

---

## Unity Native Scripting

A library to allow writing Unity scripts in native code: C, C++, assembly.

### Purpose

This project aims to give you a viable alternative to C#. Scripting in C++ isn't right for all parts of every project, but now it's an option.

### Goals

- Make scripting in C++ as easy as C#
- Low performance overhead
- Easy integration with any Unity project
- Fast compile, build, and code generation times
- Don't lose support from Unity Technologies

## Reasons to Prefer C++ Over C#

### Fast Device Build Times

Changing one line of C# code requires you to make a new build of the game. Typical Android build times tend to be at least 10 minutes because IL2CPP has to run and then a huge amount of C++ must be compiled.

By using C++, we can compile the game as a C++ plugin in about 1 second, swap the plugin into the APK, and then immediately install and run the game. That's a huge productivity boost!

### Fast Compile Times

C++ compiles much more quickly than C#. Incremental builds when just one file changes– the most common builds– can be 15x faster than with C#. Faster compilation adds up over time to productivity gains. Quicker iteration times make it easier to stay in the “flow” of programming.

---

## No Garbage Collector

Unity's garbage collector is mandatory and has a lot of problems. It's slow, runs on the main thread, collects all garbage at once, fragments the heap, and never shrinks the heap. So your game will experience "frame hitches" and eventually you'll run out of memory and crash.

A significant amount of effort is required to work around the GC and the resulting code is difficult to maintain and slow. This includes techniques like object pools, which essentially make memory management manual. You've also got to avoid boxing value types like `int` to managed types like `object`, not use `foreach` loops in some situations, and various other gotchas.

C++ has no required garbage collector and features optional automatic memory management via "smart pointer" types like `shared_ptr`. It offers excellent alternatives to Unity's primitive garbage collector.

While using some .NET APIs will still involve garbage creation, the problem is contained to only those APIs rather than being a pervasive issue for all your code.

## Total Control

By using C++ directly, you gain complete control over the code the CPU will execute. It's much easier to generate optimal code with a C++ compiler than with a C# compiler, IL2CPP, and finally a C++ compiler. Cut out the middle-man and you can take advantage of compiler intrinsics or assembly to directly write machine code using powerful CPU features like SIMD and hardware AES encryption for massive performance gains.

## More Features

C++ is a much larger language than C# and some developers will prefer having more tools at their disposal. Here are a few differences:

- Its template system is much more powerful than C# generics
- There are macros for extreme flexibility by generating code
- Cheap function pointers instead of heavyweight delegates
- No-overhead algorithms instead of LINQ
- Bit fields for easy memory savings
- Pointers and never-null references instead of just managed references
- Much more. C++ is huge.

---

## No IL2CPP Surprises

While IL2CPP transforms C# into C++ already, it generates a lot of overhead. There are many surprises if you read through the generated C++. For example, there's overhead for any function using a static variable and an extra two pointers are stored at the beginning of every class. The same goes for all sorts of features such as `sizeof()`, mandatory null checks, and so forth. Instead, you could write C++ directly and not need to work around IL2CPP.

## Industry Standard Language

C++ is the standard language for video games as well as many other fields. By programming in C++ you can more easily transfer your skills and code to and from non-Unity projects. For example, you can avoid lock-in by using the same language (C++) that you'd use in the Unreal or Lumberyard engines.

## UnityNativeScripting Features

- Code generator exposes any C# API to C++
- Supports Windows, macOS, Linux, iOS, and Android (editor and standalone)
- Works with Unity 2017.x and 5.x
- Plays nice with other C# scripts- no need to use 100% C++
- Object-oriented API just like in C#

```
GameObject go; Transform transform = go.GetTransform(); Vector3 position(1.0f, 2.0f, 3.0f);  
transform.SetPosition(position);
```

- Hot reloading: change C++ without restarting the game
- Handle `MonoBehaviour` messages in C++

```
void MyScript::Start() { String message("MyScript has started"); Debug::Log(message); }
```

- Platform-dependent compilation (e.g. `#if TARGET_OS_ANDROID`)
- CMake build system sets up any IDE project or command-line build

## Code Generator

The core of this project is a code generator. It generates C# and C++ code called "bindings" that make C# APIs available to C++ game code. It supports a wide range of language features:

- 
- Types
    - `class`
    - `struct`
    - `enum`
    - Arrays (single- and multi-dimensional)
    - Delegates (e.g. `Action`)
    - `decimal`
  - Type Contents
    - Constructors
    - Methods
    - Fields
    - Properties (`get` and `set` like `obj.x`)
    - Indexers (`get` and `set` like `obj[x]`)
    - Events (`add` and `remove` delegates)
    - Overloaded operators
    - Boxing and unboxing (e.g. casting `int` to `object` and visa versa)
  - Function Features
    - `out` and `ref` parameters
    - Generic types and methods
    - Default parameters
  - Cross-Language Features
    - Exceptions (C# to C++ and C++ to C#)
    - Implementing C# interfaces with C++ classes
    - Deriving from C# classes with C++ classes

Note that the code generator does not yet support:

- `Array`, `string`, and `object` methods (e.g. `GetHashCode`)
- Non-null string default parameters and null non-string default parameters
- Implicit `params` parameter (a.k.a. “var args”) passing
- C# pointers
- Nested types
- Down-casting

To configure the code generator, open `Unity/Assets/NativeScriptTypes.json` and notice the existing examples. Add on to this file to expose more C# APIs from Unity, .NET, or custom DLLs to your C++ code.

---

To run the code generator, choose [NativeScript](#) > [Generate Bindings](#) from the Unity editor.

## Performance

Almost all projects will see a net performance win by reducing garbage collection, eliminating IL2CPP overhead, and access to compiler intrinsics and assembly. Calls from C++ into C# incur only a minor performance penalty. In the rare case that almost all of your code is calls to .NET APIs then you may experience a net performance loss.

[Testing and benchmarks article](#)

[Optimizations article](#)

## Project Structure

When scripting in C++, C# is used only as a “binding” layer so Unity can call C++ functions and C++ functions can call the Unity API. A code generator is used to generate most of these bindings according to the needs of your project.

All of your code, plus a few bindings, will exist in a single “native” C++ plugin. When you change your C++ code, you’ll build this plugin and then play the game in the editor or in a deployed build (e.g. to an Android device). There won’t be any C# code for Unity to compile unless you run the code generator, which is infrequent.

The standard C# workflow looks like this:

1. Edit C# code in a C# IDE like MonoDevelop
2. Switch to the Unity editor window
3. Wait for the compile to finish (slow, “real” games take 5+ seconds)
4. Run the game

With C++, the workflow looks like this:

1. Edit C++ code in a C++ IDE like Xcode or Visual Studio
2. Build the C++ plugin (extremely fast, often under 1 second)
3. Switch to the Unity editor window. Nothing to compile.
4. Run the game

---

## Getting Started

1. Download or clone this repo
2. Copy everything in `Unity/Assets` directory to your Unity project's `Assets` directory
3. Edit `NativeScriptTypes.json` and specify what parts of the Unity, .NET, and custom DLL APIs you want access to from C++.
4. Edit `Unity/Assets/CppSource/Game/Game.cpp` and `Unity/Assets/CppSource/Game/Game.h` to create your game. Some example code is provided, but feel free to delete it. You can add more C++ source (`.cpp`) and header (`.h`) files here as your game grows.

## Building the C++ Plugin

### ios

1. Install CMake version 3.6 or greater
2. Create a directory for build files. Anywhere is fine.
3. Open the Terminal app in `/Applications/Utilities`
4. Execute `cd /path/to/your/build/directory`
5. Execute `cmake -G MyGenerator -DCMAKE_TOOLCHAIN_FILE=/path/to/your/project/CppSource/iOS.cmake /path/to/your/project/CppSource`. Replace `MyGenerator` with the generator of your choice. To see the options, execute `cmake --help` and look at the list at the bottom. Common choices include “Unix Makefiles” to build from command line or “Xcode” to use Apple’s IDE.
6. The build scripts or IDE project files are now generated in your build directory
7. Build as appropriate for your generator. For example, execute `make` if you chose `Unix Makefiles` as your generator or open `NativeScript.xcodeproj` and click `Product > Build` if you chose Xcode.

### macOS (Editor and Standalone)

1. Install CMake version 3.6 or greater
2. Create a directory for build files. Anywhere is fine.
3. Open the Terminal app in `/Applications/Utilities`
4. Execute `cd /path/to/your/build/directory`
5. Execute `cmake -G "MyGenerator"-DEDITOR=TRUE /path/to/your/project/CppSource`. Replace `MyGenerator` with the generator of your choice. To see the options, execute `cmake --help` and look at the list at the bottom. Common choices

---

include “Unix Makefiles” to build from command line or “Xcode” to use Apple’s IDE. Remove `-DEDITOR=TRUE` for standalone builds.

6. The build scripts or IDE project files are now generated in your build directory
7. Build as appropriate for your generator. For example, execute `make` if you chose `Unix Makefiles` as your generator or open `NativeScript.xcodeproj` and click `Product > Build` if you chose Xcode.

## Windows (Editor and Standalone)

1. Install CMake version 3.6 or greater
2. Create a directory for build files. Anywhere is fine.
3. Open a Command Prompt by clicking the Start button, typing “Command Prompt”, then clicking the app
4. Execute `cd /path/to/your/build/directory`
5. Execute `cmake -G "Visual Studio VERSION YEAR Win64"-DEDITOR=TRUE /path/to/your/project/CppSource`. Replace `VERSION` and `YEAR` with the version of Visual Studio you want to use. To see the options, execute `cmake --help` and look at the list at the bottom. For example, use `"Visual Studio 15 2017 Win64"` for Visual Studio 2017. Any version, including Community, works just fine. Remove `-DEDITOR=TRUE` for standalone builds. If you are using Visual Studio 2019, execute `cmake -G "Visual Studio 16"-A "x64"-DEDITOR=TRUE /path/to/your/project/CppSource` instead.
6. The project files are now generated in your build directory
7. Open `NativeScript.sln` and click `Build > Build Solution`.

## Linux (Editor and Standalone)

1. Install CMake version 3.6 or greater
2. Create a directory for build files. Anywhere is fine.
3. Open a terminal as appropriate for your Linux distribution
4. Execute `cd /path/to/your/build/directory`
5. Execute `cmake -G "MyGenerator"-DEDITOR=TRUE /path/to/your/project/CppSource`. Replace `MyGenerator` with the generator of your choice. To see the options, execute `cmake --help` and look at the list at the bottom. The most common choice is “Unix Makefiles” to build from command line, but there are IDE options too. Remove `-DEDITOR=TRUE` for standalone builds.
6. The build scripts or IDE project files are now generated in your build directory

- 
7. Build as appropriate for your generator. For example, execute `make` if you chose `Unix Makefiles` as your generator.

## Android

1. Install CMake version 3.6 or greater
2. Create a directory for build files. Anywhere is fine.
3. Open a terminal (macOS, Linux) or Command Prompt (Windows)
4. Execute `cd /path/to/your/build/directory`
5. Execute `cmake -G MyGenerator -DANDROID_NDK=/path/to/android/ndk /path/to/your/project/CppSource`. Replace `MyGenerator` with the generator of your choice. To see the options, execute `cmake --help` and look at the list at the bottom. To make a build for any platform other than Android, omit the `-DANDROID_NDK=/path/to/android/ndk` part.
6. The build scripts or IDE project files are now generated in your build directory
7. Build as appropriate for your generator. For example, execute `make` if you chose `Unix Makefiles` as your generator.

## Updating To A New Version

To update to a new version of this project, overwrite your Unity project's `Assets/NativeScript` directory with this project's `Unity/Assets/NativeScript` directory and re-run the code generator.

## Reference

Articles by the author describing the development of this project.

## Author

Jackson Dunstan

## Contributing

Please feel free to fork and send pull requests or simply submit an issue for features or bug fixes.



---

## License

All code is licensed MIT, which means it can usually be easily used in commercial and non-commercial applications.

All writing is licensed CC BY 4.0, which means it can be used as long as attribution is given.