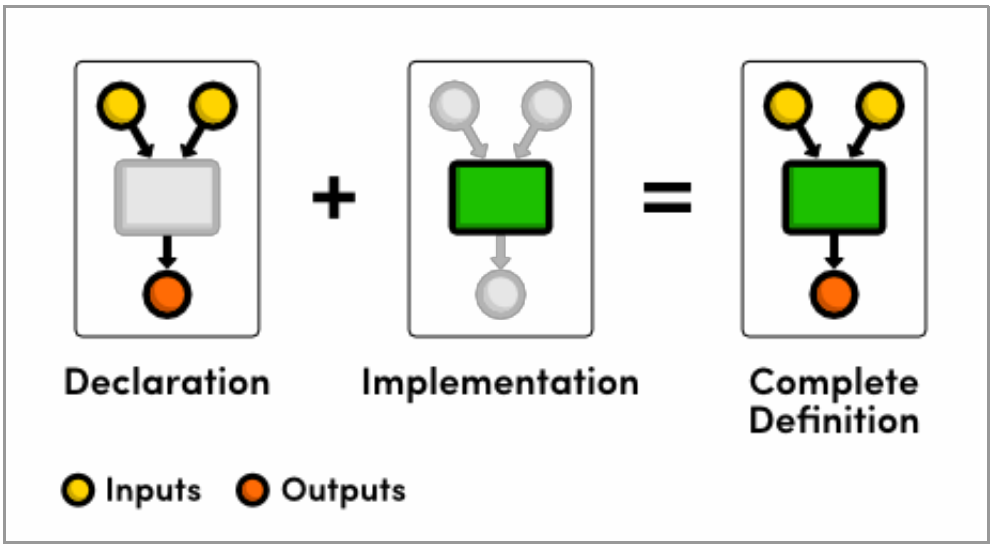
Functions let you use pointer references as return values or parameters, which means that they can be seamlessly integrated with Objective-C objects (remember that all objects are represented as pointers). For example, try changing `main.m` to the following.

```
// main.m
#import <Foundation/Foundation.h>

NSString *getRandomMake(NSArray *makes) {
    int maximum = (int)[makes count];
    int randomIndex = arc4random_uniform(maximum);
    return makes[randomIndex];
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *makes = @[@"Honda", @"Ford", @"Nissan", @"Porsche"];
        NSLog(@"Selected a %@", getRandomMake(makes));
    }
    return 0;
}
```

This `getRandomMake()` function accepts an `NSArray` object as an argument and returns an `NSString` object. Note that it uses the same asterisk syntax as pointer variable declarations.

Declarations vs. Implementations

Functions need to be defined *before* they are used. If you were to define the above `getRandomMake()` function *after* `main()`, the compiler wouldn't be able to find it when you try to call it in `main()`. This imposes a rather strict structure on developers and can make it hard to organize larger applications. To solve this problem, C lets you separate the declaration of a function from its implementation.



Function declarations vs. implementations

A function **declaration** tells the compiler what the function's inputs and outputs look like. By providing the data types for the return value and parameters of a function, the compiler can make sure that you're using it properly without knowing what it actually does. The corresponding **implementation** attaches a code block to the declared function. Together, these give you a complete function **definition**.

The following example declares the `getRandomMake()` function so that it can be used in `main()` before it gets implemented. Notice that the declaration only needs the data types of the parameters—their names can be omitted (if desired).

```
// main.m
#import <Foundation/Foundation.h>
```

```
// Declaration
NSString *getRandomMake(NSArray *);

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *makes = @[@"Honda", @"Ford", @"Nissan", @"Porsche"];
        NSLog(@"Selected a %@", getRandomMake(makes));
    }
    return 0;
}

// Implementation
NSString *getRandomMake(NSArray *makes) {
    int maximum = (int)[makes count];
    int randomIndex = arc4random_uniform(maximum);
    return makes[randomIndex];
}
```

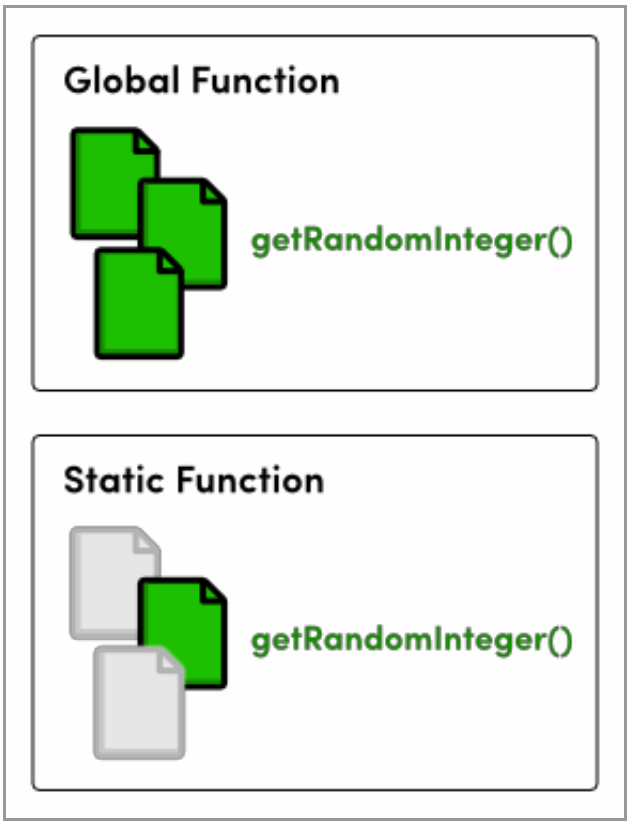
As we'll see in Function Libraries, separating function declarations from implementations is really more useful for organizing large frameworks.

The Static Keyword

The `static` keyword lets you alter the availability of a function or variable. Unfortunately, it has different effects depending on where you use it. This section explains two common use cases for the `static` keyword.

Static Functions

By default, all functions have a global scope. This means that as soon as you define a function in one file, it's immediately available everywhere else. The `static` specifier lets you limit the function's scope to the current file, which is useful for creating "private" functions and avoiding naming conflicts.



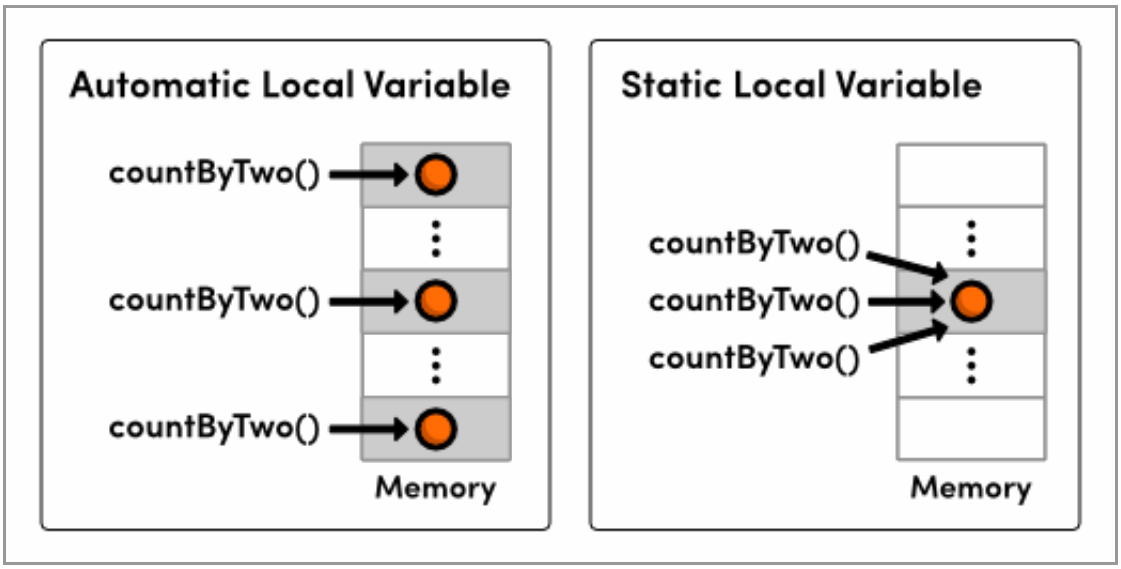
Globally-scoped functions vs. statically-scoped functions

The following example shows you how to create a static function. If you were to add this code to another file (e.g., a dedicated function library), you would not be able to access `getRandomInteger()` from `main.m`. Note that the `static` keyword should be used on both the function declaration and implementation.

```
// Static function declaration
static int getRandomInteger(int, int);

// Static function implementation
static int getRandomInteger(int minimum, int maximum) {
    return arc4random_uniform((maximum - minimum) + 1) + minimum;
}
```

Static Local Variables

Variables declared inside of a function (also called **automatic local variables**) are reset each time the function is called. This is an intuitive default behavior, as the function behaves consistently regardless of how many times you call it. However, when you use the `static` modifier on a local variable, the function "remembers" its value across invocations.



Independent automatic variables vs. shared static variables

For example, the `currentCount` variable in the following snippet never gets reset, so instead of storing the count in a variable inside of `main()`, we can let `countByTwo()` do the recording for us.

```
// main.m
#import <Foundation/Foundation.h>

int countByTwo() {
    static int currentCount = 0;
    currentCount += 2;
    return currentCount;
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"%d", countByTwo());    // 2
        NSLog(@"%d", countByTwo());    // 4
        NSLog(@"%d", countByTwo());    // 6
    }
    return 0;
}
```

But, unlike the static functions discussed in the previous section, this use of the `static` keyword does *not* affect the scope of local variables. That is to say, local variables are still only accessible inside of the function itself.

Function Libraries

Objective-C doesn't support namespaces, so to prevent naming collisions with other global functions, large frameworks need to prefix their functions (and classes) with a unique identifier. This is why you see built-in functions like `NSMakeRange()` and `CGImageCreate()` instead of just `makeRange()` and `imageCreate()`.

When creating your own function libraries, you should declare functions in a dedicated header file and implement them in a separate implementation file (just like Objective-C classes). This lets files that use the library import the header without worrying about how its functions are implemented. For example, the header for a `CarUtilities` library might look something like the following:

```
// CarUtilities.h
#import <Foundation/Foundation.h>

NSString *CUGetRandomMake(NSArray *makes);
NSString *CUGetRandomModel(NSArray *models);
NSString *CUGetRandomMakeAndModel(NSDictionary *makesAndModels);
```

The corresponding implementation file defines what these functions actually do. Since other files are not supposed to import the implementation, you can use the `static` specifier to create "private" functions for internal use by the library.

```
// CarUtilities.m
#import "CarUtilities.h"

// Private function declaration
static id getRandomItemFromArray(NSArray *anArray);

// Public function implementations
NSString *CUGetRandomMake(NSArray *makes) {
    return getRandomItemFromArray(makes);
}
NSString *CUGetRandomModel(NSArray *models) {
    return getRandomItemFromArray(models);
}
NSString *CUGetRandomMakeAndModel(NSDictionary *makesAndModels) {
```

```
    NSArray *makes = [makesAndModels allKeys];
    NSString *randomMake = CUGetRandomMake(makes);
    NSArray *models = makesAndModels[randomMake];
    NSString *randomModel = CUGetRandomModel(models);
    return [randomMake stringByAppendingFormat:@" %@", randomModel];
}

// Private function implementation
static id getRandomItemFromArray(NSArray *anArray) {
    int maximum = (int)[anArray count];
    int randomIndex = arc4random_uniform(maximum);
    return anArray[randomIndex];
}
```

Now, `main.m` can import the header and call the functions as if they were defined in the same file. Also notice that trying to call the static `getRandomItemFromArray()` function from `main.m` results in a compiler error.

```
// main.m
#import <Foundation/Foundation.h>
#import "CarUtilities.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSDictionary *makesAndModels = @{
            @"Ford": @[@"Explorer", @"F-150"],
            @"Honda": @[@"Accord", @"Civic", @"Pilot"],
            @"Nissan": @[@"370Z", @"Altima", @"Versa"],
            @"Porsche": @[@"911 Turbo", @"Boxster", @"Cayman S"]
        };
        NSString *randomCar = CUGetRandomMakeAndModel(makesAndModels);
        NSLog(@"Selected a %@", randomCar);
    }
    return 0;
}
```

Summary

This module finished up our introduction to the C programming language with an in-depth look at functions. We learned how to declare and implement functions, change their scope, make them remember local variables, and organize large function libraries.

While most of the functionality behind the Cocoa and Cocoa Touch frameworks is encapsulated as Objective-C classes, there's no shortage of built-in functions. The ones you're most likely to encounter in the real world are utility functions like `NSLog()` and convenience functions like `NSMakeRect()` that create and configure complex objects using a friendlier API.

We're now ready to start tackling the object-oriented aspects of Objective-C. In the next module, we'll learn how to define classes, instantiate objects, set properties, and call methods.

Continue to *Classes* ›