# Module 4 - Set up the First the Multiplayer Environment

*COMP280-Multiplayer Game Programming*

**Purpose:** Set up the First the Multiplayer Environment

## Contents

## 1. Intro

We will:

- Overview what we want to build.
- Create an Unreal project and get things set up properly. This involves:
    - Creating the Gameplay Framework (GF) classes (so that you have access to all the necessary features needed for development)
    - Configuring the project settings to use such classes.
- Understand basics of programming in UE5 with C++
- Setup the foundation for the multiplayer game.

Topics:

- Introducing *Unreal Shadows – Legacy of the Lichlord* (or *US_LOTL*)
- Understanding C++ in Unreal Engine
- Starting the Unreal multiplayer project
- Adding the player classes

### 1.1. Technical requirements

- We are going to use Unreal Engine 5.4.
- Some basic knowledge about C++ programming

## 2. Introducing Unreal Shadows – Legacy of the Lichlord

Consider the following passage from a (never published) fantasy book called *Unreal Shadows – Legacy of the Lichlord*:

**“The air was thick with the stench of decay as the three thieves made their way into the Lichlord’s dungeon. Their mission was clear: infiltrate the fortress, find the king’s knight, and bring him back alive. Anything they could find lying around was theirs to take home.**

**As they crept through the shadowy corridors, they all knew that they were not alone: dozens of undead minions lurked around every corner, their greenish eyes staring blankly.**

**The guild had faced their fair share of dangerous foes before, but never had they encountered such a formidable army of the undead. Quietly, they slipped past the corridors, careful not to attract any attention. The last thing they needed was to draw the entire horde down upon them.”**

Congratulations – you have just been hired to create the video game adaptation of this best-selling book! And what’s more, it will be a multiplayer game!

### 2.1. Explaining the project brief

The game will be:

- a third-person shooter that...
- ... will use the hide and seek gameplay twist

So, it will be *a stealth game* where players will *survive* only by moving *quietly* and *carefully*.

Each participant will play the role of a member of the Guild of Thieves, a secret organization of rogues and thieves, infiltrating the realm of an undead wizard.

The main aim will be to *rescue non-player characters* that are kept (hopefully alive!) in the dungeon prisons.

Additionally, players will *collect treasures* and *equipment* from past and less lucky adventurers.

Each character will have the ability to do the following:

- Move around by **walking** or **running**
- Handle **a weapon**
- Get **power-ups**
- Equip **new weapons**

- Increase **skills** by means of **experience points**

Because of the need for stealthy movement, the following will create **noise** (BAD):

- movement (especially running)
- wielding weapons

Enemies will be the **minions**, a horde of undead skeletons that will wander around the level unaware of the player characters.

**Excessive noise** from the characters or **falling into traps** will alert nearby enemies, making it nearly impossible to complete the game.

Gaining **experience points** will only be made possible only by defeating enemies, adding further engagement to the overall experience!

> We are concerned here mostly about *multiplayer game programming* rather than game design, so, balancing game mechanics will not be a primary focus of gameplay; instead, the focus will be on **making things function effectively**.

## 2.2. Starting the project

### 2.2.1 Download some 3D assets

- Go to KayKit Dungeon Remastered and click on **Download on itch.io**.
- You could do either this (I will do the second):
  - Create a brand-new project
    - Use the Blank template available in UE5
    - Add downloaded assets from the aforementioned kits (deal with formatting issues)
  - Or, used the prepackaged project (attached as .zip) as a starting point.

# 3. Understanding C++ in Unreal Engine

It is advisable to use the engine's most common features such as the built-in Reflection System and memory management to work effectively with UE's C++.

## 3.1. Blueprints and C++

UE provides two methods for programming your game logic:

- Blueprint Visual Scripting and
- C++

### 3.1.1 Blueprints Visual Scripting ...

... makes it easier for developers who don't have extensive coding experience to create complex game mechanics without writing any code.

### 3.1.2 C++ (as an object-oriented programming — OOP — language) requires more technical knowledge but offers much greater control over the game engine than Blueprints does.

C++ and Blueprints are strictly connected, as Blueprints provide a visual representation of the underlying C++ code and adhere to its principles, including inheritance and polymorphism.

While Blueprints do not demand advanced coding skills, they do follow the same principles as the programming language in terms of data types, pointers, references, and other rules.

Both languages can be used together in UE5 projects and most of the time, what you can achieve with C++ can equally be done in Blueprints; however, C++ excels in customizing core features of UE or creating plugins that extend its functionality even further beyond what’s available out of the box.

Although both Blueprint Visual Scripting and C++ offer powerful toolsets when working with UE projects, C++ provides lower-level access by way of object-oriented coding techniques – that’s why it is very important to have a strong knowledge of it once you start developing multiplayer games.

## 3.2. Understanding C++ classes

An Unreal Engine C++ class is a regular C++ class! The process of creating a new UE C++ class:

- Start by defining what **type** of object it will represent, such as an *Actor* or a *Component*
- Create the **header file (.h)** and declare variables and methods
- Create the **implementation file (*.cpp)** that implements the code logic.

While the source file behaves like a regular C++ file of its kind, the header file will let you declare additional information for variables and functions that will be used by Blueprints inheriting from your class. Additionally, it will ease the pain of managing memory at runtime.

With the release of UE5, Epic Games introduced an amazing inspection tool called **C++ Header Preview** that lets you inspect your Blueprint Classes as if they were written in C++. To activate it, simply select from the main menu **Tools → C++ Header Preview** and select your desired class.

In UE, there are these main class types that you’ll be deriving from during development:

| Class Types | Description |
|---|---|
| UObject | A **UObject** is the base class of UE and provides most of the main features available in UE, such as garbage collection (GC) (yes, UE provides it!), networking support, reflection of properties and methods, and so on. An AActor is a UObject that can be added to a game level either from the Editor or at runtime: in this second case, we say that the Actor has been spawned. In a multiplayer environment, an AActor is the base type that can be replicated during networking and it will provide information for any Component that will need synchronization. |
| UActorComponent | A **UActorComponent** is the basic class for defining Components that will be attached to an Actor or to another Component belonging to the Actor itself. |
| UStruct | A **UStruct** is used to create plain data structures and does not extend from any particular class |
| UEnum | A **UEnum** is used to represent enumerations of elements |

As a final note, throughout this book, you will notice that class names start with some letters that will not be visible once the class is used in the Editor. UE makes use of prefixes to point out the class type. The main prefixes used are as follows:

| Prefix | Description |
|---|---|
| U | generic objects deriving from UObject (for instance, components) |
| A | objects deriving from an Actor (i.e., AActor) and that can be added to a level |
| F | generic classes and structures such as the FColor structure |
| T | templates such as TArray or TMap |
| I | interfaces |
| E | enums |
| B | bool or uint8 (which may be used instead of bool) |

⚠ Most of these prefixes are mandatory; if you try to name a class deriving from Actor without the A prefix, you will get an error. UE will hide the prefix once in the Editor. This rule applies only to C++ classes; Blueprints can be named without such prefixes.

Let's see the main features of a class header in order to understand its core functionalities.

## 3.3. Anatomy of a UE C++ header

The C++ header of an Actor in UE5 will look like the following piece of code:

```
#include "Engine/StaticMEshActor.h"
#include "APickup.generated.h"
UCLASS(Blueprintable, BlueprintType)
class APickup : public AStaticMeshActor
{
  GENERATED_BODY()
public:
  APickup();
  UPROPERTY(BlueprintReadOnly, VisibleAnywhere, Category="Default")
  TObjectPtr<class uspherecomponent=""> Trigger;
  UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Default")
  int32 Points;
  UFUNCTION(BlueprintCallable)
  void Reactivate();
};
```

Notice:

- The #include “APickup.generated.h” declaration. This mandatory line of code is autogenerated by the Unreal Header Tool (UHT) and includes all the macros your code will need to properly work in the engine. This file needs to be the last include file declared in your header or the compiler will throw an error.

> The UHT is a custom parsing and code generation tool that supports the UObject system in Unreal Engine. The UHT is used to parse C++ headers for Unreal and generate the boilerplate code required for the engine to work with the user-created classes.

- The class constructor (in this case, APickup()) is used to set default values for properties as you may do with a regular C++ class; what’s more, you will be using it to create and add Components to the Actor itself.
- Declarations:
  - UCLASS(),
  - GENERATED_BODY(),
  - UPROPERTY()
  - UFUNCTION()
- Attributes (we’ll deal with their meaning next week):
  - BlueprintReadOnly
  - VisibleAnywhere
  - others

## 3.4. The Unreal Engine Reflection System

The term **Reflection** specifies the capability that allows a program to inspect its own structure at runtime.

This is a core technology of UE (developed by Epic - not natively in C++) that supports:

- detail panels in the Editor,
- serialization,
- garbage collection,

- network replication, and
- communication between Blueprint and C++.

To use Reflection, we need to annotate any type or property that you want to make visible to the system by marking them with annotations such as UCLASS(), UFUNCTION(), or UPROPERTIES().

To enable such annotations, we need to use **#include "APickup.generated.h"**

Markup elements accessible within the Reflection System:

| Element | Description |
|---|---|
| UCLASS() | Used to generate reflection data for a class that needs to derive from UObject |
| USTRUCT() | Used to generate reflection data for a struct |
| GENERATED_BODY() | This markup will be replaced with all the needed boilerplate code for the type |
| UPROPERTY() | Used to tell the engine that the associated member variable will have some additional features, such as Blueprint accessibility or replication across the network (and this will mean a lot to you later on!) |
| UFUNCTION() | Allows, among other things, to call this function from an extending Blueprint Class or override the function from the Blueprint itself |

The Reflection System is also used by the garbage collector so you don't have to worry about memory management, as you will see in the next subsection.

## 3.5. Memory management and garbage collection (GC)

**GC** is an essential part of programming that makes sure your program runs smoothly without any memory leaks or performance issues, so you can focus on creating awesome features instead.

C++ does not natively implement GC, so UE implements its own system: you will just need to ensure that valid references to the objects are maintained.

In order to enable GC for your classes, you need to assure that they inherit from UObject; then the system will keep a list of objects (also called root) that should not be garbage-collected. As long as an object is listed in the root, it will be preserved from deletion; once it is removed from this list, it will be deleted from memory the next time the garbage collector is called (i.e., at certain intervals).

> Actors are only destroyed at the level's shutdown unless you call the Destroy() method on them: in this case, they will be immediately removed from the game and deleted by the garbage collector when activated.

Let's create an empty project and extend the main Unreal GF classes.

# 4. Starting your Unreal multiplayer project

## 4.1. Creating your project file

Let’s start by creating a blank project:

1. Open the Epic Games Launcher and launch the Unreal Editor.
2. From the Games section, select the **Blank template**.
3. In Project Defaults, select **C++** as the project type.
4. Make sure the Starter Content field is **unselected** as you won’t be using it.
5. Name the project **UShadows_LOTL_{YourInitials}** or simply **US_LOTL_{YourInitials}**.
6. Click the **Create** button.
7. Once the UE project has been created, get the attached **UnrealShadows-StarterContent.zip** file and unzip it in your computer.
8. Navigate to your project Content folder located at **[Your Project Path]** → **UShadows_LOTL_{YourInitials}** → **Content**.
9. Copy the content of the unzipped file (*ExternalActors*, Blueprints, KayKit, and Maps folders) into the Content folder to add all the needed assets to your project.
10. Check that they appear in the UE Editor and be available in your project.
11. If they do not,close the Editor and open it again to let UE update the Content folder.

You will notice that I have already added two levels (Level_01 and Level_Boss) to the Maps folder: these levels will be used during the book and have been created for ease of development. You are free to create your own maps or add additional assets. which can be located at **Content** → **KayKit** → **Dungeon Elements**.