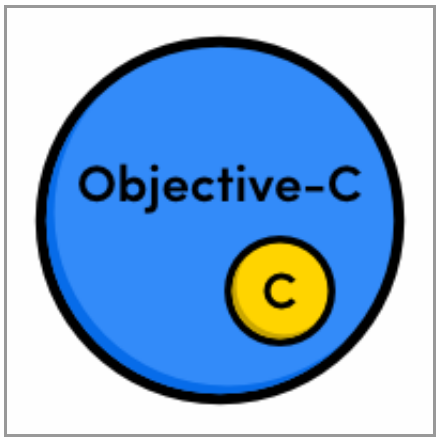
C Basics

Objective-C is a strict superset of C, which means that it's possible to seamlessly combine both languages in the same source file. In fact, Objective-C *relies* on C for most of its core language constructs, so it's important to have at least a basic foundation in C before tackling the higher-level aspects of the language.



The relationship between Objective-C and C

This module provides a concise overview the C programming language. We'll talk about comments, variables, mathematical operators, control flow, simple data structures, and the integral role of pointers in Objective-C programs. This will give us the necessary background to discuss Objective-C's object-oriented features.

Comments

There are two ways to include commentary text in a C program. **Inline comments** begin with a double slash and terminate at the end of the current line. **Block comments** can span multiple lines, but they must be enclosed in /* and */ characters. For example:

```
// This is an inline comment

/* This is a block comment.
   It can span multiple lines. */
```

Since comments are completely ignored by the compiler, they let you include extra information alongside your code. This can be useful for explaining confusing snippets; however, Objective-C is designed to be very self-documenting, so you shouldn't really need to include a lot comments in your iOS or OS X applications.

Variables

Variables are containers that can store different values. In C, variables are statically typed, which means that you must explicitly state what kind of value they will hold. To **declare** a variable, you use the `<type> <name>` syntax, and to assign a value to it you use the = operator. If you need to interpret a variable as a different type, you can **cast** it by prefixing it with the new type in parentheses.

All of this is demonstrated in the following example. It declares a variable called `odometer` that stores a value of type `double`. The `(int)odometer` statement casts the value stored in `odometer` to an integer. If you paste this code into your `main.m` file in Xcode and run the program, you should see the `NSLog()` messages displayed in your *Output* panel.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        double odometer = 9200.8;
        int odometerAsInteger = (int)odometer;

        NSLog(@"You've driven %.1f miles", odometer);        // 9200.8
        NSLog(@"You've driven %d miles", odometerAsInteger); // 9200
    }
    return 0;
}
```

Along with `double` and `int`, C defines a plethora of primitive data types. A comprehensive list can be found in the Primitives module, as well as an explanation of the `%.1f` and `%d` format specifiers used above.

Constants

The `const` variable modifier can be used to tell the compiler that a variable is never allowed to change. For example, defining a constant called `pi` and then trying to alter it will result in a compiler error:

```
double const pi = 3.14159;
pi = 42001.0;               // Compiler error
```

This is often used in function parameters to inform the caller that they can safely assume whatever value they pass will not be altered by the function.

Arithmetic

The familiar +, -, *, / symbols are used for basic arithmetic operations, and the modulo operator (%) can be used to return the remainder of an integer division. These are all demonstrated below.

```
NSLog(@"6 + 2 = %d",  6 + 2);    // 8
NSLog(@"6 - 2 = %d",  6 - 2);    // 4
NSLog(@"6 * 2 = %d",  6 * 2);    // 12
NSLog(@"6 / 2 = %d",  6 / 2);    // 3
NSLog(@"6 %% 2 = %d", 6 % 2);    // 0
```

Special care must be taken when performing operations that involve both floating-point and integer types. Please see Integer Division for details.

You'll also frequently encounter the increment (++) and decrement (--) operators when working with loops. These are convenience operators for adding or subtracting 1 from a variable. For example:

```
int i = 0;
NSLog(@"%d", i);    // 0
```

```
i++;
NSLog(@"%d", i);    // 1
i++;
NSLog(@"%d", i);    // 2
```

Conditionals

C provides the standard `if` statement found in most programming languages. Its syntax, along with a table describing the most common relational/logical operators, is shown below.

```
int modelYear = 1990;
if (modelYear < 1967) {
    NSLog(@"That car is an antique!!!");
} else if (modelYear <= 1991) {
    NSLog(@"That car is a classic!");
} else if (modelYear == 2013) {
    NSLog(@"That's a brand new car!");
} else {
    NSLog(@"There's nothing special about that car.");
}
```

Operator	Description
`a == b`	Equal to
`a != b`	Not equal to
`a > b`	Greater than
`a >= b`	Greater than or equal to
`a < b`	Less than
`a <= b`	Less than or equal to
`!a`	Logical negation
`a && b`	Logical and
`a || b`	Logical or

C also includes a `switch` statement, however it only works with integral types—not floating-point numbers, pointers, or Objective-C objects. This makes it rather inflexible when compared to the `if` conditionals discussed above.

```
// Switch statements (only work with integral types)
switch (modelYear) {
    case 1987:
        NSLog(@"Your car is from 1987.");
        break;
    case 1988:
        NSLog(@"Your car is from 1988.");
        break;
    case 1989:
    case 1990:
        NSLog(@"Your car is from 1989 or 1990.");
        break;
    default:
        NSLog(@"I have no idea when your car was made.");
        break;
}
```

Loops

The `while` and `for` loops can be used for iterating over values, and the related `break` and `continue` keywords let you exit a loop prematurely or skip an iteration, respectively.

```
int modelYear = 1990;
// While loops
int i = 0;
while (i<5) {
    if (i == 3) {
        NSLog(@"Aborting the while-loop");
        break;
    }
    NSLog(@"Current year: %d", modelYear + i);
    i++;
}
// For loops
for (int i=0; i<5; i++) {
    if (i == 3) {
```

```
        NSLog(@"Skipping a for-loop iteration");
        continue;
    }
    NSLog(@"Current year: %d", modelYear + i);
}
```

While it's technically not a part of the C programming language, this is an appropriate time to introduce the `for-in` loop. This is referred to as the **fast-enumeration** syntax because it's a more efficient way to iterate over Objective-C collections like `NSSet` and `NSArray` than the traditional `for` and `while` loops.

```
// For-in loops ("Fast-enumeration," specific to Objective-C)
NSArray *models = @[@"Ford", @"Honda", @"Nissan", @"Porsche"];
for (id model in models) {
    NSLog(@"%@", model);
}
```

Macros

Macros are a low-level way to define symbolic constants and space-saving abbreviations. The `#define` directive maps a macro name to an expansion, which is an arbitrary sequence of characters. Before the compiler tries to parse the code, the preprocessor replaces all occurrences of the macro name with its expansion. In other words, it's a straightforward search-and-replace:

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

#define PI 3.14159
#define RAD_TO_DEG(radians) (radians * (180.0 / PI))

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        double angle = PI / 2;              // 1.570795
        NSLog(@"%f", RAD_TO_DEG(angle));    // 90.0
    }
    return 0;
```

```
}
```

This code snippet demonstrates the two types of C macros: object-like macros (`PI`) and function-like macros (`RAD_TO_DEG(radians)`). The only difference is that the latter is smart enough to accept arguments and alter their expansions accordingly.

Typedef

The `typedef` keyword lets you create new data types or redefine existing ones. After typedef'ing an `unsigned char` in the following example, we can use `ColorComponent` just like we would use `char`, `int`, `double`, or any other built-in type:

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

typedef unsigned char ColorComponent;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        ColorComponent red = 255;
        ColorComponent green = 160;
        ColorComponent blue = 0;
        NSLog(@"Your paint job is (R: %hhu, G: %hhu, B: %hhu)",
              red, green, blue);
    }
    return 0;
}
```

While this adds some meaningful semantics to our code, `typedef` is more commonly used to turn `struct`'s and `enum`'s into convenient data types. This is demonstrated in the next two sections.

Structs

A `struct` is like a simple, primitive C object. It lets you aggregate several variables into a more complex data structure, but doesn't provide any OOP features (e.g., methods). For example, the following snippet uses a `struct` to group the

components of an RGB color. Also notice how we `typedef` the `struct` so that we can access it via a meaningful name.

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

typedef struct {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} Color;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Color carColor = {255, 160, 0};
        NSLog(@"Your paint job is (R: %hhu, G: %hhu, B: %hhu)",
              carColor.red, carColor.green, carColor.blue);
    }
    return 0;
}
```

To populate the new `carColor` structure, we used the `{255, 160, 0}` **initializer syntax**. This assigns values in the same order as they were declared in the `struct`. And, as you can see, each of its fields can be accessed via dot-syntax.

Enums

The `enum` keyword is used to create an **enumerated type**, which is a collection of related constants. Like `structs`, it's often convenient to `typedef` enumerated types with a descriptive name:

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

typedef enum {
    FORD,
    HONDA,
```

```
    NISSAN,
    PORSCHE
} CarModel;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        CarModel myCar = NISSAN;
        switch (myCar) {
            case FORD:
            case PORSCHE:
                NSLog(@"You like Western cars?");
                break;
            case HONDA:
            case NISSAN:
                NSLog(@"You like Japanese cars?");
                break;
            default:
                break;
        }
    }
    return 0;
}
```

Since the `myCar` variable was declared with the `CarModel` type, it can only store the four **enumerators** defined by the enumerated type: `FORD`, `HONDA`, `NISSAN`, and `PORSCHE`. Defining these in an enumerated type is more reliable than representing the various `CarModel`'s with arbitrary strings, as it's impervious to spelling errors (the compiler will let you know when you mistype one of the above enumerators).

Primitive Arrays

Since Objective-C is a superset of C, it has access to the primitive arrays found in C. Generally, the higher-level `NSArray` and `NSMutableArray` classes provided by the Foundation Framework are much more convenient than C arrays; however, primitive arrays can still prove useful for performance intensive environments. Their syntax is as follows:

```
int years[4] = {1968, 1970, 1989, 1999};
```

```
years[0] = 1967;
for (int i=0; i<4; i++) {
    NSLog(@"The year at index %d is: %d", i, years[i]);
}
```

The `int years[4]` statement allocates a contiguous block of memory large enough to store 4 `int` values. We then populate the array using the `{1968, ...}` initializer syntax and access its elements by passing offsets between square brackets (e.g., `years[i]`).

Pointers

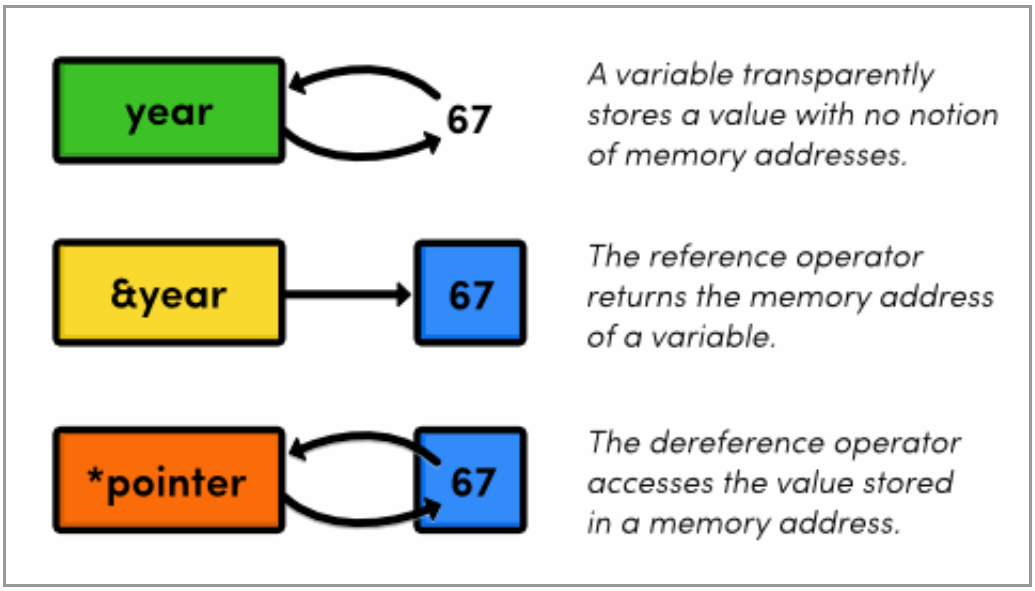
A pointer is a direct reference to a memory address. Whereas a variable acts as a transparent container for a value, pointers remove a layer of abstraction and let you see how that value is stored. This requires two new tools:

- The reference operator (`&`) returns the memory address of a normal variable. This is how you create pointers.
- The dereference operator (`*`) returns the contents of a pointer's memory address.

The following example demonstrates how to declare, create, and dereference pointers. Note that defining a pointer looks exactly like defining a normal variable, except it's prepended with an asterisk (`*`).

```
int year = 1967;          // Define a normal variable
int *pointer;             // Declare a pointer that points to an int
pointer = &year;          // Find the memory address of the variable
NSLog(@"%d", *pointer);   // Dereference the address to get its value
*pointer = 1990;          // Assign a new value to the memory address
NSLog(@"%d", year);       // Access the value via the variable
```

The behavior of these pointer operators can be visualized as follows:



In the above example, pointers are merely an unnecessary abstraction for normal variables. Their real utility comes from the fact that you can *move* a pointer to the surrounding memory addresses. This is particularly useful for navigating arrays, which are just contiguous blocks of memory. For example, the code below uses a pointer to iterate through the elements of an array.

```
char model[5] = {'H', 'o', 'n', 'd', 'a'};
char *modelPointer = &model[0];
for (int i=0; i<5; i++) {
    NSLog(@"Value at memory address %p is %c",
          modelPointer, *modelPointer);
    modelPointer++;
}
NSLog(@"The first letter is %c", *(modelPointer - 5));
```

When used with a pointer, the ++ operator moves the it to the next memory address, which we can display through `NSLog()` with the `%p` specifier. Likewise, the -- operator can be used to decrement the pointer to the previous address. And, as shown in the last line, you can access an arbitrary address relative to the current pointer position.

The Null Pointer

The null pointer is a special kind of pointer that doesn't point to anything. There is only one null pointer in C, and it is referenced through the `NULL` macro. This is useful for indicating empty variables—something that is not possible with a normal data type. For instance, the following snippet shows how `pointer` can be "emptied" using

the null pointer.

```
int year = 1967;
int *pointer = &year;
NSLog(@"%d", *pointer);     // Do something with the value
pointer = NULL;             // Then invalidate it
```

The only way to represent an empty variable using `year` on its own is to set it to `0`. Of course, the problem is that `0` is still a perfectly valid value—it's not the *absence* of a value.

Void Pointers

A void pointer is a generic type that can point to *anything*. It's essentially a reference to an arbitrary memory address. Accordingly, more information is required to interpret the contents of a void pointer. The easiest way to do this is to simply cast it to a non-void pointer. For example, the `(int *)` statement in the following code interprets the contents of the void pointer as an `int` value.

```
int year = 1967;
void *genericPointer = &year;
int *intPointer = (int *)genericPointer;
NSLog(@"%d", *intPointer);
```

The generic nature of void pointers affords a lot flexibility. For example, the `NSString` class defines the following method for converting a C array into an Objective-C string object:

```
- (id)initWithBytes:(const void *)bytes
             length:(NSUInteger)length
           encoding:(NSStringEncoding)encoding
```

The `bytes` argument points to the first memory address of any kind of C array, the `length` argument defines how many bytes to read, and `encoding` determines how those bytes should be interpreted. Using a void pointer like this makes it possible to work with *any* type of character array. The alternative would be to define dedicated methods for single-byte, UTF-8, and UTF-16 characters, etc.

Pointers in Objective-C

This is all good background knowledge, but for your everyday Objective-C development, you probably won't need to use most of it. The only thing that you really have to understand is that *all* Objective-C objects are referenced as pointers. For instance, an `NSString` object must be stored as a pointer, not a normal variable:

```
NSString *model = @"Honda";
```

When it comes to null pointers, there is a slight difference between C and Objective-C. Whereas C uses `NULL`, Objective-C defines its own macro, `nil`, as its null object. A good rule of thumb is to use `nil` for variables that hold Objective-C objects and `NULL` when working with C pointers.

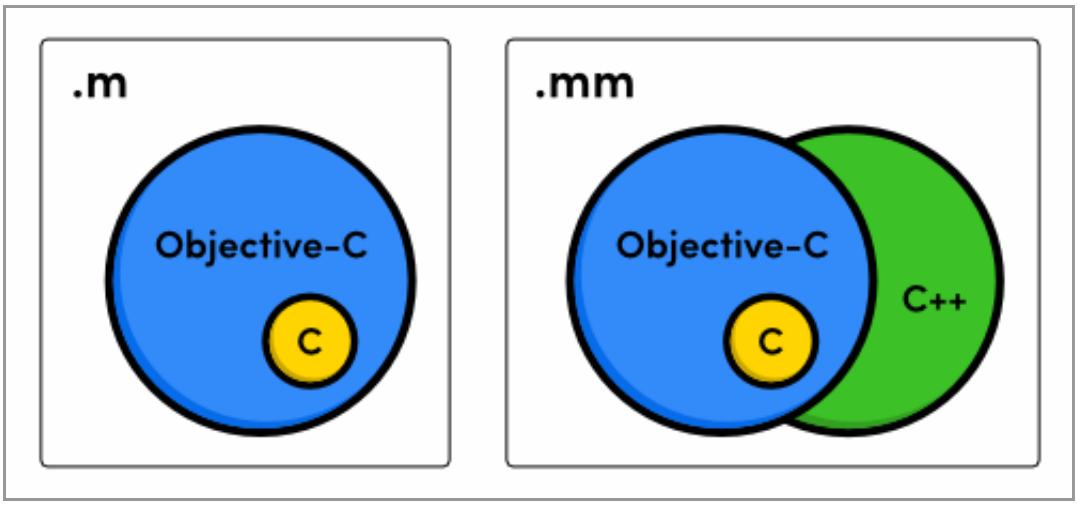
```
NSString *anObject;    // An Objective-C object
anObject = NULL;       // This will work
anObject = nil;        // But this is preferred
int *aPointer;         // A plain old C pointer
aPointer = nil;        // Don't do this
aPointer = NULL;       // Do this instead
```

Outside of variable declarations, the entire Objective-C syntax is designed to work with pointers. So, after defining an object pointer, you can basically forget about the fact that it's a pointer and interact with it as if it were a normal variable. This will be made abundantly clear from the examples throughout the rest of this tutorial.

Summary

This module introduced the fundamental aspects of the C programming language. While you're not expected to be a C expert just yet, we hope that you're feeling relatively comfortable with variables, conditionals, loops, `struct`'s, `enum`'s, and pointers. These tools form the foundation on any Objective-C program.

Objective-C *relies* on C for these basic constructs, but it also gives you the *option* of inserting C++ code directly into your source files. To tell the compiler to interpret your source code as either C, C++, or Objective-C, all you have to do is change the file extension to `.mm`.



Languages available to files with `.m` *and* `.mm` *extensions*

This unique language feature opens the door to the entire C/C++ ecosystem, which is a huge boon to Objective-C developers. For example, if you're building an iOS game and find yourself in need of a physics engine, you can leverage the well-known Box2D library (written in C++) with virtually no additional work.

The next module will complete our discussion of C with a brief look at functions. After that, we'll be more than ready to start working with Objective-C classes, methods, protocols, and the rest of its object-oriented goodness.

Continue to *Functions* ›