Collections Programming Topics for Core Foundation

Developer

Contents

Tables and Listings

Introduction

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.

Core Foundation collection objects help you store, organize, and retrieve "quantities" of data of virtually all types. This topic describes objects that let you group together other types of objects in, for example, arrays, sets, or dictionaries.

Developers using Core Foundation for their base functionality need to understand how collections work.

Organization of This Document

In addition to organizing data for quick and accurate retrieval, collection objects bring several benefits to programming:

- They allow you to customize the automatic behavior of collections to suit the type of data stored (Collection Customization (page 12)).
- They can apply program-defined logic to all data elements in a collection (Applying Program-Defined Functions to Collections (page 27)).
- They can ensure the uniqueness of data elements added to a collection.
- They enable you to modify (add, insert, delete, sort, and so forth) the data elements in a collection.
- They are essential to Core Foundation property lists, along with string objects (CFString), number objects (CFNumber and CFBoolean), and data objects (CFData). See *Property List Programming Topics for Core Foundation* for details.

Core Foundation defines several types of collection objects:

- Array objects (of the opaque type CFArray) organize elements by sequential position.
- Dictionary objects (of the opaque type CFDictionary) identify elements through an arbitrary key.
- Set objects (of the opaque type CFSet) are a collection of elements in which there are no duplicates.

- Bag objects (of the opaque type CFBag) are a collection of elements in which there can be duplicates.
- Tree objects (of the opaque type CFTree) organize elements in hierarchical (parent-children) relationships.

Collection objects, to some degree, are containers of values. (In this document, the word “value” denotes an element contained by a collection.) But there are wide differences in how collections contain and dispense their values. The organization of the concepts in this topic reflects these differences. What might be termed the “true collections”—arrays, dictionaries, sets, and bags—are described together because of their strong similarities. Then this topic goes on to describe trees, which are structurally quite different from the true collections.

The collections API allow you to do expected things with collection objects: create them, add values to them, extract values from them, and so on. Because the programming interfaces of CFArray, CFDictionary, CFSet, and CFBag are very much alike in what they do and how they do it, the following sections include all these types in their discussions. However, the programming interfaces of CFTree objects are sufficiently different that tasks related to these objects are described in Creating and Using Tree Structures (page 29).

Common Characteristics of Collections

A collection object (or simply, a collection) is a container of discrete values, usually of the same type. A collection can contain other Core Foundation objects, custom data structures, and primitive data values. Contained Core Foundation objects can be of different types. Strictly speaking, "contain" is not an exact term (although it's conceptually useful) because an element of data in a Core Foundation collection can be no larger than the size of a pointer, which varies by processor and compiler. But pointers afford a great deal of flexibility and thus collection objects permit references to Core Foundation objects and pointers to any chunk of data as well as primitive values (such as integers) that take up no more space than a pointer address.

Important: You should be careful if you use a collection object to store primitive values directly. Because the size of a pointer address on a platform can change over time or can vary among the platforms on which code is expected to run, storing primitive values directly in collections might result in odd behavior, such as the truncation of the values.

All collection objects enable you to access a contained value that satisfies a particular external property. This property, generally referred to as a "key," varies according to the organizing scheme enforced by the type of collection. For example, the key for an array is an integer that specifies position within the collection; on the other hand, a dictionary—for which the term "key" has a more of a conventional meaning—permits any arbitrary value to act as the key for retrieving a contained value. A Core Foundation object retrieved from a collection is not guaranteed to persist because the owning collection might free it, so if you want to hold onto it you should retain or copy it. See Types of Collections (page 8) for more on the keys of specific collection types.

Behavior Is Determined by Callback Functions

Most collection objects have a set of call back functions that define fundamental operations performed on the elements of that collection. These callbacks are invoked to

- retain elements added to a collection
- release elements removed from a collection
- compare one element in a collection to another
- print debugging information about contained elements
- for sets, bags, and dictionary keys, compute a hash code

Default callbacks are provided for collections that hold Core Foundation objects.

Notice in particular that the callbacks affect the memory management semantics of collection objects. When you create a collection, you specify the callbacks that it should use when items are added to or removed from the collection. The default callbacks are appropriate for collections that contain CF types—they retain an object that is added to the collection, and release an object that is removed from the collection. If you have custom data, or if the default callbacks are insufficient for your purposes, you can define your own. See Creating and Copying Collections (page 13) for more on this subject.

One type of function that is particularly interesting is the *applier function*. Most collection objects allow you to define a callback function. You then pass a pointer to this applier function in collection functions that have `ApplyFunction` embedded in their names. The behavior defined in the callback function is applied iteratively to each element in the collection. This behavior is essentially the same thing as a `for` loop enumerating a collection; the code within the loop is applied to each element in the collection.

Most Collections Offer Mutable and Immutable Variants

Except for trees (which are not true containers of other objects), collections come in two "flavors" or variants: immutable and mutable. The values in immutable collections are set for the life of the collection; you cannot add values to them or remove values from them. Mutable collections, however, let you add values, reorder values, and remove values. Mutable collections have two sub-flavors, fixed-size and variable-size, as determined by the capacity parameter in the functions that create these objects. For fixed-size collections you set the maximum number of values that the collection can contain.

Variable-size collections, on the other hand, can hold an unlimited number of values (or rather, limited only by constraints external to the collection, like the amount of available memory). Fixed-size collections tend to be higher performing, but you must be able to put a definite limit on the number of values that can be contained.

Some collection functions operate on both mutable and immutable collections (for example, getting the number of data elements and accessing data) whereas others work only with mutable collection objects, performing such operations as appending, inserting, removing, and sorting elements. See Working With Mutable Collections (page 25) for details on these kinds of operations.

Types of Collections

Although the major collection types of Core Foundation—arrays, dictionaries, sets, bags, and trees—have characteristics in common, they each have distinct ways of organizing the data elements they contain and, in some cases, restricting which elements they contain.

Arrays

A Core Foundation array—an object of opaque type CFArray—is an ordered, compact container of values. The values are in sequential order and the key for accessing these values is their *index*, an integer specifying a value's position in the sequence. The range of indexes is from 0 to *n* -1 where *n* is the number of values in the array. An index of 0 identifies the first value in the array. You can think of arrays as numbered slots.

Arrays are ideal for programmatic situations where sequential position is significant, such as lists whose items are ranked according to some scheme (priority, for example) or where there is a mapping between values in the array and items in list-type controls such as menus. However, you can use arrays as general-purpose containers for holding and iterating over collected values.

An array is said to be compact because, with mutable arrays, deleting a value does not leave a gap in the array. The values with higher-numbered indexes have their indexes decremented by 1. If you insert a value, all subsequent elements have their keys incremented by 1. Because of this behavior, the key for accessing a particular object may change over time as values are inserted into and deleted from an array. But the set of valid indexes is always between 0 and *n* -1.

The access time for a value in the array is guaranteed to be at worst `O(log N)` for any implementation, current and future, but will often be `O(1)` (that is, constant time). Linear search operations similarly have a worst-case complexity of `O(N*log N)`, though typically the bounds will be tighter. Insertion or deletion operations are typically linear in the number of objects in the array, but may be `O(N*log N)` clearly in the worst case in some implementations. There are no favored positions within the array for performance; for example, it is not necessarily faster to access values with low indexes or to insert or delete values with high indices.

Dictionaries

A dictionary—an object of the CFDictionary type—is a hashing-based collection whose keys for accessing its values are arbitrary, program-defined pieces of data (or pointers to data). Although the key is usually a string (or, in Core Foundation, a CFString object), it can be anything that can fit into the size of a pointer—an integer, a reference to a Core Foundation object, even a pointer to a data structure (unlikely as that might be). The keys of dictionaries are unlike the keys of the other collection objects in that, conceptually, they are also contained by the collection along with the values. Dictionaries are primarily useful for holding and organizing data that can be labeled, such as values extracted from text fields in the user interface.

In Core Foundation, a dictionary differs from an array in that the key used to access a particular value in the dictionary remains the same as values are added to or removed from the dictionary—that is, until a value associated with a particular key is replaced or removed. In an array, the key (that is, the index) that is used to retrieve a particular value can change over time as values are added to or deleted from the array. Also, unlike an array, a dictionary does not put its values in any order. To enable later retrieval of a value, the key of the key-value pair should be constant (or be treated as constant); if the key changes after being used to put a value in the dictionary, the value might not be retrievable. The keys of a dictionary form a set; in other words, keys are guaranteed to be unique in a dictionary.

Although a customization of the CFDictionary type might not use hashing and a hash table for storage of the values, the keys require both a function that generates hash codes and a function that tests for equality of two keys based on their hash codes. These two functions together must maintain the invariant that

```
if equal(X,Y) then
    hash(X) == hash(Y)
```

Although the converse of this logic is not generally true, the contra positive is required by Boolean logic:

```
if hash(X) != hash(Y),
    then !equal(X,Y)
```

If the `hash` and `equal` key callbacks are `NULL`, the key is used as a pointer-sized integer and pointer equality is used. For best performance, you should try to provide a `hash` callback that computes sufficiently dispersed hash codes for the key set.

The access time for a value in a CFDictionary object is guaranteed to be at worst `O(log N)` for any implementation, but is often `O(1)` (constant time). Insertion or deletion operations are typically in constant time as well, but are `O(N*log N)` in the worst cases. It is faster to access values through a key than accessing them directly. Dictionaries tend to use significantly more memory than an array with the same number of values.

Sets and Bags

Sets and bags are related types of collections. What they have in common is that the key for accessing a value in the collection is the value itself. The difference between sets and bags is the membership "rule" for values. With sets, if the value already exists in the collection, an identical value cannot be added to the set; conversely, you can add any value to a bag even if the bag already holds that value.

This "uniquing" functionality (or lack thereof) can have many uses. Say, for example, that your program is searching the Internet and you want to keep all qualifying URLs but don't want duplicates; a set would work nicely for this purpose. A bag could be useful for statistical sampling; you could put all collected values in it and after you finish collecting data, you could query it for the frequency of each value.

Equality of key value and contained value is determined by a callback function that you can define. This function could set any criterion for equality. For example, the values of a set or bag could be custom structures representing a customer record; the function could decide that the external (key) and contained values are the same if the value of the account-number field in both structures is the same. (As with all collections, you can also specify default callback functions when Core Foundation objects are the values in a set or bag.) For sets (as with dictionaries), the `equal` callback function also decides which values can legally be added to the collection.

Trees

You use trees—objects of the CFTree type—to represent nodes in hierarchically organized structures, such as a file system with its files and directories. Each node in this tree structure is a tree object, and that object has relationships to other tree objects. Conceptually, objects of the CFTree type are significantly different from other Core Foundation collection objects. Arrays, dictionaries, sets, and bags are containers of (typically) multiple values, but tree objects have a one-to-one relationship with their data. In other words, a tree object is associated with only one pointer-size value (although that value can be, for instance, a pointer to a structure with multiple members or even a reference to a collection object). However, the essential characteristic of a tree object is that it keeps references to multiple subtrees, each with its own chunk of data, and in that sense a interlinked group of tree objects—the tree structure—can be considered a collection.

The parent-child paradigm is useful for understanding tree objects. As stated above, these objects stand in a hierarchical relationship to each other, and at the top of the hierarchy is the "root" tree. This tree is not a subtree of any other tree—that is, it has no parent tree—but it has one or more subtrees, or child trees. And these child trees can be the parents of other trees.

Tree objects have certain rules governing their relationships. When you add children to a tree, the parent retains them, but the children do not retain their parent. Additionally, a tree may not have as one of its child trees a tree of which it is itself a child; no cycles are allowed in the traversal of a tree and its subtrees. Two trees

with the same parent tree are called “siblings” of each other. Conceptually, siblings are in sequential order, as in a linked list. Operations performed on the children of a tree apply only to those direct descendants; in other words, they do not recursively affect any subtrees of those children.

Unlike the other collection objects, a tree has no need for callbacks that operate on contained values because, structurally, these values can be only of the type CFTree. However, to be useful a tree object must have some data associated with it, and this data can determine its relationships with other trees. When you create a Core Foundation tree object, you must define a “context” for the tree. The context identifies the associated data and defines callbacks that are invoked to perform necessary operations on the data, such as retaining, releasing, comparing, and describing it.

Collection Customization

Core Foundation permits you to customize how collections operate on the values they contain. When you create a collection object you must pass in an initialized structure that includes pointers to callback functions that define much of the behavior of the collection. The type of structure passed in for tree objects is different from the structure passed in for the other types of collection objects.

For arrays, dictionaries, sets, and bags, you pass a pointer to a callback structure (for example, `CFArrayCallBacks`) into the creation function. This structure contains pointers to functions that are invoked by Core Foundation to retain, release, describe, and compare the values put into the collection. Ideally the behavior of a collection is to somehow retain a value put into it so that, in case the code that put the value there later frees it, the value still remains in the collection. (You can elect not to retain values put into a collection, but this can lead to undefined behavior in your program.) A collection should release values removed from it if they were earlier retained, or free them if they were earlier allocated. The callback that compares values is used in sorting and searching operations. The callback that describes values is invoked by the `CFCopyDescription` (which is invoked in turn by `CFShow`) to print debugging information to the console.

For trees, you pass an initialized context structure (`CFTreeContext`) into the creation function. This structure has pointers to the same kinds of functions as the callback structure—retain, release, compare, and describe—but these functions operate on an `info` member of the context structure, which holds or points to the data associated with the tree.

You may assign `NULL` to a callback or context function pointer if the operation is not required for the values in the collection (such as retaining certain types of data). In addition, each collection object created with a callback structure—arrays, dictionaries, sets, and bags—allows you to specify a predefined structure initialized with default callbacks when the collection contains Core Foundation objects.

Creating and Copying Collections

You have more options for creating and copying collection objects than with most other Core Foundation types. Collection objects can be immutable or mutable, and, if the latter, can be either fixed-size or variable-size (immutable objects are, of course, always fixed-size). Each of these variants has its own possibilities and limitations.

Because the values (including, for dictionaries, keys) in an immutable collection cannot change once the collection is created, you must supply these values when you create the object. The acceptable form for these initializing values is a C array (unless the collection is to hold only one value). The input parameters must specify the address of this C array. Listing 1 (page 13) illustrates how a CFArray object might be created.

Listing 1 Creating an immutable CFArray object

```
CFStringRef strs[3];
CFArrayRef anArray;

strs[0] = CFSTR("String One");
strs[1] = CFSTR("String Two");
strs[2] = CFSTR("String Three");

anArray = CFArrayCreate(NULL, (void *)strs, 3, &kCFTypeArrayCallBacks);
CFShow(anArray);
CFRelease(anArray);
```

Notice the final parameter of the `CFArrayCreate` call, the address of the `kCFTypeArrayCallBacks` constant. This constant identifies a predefined *callback structure* for the CFArray type. The functions that create and copy collection objects such as arrays, dictionaries, sets, and bags require you to specify callback structures. These structures contain pointers to callbacks function that control how values (and keys) of a collection are kept, evaluated, and described. Each of the collection types listed above defines one or more predefined callback structures that you can use when the values of the collection are Core Foundation objects. All of the pre-defined collection callback structures use `CFRetain` as the retain callback and `CFRelease` as the release callback so that objects added to the collection are retained, and objects removed from the collection are released.

The `CFDictionaryCreate` function, which creates an immutable dictionary object, is somewhat different from the related collection functions. It requires you to specify not only one or more values but a matching set of keys for these values. The typical means for specifying these lists of values and keys are two C arrays. Listing 2 (page 14) provides a simple example.

Listing 2 Creating an immutable CFDictionary object

```
CFStringRef keys[3];
CFStringRef values[3];
CFDictionaryRef aDict;

keys[0] = CFSTR("Key1");
keys[1] = CFSTR("Key2");
keys[2] = CFSTR("Key3");
values[0] = CFSTR("Value1");
values[1] = CFSTR("Value2");
values[2] = CFSTR("Value3");

aDict = CFDictionaryCreate(NULL, (void **)keys, (void **)values, 3,
&kCFCopyStringDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
CFShow(aDict);
CFRelease(aDict);
```

The keys in one array are positionally matched with the values in the other array. Thus, in the example above, the third element in the `keys` C array is the key for the third element in the `values` array. Note that for creating dictionary objects you must specify initialized callback structures for both the keys and the values.

To create a mutable collection object you call the `CreateMutable` function appropriate to a given type. This call creates an empty (that is, valueless) collection to which you then add values.

```
CFMutableDictionaryRef myDictionary = CFDictionaryCreateMutable(NULL, 0,
&kCFCopyStringDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
CFDictionaryAddValue(myDictionary, CFSTR(“Age”), CFSTR(“35”));
```

A similar interface exists for creating mutable copies, except that these functions do not require you to specify callbacks. The reason for this omission—applicable to both mutable and immutable copies—is that the callbacks used by the original object are used by the copy to retain, release, compare, and describe its values.

```
/* props is an existing dictionary */

CFMutableArrayRef urls = CFArrayCreateMutableCopy(NULL, 0,
(CFArrayRef)CFDictionaryGetValue(props, kCFURLFileDirectoryContents));
```

In functions that create or copy mutable collection objects the second parameter is an integer that specifies the *capacity* of the collection, or the maximum number of values that the collection can safely store. A mutable collection with a capacity greater than 0 is said to be fixed-size. If this parameter is 0, as in the above example, the call requests a variable-size collection. A variable-size collection can contain any number of values, limited only by address space and available memory.

Defining Custom Collection Callbacks

All the code excerpts listed so far in this task show creation functions with one of the predefined collection callback structures specified in a parameter. However, you can define and use your own custom callback structures for your collection objects. There are at least two occasions when you might want to do this. One is when the values stored in the collection are custom data structures that require their own retaining, releasing, equality-testing, or descriptive behavior. Another occasion is when you want to use a predefined callback structure but need to modify an aspect of its behavior.

Listing 3 (page 15) shows an instance of the latter case. It defines a custom `CFDictionaryValueCallBacks` structure based on the predefined `kCFTypeDictionaryValueCallBacks` structure. But then it sets the `retain` and `release` function pointers to `NULL`. The values added to and removed from this collection will not be retained or released. This might be the desired behavior for some types of data.

Listing 3 Creating a CFDictionary object with modified predefined callbacks

```
CFMutableDictionaryRef bundlesByURL;

{CFDictionaryValueCallBacks nonRetainingDictionaryValueCallbacks =
kCFTypeDictionaryValueCallBacks;

nonRetainingDictionaryValueCallbacks.retain = NULL;

nonRetainingDictionaryValueCallbacks.release = NULL;

bundlesByURL = CFDictionaryCreateMutable(NULL, 0, &kCFTypeDictionaryKeyCallBacks,
 &nonRetainingDictionaryValueCallbacks);

/* assume url and bundle come from somewhere */

CFDictionarySetValue(bundlesByURL, url, bundle);
```

The extended code example in Listing 4 (page 16) illustrates the creation of a mutable CFDictionary object whose keys are integers and whose values are a program-defined structure; custom callbacks are defined for both value and keys.

Listing 4 Creating a CFDictionary object with custom value and key callbacks

```
typedef struct {
    int someInt;
    float someFloat;
} MyStructType;

const void *myStructRetain(CFAllocatorRef allocator, const void *ptr) {
    MyStructType *newPtr = (MyStructType *)CFAllocatorAllocate(allocator,
sizeof(MyStructType), 0);
    newPtr->someInt = ((MyStructType *)ptr)->someInt;
    newPtr->someFloat = ((MyStructType *)ptr)->someFloat;
    return newPtr;
}

void myStructRelease(CFAllocatorRef allocator, const void *ptr) {
    CFAllocatorDeallocate(allocator, (MyStructType *)ptr);
}

Boolean myStructEqual(const void *ptr1, const void *ptr2) {
    MyStructType *p1 = (MyStructType *)ptr1;
    MyStructType *p2 = (MyStructType *)ptr2;
    return (p1->someInt == p2->someInt) && (p1->someFloat == p2->someFloat);

}
CFStringRef myStructCopyDescription(const void *ptr) {
    MyStructType *p = (MyStructType *)ptr;
    return CFStringCreateWithFormat(NULL, NULL, CFSTR("[%d, %f]"), p->someInt,
p->someFloat);
}

Boolean intEqual(const void *ptr1, const void *ptr2) {
```

```
    return (int)ptr1 == (int)ptr2;
}

CFHashCode intHash(const void *ptr) {
    return (CFHashCode)((int)ptr);
}

CFStringRef intCopyDescription(const void *ptr) {
    return CFStringCreateWithFormat(NULL, NULL, CFSTR("%d"), (int)ptr);
}

void customCallBackDictionaryExample(void) {
   CFDictionaryKeyCallBacks intKeyCallBacks = {0, NULL, NULL, intCopyDescription,
 intEqual, intHash};
    CFDictionaryValueCallBacks myStructValueCallBacks = {0, myStructRetain,
myStructRelease, myStructCopyDescription, myStructEqual};
    MyStructType localStruct;
    CFMutableDictionaryRef dict;
    CFTypeRef value;
    /* Create a mutable dictionary with int keys and custom struct values
    ** whose ownership is transferred to and from the dictionary. */
    dict = CFDictionaryCreateMutable(NULL, 0, &intKeyCallBacks,
&myStructValueCallBacks);

    /* Put some stuff in the dictionary
    ** Because the values are copied by our retain function, we just
    ** set some local struct and pass that in as the value. */

    localStruct.someInt = 1000; localStruct.someFloat = -3.14;
    CFDictionarySetValue(dict, (void *)42, &localStruct);

    localStruct.someInt = -1000; localStruct.someFloat = -3.14;
    CFDictionarySetValue(dict, (void *)43, &localStruct);

    /* Because the same key is used, this next call ends up replacing the earlier
 value (which is freed). */
```

```
    localStruct.someInt = 44; localStruct.someFloat = -3.14;
    CFDictionarySetValue(dict, (void *)42, &localStruct);
    show(CFSTR("Dictionary: %@"), dict);

    value = CFDictionaryGetValue(dict, (void *)43);

    if (value) {
        MyStructType result = *(MyStructType *)value;
        CFStringRef description = myStructCopyDescription(&result);
        show(CFSTR("Value for key 43: %@"), description);
        CFRelease(description);
    }
    CFRelease(dict);
}
```

The collection types CFArray, CFDictionary, CFSet, and CFBag declare the following structure types for callbacks:

- `CFArrayCallBacks`
- `CFDictionaryKeyCallBacks`
- `CFDictionaryValueCallBacks`
- `CFSetCallBacks`
- `CFBagCallBacks`

The function-pointer members of these structures are similar in acceptable values, expected behavior of pointed-to functions, and caveats. Table 1 (page 19) describes some of the general characteristics of these callbacks; for detailed information, see the reference documentation for the callback structure types.

Table 1

Function- pointer variable	Collection type	Description of callback function
`retain`	All	Invoked to retain values as they are added to the collection. The nature of reference counting can vary according to the type of the data and the purpose of the collection; for example, it could increment a reference count. The function returns the value to store in the collection, which is usually the value passed, but can be a different value if that value should be stored. The function pointer can be `NULL`.
`release`	All	Invoked when values are removed from the collection. It reverses the effect of the `retain` callback by, for example, decrementing a reference count or freeing the memory allocated to a value. The function pointer can be `NULL`.
`equal`	all	A callback function that compares two values. It is invoked when some operation requires the comparison of values in the collection. For collection values, the function pointer can be `NULL`. Can be `NULL` for collection keys, in which case pointer equality is used as the basis of comparison.
`copyDescription`	all	A callback function that creates and returns a description (as a CFString object) of each value in the collection. This callback is invoked by the `CFCopyDescription` and `CFShow` functions. The function pointer can be `NULL`, in which case the collection composes a simple description.
`hash`	CFDictionary keys, CFSet, CFBag	A callback function invoked to compute a hash code for keys as they are used to access, add, or remove values in the collection. If `NULL` is assigned, the default implementation is to use the pointer addresses as hash codes. See Dictionaries (page 9) for more on the relation between the `equal` and `hash` function callbacks.

Getting the Values of Collections

As described in Common Characteristics of Collections (page 6), with CFArray, CFDictionary, CFSet, and CFBag collection objects, you use keys (as understood in a general sense) to retrieve stored values. The key used to retrieve values varies according to the type of collection.

- In CFArray objects the key is an index integer identifying relative position in the array.
- In CFDictionary objects the key is an arbitrary though constant piece of data associated with the value in the dictionary. The term "key" is conventionally used to designate this piece of associated data.
- In CFSet and CFBag objects the key is the value itself.

All collection objects define value-obtaining functions whose names contain the substring "GetValue". These functions take the key of the appropriate kind as a parameter.

For accessing the values of arrays, a common technique is to iterate over the collection in a loop, incrementing the index at each pass. Within the body of the loop you access a value using the current index as the key and test or use the value as necessary. Listing 1 (page 20) gives an example of this technique.

Listing 1 Getting the values in a CFArray object

```
if (URLs != NULL) { /* URLs is a CFArray object */
    CFIndex i, c = CFArrayGetCount(URLs);
    CFURLRef curURL;
    CFBundleRef curBundle;

    for (i=0; i<c; i++) {
        curURL = CFArrayGetValueAtIndex(URLs, i);
        curBundle = CFBundleCreate(alloc, curURL);
        if (curBundle != NULL) {
            CFArrayAppendValue(bundles, curBundle);
            CFRelease(curBundle);
        }
    }
    CFRelease(URLs);
```

```
}
```

The primary function for obtaining values in CFDictionary objects is `CFDictionaryGetValue`, which requires you to specify a value's key. Listing 2 (page 21) gives an example of this.

Listing 2 Getting a value in a CFDictionary object

```
CFStringRef theName = mappingTable ? (CFStringRef)CFDictionaryGetValue(mappingTable,
 (const void*)encoding) : NULL;
```

To get values from CFSet and CFBag objects, you must specify the value itself as the key. This might seem odd, but remember, you can define the callbacks that determine equality for created objects (`hash` and `equal`), so the value used as a key doesn't have to be exactly identical to a stored value.

CFSet, CFBag, and CFDictionary all define functions that get a specified value only if it exists in the collection. Because `NULL` can be a valid value in these collections, the `CF`*Type*`GetValueIfPresent` functions accurately report the existence of a contained value. Listing 3 (page 21) shows the application of the function `CFSetGetValueIfPresent` within a function that uses a CFSet object to ensure the uniqueness of strings.

Listing 3 Using a CFSet object to unique values

```
static CFMutableSetRef uniquedStrings = NULL;

CFStringRef uniqueString(CFStringRef string, Boolean addIfAbsent) {
    CFStringRef member = NULL;
    Boolean present;
    if (!string) {
        return NULL;
    }
    if (!uniquedStrings) {
        if (addIfAbsent) {
            uniquedStrings = CFSetCreateMutable(NULL, 0, & kCFTypeSetCallBacks);
        } else {
            return NULL;
        }
    }
    present = CFSetGetValueIfPresent(uniquedStrings, string, (void **)&member);
    if (!present) {
```

```
        if (addIfAbsent) {
            string = CFStringCreateCopy(NULL, string);
            CFSetAddValue(uniquedStrings, string);
            CFRelease(string);
        }
        member = string;
    }
    return member;
}
```

The collection types CFArray, CFDictionary, CFSet, and CFBag include other Get functions. Some functions obtain all values (and keys) in a collection, some return the count of values (or keys) in a collection, and some get the index or key associated with a specified value.

Searching Collections

Core Foundation includes several programming interfaces for finding values (and, in the case of CFDictionary, keys) in collection objects. The `CF`*Type*`GetValueIfPresent` functions, described in Getting the Values of Collections (page 20), report on the existence of values in dictionaries, sets, and bags. You can also use functions with "Contains" in their names to determine whether a collection holds a value or key. Listing 1 (page 23) illustrates how the `CFDictionaryContainsKey` function might be used.

Listing 1 Searching a CFDictionary object for a key

```
if (CFDictionaryContainsKey(mappingTable, (const void*)lowerCharsetName)) {
    result = (CFStringEncoding)CFDictionaryGetValue(mappingTable, (const
void*)lowerCharsetName);
}
```

For CFArray objects, the `CFArrayBSearchValues` function offers a more sophisticated search option. This function searches for a specified value in a sorted array using a binary search algorithm. If the value doesn't exist in the collection, the function tells you where it should go; the `CFArrayBSearchValues` function thus helps you to keep a sorted mutable array sorted. Listing 2 (page 23) shows how this function might be called.

Listing 2 Searching for a value in a CFArray object

```
CFIndex position = CFArrayBSearchValues(aMutArray,
CFRangeMake(0,CFArrayGetCount(anArray)), (const void *)CFSTR("String Three"),
CFStringCompare, 0);
```

The fourth parameter of the `CFArrayBSearchValues` function must be a pointer to a function that conforms to the `CFComparatorFunction` type. This comparator function is supposed to know how to compare values in the array. In the given example `CFStringCompare` is used because it conforms to `CFComparatorFunction` and it knows how to compare CFString values. There are other predefined Core Foundation comparator functions that you can use, such as `CFNumberCompare` and `CFDateCompare`.

Upon return of the above call, the function's `CFIndex` result can indicate one of the following:

- If the value exists, its index in the array

- An index greater than or equal to the end point of the range if the specified value is greater than all the values in the rangeThe index of the value greater than the specified value if the value lies between two of (or less than all of) the values in the range.

You can use the `CFArrayContainsValue` to determine whether the result is the first of the listed alternatives.

The CFTree type defines a different group of functions for locating contained values (that is, subtrees). See Creating and Copying Collections (page 13) for more information.

Working With Mutable Collections

The collection opaque types CFArray, CFDictionary, CFSet, and CFBag offer similar sets of functions for manipulating the values contained by mutable collections: adding values, removing values, replacing values, and so on. There are some differences in behavior based on whether the collection ensures the uniqueness of keys and values. Table 1 summaries the behavior of the mutability functions. The Operation column identifies the type of operation using the parameter found in a mutability function, such as "Replace" in `CFDictionaryReplaceValue`.

Table 1 Semantics of mutable collection operations

Operation	Collection Type	What it Means
Append	CFArray	Insert the value after all other values (index=count).
Insert	CFArray	Insert the value at the given index of the collection.
Add	all except CFArray	For CFDictionary and CFSet, add value if it is absent, do nothing if it is present. For CFBag, add value even if it is already present.
Replace	all	If the specified value is present, replace it with another value; otherwise, do nothing.
Set	all	Add the value if it is absent, replace it if it is present.
Remove	all	Remove the value if it is present, do nothing if it is absent.

With fixed-size mutable collections, you must take care to avoid adding beyond the capacity limit. A fixed-size collection will let you add as many values as you want, but gives no notice when you exceed the capacity. However, doing so will result in undefined behavior that is most likely undesirable.

The CFArray type features one mutability operation that is special to it. With the `CFArraySortValues` function you can sort the values contained by the array. A comparator function, which must conform to the `CFComparatorFunction` type, is used to compare values. Listing 1 gives an example of the use of the `CFArraySortValues` function.

Listing 1 Sorting an array

```
CFMutableArrayRef createSortedArray(CFArrayRef anArray) {
```

```
    CFIndex count = CFArrayGetCount(anArray);
    CFMutableArrayRef marray = CFArrayCreateMutableCopy(NULL, count, anArray);
    CFArraySortValues(marray, CFRangeMake(0, count),
(CFComparatorFunction)CFStringCompare, NULL);
    return marray;
}
```

Notice that the `CFStringCompare` function is used, in this case, to compare CFString objects. Core Foundation provides other comparator functions that are of the `CFComparatorFunction` type, notably `CFDateCompare` and `CFNumberCompare`. When an array holds Core Foundation objects, you can pass in an appropriate predefined comparator function to the `CFArraySortValues` function to sort those objects.

Applying Program-Defined Functions to Collections

A particularly useful feature of collections is the capability for applying a program-defined function to each value in a collection object. This applier function must conform to a prototype defined for each collection type. You must specify a pointer to this function in the collection functions (of the form `CF`*Type*`ApplyFunction`) that invokes it on each contained value.

Listing 1 (page 27) provides a simple example that applies a character-counting function to the CFString objects stored in a CFArray object. This function is of the type `CFArrayApplierFunction`. This prototype has two parameters: the first is a value in the array (or a pointer to that value) and the second is some program-defined value (or a pointer to that value).

Listing 1 Applying a function to an array

```
void countCharacters(const void *val, void *context) {
    CFStringRef str = (CFStringRef)val;
    CFIndex *cnt = (CFIndex *)context;
    CFIndex numchars = CFStringGetLength(str);
    *cnt += numchars;
}

void countCharsInArray() {
    CFStringRef strs[3];
    CFArrayRef anArray;
    CFIndex count=0;

    strs[0] = CFSTR("String One");
    strs[1] = CFSTR("String Two");
    strs[2] = CFSTR("String Three");

    anArray = CFArrayCreate(NULL, (void *)strs, 3, &kCFTypeArrayCallBacks);
    CFArrayApplyFunction(anArray, CFRangeMake(0,CFArrayGetCount(anArray)),
countCharacters, &count);
    printf("The number of characters in the array is %d", count);
```

```
    CFRelease(anArray);
}
```

Often an applier function is used to iterate over a mutable collection in order to remove objects matching certain criteria. It is never safe to mutate a collection while an applier function is iterating over it. However, there are some safe ways to use an applier function to mutate a collection:

- Mutate after iterating. Use the applied function to record where changes are needed in the collection, and then mutate the collection after an applier function finishes executing.
- Mutate the original. If the collection is mutable, make a copy of the collection and use an applier function to iterate over the copy and mutate the original.

Which approach is easier depends on the situation. If the original collection is immutable, then you can use a variation:

- Mutate a copy. Make a mutable copy of the collection and use an applier function to iterate over the original and mutate the copy.

Creating and Using Tree Structures

With the CFTree opaque type you can create tree structures in memory to represent hierarchical organizations of information. In such structures, each tree node has exactly one parent tree (except for the root tree, which has no parent) and can have multiple child trees. Each CFTree object in the extended structure has a context associated with it; this context includes some program-defined data as well as callbacks that operate on that data. The program-defined data is often used to store information about each node in the structure.

This task discusses how to create CFTree objects, fit them into the structure of trees, and then later find, get, and modify them. The first parameter of most CFTree functions is a reference to a *parent* CFTree object; the operation itself usually involves one or more children of that parent. In other words, most CFTree routines work "down" the hierarchy from a given parent but affect only the children of that parent.

One important fact to remember with working with CFTree objects is that functions that operate on children are not recursive. They affect only the immediate child trees. If you want to traverse all subtrees from any given parent node, your code must supply the traversal logic.

Creating a CFTree Object

Before you can create a CFTree object you must define its context. This means you must declare and initialize a structure of `CFTreeContext`. This structure has the following definition:

```
typedef struct {
    CFIndex version;
    void *info;
    const void *(*retain)(const void *info);
    void (*release)(const void *info);
    CFStringRef (*copyDescription)(const void *info);
} CFTreeContext;
```

The `info` member of this structure points to data that you define (and allocate, if necessary). The other members of the context structure (except for the version member) point to functions that take the `info` pointer as an argument and perform specific operations related to the pointed-to data, such as retaining it, releasing it, and describing it.

When you've properly initialized a `CFTreeContext` structure, call the `CFTreeCreate` function, passing in a pointer to the structure. Listing 1 (page 30) gives an example of this technique.

Listing 1 Creating a CFTree object

```
static CFTreeRef CreateMyTree(CFAllocatorRef allocator) {
    MyTreeInfo *info;
    CFTreeContext ctx;
    info = CFAllocatorAllocate(allocator, sizeof(MyTreeInfo), 0);
    info->address = 0;
    info->symbol = NULL;
    info->countCurrent = 0;
    info->countTotal = 0;
    ctx.version = 0;
    ctx.info = info;
    ctx.retain = AllocTreeInfo;
    ctx.release = FreeTreeInfo;
    ctx.copyDescription = NULL;
    return CFTreeCreate(allocator, &ctx);
}
```

As this example shows, you can initialize the function-pointer members of the `CFTreeContext` structure to `NULL` if you do not want to define callback functions for the CFTree object's context.

Adding a Tree to its Parent

To be of any use, a CFTree object must be inserted into the tree structure; it must be placed in a hierarchical relationship to other CFTree objects. To do this, you must use one of the following CFTree functions to make the object a child or sibling tree in relation to some other tree:

`CFTreeAppendChild`

`CFTreePrependChild`

`CFTreeInsertSibling`

Listing 2 (page 31) shows a child tree being appended to its parent's list of children.

Listing 2 Adding a child CFTree to its parent

```
/* assume anAddress and curTree already exist */
CFTreeRef child = FindTreeChildWithAddress(curTree, anAddress);if (NULL == child)
 {
    CFTreeContext ctx;
    child = CreateMyTree(CFGetAllocator(curTree));
    CFTreeGetContext(child, &ctx);
    ((MyTreeInfo *)ctx.info)->address = anAddress;
    CFTreeAppendChild(curTree, child);
    CFRelease(child);
}
```

The code example also illustrates another aspect of the CFTree programming interface. Sometimes you may need to modify the program-defined data associated with a CFTree object that is already created. To do this, call the `CFTreeGetContext` function on that object to get the tree's context. Once you have this structure, you can access the program-defined data through the `info` pointer.

Getting Child Trees

The CFTree type has several functions that obtain child trees. Because sibling trees are in sequential order, you typically use only two of these methods to traverse the child trees of one parent, `CFTreeGetFirstChild` and `CFTreeGetNextSibling`. Listing 3 (page 31) shows how you do this.

Listing 3 Traversing the child trees of a parent tree

```
static CFTreeRef FindTreeChildWithAddress(CFTreeRef tree, UInt32 addr) {
    CFTreeRef curChild = CFTreeGetFirstChild(tree);
    for (; curChild; curChild = CFTreeGetNextSibling(curChild)) {
        CFTreeContext ctx;
        CFTreeGetContext(tree, &ctx);
        if (((MyTreeInfo *)ctx.info)->address == addr) {
            return curChild;
        }
    }
    return NULL;
}
```

Not all CFTree functions act on or return child trees. The `CFTreeGetParent` function, for example, obtains the parent tree of a given tree. The `CFTreeFindRoot` function obtains the root CFTree object of the current tree structure—that is, the tree of the structure that has no parent tree.

Other Operations With CFTree Structures

The CFTree type includes two functions that are very similar to other collection functions. The `CFTreeApplyFunctionToChildren` function applies a program-defined applier function to the children of a parent CFTree object. The `CFTreeSortChildren` function sorts the children of a parent CFTree object using a comparator function that you can define (or employ, as long as it conforms to the `CFComparatorFunction` type).

- See Applying Program-Defined Functions to Collections (page 27) for usage details pertinent to `CFTreeApplyFunctionToChildren`.
- See Working With Mutable Collections (page 25) for information on using the `CFArraySortValues` function; much of this information is applicable to the `CFTreeSortChildren` function.

Document Revision History

This table describes the changes to *Collections Programming Topics for Core Foundation*.

Date	Notes
2011-01-18	Corrected typographical errors.
2003-08-07	Fixed errors in sample code.
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