Welcome

To Advance through Presentation
Use Page Up and Page Down Keys

CoreFoundation Overview

Ali Ozer and Chris Kane

Application Frameworks Engineering

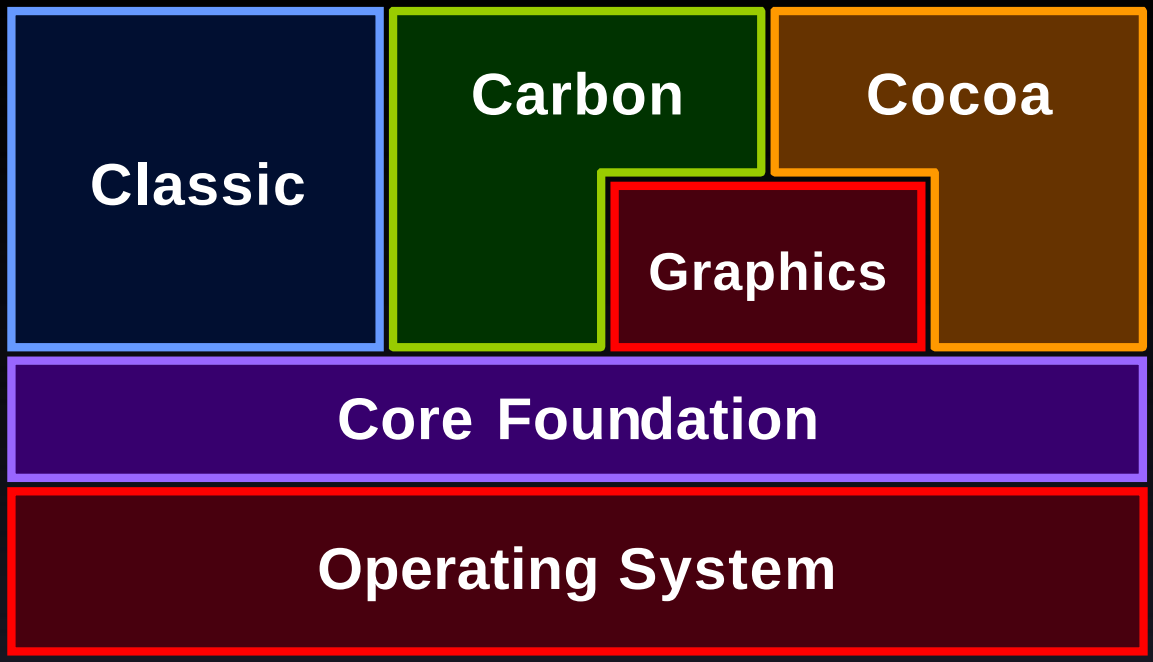
Overview

- What is CoreFoundation?
- Main features of CoreFoundation
- Various paradigms in CoreFoundation

What Is CoreFoundation?

- A new set of C APIs in Mac OS X and 8
- Substrate for the implementation of Carbon and Cocoa toolkits
- Also known as “CF”

Where CF Fits



Design Goals of CF

- Act as common substrate beneath Carbon, Classic, and Cocoa
- High performance
- Portability
- Consistency in API and semantics
- Use some object-oriented paradigms

However. . .

- Not meant to be extensible by developers
- Not complete object-oriented experience
  - No inheritance, limited polymorphism
- Not protective
  - Production binary does exactly what you tell it to do
  - Debug library has assertions and checking

Where Is CF Available?

- Mac OS X
- Sonata via CarbonLib
- Mac OS 8 via CarbonLib

Why Should You Use It?

- Access to new functionality
- Common low level abstractions
- Internationalization of Carbon apps
- Platform independence

Main Features

Main Features

- Strings and character sets
- Collections and other data types
- Property lists
- Preferences
- XML parsing
- URLs

CFString

- Conceptually an array of Unicode characters
- Goals
  - Elevate strings to a new level of abstraction
  - Make internationalization easy
  - Assure high performance
  - Become the way to communicate strings in APIs
  - Provide convenient and efficient bridging to C and Pascal string APIs

CFString

- Rich functionality
  - Many creation functions
  - Encoding conversion
  - Comparison, find
  - Explode, combine
  - Format, parse

CFString

- Some basic creation and access functions
  - **CFStringCreateWithCharacters(allocator, buffer, length)**
  - **CFStringCreateWithPascalString(allocator, pStr, encoding)**
  - **CFStringGetLength(str)**
  - **CFStringGetCharacterAtIndex(str, index)**
  - **CFStringGetCharacters(str, range, buffer)**
  - **CFStringGetPascalString(str, buffer, size, encoding)**
- “Constant” strings
  - **CFSTR(“Hello World”)**

CFString

- Can provide hints for not copying contents

  CFStringCreateWithPascalStringNoCopy(allocator, pStr, encoding, contentsAllocator)

  - This might copy

  CFStringGetPascalStringPtr(str, encoding)

  - This might return NULL

- Other, more sophisticated and predictable "no copy" functionality also available

CFString

- Provides powerful editing capabilities

  CFStringAppend(str, appendedStr)

  CFStringReplace(str, range, replaceStr)

  CFStringTrim(str, trimStr)

CFString

- Sample CFString usage in Carbon Toolbox
  - Today

    void SetWTitle(WindowPtr, ConstStr255Param)
    void GetWTitle(WindowPtr, Str255)
  - To be added (exact names TBD)

    void SetWTitleCFString(WindowPtr,CFStringRef)
    CFStringRef GetWTitleCFString(WindowPtr)

CFCharacterSet

- Set of Unicode characters
- Functions to add and remove characters, test for presence, evaluate union and intersection
- Many predefined sets
    - kCFCharacterSetWhitespace
    - kCFCharacterSetAlphaNumeric
    - kCFCharacterSetPunctuation

Collections

- Containers for pointer-sized values
  - CFArray
    - Compact, ordered vector
  - CFDictionary
    - Mapping from unique keys to values
  - CFSet
    - Unordered unique set of values
  - CFBag
    - Unordered non-unique set of values

Collections

- Other collections
  (implemented in future release)
  - CFBitVector
  - CFBinaryHeap
  - CFTree

Collections

- Callbacks configure behavior
- Default sets of callbacks provided for putting CF Types in collections
- Allocators allow memory usage to be controlled

CFArray Example

```
CFArrayRef array =
  CFArrayCreateMutable(NULL, 0,
        &kCFTypeArrayCallBacks);

CFStringRef value = CFSTR(“Apple”);
CFArrayAppend(array, value);
count  = CFArrayGetCount(array);
/* count is now one */
```

CFDictionary Example

```
CFDictionaryRef dict  =
  CFDictionaryCreateMutable(NULL, 0,
        &kCFTypeDictionaryKeyCallBacks,
        &kCFTypeDictionaryValueCallBacks);

CFStringRef key = CFSTR(“Name”);
CFStringRef value = CFSTR(“Apple”);
CFDictionarySetValue(dict, key, value);
count  = CFDictionaryGetCount(dict);
/* count is now one */
```

Other Data Types

- CFData
  - A sequence or chunk of bytes
- CFDate, CFTimeZone
  - Time and date facilities
- CFNumber, CFBoolean
  - Wrappers for property lists

Property Lists

- A tree of instances of these CF Types:
  - CFArray, CFDictionary, CFData, CFString, CFDate, CFNumber, and CFBoolean
- Have a flattened XML representation
  - Can convert to and from a CFData
- Not a graph, and does not contain instances of other types

Property Lists

- Property lists are used for…
  - Configuration files
    - Applications, plug-ins, system parameters
  - Storage for user preferences and settings

Preferences

- Save and retrieve preference settings
- Stored per user, per host, and per application
- Any property list CF Type can be used as value
- User preferences stored as XML

Preferences

- API Sample
  - Primitive

    ```
    CFPreferencesSetValue(key, value, appName,
        user, host)
    ```

  - Convenience

    ```
    CFPreferencesSetAppValue(key, value)
    ```

- Use “App” Preferences API whenever possible

XML Parsing

- Configurable XML parser
- Both simple API and low-level, more powerful API
    - Simple API parses to and from a CFTree
    - Low-level API allows for precise control via callbacks
- Available in Developer Preview 2

URLs

- Uniform way to represent “things”
- Like CFStrings, work with Unicode characters
- Used throughout APIs where files are referred to
- Future plug-in API will allow adding support for new URL schemes

Other CF Features

- CFBundle
  - Represents an executable package
- CFPlugIn
  - Abstraction for plug-ins
- CFNotificationCenter
  - Cross-process notifications
- CFRunLoop
  - Input and event handling

Paradigms

CF Paradigms

- Consistency conventions
- Object-orientation
- Flavors of CF Types
- Callbacks
- Memory ownership
- Memory allocation
- Thread-safety

Consistency Conventions

- For consistency, naming conventions
  - “CF” prefix, “kCF” prefix for constants
  - These prefixes, and with leading underbars, are reserved by Apple
  - Symbols starting with underbar are private to Apple, and must not be used!

Consistency Conventions

- More naming conventions
  - Opaque CF Types have “Ref” suffix, and are not used as pointers

```
typedef struct __CFFoo *CFFooRef;
CFFooRef myFoo;
```

  - Functions: type name, then operation
    - CFArraySort
    - CFStringGetLength
    - CFDataGetBytes

Consistency Conventions

- More naming conventions
  - First argument usually instance:

```
CFArraySort(CFArrayRef array, ...)
CFStringGetLength(CFStringRef str)
CFDataGetBytes(CFDataRef data, ...)
```

  - Verbs have consistent meanings
    - Like add, replace, get, set, remove…

Consistency Conventions

- Arguments tend to fall into predictable patterns
    - Allocator arguments are always first
    - “Out” parameters tend to be at the end
- Memory ownership has naming conventions
- When you know the conventions, the API is predictable

Object-Orientation

- Encapsulation, or opaque types, is used heavily
- No CF Types have public data members
- All CF Type instances can be used in the polymorphic functions
    - CFRetain(), CFRelease(), CFGetRetainCount(), CFEqual(), CFHash(), CFCopyDescription(), CFGetAllocator()

Three CF Type Flavors

- Immutable: Contents fixed, size fixed
- Fixed-size: Contents changeable, maximum size is fixed
- Mutable: Contents changeable, size is dynamic

Three Flavors

- Some CF Types come in all three flavors
  - CFString, CFArray, CFDictionary, CFData…
- Some CF Types have only one flavor
  - Immutable: CFDate, CFNumber
  - Mutable or Fixed-Size: CFBinaryHeap
  - Mutable: CFTree

Callbacks

- Callback functions are used heavily
- Allow for parameterization of types and operations
    - Callout when a value is added to or removed from a collection
    - Callout when something happens to an instance
    - Comparison function used for sorting

Memory Ownership

- Reference counting used to manage memory ownership
    - CFRetain(value) and CFRelease(value) work on any CF Type instance
- Must release a reference you receive, at some point
- You can create a reference by retaining

Memory Ownership

- The verb in the function name indicates how memory is being returned
  - “Get”: value returned without reference
  - “Copy”: value returned with reference
  - “Create”: value returned with reference
- “Create” functions take an allocator argument, “Copy” functions do not

Memory Allocation

- CFAllocator CF Type to encapsulate memory allocation functionality
- Allocators used by CF to allocate memory
- Custom allocators can be created
- There is a default, per-thread allocator
    - Specified with “NULL” as an allocator argument
    - Previous default allocator should be saved and restored if you change it

Thread-safety

- Simple, basic things are not thread-safe
  - Collections, strings, dates…
  - Immutable instances are thread-safe automatically
- More complex services are thread-safe
  - Preferences, input handling facilities…
- Internal global data is always thread-safe
- Reference-counting system is thread-safe

Wrap-Up

- To recap:
  - A new set of C APIs in Mac OS X and 8
  - Substrate for implementation of many things
  - Use it when you need or want access to the new functionality or where other APIs use CF types
- See release notes and examples on system

Other Sessions

CoreFoundation: Plug-Ins A CF-based plug-in model	Hall A1 **Fri., 4:00pm**
Carbon Overview (Repeat) What is Carbon up to?	Hall C **Fri., 9:00am**
Intro to the Cocoa (Yellow) Framework What is Cocoa?	Hall B **Fri., 9:00am**

Q&A

Think different.™

Welcome

To Advance through Presentation
Use Page Up and Page Down Keys