Android Developers

Add C and C++ Code to Your Project

2 Using Android Studio 2.2 or higher (https://developer.android.com/studio/index.html) with the Android plugin for Gradle version 2.2.0 or higher (https://developer.android.com/studio/releases/gradle-plugin.html), you can add C and C++ code to your app by compiling it into a native library that Gradle can package with your APK. Your Java code can then call functions in your native library through the Java Native Interface (JNI). If you want to learn more about using the JNI framework, read JNI tips for Android (https://developer.android.com/training/articles/perf-jni.html).

In this document

Android Studio's default build tool for native libraries is CMake. Android Studio also supports ndk-build (https://developer.android.com/ndk/guides/ndk-build.html) due to the large number of existing projects that use the build toolkit to compile their native code. If you want to import an existing ndk-build library into your Android Studio project, see the section about how to configure Gradle to link to your native library (#link-gradle). However, if you are creating a new native library, you should use CMake.

This page gives you the information you need to set up Android Studio with the necessary build tools, create or configure a project to support native code on Android, and build and run your app.

> **Note:** If your existing project uses the deprecated `ndkCompile` tool, you should migrate to using either CMake or ndk-build. To learn more, go to the section about how to Migrate from ndkCompile (#ndkCompile).

> **Attention experimental Gradle users:** Consider migrating to plugin version 2.2.0 or higher (http://tools.android.com/tech-docs/new-build-system/gradle-experimental/migrate-to-stable), and using CMake or ndk-build to build your native libraries if any of the following apply to you: Your native project already uses CMake or ndk-build; you would rather use a stable version of the Gradle build system; or you want support for add-on tools, such as CCache (https://ccache.samba.org/) . Otherwise, you can continue to use the experimental version of Gradle and the Android plugin (http://tools.android.com/tech-docs/new-build-system/gradle-experimental).

Download the NDK and Build Tools

To compile and debug native code for your app, you need the following components:

- *The Android Native Development Kit (NDK)* *(https://developer.android.com/ndk/index.html)*: a toolset that allows you to use C and C++ code with Android, and provides platform libraries that allow you to manage native activities and access physical device components, such as sensors and touch input.
- *CMake* *(https://cmake.org/)* : an external build tool that works alongside Gradle to build your native library. You do not need this component if you only plan to use ndk-build.
- *LLDB* *(http://lldb.llvm.org/)* : the debugger Android Studio uses to debug native code (https://developer.android.com/studio/debug/index.html).

You can install these components using the SDK Manager (https://developer.android.com/studio/intro/update.html#sdk-manager):

1. From an open project, select **Tools > Android > SDK Manager** from the menu bar.
2. Click the **SDK Tools** tab.
3. Check the boxes next to **LLDB**, **CMake**, and **NDK**, as shown in figure 1.

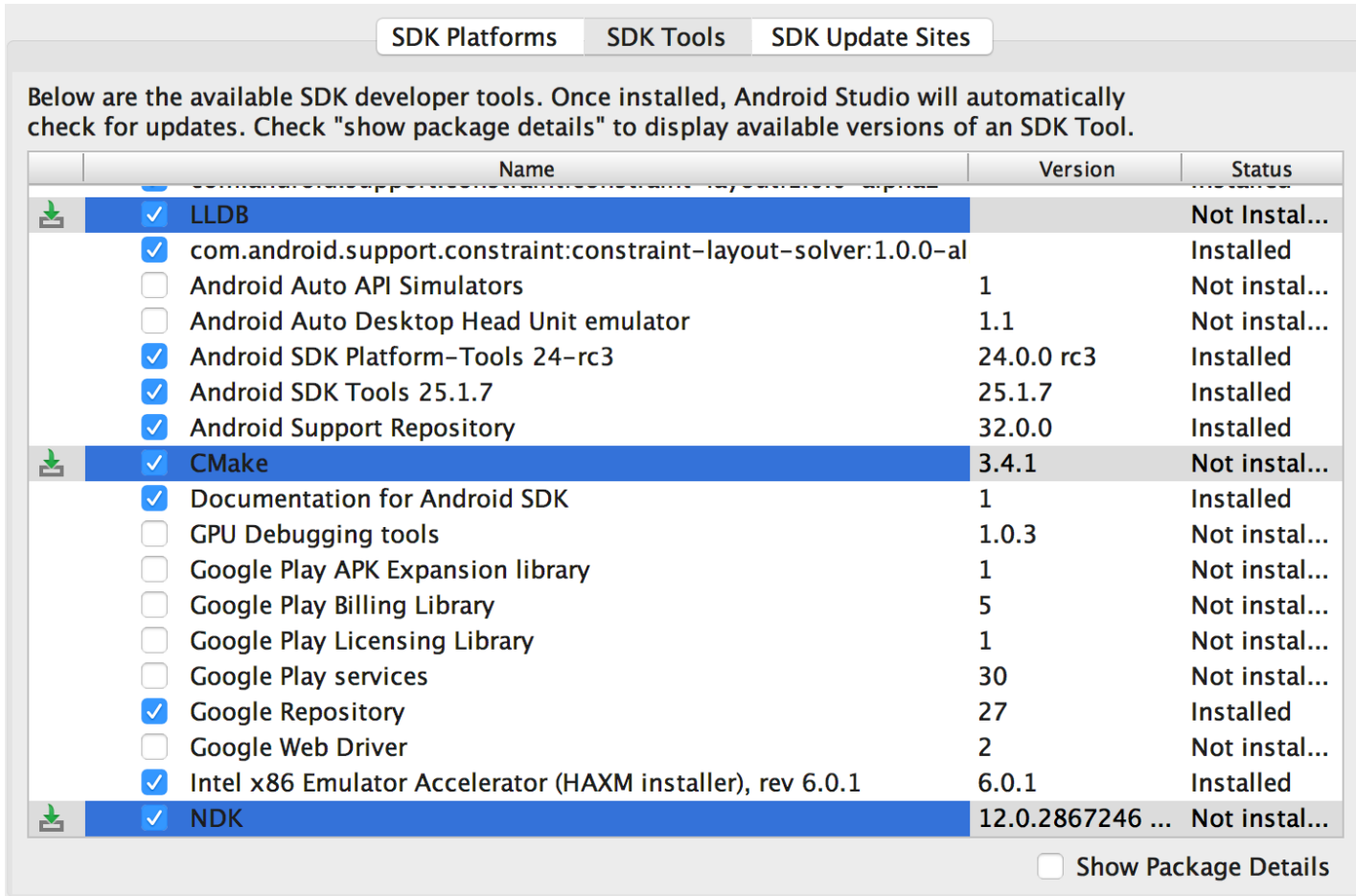

Figure 1. Installing LLDB, CMake, and the NDK from the SDK Manager.

4. Click **Apply**, and then click **OK** in the pop-up dialog.
5. When the installation is complete, click **Finish**, and then click **OK**.

Create a New Project with C/C++ Support

Creating a new project with support for native code is similar to creating any other Android Studio project (https://developer.android.com/studio/projects/create-project.html), but there are a few additional steps:

1. In the **Configure your new project** section of the wizard, check the **Include C++ Support** checkbox.
2. Click **Next**.
3. Complete all other fields and the next few sections of the wizard as normal.
4. In the **Customize C++ Support** section of the wizard, you can customize your project with the following options:
   - **C++ Standard**: use the drop-down list to select which standardization of C++ you want to use. Selecting **Toolchain Default** uses the default CMake setting.
   - **Exceptions Support**: check this box if you want to enable support for C++ exception handling. If enabled, Android Studio adds the `-fexceptions` flag to `cppFlags` in your module-level `build.gradle` file, which Gradle passes to CMake.
   - **Runtime Type Information Support**: check this box if you want support for RTTI. If enabled, Android Studio adds the `-frtti` flag to `cppFlags` in your module-level `build.gradle` file, which Gradle passes to CMake.
5. Click **Finish**.

After Android Studio finishes creating your new project, open the **Project** pane from the left side of the IDE and select the **Android** view. As shown in figure 2, Android Studio adds the **cpp** and **External Build Files** groups:

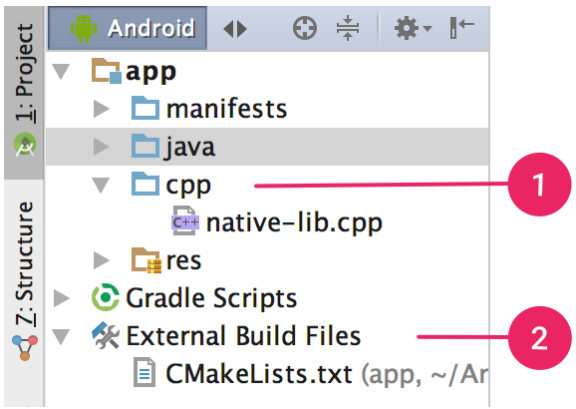

Figure 2. Android view groups for your native sources and external build scripts.

> **Note:** This view does not reflect the actual file hierarchy on disk, but groups similar files to simplify navigating your project.

The **cpp** group is where you can find all the native source files, headers, and prebuilt libraries that are a part of your project. For new projects, Android Studio creates a sample C++ source file, `native-lib.cpp`, and places it in the `src/main/cpp/` directory of your app module. This sample code provides a simple C++ function, `stringFromJNI()`, that returns the string "Hello from C++". You can learn how to add additional source files to your project in the section about how to Create new native source files (#create-sources).

> **Known Issue:** Android Studio currently shows you only the header files that have matching source file—even if you specify other headers in your CMake build script (#create-cmake-script). See Issue #38068472 (https://issuetracker.google.com/issues/38068472)

The **External Build Files** group is where you can find build scripts for CMake or ndk-build. Similar to how `build.gradle` files tell Gradle how to build your app, CMake and ndk-build require a build script to know how to build your native library. For new projects, Android Studio creates a CMake build script, `CMakeLists.txt`, and places it in your module's root directory. You can learn more about the contents of this build script in the section about how to Create a Cmake Build Script (#create-cmake-script).

Build and run the sample app

When you click **Run** ▶, Android Studio builds and launches an app that displays the text "Hello from C++" on your Android device or emulator. The following overview describes the events that occur in order to build and run the sample app:

1. Gradle calls upon your external build script, `CMakeLists.txt`.
2. CMake follows commands in the build script to compile a C++ source file, `native-lib.cpp`, into a shared object library and names it `libnative-lib.so`, which Gradle then packages into the APK.
3. During runtime, the app's `MainActivity` loads the native library using `System.loadLibrary()` (https://developer.android.com/reference/java/lang/System.html#loadLibrary(java.lang.String)). The library's native function, `stringFromJNI()`, is now availableto the app.
4. `MainActivity.onCreate()` calls `stringFromJNI()`, which returns "Hello from C++", and uses it to update the `TextView` (https://developer.android.com/reference/android/widget/TextView.html).

> **Note:** Instant Run (https://developer.android.com/studio/run/index.html#instant-run) is not compatible with components of your project written in native code.

If you want to verify that Gradle packages the native library in the APK, you can use the APK Analyzer (https://developer.android.com/studio/build/apk-analyzer.html):

1. Select **Build > Analyze APK**.
2. Select the APK from the `app/build/outputs/apk/` directory and click **OK**.
3. As shown in figure 3, you can see `libnative-lib.so` in the APK Analyzer window under `lib/<ABI>/`.

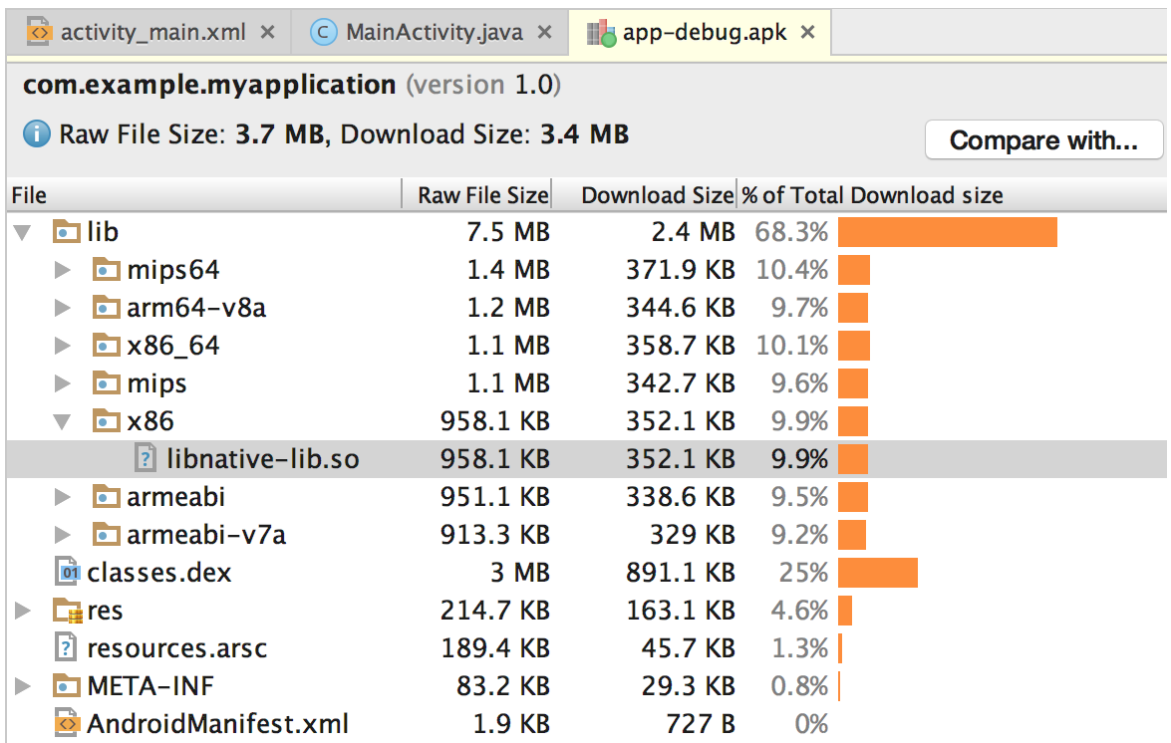

Figure 3. Locating a native library using the APK Analyzer.

> **Tip:** If you want to experiment with other Android apps that use native code, click **File > New > Import Sample** and select a sample project from the **Ndk** list.

Add C/C++ Code to an Existing Project

If you want to add native code to an existing project, perform these steps:

1. Create new native source files (#create-sources) and add them to your Android Studio project.
   - You can skip this step if you already have native code or want to import a prebuilt native library.
2. Create a CMake build script (#create-cmake-script) to build your native source code into a library. You also require this build script if you are importing and linking against prebuilt or platform libraries.
   - If you have an existing native library that already has a `CMakeLists.txt` build script, or uses ndk-build and includes an `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) build script, you can skip this step.
3. Link Gradle to your native library (#link-gradle) by providing a path to your CMake or ndk-build script file. Gradle uses the build script to import source code into your Android Studio project and package your native library (the SO file) into the APK.

Once you configure your project, you can access your native functions from Java code using the JNI framework (http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html) . To build and run your app, simply click **Run** ▶. Gradle adds your external native build process as a dependency to compile, build, and package your native library with your APK.

Create new native source files

To create a `cpp/` directory with new native source files in the main sourceset of your app module, proceed as follows:

1. Open the **Project** pane from the left side of the IDE and select the **Project** view from the drop-down menu.
2. Navigate to ***your-module*** **> src**, right-click on the **main** directory, and select **New > Directory**.
3. Enter a name for the directory (such as `cpp`) and click **OK**.
4. Right-click on the directory you just created and select **New > C/C++ Source File**.
5. Enter a name for your source file, such as `native-lib`.
6. From the **Type** drop-down menu, select the file extension for your source file, such as `.cpp`.
   - You can add other file types to the drop-down menu, such as `.cxx` or `.hxx`, by clicking **Edit File Types** 🔧. In the **C/C++** dialog box that pops up, select another file extension from the **Source Extension** and **Header Extension** drop-down menus and click **OK**.
7. If you also want to create a header file, check the **Create an associated header** checkbox.
8. Click **OK**.

Create a CMake build script

If your native sources don't already have a CMake build script, you need to create one yourself and include the appropriate CMake commands. A CMake build script is a plain text file that you must name `CMakeLists.txt`. This section covers some basic commands you should include in your build script in order to tell CMake which sources to use when creating your native library. To learn more, read the official documentation about CMake commands (https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html) .

> **Note:** If your project uses ndk-build, you don't need to create a CMake build script. You can link Gradle to your native library (#link-gradle) by providing a path to your `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) file.

To create a plain text file that you can use as your CMake build script, proceed as follows:

1. Open the **Project** pane from the left side of the IDE and select the **Project** view from the drop-down menu.
2. Right-click on the root directory of ***your-module*** and select **New > File**.

> **Note:** You can create the build script in any location you want. However, when configuring the build script, paths to your native source files and libraries are relative to the location of the build script.

3. Enter "CMakeLists.txt" as the filename and click **OK**.

You can now configure your build script by adding CMake commands. To instruct CMake to create a native library from native source code, add the `cmake_minimum_required()` (https://cmake.org/cmake/help/latest/command/cmake_minimum_required.html) and `add_library()` (https://cmake.org/cmake/help/latest/command/add_library.html) commands to your build script:

```
# Sets the minimum version of CMake required to build your native library.
# This ensures that a certain set of CMake features is available to
# your build.

cmake_minimum_required(VERSION 3.4.1)

# Specifies a library name, specifies whether the library is STATIC or
# SHARED, and provides relative paths to the source code. You can
# define multiple libraries by adding multiple add.library() commands,
# and CMake builds them for you. When you build your app, Gradle
# automatically packages shared libraries with your APK.

add_library( # Specifies the name of the library.
             native-lib

             # Sets the library as a shared library.
             SHARED

             # Provides a relative path to your source file(s).
             src/main/cpp/native-lib.cpp )
```

When you add a source file or library to your CMake build script using `add_library()`, Android Studio also shows associated header files in the **Project** view after you sync your project. However, in order for CMake to locate your header files during compile time, you need to add the `include_directories()` (https://cmake.org/cmake/help/latest/command/include_directories.html) command to your CMake build script and specify the path to your headers:

```
add_library(...)

# Specifies a path to native header files.
include_directories(src/main/cpp/include/)
```

The convention CMake uses to name the file of your library is as follows:

```
lib library-name .so
```

For example, if you specify "native-lib" as the name of your shared library in the build script, CMake creates a file named `libnative-lib.so`. However, when loading this library in your Java code, use the name you specified in the CMake build script:

```
static {
    System.loadLibrary("native-lib");
}
```

> **Note:** If you rename or remove a library in your CMake build script, you need to clean your project before Gradle applies the changes or removes the older version of the library from your APK. To clean your project, select **Build > Clean Project** from the menu bar.

Android Studio automatically adds the source files and headers to the **cpp** group in the **Project** pane. By using multiple `add_library()` commands, you can define additional libraries for CMake to build from other source files.

Add NDK APIs

The Android NDK provides a set of native APIs and libraries that you may find useful. You can use any of these APIs by including the NDK libraries (https://developer.android.com/ndk/guides/stable_apis.html) in your project's `CMakeLists.txt` script file.

Prebuilt NDK libraries already exist on the Android platform, so you don't need to build them or package them into your APK. Because the NDK libraries are already a part of CMake's search path, you don't even need to specify the location of the library in your local NDK installation—you only need to provide CMake with the name of the library you

want to use and link it against your own native library.

Add the `find_library()` (https://cmake.org/cmake/help/latest/command/find_library.html) command to your CMake build script to locate an NDK library and store its path as a variable. You use this variable to refer to the NDK library in other parts of the build script. The following sample locates the Android-specific log support library (https://developer.android.com/ndk/guides/stable_apis.html#a3) and stores its path in `log-lib`:

```
find_library( # Defines the name of the path variable that stores the
              # location of the NDK library.
              log-lib

              # Specifies the name of the NDK library that
              # CMake needs to locate.
              log )
```

In order for your native library to call functions in the `log` library, you need to link the libraries using the `target_link_libraries()` (https://cmake.org/cmake/help/latest/command/target_link_libraries.html) command in your CMake build script:

```
find_library(...)

# Links your native library against one or more other native libraries.
target_link_libraries( # Specifies the target library.
                       native-lib

                       # Links the log library to the target library.
                       ${log-lib} )
```

The NDK also includes some libraries as source code that you need to build and link to your native library. You can compile the source code into a native library by using the `add_library()` command in your CMake build script. To provide a path to your local NDK library, you can use the `ANDROID_NDK` path variable, which Android Studio automatically defines for you.

The following command tells CMake to build `android_native_app_glue.c`, which manages `NativeActivity` (https://developer.android.com/reference/android/app/NativeActivity.html) lifecycle events and touch input, into a static library and links it to `native-lib`:

```
add_library( app-glue
             STATIC
             ${ANDROID_NDK}/sources/android/native_app_glue/android_native_app_glue.c )

# You need to link static libraries against your shared native library.
target_link_libraries( native-lib app-glue ${log-lib} )
```

Add other prebuilt libraries

Adding a prebuilt library is similar to specifying another native library for CMake to build. However, because the library is already built, you need to use the `IMPORTED` (https://cmake.org/cmake/help/latest/prop_tgt/IMPORTED.html#prop_tgt:IMPORTED) flag to tell CMake that you only want to import the library into your project:

```
add_library( imported-lib
             SHARED
             IMPORTED )
```

You then need to specify the path to the library using the `set_target_properties()` (https://cmake.org/cmake/help/latest/command/set_target_properties.html) command as shown below.

Some libraries provide separate packages for specific CPU architectures, or Application Binary Interfaces (ABI) (https://developer.android.com/ndk/guides/abis.html), and organize them into separate directories. This approach helps libraries take advantage of certain CPU architectures while allowing you to use only the versions of the library you want. To add multiple ABI versions of a library to your CMake build script, without having to write multiple commands for each version of the library, you can use the `ANDROID_ABI` path variable. This variable uses a list of the default ABIs that the NDK supports (https://developer.android.com/ndk/guides/abis.html#sa), or a filtered list of ABIs you manually configure Gradle (#specify-abi) to use. For example:

```
add_library(...)
```

```
set_target_properties( # Specifies the target library.
                       imported-lib

                       # Specifies the parameter you want to define.
                       PROPERTIES IMPORTED_LOCATION

                       # Provides the path to the library you want to import.
                       imported-lib/src/${ANDROID_ABI}/libimported-lib.so )
```

For CMake to locate your header files during compile time, you need to use the `include_directories()` command and include the path to your header files:

```
include_directories( imported-lib/include/ )
```

> **Note:** If you want to package a prebuilt library that is not a build-time dependency—for example, when adding a prebuilt library that is a dependency of `imported-lib`, you do not need perform the following instructions to link the library.

To link the prebuilt library to your own native library, add it to the `target_link_libraries()` command in your CMake build script:

```
target_link_libraries( native-lib imported-lib app-glue ${log-lib} )
```

To package the prebuilt library into your APK, you need to manually configure Gradle (#configure-gradle) with the `sourceSets` block to include the path to your `.so` file. After building your APK, you can verify which libraries Gradle packages into your APK by using the APK Analyzer (https://developer.android.com/studio/build/apk-analyzer.html).

Include other CMake projects

If you want to build multiple CMake projects and include their outputs in your Android project, you can use one `CMakeLists.txt` file as the top-level CMake build script (which is the one you link to Gradle (#link-gradle)) and add additional CMake projects as dependencies of that build script. The following top-level CMake build script uses the `add_subdirectory()` (https://cmake.org/cmake/help/latest/command/add_subdirectory.html) command to specify another `CMakeLists.txt` file as a build dependency and then links against its output just as it would with any other prebuilt library.

```
# Sets lib_src_DIR to the path of the target CMake project.
set( lib_src_DIR ../gmath )

# Sets lib_build_DIR to the path of the desired output directory.
set( lib_build_DIR ../gmath/outputs )
file(MAKE_DIRECTORY ${lib_build_DIR})

# Adds the CMakeLists.txt file located in the specified directory
# as a build dependency.
add_subdirectory( # Specifies the directory of the CMakeLists.txt file.
                  ${lib_src_DIR}

                  # Specifies the directory for the build outputs.
                  ${lib_build_DIR} )

# Adds the output of the additional CMake build as a prebuilt static
# library and names it lib_gmath.
add_library( lib_gmath STATIC IMPORTED )
set_target_properties( lib_gmath PROPERTIES IMPORTED_LOCATION
                       ${lib_build_DIR}/${ANDROID_ABI}/lib_gmath.a )
include_directories( ${lib_src_DIR}/include )

# Links the top-level CMake build output against lib_gmath.
target_link_libraries( native-lib ... lib_gmath )
```

Link Gradle to your native library

To link Gradle to your native library, you need to provide a path to your CMake or ndk-build script file. When you build your app, Gradle runs CMake or ndk-build as a dependency, and packages shared libraries with your APK. Gradle also uses the build script to know which files to pull into your Android Studio project, so you can access them from the **Project** window. If you don't have a build script for your native sources, you need to create a CMake build script (#create-cmake-script) before you proceed.

Each module in your Android project can link to only one CMake or ndk-build script file. So, for example, if you want to build and package outputs from multiple CMake projects, you need to use one `CMakeLists.txt` file as your top-level CMake build script (which you then link Gradle to) and add other CMake projects (#include-other-cmake-projects) as dependencies of that build script. Similarly, if you're using ndk-build, you can include other Makefiles (https://www.gnu.org/software/make/manual/html_node/Include.html) in your top-level `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) script file.

Once you link Gradle to a native project, Android Studio updates the **Project** pane to show your source files and native libraries in the **cpp** group, and your external build scripts in the **External Build Files** group.

> **Note:** When making changes to the Gradle configuration, make sure to apply your changes by clicking **Sync Project** in the toolbar. Additionally, when making changes to your CMake or ndk-build script file after you have already linked it to Gradle, you should sync Android Studio with your changes by selecting **Build > Refresh Linked C++ Projects** from the menu bar.

Use the Android Studio UI

You can link Gradle to an external CMake or ndk-build project using the Android Studio UI:

1. Open the **Project** pane from the left side of the IDE and select the **Android** view.
2. Right-click on the module you would like to link to your native library, such as the **app** module, and select **Link C++ Project with Gradle** from the menu. You should see a dialog similar to the one shown in figure 4.
3. From the drop-down menu, select either **CMake** or **ndk-build**.
   a. If you select **CMake**, use the field next to **Project Path** to specify the `CMakeLists.txt` script file for your external CMake project.
   b. If you select **ndk-build**, use the field next to **Project Path** to specify the `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) script file for your external ndk-build project. Android Studio also includes the `Application.mk` (https://developer.android.com/ndk/guides/application_mk.html) file if it is located in the same directory as your `Android.mk` file.

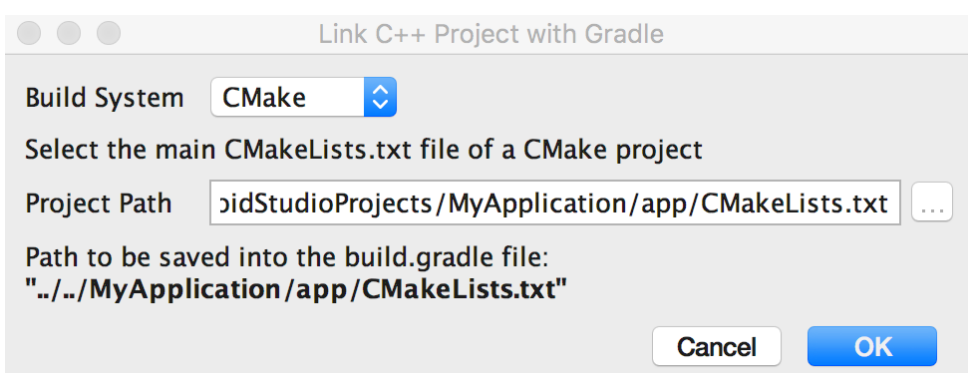

Figure 4. Linking an external C++ project using the Android Studio dialog.

4. Click **OK**.

Manually configure Gradle

To manually configure Gradle to link to your native library, you need to add the `externalNativeBuild` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ExternalNativeBuild.html) block to your module-level `build.gradle` file and configure it with either the `cmake` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.CmakeOptions.html) or `ndkBuild` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.NdkBuildOptions.html) block:

```
android {
  ...
  defaultConfig {...}
  buildTypes {...}

  // Encapsulates your external native build configurations.
  externalNativeBuild {

    // Encapsulates your CMake build configurations.
    cmake {

      // Provides a relative path to your CMake build script.
      path "CMakeLists.txt"
    }
```

```
  }

  // If you want Gradle to package prebuilt native libraries
  // with your APK, modify the default source set configuration (https://developer.android.com/stu
  // to include the directory of your prebuilt .so files as follows.
  sourceSets {
      main {
          jniLibs.srcDirs 'imported-lib/src/', 'more-imported-libs/src/'
      }
  }
}
```

> **Note:** If you want to link Gradle to an existing ndk-build project, use the `ndkBuild` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.NdkBuildOptions.html) block instead of the `cmake` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.CmakeOptions.html) block, and provide a relative path to your `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) file. Gradle also includes the `Application.mk` (https://developer.android.com/ndk/guides/application_mk.html) file if it is located in the same directory as your `Android.mk` (https://developer.android.com/ndk/guides/android_mk.html) file.

Specify optional configurations

You can specify optional arguments and flags for CMake or ndk-build by configuring another `externalNativeBuild` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ExternalNativeBuildOptions.html) block within the `defaultConfig` block of your module-level `build.gradle` file. Similar to other properties in the `defaultConfig` block, you can override these properties for each product flavor in your build configuration.

For example, if your CMake or ndk-build project defines multiple native libraries, you can use the `targets` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ExternalNativeCmakeOptions.html#com.android.build.gradle.internal.dsl.ExternalNativeCmakeOptions:targets) property to build and package only a subset of those libraries for a given product flavor. The following code sample describes some of the properties you can configure:

```
android {
  ...
  defaultConfig {
    ...
    // This block is different from the one you use to link Gradle
    // to your CMake or ndk-build script.
    externalNativeBuild {

      // For ndk-build, instead use the ndkBuild block.
      cmake {

        // Passes optional arguments to CMake.
        arguments "-DANDROID_ARM_NEON=TRUE", "-DANDROID_TOOLCHAIN=clang"

        // Sets optional flags for the C compiler.
        cFlags "-fexceptions", "-frtti"

        // Sets a flag to enable format macro constants for the C++ compiler.
        cppFlags "-D__STDC_FORMAT_MACROS"
      }
    }
  }

  buildTypes {...}

  productFlavors {
    ...
    demo {
      ...
      externalNativeBuild {
        cmake {
          ...
          // Specifies which native libraries to build and package for this
          // product flavor. If you don't configure this property, Gradle
          // builds and packages all shared object libraries that you define
          // in your CMake or ndk-build project.
          targets "native-lib-demo"
        }
      }
```

```
    }

    paid {
      ...
      externalNativeBuild {
        cmake {
          ...
          targets "native-lib-paid"
        }
      }
    }
  }

  // Use this block to link Gradle to your CMake or ndk-build script.
  externalNativeBuild {
    cmake {...}
    // or ndkBuild {...}
  }
}
```

To learn more about configuring product flavors and build variants, go to Configure Build Variants (https://developer.android.com/studio/build/build-variants.html). For a list of variables you can configure for CMake with the `arguments` property, see Using CMake Variables (https://developer.android.com/ndk/guides/cmake.html#variables).

Specify ABIs

By default, Gradle builds your native library into separate `.so` files for the ABIs the NDK supports (https://developer.android.com/ndk/guides/abis.html#sa) and packages them all into your APK. If you want Gradle to build and package only certain ABI configurations of your native libraries, you can specify them with the `ndk.abiFilters` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.NdkOptions.html) flag in your module-level `build.gradle` file, as shown below:

```
android {
  ...
  defaultConfig {
    ...
    externalNativeBuild {
      cmake {...}
      // or ndkBuild {...}
    }

    // Similar to other properties in the defaultConfig block,
    // you can configure the ndk block for each product flavor
    // in your build configuration.
    ndk {
      // Specifies the ABI configurations of your native
      // libraries Gradle should build and package with your APK.
      abiFilters 'x86', 'x86_64', 'armeabi', 'armeabi-v7a',
                 'arm64-v8a'
    }
  }
  buildTypes {...}
  externalNativeBuild {...}
}
```

In most cases, you only need to specify `abiFilters` in the `ndk` block, as shown above, because it tells Gradle to both build and package those versions of your native libraries. However, if you want to control what Gradle should build, independently of what you want it to package into your APK, configure another `abiFilters` flag in the `defaultConfig.externalNativeBuild.cmake` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ExternalNativeCmakeOptions.html#com.android.build.gradle.internal.dsl.ExternalNativeCmakeOptions:abiFilters) block (or `defaultConfig.externalNativeBuild.ndkBuild` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ExternalNativeNdkBuildOptions.html#com.android.build.gradle.internal.dsl.ExternalNativeNdkBuildOptions:abiFilters) block). Gradle builds those ABI configurations but only packages the ones you specify in the `defaultConfig.ndk` (http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.NdkOptions.html) block.

To further reduce the size of your APK, consider configuring multiple APKs based on ABI (https://developer.android.com/studio/build/configure-apk-splits.html#configure-abi-split)—instead of creating one large APK with the all versions of your native libraries, Gradle creates a separate APK for each ABI you want to support and only packages the files each ABI needs. If you configure multiple APKs per ABI without specifying the `abiFilters` flag as shown in the code sample above, Gradle builds all supported ABI versions of your native libraries, but only packages those you specify

in your multiple APK configuration. To avoid building versions of your native libraries that you don't want, provide the same list of ABIs for both the `abiFilters` flag and your per-ABI multiple APK configuration.

Migrate from ndkCompile

If you're using the deprecated `ndkCompile`, you should migrate to using either CMake or ndk-build. Because `ndkCompile` generates an intermediate `Android.mk` file for you, migrating to ndk-build may be a simpler choice.

To migrate from `ndkCompile` to ndk-build, proceed as follows:

1. Compile your project with `ndkCompile` at least once by selecting **Build > Make Project**. This generates the `Android.mk` file for you.
2. Locate the auto-generated `Android.mk` file by navigating to ***project-root/module-root***`/build/intermediates/ndk/debug/Android.mk`.
3. Relocate the `Android.mk` file to some other directory, such as the same directory as your module-level `build.gradle` file. This makes sure that Gradle doesn't delete the script file when running the `clean` task.
4. Open the `Android.mk` file and edit any paths in the script such that they are relative to the current location of the script file.
5. Link Gradle to the `Android.mk` file (#link-gradle).
6. Disable `ndkCompile` by opening the `build.properties` file and removing the following line:

```
// Remove this line
android.useDeprecatedNdk = true
```

7. Apply your changes by clicking **Sync Project** in the toolbar.