

10 Native Programming

Contents

[Overview 3](#_Toc508201243)

[Introduction 4](#_Toc508201244)

[10.1. Extending C++ into Blueprint 4](#_Toc508201245)

[10.2. Hot Reload 5](#_Toc508201246)

[Exercises 6](#_Toc508201247)

[Exercise 10A: Developing a C++ Actor 6](#_Toc508201248)

[Exercise 10B: Adding Basic C++ Code 7](#_Toc508201249)

# Overview

**PURPOSE**

The purpose of this tutorial is to introduce the concepts and processes required to write native Unreal Engine 4 classes and to understand how to allow those classes to interact with Blueprint.

**SCOPE**

This section will provide an introductory overview of the following:

* Setting up your development environment
* Comparing C++ and Blueprint
* Programming workflow
* Developing a simple Blueprint Node

**PREREQUISITES**

It is assumed that you have some C++ experience. If you don’t but know C#, or another C-style language, you should find many aspects familiar. We also don’t teach any general programming, so it is also assumed that you have an understanding of programming and the associated concepts.

|  |  |
| --- | --- |
| C:\Users\Melanie Nikdel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tips-png-4.png | NOTE  You must install Visual Studio before beginning the exercises. For more information, refer to  <https://docs.unrealengine.com/latest/INT/Programming/Development/VisualStudioSetup/>. |
|  |  |

## Introduction

In Unreal Engine 4, there are two methods available to create new gameplay elements, Unreal C++ and Blueprints Visual Scripting. Using C++, programmers can add the base gameplay systems that designers can then build upon and/or use to create the custom gameplay for a specific level or the game itself. Both methods are quite flexible, and it’s possible to use either to develop functionality; however, there are situations where one method may have advantages over the other.

* **Blueprint**

Blueprint is a node-based visual scripting language designed to make it fast and easy to develop functionality for your projects. The system is an extremely flexible and powerful tool that allows designers to program gameplay behaviors without taking up engineer time. In addition, it has a very short iteration time, allowing you to quickly iterate prototypes and features. The iteration of visual effects is also much more intuitive, allowing artists to quickly and easily implement their own art in the engine.

* **C++**

C++, although more complex, has many strengths. Native debugging tools are more capable and powerful, making it easier to track down issues. Native source files are also much more suitable for source control packages, giving you easy access to merge and diff tools that aren’t reliant on the Unreal Editor. Some areas, such as mathematical operations, are less verbose and easier to maintain in native C++. There are also many areas of the code base that can only be accessed through the C++ API.

Both systems are powerful by themselves; however, the engine really shines when these methods are used in conjunction with each other. Programmers can create the gameplay building blocks in C++, and designers can then take those blocks and develop interesting gameplay.

## 10.1. Extending C++ into Blueprint

For our native class to be eligible as a base class for Blueprint, it first needs to be marked as a “**Blueprintable**”class. This is done by adding the “**Blueprintable**”specifier to the“**UCLASS() macro**”. This macro is automatically added when the class is generated by the **Class Wizard**.

The **Blueprintable** specifier is inherited by subclasses, so it doesn’t need to be added if it already exists in the parent class. You can use the “**NotBlueprintable**”specifier should you wish to negate this behavior in the base class.

The process for creating Blueprint classes that extend your native classes is the same as that for extending any other class. When you create a new Blueprint class, your native classes should appear in the list of base classes in the “**Pick Parent Class**” dialog box.

## 10.2. Hot Reload

Hot Reload is an Unreal Engine feature that you might be surprised about if you are accustomed to programming C++ in other projects. The **Hot Reload** feature allows you to compile your C++ changes without shutting down the editor!

This can be done in one of two ways:

1. With the editor still running, go ahead and build from **Visual Studio** like you normally would. The editor will detect the newly compiled DLLs and reload your changes instantly.
2. Or, simply click the **Compile** button on the editor's main toolbar.

Hot Reload is a powerful feature that decreases your iteration time and eliminates time lost through the continuous closing and reopening of the Editor.

# Exercises

## Exercise 10A: Developing a C++ Actor

**Deliverables:** Packaged game files (as a .zip with student details)

**Instructor Task:** Discuss and demonstrate the creation of a custom Actor using the Unreal C++ development environment.

**Student Task:** For this exercise, you will create a new UE4 Actor within the C++ development environment.

**Directions**

1. Open **Unreal Engine** from the Launcher. The [Project Browser](https://docs.unrealengine.com/latest/INT/Engine/Basics/Projects/Browser/index.html) will appear.
2. Click on the **New Project** tab and then select the **C++** tab. From there, select **Basic Code** for a clean starting point, and make sure **With Starter Content** is set. Set “QuickStart” as the project name and click **Create Project**.
3. In the **Unreal Editor**, create a new C++ class with the **New C++ Class...** command, located in the **File** drop-down menu.
4. The **Choose Parent Class** menu will open. Since **Actor** is the most basic class that can be placed in an **Unreal Engine** level, use the **Actor** class as the base.
5. The **Name Your New Actor** menu will open. For this example, enter the name “FloatingActor”, and then click **Create Class.**
6. Now that you have created a C++ class, you can switch to Visual Studio to program it. FloatingActor.cpp will be opened automatically, and Unreal Engine will automatically compile and reload the code with our new class.

|  |  |
| --- | --- |
| C:\Users\Melanie Nikdel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tips-png-4.png | NOTE  For more information on the above process, see  <https://docs.unrealengine.com/latest/INT/Programming/QuickStart/index.html>. |

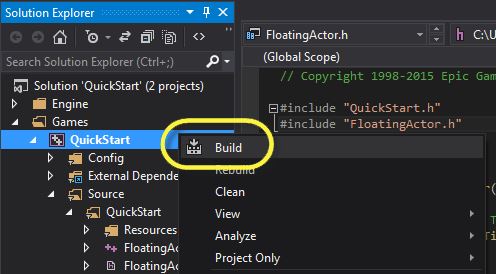
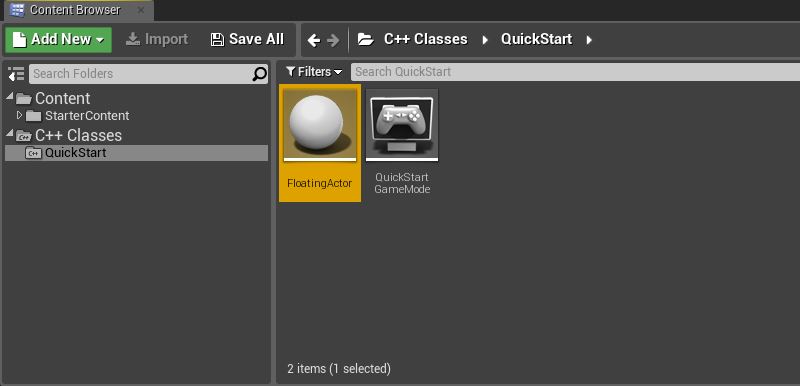
## Exercise 10B: Adding Basic C++ Code

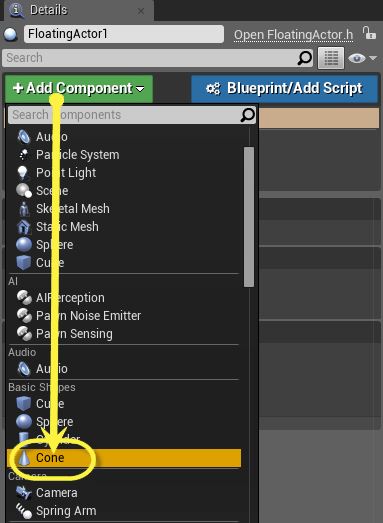
**Deliverables:** Packaged game files (as a .zip with student details)

**Instructor Task:** Teach the basics of adding C++ code to a class and extending it with Blueprint.

**Student Task:** Develop basic C++ code for a “Floating Actor” class and derive a Blueprint class from it to view it in-game.

**Directions**

1. Visual Studio will have automatically opened upon creating the C++ class in the last exercise. Open “FloatingActor.h” and create a variable called **RunningTime** by adding “***float*** *RunningTime;*” before the closing brace at the end of the file.
2. In FloatingActor.cpp, add the following code:
3. CompileFromEditor.pngCompile the code by right-clicking on the Project name in the Solution Explorer and selecting Build or by selecting Compile from Unreal Editor after saving the files.  
   
4. In the Content Browser, expand the folder called “C++ Classes”. Within your project folder you will see the **FloatingActor** class. 
5. Drag the **FloatingActor** class directly into the level editor window to add it to the world and, while it’s still selected, choose “Add Component” from the details panel. Add a Cone component to see the Actor in the world.



1. Hit Play to ensure your new Actor works.
2. Save the version of the **FloatingActor**, featuring a cone, as a Blueprint by clicking “Blueprint/Add Script” in the Details panel.
3. You’re done! Don’t forget to save your work for submission and inform your instructor.