String Programming Guide for Core Foundation

Developer

Contents

Figures, Tables, and Listings

Introduction to Strings Programming Guide for Core Foundation

Core Foundation string objects give software developers a solid foundation for easy, robust, and consistent internationalization. String objects offer a full suite of fast and efficient string functionality, including utilities for converting among various encodings and buffer formats.

Read this document to learn how to use Core Foundation strings. If you are writing a program using Objective-C, you should consider using `NSString` objects instead (see *String Programming Guide*).

Organization of This Document

This document contains the following articles:

- About Strings (page 8) describes issues related to managing and representing string
- The Unicode Basis of CFString Objects (page 10) describes the conceptual basis for the representation of strings in Core Foundation
- String Storage (page 12) describes how string data is stored in Core Foundation
- Creating and Copying Strings (page 14) describes how to create and copy string objects
- Accessing the Contents of String Objects (page 21) describes how to access the contents of CFString objects as a C or Unicode string, and how to iterate over the contents of a string one character at a time
- Comparing, Sorting, and Searching String Objects (page 25) describes how to search the contents of a string and how to compare two strings
- Manipulating Mutable String Objects (page 29) describes operations such as combining strings and padding the contents of a string.
- Converting Between String Encodings (page 32) describes how to convert between different string encodings, and what encodings are supported by CFString
- Handling External Representations of Strings (page 36) describes how to represent a string in a form that can be written to disk and read back in on the same platform or on a different platform
- Creating and Using Ranges (page 39) describes how to create and use CFRange structures
- Character Sets (page 40) describes the basics of CFCharacterSet
- String Format Specifiers (page 41) describes `printf`-style format specifiers supported by CFString

Not all functions are described. Some of the functions not discussed in detail are:

- `CFStringGetLength` lets you obtain the number of Unicode characters represented by a `CFString` object.
- `CFStringGetLineBounds` tells you how many lines a string (or a range of the string) spans.
- `CFStringCreateByCombiningStrings` creates a single string from an array (`CFArray`) of strings; the counterpart of this function, `CFStringCreateArrayBySeparatingStrings`, creates a `CFArray` object from a single string, using a delimiter character to separate the substrings.
- `CFStringGetIntValue` and `CFStringGetDoubleValue` convert a `CFString` object representing a number to the actual numeric value.

About Strings

One of the biggest challenges of developing software for a global market is that posed by text—or, in programming terms, "strings," which denotes the characters of a language in a form suitable for computerized representation. Most of the difficulty with strings is historical; over the years (since computers have been around), various encoding schemes have been devised to represent strings in one script or another. Some encodings are intended for a language or family of languages (Shift-JIS, for example) while others are specific to a particular computer system (Windows Latin 1, for example). The proliferation of encodings complicates the burdens of cross-platform compatibility and internationalization.

Core Foundation string objects give software developers a solid foundation for easy, robust, and consistent internationalization. String objects offers a full suite of fast and efficient string functionality, including utilities for converting among various encodings and buffer formats.

String objects are implemented by the `CFString` opaque type. A `CFString` "object" represents a string as an array of Unicode characters; its only other property aside from this array is an integer indicating the number of characters. It is flexible enough to hold up to several megabytes worth of characters. Yet it is simple and fundamental enough for use in all programming interfaces that communicate character data. In Core Foundation, string operations take place with performance characteristics not much different from standard C strings. `CFString` objects come in immutable and mutable variants.

The Unicode basis of `CFString` along with comprehensive encoding-conversion facilities make string objects an essential vehicle for internationalizing programs. String objects also allow you to convert strings among C, byte buffer, and native Unicode buffer formats. Taken together, these features make it possible for programs to pass each other string data despite differing programming languages, libraries, frameworks, or platforms.

String objects also includes the `CFCharacterSet` opaque type. Programming interfaces can use `CFCharacterSet` objects to specify characters to include or exclude in parsing, comparison, or search operations.

`CFString` objects are fundamental in that they represent strings but they do not carry any display or supplemental information, such as text styles, formatting attributes, or language tags. If you want this functionality, use an attributed string (see *CFAttributedString Reference*). In addition, a `CFString` object cannot be used to hold random bytes because it attaches semantic value to its contents (interpreting it as Unicode characters or even characters in other encodings). If you need a Core Foundation object to hold non-character data, use an object based on the `CFData` opaque type (see *CFData Reference*).

String objects provide functions that perform a variety of operations with `CFString` objects, such as

- Converting between `CFString` objects and strings in other encodings and buffer formats
- Comparing and searching strings
- Manipulating mutable strings by appending string data to them, inserting and deleting characters, and padding and trimming characters
- Disclosing the contents of `CFString` objects in supported debuggers

`CFString` and other Core Foundation objects do not provide more advanced string-handling utilities such as drawing, text layout, font handling, and sophisticated search and comparison functionality. Higher software layers provide these facilities. Nonetheless, these higher layers communicate string data using `CFString` objects, or their "toll-free bridged" Cocoa equivalent, `NSString`.

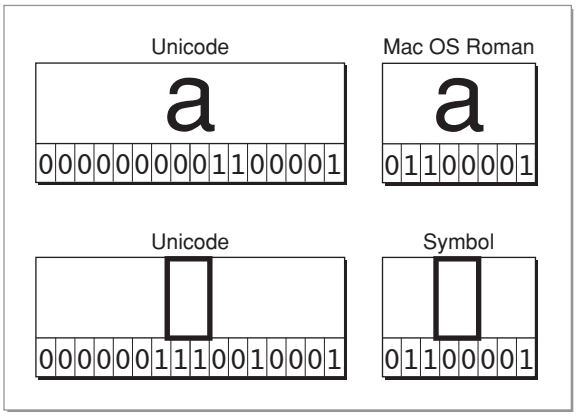
The Unicode Basis of CFString Objects

Conceptually, a CFString object represents an array of 16-bit Unicode characters (`UniChar`) along with a count of the number of characters. Unicode-based strings in Core Foundation provide a solid basis for internationalizing the software you develop. Unicode makes it possible to develop and localize a single version of an application for users who speak most of the world's written languages, including Russian (Cyrillic), Arabic, Chinese, and Japanese.

The Unicode standard is published by the Unicode Consortium (http://www.unicode.org), an international standards organization. The standard defines three encoding forms (UTF-8, UTF-16, UTF-32) that use a common repertoire of characters and allow for encoding as many as a million characters. This is sufficient for all known character encoding requirements. A "character" in this scheme is the smallest useful element of text in a language; thus it can be a character as understood in most European languages, an ideogram (Chinese Han), a syllable (Japanese hiragana), or some other linguistic unit. Encoded characters also include mathematical, technical, and other symbols as well as diacritics and computer control characters. Each Unicode character is represented by a "code point" having a glyph, a name, and a unique numeric value.

With UTF-16 (16-bit) encoding, Unicode makes over 65,000 code points possible. This capacity is in marked contrast to standard 8-bit encodings, which permit only 256 characters and thus necessitate elaborate ancillary schemes, such as shift or escape bits, to express characters other than those found in the common Indo-European scripts. All the heavily used characters fit into a single 16-bit code unit, while all other characters are accessible via pairs of 16-bit code units called surrogate pairs. A surrogate pair is a sequence of two UTF-16 units, taken from specific reserved ranges, that together represent a single Unicode code point. CFString has functions for converting between surrogate pairs and the UTF-32 representation of the corresponding Unicode code point.

Figure 1 Unicode versus other encodings of the same characters



In addition to its encoding scheme, the Unicode standard specifies mappings from the Unicode scheme to repertoires of international, national, and industry character sets. Figure 1 (page 10) illustrates two of these mappings. String objects make frequent use of the encoding mappings. The underlying representation (and in many cases the underlying storage) of strings is Unicode-based. However, the encodings required by the programming interfaces and output devices that actually display the strings in the user interface are commonly 8-bit. Thus there is a need for efficient and accurate conversion between Unicode and other encodings. String objects have functions that purpose, as described in Converting Between String Encodings (page 32).

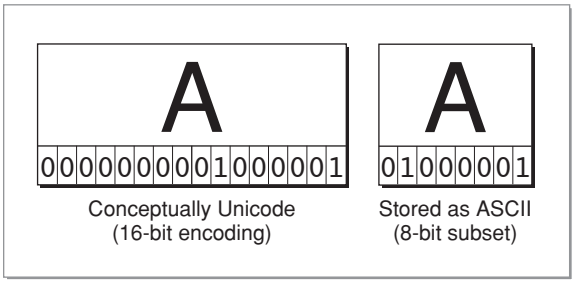
For more information on the Unicode standard, see the consortium's website. The consortium also publishes charts of Unicode code points and glyphs at www.unicode.org/charts/.

String Storage

Although *conceptually* CFString objects store strings as arrays of Unicode characters, in practice they often store them more efficiently. The memory a CFString object requires to represent a string could often be less than that required by a simple `UniChar` array.

For immutable strings, this efficiency is possible because some standard 8-bit encodings of a character value—namely ASCII and related encodings such as ISO Latin-1—are subsets of the 16-bit Unicode representation of the same value. With ASCII character values in the Unicode scheme, for example, the left most eight bits are zeros; the right most eight bits are identical to those in the 8-bit encoding. String objects only attempts this compressed type of storage if the encoding allows fast (O(1)) conversion to Unicode characters.

Figure 1 Storage of an immutable CFString derived from ASCII encoding



Mutable CFString objects perform a similar type of optimization. For example, a mutable string might have 8-bit backing store until a character above the ASCII range is inserted.

CFString objects perform other "tricks" to conserve memory, such as incrementing the reference count when a CFString is copied. For larger strings, they might lazily load them from files or resources and store them internally in B-tree structures.

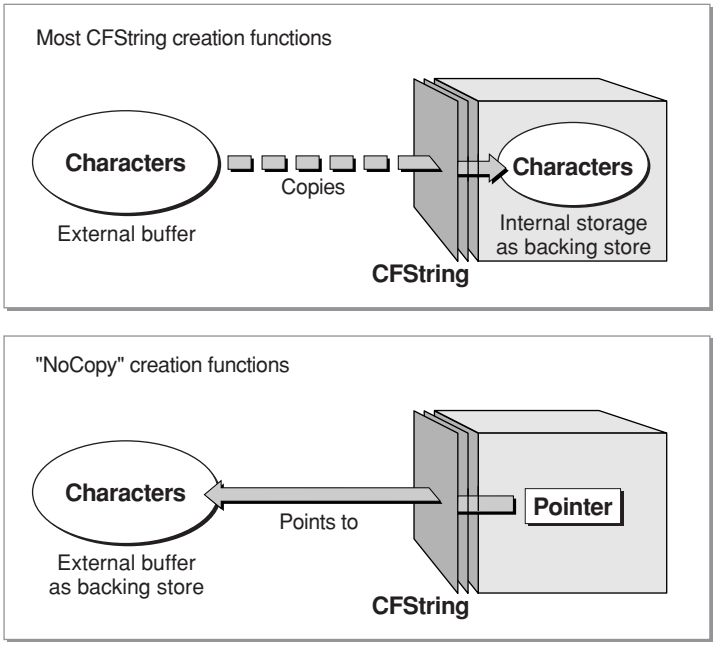
There is some memory overhead associated with CFString objects. It typically ranges from 4 to 12 bytes, depending on the mutability characteristic and the platform. But the memory-saving strategies employed by string objects more than compensate for this overhead.

In addition to its internal storage mechanisms, some of the programming interfaces of string objects grant you ownership of the string's backing store or give you quick access to it. Some functions of string objects fetch all stored characters into a local buffer or, for large strings, allow you to process characters efficiently in an in-line buffer.

Most CFString creation functions copy the string in the user-supplied buffer to the backing store of the created object. In some advance usage scenarios, you might find it useful to provide the backing store yourself. The creation functions containing `NoCopy` make the user's buffer the backing store and allow the created CFString object to point to it. (See Figure 2 (page 13) for an illustration of this.) The `NoCopy` qualifier, however, is just a "hint"; in some cases the CFString object might copy the buffer's contents to its internal storage.

You can get further control over the backing store of a string with the `CFStringCreateMutableWithExternalCharactersNoCopy` function. This function creates a reference to a mutable CFString object but allows you to retain full ownership of the Unicode buffer holding the object's characters; the object itself points to the buffer as its backing store. When you change the contents of the buffer you just need to notify the object. See Mutable Strings With Client-Owned Buffers (page 19) for more on this subject.

Figure 2 CFString objects and their backing stores



Creating and Copying Strings

String objects give you a variety of ways to create CFString objects—from constant strings, from buffers, from formatted strings, and by using existing CFString objects. The following sections describe each of these techniques.

Some functions that return references to CFString objects are described elsewhere. The `CFStringCreateWithBytes` function is described in Converting Between String Encodings (page 32). The section Handling External Representations of Strings (page 36) describes the `CFStringCreateFromExternalRepresentation` function.

Creating CFString Objects From Constant Strings

The easiest way to create immutable CFString objects is to use the `CFSTR` macro. The argument of the macro must be a constant compile-time string—text enclosed in quotation marks. `CFSTR` returns a reference to a CFString object.

Here's an example:

```
CFStringRef hello = CFSTR("Hello, world.");
```

The returned CFString has the following semantics:

- Because `CFSTR` is not a Create or Copy function, you are not responsible for releasing the string when you no longer need it.
- The string is not released by CFString. In other words, CFString guarantees its validity until the program terminates.
- The string can be retained and released in a balanced fashion, like any other CFString.

If there are two or more exact instances of a constant string in an executable, in some cases only one might be stored. A common use of the `CFSTR` macro is in the creation of formatted strings (see Creating String Objects From Formatted Strings (page 16) for more information).

Creating CFString Objects From String Buffers

A common technique for creating a CFString object is to call functions that take C character buffers (or string pointers) as "source" for the object. These functions are the counterparts of functions that convert CFString objects to C strings; see Accessing the Contents of String Objects (page 21) for more on these functions.

These functions come in two varieties. One set of functions copies the buffer into the internal storage of the created CFString object. Once you create the object you are free to dispose of the buffer. Related functions create CFString objects from C string buffers (`CFStringCreateWithCString`) and from Unicode string buffers (`CFStringCreateWithCharacters`). The latter function takes an extra parameter for character count but does not include the encoding parameter.

A parallel set of functions have corresponding names that end with `NoCopy`. These functions also create CFString objects from a user-supplied string buffer but they do not always copy the buffer to the object's internal storage. They try to but are not guaranteed to take the provided pointer as-is, using the buffer as the backing store without copying the data. Obviously you must ensure that you do not free the buffer while the CFString exists. The character data should never be on the stack or be data with a lifetime you cannot guarantee.

In practice, these `NoCopy` functions are useful in a limited number of circumstances:

- You have compile-time constant data such as a C string ("Hello"). The `NoCopy` functions offer an efficient way to make CFString objects from this data and, if you specify `kCFAllocatorNull` as the last parameter (see below), when the CFString ceases to exist the buffer is not automatically deallocated. (Often you can use the `CFSTR` macro for the same purpose.)
- You allocate some memory for some string data and you want to put a CFString object in it but otherwise you don't need the original memory. Of course, you can create a CFString object with one of the non-`NoCopy` functions (which copies the data) and then free the buffer. But the `NoCopy` functions allow you to transfer ownership of the memory to a CFString object, saving you the need to free it yourself.

The `NoCopy` functions include an extra parameter (`contentsDeallocator`) for passing a reference to a CFAllocator object that is used for deallocating the buffer when it is no longer needed. If the default CFAllocator object is sufficient for this purpose, you can pass `NULL`. If you do not want the CFString object to deallocate the buffer, pass `kCFAllocatorNull`.

Listing 1 (page 15) shows the creation of a CFString object with the `CFStringCreateWithCStringNoCopy` function:

Listing 1 Creating a CFString object with a NoCopy function

```
const char *bytes;
CFStringRef str;
bytes = CFAllocatorAllocate(CFAllocatorGetDefault(), 6, 0);
```

```
strcpy(bytes, "Hello");
str = CFStringCreateWithCStringNoCopy(NULL, bytes,
    kCFStringEncodingMacRoman, NULL);
/* do something with str here...*/
CFRelease(str); /* default allocator also frees bytes */
```

Important: The CFString objects created by the `NoCopy` function do not necessarily use the buffer you supply. In some cases the object might free the buffer and use something else; for instance, it may decide to use Unicode encoding internally. This behavior may change from release to release.

You can also create mutable CFString objects with source buffers that you control entirely; see Mutable Strings With Client-Owned Buffers (page 19) for more on this matter.

Creating String Objects From Formatted Strings

String objects includes functions that create CFString objects from formatted strings—strings incorporating `printf`-style specifiers for substituting variable values into the string, after converting them (if necessary) to character data. String format specifiers are defined in String Format Specifiers (page 41). Formatted strings are useful when it is necessary to display information that may have changeable elements. For example, you might need to use these functions when you put up a dialog box to show the progress of an operation, such as "Copying file *x* of *y*."

The `CFStringCreateWithFormat` function creates a CFString object from a simple formatted string, as shown in Listing 2 (page 16).

Listing 2 Creating a CFString object from a formatted string

```
CFStringRef PrintGross(CFStringRef employeeName, UInt8 hours, float wage) {
    return CFStringCreateWithFormat(NULL, NULL, CFSTR("Employee %@
    earned $%.2f this week."), employeeName, hours * wage);
}
```

The first parameter, as usual, specifies the allocator object to use (`NULL` means use the default CFAllocator object). The second parameter is for locale-dependent format options, such as thousand and decimal separators; it is currently not used. The remaining parameters are for the format string and the variable values.

As mentioned earlier, the format string has `printf`-style specifiers embedded in it (for example, "`%d %s %2.2f`"). Core Foundation introduces a couple of extensions to this convention. One is the `%@` specifier (shown in Listing 2 (page 16)) which indicates any Core Foundation object. Another new specifier indicates argument order. This specifier takes the form *n* `$` where *n* is the order-number of the argument following the string. This argument-order feature is useful when you want to localize whole sentences or even paragraphs to other languages without worrying about the order of arguments, which might vary from one language to another.

For example, the function above would result in a string such as "John Doe earned $1012.32 this week." But in another language the grammatically proper way of expressing the same sentence might be (roughly translated) "$1012.32 was earned by John Doe this week." You wouldn't have to call `CFStringCreateWithFormat` again with the arguments in a different order. Instead, you would have a function call that looked like this:

```
return CFStringCreateWithFormat(NULL, NULL, CFSTR("$%2$.2f was earned by
    employee %1$@."), employeeName, hours * wage);
```

Of course, the string itself would not be hard-coded, but would be loaded from a file (for instance, an XML property list or an OpenStep ASCII property list) that contains localized strings and their translations.

Another CFString function, `CFStringCreateWithFormatAndArguments`, takes a variable argument list (`vararg`) as well as a format string. This function allows the formatting of `varargs` passed into your function. Listing 3 (page 17) shows how it might be used:

Listing 3 Creating a CFString from a variable argument list

```
void show(CFStringRef formatString, ...) {
    CFStringRef resultString;
    CFDataRef data;
    va_list argList;

    va_start(argList, formatString);
    resultString = CFStringCreateWithFormatAndArguments(NULL, NULL,
        formatString, argList);
    va_end(argList);

    data = CFStringCreateExternalRepresentation(NULL, resultString,
        kCFStringEncodingMacRoman, '?');

    if (data != NULL) {
```

```
        printf ("%.*s\n\n", (int)CFDataGetLength(data),
            CFDataGetBytePtr(data));
        CFRelease(data);
    }

    CFRelease(resultString);
}
```

Creating Mutable String Objects

String objects includes only a handful of functions for creating mutable CFString objects. The reason for this much smaller set is obvious. Because these are mutable objects, you can modify them after you create them with the functions described in Manipulating Mutable String Objects (page 29).

There are two basic functions for creating mutable CFString objects. The `CFStringCreateMutable` function creates an "empty" object; the `CFStringCreateMutableCopy` makes a mutable copy of an immutable CFString object. Listing 4 (page 18) illustrates the latter function and shows a character being appended to the created object:

Listing 4 Creating a mutable copy of a CFString object

```
const UniChar u[] = {'5', '+', '*', 'd', 'x', '4', 'Q', '?'};
CFMutableStringRef str;

str = CFStringCreateMutableCopy(alloc, 0, CFSTR("abc"));
CFStringAppendCharacters(str, &u[3], 1);
CFRelease(str);
```

The second parameter of both functions is a `CFIndex` value named `maxLength`. This value specifies the maximum numbers of characters in the string and allows the created object to optimize its storage and catch errors if too many characters are inserted. If 0 is specified for this parameter (as above), the string can grow to any size.

Mutable Strings With Client-Owned Buffers

When you create most Core Foundation objects, the object takes the initializing data you provide and stores that data internally. String objects allow some exceptions to this behavior, and for mutable CFString objects that exception is the `CFStringCreateMutableWithExternalCharactersNoCopy` function. This function creates a mutable CFString object whose backing store is some Unicode buffer that you create and own. You can test and manipulate this buffer independently of the object.

Listing 5 (page 19) shows how to create such a cheap mutable CFString "wrapper" for your character buffer.

Listing 5 Creating a mutable CFString object with independent backing store

```
void stringWithExternalContentsExample(void) {
#define BufferSize 1000
    CFMutableStringRef mutStr;
    UniChar *myBuffer;

    myBuffer = malloc(BufferSize * sizeof(UniChar));

    mutStr = CFStringCreateMutableWithExternalCharactersNoCopy(NULL, myBuffer, 0,
 BufferSize, kCFAllocatorNull);
    CFStringAppend(mutStr, CFSTR("Appended string... "));
    CFStringAppend(mutStr, CFSTR("More stuff... "));
    CFStringAppendFormat(mutStr, NULL, CFSTR("%d %4.2f %@..."), 42, -3.14,
CFSTR("Hello"));

    CFRelease(mutStr);
    free(myBuffer);
}
```

The third and fourth parameters in the creation function specify the number of characters in the buffer and the buffer capacity. The final parameter, `externalCharsAllocator`, specifies the CFAllocator object to use for reallocating the buffer when editing takes place and for deallocating the buffer when the CFString object is deallocated. In the above example, `kCFAllocatorNull` is specified, which tells the object that the client assumes responsibility for these actions. If you specified an allocator object to use, such as `NULL` for the default allocator, there is usually no need to worry about reallocation or deallocation of the buffer.

The example illustrates how you can modify the contents of the buffer with CFString functions. You can also modify the contents of the buffer directly, but if you do so, you must notify the mutable CFString "wrapper" object with the `CFStringSetExternalCharactersNoCopy` function. You can also substitute an entirely

different buffer with this function because it makes the mutable CFString object point directly at the specified `UniChar` array as its backing store. (However, the CFString object must have been created with the `CFStringCreateMutableWithExternalCharactersNoCopy` function.) The `CFStringSetExternalCharactersNoCopy` function does not free the previous buffer.

Using these functions comes at a cost because some CFString optimizations are invalidated. For example, mutable CFString objects can no longer use a gap for editing, and they cannot optimize storage by using 8-bit characters.

Accessing the Contents of String Objects

The two essential properties of CFString objects are an array of Unicode characters and a count of those characters. Several CFString functions not only obtain those properties, particularly the characters, but perform conversions to almost any desired format.

The `CFStringGetBytes` function, which copies the contents of a CFString object into a client-supplied byte buffer, is described in The Basic Conversion Routines (page 33). It is described there instead of in this section because it has features that make it particularly suitable for encoding conversions.

Getting the Contents as a C String

You may need to use programming interfaces that require C strings for some of their parameters. For performance reasons, a common strategy for accessing the contents of CFStrings as a C string is to first try to get a pointer of the appropriate type to these strings and, if that fails, to copy the contents into a local buffer. Listing 1 (page 21) illustrates this strategy for C strings using the `CFStringGetCStringPtr` and `CFStringGetCString` functions.

Listing 1 Accessing CFString contents as a C string

```
CFStringRef str;
CFRange rangeToProcess;
const char *bytes;

str = CFStringCreateWithCString(NULL, "Hello World!", kCFStringEncodingMacRoman);

bytes = CFStringGetCStringPtr(str, kCFStringEncodingMacRoman);

if (bytes == NULL) {
    char localBuffer[10];
    Boolean success;
   success = CFStringGetCString(str, localBuffer, 10, kCFStringEncodingMacRoman);
}
```

These functions allow you to specify the encoding that the Unicode characters should be converted to. The functions that end with "Ptr" either return the desired pointer quickly, in constant time, or they return `NULL`. If the latter is the case, you should use `CFStringGetCString`.

The buffer for the `CFStringGetCString` functions can either be on the stack or a piece of allocated memory. These functions might still fail to get the characters, but that only happens in two circumstances: the conversion from the `UniChar` contents of CFString to the specified encoding fails or the buffer is too small. If you need a copy of the character buffer or if the code in question is not that performance-sensitive, you could simply call the `CFStringGetCString` function without even attempting to get the pointer first.

Getting the Contents as Unicode Strings

String objects offer a pair of functions similar to those for C strings for accessing the contents of a CFString as a 16-bit Unicode buffer: `CFStringGetCharactersPtr` and `CFStringGetCharacters`. The typical usage of these functions is also identical: you first optionally try to get a pointer to the characters and, if that fails, you try to copy the characters to a buffer you provide. These functions are different, however, in that they require a parameter specifying the length of the string.

Listing 2 (page 22) illustrates the common strategy for using these functions.

Listing 2 Accessing CFString contents as Unicode characters

```
CFStringRef str;
const UniChar *chars;

str = CFStringCreateWithCString(NULL, "Hello World", kCFStringEncodingMacRoman);
chars = CFStringGetCharactersPtr(str);
if (chars == NULL) {
    CFIndex length = CFStringGetLength(str);
    UniChar *buffer = malloc(length * sizeof(UniChar));
    CFStringGetCharacters(str, CFRangeMake(0, length), buffer);
    // Process the characters...
    free(buffer);
}
```

This example shows an allocated buffer (`malloc`) rather than a stack buffer. You can use one or the other. Because you need to know the size of the buffer for the `CFStringGetCharacters` function, allocating memory is easier to do but is less efficient. If you allocate memory for the characters you must, of course, free the buffer when you no longer need it.

Character Processing

Sometimes you might want to receive the contents of a CFString not as an entire block of characters but one Unicode character at a time. Perhaps you might be looking for a particular character or sequence of characters, such as special control characters indicating the start and end of a "record." String objects give you three ways to process Unicode characters.

The first way is to use the `CFStringGetCharacters` function described in Getting the Contents as Unicode Strings (page 22) to copy the characters to a local buffer and then cycle through the characters in the buffer. But this technique can be expensive memory-wise, especially if a large number of characters is involved.

The second way to access characters one at a time is to use the `CFStringGetCharacterAtIndex` function, as Listing 3 (page 23) illustrates.

Listing 3 Getting a character at a time

```
CFIndex length, i;
UniChar uchar;
CFStringRef str;

str = CFStringCreateWithCString(NULL, "Hello World", kCFStringEncodingMacRoman);
length = CFStringGetLength(str);
for (i=0; i < length; i++) {
    uchar = CFStringGetCharacterAtIndex(str, i);
    // Process character....
}
```

Although this function does not require a large chunk of memory to hold a block of characters, using it in a loop can be inefficient. For such cases, use the `CFStringGetCharacters` function instead.

The third technique for character processing, exemplified in Listing 4 (page 24), combines the convenience of one-at-a-time character access with the efficiency of bulk access. The in-line functions `CFStringInitInlineBuffer` and `CFStringGetCharacterFromInlineBuffer` give fast access to the contents of a string when you are doing sequential character processing. To use this programming interface,

call the `CFStringInitInlineBuffer` function with a `CFStringInlineBuffer` structure (on the stack, typically) and a range of the CFString's characters. Then call `CFStringGetCharacterFromInlineBuffer` as many times as you want using an index into that range relative to the start of the range. Because these are in-line functions they access the CFString object only periodically to fill the in-line buffer.

Listing 4 Processing characters in an in-line buffer

```
CFStringRef str;
CFStringInlineBuffer inlineBuffer;CFIndex length;CFIndex cnt;

str = CFStringCreateWithCString(NULL, "Hello World", kCFStringEncodingMacRoman);
length = CFStringGetLength(str)
CFStringInitInlineBuffer(str, &inlineBuffer, CFRangeMake(0, length));

for (cnt = 0; cnt < length; cnt++) {
     UniChar ch = CFStringGetCharacterFromInlineBuffer(&inlineBuffer, cnt);
     // Process character...
}
```

Comparing, Sorting, and Searching String Objects

Core Foundation string objects include a number of functions for searching the contents of strings and for comparing two strings. Because these operations are semantically related, it is not surprising that the main functions for each operation—`CFStringFindWithOptions` and `CFStringCompareWithOptions`—have some things in common. Their first four parameters are almost identical: two references to CFString objects (the strings to be compared or the substring to find in the main string), a range of characters to include in the operation, and a bitmask for specifying options. If you are sorting strings to present to the user, you should perform a localized comparison with the user's local using `CFStringCompareWithOptionsAndLocale`.

Comparing and Searching Strings

Although `CFStringFindWithOptions` and `CFStringCompareWithOptions` have features in common, they have important differences too. The `CFStringCompareWithOptions` function returns a result of type `Comparison Results`; this `enum` constant indicates whether the comparison found the strings equal or whether the first specified string was greater than or less than the second string. The `CFStringFindWithOptions` function, on the other hand, returns a `Boolean` result that indicates the success of the operation. The more useful result, returned indirectly by this function, is a range (a structure of type `CFRange`) pointed to by its final parameter; this range contains the location of the found string in the main string.

Listing 1 (page 25) illustrates the use of both `CFStringCompareWithOptions` and `CFStringFindWithOptions` (it also makes use of the `show` function given in Listing 2 (page 16) of Creating and Copying Strings (page 14)).

In this example, both the find and compare functions specify the `kCFCompareCaseInsensitive` flag as an option for the operation, causing it to ignore differences in case. Other option flags are available, including `kCFCompareBackwards` (start the operation from the end of the string), `kCFCompareNumerically` (compare similar strings containing numeric substrings numerically), and `kCFCompareLocalized` (use the user's default locale for the operation).

Listing 1 Comparing and searching CFString contents

```
void compareAndSearchStringsExample() {
    CFStringRef str1 = CFSTR("ABCDEFG");
```

```
    CFStringRef str2 = CFSTR("abcdefg");
    CFStringRef str3 = CFSTR("Kindergarten is the time to start teaching the
ABCDEFG's");
    CFRange foundRange;
    CFComparisonResult result;


    result = CFStringCompareWithOptions(str1, str2,
CFRangeMake(0,CFStringGetLength(str1)), kCFCompareCaseInsensitive);
    if (result == kCFCompareEqualTo) {
        show(CFSTR("%@ is the same as %@"), str1, str2);
    } else {
        show(CFSTR("%@ is not the same as %@"), str1, str2);
    }
   if ( CFStringFindWithOptions(str3, str1, CFRangeMake(0,CFStringGetLength(str3)),
 kCFCompareCaseInsensitive, &foundRange) == true ) {
        show(CFSTR("The string \"%@\" was found at index %d in string \"%@\"."),
str1, foundRange.location, str3);
    } else {
        show(CFSTR("The string \"%@\" was not found in string \"%@\"."), str1,
str3);
    }
}
```

This code generates the following output:

```
ABCDEFG is the same as abcdefg
The string "ABCDEFG" was found at index 47 in string "Kindergarten is the time to
 start teaching the ABCDEFG's".
```

By default, the basis for comparison of CFString objects is a character-by-character literal comparison. In some circumstances this may not give you results you expect, since some characters can be represented in several different ways (for example, “ö” can be represented as two distinct characters (“o” and “umlaut”) or by a single character (“o-umlaut”). If you want to allow loose equivalence, use a search or compare function with the `kCFCompareNonliteral` flag as an option. Note that if you do specify a non-literal comparison, the length of the range returned from a find function might not be the same as the length of the search string.

In addition to the main compare and find functions, string objects provide some convenience functions. `CFStringFind` and `CFStringCompare` are similar to the "main" functions described above but they do not require the specification of a range (the entire string is assumed). Note that you can use `CFStringCompare` elsewhere in Core Foundation when a function pointer conforming to the `CFComparatorFunction` type is required.

Other search and comparison functions of string objects are `CFStringHasPrefix`, `CFStringHasSuffix`, and `CFStringCreateArrayWithFindResults`. The last of these functions is useful when you expect multiple hits with a search operation; it returns an array of `CFRange` structures, each of which specifies the location of a matching substring in the main string.

Sorting Strings

If you sort strings and present the results to the user, you should make sure that you perform a localized comparison using the user's locale. You may also want to arrange strings as they would appear in Finder—for example, these strings { "String 12", "String 1", "string 22", "string 02" } should be sorted as { "String 1", "string 02", "String 12", "string 22" }.

To achieve this, you can use `CFStringCompareWithOptionsAndLocale` with the options `kCFCompareCaseInsensitive`, `kCFCompareNonliteral`, `kCFCompareLocalized`, `kCFCompareNumerically`, `kCFCompareWidthInsensitive`, and `kCFCompareForcedOrdering`. First, implement a function to perform the appropriate comparison:

```
CFComparisonResult CompareStringsLikeFinderWithLocale (
    const void *string1, const void *string2, void *locale)
{
    static CFOptionFlags compareOptions = kCFCompareCaseInsensitive |
                                          kCFCompareNonliteral |
                                          kCFCompareLocalized |
                                          kCFCompareNumerically |
                                          kCFCompareWidthInsensitive |
                                          kCFCompareForcedOrdering;

    CFRange string1Range = CFRangeMake(0, CFStringGetLength(string1));

    return CFStringCompareWithOptionsAndLocale
             (string1, string2, string1Range, compareOptions, (CFLocaleRef)locale);
}
```

Then perform the comparison using that function:

```
// ignore memory management for the sake of clarity and brevity
CFMutableArrayRef theArray = CFArrayCreateMutable(kCFAllocatorDefault, 4, NULL);
CFArrayAppendValue(theArray, CFSTR("String 12"));
CFArrayAppendValue(theArray, CFSTR("String 1"));
CFArrayAppendValue(theArray, CFSTR("string 22"));
CFArrayAppendValue(theArray, CFSTR("string 02"));

CFRange arrayRange = CFRangeMake(0, CFArrayGetCount(theArray));
CFLocaleRef locale = CFLocaleCopyCurrent();

CFArraySortValues (theArray, arrayRange,
                   CompareStringsLikeFinderWithLocale, (void *)locale);

// theArray now contains { "String 1", "string 02", "String 12", "string 22" }
```

Manipulating Mutable String Objects

You can choose from a variety of string object functions to add to and modify the contents of mutable `CFString` objects. These functions, as one might expect, do not work on immutable `CFString` objects. If you want to change the contents of a `CFString` object, you must either start with a content-less mutable `CFString` object or make a mutable copy of an immutable `CFString` object. See Creating Mutable String Objects (page 18) for information on creating objects of this kind.

Forms of Mutation

The functions that manipulate mutable `CFString` objects fall into several categories, described in the following sections.

Appending

You can append strings in a variety of formats to a mutable `CFString` object: other `CFString` objects (`CFStringAppend`), C strings (`CFStringAppendCString`), Unicode characters (`CFStringAppendCharacters`), and formatted strings (`CFStringAppendFormat` and `CFStringAppendFormatAndArguments`).

Inserting, deleting, replacing

The functions `CFStringInsert`, `CFStringDelete`, and `CFStringReplace` perform the corresponding operations. These functions require you to specify a zero-based index into, or range of, the string to be modified.

Padding and trimming

The `CFStringPad` function extends or truncates a mutable `CFString` to a given length; if it extends the string, it pads with a specified character or characters. The `CFStringTrim` function trims a specific character from both sides of the string. For example, the call:

```
CFStringTrim(CFStringCreateMutableCopy(NULL, NULL, CFSTR("xxxabcx")), CFSTR("x"));
```

would result in the string “abc”. A related function, `CFStringTrimWhitespace`, does the same thing with whitespace characters, which include such characters as tabs and carriage returns.

Case operations

Three functions modify the case of a mutable string, making it all uppercase (`CFStringUppercase`), all lowercase (`CFStringLowercase`), or just the first character of each word in a string uppercase (`CFStringCapitalize`).

Code Examples

Listing 1 (page 30) exemplifies several of the functions that manipulate mutable `CFString` objects:

Listing 1 Various operations on a mutable string

```
void mutableStringOperations() {


    CFMutableStringRef mstr;
    CFRange range;
    StringPtr pbuf;
    CFIndex length;


    mstr = CFStringCreateMutable(NULL, 0);
    CFStringAppend(mstr, CFSTR("Now is the time for all good men to come to the
aid of their "));
    CFStringAppend(mstr, CFSTR("party."));
    CFShow(CFSTR("Mutable String 1 - Appended CFStrings"));
    CFShow(mstr);


    range = CFStringFind(mstr, CFSTR("good"), 0);
    if (range.length > 0) {
        CFStringReplace(mstr, range, CFSTR("bad"));
        CFShow(CFSTR("Mutable String 2 - Replaced substring"));
        CFShow(mstr);
    }
```

```
    CFStringUppercase(mstr, NULL);
    CFShow(CFSTR("Mutable String 3 - Convert to uppercase:"));
    CFShow(mstr);
}
```

When compiled and run, this code generates the following output:

```
Mutable String 1 - Appended CFStrings
Now is the time for all good men to come to the aid of their party.
Mutable String 2 - Replaced substring
Now is the time for all bad men to come to the aid of their party.
Mutable String 3 - Convert to uppercase:
NOW IS THE TIME FOR ALL BAD MEN TO COME TO THE AID OF THEIR PARTY.
```

Converting Between String Encodings

String objects give you a number of tools for converting between string encodings. Some routines do the actual conversions while others show which encodings are available and help you chose the best encoding for the current situation.

If you want to convert between any two non-Unicode encodings, you can use a `CFString` object as an intermediary. Say you have a string encoded as Windows Latin 1 and you want to encode it as Mac OS Roman. Just convert the string to Unicode first (the `CFString` object), then convert the string's contents to the desired encoding.

Many of the creation and content-accessing functions described in earlier sections of this document include an encoding parameter typed `CFStringEncoding`. These functions are listed in Table 1 (page 32). To specify the encoding of the source or destination string (depending on whether you're creating a `CFString` object or accessing its contents), specify the `enum` value for the desired encoding in this parameter when you call one of these functions. Use the `CFStringIsEncodingAvailable` function to test for the availability of an "external" encoding on your system before you call a conversion function.

Table 1 Encoding-conversion functions

Converts to CFString (Unicode)
`CFStringCreateWithCString`
`CFStringCreateWithCStringNoCopy`
`CFStringCreateWithBytes`
`CFStringCreateFromExternalRepresentation`

Converts from CFString (Unicode)
`CFStringGetCString`
`CFStringGetCStringPtr`
`CFStringGetBytes`
`CFStringCreateExternalRepresentation`

A word of caution: not all conversions are guaranteed to be successful. This is particularly true if you are trying to convert a `CFString` object with characters that map to a variety of character sets. For example, let's say you have a Unicode string that includes ASCII characters and accented Latin characters. You could convert this string to Mac OS Roman but not to Mac OS Japanese. In these cases, you can specify "lossy" conversion using the `CFStringGetBytes` function; this kind of conversion substitutes a "loss" character for each character that cannot be converted. The `CFStringGetBytes` function is described in the next section

The Basic Conversion Routines

Among the string object functions that convert the encodings of characters in `CFString` objects are the two low-level conversion functions, `CFStringGetBytes` and `CFStringCreateWithBytes`. As their names suggest, these functions operate on byte buffers of a known size. In addition to performing encoding conversions, they also handle any special characters in a string (such as a BOM) that makes the string suitable for external representation.

However, the `CFStringGetBytes` function is particularly useful for encoding conversions because it allows the specification of a *loss* byte. If you specify a character for the loss byte, the function substitutes that character when it cannot convert the Unicode value to the proper character. If you specify 0 for the loss byte, this "lossy conversion" is not allowed and the function returns (indirectly) an partial set of characters when it encounters the first character it cannot convert. All other content-accessing functions of `CFString` disallow lossy conversion.

Listing 1 (page 33) illustrates how `CFStringGetBytes` might be used to convert a string from the system encoding to Windows Latin 1. Note one other feature of the function: it allows you to convert a string into a fixed-size buffer one segment at a time.

Listing 1 Converting to a different encoding with CFStringGetBytes

```
CFStringRef str;
CFRange rangeToProcess;


str = CFStringCreateWithCString(NULL, "Hello World", kCFStringEncodingMacRoman);


rangeToProcess = CFRangeMake(0, CFStringGetLength(str));
while (rangeToProcess.length > 0) {
    UInt8 localBuffer[100];
    CFIndex usedBufferLength;
    CFIndex numChars = CFStringGetBytes(str, rangeToProcess,
kCFStringEncodingWindowsLatin1, '?', FALSE, (UInt8 *)localBuffer, 100,
&usedBufferLength);
```

```
    if (numChars == 0) break;   // Failed to convert anything...
    processCharacters(localBuffer, usedBufferLength);
    rangeToProcess.location += numChars;
    rangeToProcess.length -= numChars;
}
```

If the size of the string to convert is relatively small, you can take a different approach with the `CFStringGetBytes` function. With the buffer parameter set to `NULL` you can call the function to find out two things. If the function result is greater than 0 conversion is possible. And, if conversion is possible, the last parameter (`usedBufLen`) will contain the number of bytes required for the conversion. With this information you can allocate a buffer of the needed size and convert the string at one shot into the desired encoding. However, if the string is large this technique has its drawbacks; asking for the length could be expensive and the allocation could require a lot of memory.

Encoding-Conversion Utilities

Besides the functions that convert between encodings, string objects offer a number of functions that can help you to find out which encodings are available and, of these, which are the best to use in your code.

Encoding by characteristic

The `CFStringGetSmallestEncoding` function determines the smallest encoding that can be used on a particular system (smallest in terms of bytes needed to represent one character). The `CFStringGetFastestEncoding` function gets the encoding on the current system with the fastest conversion time from Unicode. The `CFStringGetSystemEncoding` function obtains the encoding used by strings generated by the operating system.

Available encodings

Use the `CFStringIsEncodingAvailable` and `CFStringGetListOfAvailableEncodings` functions to obtain information about encodings available on your system.

Mappings to encoding sets

You can use the `CFStringConvertEncodingToWindowsCodepage` and `CFStringConvertWindowsCodepageToEncoding` functions to convert between Windows codepage numbers and `CFStringEncoding` values. Similar sets of functions exist for Cocoa NSString encoding constants and IANA "charset" identifiers used by MIME encodings.

Supported Encodings

Core Foundation string objects supports conversions between Unicode encodings of CFString objects and a wide range of international, national, and industry encodings. Supported encodings come in two sets, an "internal" set defined in `CFString.h` by the `CFStringBuiltInEncodingsenum`, and an "external" set defined in `CFStringEncodingExt.h` by the `CFStringEncodingsenum`. The encodings in the internal set are guaranteed to be available on all platforms for conversions to and from CFString objects. The built-in encodings (as designated by the constant names in `CFStringBuiltInEncodings`) include:

```
kCFStringEncodingMacRoman
kCFStringEncodingWindowsLatin1
kCFStringEncodingISOLatin1
kCFStringEncodingNextStepLatin
kCFStringEncodingASCII
kCFStringEncodingUnicode
kCFStringEncodingUTF8
kCFStringEncodingNonLossyASCII
kCFStringEncodingUTF16
kCFStringEncodingUTF16BE
kCFStringEncodingUTF32
```

Conversions using the encodings in the external set are possible only if the underlying system supports the encodings.

Handling External Representations of Strings

An *external representation* of a CFString object in Core Foundation is the string data in a form that can be written to disk and read back in on the same platform or on a different platform. The format of an externally represented CFString object is a CFData object. If the encoding of the characters is Unicode, the data usually includes a special character called a BOM (for "byte order mark") that designates the endianness of the data. When the external representation of a string is read, Core Foundation evaluates the BOM and does any necessary byte swapping. If the encoding is Unicode and there is no BOM, the data is assumed to be big-endian. When you use string objects to write out an external representation of Unicode characters, the BOM is inserted, except for representations created with encoding constants kCFStringEncodingUTF16BE, kCFStringEncodingUTF16LE, kCFStringEncodingUTF32BE, and kCFStringEncodingUTF32LE. These encodings do not require a BOM because the byte order is explicitly indicated by the letters "BE" (big-endian) and "LE" (little-endian).

When you want the character data represented by a CFString object to persist, either as a file on disk or as data sent over a network, you should first convert the CFString object to a CFData object using the function `CFStringCreateExternalRepresentation`. The CFData object is called an "external representation" of the CFString object; if the encoding is Unicode, the function automatically inserts a BOM (byte order marker) in the data to specify endianness. You can convert an external-representation CFData object back to a CFString object with the `CFStringCreateFromExternalRepresentation` function.

Listing 1 (page 36) shows how the external-representation functions might be used. The last parameter of the `CFStringCreateExternalRepresentation` function specifies a loss byte, the value to be assigned to characters that cannot be converted to the specified encoding. If the loss byte is 0 (as in the example below) and conversion errors occur, the result of the function is `NULL`. This feature is similar to that provided by the `CFStringGetBytes` function; however the `CFStringCreateExternalRepresentation` function is more convenient since it gives you a CFData object.

Listing 1 Using the external-representation functions

```
CFDataRef appendTimeToLog(CFDataRef log) {
    CFMutableStringRef mstr;
    CFStringRef str;
    CFDataRef newLog;
    CFGregorianDate date =
        CFAbsoluteTimeGetGregorianDate(CFAbsoluteTimeGetCurrent(),
```

```
            CFTimeZoneCopySystem());

    str = CFStringCreateFromExternalRepresentation(NULL, log,
            kCFStringEncodingUTF8);
    CFShow(str);
    mstr = CFStringCreateMutableCopy(NULL, 0, str);
    CFStringAppendFormat(mstr, NULL,
        CFSTR("Received at %d/%d/%d %.2d:%.2d:%2.0f\n"),
        date.month, date.day, date.year, date.hour, date.minute,
        date.second);
    CFShow(mstr);
    newLog = CFStringCreateExternalRepresentation(NULL, mstr,
        kCFStringEncodingUTF8, '?');
    CFRelease(str);
    CFRelease(mstr);
    CFShow(newLog);
    return newLog;
}
```

This code generates output similar to the following snippet:

```
Master Log


Master Log


Received at 7/20/1999 19:23:16


<CFData 0x103c0 [0x69bce158]>{length = 43, capacity = 43, bytes =
0x4d6173746572204c6f670a0a52656365 ... 393a32333a31360a}
```

As the example shows, the CFString object in its external representation is immutable, regardless of its mutability status before being stored as a CFData object. If you want to modify the CFString object returned from `CFStringCreateFromExternalRepresentation`, you need to make a mutable copy of it.

Instead of using the `CFStringCreateFromExternalRepresentation` function to create a CFString object and *then* access the characters in the object, you can use CFData functions to get at the characters directly. Listing 3 (page 17) shows how this is done using the CFData functions `CFDataGetLength` and `CFDataGetBytePtr`.

Creating and Using Ranges

Many Core Foundation take ranges—a structure of type `CFRange`—as parameters. A range is a measure of a linear segment; it has a beginning location and a length. To create and initialize this structure you can use the convenience function `CFRangeMake`.

The following code fragment gets the number of subsequent elements in an array that match the first element:

```
CFRange aRange = CFRangeMake(1, CFArrayGetCount(array) - 1);
// Since start is 1, length of remainder of range is count-1
const void *aValue = CFArrayGetValueAtIndex(array, 0);
CFIndex numVals = CFArrayGetCountOfValue(array, aRange, aValue);
```

Character Sets

In Core Foundation, a *character set*, as represented by a CFCharacterSet object, represents a set of Unicode characters. Functions can use character sets to group characters together for searching and parsing operations, so that they can find or exclude any of a particular set of characters during a search. Aside from testing for membership in a character set, a character-set object simply holds a set of character values to limit operations on strings.

You use character sets in search, parsing, and comparison operations involving strings. Programmatic interfaces that require references to CFCharacterSet objects are currently under development in both Core Foundation and Carbon.

To obtain a CFCharacterSet object that can be passed into a function, you can either use one of the predefined character sets or create your own. To use one of the predefined sets—including such things as whitespace, alphanumeric characters, and decimal digits—call `CFCharacterSetGetPredefined` with one of the `CFCharacterSetPredefinedSet` constants. Several CFCharacterSet functions create character sets from strings and bitmapped data and others allow you to create mutable character sets. You can use a predefined character set as a starting point for building a custom set by making a mutable copy of it and changing that.

Because character sets often participate in performance-critical code, you should be aware of the aspects of their use that can affect the performance of your application. Mutable character sets are generally much more expensive than immutable character sets. They consume more memory and are costly to invert (an operation often performed in scanning a string). Because of this, you should follow these guidelines:

- Create as few mutable character sets as possible.
- Cache character sets (in a global dictionary, perhaps) instead of continually recreating them.
- When creating a custom set that doesn't need to change after creation, make an immutable copy of the final character set for actual use, and dispose of the working mutable character set.

String Format Specifiers

This article summarizes the format specifiers supported by string formatting methods and functions.

Format Specifiers

The format specifiers supported by the `NSString` formatting methods and CFString formatting functions follow the IEEE printf specification; the specifiers are summarized in Table 1 (page 41). Note that you can also use the "`n$`" positional specifiers such as `%1$@ %2$s`. For more details, see the IEEE printf specification. You can also use these format specifiers with the `NSLog` function.

Table 1 Format specifiers supported by the `NSString` formatting methods and CFString formatting functions

Specifier	Description
`%@`	Objective-C object, printed as the string returned by `descriptionWithLocale:` if available, or `description` otherwise. Also works with `CFTypeRef` objects, returning the result of the `CFCopyDescription` function.
`%%`	`'%'` character.
`%d`, `%D`	Signed 32-bit integer (`int`).
`%u`, `%U`	Unsigned 32-bit integer (`unsigned int`).
`%x`	Unsigned 32-bit integer (`unsigned int`), printed in hexadecimal using the digits 0–9 and lowercase a–f.
`%X`	Unsigned 32-bit integer (`unsigned int`), printed in hexadecimal using the digits 0–9 and uppercase A–F.
`%o`, `%O`	Unsigned 32-bit integer (`unsigned int`), printed in octal.
`%f`	64-bit floating-point number (`double`).
`%e`	64-bit floating-point number (`double`), printed in scientific notation using a lowercase e to introduce the exponent.
`%E`	64-bit floating-point number (`double`), printed in scientific notation using an uppercase E to introduce the exponent.

Specifier	Description
%g	64-bit floating-point number (double), printed in the style of %e if the exponent is less than –4 or greater than or equal to the precision, in the style of %f otherwise.
%G	64-bit floating-point number (double), printed in the style of %E if the exponent is less than –4 or greater than or equal to the precision, in the style of %f otherwise.
%c	8-bit unsigned character (unsigned char), printed by NSLog() as an ASCII character, or, if not an ASCII character, in the octal format \\ddd or the Unicode hexadecimal format \\udddd, where d is a digit.
%C	16-bit Unicode character (unichar), printed by NSLog() as an ASCII character, or, if not an ASCII character, in the octal format \\ddd or the Unicode hexadecimal format \\udddd, where d is a digit.
%s	Null-terminated array of 8-bit unsigned characters. Because the %s specifier causes the characters to be interpreted in the system default encoding, the results can be variable, especially with right-to-left languages. For example, with RTL, %s inserts direction markers when the characters are not strongly directional. For this reason, it's best to avoid %s and specify encodings explicitly.
%S	Null-terminated array of 16-bit Unicode characters.
%p	Void pointer (void *), printed in hexadecimal with the digits 0–9 and lowercase a–f, with a leading 0x.
%a	64-bit floating-point number (double), printed in scientific notation with a leading 0x and one hexadecimal digit before the decimal point using a lowercase p to introduce the exponent.
%A	64-bit floating-point number (double), printed in scientific notation with a leading 0X and one hexadecimal digit before the decimal point using a uppercase P to introduce the exponent.
%F	64-bit floating-point number (double), printed in decimal notation.

Table 2 Length modifiers supported by the NSString formatting methods and CFString formatting functions

Length modifier	Description
h	Length modifier specifying that a following d, o, u, x, or X conversion specifier applies to a short or unsigned short argument.

Length modifier	Description
`hh`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument.
`l`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long` or `unsigned long` argument.
`ll`, `q`	Length modifiers specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long` or `unsigned long long` argument.
`L`	Length modifier specifying that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.
`z`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument.
`t`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument.
`j`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `intmax_t` or `uintmax_t` argument.

Platform Dependencies

OS X uses several data types—`NSInteger`, `NSUInteger`,`CGFloat`, and `CFIndex`—to provide a consistent means of representing values in 32- and 64-bit environments. In a 32-bit environment, `NSInteger` and `NSUInteger` are defined as `int` and `unsigned int`, respectively. In 64-bit environments, `NSInteger` and `NSUInteger` are defined as `long` and `unsigned long`, respectively. To avoid the need to use different printf-style type specifiers depending on the platform, you can use the specifiers shown in Table 3. Note that in some cases you may have to cast the value.

Table 3 Format specifiers for data types

Type	Format specifier	Considerations
`NSInteger`	`%ld` or `%lx`	Cast the value to `long`.
`NSUInteger`	`%lu` or `%lx`	Cast the value to `unsigned long`.
`CGFloat`	`%f` or `%g`	`%f` works for floats and doubles when formatting; but note the technique described below for scanning.

Type	Format specifier	Considerations
`CFIndex`	`%ld` or `%lx`	The same as `NSInteger`.
pointer	`%p` or `%zx`	`%p` adds `0x` to the beginning of the output. If you don't want that, use `%zx` and no typecast.

The following example illustrates the use of `%ld` to format an `NSInteger` and the use of a cast.

```
NSInteger i = 42;
printf("%ld\n", (long)i);
```

In addition to the considerations mentioned in Table 3, there is one extra case with scanning: you must distinguish the types for `float` and `double`. You should use `%f` for float, `%lf` for double. If you need to use `scanf` (or a variant thereof) with `CGFloat`, switch to `double` instead, and copy the `double` to `CGFloat`.

```
CGFloat imageWidth;
double tmp;
sscanf (str, "%lf", &tmp);
imageWidth = tmp;
```

It is important to remember that `%lf` does not represent `CGFloat` correctly on either 32- or 64-bit platforms. This is unlike `%ld`, which works for `long` in all cases.

Document Revision History

This table describes the changes to *String Programming Guide for Core Foundation*.

Date	Notes
2014-02-11	Clarified discussion in "The Unicode Basis of CFString Objects."
2012-07-17	Removed statement that character set is restricted to ASCII.
2008-03-11	Added information to "Handling External Representations of Strings" about string encodings that do not include a BOM.
2007-07-10	Added section to "Comparing, Sorting, and Searching String Objects" to illustrate how to sort strings like Finder.
2006-05-23	Included the "String Format Specifiers" article.
2006-01-10	Changed title from "Strings." Updated "About Strings" to include a reference to CFAttributedString.
2005-08-11	Corrected description of character-by-character literal comparisons in "Comparing and Searching String Objects."
2003-10-22	Corrected use of `CFRangeMake` in Accessing the Contents of String Objects.
2003-08-07	Corrected the description of the `CFStringCapitalize` function.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.

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