C Primitives

The vast majority of Objective-C's primitive data types are adopted from C, although it does define a few of its own to facilitate its object-oriented capabilities. The first part of this module provides a practical introduction to C's data types, and the second part covers three more primitives that are specific to Objective-C.

The examples in this module use `NSLog()` to inspect variables. In order for them to display correctly, you need to use the correct format specifier in `NSLog()`'s first argument. The various specifiers for C primitives are presented alongside the data types themselves, and all Objective-C objects can be displayed with the `%@` specifier.

The void Type

The `void` type is C's empty data type. Its most common use case is to specify the return type for functions that don't return anything. For example:

```
void sayHello() {
    NSLog(@"This function doesn't return anything");
}
```

The `void` type is not to be confused with the void *pointer*. The former indicates the *absence* of a value, while the latter represents *any* value (well, any pointer, at least).

Integer Types

Integer data types are characterized by their size and whether they are signed or unsigned. The `char` type is always 1 byte, but it's very important to understand that the exact size of the other integer types is implementation-dependent. Instead of being defined as an *absolute* number of bytes, they are defined relative to each other. The only guarantee is that `short <= int <= long <= long long`; however it is possible to determine their exact sizes at runtime.

C was designed to work closely with the underlying architecture, and different systems support different variable sizes. A relative definition provides the flexibility to, for example, define `short`, `int`, and `long` as the same number of bytes when the target chipset can't differentiate between them.

```
BOOL isBool = YES;
NSLog(@"%d", isBool);
NSLog(@"%@", isBool ? @"YES" : @"NO");

char aChar = 'a';
unsigned char anUnsignedChar = 255;
NSLog(@"The letter %c is ASCII number %hhd", aChar, aChar);
NSLog(@"%hhu", anUnsignedChar);

short aShort = -32768;
unsigned short anUnsignedShort = 65535;
NSLog(@"%hd", aShort);
NSLog(@"%hu", anUnsignedShort);

int anInt = -2147483648;
unsigned int anUnsignedInt = 4294967295;
NSLog(@"%d", anInt);
NSLog(@"%u", anUnsignedInt);

long aLong = -9223372036854775808;
unsigned long anUnsignedLong = 18446744073709551615;
NSLog(@"%ld", aLong);
NSLog(@"%lu", anUnsignedLong);

long long aLongLong = -9223372036854775808;
unsigned long long anUnsignedLongLong = 18446744073709551615;
NSLog(@"%lld", aLongLong);
NSLog(@"%llu", anUnsignedLongLong);
```

The `%d` and `%u` characters are the core specifiers for displaying signed and unsigned integers, respectively. The `hh`, `h`, `l` and `ll` characters are modifiers that tell `NSLog()` to treat the associated integer as a `char`, `short`, `long`, or `long long`, respectively.

It's also worth noting that the `BOOL` type is actually part of Objective-C, not C. Objective-C uses `YES` and `NO` for its Boolean values instead of the `true` and `false` macros used by C.

Fixed-Width Integers

While the basic types presented above are satisfactory for most purposes, it is sometimes necessary to declare a variable that stores a specific number of bytes. This is particularly relevant for algorithms like `arc4random()` that operate on a fixed-width integer.

The `int<n>_t` data types allow you to represent signed and unsigned integers that are exactly 1, 2, 4, or 8 bytes, and the `int_least<n>_t` variants let you constrain the *minimum* size of a variable without specifying an exact number of bytes. In addition, `intmax_t` is an alias for the largest integer type that the system can handle.

```
// Exact integer types
int8_t aOneByteInt = 127;
uint8_t aOneByteUnsignedInt = 255;
int16_t aTwoByteInt = 32767;
uint16_t aTwoByteUnsignedInt = 65535;
int32_t aFourByteInt = 2147483647;
uint32_t aFourByteUnsignedInt = 4294967295;
int64_t anEightByteInt = 9223372036854775807;
uint64_t anEightByteUnsignedInt = 18446744073709551615;

// Minimum integer types
int_least8_t aTinyInt = 127;
uint_least8_t aTinyUnsignedInt = 255;
int_least16_t aMediumInt = 32767;
uint_least16_t aMediumUnsignedInt = 65535;
int_least32_t aNormalInt = 2147483647;
uint_least32_t aNormalUnsignedInt = 4294967295;
int_least64_t aBigInt = 9223372036854775807;
uint_least64_t aBigUnsignedInt = 18446744073709551615;

// The largest supported integer type
```

```
intmax_t theBiggestInt = 9223372036854775807;
uintmax_t theBiggestUnsignedInt = 18446744073709551615;
```

Floating-Point Types

C provides three floating-point types. Like the integer data types, they are defined as relative sizes, where `float <= double <= long double`. Literal decimal values are represented as doubles—floats must be explicitly marked with a trailing `f`, and long doubles must be marked with an `L`, as shown below.

```
// Single precision floating-point
float aFloat = -21.09f;
NSLog(@"%f", aFloat);
NSLog(@"%8.2f", aFloat);

// Double precision floating-point
double aDouble = -21.09;
NSLog(@"%8.2f", aDouble);
NSLog(@"%e", aDouble);

// Extended precision floating-point
long double aLongDouble = -21.09e8L;
NSLog(@"%Lf", aLongDouble);
NSLog(@"%Le", aLongDouble);
```

The `%f` format specifier is used to display floats and doubles as decimal values, and the `%8.2f` syntax determines the padding and the number of points after the decimal. In this case, we pad the output to fill 8 digits and display 2 decimal places. Alternatively, you can format the value as scientific notation with the `%e` specifier. Long doubles require the `L` modifier (similar to `hh`, `l`, etc).

Determining Type Sizes

It's possible to determine the exact size of any data type by passing it to the `sizeof()` function, which returns the number of bytes used to represent the specified type. Running the following snippet is an easy way to see the size of the basic data types on any given architecture.

```
NSLog(@"Size of char: %zu", sizeof(char));   // This will always be 1
NSLog(@"Size of short: %zu", sizeof(short));
NSLog(@"Size of int: %zu", sizeof(int));
NSLog(@"Size of long: %zu", sizeof(long));
NSLog(@"Size of long long: %zu", sizeof(long long));
NSLog(@"Size of float: %zu", sizeof(float));
NSLog(@"Size of double: %zu", sizeof(double));
NSLog(@"Size of size_t: %zu", sizeof(size_t));
```

Note that `sizeof()` can also be used with an array, in which case it returns the number of bytes used by the array. This presents a new problem: the programmer has no idea which data type is required to store the maximum size of an array. Instead of forcing you to guess, the `sizeof()` function returns a special data type called `size_t`. This is why we used the `%zu` format specifier in the above example.

The `size_t` type is dedicated solely to representing memory-related values, and it is guaranteed to be able to store the maximum size of an array. Aside from being the return type for `sizeof()` and other memory utilities, this makes `size_t` an appropriate data type for storing the indices of very large arrays. As with any other type, you can pass it to `sizeof()` to get its exact size at runtime, as shown in the above example.

If your Objective-C programs interact with a lot of C libraries, you're likely to encounter the following application of `sizeof()`:

```
size_t numberOfElements = sizeof(anArray)/sizeof(anArray[0]);
```

This is the canonical way to determine the number of elements in a primitive C array. It simply divides the size of the array, `sizeof(anArray)`, by the size of each element, `sizeof(anArray[0])`.

Limit Macros

While it's trivial to determine the potential range of an integer type once you know how how many bytes it is, C implementations provide convenient macros for accessing the minimum and maximum values that each type can represent:

```
NSLog(@"Smallest signed char: %d", SCHAR_MIN);
NSLog(@"Largest signed char: %d", SCHAR_MAX);
NSLog(@"Largest unsigned char: %u", UCHAR_MAX);

NSLog(@"Smallest signed short: %d", SHRT_MIN);
NSLog(@"Largest signed short: %d", SHRT_MAX);
NSLog(@"Largest unsigned short: %u", USHRT_MAX);

NSLog(@"Smallest signed int: %d", INT_MIN);
NSLog(@"Largest signed int: %d", INT_MAX);
NSLog(@"Largest unsigned int: %u", UINT_MAX);

NSLog(@"Smallest signed long: %ld", LONG_MIN);
NSLog(@"Largest signed long: %ld", LONG_MAX);
NSLog(@"Largest unsigned long: %lu", ULONG_MAX);

NSLog(@"Smallest signed long long: %lld", LLONG_MIN);
NSLog(@"Largest signed long long: %lld", LLONG_MAX);
NSLog(@"Largest unsigned long long: %llu", ULLONG_MAX);

NSLog(@"Smallest float: %e", FLT_MIN);
NSLog(@"Largest float: %e", FLT_MAX);
NSLog(@"Smallest double: %e", DBL_MIN);
NSLog(@"Largest double: %e", DBL_MAX);

NSLog(@"Largest possible array index: %llu", SIZE_MAX);
```

The `SIZE_MAX` macro defines the maximum value that can be stored in a `size_t` variable.

Working With C Primitives

This section takes a look at some common "gotchas" when working with C's primitive data types. Keep in mind that these are merely brief overviews of computational topics that often involve a great deal of subtlety.

Choosing an Integer Type

The variety of integer types offered by C can make it hard to know which one to use in any given situation, but the answer is quite simple: use `int`'s unless you have a compelling reason not to.

Traditionally, an `int` is defined to be the native word size of the underlying architecture, so it's (generally) the most efficient integer type. The only reason to use a `short` is when you want to reduce the memory footprint of very large arrays (e.g., an OpenGL index buffer). The `long` types should only be used when you need to store values that don't fit into an `int`.

Integer Division

Like most programming languages, C differentiates between integer and floating-point operations. If both operands are integers, the calculation uses integer arithmetic, but if at least one of them is a floating-point type, it uses floating-point arithmetic. This is important to keep in mind for division:

```
int integerResult = 5 / 4;
NSLog(@"Integer division: %d", integerResult);        // 1
double doubleResult = 5.0 / 4;
NSLog(@"Floating-point division: %f", doubleResult);  // 1.25
```

Note that the decimal will always be truncated when dividing two integers, so be sure to use (or cast to) a `float` or a `double` if you need the remainder.

Floating-Point Equality

Floating-point numbers are inherently *not* precise, and certain values simply cannot be represented as a floating-point value. For example, we can inspect the imprecision of the number `0.1` by displaying several decimal places:

```
NSLog(@"%.17f", .1);        // 0.10000000000000001
```

As you can see, `0.1` is not actually represented as `0.1` by your computer. That extra 1 in the 17th digit occurs because converting 1/10 to binary results in a repeating decimal. Of course, a computer cannot store the infinitely many digits required for the exact value, so this introduces a rounding error. The error gets magnified when you start doing calculations with the value, resulting in counterintuitive situations like the following:

```
NSLog(@"%.17f", 4.2 - 4.1); // 0.10000000000000053
if (4.2 - 4.1 == .1) {
    NSLog(@"This math is perfect!");
} else {
    // You'll see this message
    NSLog(@"This math is just a tiny bit off...");
}
```

The lesson here is: don't try to check if two floating-point values are exactly equal, and definitely don't use a floating-point type to store precision-sensitive data (e.g., monetary values). To represent exact quantities, you should use the fixed-point `NSDecimalNumber` class.

For a comprehensive discussion of the issues surrounding floating-point math, please see What Every Computer Scientist Should Know About Floating-Point Arithmetic by David Goldberg.

Objective-C Primitives

In addition to the data types discussed above, Objective-C defines three of its own primitive types: `id`, `Class`, and `SEL`. These are the basis for Objective-C's dynamic typing capabilities. This section also introduces the anomalous `NSInteger` and `NSUInteger` types.

The id Type

The `id` type is the generic type for all Objective-C objects. You can think of it as the object-oriented version of C's void pointer. And, like a void pointer, it can store a reference to *any* type of object. The following example uses the same `id` variable to hold a string and a dictionary.

```
id mysteryObject = @"An NSString object";
NSLog(@"%@", [mysteryObject description]);
mysteryObject = @{@"model": @"Ford", @"year": @1967};
NSLog(@"%@", [mysteryObject description]);
```

Recall that all Objective-C objects are referenced as pointers, so when they are statically typed, they must be declared with pointer notation: `NSString *mysteryObject`. However, the `id` type automatically implies that the variable is a pointer, so this is not necessary: `id mysteryObject` (without the asterisk).

The Class Type

Objective-C classes are represented as objects themselves, using a special data type called `Class`. This lets you, for example, dynamically check an object's type at runtime:

```
Class targetClass = [NSString class];
id mysteryObject = @"An NSString object";
if ([mysteryObject isKindOfClass:targetClass]) {
    NSLog(@"Yup! That's an instance of the target class");
}
```

All classes implement a class-level method called `class` that returns its associated class object (apologies for the redundant terminology). This object can be used for introspection, which we see with the `isKindOfClass:` method above.

The SEL Type

The `SEL` data type is used to store selectors, which are Objective-C's internal representation of a method name. For example, the following snippet stores a method called `sayHello` in the `someMethod` variable. This variable could be used to dynamically call a method at runtime.

```
SEL someMethod = @selector(sayHello);
```

Please refer to the Methods module for a thorough discussion of Objective-C's selectors.

NSInteger and NSUInteger

While they aren't technically native Objective-C types, this is a good time to discuss the Foundation Framework's custom integers, `NSInteger` and `NSUInteger`. On 32-bit systems, these are defined to be 32-bit signed/unsigned integers, respectively, and on 64-bit systems, they are 64-bit integers. In other words, they are guaranteed to be the natural word size on any given architecture.

The original purpose of these Apple-specific types was to facilitate the transition from 32-bit architectures to 64-bit, but it's up to you whether or not you want to use `NSInteger` over the basic types (`int`, `long`, `long long`) and the `int<n>_t` variants. A sensible convention is to use `NSInteger` and `NSUInteger` when interacting with Apple's APIs and use the standard C types for everything else.

Either way, it's still important to understand what `NSInteger` and `NSUInteger` represent, as they are used extensively in Foundation, UIKit, and several other frameworks.

Continue to *NSNumber* ›

- © 2012-2013 RyPress.com
- All Rights Reserved
- Terms of Service