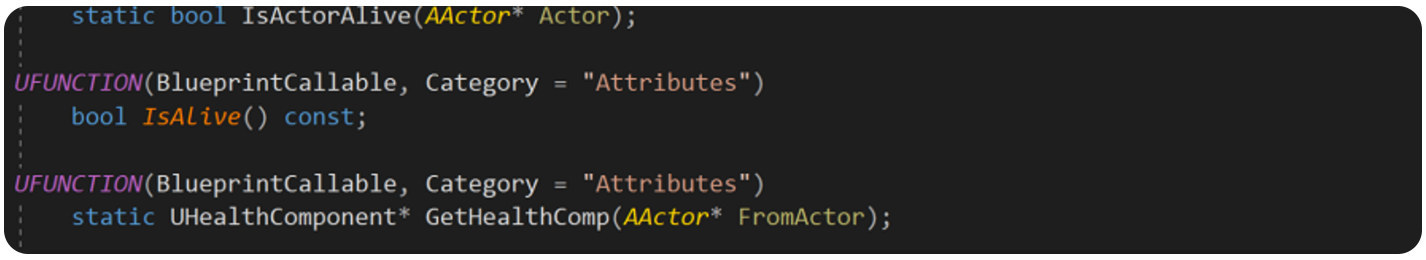
UFunctions

Overview for creating and implementing functions for gameplay Classes



```
    static bool IsActorAlive(AActor* Actor);

UFUNCTION(BlueprintCallable, Category = "Attributes")
    bool IsAlive() const;

UFUNCTION(BlueprintCallable, Category = "Attributes")
    static UHealthComponent* GetHealthComp(AActor* FromActor);
```

UFunction Declaration

A **UFunction** is a C++ function that is recognized by the Unreal Engine reflection system. Any `UObject` or Blueprint function library can declare a member function as a UFunction by placing the `UFUNCTION` macro on the line above the function declaration in the header file. The macro will support **Function Specifiers** to change how Unreal Engine interprets and uses a function.

```
UFUNCTION([specifier1=setting1, specifier2, ...], [meta(key1="value1", key2, ...)])
ReturnType FunctionName([Parameter1, Parameter2, ..., ParameterN1=DefaultValueN1, ParameterN2=DefaultValueN2]) [const];

```

Copy full snippet

With Function Specifiers, you can expose UFunctions to Blueprint Visual Scripting graphs, which provide a way for developers to call or extend UFunctions from Blueprint Assets without having to alter C++ code.

UFunctions are able to bind to Delegates in the default properties of a Class, enabling them to perform such tasks as associating actions with user inputs. They can also act as network callbacks, meaning you can use them to receive a notification and run custom code whenever a certain variable is affected by a network update.

You can even create your own *console commands* (often called *debug*, *configuration*, or *cheat code* commands) that you can call from the game console in development builds, or add buttons with custom functionality to game objects in the Level Editor.

Function Specifiers

When declaring functions, **Function Specifiers** can be added to the declaration to control how the function behaves with various aspects of the engine and the editor.

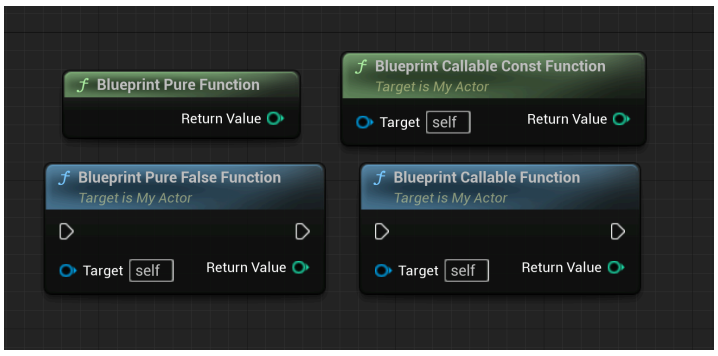
Function Specifier	Effect
`BlueprintAuthorityOnly`	This function will only execute from Blueprint code if running on a machine with network authority (a server, dedicated server, or single-player game).
`BlueprintCallable`	The function can be executed in a Blueprint or Level Blueprint graph.
`BlueprintCosmetic`	This function is cosmetic and will not run on dedicated servers.
`BlueprintImplementableEvent`	The function can be implemented in a Blueprint or Level Blueprint graph.
`BlueprintNativeEvent`	This function is designed to be overridden by a Blueprint, but also has a default native implementation. Declares an additional function named the same as the main function, but with `_Implementation` added to the end, which is where code should be written. The autogenerated code will call the `_Implementation` method if no Blueprint override is found.
`BlueprintPure`	The function does not affect the owning object in any way and can be executed in a Blueprint or Level Blueprint graph. By default functions that are marked `const` will be exposed as pure functions. To make a const function not be a pure function you can declare: `BlueprintPure=false`

```
`UFUNCTION(BlueprintPure)
float  BlueprintPureFunction;

UFUNCTION(BlueprintCallable)
float
BlueprintCallableFunction

UFUNCTION(BlueprintCallable)
```

Function Specifier	Effect

	`int32 BlueprintCallableConstFunction() const`
	
	`UFUNCTION(BlueprintPure=fasle)`
	`Int32 BlueprintPureFalseFunction() const``
	
	Copy full snippet
	 Blueprint Pure Function
	Return Value
	Blueprint Callable Const Function
	Target is My Actor
	Target self
	Return Value
	Blueprint Pure False Function
	Target is My Actor
	Target self
	Return Value
	Blueprint Callable Function
	Target is My Actor
	Target self
	Return Value
	Pure functions do not cache their results, therefore you should be cautious when doing any non-trivial amount of work with a blueprint function. It is good practice to avoid outputting array properties in Blueprint pure functions.

`CallInEditor`	This function can be called in the editor on selected instances via a button in the Details panel.
`Category = "TopCategory|SubCategory|Etc"`	Specifies the category of the function when displayed in Blueprint editing tools. Define nested categories using the | operator.
`Client`	The function is only executed on the client that owns the Object on which the function is called. Declares an additional function named the same as the main function, but with `_Implementation` added to the end. The autogenerated code will call the `_Implementation` method when necessary.
`CustomThunk`	The `UnrealHeaderTool` code generator will not produce a thunk for this function; it is up to the user

Function Specifier	Effect
	to provide one with the `DECLARE_FUNCTION` or `DEFINE_FUNCTION` macros.
`Exec`	The function can be executed from the in-game console. Exec commands only function when declared within certain Classes.
`NetMulticast`	The function is executed both locally on the server, and replicated to all clients, regardless of the Actor's `NetOwner`.
`Reliable`	The function is replicated over the network, and is guaranteed to arrive regardless of bandwidth or network errors. Only valid when used in conjunction with `Client` or `Server`.
`SealedEvent`	This function cannot be overridden in subclasses. The `SealedEvent` keyword can only be used for events. For non-event functions, declare them as `static` or `final` to seal them.
`ServiceRequest`	This function is an RPC (Remote Procedure Call) service request. This implies `NetMulticast` and `Reliable`.
`ServiceResponse`	This function is an RPC service response. This implies `NetMulticast` and `Reliable`.
`Server`	The function is only executed on the server. Declares an additional function named the same as the main function, but with `_Implementation` added to the end, which is where code should be written. The autogenerated code will call the `_Implementation` method when necessary.
`Unreliable`	The function is replicated over the network but can fail due to bandwidth limitations or network errors. Only valid when used in conjunction with `Client` or `Server`.
`WithValidation`	Declares an additional function named the same as the main function, but with `_Validate` added to the end. This function takes the same parameters, and returns a `bool` to indicate whether or not the call to the main function should proceed.

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.

Function Meta Tag	Effect
`AdvancedDisplay="Parameter1, Parameter2, .."`	The comma-separated list of parameters will show up as advanced pins (requiring UI expansion).
`AdvancedDisplay=N`	Replace `N` with a number, and all parameters after the Nth will show up as advanced pins (requiring UI expansion). For example, 'AdvancedDisplay=2' will mark all but the first two parameters as advanced).
`ArrayParm="Parameter1, Parameter2, .."`	Indicates that a `BlueprintCallable` function should use a Call Array Function node and that the listed parameters should be treated as wild card array properties.
`ArrayTypeDependentParams="Parameter"`	When `ArrayParm` is used, this specifier indicates one parameter which will determine the types of all parameters in the `ArrayParm` list.
`AutoCreateRefTerm="Parameter1, Parameter2, .."`	The listed parameters, although passed by reference, will have an automatically created default if their pins are left disconnected. This is a convenience feature for Blueprints, often used on array pins.
`BlueprintAutocast`	Used only by static `BlueprintPure` functions from a Blueprint function library. A cast node will be automatically added for the return type and the type of the first parameter of the function.
`BlueprintInternalUseOnly`	This function is an internal implementation detail, used to implement another function or node. It is never directly exposed in a Blueprint graph.

Function Meta Tag	Effect
`BlueprintProtected`	This function can only be called on the owning Object in a Blueprint. It cannot be called on another instance.
`CallableWithoutWorldContext`	Used for `BlueprintCallable` functions that have a `WorldContext` pin to indicate that the function can be called even if its Class does not implement the `GetWorld` function.
`CommutativeAssociativeBinaryOperator`	Indicates that a `BlueprintCallable` function should use the Commutative Associative Binary node. This node lacks pin names, but features an **Add Pin** button that creates additional input pins.
`CompactNodeTitle="Name"`	Indicates that a `BlueprintCallable` function should display in the compact display mode, and provides the name to display in that mode.
`CustomStructureParam="Parameter1, Parameter2, .."`	The listed parameters are all treated as wildcards. This specifier requires the `UFUNCTION`-level specifier, `CustomThunk`, which will require the user to provide a custom `exec` function. In this function, the parameter types can be checked and the appropriate function calls can be made based on those parameter types. The base `UFUNCTION` should never be called, and should assert or log an error if it is. To declare a custom `exec` function, use the syntax `DECLARE_FUNCTION(execMyFunctionName)` where `MyFunctionName` is the name of the original function.
`DefaultToSelf`	For `BlueprintCallable` functions, this indicates that the Object property's named default value should be the self context of the node.
`DeprecatedFunction`	Any Blueprint references to this function will cause compilation warnings telling the user that the function is deprecated. You can add to the deprecation warning message (for example, to provide instructions on replacing the deprecated function) using the `DeprecationMessage` metadata specifier.

Function Meta Tag	Effect
`DeprecationMessage`="Message Text"	If the function is deprecated, this message will be added to the standard deprecation warning when trying to compile a Blueprint that uses it.
`DeterminesOutputType="Parameter"`	The return type of the function will dynamically change to match the input that is connected to the named parameter pin. The parameter should be a templated type like `TSubClassOf<X>` or `TSoftObjectPtr<X>`, where the function's original return type is `X*` or a container with `X*` as the value type, such as `TArray<X*>`.
`DevelopmentOnly`	Functions marked as `DevelopmentOnly` will only run in Development mode. This is useful for functionality like debug output, which is expected not to exist in shipped products.
`DisplayName="Blueprint Node Name"`	The name of this node in a Blueprint will be replaced with the value provided here, instead of the code-generated name.
`ExpandEnumAsExecs="Parameter"`	For `BlueprintCallable` functions, this indicates that one input execution pin should be created for each entry in the `enum` used by the parameter. The parameter must be of an enumerated type that has the `UENUM` tag.
`ForceAsFunction`	Change a `BlueprintImplementableEvent` with no return value from an event into a function.
`HidePin="Parameter"`	For `BlueprintCallable` functions, this indicates that the parameter pin should be hidden from the user's view. Only one pin per function can be hidden in this manner.
`HideSelfPin`	Hides the "self" pin, which indicates the object on which the function is being called. The "self" pin is automatically hidden on `BlueprintPure` functions that are compatible with the calling Blueprint's Class. Functions that use the `HideSelfPin` Meta Tag frequently also use the `DefaultToSelf` Specifier.
`InternalUseParam="Parameter"`	Similar to `HidePin`, this hides the named parameter's pin from the user's view, and can only be used for one parameter per function.

Function Meta Tag	Effect
`KeyWords="Set Of Keywords"`	Specifies a set of keywords that can be used when searching for this function, such as when placing a node to call the function in a Blueprint Graph.
`Latent`	Indicates a latent action. Latent actions have one parameter of type `FLatentActionInfo`, and this parameter is named by the `LatentInfo` specifier.
`LatentInfo="Parameter"`	For Latent `BlueprintCallable` functions, indicates which parameter is the LatentInfo parameter.
`MaterialParameterCollectionFunction`	For `BlueprintCallable` functions, indicates that the material override node should be used.
`NativeBreakFunc`	For `BlueprintCallable` functions, indicates that the function should be displayed the same way as a standard Break Struct node.
`NotBlueprintThreadSafe`	Only valid in Blueprint function libraries. This function will be treated as an exception to the owning Class's general `BlueprintThreadSafe` metadata.
`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.
`ToolTip="Hand-written tooltip`	Overrides the automatically generated tooltip from code comments.
`UnsafeDuringActorConstruction`	This function is not safe to call during Actor construction.
`WorldContext="Parameter"`	Used by `BlueprintCallable` functions to indicate which parameter determines the World in which the operation takes place.
`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.

Function Parameter Specifiers

Parameter Specifier	Description
Out	Declares the parameter as being passed by reference, allowing it to be modified by the function.
Optional	With the optional keyword, you can make certain function parameters optional, as a convenience to the caller. The values for optional parameters which the caller does not specify depend on the function. For example, the `SpawnActor` function takes an optional location and rotation, which defaults to the location and rotation of the spawning Actor's root component. The default value of optional arguments can be specified by adding `= [value]`. For example: `function myFunc(optional int x = -1)`. In most cases, the default value for the type of variable, or zero (0, false, "", none), is used when no value is passed to an optional parameter.

Delegates

Delegates can call member functions on C++ objects in a generic, type-safe way. A delegate can be bound dynamically to a member function of an arbitrary object, calling the function on the object at a future time, even if the caller does not know the object`s type.

See the Delegates page for reference and usage information.

Timers

Timers schedule actions to be performed after a delay, or over a period of time. For example, you may want to make the player invulnerable after obtaining a power-up item, then restore vulnerability after 10 seconds. Or you may want to apply damage once per second while the player moves through a room filled with toxic gas. Such actions can be achieved through the use of timers.

See the Gameplay Timers page for reference and usage information.