Binary Data Programming Guide

Developer

Contents

Listings

Introduction to Binary Data Programming Guide for Cocoa

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

Binary data can be wrapped inside of Foundation and Core Foundation data objects which provides object-oriented behaviors for manipulating the data. Because data objects are bridged objects, you can use the Foundation and Core Foundation data objects interchangeably. Data objects can manage the allocation and deallocation of byte buffers automatically. Among other things, data objects can be stored in collections, written to property lists, saved to files, and transmitted over communication ports.

Organization of This Document

The following article explains how data objects work:

- Data Objects (page 5) describes how data objects are used as wrappers for byte buffers.

The following articles cover common tasks:

- Working With Binary Data (page 6) explains how to create and use binary data objects.
- Working With Mutable Binary Data (page 9) explains how to modify the bytes in mutable binary data objects.

Data Objects

Data objects are object-oriented wrappers for byte buffers. In these data objects, simple allocated buffers (that is, data with no embedded pointers) take on the behavior of other objects—that is, they encapsulate data and provide operations to manipulate that data. Data objects are typically used to store data. They are also useful in internet and intranet applications because the data contained in data objects can be copied or moved between applications.

> **Important:** The Cocoa Foundation classes, `NSData` and `NSMutableData`, are "toll-free bridged" with their Core Foundation counterparts, `CFData` (see *CFData Reference*) and `CFMutableData` (see *CFMutableData Reference*). This means that the Core Foundation opaque type is interchangeable in function or method calls with the bridged Foundation object. In other words, in an API having an `NSData *` parameter, you can pass in a `CFDataRef`, and in an API having a `CFDataRef` parameter, you can pass in an `NSData` instance. You cannot, however, pass an `NSData` object to an API that expects a mutable `CFData` reference; you must use an `NSMutableData` object instead. This document refers to these objects as simply *data objects* or *mutable data objects* for objects that can be changed after creation.

The size of the data that an instance of `NSData` or `NSMutableData` can wrap is subject to platform-dependent limitations—see *NSData Class Reference*. When the data size is more than a few memory pages, the object uses virtual memory management. A data object can also wrap preexisting data, regardless of how the data was allocated. The object contains no information about the data itself (such as its type); the responsibility for deciding how to use the data lies with the client. In particular, it will not handle byte-order swapping when distributed between big-endian and little-endian machines. Instead, use `NSValue` for typed data.

Data objects provide an operating system–independent way to benefit from copy-on-write memory. The copy-on-write technique means that when data is copied through a virtual memory copy, an actual copy of the data is not made until there is an attempt to modify it.

Typically, you specify the bytes and the length of the bytes stored in a data object when creating that object. You can also extract bytes of a given range from a data object, compare data stored in two data objects, and write data to a URL. You use mutable data objects when you need to modify the data after creation. You can truncate, extend the length of, append data to, and replace a range of bytes in a mutable data object.

Working With Binary Data

This article contains code examples of common tasks that apply to both immutable and mutable data objects, `NSData` and `NSMutableData` objects. Because of the nature of class clusters in Foundation, data objects are not actual instances of the `NSData` or `NSMutableData` classes but instead are instances of one of their private subclasses. Although a data object's class is private, its interface is public, as declared by these abstract superclasses, `NSData` and `NSMutableData`.

Creating Data Objects From Raw Bytes

Generally, you create a data object from raw bytes using one of the `data...` class messages to either the `NSData` or `NSMutableData` class object. These methods return a data object containing the bytes you specify.

Typically, the creation methods (such as `dataWithBytes:length:`) make a copy of the bytes you pass as an argument. In this case, the copied bytes are owned by the data object and are freed when the data object is released. It is your responsibility to free the original bytes.

However, if you create an `NSData` object with one of the methods whose name includes `NoCopy` (such as `dataWithBytesNoCopy:length:`), the bytes are not copied. Instead, the data object takes ownership of the bytes passed in as an argument and frees them when the object is released. (`NSMutableData` responds to these methods, too, but the bytes are copied anyway and the buffer is freed immediately.) For this reason, the bytes you pass to the `NoCopy` methods must have been allocated using `malloc`.

If you prefer that the bytes not be copied or freed when the object is released, you can use the `dataWithBytesNoCopy:length:freeWhenDone:` or `initWithBytesNoCopy:length:freeWhenDone:` methods passing `NO` as the `freeWhenDone:` argument.

Creating Data Objects From Files or URLs

You use the `dataWithContentsOfFile:` or `dataWithContentsOfURL:` class methods to create a data object containing the contents of a file or URL. The following code example creates a data object, `myData`, initialized with the contents of `myFile.txt`. The path must be absolute.

```
NSString *thePath = @"/u/smith/myFile.txt";
```

```
NSData *myData = [NSData dataWithContentsOfFile:thePath];
```

Accessing and Comparing Bytes

The two `NSData` primitive methods—`bytes` and `length`—provide the basis for all other methods in the class. The `bytes` method returns a pointer to the bytes contained in the data object. The `length` method returns the number of bytes contained in the data object.

`NSData` provides access methods for copying bytes from a data object into a specified buffer. The `getBytes:length:` method copies bytes into a buffer. For example, the following code fragment initializes a data object, `myData`, with the string `myString`. It then uses `getBytes:length:` to copy the contents of `myData` into `aBuffer`.

```
unsigned char aBuffer[20];
NSString *myString = @"Test string.";
const char *utfString = [myString UTF8String];
NSData *myData = [NSData dataWithBytes: utfString length: strlen(utfString)];

[myData getBytes:aBuffer length:20];
```

The `getBytes:range:` method copies a range of bytes from a starting point within the bytes themselves.

To extract a data object that contains a subset of the bytes in another data object, use the `subdataWithRange:` method. For example, the following code fragment initializes a data object, `data2`, to contain a subrange of `data1`:

```
NSString *myString = @"ABCDEFG";
const char *utfString = [myString UTF8String];
NSRange range = {2, 4};
NSData *data1, *data2;

data1 = [NSData dataWithBytes:utfString length:strlen(utfString)];

data2 = [data1 subdataWithRange:range];
```

To determine if two data objects are equal, use the `isEqualToData:` method, which does a byte-for-byte comparison.

Copying Data Objects

You can copy data objects to create a read-only copy or to create a mutable copy. `NSData` and `NSMutableData` adopt the `NSCopying` and `NSMutableCopying` protocols, making it convenient to convert between efficient, read-only data objects and mutable data objects. You use `copy` to create a read-only copy, and `mutableCopy` to create a mutable copy.

Saving Data Objects

You can save data objects to a local file or to the internet. The `writeToFile:atomically:` and `writeToURL:atomically:` methods let you write the contents of a data object to a local file.

Working With Mutable Binary Data

This article contains code examples of common tasks that apply specifically to mutable data objects, `NSMutableData` objects. Basically, you can change the bytes in a mutable binary data object by getting the byte array to modify directly, appending bytes to them, or replacing a range of bytes.

Modifying Bytes

The two `NSMutableData` primitive methods—`mutableBytes` and `setLength:`—provide the basis for all other methods in the class. The `mutableBytes` method returns a pointer for writing into the bytes contained in the mutable data object. The `setLength:` method allows you to truncate or extend the length of a mutable data object. The `increaseLengthBy:` method also allows you to change the length of a mutable data object.

In Listing 1 (page 9), `mutableBytes` is used to return a pointer to the bytes in `data2`. The bytes in `data2` are then overwritten with the contents of `data1`.

Listing 1 Modifying bytes

```
NSMutableData *data1, *data2;
NSString *myString = @"string for data1";
NSString *yourString = @"string for data2";
const char *utfMyString = [myString UTF8String];
const char *utfYourString = [yourString UTF8String];
unsigned char *firstBuffer, secondBuffer[20];

/* initialize data1, data2, and secondBuffer... */
data1 = [NSMutableData dataWithBytes:utfMyString length:strlen(utfMyString)+1];
data2 = [NSMutableData dataWithBytes:utfYourString length:strlen(utfYourString)+1];

[data2 getBytes:secondBuffer length:20];
NSLog(@"data2 before: \"%s\"\n", (char *)secondBuffer);

firstBuffer = [data2 mutableBytes];
```

```
[data1 getBytes:firstBuffer length:[data2 length]];
NSLog(@"data1: \"%s\"\n", (char *)firstBuffer);

[data2 getBytes:secondBuffer length:20];
NSLog(@"data2 after: \"%s\"\n", (char *)secondBuffer);
```

This is the output from Listing 1 (page 9):

```
Oct  3 15:59:51 [1113] data2 before: "string for data2"
Oct  3 15:59:51 [1113] data1: "string for data1"
Oct  3 15:59:51 [1113] data2 after: "string for data1"
```

Appending Bytes

The `appendBytes:length:` and `appendData:` methods let you append bytes or the contents of another data object to a mutable data object. For example, Listing 2 (page 10) copies the bytes in `data2` into `aBuffer`, and then appends `aBuffer` to `data1`:

Listing 2 Appending bytes

```
NSMutableData *data1, *data2;
NSString *firstString  = @"ABCD";
NSString *secondString = @"EFGH";
const char *utfFirstString = [firstString UTF8String];
const char *utfSecondString = [secondString UTF8String];
unsigned char *aBuffer;
unsigned len;

data1 = [NSMutableData dataWithBytes:utfFirstString length:strlen(utfFirstString)];
data2 = [NSMutableData dataWithBytes:utfSecondString length:strlen(utfSecondString)];

len = [data2 length];
aBuffer = malloc(len);

[data2 getBytes:aBuffer length:[data2 length]];
```

```
[data1 appendBytes:aBuffer length:len];
```

The final value of `data1` is the series of ASCII characters `"ABCDEFGH"`.

Replacing Bytes

You can replace a range of bytes in a mutable data object with zeros using the `resetBytesInRange:` method, or with different bytes using the `replaceBytesInRange:withBytes:` method. In Listing 3 (page 11), a range of bytes in `data1` is replaced by the bytes in `data2`, and the content of `data1` changes from “Liz and John” to “Liz and Larry”:

Listing 3 Replacing bytes

```
NSMutableData *data1, *data2;
NSString *myString = @"Liz and John";
NSString *yourString = @"Larry";
const char *utfMyString = [myString UTF8String];
const char *utfYourString = [yourString UTF8String];
unsigned len;
unsigned char *aBuffer;
NSRange range = {8, strlen(utfYourString)};

data1 = [NSMutableData dataWithBytes:utfMyString length:strlen(utfMyString)];
data2 = [NSMutableData dataWithBytes:utfYourString length:strlen(utfYourString)];

len = [data2 length];
aBuffer = malloc(len);
[data2 getBytes:aBuffer length:len];
[data1 replaceBytesInRange:range withBytes:aBuffer];
```

Document Revision History

This table describes the changes to *Binary Data Programming Guide*.

Date	Notes
2013-01-28	Updated code listings to use getBytes:length: instead of getBytes:.
2009-08-06	Added links to Cocoa Core Competencies.
2009-05-06	Corrected the code listing under Modifying Bytes to account for the null terminator on the strings.
2007-03-06	Clarified note about limits on data size using NSData.
2006-01-10	Changed title from "Binary Data."
2003-10-27	Corrected results from sample code in Working With Mutable Binary Data (page 9).
2003-08-07	Revised content and added more code examples.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010