Core Foundation Design Concepts

2005-08-11

Contents

Introduction to Design Concepts 7

Organization of This Document 7

About Core Foundation 9

Software Layers 10
Data Sharing 10
Operating-System Independence 10
Internationalization 11

Opaque Types 13

Advantages of Opaque Types 14

Object References 15

Polymorphic Functions 17

Varieties of Objects 19

Naming Conventions 21

Other Types 23

Comparing Objects 25

Inspecting Objects 27

Document Revision History 29

Figures and Listings

About Core Foundation 9

Figure 1	Core Foundation and Carbon on Mac OS 9 9
Figure 2	Core Foundation and other software layers on Mac OS X 10

Opaque Types 13

Figure 1	An opaque type 13

Comparing Objects 25

Listing 1	Comparing Core Foundation objects 25

Inspecting Objects 27

Listing 1	Using CFCopyDescription 27

Introduction to Design Concepts

Core Foundation supports the transition from earlier Mac OS to Mac OS X systems. The evolution of the Mac OS into Mac OS X requires the participation of application developers. By slightly modifying your current code to use new programming interfaces and techniques, your applications can run smoothly on a radically different operating system, one with features such as protected memory and preemptive multitasking.

Organization of This Document

The programming interfaces of Core Foundation objects have been designed for ease of use and reuse. Before you can reach any level of comfortable competency with these programming interfaces, however, you should understand a few concepts on which Core Foundation is based. If you are new to Core Foundation then read this topic as well as "Memory Management" before using Core Foundation objects in your code.

This concept discusses the Core Foundation architecture and its benefits:

- "About Core Foundation" (page 9)

These concepts and tasks discuss the object model used in Core Foundation:

- "Opaque Types" (page 13)
- "Object References" (page 15)
- "Polymorphic Functions" (page 17)
- "Varieties of Objects" (page 19)
- "Comparing Objects" (page 25)
- "Inspecting Objects" (page 27)

In addition, there are other non-object types, and API conventions that you want to be familiar with before using Core Foundation:

- "Naming Conventions" (page 21)
- "Other Types" (page 23)

About Core Foundation

Core Foundation is a library with a set of programming interfaces conceptually derived from the Foundation framework of the Cocoa object layer but implemented in the C language. To do this, Core Foundation implements a limited object model in C. Core Foundation defines opaque types that encapsulate data and functions, hereafter referred to as "objects."

At a general level, Core Foundation

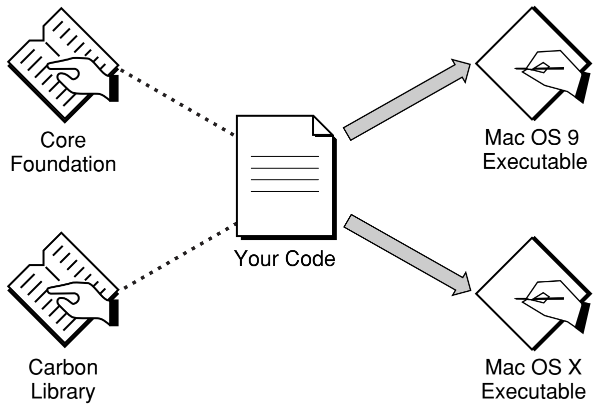
- enables sharing of code and data among various frameworks and libraries
- makes some degree of operating-system independence possible
- supports internationalization with Unicode strings
- provides common API and other useful capabilities, including a plug-in architecture, XML property lists, and preferences

It offers developers many fundamental software services on several platforms:

- Mac OS 9 (when developing with the Carbon library)
- Mac OS X (when developing with either Carbon or Cocoa)

As Figure 1 (page 9) illustrates, Core Foundation on Mac OS 9 is a library that, when used together with Carbon, enables you to develop applications that can run on Mac OS 9, and Mac OS X.

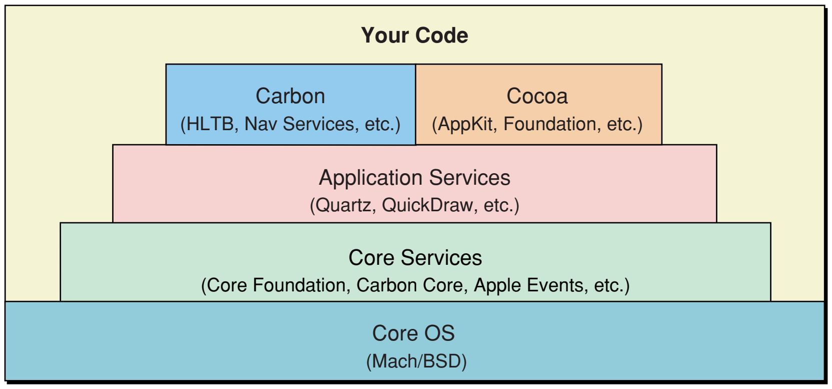
Figure 1 Core Foundation and Carbon on Mac OS 9



Software Layers

On Mac OS X, you can think of Core Foundation as part of the substrata of system software called Core Services. This layer is immediately above the core operating system and below the services, frameworks and libraries used in application development. Figure 2 (page 10) depicts these relationships.

Figure 2 Core Foundation and other software layers on Mac OS X


Your Code
Carbon
(HLTB, Nav Services, etc.)
Cocoa
(AppKit, Foundation, etc.)
Application Services
(Quartz, QuickDraw, etc.)
Core Services
(Core Foundation, Carbon Core, Apple Events, etc.)
Core OS
(Mach/BSD)

Data Sharing

Core Foundation makes it possible for the different frameworks and libraries on Mac OS X to share code and data. Carbon, Cocoa, and Mac OS 9 applications, libraries, and frameworks can define C routines that incorporate Core Foundation types in their external interfaces; they can thus communicate data—as Core Foundation objects—to each other through these interfaces. Indeed a Carbon developer could, for example, provide a Core Foundation-based service that can be used by any running application on a Mac OS X system regardless of its origin.

Core Foundation also provides “toll-free bridging” between certain services and the Cocoa’s Foundation framework. Toll-free bridging enables you to substitute Cocoa objects for Core Foundation objects in function parameters and vice versa.

Operating-System Independence

Some Core Foundation types and functions are abstractions of things that have specific implementations on different operating systems. Code that makes use of these APIs is thus easier to port to different platforms.

Date and number types abstract time utilities and offers facilities for converting between absolute and Gregorian measures of time. It also abstracts numeric values and provides facilities for converting between different internal representations of those values.

Several other services that abstract operating-system utilities are available to Mac OS X native applications but not Carbon applications. Among these are inter-process notification and run-loop services.

Internationalization

One of the major benefits Core Foundation brings to application development is internationalization support. Through its String objects, Core Foundation facilitates easy, robust, and consistent internationalization across all Mac OS X and Cocoa programming interfaces and implementations. The essential part of this support is a type, CFString, instances of which represent an array of 16-bit Unicode characters. A CFString object is flexible enough to hold megabytes worth of characters and yet simple and low-level enough for use in all programming interfaces communicating character data. It accomplishes this with performance not much different than that associated with standard C strings.

Core Foundation String objects (which defines CFString) has dozens of associated functions that do expected things with strings such as comparing, inserting, and appending strings, and searching for substrings. String objects also provides functions that convert Unicode strings (that is, CFString objects) to and from other encodings, particularly 8-bit encodings stored as Pascal and C strings. Because most strings in programs today are 8-bit, a CFString object uses less memory for storing such strings whenever possible.

Opaque Types

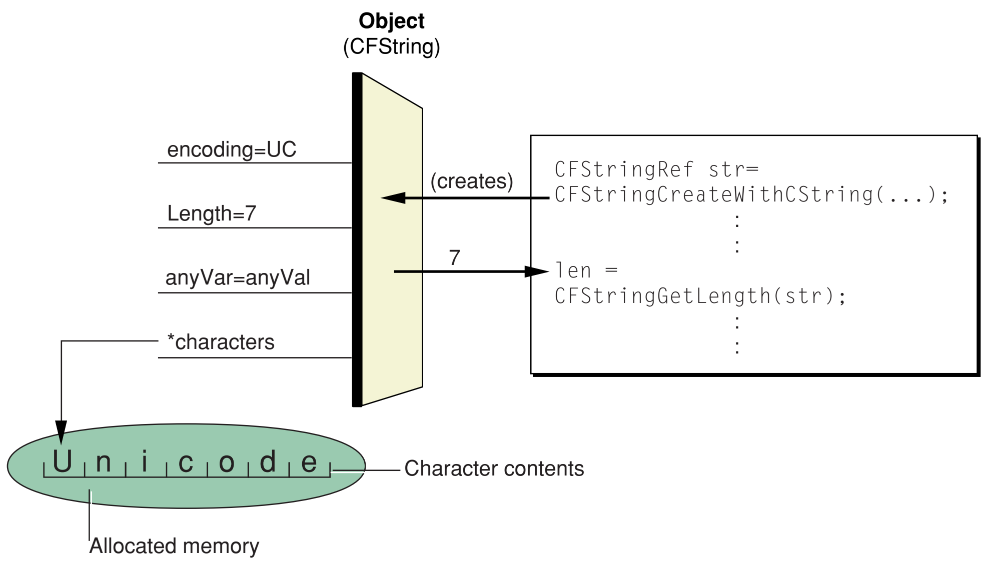
The Core Foundation's object model that supports encapsulation and polymorphic functions is based on opaque types.

The individual fields of an object based on an opaque type are hidden from clients, but the type's functions offer access to most values of these fields. Figure 1 (page 13) depicts an opaque type in the data it "hides" and in the interface it presents to clients.

Note: "Class" is not used to refer to opaque types because, despite the conceptual similarity of class and opaque type, many might find the term confusing. However, the Core Foundation documentation frequently refers to specific, data-bearing instances of these types as "objects."

Core Foundation has many opaque types, and the names of these types reflect their intended uses. For example, CFString is an opaque type that "represents" and operates on Unicode character arrays. ("CF" is, of course, a prefix for Core Foundation.) CFArray is an opaque type for indexed-based collection functionality. The functions, constants, and other secondary data types in support of an opaque type are generally defined in a header file having the name of the type; `CFArray.h`, for example, contains the symbol definitions for the CFArray type.

Figure 1 An opaque type



```
CFStringRef str=
CFStringCreateWithCString(...);
            :
            :
len =
CFStringGetLength(str);
            :
            :
```

Advantages of Opaque Types

To some, an opaque type might seem to impose an unnecessary limitation by discouraging direct access of the structure's contents. There also might seem to be overhead associated with opaque types that could affect program performance. But the benefits of opaque types outweigh these seeming limitations.

Opaque types provide better abstraction and more flexibility for how the underlying functionality is implemented. By hiding details such as the fields of structures, Core Foundation reduces the chance for errors that might occur in client code when those details change. Moreover, opaque types permit optimizations that might be confusing if exposed. For example, CFString "officially" represents an array of 16-bit characters of the type `UniChar`. However, a CFString might choose to store a range of characters in the ASCII range as 8-bit values. Copying an immutable object might (and usually does) result in a shared reference to the object instead of two separate objects in memory (see "Memory Management").

Continuing with the example of CFString, it might seem heavyweight to use an opaque type to store characters. As it turns out, however, the CPU cost of such storage is not much higher than using a simple C array of characters and the memory cost is often less. In addition, opacity does not necessarily mean that an opaque type can never provide mechanisms for accessing content directly. CFString, for instance, provides the `CFStringGetCStringPtr` function for this purpose.

Finally, you can customize some opaque types to some degree. For example, the collection types allow you to define callbacks for invoking a function on every member of a collection.

Object References

You refer to Core Foundation objects (opaque types) through references. In every header file for an opaque type, you will notice a line or two similar to the following:

```
typedef const struct __CFArray * CFArrayRef;
typedef struct __CFArray * CFMutableArrayRef;
```

Declarations such as these are pointer references to immutable and mutable versions of the (private) structure defining the opaque type. The parameters and return values of many Core Foundation functions take the type of these object references and never a `typedef` of the private structure. For example:

```
CFStringRef CFStringCreateByCombiningStrings(CFAllocatorRef alloc, CFArrayRef
array, CFStringRef separatorString);
```

See "Varieties of Objects" (page 19) for more on immutable, mutable, and other variants of opaque-type objects.

Every Core Foundation opaque type defines a unique type ID for its objects, as in `CFArrayRef` above for CFArray objects. A type ID is an integer of type `CFTypeID` that identifies the opaque type to which a Core Foundation object "belongs." You use type IDs in various contexts, such as when you are operating on heterogeneous collections. Core Foundation provides programmatic interfaces for obtaining and evaluating type IDs.

Important: Because the value for a type ID can change from release to release, your code should not rely on stored or hard-coded type IDs nor should it hard-code any observed properties of a type ID (such as, for example, it being a small integer).

In addition, Core Foundation defines a generic object-reference type, `CFTypeRef`, analogous to a root class in some object-oriented programming languages. This generic reference serves as a placeholder type for parameters and returned values of polymorphic functions, which can take references to any Core Foundation object. See "Polymorphic Functions" (page 17) for more on this subject. See "Memory Management" for issues relating to memory management when using object references.

Polymorphic Functions

Core Foundation provides several polymorphic functions. These functions can take any Core Foundation object as a parameter and (in one instance, `CFRetain`) can return any Core Foundation object. These parameters and return values are given the type of `CFTypeRef`, a generic object-reference type. CFType is analogous to a root class in object-oriented languages because its functions can be reused by all other objects.

You use polymorphic functions for operations that are common to all Core Foundation objects:

- Reference counting.

  CFType provides several polymorphic functions for manipulating and obtaining the reference count of objects. See “Memory Management” for more about these functions.

- Comparing objects.

  The `CFEqual` function compares any two Core Foundation objects (see "Comparing Objects" (page 25)). The basis of equality depends on the type of objects compared. For example, if both are CFString objects the test involves a character-by-character comparison.

- Hashing objects.

  The `CFHash` function returns a unique hash code identifying a Core Foundation object (see "Comparing Objects" (page 25)). You can use the hash code as a table address in a hash table structure. If two objects are equal (as determined by the `CFEqual` function), they must have the same hash value.

- Inspecting objects.

  CFType gives you the means to inspect objects and thereby learn about their contents and the type to which they “belong.” The `CFCopyDescription` function returns a string (more precisely, a reference to a CFString object) that describes an object. The `CFCopyTypeIDDescription` function, which takes a `CFTypeID` rather than a `CFTypeRef` parameter, returns a string reference that describes the opaque type identified by the type ID. These functions are primarily intended to assist debugging; see "Inspecting Objects" (page 27) for more on these functions.

  You can also determine the opaque type to which a generically typed object belongs by getting its type ID with the `CFGetTypeID` function and then comparing that value with known type IDs. See "Inspecting Objects" (page 27) for more on this task.

Varieties of Objects

Opaque types come in up to three basic varieties, or "flavors," based on the characteristics of editability and expandability expected in their objects:

- immutable and fixed size
- mutable and fixed size
- mutable and variable size

Mutable objects are editable, meaning their contents can be changed. Immutable objects are not editable; once they are created they cannot be changed. Any attempt to change an immutable object usually results in an error of some kind. A fixed-size object has a maximum limit that it can grow to; in the case of a CFString, that would be the number of characters, and for a collection the limit would be the number of elements.

Some opaque types, such as CFString and CFArray, can create all three flavors of objects. Most opaque types can create immutable, fixed-size objects and typically have at least one unqualified creation function to do the job (such as `CFArrayCreate`). The determinant for mutable fixed-size versus variable-size is the value of the capacity or maximum-length parameter in the *Type*`CreateMutable` function; any positive value results in a fixed-size object, but a 0 specifies a variable-size object.

References to mutable objects include "Mutable" in the type name, for example, `CFMutableStringRef`.

Naming Conventions

A major programming-interface convention in Core Foundation is to use the name of the opaque type that is most closely related to a symbol as the symbol's prefix. For functions, this prefix identifies not only the type to which the function "belongs" but usually the type of object that is the target of the function's action. (An exception to this convention are constants, which put "k" before the type prefix.) Here are a few examples from the header files:

```
/* from CFDictionary.h */
CF_EXPORT CFIndex CFDictionaryGetCountOfKey(CFDictionaryRef dict, const void
*key);
/* from CFString.h */
typedef UInt32 CFStringEncoding;
/* from CFCharacterSet.h */
typedef enum {
    kCFCharacterSetControl = 1,
    kCFCharacterSetWhitespace,
    kCFCharacterSetWhitespaceAndNewline,
    kCFCharacterSetDecimalDigit,
    kCFCharacterSetLetter,
    kCFCharacterSetLowercaseLetter,
    kCFCharacterSetUppercaseLetter,
    kCFCharacterSetNonBase,
    kCFCharacterSetDecomposable,
    kCFCharacterSetAlphaNumeric,
    kCFCharacterSetPunctuation,
    kCFCharacterSetIllegal
} CFCharacterSetPredefinedSet;
```

Core Foundation has a few programming-interface conventions in addition to those related to opaque types and memory management.

- There is an important distinction between Get, and Copy and Create, in names of functions that return values. If you use a Get function, you cannot be certain of the returned object's life span. To ensure the persistence of such an object you can retain it (using the CFRetain function) or, in some cases, copy it. If you use a Copy or Create function, you are responsible for releasing the object (using the CFRelease function). For more details, see "Memory Management".
- Some Core Foundation objects have their own naming conventions to impose consistency among common operations. For example, collections embed the following verbs in function names to mean specific operations on the elements of a collection:
    - "Add" means "add if absent, do nothing if present" (if a uniquing collection).
    - "Replace" means "replace if present, do nothing if absent."
    - "Set" means "add if absent, replace if present."
    - "Remove" means "remove if present, do nothing if absent."

- The `CFIndex` type is used for index, count, length, and size parameters and return values. The integer value this type represents (currently 32 bits) can grow over time as the processor's address size grows. On architectures where pointer sizes are different, say 64 bits, `CFIndex` might be declared to be 64 bits, independent of the size of `int`. By using `CFIndex` for variables that interact with Core Foundation arguments of the same type, you ensure a higher degree of source compatibility for your code.
- Some Core Foundation header files may seem to define opaque types but actually contain convenience functions not associated with a specific type. A case in point is `CFPropertyList.h`. CFPropertyList is a placeholder type for any of the property-list types: CFString, CFData, CFBoolean, CFNumber, CFDate, CFArray, and CFDictionary.
- Unless otherwise specified, all by-reference parameters intended for the return of values can accept `NULL`. This indicates that the caller is not interested in that return value.

Other Types

Core Foundation defines a number of data types for general use in functions. The purpose of some of these types is to abstract primitive values that might have to change as the processor address space changes. The CFIndex type, for example, is used in index, count, length, and size parameters. The CFOptionFlags type is used for bitfield parameters and the CFHashCode type holds hashing results returned from the `CFHash` function and certain hashing callbacks.

Other base types are used in functions that take and return comparison and range values. CFRange is a structure that specifies any part of a linear sequence of items, from characters in a string to elements in a collection. For comparison functions, the CFComparisonResult type defines `enum` constants to represent appropriate return values (equal, less than, greater than). Some Core Foundation functions take callbacks to comparator functions; if you want a custom comparator, the function must conform to the signature specified by the CFComparatorFunction type.

Important: The integer value certain Core Foundation types, particularly CFIndex and CFTypeID, can grow over time as the processor's address size grows. By using the base types for variables that interact with Core Foundation arguments of the same type, you will ensure a higher degree of source compatibility for your code.

Other opaque types provided by Core Foundation are discussed in separate topics.

Comparing Objects

You compare two Core Foundation objects with the `CFEqual` function. If the two objects are essentially equal, the function returns a boolean true value. "Essential" equality depends on the type of objects compared. For example, when you compare two CFString objects, Core Foundation considers them essentially equal when they match character by character, regardless of their encodings or mutability attribute. Two CFArray objects are considered equal when they have the same count of elements *and* each element object in one array is essentially equal with its counterpart in the other array. Obviously, compared objects must be of the same type (or a mutable or immutable variant of the same type) to be considered equal.

The following code fragment shows how you might use the `CFEqual` function to compare a constant with a passed-in parameter:

Listing 1 Comparing Core Foundation objects

```
void stringTest(CFStringRef myString) {
    Boolean equal = CFEqual(myString, CFSTR("Kalamazoo"));
    if (!equal) {
        printf("They're not equal!");
    }
    else {
        printf("They're equal!"):
    }
}
```

Inspecting Objects

A primary characteristic of Core Foundation objects is that they're based on an opaque (or private) type; it is thus difficult to inspect the internal data of an object directly. Base Services, however, provide two functions with which you can inspect Core Foundation objects. These functions return descriptions of an object and of the object's type.

To find out the contents of a Core Foundation object, call the `CFCopyDescription` function on that object and then print the character sequence "contained" in the referred-to string object:

Listing 1 Using CFCopyDescription

```
void describe255(CFTypeRef tested) {
    char buffer[256];
    CFIndex got;
    CFStringRef description = CFCopyDescription(tested);
    CFStringGetBytes(description,
        CFRangeMake(0, CFStringGetLength(description)),
        CFStringGetSystemEncoding(), '?', TRUE, buffer, 255, &got);
    buffer[got] = (char)0;
    fprintf(stdout, "%s", buffer);
    CFRelease(description);
}
```

This example shows just one approach for printing a description. You could use CFString functions other than `CFStringGetBytes` to get the actual string.

To determine the type of an "unknown" object, obtain its type ID with the `CFGetTypeID` function and compare that value with known type IDs until you find a match. You obtain an object's type ID with the `CFGetTypeID` function. Each opaque type also defines a function of the form `CF`*Type*`GetTypeID` (for example, `CFArrayGetTypeID`); this function returns the type ID for that type. Therefore, you can test whether a CFType object is a member of a specific opaque type as in:

```
CFTypeID type = CFGetTypeID(anObject);
if (CFArrayGetTypeID() == type)
    printf("anObject is an array.");
else
    printf("anObject is NOT an array.");
```

To display information about the type of a Core Foundation object in the debugger, use the `CFGetTypeID` function to get its type ID, then pass that value to the `CFCopyTypeIDDescription` function:

```
/* aCFObject is any Core Foundation object */
CFStringRef descrip = CFCopyTypeIDDescription(CFGetTypeID(aCFObject));
```

Note: String Services include two functions, both declared in `CFString.h`, that you can call in supported debuggers to print descriptions of Core Foundation objects: `CFShow` and `CFShowStr`.

Important: The `CFCopyDescription` and the `CFCopyTypeIDDescription` functions are for debugging only. Because the information in the descriptions and their format are subject to change, do not create dependencies on them in your code.

Document Revision History

This table describes the changes to *Core Foundation Design Concepts*.

Date	Notes
2005-08-11	Included a link to the memory management article from the "Object References" and "Naming Conventions" sections.
2004-11-02	Correction of minor typographical error.
2003-01-17	Converted existing Core Foundation documentation into topic format. Added revision history.