

Delegates

Data types that reference and execute member functions on C++ Objects



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. Delegates are safe to copy. You can also pass them by value, but this is not recommended because this process will allocate memory on the heap; whenever possible, pass delegates by reference. There are three types of delegates supported by the Engine:

- Single
- [Multicast](#)
- [Dynamic \(UObject, serializable\)](#)

Declaring Delegates

To declare a delegate, use one of the macros below; select the macro based on the signature of the function (or functions) that you intend to bind to the delegate. Each macro features parameters for the new delegate type name, as well as the function's return type (if not `void`) and its parameters. Currently, delegate signatures support any combination of the following:

- Functions returning a value

- Functions declared as `const`
- Up to four "payload" variables
- Up to eight function parameters

Use this table to find the declaration macro to use to declare your delegate.

Function signature	Declaration macro
<code>void Function()</code>	<code>DECLARE_DELEGATE(DelegateName)</code>
<code>void Function(Param1)</code>	<code>DECLARE_DELEGATE_OneParam(DelegateName, Param1Type)</code>
<code>void Function(Param1, Param2)</code>	<code>DECLARE_DELEGATE_TwoParams(DelegateName, Param1Type, Param2Type)</code>
<code>void Function(Param1, Param2, ...)</code>	<code>DECLARE_DELEGATE_<Num>Params(DelegateName, Param1Type, Param2Type, ...)</code>
<code><RetValType> Function()</code>	<code>DECLARE_DELEGATE_RetVal(RetValType, DelegateName)</code>
<code><RetValType> Function(Param1)</code>	<code>DECLARE_DELEGATE_RetVal_OneParam(RetValType, DelegateName, Param1Type)</code>
<code><RetValType> Function(Param1, Param2)</code>	<code>DECLARE_DELEGATE_RetVal_TwoParams(RetValType, DelegateName, Param1Type, Param2Type)</code>
<code><RetValType> Function(Param1, Param2, ...)</code>	<code>DECLARE_DELEGATE_RetVal_<Num>Params(RetValType, DelegateName, Param1Type, Param2Type, ...)</code>

Delegate functions support the same [Specifiers](#) as [UFunctions](#), but use the `UDELEGATE` macro instead of `UFUNCTION`. For example, the following code adds the `BlueprintAuthorityOnly` Specifier to the `FInstigatedAnyDamageSignature` delegate

```
1 UDELEGATE(BlueprintAuthorityOnly)
```



```
2 DECLARE_DYNAMIC_MULTICAST_DELEGATE_FourParams(FInstigatedAnyDamageSignature,  
float, Damage, const UDamageType*, DamageType, AActor*, DamagedActor,  
AActor*, DamageCauser);  
3
```

Copy full snippet

Variations of the macros above for multi-cast, dynamic, and wrapped delegates are as follows:

- DECLARE_MULTICAST_DELEGATE...
- DECLARE_DYNAMIC_DELEGATE...
- DECLARE_DYNAMIC_MULTICAST_DELEGATE...
- DECLARE_DYNAMIC_DELEGATE...
- DECLARE_DYNAMIC_MULTICAST_DELEGATE...

Delegate signature declarations can exist at global scope, within a namespace, or even within a class declaration. These declarations may not exist within the body of a function.

See [Dynamic Delegates](#) and [Multi-cast Delegates](#) for more information on declaring these types of delegates.

Delegate functions support The same Specifiers as [UFunctions](#), but use the `UDELEGATE` macro instead of `UFUNCTION`.

Binding Delegates

The delegate system understands certain types of objects, and additional features are enabled when using these objects. If you bind a delegate to a member of a UObject or shared pointer class, the delegate system can keep a weak reference to the object, so that if the object gets destroyed out from underneath the delegate, you will be able to handle these cases by calling `IsBound` or `ExecuteIfBound` functions. Note the special binding syntax for the various types of supported objects.

Function	Description
<code>Bind</code>	Binds to an existing delegate object.

Function	Description
<code>BindStatic</code>	Binds a raw C++ pointer global function delegate.
<code>BindRaw</code>	Binds a raw C++ pointer delegate. Since raw pointers do not use any sort of reference, calling <code>Execute</code> or <code>ExecuteIfBound</code> after deleting the target object is unsafe.
<code>BindLambda</code>	Binds a functor. This is generally used for lambda functions.
<code>BindSP</code>	Binds a shared pointer-based member function delegate. Shared pointer delegates keep a weak reference to your object. You can use <code>ExecuteIfBound</code> to call them.
<code>BindUObject</code>	Binds a <code>UObject</code> member function delegate. <code>UObject</code> delegates keep a weak reference to the <code>UObject</code> you target. You can use <code>ExecuteIfBound</code> to call them.
<code>UnBind</code>	Unbinds this delegate.

See `DelegateSignatureImpl.inl` (located in `..\Engine\Source\Runtime\Core\Public\Templates\`) for the variations, arguments, and implementations of these functions.

Payload Data

When binding to a delegate, you can pass payload data along. These are arbitrary variables that will be passed directly to any bound function when it is invoked. This is really useful as it allows you to store parameters within the delegate itself at bind-time. All delegate types (except for "dynamic") support payload variables automatically. This example passes two custom variables, a bool and an int32 to a delegate. Then when the delegate is invoked, these parameters will be passed to your bound function. The extra variable arguments must always be accepted after the delegate type parameter arguments.

```
1 MyDelegate.BindRaw( &MyFunction, true, 20 );
2
```

Executing Delegates

The function bound to a delegate is executed by calling the delegate's `Execute()` function. You must check if delegates are "bound" before executing them. This is to make the code more safe as there may be cases where delegates have return values and output parameters that are uninitialized and subsequently accessed. Executing an unbound delegate could actually scribble over memory in some instances. You can call `IsBound()` to check if the delegate is safe to execute. Also, for delegates that have no return value, you can call `ExecuteIfBound()`, but be wary of output parameters that may be left uninitialized.

Execution Functions	Description
<code>Execute</code>	Executes a delegate without checking its bindings
<code>ExecuteIfBound</code>	Checks that a delegate is bound, and if so, calls <code>Execute</code>
<code>IsBound</code>	Checks whether or not a delegate is bound, often before code that includes an <code>Execute</code> call

See [Multi-cast Delegates](#) for details on executing multi-cast delegates.

Example Usage

Suppose you have a class with a method that you would like to be able to call from anywhere:

```
1 class FLogWriter
2 {
3     void WriteToLog(FString);
4 };
5
```

To call the `WriteToLog` function, we will need to create a delegate type for that function's signature. To do this, you will first declare the delegate using one of the macros below. For example, here is a simple delegate type:

```
1 DECLARE_DELEGATE_OneParam(FStringDelegate, FString);
2
```

 Copy full snippet

This creates a delegate type called `FStringDelegate` that takes a single parameter of type `FString`.

Here is an example of how you would use this `FStringDelegate` in a class:

```
1 class FMyClass
2 {
3     FStringDelegate WriteToLogDelegate;
4 };
5
```

 Copy full snippet

This allows your class to hold a pointer to a method in an arbitrary class. The only thing the class really knows about this delegate is its function signature.

Now, to assign the delegate, simply create an instance of your delegate class, passing along the class that owns the method as a template parameter. You will also pass the instance of your object and the actual function address of the method. So, here we will create an instance of our `FLogWriter` class, then create a delegate for the `WriteToLog` method of that object instance:

```
1 TSharedRef<FLogWriter> LogWriter(new FLogWriter());
2
3 WriteToLogDelegate.BindSP(LogWriter, &FLogWriter::WriteToLog);
4
```

 Copy full snippet

You have just dynamically bound a delegate to a method of a class! Pretty simple, right?

Note that the SP part of `BindSP` stands for shared pointer, because we are binding to an object that is owned by a shared pointer. There are versions for different object types, such as `BindRaw()` and `BindUObject()`.


Now, your `WriteToLog` method can be called by `FMyClass` without it even knowing anything about the `FLogWriter` class! To call your delegate, just use the `Execute()` method:

```
1 WriteToLogDelegate.Execute(TEXT("Delegates are great!"));  
2
```

 Copy full snippet

If you call `Execute()` before binding a function to the delegate, an assertion will be triggered. In many cases, you will instead want to do this:

```
WriteToLogDelegate.ExecuteIfBound(TEXT("Only executes if a function was bound!"))
```

 Copy full snippet