TriDroid West - Thu 16 JULY 2015 - ATC, Durham, NC

C In Android Apps – Why and How

Eric Reifsnider
Mobile Tech Lead
Validic, Inc.

Why C on Android

- cross-platform compatibility onto iOS
  - C code will run “as is” on iOS if you avoid Android-specific constructs (or manage them with #define)
  - industry examples: card.io handles its computer vision in C and shares that C code between Android and iOS … Dropbox’s Carousel and Mailbox apps use C++ to share between Android and iOS.
- speed
  - C code if written well can be more efficient than equivalent Java code … though (IMHO) harder to maintain
  - JVM is written in C … this means the Java bytecode is a program that is being interpreted by another program written in C
  - Dalvik JVM more than 10 times slower than C, ART JVM more than 5 times slower
    - http://www.learnopengles.com/a-performance-comparison-between-java-and-c-on-the-nexus-5/
  - ART is still not as fast as C because it still does all the extra work that Java does - it just doesn’t have to compile from bytecode to native code at runtime … ART native code is still slower than C code for the same task (for some tasks and for an efficient C implementation)
- existing libraries esp. for statistics and other numerical solutions may be more readily available in C … e.g. OpenCV

Why NOT use C on Android

- Many of the best speed optimizations are also harder to maintain
  - e.g. using pointers instead of array indexing
- 2 languages one project

Build configuration

- inline functions in C to improve speed
  - general rules for gcc inlining here: https://gcc.gnu.org/onlinedocs/gcc-5.1.0/gnat_ugn/Inlining-of-Subprograms.html)
  - default options for NDK are -finline-functions, -finline-limit=300, and -fno-inline-functions-called-once
  - that means we do get inlining, but not for functions in local scope called only once, and only for functions < 300 lines
  - in general, NDK options are in the setup.mk files under the NDK's toolchains folder, e.g. android-ndk-r10d/toolchains/arm-linux-androideabi-4.9/setup.mk
  - there is no -finline-functions-called-once, so you can't override the called-once setting with ndk cflags in your gradle file - you have to modify the setup.mk in all architecture folders of the NDK install itself to get inlining of functions called once (similar situation with some other C flags)

Build configuration (contd.)

- optimization level
  - -O2 (found in CFLAGS of setup.mk just like the inlining options) … but only for release mode! debug mode is -O0, so no optimizations and no inlining in debug mode … to get release mode you have to export APK and choose release build, or build from command line and specify release… with -O2 and -finline-functions, even functions not marked inline will be candidates for inlining (as you can see using gobjdump)

What is JNI?

- JNI is the mechanism by which we call into C from Java, in Android and elsewhere
  - JNI calls into C provide a JNIEnv reference which allows C to interact with the JVM e.g. to allocate Java object instances
  - JNI also provides C with a reference to the Java object that made the call into C
  - JNI allows you to use the JNIEnv reference and a reflection-like mechanism to call methods of classes … it's pretty clunky but it works
- This talk is <u>not</u> a JNI tutorial - the material out there on JNI is actually OK

JNI source structure for iOS

- The JNI infrastructure is not used in iOS - iOS can call C directly
- So, you need to isolate the code containing JNI, so that the iOS app can simply ignore those files
  - Java native wrapper file > JNI header file > C header file > C source
  - iOS Objective-C > C header file > C source
  - iOS Swift > bridging header > C header file > C source
- Dropbox's Djinni can generate JNI and Objective-C++ interface layers from a descriptor file … really useful if you have a ton of different functions at the interface … but … avoid that if you can!

C++ vs C

- In iOS, C code is “bridgeless” into ObjectiveC — that is, no interface code is required
- in iOS, C++ code requires Objective-C++ bridge code - serves the role of JNI, arguably easier than JNI - but it’s different from both C ++ and Objective C, so there’s a third language involved
- You have to be more careful in C++ to harvest performance gains
- JNI can’t call C++ class member functions directly - can only call static function members … so essentially the C++ code has to provide a C front end for JNI via statics
- Conclusion: if you’re implementing business logic, consider going with C++, otherwise for performance and simplicity C might be your best bet

Android Studio and native samples

- under the “samples” folder of your NDK install
- In Android Studio, not available within the File > Import Sample menu command
- Instead, use File > New > Import Project to open one of the projects, selecting its folder under the NDK samples directory

Importing a native sample

- unadulterated import of hello-jni into Android Studio will fail to build...
  - Error:Execution failed for task ':app:compileDebugNdk'.
  - > NDK not configured.
  - Download the NDK from http://developer.android.com/tools/sdk/ndk/. Then add ndk.dir=path/to/ndk in local.properties.
- Easy fix: do exactly what it says
  - My local.properties ends up with two lines instead of the default one line - the first line is the default Android non-native SDK, and then you manually add a line for the NDK
  - sdk.dir=/Users/ericr/Library/Android/sdk
  - ndk.dir=/Users/ericr/Documents/android-ndk-r10d
- With this single fix, the hello-jni sample from the NDK r10d version, imported into Android Studio 1.2, works fine.

*.mk no longer needed

- For those who have tinkered with this before - NDK integration (even in Android Studio 1.2) means that Android.mk and Application.mk are not needed
  - These files specify among other things C/C++ source files and module name
- If you delete them from the imported native hello-jni sample app, it has no effect
- You can use the *.mk files, by disabling this feature, if you want features in the *.mk system - but I'd suggest checking out the NDK integration in Android 1.3 before going down this road

Gradle's "ndk" section

- You will need an "ndk" section under your build.gradle's android > defaultConfig section
- By default the import conversion of a sample project will create such a section, listing your NDK "module", but without ldLibs for logging etc.
- Add ldLibs for log (Android logging), jnigraphics (Android bitmaps), and other linkages
  - OpenGL: GLESv1_CM or GLESv2
  - EGL: EGL
  - Android native app API: android
- Can specify cflags, stl choice - the example code shows cflags usage

Native library call reference

- For some reason I keep coming back to this (perhaps dated) site - so here it is:

- http://mobilepearls.com/labs/native-android-api/

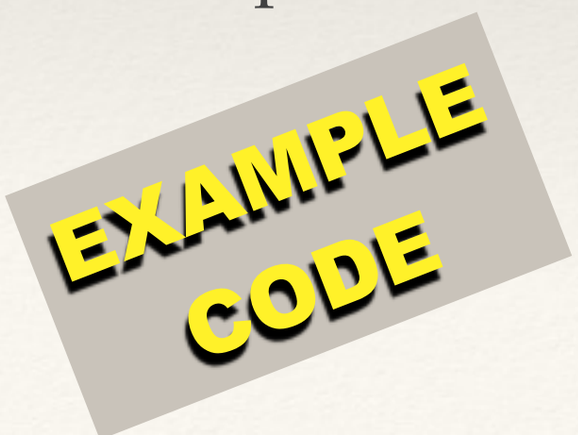
- Lists both the libraries (and any gradle ndk ldLibs you need) as well as the #includes, side by side - very convenient

Code that needs to run differently on iOS

- Use the compiler definitions to #ifdef code that is different on iOS
  - the example code handles logging in this way
- gradle cflags and clang cflags can be used to finesse any tricky situations
- it's much easier to simply avoid platform-specific code where possible - or put the platform-specific code in its own file and only link in the file used by your platform
  - the example code handles the JNI frippery this way

Logging

- include android/log.h, and put ldLibs “log” into your ndk section in gradle
- Similar to Log.d(LOG_TAG, “whatever”) we use in Java, but with C-style printf formatting
  - __android_log_print(ANDROID_LOG_DEBUG, “hello-jni”,
- You may find it convenient to define a macro in order to clean up the ugly naming convention for the logging call - the macro can be different for iOS
  - #define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG,LOG_TAG,__VA_ARGS__)
- Output is conventional logcat, mixed in with the Java logcat output



Store & retrieve data from C

- You can store data in one Java call to a persistent C variable e.g. a global, and retrieve it later from another Java call

Local vs. global JNI references

- Android JVM moves stuff around as part of garbage collection - the actual storage of objects can be moved as a consequence of garbage collection of stale objects
  - Not all JVMs do this - pre-Honeycomb, Android's JVM didn't do this
- C can't handle this movement, so JNI wraps JVM objects in a handle e.g. jint, jstring - and jobject itself
- CheckJNI should find such problems and it's on by default
- If you must store these objects, generate a global reference and store that … but don't forget to delete that global reference when you are done … the global ref leak detection threshold for CheckJNI is 2000 instances…
- For portable code, much better not to do this - use your JNI layer to convert to a C object that will work on iOS and store that C object instead

Initializing your C library

- There is a method for initializing your C library, when it's loaded, that is portable across Android and iOS
- __attribute__((constructor)), and also ((destructor), can be used to mark C functions that will consequently be run when the C library is loaded/unloaded
- This is the sort of stuff you'd run in the static block of a Java class - saves you from having to use an initializer function that the Java code would have to remember to call

Android Studio 1.3 and C/C++

- C/C++ support greatly improved in Android Studio 1.3
  - NDK is bundled, not distributed separately (I had to install it separately)
  - Debugging in native code is supported (new run configuration for "Android Native" available, and a new "Native Debugger" tab alongside the Logcat tab)
  - Other features we're used to for our Java code - Code completion, navigation e.g. go to declaration, jump between header and implementation, quick fixes, refactoring, formatting
- I have not yet switched: I am in the middle of a project at work, and the beta is too raw - even the Gradle syntax is still changing substantially
  - I have been muddling through with CppTools plugin - does not correctly handle syntax for long files, and therefore does not do proper code completion, quick fixes, etc. - sometimes I think I should just turn it off
  - Debugging is via logcat...
- handy URLs
  - Android Studio new NDK features: http://tools.android.com/tech-docs/android-ndk-preview
  - CppTools plugin for older Android Studio: https://plugins.jetbrains.com/plugin/1373?pr=

NativeActivity

- Making apps with NativeActivity is beyond today's scope
- Good sample app showing NativeActivity, but also showing OpenGL interaction: the Teapot sample: https://developer.android.com/ndk/samples/sample_teapot.html

Memory consistency issues

- If there are status variables in C that you are checking from multiple threads in Java, be sure to declare them as volatile (e.g. a volatile int in C is both atomic and proof against memory consistency problems) … if you're paranoid use sig_atomic_t instead of int
- Coordination between threads, that has to cross the JNI boundary, can be confusing
  - Better for simplicity's sake to manage concurrency that involves the app, at the app level, to leverage the strength of platform-specific concurrency mechanisms that interact well with UI driven apps
  - synchronized, locks, etc. in Android - also Loader, Handler, Service
- Avoid temptation to use JNI's MonitorEnter() as a substitute for Java synchronized - it's not portable to non-Java situations
- If concurrency is entirely within the C code, use C-specific threading and sync
  - pthread_start() and pthread_join() work fine in Android and iOS

More information

- Dropbox's project:
  - https://github.com/libmx3/mx3
- NDK references by Google
  - https://developer.android.com/ndk/reference/index.html
  - https://developer.android.com/ndk/index.html
- JNI with Android reference page from Google (important!!!):
  - http://developer.android.com/training/articles/perf-jni.html
- NDK features in new Android Studio 1.3
  - http://tools.android.com/tech-docs/android-ndk-preview