Topics in CoreFoundation

Christopher Kane
Mac OS X Application Frameworks
Apple, Inc.

Introduction

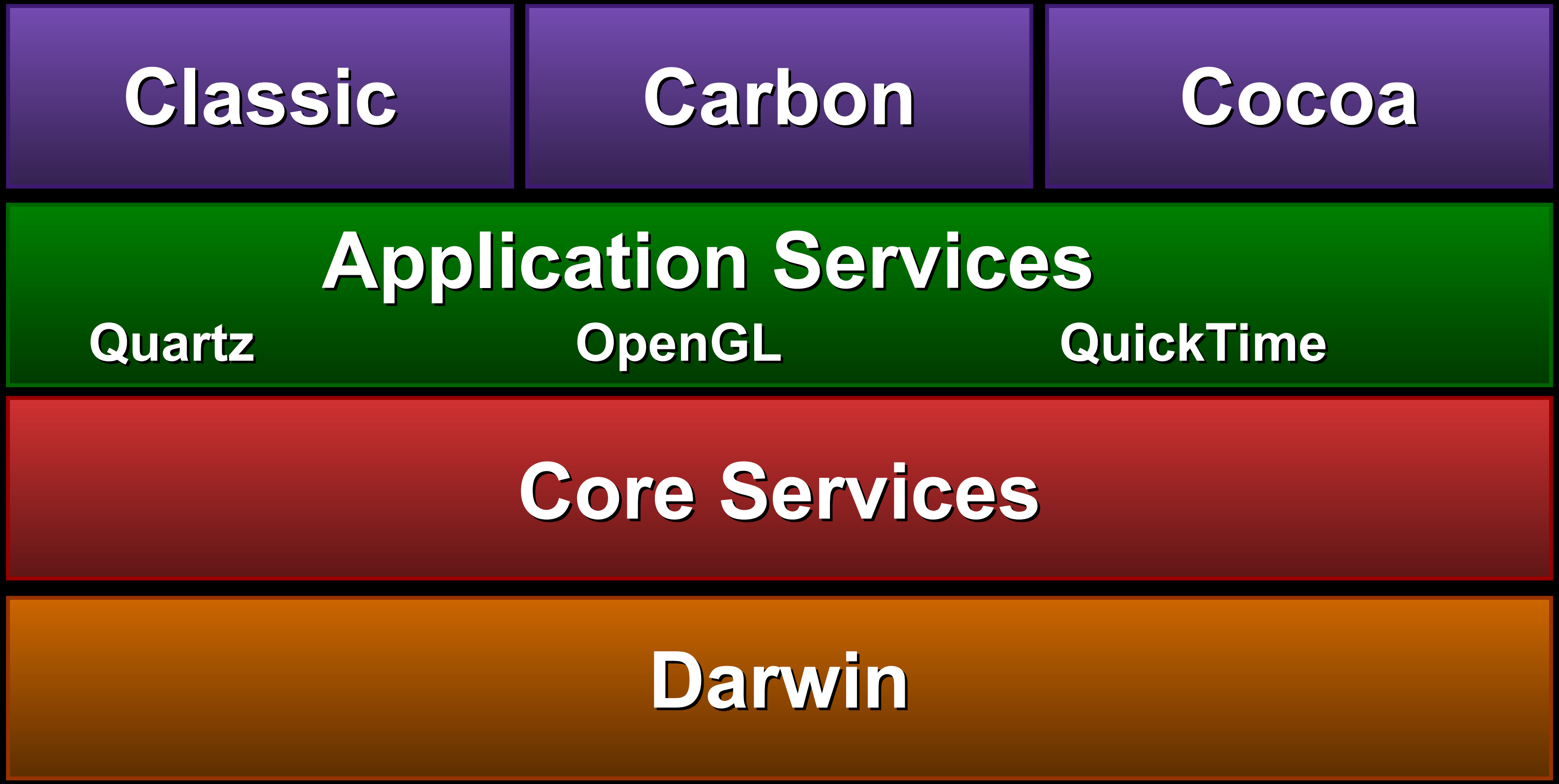
- CoreFoundation (CF) in brief
  - How does it fit in Mac OS X?
  - What does it provide?
- Deep Diving
  - Custom CFAllocators
  - Fast CFString access
  - Using CFRunLoop

CoreFoundation

CoreFoundation

- Non-graphical substrate library for Carbon, Cocoa, and Classic
- Provides common data types and services
- C API and implementation
- Available on Mac OS X
    - Subset in Mac OS 9 via CarbonLib
    - Subset in Darwin

Mac OS X layering



Classic

Carbon

Cocoa

Application Services

Quartz

OpenGL

QuickTime

Core Services

Darwin

Within CoreServices

Other stuff

Open Transport

Low-level Carbon

CoreFoundation

Other stuff

What does CF provide?

- Basic data type abstractions
  - String, array, dictionary...
  - Property lists
- Non-graphical app services needed by all stacks
  - Localization support
  - User preferences

Using CoreFoundation

- You don’t *have* to use CF
  - Carbon & Cocoa are complete without it
- In some cases you will want to use it, for the new functionality
  - Some new Carbon functions take CF type arguments
  - Bundles, Plug-ins
  - Preferences

General philosophy

- High performance
- Minimal, powerful C API
- No safety net
    - Debug library to catch common errors

Object-orientation

- Each type is opaque and acts as a pseudo-class

  CF<class>Ref
- Related functions act as methods

  CF<class><action>
- Related constants

  kCF<class><description>
- Example: CFStrings

  CFStringRef
  CFStringAppend()
  kCFStringEncodingASCII

Polymorphic functions

- A small number of functions can be used with any CFType
- Equality/hashing

  CFEqual(), CFHash()
- Introspection

  CFGetTypeID(), CFCopyDescription()
- Memory management

  CFRetain(), CFRelease(), CFGetRetainCount(), CFGetAllocator()

Memory management

- All CFTypes are reference counted
- CFRetain() to take a reference; CFRelease() to release it
- If a function returns a CFType, who has the reference?

Memory management

- Functions with Get do *not* give the caller a reference
    - Retain if you wish to keep the object
- Functions with Copy do
    - Release when you are finished with it
- Functions with Create return new instances
    - Release when you are finished with it

Memory management

- Copy functions might not perform any memory copies
- Create functions might not perform any allocation
- CF is just being more efficient; this shouldn’t matter to the caller

CoreFoundation types

- Collections
- Strings
- Wrapper types
- Property lists
- Application services

Collections

- Containers for pointer-sized values
- Configured via sets of callback functions
    - Specified when the collection is created
    - Determine how values are compared, added, removed, etc.

Collections

- CFArray (ordered list of values)
- CFDictionary (key-value pairs)
- Others
    - CFSet, CFBag, CFBitVector, CFTree

Mutability

- Three kinds of mutability
    - Immutable: Contents fixed, size fixed
    - Fixed-size: Contents changeable, maximum size is fixed
    - Mutable: Contents changeable, size is dynamic

Mutability

```
CFArrayCreate(kCFAllocatorDefault, someStrings,
    numStrings, &kCFTypeArrayCallBacks)
```

- An immutable array

```
CFDictionaryCreateMutable(kCFAllocatorDefault, 10,
    &kCFTypeDictionaryKeyCallBacks,
    &kCFTypeDictionaryValueCallBacks)
```

- A mutable dictionary that can never exceed 10 key-value pairs

```
CFStringCreateMutable(kCFAllocatorDefault, 0)
```

- A mutable string of unlimited length

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

CFString

- Rich functionality
  - Many creation functions
  - Encoding conversion
  - Comparison, find
  - Explode, combine
  - Format, parse
- Storage optimizations
  - Does not necessarily store Unicode

Other types

- Wrapper types
  - CFData (chunk of bytes)
  - CFNumber (numbers)
  - CFDate (dates)
- CFURL

Property lists

- Any tree built entirely from:
  - CFStrings, CFDatas, CFArrays, CFDictionarys, CFDates, CFNumbers, and CFBooleans
  - Dictionary keys must be strings
- Have a flattened XML representation

Application services

- CFBundle
- CFPlugIn
- CFXMLParser
- CFPreferences
- CFRunLoop and related
- Pasteboard
    - Private service to Carbon & Cocoa

Custom CFAllocators

Allocators

- Allocators determine how memory is allocated and freed
- Create functions take CFAllocators as first argument
    - Normally, pass kCFAllocatorDefault to use the current default allocator

Custom allocators

- Custom allocators are used to define custom allocation behaviors
- However, overuse will tend to cause an app to use more memory and swap more

Custom allocators

- Define a CFAllocatorContext
    - Pointer to user-defined data, usually the allocator’s management info
    - retain, release, and copyDescription callbacks for the user-defined info
    - Define allocate, reallocate, deallocate, and preferredSize callbacks with your own functions
- myAlloc = CFAllocatorCreate(

  allocator, &context);

Custom allocators

- Example: all callbacks NULL would be an allocator which doesn’t allocate or deallocate any memory
    - kCFAllocatorNull
- Example: an allocator which allocated from shared memory
    - But be careful: shared regions must be at same address

Fast CFString Access

Basic CFString API

- CFStringGetLength
  - Returns number of Unicode characters in string
- CFStringGetCharacterAtIndex
  - Returns Unicode character at given (zero-based) index

Coding sample

```
len = CFStringGetLength(str);
for (i = 0; i < len; i++) {
   UniChar c = CFStringGetCharacterAtIndex(str, i);
    … do something with c ...
}
```

Optimization #1: good

- Batch access to characters with CFStringGetCharacters()

```
len = CFStringGetLength(str);
UniChar *buffer = malloc(sizeof(UniChar) * len);
CFStringGetCharacters(str, CFRangeMake(0, len), buffer);
for (i = 0; i < len; i++) {
        … do something with buffer[i] ...
}
```

- Alteratively, use a stack buffer, and process a subrange of the string at a time

Optimization #2: better

- Batch access using direct pointer with CFStringGetCharactersPtr()

```
UniChar *buffer = CFStringGetCharactersPtr(str);
if (NULL != buffer) {
  len = CFStringGetLength(str);
  for (i = 0; i < len; i++) {
      … do something with buffer[i] ...
  }
} else { … optimization #1 ? … }
```

Optimization #3: best

- Inline buffer functions

```
CFStringInlineBuffer buf;
len = CFStringGetLength(str);
CFStringInitInlineBuffer(str, &buf, CFRangeMake(0, len));
for (i = 0; i < len; i++) {
    UniChar c = CFStringGetCharacterFromInlineBuffer(&buf, i);
      … do something with c ...
}
```

- Combines #1 & #2

Another possibility

- Developer-provided external backing store

```
CFStringCreateMutableWithExternalCharactersNoCopy(
  allocator, buffer, bufLen, bufCapacity, bufAllocator)
CFStringSetExternalCharactersNoCopy(
  str, buffer, bufLen, bufCapacity)
```

  - bufAllocator is custom allocator used when CFString needs to grow buffer
  - You access external buffer directly
  - Only useful if you’re wrapping a UniChar buffer with a CFString

Other CFString access

- CFStringGetCStringPtr
- CFStringGetPascalStringPtr
  - Return NULL if pointer can not be immediately returned
  - These do not allocate memory
  - Characters are encoded in the system encoding
- CFStringGetBytes
  - Get contents of string in any encoding

Using CFRunLoop

CFRunLoop

- CFRunLoop is the lowest event loop for Mac OS X
    - An event demultiplexor or dispatcher
- Listens on many types of input sources, and performs callouts when they are ready/signaled
- Normally, use Carbon or Cocoa event systems

CFRunLoop types

- CFRunLoop
    - Manages sets of input sources
- CFRunLoopSource
    - Abstract representation of input
- CFRunLoopTimer
    - Periodic events
- CFRunLoopObserver
    - Events for event loop cycle

CFRunLoop

- Manages sets of input sources called modes
- Run loop must be run to have it monitor the input sources
- Calls the input source’s callout when source becomes ready

CFRunLoop

- One run loop per thread
- Can be used reentrantly (i.e. from within a run loop callout)
- Causes the app to sleep when no input is available
- Most work of an event-driven app happens during a run loop callout

CFRunLoopSources

- Specify several callbacks at creation time to customize a source
- Sources usually implemented as “classes”, and you use that API
- The “class” takes care of satisfying the CFRunLoopSource API

CFRunLoopSources

- When a source becomes signaled the run loop will call its perform() callback
- The “class” does the actual class-specific monitoring
- Calls CFRunLoopSourceSignal(src) when input is ready

CFRunLoopSources

- During the perform() callback, a source may re-signal itself if there’s yet more input to be processed
- How a “class” monitors input is up to it, but a separate thread is common (better than polling)

CFRunLoopTimers

- Specialized form of run loop source to generate periodic callouts
- Create with start date and interval
- Only fired while run loop is running
- Missed fire dates are coalesced

CFRunLoopObservers

- Allows for callouts to be performed at various points in the run loop cycle
  - When entered, exited
  - Before/after sleeping
- Sometimes useful to do something, for example, before the run loop goes to sleep

CFRunLoop sources

- Only one specialized input source available so far
    - CFSocket
- Another soon
    - CFMessagePort
- Some higher-level subsystem may create their own (IOKit?)

Getting More Info

- Documentation on-line
    - http://developer.apple.com/techpubs/corefoundation
    - /System/Developer/Documentation/CoreFoundation
- Example code
    - /System/Developer/Examples/CoreFoundation

Topics in CoreFoundation

Q&A

Think different.