Ownership Policy

Applications using Core Foundation constantly access and create and dispose of objects. In order to ensure that you do not leak memory, Core Foundation defines rules for getting and creating objects.

Fundamentals

When trying to understand memory management in a Core Foundation application, it is helpful to think not in terms of memory management per se, but instead in terms of object ownership. An object may have one or more owners; it records the number of owners it has using a retain count. If an object has no owners (if its retain count drops to zero), it is disposed of (freed). Core Foundation defines the following rules for object ownership and disposal.

- If you create an object (either directly or by making a copy of another object—see The Create Rule), you own it.
- If you get an object from somewhere else, you do not own it. If you want to prevent it being disposed of, you must add yourself as an owner (using `CFRetain`).
- If you are an owner of an object, you must relinquish ownership when you have finished using it (using `CFRelease`).

Naming Conventions

There are many ways in which you can get a reference to an object using Core Foundation. In line with the Core Foundation ownership policy, you need to know whether or not you own an object returned by a function so that you know what action to take with respect to memory management. Core Foundation has established a naming convention for its functions that allows you to determine whether or not you own an object returned by a function. In brief, if a function name contains the word "Create" or "Copy", you own the object. If a function name contains the word "Get", you do not own the object. These rules are explained in greater detail in The Create Rule and The Get Rule.

Important: Cocoa defines a similar set of naming conventions for memory management (see *Advanced Memory Management Programming Guide*). The Core Foundation naming conventions, in particular use of the word “create”, only apply to C functions that return Core Foundation objects. Naming conventions for Objective-C methods are governed by the Cocoa conventions, irrespective of whether the method returns a Core Foundation or Cocoa object.

The Create Rule

Core Foundation functions have names that indicate when you own a returned object:

- Object-creation functions that have “Create” embedded in the name;
- Object-duplication functions that have “Copy” embedded in the name.

If you own an object, it is your responsibility to relinquish ownership (using `CFRelease`) when you have finished with it.

Consider the following examples. The first example shows two create functions associated with CFTimeZone and one associated with CFBundle.

```
CFTimeZoneRef   CFTimeZoneCreateWithTimeIntervalFromGMT (CFAllocatorRef allocator,
CFTimeInterval ti);

CFDictionaryRef CFTimeZoneCopyAbbreviationDictionary (void);

CFBundleRef     CFBundleCreate (CFAllocatorRef allocator, CFURLRef bundleURL);
```

The first function contains the word "Create" in its name, and it creates a new CFTimeZone object. You own this object, and it is your responsibility to relinquish ownership. The second function contains the word "Copy" in its name, and creates a copy of an attribute of a time zone object. (Note that this is different from getting the attribute itself—see The Get Rule.) Again, you own this object, and it is your responsibility to relinquish ownership. The third function, `CFBundleCreate`, contains the word "Create" in its name, but the documentation states that it may return an existing CFBundle. Again, though, you own this object whether or not a new one is actually created. If an existing object is returned, its retain count is incremented so it is your responsibility to relinquish ownership.

The next example may appear to be more complex, but it still follows the same simple rule.

```
/* from CFBag.h */

CF_EXPORT CFBagRef  CFBagCreate(CFAllocatorRef allocator, const void **values,
CFIndex numValues, const CFBagCallBacks *callBacks);

CF_EXPORT CFMutableBagRef   CFBagCreateMutableCopy(CFAllocatorRef allocator, CFIndex
capacity, CFBagRef bag);
```

The CFBag function `CFBagCreateMutableCopy` has both “Create” and “Copy” in its name. It is a creation function because the function name contains the word “Create”. Note also that the first argument is of type `CFAllocatorRef`—this serves as a further hint. The “Copy” in this function is a hint that the function takes a `CFBagRef` argument and produces a duplicate of the object. It also refers to what happens to the element objects of the source collection: they are copied to the newly created bag. The secondary “Copy” and “NoCopy” substrings of function names indicate how objects owned by some source objects are treated—that is, whether they are copied or not.

The Get Rule

If you receive an object from any Core Foundation function other than a creation or copy function—such as a Get function—you do not own it and cannot be certain of the object’s life span. If you want to ensure that such an object is not disposed of while you are using it, you must claim ownership (with the `CFRetain` function). You are then responsible for relinquishing ownership when you have finished with it.

Consider the `CFAttributedStringGetString` function, which returns the backing string for an attributed string.

```
CFStringRef CFAttributedStringGetString (CFAttributedStringRef aStr);
```

If the attributed string is *freed*, it relinquishes ownership of the backing string. If the attributed string was the backing string's only owner, then the backing string now has no owners and it is itself freed. If you need to access the backing string after the attributed string has been disposed of, you must claim ownership (using `CFRetain`)—or make a copy of it. You must then relinquish ownership (using `CFRelease`) when you have finished with it, otherwise you create a memory leak.

Instance Variables and Passing Parameters

A corollary of the basic rules is that when you pass an object to another object (as a function parameter), you should expect that the receiver will take ownership of the passed object if it needs to maintain it.

To understand this, put yourself in the position of the implementer of the receiving object. When a function receives an object as a parameter, the receiver does not initially own the object. The object may therefore be deallocated at any time—unless the receiver takes ownership of if (using `CFRetain`). When the receiver has finished with the object—either because it is replaced a new value or because the receiver is itself being deallocated—the receiver is responsible for relinquishing ownership (using `CFRelease`).

Ownership Examples

To prevent runtime errors and memory leaks, you should ensure that you consistently apply the Core Foundation ownership policy wherever Core Foundation objects are received, passed, or returned. To understand why it may be necessary to become an owner of an object you did not create, consider this example. Suppose you get a value from another object. If the value's "containing" object is subsequently deallocated, it relinquishes ownership of the "contained" object. If the containing object was the only owner of the value, then the value has no owners and it is deallocated too. You now have a reference to a freed object, and if you try to use it your application will crash.

The following fragments of code illustrate three common situations: a Set accessor function, a Get accessor function, and a function that holds onto a Core Foundation object until a certain condition is met. First the Set function:

```
static CFStringRef title = NULL;
void SetTitle(CFStringRef newTitle) {
    CFStringRef temp = title;
    title = CFStringCreateCopy(kCFAllocatorDefault , newTitle);
    CFRelease(temp);
}
```

The above example uses a static `CFStringRef` variable to hold the retained CFString object. You could use other means for storing it but you must put it, of course, in some place that isn't local to the receiving function. The function assigns the current title to a local variable before it copies the new title and releases the old title. It releases after copying in case the CFString object passed in is the same object as the one currently held.

Notice that in the above example the object is copied rather than simply retained. (Recall that from an ownership perspective these are equivalent—see Fundamentals.) The reason for this is that the title property might be considered an attribute. It is something that should not be changed except through accessor methods. Even though the parameter is typed as `CFStringRef`, a reference to a CFMutableString object might be passed in, which would allow for the possibility of the value being changed externally. Therefore you copy the object so that it won't be changed while you hold it. You should copy an object if the object is or could be mutable and you need your own unique version of it. If the object is considered a relationship, then you should retain it.

The corresponding Get function is much simpler:

```
CFStringRef GetTitle() {
    return title;
}
```

By simply returning an object you are returning a weak reference to it. In other words, the pointer value is copied to the receiver's variable but the reference count is unchanged. The same thing happens when an element from a collection is returned.

The following function retains an object retrieved from a collection until it no longer needs it, then releases it. The object is assumed to be immutable.

```
static CFStringRef title = NULL;
void MyFunction(CFDictionary dict, Boolean aFlag) {
```

```
    if (!title && !aFlag) {
        title = (CFStringRef)CFDictionaryGetValue(dict, CFSTR("title"));
        title = CFRetain(title);
    }
    /* Do something with title here. */
    if (aFlag) {
        CFRelease(title);
    }
}
```

The following example shows passing a number object to an array. The array's callbacks specify that objects added to the collection are retained (the collection owns them), so the number can be released after it's added to the array.

```
float myFloat = 10.523987;
CFNumberRef myNumber = CFNumberCreate(kCFAllocatorDefault,
                                      kCFNumberFloatType, &myFloat);
CFMutableArrayRef myArray = CFArrayCreateMutable(kCFAllocatorDefault, 2,
&kCFTypeArrayCallBacks);
CFArrayAppendValue(myArray, myNumber);
CFRelease(myNumber);
// code continues...
```

Note that there is a potential pitfall here if (a) you release the array, and (b) you continue to use the number variable after releasing the array:

```
CFRelease(myArray);
CFNumberRef otherNumber = // ... ;
CFComparisonResult comparison = CFNumberCompare(myNumber, otherNumber, NULL);
```

Unless you retained the number or the array, or passed either to some other object which maintains ownership of it, the code will fail in the comparison function. If no other object owns the array or the number, when the array is released it is also deallocated, and so it releases its contents. In this situation, this will also result in the deallocation of the number, so the comparison function will operate on a freed object and thus crash.