Unreal Interfaces

Create and implement Unreal Interfaces in C++ and Blueprints.

PREREQUISITE TOPICS

In order to understand and use the content on this page, make sure you are familiar with the following topics:

- Unreal Engine Reflection System

When a class inherits from an **Unreal Interface** class, the interface ensures that the new class implements a common set of functions. This is useful when certain functionality might be shared by large, complex classes that are otherwise dissimilar.

For example, suppose your game has a system where a player character entering a trigger volume can activate traps, alert enemies, or award points to the player depending on the circumstances. You might implement this with a `ReactToTrigger` function on traps, enemies, or point awards. Even though all of these activating objects implement the `ReactToTrigger` function, they might otherwise be quite dissimilar. For example:

- Traps derive from `AActor`.
- Enemies derive from `APawn` or `ACharacter`.
- Point-awards derive from `UDataAsset`.

These classes need shared functionality, but have no common parent other than the base `UObject`. In this case, an Unreal Interface can enforce all of these objects to implement the necessary functions.

Declare an Interface in C++

Declaring an interface in C++ is similar to declaring an ordinary Unreal class. However, there are some primary differences:

- Interface classes use the `UINTERFACE` macro instead of the `UCLASS` macro.

- Interface classes inherit from `UInterface` instead of `UObject`.

The `UINTERFACE` class is not the actual interface, but rather an empty class that provides visibility to the reflection system.

C++ Class Wizard

To create a new Unreal Interface class from the Unreal Editor, follow the steps in the C++ Class Wizard documentation with the following information:

- **Class:** Unreal Interface

C++ Interface Declaration Example

The following is an example of a C++ interface declaration named `ReactToTriggerInterface`:

ReactToTriggerInterface.h

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "ReactToTriggerInterface.generated.h"

/*
This class does not need to be modified.
Empty class for reflection system visibility.
Uses the UINTERFACE macro.
Inherits from UInterface.
*/
UINTERFACE(MinimalAPI, Blueprintable)
class UReactToTriggerInterface : public UInterface
{
GENERATED_BODY()
};

/* Actual Interface declaration. */
class IReactToTriggerInterface
{
GENERATED_BODY()

// Add interface functions to this class. This is the class that will be inherited to implement this interface.
public:
// Add interface function declarations here
};
```

Copy full snippet

As you can see in this example, the actual interface has the same name as the empty class, but the `U`-prefix is replaced by `I`. The `U`-prefixed class needs no constructor or any other functions. The `I`-prefixed class contains all interface functions and is the class that is inherited from by classes intended to implement the interface.

The `Blueprintable` specifier is required if you want a Blueprint to implement this interface.

Interface Specifiers

Use interface specifiers to expose your class to the Unreal Reflection System. The following table contains relevant interface specifiers:

Interface Specifier	Description
`Blueprintable`	Exposes this interface so it can be implemented by a Blueprint. Interfaces can't be exposed to Blueprint if they contain anything other than `BlueprintImplementableEvent` and `BlueprintNativeEvent` functions. Use `NotBlueprintable` or `meta=(CannotImplementInterfaceInBlueprint)` to specify that an interface is not safe to implement in a Blueprint.
`BlueprintType`	Exposes this class as a type that can be used for variables in Blueprints.
`DependsOn=(ClassName1, ClassName2, ...)`	The build system compiles all classes listed with this specifier before it compiles this class. `ClassName` must specify a class in the same (or a previous) package. You can specify multiple dependency classes using a single `DependsOn` line delimited by commas, or can be specified using a separate `DependsOn` line for each class.
`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 avoiding exporting everything for classes that do not need all of their functions accessible in other modules.

Implement an Interface in C++

To use your interface in a new class:

- Include your interface header file.
- Inherit from your `I`-prefixed interface class.

The following is an example of the trap mentioned in the introduction to this page:

Trap.h

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ReactToTriggerInterface.h"
#include "Trap.generated.h"

UCLASS(Blueprintable, Category="MyGame")
class ATrap : public AActor, public IReactToTriggerInterface
{
GENERATED_BODY()

public:
// Add interface function overrides here
};
```

Copy full snippet

Declare Interface Functions

There are several methods you can use to declare functions in your interfaces, each of which is implementable or callable in different contexts. All of them must be declared in the `I`-prefixed class for your interface, and they must be public in order to be visible to external classes.

C++ Only Interface Functions

You can declare a virtual C++ function in your interface's header file, with no `UFUNCTION` specifiers. These functions must be virtual so you can override them in classes that implement your interface.

Interface Class

The following is an example of what this would look like in the `ReactToTriggerInterface` class:

ReactToTriggerInterface.h

```
#pragma once

#include "ReactToTriggerInterface.generated.h"

/*
```

```
Empty class for reflection system visibility.
Uses the UINTERFACE macro.
Inherits from UInterface.
*/
UINTERFACE(MinimalAPI, Blueprintable)
class UReactToTriggerInterface : public UInterface
{
GENERATED_BODY()
};

/* Actual Interface declaration. */
class IReactToTriggerInterface
{
GENERATED_BODY()

public:
virtual bool ReactToTrigger();
};
```

Copy full snippet

You can provide a default implementation either within the header itself or within the interface’s `.cpp` file.

ReactToTriggerInterface.cpp

```
#include "ReactToTriggerInterface.h"

bool IReactToTriggerInterface::ReactToTrigger()
{
return false;
}
```

Copy full snippet

Derived Class

When you implement your interface in a derived class, you can create and implement an override specific to that class. The following is an example of what this would look like if the `ATrap` actor implemented the `IReactToTriggerInterface`:

Trap.h

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ReactToTriggerInterface.h"
#include "Trap.generated.h"

UCLASS(Blueprintable, Category="MyGame")
class ATrap : public AActor, public IReactToTriggerInterface
{
GENERATED_BODY()

public:
virtual bool ReactToTrigger() override;
};
```

Copy full snippet

Trap.cpp

```
#include "Trap.h"

bool ATrap::ReactToTrigger()
{
return false;
}
```

Copy full snippet

C++ interface functions declared in this way are not visible to Blueprint and cannot be used in Blueprintable interfaces.

Blueprint Callable Interface Functions

To make a Blueprint callable interface function, you must do the following:

- Specify a `UFUNCTION` macro in the function's declaration with the `BlueprintCallable` specifier.
- Use either the `BlueprintImplementableEvent` or `BlueprintNativeEvent` specifiers.

Blueprint callable interface functions cannot be virtual.

Functions with the `BlueprintCallable` specifier can be called in C++ or Blueprint using a reference to an object that implements your interface.

If your Blueprint callable function does not return a value, Unreal Engine treats your function as an event in Blueprint.

Blueprint Implementable Event

Functions with the `BlueprintImplementableEvent` specifier can not be overridden in C++, but can be overridden in any Blueprint class that implements or inherits your interface. The following is an example of a C++ interface declaration for a `BlueprintImplementableEvent`:

ReactToTriggerInterface.h

```
#pragma once

```

```
#include "ReactToTriggerInterface.generated.h"

/*
Empty class for reflection system visibility.
Uses the UINTERFACE macro.
Inherits from UInterface.
*/
UINTERFACE(MinimalAPI, Blueprintable)
class UReactToTriggerInterface : public UInterface
{
GENERATED_BODY()
};

/* Actual Interface declaration. */
class IReactToTriggerInterface
{
GENERATED_BODY()

public:
/* A version of the React To Trigger function that can only be implemented
in Blueprint. */
UFUNCTION(BlueprintCallable, BlueprintImplementableEvent, Category=Trigger
Reaction)
bool ReactToTrigger();
};
```

Copy full snippet

Blueprint Native Event

Functions with the `BlueprintNativeEvent` specifier can be implemented in C++ or Blueprint. The following is an example of a C++ interface declaration for a `BlueprintNativeEvent`:

ReactToTriggerInterface.h

```
#pragma once

#include "ReactToTriggerInterface.generated.h"

/*
Empty class for reflection system visibility.
Uses the UINTERFACE macro.
Inherits from UInterface.
*/
UINTERFACE(MinimalAPI, Blueprintable)
class UReactToTriggerInterface : public UInterface
{
GENERATED_BODY()
};

/* Actual Interface declaration. */
class IReactToTriggerInterface
{
```

```
GENERATED_BODY()

public:
/* A version of the React To Trigger function that can be implemented in C++
or Blueprint. */
UFUNCTION(BlueprintCallable, BlueprintNativeEvent, Category=Trigger
Reaction)
bool ReactToTrigger();
};
```

Copy full snippet

Override a Blueprint Native Event in C++

To implement a `BlueprintNativeEvent` in C++, create an additional function with the same name as the `BlueprintNativeEvent` with an additional `_Implementation` suffix appended to the name. The following is an example of what this looks like from the `ATrap` example:

Trap.h

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "ReactToTriggerInterface.h"
#include "Trap.generated.h"

UCLASS(Blueprintable, Category="MyGame")
class ATrap : public AActor, public IReactToTriggerInterface
{
GENERATED_BODY()

public:
virtual bool ReactToTrigger() override;

// Blueprint Native Event override
bool ReactToTrigger_Implementation() override;
};
```

Copy full snippet

Trap.cpp

```
#include "Trap.h"

bool ATrap::ReactToTrigger()
{
return false;
}

// Blueprint Native Event override implementation
bool ATrap::ReactToTrigger_Implementation()
{
return false;
}
```

Copy full snippet

Override a Blueprint Native Event in Blueprint

The `BlueprintNativeEvent` specifier also allows implementations to be overridden in Blueprint. To implement a `BlueprintNativeEvent` in Blueprint, see the Implement Blueprint Interfaces documentation for more information.

Call a Blueprint Event from C++

To safely call a `BlueprintImplementableEvent` or `BlueprintNativeEvent` on a `Blueprintable` interface from C++, you must use the special static `Execute_` function wrapper. The following example call works for interfaces whether they are implemented in C++ or Blueprint:

```
// OriginalObject is an object that implements the IReactToTriggerInterface
bool bReacted =
IReactToTriggerInterface::Execute_ReactToTrigger(OriginalObject);
```

Copy full snippet

Interface Function Types

There are three different types of interface functions:

- Base
- Implementation
- Execute

The following table explains what each type is used for:

Type	Definition Location	Purpose	Use this to…
Base function	Base interface class. (`MyInterface.h`)	Function definition you can implement in child classes.	Only use if interface and implementations are only defined in C++.
Implementation wrapper	C++ Class that implements interface. (`MyInterfaceActor.h`, `MyInterfaceActor.cpp`)	Implement interface functionality in C++.	Call only C++ implementation, but not any Blueprint overrides.
Execute wrapper	Created automatically by Unreal Engine's	Facilitates communication	Call function implementations

Type	Definition Location	Purpose	Use this to...
	reflection system. (`MyInterface.generated.h`, `MyInterface.gen.cpp`)	between implementations defined in C++ and Blueprint.	including C++ and Blueprint overrides.

Consider the following example:

- `MyFunction` is a `BlueprintNativeEvent` interface function defined in `MyInterface.h`.
- `MyInterfaceActor` implements `MyInterface`.
- `MyFunction_Implementation` is defined in `MyInterfaceActor.cpp`.
- A variety of C++ and Blueprint spawned actors that inherit from `MyInterfaceActor`.

To safely call `MyFunction` for all Blueprint and C++ objects that inherit from `MyInterfaceActor`, you can do the following:

```
TArray<AActor*> OutActors;
UGameplayStatics::GetAllActorsOfClass(GetWorld(),
AMyInterfaceActor::StaticClass(), OutActors);

// OutActors contains all BP and C++ actors that are or inherit from
AMyInterfaceActor
for (AActor* CurrentActor : OutActors)
{
// Each CurrentActor calls its own MyFunction implementation
UE_LOG(LogTemp, Log, TEXT("%s : %s"), *CurrentActor->GetName(),
*IMyInterface::Execute_MyFunction(Cast<AMyInterfaceActor>(CurrentActor)));
}
```

Copy full snippet

Determine Whether a Class Implements an Interface

For compatibility with both C++ and Blueprint classes that implement your interface, use any of the following functions to determine whether a class implements an interface:

```
bool bIsImplemented;

/* bIsImplemented is true if OriginalObject implements
UReactToTriggerInterface */
bIsImplemented = OriginalObject->GetClass()-
>ImplementsInterface(UReactToTriggerInterface::StaticClass());

/* bIsImplemented is true if OriginalObject implements
UReactToTriggerInterface */
bIsImplemented = OriginalObject->Implements<UReactToTriggerInterface>();

```

```
/* ReactingObject is non-null if OriginalObject implements
UReactToTriggerInterface in C++ */
IReactToTriggerInterface* ReactingObject = Cast<IReactToTriggerInterface>
(OriginalObject);
```

Copy full snippet

The templated `Cast<>` method only works if the interface is implemented in a C++ class. Interfaces implemented in a Blueprint do not exist in the C++ version of the object, so `Cast<>` returns null. `TScriptInterface<>` can also be used in C++ code to safely copy an interface pointer and the `UObject` that implements it.

Cast to Other Unreal Types

Unreal Engine's casting system supports casting from one interface to another, or from an interface to an Unreal type where appropriate. The following examples are some methods that you can use to cast interfaces:

```
/* ReactingObject is non-null if the interface is implemented */
IReactToTriggerInterface* ReactingObject = Cast<IReactToTriggerInterface>
(OriginalObject);

/* DifferentInterface is non-null if ReactingObject is non-null and it
implements ISomeOtherInterface */
ISomeOtherInterface* DifferentInterface = Cast<ISomeOtherInterface>
(ReactingObject);

/* ReactingActor is non-null if ReactingObject is non-null and OriginalObject
is an AActor or AActor-derived class */
AActor* ReactingActor = Cast<AActor>(ReactingObject);
```

Copy full snippet

Safely Store Object and Interface Pointers

To store a reference to an object that implements a specific interface, you can use `TScriptInterface`. If you have an object that implements an interface, you can initialize `TScriptInterface` as follows:

```
UMyObject* MyObjectPtr;
TScriptInterface<IMyInterface> MyScriptInterface;

if (MyObjectPtr->Implements<UMyInterface>())
{
MyScriptInterface = TScriptInterface<IMyInterface>(MyObjectPtr);
}

// MyScriptInterface holds a reference to MyObjectPtr and MyInterfacePtr
```

Copy full snippet

To retrieve a pointer to the original object, use `GetObject`:

```
UMyObject* MyRetrievedObjectPtr = MyScriptInterface.GetObject();
```

Copy full snippet

To retrieve a pointer to the interface that the original object implements, use `GetInterface`:

```
IMyInterface* MyRetrievedInterfacePtr = MyScriptInterface.GetInterface();
```

Copy full snippet

For more information about `TScriptInterface`, see `TScriptInterface` and the linked `FScriptInterface` API pages.

Blueprint Implementable Interfaces

If you want a Blueprint to implement this interface, you must use the `Blueprintable` metadata specifier. Every interface function (other than static functions) must be a `BlueprintNativeEvent` or a `BlueprintImplementableEvent`. When a Blueprint implements an interface declared in C++, it works like a Blueprint interface asset. This means that an instance of that Blueprint class will not actually contain the C++ version of the interface, so it cannot be used with `Cast<>`. From C++, only the `Execute_` static wrapper functions will work properly.