MOBILE APPLICATION DEVELOPMENT

ANDROID (2017)

LECTURE 23: NDK (PART 2)

MEMORY MANAGEMENT

- As discussed previously, using C or most other native code with Java means that the programmer must address memory management.
- Simply freeing memory for an underlying C object when a wrapper Java object goes out of scope is insufficient, because the Java object (and its reference to the C object) may have been duplicated.
- The programmer must create a system by which they can know for sure that a given C object should be deallocated, using a memory management strategy that allows for things like copies of objects.

REFERENCE COUNTING

- Reference counting is a form of manual memory management which associates a 'retain count' with each instance of allocated memory that the programmer creates. It allows the program to understand how many references there are to a particular piece of memory.
- Each time a new object needs to hold onto the same piece of memory, the retain count for that memory is incremented.
- Each time an object no longer needs a particular piece of memory, the retain count for that memory is decremented.
- When the retain count for a given piece of memory hits 0 (or whatever value the programmer defines to mean 'no retainers'), that memory should be freed.

WRAPPING MEMORY-MANAGED NATIVE CODE

- Many classes in C++ and other native languages manage their own memory (in those languages). For example, C++ classes use what are called 'destructors' to free their own memory when the last reference to an instance of that class goes out of scope.
- When using these classes with Java wrapper classes, the memory management model of the original language is broken, because the class instances must be referenced via a pointer in the Java wrapper class.
- Even for memory managed native languages, the programmer needs to implement their own memory management to handle wrapped instances of native classes.

ORGANIZING NATIVE COMPONENTS

- By default, Android studio sets up its native code example by linking to a single C++ file which is used as a native library. This is fine if only one part of the application needs access to native code, but it quickly becomes difficult to manage.
- Organizing native code into files specific to the Java classes that use them is a structure that is easier to manage as well as more compartmentalized.
- Having a separate library for the code being wrapped as well as individual files for JNI wrapper code helps keep the important parts of the native code portable.

DESIGNING FOR MULTIPLE WRAPPER LANGUAGES

- The point of using native code for portability is being able to use shared functionality on multiple platforms without having to frequently rewrite code.
- With this in mind, it should also be a goal of most shared codebases to fit in well with the languages that will be wrapping them on specific platforms.
- Avoid trying to expose certain features of native code to wrapper languages if those languages cannot use those features intuitively (examples: pointers in Java, unions, etc.).
- Don't try to reinvent complex parts of the wrapper languages that don't need reinvention (threading, time libraries, encryption).