Multi Language Projects

Why mix languages?

- Reuse existing libraries
- Take advantage of developer expertise
- No choice (platform constraints)
- Mix dynamic and static languages (ie scripting language in a C++ game engine)
- Transition to greener pastures (ie gradual “rewrite it in Rust”)

Easy mode: compiled languages with the same ABI

- Calling C code is easy from basically every compiled langauge (C++, Rust, D, etc)
- The C ABI is stable + relatively simple on all platforms
- ABI = “Application Binary Interface” which basically means where do function parameters go, and where does the return value go
- For example, on ARM64 (I think on all OSs) Parameters get passed in registers `x0 to x7`, and the return values go in registers `x0` and `x1`

An issue: linking and function overloading

- Recall the steps to building a program: Compile into object files then Link into executables or libraries
- Object files/libs have “symbol tables” which map function names to the address of their code
- We refer to functions in other object files by name until linking, because we can't know the address until then
- Depending on whether we have static or dynamic libraries, the name gets “resolved” to an address at link time or runtime
- Each symbol can only map to one address, so what happens in my C++ code with multiple overloads of `f`? `void f(int)` and `double f(int, int)` for example

Name Mangling

- To interoperate with the existing linker system, C++ “mangles” each overload to a different symbol name, something like f__i and f__ii
- In fact, because it doesn't know if a function might be overloaded, ALL function symbols in C++ are mangled
- Calling code needs to know that so that it uses the right symbol
- If my C++ code wants to call C code, it should NOT mangle the name
- In C++ we can specify “linkage” with the extern keyword, most commonly we use extern "C" meaning “these are C functions, they can't be overloaded, and their symbols are unmangled”
- We use this to let our C++ functions be called from C easily, or to specify that a function definition is from a library using C linkage

Next issue: passing objects

- Passing primitive types between function is pretty easy (ints, floats, pointers, etc)
- What about objects?
- Both the caller and callee need to know the layout of the object to know its size and where each of its fields are
- This might be harder than it sounds. By default the rust compiler is allowed to reorder struct fields to optimize its size, so equivalent C and rust definitions might have different layouts!
- If we're going to try to call virtual methods (methods you can override when doing inheritance), we need to understand the layout of the “vtable” which stores pointers to the various virtual methods

Serialization/Deserialization

- One way to skirt this issue is to share objects in a format that any language knows how to produce/consume
- Examples of this would be JSON strings or Google's Protobuf library
- When passing objects between languages we convert between the "native" representations via the serialization format. What do we do with reference (pointer) fields?
- This is also very expensive, but may be necessary, especially if we're sharing data across processes or across the internet

A scenario: embedding a “scripting” language in a compiled application

- This comes up in game engines all the time. Lots of game code is written in languages like python or other special purpose scripting languages, but the engine is written in C or C++
- How do you call python functions from C++? First you need an interpreter
- These languages usually have an “embedding” API that you can use
- There is some object representing the “interpreter state” and methods for executing some python code on that interpreter, possibly taking python code as a string
- For python, to pass objects from C++ to python you'll need convert native types to python types, such as the in-memory `PyObject` type

Java, JNI

- We're interested in calling native code from the JVM which is running our mostly Java/Kotlin code
- The bridge is called the “Java Native Interface”
- Basically, we declare functions in our Java code (specify their names + signatures) that will be implemented in C++.
- The C++ functions are passed an “environment” object representing the state of the JVM in addition to other parameters
- The JNI defines a bunch of functions that can be called from C++ to access members/methods of Java objects so the C++ compiler doesn't need to know the layout of Java objects (everything is passed as a pointer)

Loading Native Code

- Any native code must be compiled to a “shared library” (on windows these are .dll files, .dylib on mac, and .so on traditional unixes, including Android)
- The shared library is loaded at runtime via the `System.LoadLibrary` method which takes the “undecorated” library name (filenames are typically `libmylibrary.so`, so `mylibrary` is the undecorated name)
- When we call one of the native methods that we declared, it will find the matching symbol in the shared library and call it with the appropriate parameters

The JNI functions

- Our C++ code gets passed “opaque” pointers from Java meaning we can't directly access their members/methods
- We call methods from the JNIEnv class to do that
- Example: to call a method
  - use `GetObjectClass` to get the class object
  - use `GetMethodID` to get an ID for the method of a given signature in that class (we specify the signature as a weirdly formatted string)
  - use `CallLongMethod` or similar to run the method with the given ID with a particular object
- We'll see a complete example in today's example code

Integration with Android

- We include out C++ code in our Android project, and can provide a `CMakeLists` file that gradle will run for us to build the appropriate `.so` file
- In the project menu, right click your main package and select “add C++ to module” to get some template code + CMake files
- If you declare an `external` fun, android studio will provide a “quick fix” option to create the method with the appropriate signature in your c++ code
- It also helps with other annoying things, like specifying the method signatures when calling `GetMethodID`

The pieces

```
//declaration in Kotlin
private external fun someFun(p1: Long, p2: SomeClass): Int

//in a .cpp file
extern "C"
JNIEXPORT jint JNICALL
Java_com_example_mypackage_SomeClass_someFun(JNIEnv *env, jobject thiz, jlong p1,
    jobject p2){ ... }
```

The pieces

```
//declaration in Kotlin
private external fun someFun(p1: Long, p2: SomeClass): Int

//in a .cpp file
extern "C"
JNIEXPORT jint JNICALL
Java_com_example_mypackage_SomeClass_someFun(JNIEnv *env, jobject thiz, jlong p1,
    jobject p2){ ... }
```

CMake recap

- CMake is a “meta build system” which lets you describe how to build a program/library, and the outputs low level instructions for a tool like `make` or `ninja`
- For our projects, our main goal is to build a shared library, so we'll use the CMake `add_library` function
- We specify the source files and other relevant build info
- Gradle will run CMake and then Ninja to build our code
- The Android studio CMake integration seems to be really good!
- The CMake language is terrible

Today's Example: Simple “game”

- We want to write a “high performace” game using a fast physics simulator
- We'll use Box2D which is a really popular engine written in C++
- We'll use Compose for the UI, so part of our app will be C++ code, and part will be in Kotlin
- How should we split it up?

Considerations

- Box2D is a pretty big library, to use B2D stuff from Kotlin, we'll need to write a bunch of Kotlin classes wrapping the C++ types.
- We can easily inspect/access Java types from C++ with the JNI functions, but not vice versa! (why?)
- Cross language calls are relatively slow because of type conversions, and the fact that the JIT compiler can't optimize our C++ code

The plan

- Most of our “game” code will be C++
- We'll write a bridge function with a signature like `step(uiData)` which will be implemented in C++
- It will step the simulation and the update the uiData with new object positions, etc
- It'll be called once a frame so we won't have many cross language calls, and only a small amount of code will need to deal with translating between C++/Java types

Hacks

- In order to avoid global vars in our C++ code, we'll store pointers to data in Java objects as `long`s
- We'll use `reinterpret_cast` to treat the long as a pointer when we need it.
- We'll be mutating our game state, not creating new objects each frame, as we've seen compose doesn't like this...
- We'll trigger timesteps in a `LaunchedEffect` and have an extra, dummy state variable to trigger our Canvas to be redrawn... there's probably better alternatives

Recap

- Mixed language projects aren't uncommon
- The languages + tools available can mitigate the pain caused by this
- JNI is straightforward, but pretty clunky