

# Exposing Gameplay Elements to Blueprints

Technical guide for gameplay programmers exposing gameplay elements to Blueprints.



As a programmer, working with Blueprints offers you many benefits by making your code extremely flexible. For example, a gameplay designer might want a new type of weapon implemented in the game. As the programmer, you code the weapon as you traditionally would except that you expose some important functionality, such as the fire rate and the `Fire()` function. After play testing, the designer decides that they need to change the fire rate of the gun to work on a curve. Rather than having to recode the fire rate of the gun and recompile the game, the designer can simply go into the Blueprint and change the fire rate directly, saving both the designer and the programmer time.


For more details about optimizing the C++/Blueprints distribution in your project, or tips to keep in mind when creating an API exposed to Blueprints, see the [Exposing C++ to Blueprints](#).

## Making Classes Blueprintable

In order to create Blueprints that extend from a class, that class must be defined as **Blueprintable**; which involves adding this keyword inside the **UCLASS()** macro preceding the class definition. This keyword makes the Blueprint system aware of the class so that it shows in the class list of the **New Blueprint** dialog and can be selected as the parent for the Blueprint being created.

The simplest form of declaration for a Blueprintable class is similar to the following:

```
1 UCLASS(Blueprintable)
2 class AMyBlueprintableClass : AActor
3 {
4     GENERATED_BODY()
5 }
```

 Copy full snippet

Keyword	Description
<b>Blueprintable</b>	Exposes this class as an acceptable base class for creating Blueprints. The default is NotBlueprintable, unless inherited otherwise. This is inherited by subclasses.
<b>BlueprintType</b>	Exposes this class as a type that can be used for variables in Blueprints.
<b>NotBlueprintable</b>	Specifies that this class is NOT an acceptable base class for creating Blueprints. Negates the effect of a parent class having the Blueprintable keyword specified.

## Readable and Writable Properties

In order to expose a variable defined in a C++ class to a Blueprint extending from that class, that variable must be defined using one of the keywords listed below inside the **UPROPERTY()** macro preceding the variable definition. These keywords make the Blueprint system aware of the variable so that it shows in the **My Blueprint** panel and its value can be set or accessed.

```

1 //Character's Health
2 UPROPERTY(EditAnywhere, BlueprintReadWrite, Category="Character")
3 float health;
4

```

 Copy full snippet

Keyword	Description
BlueprintReadOnly	This property can be read by blueprints, but not modified.
BlueprintReadWrite	This property can be read or written from a Blueprint.
<b>Multicast Delegate Keywords</b>	
BlueprintAssignable	Property should be exposed for assigning in Blueprints.
BlueprintCallable	Property should be exposed for calling in Blueprint graphs.

## Executable and Overridable Functions

In order to call a native function from a Blueprint, that function must be defined using one of the keywords listed below inside the **UFUNCTION()** macro preceding the function definition. These keywords make the Blueprint system aware of the function so that it shows in the context menu or Palette and can be added to a graph and executed - or, in the case of events, so that they can be overridden and executed.

The simplest form of declaration for a BlueprintCallable Function is similar to the following:

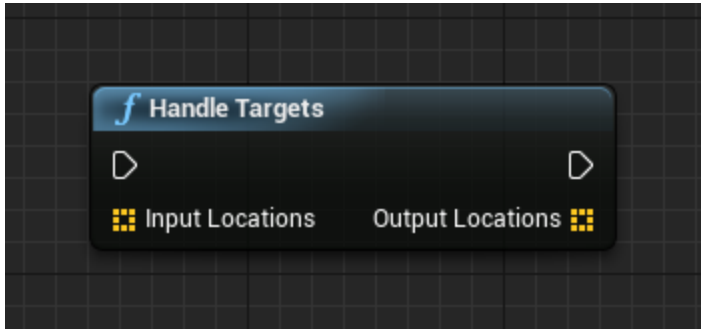
```

1 //Fire a Weapon
2 UFUNCTION(BlueprintCallable, Category="Weapon")
3 void Fire();
4

```

 Copy full snippet

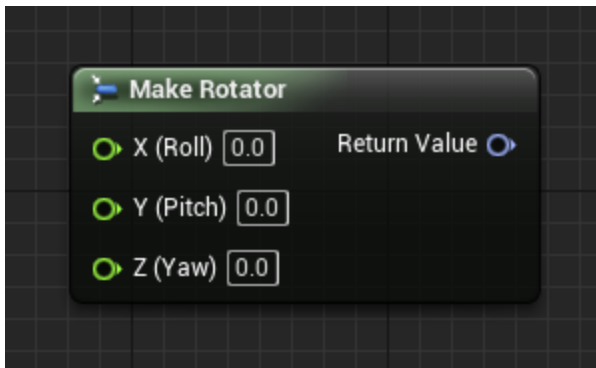
When creating your function's signature, note that making a parameter pass by reference will make it an output pin on the Blueprint node. To make a parameter pass by reference and still display as an input, use the `UPARAM()` macro.



```
1 UFUNCTION(BlueprintCallable, Category = "Example Nodes")
2 static void HandleTargets(UPARAM(ref) TArray<FVector>& InputLocations,
3   TArray<FVector>& OutputLocations);
```

 Copy full snippet

You can also use `UPARAM()` to change the display name of a pin. For example, the `MakeRotator` function in `KismetMathLibrary` uses `UPARAM()` and the `DisplayName` keyword to change how the Roll, Pitch, and Yaw parameters appear in Blueprints.



```
1 /** Makes a rotator {Roll, Pitch, Yaw} from rotation values supplied in
2   degrees */
3 UFUNCTION(BlueprintPure, Category="Math|Rotator", meta=(Keywords="construct
4   build rotation rotate rotator makerotator", NativeMakeFunc))
5 static FRotator MakeRotator(
6   UPARAM(DisplayName="X (Roll)") float Roll,
7   UPARAM(DisplayName="Y (Pitch)") float Pitch,
8   UPARAM(DisplayName="Z (Yaw)") float Yaw);
```

Keyword	Description
<b>Blueprint to Native Communication</b>	
BlueprintCallable	<p>This is a native function that can be called from a Blueprint, which executes native code that changes something about the object it is being called on, or some other global state. This means it has to be "scheduled," or told explicitly the order in which it executes relative to other nodes. We do this by the white execution line. All Blueprint callable functions will be called in the order in which they appear along the white execution line.</p>
BlueprintPure	<p>This is a native function that can be called from a Blueprint, which executes native code that does not change anything about the object it is being called on, or any other global state. This means that nothing is changed by calling this node, it just takes input, and tells you an output. These are things like math nodes (+, -, *, etc), or variable getters, or anything that does not change anything permanently. These do not need to be scheduled, and do not have a connection for the white execution line. They are automatically figured out by the compiler, based on which BlueprintCallable nodes need the data produced by these nodes.</p>
<b>Native to Blueprint Communication</b>	
BlueprintImplementableEvent	<p>This is the primary way we allow native functions to call up into Blueprints. They are like virtual functions that you implement in the Blueprint themselves. If there is no implementation, the function call is ignored. It is important to note that if a BlueprintImplementableEvent does not have a return value or an out parameter, it will appear as an event, available by <b>Right-clicking</b> and selecting it in the event graph of the Blueprint. If it does have a return value or any out parameters, it will be listed in the <b>My Blueprints</b></p>

Keyword	Description
	tab, and can then be overridden by <b>Right-clicking</b> and selecting implement function. Note that BlueprintImplementableEvents do not have a native implementation of the function.
BlueprintNativeEvent	These are the same as above, except there is a native default implementation of the function that is called if the Blueprint does not override the function. This is useful for things where you want some sort of default behavior if the Blueprint does not implement it, but want the Blueprint to be able to override the functionality if desired. These are more costly, so we only put them in where the functionality is needed. When you override the BlueprintNativeEvent, you can still call the native implementation, if desired, by <b>Right-clicking</b> on the event or function entry node, and selecting "Add call to parent".