Compiling and Using a C++ Library on Android with Android Studio

Published: 2013-08-07

This post falls into the category of "write it down before I forget it". I know next to nothing about Android/Java development (approx 12 hours worth) but I knew I needed a certain C++ library for an upcoming app. I managed to get the C++ library working from java after 20+ attempts, 4 coffees and the better part of an evening.

References

Most of the code here is cobbled together from these sources:

- Android Native Development Kit (NDK), and included documentation.
- Running Native Code on Android Presentation by Cédric Deltheil
- StackOverflow: How to build c-ares library in Android (NDK)

Overview

These are the steps:

1. Compile your library for Android
2. Write the C/C++ wrapper for your library
3. Configure gradle to package up your library
4. Test from java

1. Compile your library for Android

First, grab the Android Native Development Kit (NDK). This includes a toolchain for cross-compiling C/C++ to Android. Extract the NDK somewhere sane, and add the tools to your path.

```
$ PATH="<your_android_ndk_root_folder>:${PATH}"
$ export PATH
```

The key documentation file to read is called `STANDALONE-TOOLCHAIN.HTML` as we will be using a standalone toolchain to build the third party library. Install the standard toolchain. The commands below will install it to `/tmp/my-android-toolchain` .

```
$ /path/to/ndk/build/tools/make-standalone-toolchain.sh
  --platform=android-8 \
  --install-dir=/tmp/my-android-toolchain
$ cd /tmp/my-android-toolchain
```

Set some environment variables so that the configuration and build process will use the right compiler.

```
$ export PATH=/tmp/my-android-toolchain/bin:$PATH
$ export CC="arm-linux-androideabi-gcc"
$ export CXX="arm-linux-androideabi-g++"
```

Extract your library tarball and start the configuration and building process. It is important to tell your configure script which toolchain to use, as well as specifying a folder (prefix) for the output. Since we are building a static library we will also instruct it to build one.

```
$ cd yourLibrary
$ mkdir build
$ ./configure --prefix=$(pwd)/build --host=arm-linux-ar
$ make
$ make install
```

You should now have a `yourLibrary.a` file in `build/lib` and a whole pile of headers in `build/include`. Create a folder called `prebuild` in your Android project root folder. (The root folder is one level down from the `YourAppNameProject` folder and is usually named after your app) Copy the `yourLibrary.a` file to the `prebuild` folder and also copy the `include` folder.

```
$ mkdir ~/AndroidStudioProjects/YourAppNameProject/AppN
$ cp build/lib/yourLibrary.a ~/AndroidStudioProjects/Yo
$ cp -r build/include ~/AndroidStudioProjects/YourAppNa
```

2. Write the C/C++ wrapper for your library

This will depend on which library you are wrapping. Modify one of the following to carry out some simple task using the library you are wrapping. These are derived from the `hello-jni` sample app in the NDK - check there for more info on how they work. Your wrapper files and the `.mk` files should be placed in the `project_root/jni` folder.

```
/* C Version */

#include <string.h>
#include <jni.h>
#include <YourLibrary/YourLibrary.h>

/*
 * replace com_example_whatever with your package name
 *
 * HelloJni should be the name of the activity that wi
 * call this function
 *
 * change the returned string to be one that exercises
 * some functionality in your wrapped library to test
 * it all works
 *
 */

jstring
Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEr
                                                 jobje
{
    return (*env)->NewStringUTF(env, "Hello from JNI !
}
```

```
/* C++ Version */

#include <string.h>
#include <jni.h>
#include <YourLibrary/YourLibrary.h>

/*
 * replace com_example_whatever with your package name
 *
 * HelloJni should be the name of the activity that wi
 * call this function
 *
 * change the returned string to be one that exercises
 * some functionality in your wrapped library to test
 * it all works
 *
 */

extern "C" {
    JNIEXPORT jstring JNICALL
    Java_com_example_hellojni_HelloJni_stringFromJNI(J
                                                    j
    {
        return env->NewStringUTF("Hello from C++ JNI !
    }
}
```

Next, set up the `Android.mk` file for your wrapper. This is like a `makefile` for the ndk-build command that will build your wrapper.

```
LOCAL_PATH := $(call my-dir)

# static library info
LOCAL_MODULE := libYourLibrary
LOCAL_SRC_FILES := ../prebuild/libYourLibrary.a
LOCAL_EXPORT_C_INCLUDES := ../prebuild/include
include $(PREBUILT_STATIC_LIBRARY)

# wrapper info
include $(CLEAR_VARS)
LOCAL_C_INCLUDES += ../prebuild/include
LOCAL_MODULE    := your-wrapper
LOCAL_SRC_FILES := your-wrapper.cpp
LOCAL_STATIC_LIBRARIES := libYourLibrary
include $(BUILD_SHARED_LIBRARY)
```

I also needed the following in my `Application.mk` file:

```
APP_STL := gnustl_static
APP_PLATFORM := android-8
```

At this point, you should be able to build your library from the `jni` folder.

```
$ ndk-build

Gdbserver      : [arm-linux-androideabi-4.6] libs/armea
Gdbsetup       : libs/armeabi/gdb.setup
Install        : your-wrapper.so => libs/armeabi/your-w
```

You can check the `project_root/libs/armeabi` folder for your new library.

3. Configure gradle to package up your library

Android Studio doesn't currently support NDK development so some gradle hacks are required. In a nutshell, the modifications copy and package up the .so file so that it is copied and installed with your app. Check the references for more detail. In build.gradle add the following:

```
task nativeLibsToJar(type: Zip, description: 'create a
    destinationDir file("$buildDir/native-libs")
    baseName 'native-libs'
    extension 'jar'
    from fileTree(dir: 'libs', include: '**/*.so')
    into 'lib/'
}

tasks.withType(Compile) {
    compileTask -> compileTask.dependsOn(nativeLibsToJ
}
```

(Update August 2015 - I've been informed that `tasks.withType(Compile)` should now be `tasks.withType(JavaCompile)` .)

Also add the following to the `dependencies{...} section` :

```
compile fileTree(dir: "$buildDir/native-libs", include:
```

4. Test from java

In the activity you are calling your wrapper from, add the following, modifying names as appropriate:

```java
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "If this doesn't crash you are a genius
    Log.d(TAG, testWrapper());

// the java declaration for your wrapper test functior
public native String testWrapper();

// tell java which library to load
static {
    System.loadLibrary("your-wrapper");
}
```

If it doesn't crash, you have probably done it. Time to celebrate!