Gameplay Classes

Reference for creating and implementing gameplay classes.

Every gameplay class in Unreal Engine consists of a class header file (`.h`) and a class source file (`.cpp`). The class header contains the declarations of the class and its members, such as variables and functions, while the class source file is where the functionality of the class is defined by implementing the functions that belong to the class.

Classes in Unreal Engine have a standardized naming scheme so that you know instantly what kind of class it is simply by looking at the first letter, or prefix. The prefixes for gameplay classes are:

Prefix	Meaning
`A`	Extends from the base class of spawnable gameplay objects. These are Actors, and can be spawned directly into the world.
`U`	Extend from the base class of all gameplay objects. These cannot be directly instanced into the world; they must belong to an Actor. These are generally objects like Components.

Adding Classes

The C++ Class Wizard sets up the header file and source file you need for your new class, and also updates your game module accordingly. The header file and source file automatically include the class declaration and class constructor, as well as Unreal Engine-specific code like the `UCLASS()` macro.

Class Headers

Gameplay classes in Unreal Engine generally have separate and unique class header files. These files are usually named to match the class being defined within, minus the `A` or `U` prefix, and using the `.h` file extension. So, the class header file for the `AActor` class is named `Actor.h`. Although Epic code follows these guidelines, no formal relationship between class name and source file name exists in the current engine.

Class header files for gameplay classes use standard C++ syntax in conjunction with specialized macros to simplify the process of declaring classes, variables, and functions.

At the top of each gameplay class header file, the generated header file (created automatically) needs to be included. So, at the top of `ClassName.h`, the following line must appear:

```
#include "ClassName.generated.h"

```

Copy full snippet

Class Declaration

The class declaration defines the name of the class, what class it inherits from and, thus, any functions and variables it inherits. The class declaration also defines other engine and editor specific behavior that may be desired via class specifiers and metadata.

The syntax for declaring a class is as follows:

```
UCLASS([specifier, specifier, ...], [meta(key=value, key=value, ...)])
class ClassName : public ParentName
{
GENERATED_BODY()
}

```

Copy full snippet

The declaration consists of a standard C++ class declaration for the class. Above the standard declaration, descriptors such as class specifiers and metadata are passed to the `UCLASS` macro. These are used to create the `UClass` for the class being declared, which can be thought of as the engine's specialized representation of the class. Also, the `GENERATED_BODY()` macro must be placed at the very beginning of the class body.

Class Specifiers

When declaring classes, **Class Specifiers** can be added to the declaration to control how the class behaves with various aspects of the Engine and Editor.

Class Specifier	Effect
`Abstract`	The **Abstract** Specifier declares the class as an "abstract base class", preventing the user from adding Actors of this class to Levels. This is useful for classes which are not meaningful on their own. For example, the `ATriggerBase` base class is abstract, while the `ATriggerBox` subclass is not abstract and can be placed in a Level.
`AdvancedClassDisplay`	The **AdvancedClassDisplay** Specifier forces all properties of the class to show only in the advanced sections of any details panel where they appear. To override this on an individual property, use the `SimpleDisplay` specifier on that property.
`AutoCollapseCategories=(Category1, Category2, ...)`	The `AutoCollapseCategories` Specifier negates the effects, for the listed categories, of the **AutoExpandCategories** Specifier on a parent class.
`AutoExpandCategories=(Category1, Category2, ...)`	Specifies one or more categories that should be automatically expanded in the Unreal Editor Property window for Objects of this class. To auto-expand variables declared with no category, use the name of the class which declares the variable.
`Blueprintable`	Exposes this class as an acceptable base class for creating Blueprints. The default is `NotBlueprintable`, unless inherited otherwise. This Specifier is inherited by subclasses.
`BlueprintType`	Exposes this class as a type that can be used for variables in Blueprints.
`ClassGroup=GroupName`	Indicates that Unreal Editor's **Actor Browser** should include this class and any subclass of this class within the specified `GroupName` when **Group View** is enabled in the Actor Browser.
`CollapseCategories`	Indicates that properties of this class should not be grouped in categories in Unreal Editor Property windows. This Specifier is propagated to child classes, and can be overridden by the `DontCollapseCategories` Specifier.

Class Specifier	Effect
`Config=ConfigName`	Indicates that this class is allowed to store data in a configuration file (`.ini`). If there are any class properties declared with the `config` or `globalconfig` Specifiers, this Specifier causes those properties to be stored in the named configuration file. This Specifier is propagated to all child classes and cannot be negated, but child classes can change the config file by re-declaring the `config` Specifier and providing a different `ConfigName`. Common `ConfigName` values are "Engine", "Editor", "Input", and "Game".
`Const`	All properties and functions in this class are `const` and are exported as `const`. This Specifier is inherited by subclasses.
`ConversionRoot`	A root convert limits a subclass to only be able to convert to child classes of the first root class going up the hierarchy.
`CustomConstructor`	Prevents automatic generation of the constructor declaration.
`DefaultToInstanced`	All instances of this class are considered "instanced". Instanced classes (components) are duplicated upon construction. This Specifier is inherited by subclasses.
`DependsOn=(ClassName1, ClassName2, ...)`	All classes listed will be compiled before this class. The class names provided must indicate classes in the same (or a previous) package. Multiple dependency classes can be identified using a single `DependsOn` line delimited by commas, or can be specified using a separate `DependsOn` line for each class. This is important when a class uses a struct or enum declared in another class, as the compiler only knows what is in the classes it has already compiled.
`Deprecated`	This class is deprecated and Objects of this class will not be saved when serializing. This Specifier is inherited by subclasses.
`DontAutoCollapseCategories=(Category, Category, ...)`	Negates the `AutoCollapseCategories` Specifier for the listed categories inherited from a parent class.

Class Specifier	Effect
`DontCollapseCategories`	Negates a `CollapseCatogories` Specifier inherited from a base class.
`EditInlineNew`	Indicates that Objects of this class can be created from the Unreal Editor Property window, as opposed to being referenced from an existing Asset. The default behavior is that only references to existing Objects may be assigned through the Property window). This Specifier is propagated to all child classes; child classes can override this with the `NotEditInlineNew` Specifier.
`HideCategories=(Category1, Category2, ...)`	Lists one or more categories that should be hidden from the user entirely. To hide properties declared with no category, use the name of the class which declares the variable. This Specifier is propagated to child classes.
`HideDropdown`	Prevents this class from showing up in property window combo boxes.
`HideFunctions=(Category1, Category2, ...)`	Hides all functions in the specified category from the user entirely.
`HideFunctions=FunctionName`	Hides the named functions from the user entirely.
`Intrinsic`	This indicates that the class was declared directly in C++, and has no boilerplate generated by **Unreal Header Tool**. Do not use this Specifier on new classes.
`MinimalAPI`	Causes only the class's type information to be exported for use by other modules. The class can be cast to, but the functions of the class cannot be called (with the exception of inline methods). This improves compile times by not exporting everything for classes that do not need all of their functions accessible in other modules.
`NoExport`	Indicates that this class's declaration should not be included in the automatically-generated C++ header file by the header generator. The C++ class declaration must be defined manually in a separate header file. Only valid for native classes. Do not use this for new classes.

Class Specifier	Effect
`NonTransient`	Negates a `Transient` Specifier inherited from a base class.
`NotBlueprintable`	Specifies that this class is not an acceptable base class for creating Blueprints. This is the default and is inherited by subclasses.
`NotPlaceable`	Negates a `Placeable` Specifier inherited from a base class. Indicates that Objects of this class may not be placed into a Level, UI scene, or Blueprint in the Editor.
`PerObjectConfig`	Configuration information for this class will be stored per Object, where each object has a section in the `.ini` file named after the Object in the format `[ObjectName ClassName]`. This Specifier is propagated to child classes.
`Placeable`	Indicates that this class can be created in the Editor and placed into a level, UI scene, or Blueprint (depending on the class type). This flag is propagated to all child classes; child classes can override this flag using the `NotPlaceable` Specifier.
`ShowCategories=(Category1, Category2, ...)`	Negates a `HideCategories` Specifier (inherited from a base class) for the listed categories.
`ShowFunctions=(Category1, Category2, ...)`	Shows all functions within the listed categories in a property viewer.
`ShowFunctions=FunctionName`	Shows the named function in a property viewer.
`Transient`	Objects belonging to this class will never be saved to disk. This is useful in conjunction with certain kinds of native classes which are non-persistent by nature, such as players or windows. This Specifier is propagated to child classes, but can be overridden by the `NonTransient` Specifier.
`Within=OuterClassName`	Objects of this class cannot exist outside of an instance of an `OuterClassName` Object. This means that creating an Object of this class requires that an instance of `OuterClassName` is provided as its `Outer` Object.

Metadata Specifiers

When declaring classes, interfaces, structs, enums, enum values, functions, or properties, you can add **Metadata Specifiers** to control how they interact with various aspects of the engine and editor. Each type of data structure or member has its own list of Metadata Specifiers.

Metadata only exists in the editor; do not write game logic that accesses metadata.

Classes can use the following Metatag Specifiers:

Class Meta Tag	Effect
`BlueprintSpawnableComponent`	If present, the component Class can be spawned by a Blueprint.
`BlueprintThreadSafe`	Only valid on Blueprint function libraries. This specifier marks the functions in this class as callable on non-game threads in animation Blueprints.
`ChildCannotTick`	Used for Actor and Component classes. If the native class cannot tick, Blueprint-generated classes based on this Actor or Component can never tick, even if `bCanBlueprintsTickByDefault` is true.
`ChildCanTick`	Used for Actor and Component classes. If the native class cannot tick, Blueprint-generated classes based on this Actor or Component can have the `bCanEverTick` flag overridden, even if `bCanBlueprintsTickByDefault` is false.
`DeprecatedNode`	For behavior tree nodes, indicates that the class is deprecated and will display a warning when compiled.
`DeprecationMessage="Message Text"`	Deprecated classes with this metadata will include this text with the standard deprecation warning that Blueprint Scripts generate during compilation.
`DisplayName="Blueprint Node Name"`	The name of this node in a Blueprint Script will be replaced with the value provided here, instead of the code-generated name.
`DontUseGenericSpawnObject`	Do not spawn an Object of the class using Generic Create Object node in Blueprint Scripts;

Class Meta Tag	Effect
	this specifier applies only to Blueprint-type classes that are neither Actors nor Actor Components.
`ExposedAsyncProxy`	Expose a proxy Object of this class in Async Task nodes.
`IgnoreCategoryKeywordsInSubclasses`	Used to make the first subclass of a class ignore all inherited `ShowCategories` and `HideCategories` Specifiers.
`IsBlueprintBase="true/false"`	States that this class is (or is not) an acceptable base class for creating Blueprints, similar to the `Blueprintable` or `NotBlueprintable` Specifiers.
`KismetHideOverrides="Event1, Event2, .."`	List of Blueprint events that are not allowed to be overridden.
`ProhibitedInterfaces="Interface1, Interface2, .."`	Lists Interfaces that are not compatible with the class.
`RestrictedToClasses="Class1, Class2, .."`	Blueprint function library classes can use this to restrict usage to the classes named in the list.
`ShortToolTip="Short tooltip"`	A short tooltip that is used in some contexts where the full tooltip might be overwhelming, such as the Parent Class Picker dialog.
`ShowWorldContextPin`	Indicates that Blueprint nodes placed in graphs owned by this class must show their World context pins, even if they are normally hidden, because Objects of this class cannot be used as World context.
`UsesHierarchy`	Indicates the class uses hierarchical data. Used to instantiate hierarchical editing features in Details panels.
`ToolTip="Hand-written tooltip"`	Overrides the automatically generated tooltip from code comments.
`ScriptName="DisplayName"`	The name to use for this clas, property, or function when exporting it to a scripting language. You may include deprecated names as additional semi-colon-separated entries.

Class Implementation

All gameplay classes must use the `GENERATED_BODY` macro in order to be implemented properly. This is done in the class header (.h) file that defines the class and all of its variables and functions. A best practice is for the class source and header files to be named to match the class being implemented, minus the `A` or `U` prefix. So, the source file for the `AActor` class is named `Actor.cpp`, and its header file is named `Actor.h`. This is handled automatically for classes created by the "Add C++ Class" menu option within the editor.

The source file (.cpp) must include the header file (.h) that contains the C++ class declaration, which is usually generated automatically but can also be created manually (if desired). For example, the C++ declaration for the `AActor` class is automatically generated in the `EngineClasses.h` header file. This means the `Actor.cpp` file must include the `EngineClasses.h` file or another file that in turn includes it. Generally, you just include the header file for your game project, which will include the headers for the gameplay classes in your game project. In the case of `AActor` and `EngineClasses.h`, the `EnginePrivate.h` header is included, which includes `Engine.h` - the header file for the **Engine** project - and that includes the `EngineClasses.h` header file.

```
#include "EnginePrivate.h"

```

Copy full snippet

You may also need to include additional files if you reference other classes in the implementation of the class's functions that are not included simply by the inclusion of that one file.

Class Constructor

`UObjects` use **Constructors** to set default values for properties and other variables as well as perform other necessary initialization. The class constructor is generally placed within the class implementation file, e.g. the `AActor::AActor` constructor is in `Actor.cpp`.

Some constructors may be located in a special "constructors" file on a per-module basis. For example, the `AActor::AActor` constructor may be found in `EngineConstructors.cpp`. This is the result of an automatic conversion process from the previous use of a `DEFAULTS` block to the use of constructors. Over time, these will be moved to their respective class source files.

It is also possible to place the constructor inline in the class header file. However, if the constructor is in the class header file, the UClass must be declared with the `CustomConstructor` specifier as this prevents the automatic code generator from creating a constructor declaration in the header.

Constructor Format

The most basic form of a UObject constructor is shown below:

```
UMyObject::UMyObject()
{
// Initialize Class Default Object properties here.
}

```

Copy full snippet

This constructor initializes the Class Default Object (CDO), which is the master copy on which future instances of the class are based. There is also a secondary constructor that supports a special property-altering structure:

```
UMyObject::UMyObject(const FObjectInitializer& ObjectInitializer)
: Super(ObjectInitializer)
{
// Initialize CDO properties here.
}

```

Copy full snippet

Although neither of the above constructors actually perform any initialization, the engine will have already initialized all fields to zero, NULL, or whatever value their default constructors implement. However, any initialization code written into the constructor will be applied to the CDO, and will therefore be copied to any new instances of the object created properly within the engine, as with `CreateNewObject` or `SpawnActor`.

The `FObjectInitializer` parameter that is passed into the constructor, despite being marked as const, can be configured via built-in mutable functions to override properties and subobjects. The `UObject` being created will be affected by these changes, and this can be used to change the values of registered properties or components.

```
AUDKEmitterPool::AUDKEmitterPool(const FObjectInitializer& ObjectInitializer)
:
Super(ObjectInitializer.DoNotCreateDefaultSubobject(TEXT("SomeComponent")).DoN
{
// Initialize CDO properties here.
}

```

Copy full snippet

In the above case, the superclass was going to create the subobjects named "SomeComponent" and "SomeOtherComponent" in its constructor, but it will not do so because of the FObjectInitializer. In the following case, `SomeProperty` will default to 26 in the CDO, and thus in every fresh instance of AUTDemoHUD.

```
AUTDemoHUD::AUTDemoHUD()
```

```
{
// Initialize CDO properties here.
SomeProperty = 26;
}

```

Copy full snippet

Constructor Statics and Helpers

Setting values for more complex data types, especially class references, names, and asset references, requires defining and instantiating a **ConstructorStatics** struct in the constructor to hold the various property values needed. This `ConstructorStatics` struct is only created the first time the constructor is run. On subsequent runs, it just copies a pointer, which makes it extremely fast. When the `ConstructorStatics` struct is created, the values are assigned to the members of the struct for accessing when assigning the values to the actual properties themselves later on in the constructor.

ContructorHelpers is a special namespace defined in `ObjectBase.h` that contains helper templates that are used to perform common actions specific to constructors. For instance, there are helper templates for finding references to assets or classes as well as creating and finding components.

Asset References

Ideally, asset references in classes do not exist. Hardcoded asset references are brittle and the preferred method is to use Blueprints for configuring asset properties. However, hardcoded references are still fully supported. We do not want to search for assets every time we construct an object, so these searches are only done once. This is accomplished via a static struct which ensures that we do our asset searches only once:

`ConstructorHelpers::FObjectFinder` finds a reference to the specified `UObject` using `StaticLoadObject`. This is generally used to reference assets stored in content packages. Reports failure if the object is not found.

```
ATimelineTestActor::ATimelineTestActor()
{
// Structure to hold one-time initialization
struct FConstructorStatics
{
ConstructorHelpers::FObjectFinder<UStaticMesh> Object0;
FConstructorStatics()
: Object0(TEXT("StaticMesh'/Game/UT3/Pickups/Pickups/Health_Large/Mesh/S_Pickup
{
}
};
static FConstructorStatics ConstructorStatics;

// Property initialization

StaticMesh = ConstructorStatics.Object0.Object;
}

```

Copy full snippet

Class References

`ConstructorHelpers::FClassFinder` finds a reference to the specified `UClass` and reports a failure if the class is not found.

```
APylon::APylon(const class FObjectInitializer& ObjectInitializer)
: Super(ObjectInitializer)
{
// Structure to hold one-time initialization
static FClassFinder<UNavigationMeshBase>
ClassFinder(TEXT("class'Engine.NavigationMeshBase'"));
if (ClassFinder.Succeeded())
{
NavMeshClass = ClassFinder.Class;
}
else
{
NavMeshClass = nullptr;
}
}

```

Copy full snippet

In many cases, you can just use `USomeClass::StaticClass()` and skip the complexity of the ClassFinder altogether. For example, you can use the method below in most circumstances:

```
NavMeshClass = UNavigationMeshBase::StaticClass();

```

Copy full snippet

For cross-module references, it is probably better to use the ClassFinder method.

Components and Sub-Objects

Creating component subobjects and attaching them to the actor's hierarchy can also be done inside of the constructor. When spawning an actor, its components will be cloned from the CDO. In order to ensure that components are always created, destroyed, and properly garbage-collected, a pointer to every component created in the constructor should be stored in a UPROPERTY of the owning class.

```cpp
UCLASS()
class AWindPointSource : public AActor
{
GENERATED_BODY()
public:

UPROPERTY()
UWindPointSourceComponent* WindPointSource;

UPROPERTY()
UDrawSphereComponent* DisplaySphere;
};

AWindPointSource::AWindPointSource()
{
// Create a new component and give it a name.
WindPointSource = CreateDefaultSubobject<UWindPointSourceComponent>
(TEXT("WindPointSourceComponent0"));

// Set our new component as the RootComponent of this actor, or attach it to
the root if one already exists.
if (RootComponent == nullptr)
{
RootComponent = WindPointSource;
}
else
{
WindPointSource->AttachTo(RootComponent);
}

// Create a second component. This will be attached to the component we just
created.
DisplaySphere = CreateDefaultSubobject<UDrawSphereComponent>
(TEXT("DrawSphereComponent0"));
DisplaySphere->AttachTo(RootComponent);

// Set some properties on the new component.
DisplaySphere->ShapeColor.R = 173;
DisplaySphere->ShapeColor.G = 239;
DisplaySphere->ShapeColor.B = 231;
DisplaySphere->ShapeColor.A = 255;
DisplaySphere->AlwaysLoadOnClient = false;
DisplaySphere->AlwaysLoadOnServer = false;
DisplaySphere->bAbsoluteScale = true;
}

```

Copy full snippet

Modifying a component belonging to the parent class is generally not necessary. However, a current list of all attached components, including components created by the parent class, is available by calling `GetAttachParent`, `GetParentComponents`, `GetNumChildrenComponents`, `GetChildrenComponents`, and `GetChildComponent` on any `USceneComponent`, including the root component.