Core Redirects

Core Redirects enable remapping classes, enums, functions, packages, properties, and structs at load time.

During development, there are occasions when an existing class, property, function name, or similar code member needs to be renamed. If there are a high number of Assets affected by these changes, though, simply renaming the code member and recompiling your project will cause considerable data loss, because **Unreal Engine (UE)** will no longer recognize existing Assets. To address this issue, the Engine uses **Core Redirects**. Core Redirects should be configured in your project's `DefaultEngine.ini` file, or, in the case of a Plugin, the prefixed, self-named `.ini` file for that Plugin (for example, `BasePaper2D.ini` for the Engine's Paper2D Plugin, or `Default<GamePluginName>.ini` for a game Plugin). In either case, the Core Redirects will be placed in the "[CoreRedirects]" section. These Core Redirects will automatically remap obsolete data while loading Assets, thus preventing data loss resulting from the renaming process. For examples of Core Redirects currently in effect, check the `BaseEngine.ini` file.

Supported Core Redirect Types

When specifying a name of a class or struct in an Core Redirect, the name should be written as it appears to the Unreal Engine's reflection system, meaning the prefix letter is dropped.

For example, `AMyActor` would be written as `MyActor`, and `FMyStruct` would be written as `MyStruct`. Since Unreal Engine's reflection system does not use a prefix for enumerated types, enumerated type names appear in Core Redirects exactly as they do in code. For example, `ESampleEnum` would remain `ESampleEnum` when referenced by a Core Redirect.

The following Core Redirect formats are currently supported:

- `ClassRedirects` - Changes objects and properties using an obsolete (or removed) UCLASS to refer to a new UCLASS.

Field	Type	Purpose
`OldName`	String	Specifies the name of the obsolete (or removed) UCLASS.
`NewName`	String	Specifies the name of the new UCLASS.
`MatchSubstring`	Bool	(Optional) If present and set to `true`, this Core Redirect will apply to any class containing the value of `OldName`, rather than requiring an exact match.
`OverrideClassName`	String	(Optional) Specifies a change to the underlying class of the UCLASS. This is generally used to change a Blueprint class to a native class (`/Script/CoreUObject.Class`).

Field	Type	Purpose
`InstanceOnly`	Bool	(Optional) If present and set to `true`, indicates that the original class still exists and can be referenced, but any existing instances of the old class (such as Actors or Components placed in Levels) should be remapped to the new class. This is especially useful when your project has a specialized version of a class that exists in the engine, but your Levels are full of instances of the original class, and you want to change them all to the project-specific version.
`ValueChanges`	List of String pairs	(Optional) Renames instances of the old class that match the first String of a pair. The new name will be the second String of that pair.

```
[CoreRedirects]
+ClassRedirects=(OldName="Pawn",NewName="MyPawn",InstanceOnly=true)
+ClassRedirects=(OldName="/Script/MyModule.MyOldClass",NewName="/Script/My
+ClassRedirects=(OldName="PointLightComponent",NewName="PointLightComponen
+ClassRedirects=
(OldName="AnimNotify_PlayParticleEffect_C",NewName="/Script/Engine.AnimNot

```

Copy full snippet

- `EnumRedirects` - Remaps obsolete UENUM types and/or obsolete values within an enumerated type.

Field	Type	Purpose
`OldName`	String	Specifies the name of the obsolete UENUM (if `NewName` is specified), or the name of the existing UENUM (if only remapping values).

Field	Type	Purpose
`NewName`	String	(Optional) Specifies the name of the new UENUM, if remapping from an obsolete UENUM to a new one.
`MatchSubstring`	Bool	(Optional) If present and set to `true`, this Core Redirect will apply to any enumerated type containing the value of `OldName`, rather than requiring an exact match.
`ValueChanges`	List of String pairs	The first string in the pair is the old enumerated value, and the second string is the new value. If both values are in the same class, the old value must no longer exist in code.

```
[CoreRedirects]
+EnumRedirects=(OldName="ENumbers",NewName="ELetters",ValueChanges=
(("NumberTwo","LetterB"),("NumberThree","LetterC")))

```

Copy full snippet

- `FunctionRedirects` - Remaps an obsolete UFUNCTION to a new one.

Field	Type	Purpose
`OldName`	String	Specifies the name of the obsolete (or removed) UFUNCTION. The function name is period-delimited so that the class name is included.
`NewName`	String	Specifies the name of the new UFUNCTION. The function name can be period-delimited, which makes it possible to remap a function from one class to another class.
`MatchSubstring`	Bool	(Optional) If present and set to `true`, this Core Redirect will apply to any function containing the

Field	Type	Purpose
		value of `OldName`, rather than requiring an exact match.

```
[CoreRedirects]
+FunctionRedirects=
(OldName="MyOldActor.OldFunction",NewName="MyNewActor.NewFunction")
+FunctionRedirects=(OldName="MyActor.OldFunction",NewName="NewFunction")

```

Copy full snippet

- `PackageRedirects` - Remapping from one package to another, or suppressing warnings about references to a deleted package (references will be cleared or set to null).

Field	Type	Purpose
`OldName`	String	Specifies the name of the obsolete or deleted package.
`NewName`	String	(Optional) If remapping is desired, this specifies the name of the package that replaces the obsolete or deleted package. If this is not present, `Removed` should be present and set to `true`.
`MatchSubstring`	Bool	(Optional) If present and set to `true`, this Core Redirect will apply to any package containing the value of `OldName`, rather than requiring an exact match.
`Removed`	Bool	(Optional) If present and set to `true`, the named package has been removed. References to any of the removed content will be set to null without generating warnings or errors. The `NewName` argument should not be present if this is the case.

```
[CoreRedirects]
```

```
+PackageRedirects=
(OldName="OldPlugin",NewName="/NewPlugin/",MatchSubstring=true)
+PackageRedirects=(OldName="/Game/DeletedContentPackage",Removed=true)

```

Copy full snippet

- `PropertyRedirects` - Remaps removed properties to new properties.

Field	Type	Purpose
`OldName`	String	The name of the removed property. This name is period-delimited so that the class name and any sub-variable names are included, For example, `MyActor.MyStruct.MyProperty`.
`NewName`	String	The name of the new property. This name can be fully period-delimited like `OldName`, or it can be just the variable name if it exists in the same namespace as `OldName`.
`MatchSubstring`	Bool	(Optional) If present and set to `true`, this Core Redirect will apply to any property containing the value of `OldName`, rather than requiring an exact match.

```
[CoreRedirects]
+PropertyRedirects=
(OldName="MyOldActor.OldIntProperty",NewName="MyNewActor.NewIntProperty")
+PropertyRedirects=
(OldName="MyActor.OldFloatProperty",NewName="NewFloatProperty")

```

Copy full snippet

- `StructRedirects` - Changes properties using an obsolete (or removed) USTRUCT to refer to a new USTRUCT.

Field	Type	Purpose
`OldName`	String	Specifies the name of the obsolete (or removed) USTRUCT.
`NewName`	String	Specifies the name of the new USTRUCT.
`MatchSubstring`	Bool	(Optional) If present and set to `true`, this Core Redirect will apply to any struct containing the value of `OldName`, rather than requiring an exact match.

```
[CoreRedirects]
+StructRedirects=(OldName="MyStruct",NewName="MyNewStruct")

```

Copy full snippet

Name Flexibility and Specificity

Names used when describing classes, structs, properties, and functions can be written with varying degrees of specificity. Additionally, the Core Redirects system will use as much (or as little) information as you provide. The following table provides some examples of these levels of specificity.

Example Format	Scope of application
`/Script/MyModule.MyActor.MyFunctionOrProperty`	Will apply only to the function or property called `MyFunctionOrProperty` within the `MyActor` class in the `MyModule` module.
`MyActor.MyFunctionOrProperty`	Will apply to any function or property called `MyFunctionOrProperty` within the `MyActor` class, regardless of what module that class and function exist in.

Example Format	Scope of application
`MyFunctionOrProperty`	Will apply to any function or property called `MyFunctionOrProperty` in any class, within any module.

Some obsolete Core Redirects can be found in certain `.ini` files in games and samples older than version 4.16. While the formats they use are still supported for backward-compatibility purposes, it is not recommended that they be used as templates for writing your own Core Redirects going forward. Instead, use only the formats specified on this page.

Substring Matching

The `MatchSubstring` argument can be used in any Core Redirect type. If present and set to `true`, the `OldName` and `NewName` fields will be treated as substrings rather than requiring exact matches. This enables multiple matches with a single Core Redirect. In the following example, we will start with a struct and a class.

Original code and values:

```
USTRUCT()
struct FMyStruct
{
GENERATED_BODY()

UPROPERTY(EditAnywhere, Category = "Documentation")
int32 TestInt;

UPROPERTY(EditAnywhere, Category = "Documentation")
int32 TestIntFromStruct;
};

UCLASS()
class REDIRECTORSTEST_API AMyActor : public AActor
{
GENERATED_BODY()
```

```
17
18 public:
19 UPROPERTY(EditAnywhere, Category = "Documentation")
20 int32 TestInt;
21
22 UPROPERTY(EditAnywhere, Category = "Documentation")
23 int32 MainClassTestInt;
24
25 UPROPERTY(EditAnywhere, Category = "Documentation")
26 FMyStruct TestStruct;
27 };
```

Copy full snippet

This is the original code and the original set of values we're saving into our AMyActor Asset.

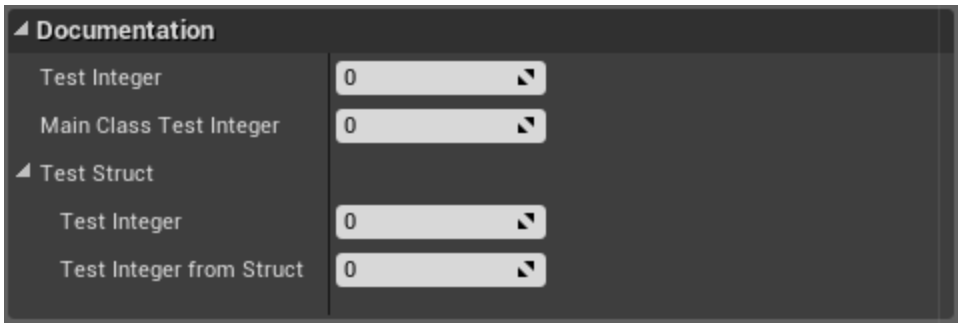
After creating and saving an `AMyActor` Asset with the values shown above, we can close the Editor and alter the the code in the `.h` file and the Core Redirects in the game's `.ini` file. We will change the code to read as follows, changing the names of our `int32` properties:

```
1  USTRUCT()
2  struct FMyStruct
3  {
4  GENERATED_BODY()
5
6  UPROPERTY(EditAnywhere, Category = "Documentation")
7  int32 TestInteger;
8
9  UPROPERTY(EditAnywhere, Category = "Documentation")
10 int32 TestIntegerFromStruct;
11 };
12
13 UCLASS()
14 class REDIRECTORSTEST_API AMyActor : public AActor
```

```
{
GENERATED_BODY()

public:
UPROPERTY(EditAnywhere, Category = "Documentation")
int32 TestInteger;

UPROPERTY(EditAnywhere, Category = "Documentation")
int32 MainClassTestInteger;

UPROPERTY(EditAnywhere, Category = "Documentation")
FMyStruct TestStruct;
};
```

Copy full snippet

With this change, we can examine the effects of a Core Redirect, and in particular the impact of `MatchSubstring`. Results follow:



The properties were renamed in code, but no Core Redirect was created. As a result, no data values have migrated to the new properties.

PropertyRedirects=(OldName="TestInt",NewName="TestInteger") causes only the two preoperties with exact name matches to migrate their data.

PropertyRedirects=(OldName="TestInt",NewName="TestInteger",MatchSubstring=true) causes all four of our properties to migrate, due to substring matching.

> Because `MatchSubtring` requires checking incoming Assets much more thoroughly, it can impact startup times. `MatchSubstring` is intended to be used temporarily as a fixup when making sweeping changes. It is recommended that Assets involved in these changes be resaved immediately and checked into your project's source control database with any related code changes, and that the Core Redirect be deleted without entering source control.

Debug Core Redirects

You can use the `-DebugCoreRedirects` command-line argument to help you debug core redirect issues. This command-line argument adds additional information to UE logs to help identify core redirect issues, including typos.