Unreal Engine 4 C++ Cheat Sheet XL

(C) J. Böhmer, March 2018

Version 1.1

1 Reflection System

1.1 UFUNCTION()

BlueprintAuthorityOnly	This function will not execute from Blueprint code if running on something without network authority
BlueprintCosmetic	This function is cosmetic-only and will not run on dedicated servers
Blueprint-ImplementableEvent	This function is designed to be overriden by a blueprint. Dont provide a body for this function in C++.
BlueprintNativeEvent	This function is designed to be overriden by a blueprint, but also has a native implementation. Provide a body named [Function-Name]_Implementation
BlueprintPure	This function has no side effects on the object. Useful for "Get" functions. Implies Blueprint-Callable
BlueprintCallable	This function can be called from Blueprints and/or C++.
Category	Specifies the category of the function within the Editor. Supports sub-categories separated by "|"
Exec	This function is callable from the Console CLI.
AdvancedDisplay (Meta)	List parameter names seperated by commas. Every parameter in this list will appear as advanced pins in Blueprint Editor.
DevelopmentOnly (Meta)	Functions marked like this, will only run in Development builds and not in shipping builds. This is useful for things like Debug Output
BlueprintProtected (Meta)	This function can only be called on the owning object in a Blueprint not on any other instances.

```
UFUNCTION(Exec)
void ConsoleCommand(float param);
UFUNCTION(BlueprintPure)
static FRotator MakeRotator(flat f);
UFUNCTION(BlueprintImplementableEvent)
void ImportantEvent(int param);
UFUNCTION(meta=(DevelopmentOnly))
float NotSoImportantDebugFunc();
```

1.2 UENUM()

BlueprintType	Enums marked with this attribute, can be used from within Blueprints.
Bitflags (Meta)	Enums marked with this attribute, can be used as Bit-Mask in UPROPERTYs.
DisplayName (UMETA)	Use this inside of an Enum, to override the name displayed inside the Editor. See usage below:

```
//Note that enums must be declared before any class
UENUM(BlueprintType)
enum class ETestEnum {
    FirstEntry UMETA(DisplayName="OtherName"),
    //...
}
UENUM(meta=(Bitflags))
enum class EMyEnum {
FirstBit,
SecondBit
}
```

1.3 UPARAM()

ref	By default, when you pass an reference to a Blueprint callable function, it will appear as an output node in the Editor. Use this attribute, to mark it as an input.
DisplayName	Use this name to override the name under which the parameter appears in the Blueprint Editor

```
UFUNCTION(BlueprintCallable)
void MyFunc(UPARAM(ref) TArray<int>* IntArray);
UFUNCTION(BlueprintCallable)
void OtherFunc(UPARAM(DisplayName="Name" float f);
```

1.4 UCLASS()

Abstract	An class that is marked as abstract can not be placed or instanced during runtime. This is especially useful for classes, that does not provide functionality on their own and must be inherited and modified for meaningful usage.
Blueprintable	Classes marked with this attribute can be used as a base class for creating Blueprints. On Default this is deactivated. The attribute is inherited by child classes, use NotBlueprintable on childs to disable this.
BlueprintType	Classes with this attribute can be used as variable type in Blueprints.
Placeable	Classes marked as Placable, can be created and placed in a level, UI Scene or Blueprint via the Editor. The flag is inherited by all child classes, use NotPlacable on child to disable this.

```
UCLASS(Blueprintable)
class MyClass : public UObject {
//Class code ...
}
```

1.5 UPROPERTY()

BlueprintAssignable	Multicast Delegates only. Exposes property for assigning in Blueprints
BlueprintCallable	Multicast Delegates only. Property property for calling in Blueprints
BlueprintReadOnly	This property can be read by blueprints, but not modified
BlueprintReadWrite	This property can be read or written from a blueprint
Category	Specifies the category of the property within the Editor. Supports sub-categories separated by "|"
EditAnywhere	Indicates that this property can be edited via property windows, archetypes and instances within the Editor
EditDefaultsOnly	Indicates that this property can be edited by property windows, but only on archetypes. This operator is incompatible with the Visible* specifiers
EditFixedSize	Indicates that elements of an array can be modified in Editor, but its size cannot be changed
EditInstanceOnly	Indicates that this property can be edited by property windows, but only on instances, not on archetypes
Transient	Property is transient: shouldn't be saved, zero-filled at load time
VisibleAnywhere	Indicates that this property is visible in property windows, but cannot be edited at all
VisibleDefaultsOnly	Indicates that this property is only visible in property windows for archetypes, and cannot be edited
VisibleInstanceOnly	Indicates that this property is only visible in property windows for instances, not for archetypes, and cannot be edited
AdvancedDisplay	Moves the property into the Advanced dropdown in the Details panel within the Editor
NoClear	Hides the clear and browse button in the editor for this property.
EditCondition (Meta)	The property can only be edited in Editor if the specified bool Property is true. Use ! to invert logic (so you can only edit property if bool is false).
BitMask (mMeta)	Change the value of this property in the Editor using a BitMask, which means you can select the value of each single bits. Use BitMask to specify a Enum, which entries will be used to name each bit.
ClampMax / ClampMin (Meta)	Use this on float or int property, to specify a maximum/minimum, that can be entered for this property in the Editor
MakeEditWidget (Meta)	Use this on a Transform or Rotator property, and the property value can be changed using a widget inside the editor viewport.

```
UPROPERTY(EditAnywhere, Category="Category|Sub")
bool BoolProperty;
UPROPERTY(BlueprintReadOnly, AdvancedDisplay)
TSubclassOf<UStaticMesh> AdvancedMeshClass;
UPROPERTY(meta=(EditCondition="BoolProperty"))
uint16 ConditionalInt;
UPROPERTY(meta=(BitMask, BitMaskEnum="EMyEnum"))
int32 BitFlags;
UPROPERTY(meta=(ClampMin="3", ClampMax="4"))
float myFloat;
```

1.6 USTRUCT()

BlueprintType	Structs with this attribute can be used as type inside Blueprints. Make and Break nodes get automatically generated.

```
USTRUCT(BlueprintType)
struct MyStruct {
// ...
}
```

1.7 Delegates

Delegates allow to call variable functions via a type-safe way. There are 3 big types of delegates:

- Single-cast Delegates, which can have a single function target, declared with `DECLARE_DELEGATE_`
- Multi-cast Delegates, which can have multiple function targets, declared with `DECLARE_MULTICAST_DELEGATE_`
- Dynamic Multicasts, which can be serialized, and functions can be found by name, declared with `DECLARE_DYNAMIC_DELEGATE_` or `DECLARE_DYNAMIC_MULTICAST_DELEGATE_`

All Delegate macros have the syntax: `_DELEGATE_<Num>Params(Name,Param1Type,Param2Type,...)` or for functions with return value: `DECLARE_DELEGATE_RetVal(RetValType, Name)`

Code example:

```
DECLARE_MULTICAST_DELEGATE(VoidDelegate)
DECLARE_DELEGATE_OneParam(IntParamDelegate, int32)
DECLARE_DELEGATE_TwoParams(MyDelegate, int32, AActor*)
DECLARE_DELEGATE_RetVal_OneParam(int, Delegate2, uint8)

void MyFunc;
void MyFunc2(int32);

VoidDelegate Del1;
//Somewhere in func body
Del1.Add(this, FName("MyFunc")); //Add MyFunc
Del1.Broadcast(); //Call all bound functions

IntParamDelegate Del2;
Del2.Add(this, FName("MyFunc2")); //Bind MyFunc2
Del2.ExecuteIfBound(10); //Call MyFunc2
```

2 Useful Console commands

- **Show Collision:** Show collision components in game.
- **ToggleDebugCamera:** Switch to a separate camera, which you can move in world freely and shows some additional debug infos.

- **HighResShot [number]:** Makes a screen shot with [number] times your normal screen resolution. Instead of [number] you can provide a resolution the screenshot should have.
- **[CVar] ?:** Add a ? after a CVar name and a description about the CVar will be shown.
- **DumpConsoleCommands:** Prints a list of all available console commands and CVars.
- **slomo [float]:** Slow down or speed up the game. slomo 1.0 is default. slomo 1.5 is faster than normal, slomo 0.5 is slower.
- **open [mapname]:** Load and opens the map with the given name.
- **help:** Opens a page in browser which, lists all console commands and variables with a description. Searching and filter for specific commands is possible.

3 Classes and Functions

3.1 Base Gameplay Classes

- **UObject:** The base class, all classes, that should be used within C++ must extend. The name of child classes should start with U (e.g. UMyNewClass).
- **AActor:** Actor is the base class for all objects, that can be placed in a level. An Actor can has various Components. Child classes should start with A (e.g AMyNewActor).
- **APawn:** The base class, for all actors, that should be controled by players or AI.
- **ACharacter:** Characters are Pawn, which has a mesh collision and movement logic. They represent physical characters in the game world and can use CharacterMovementComponent for walking, flying, jumping and swiming logic.
- **UActorComponent:** The base class for all actor components. Components defines some reusable behavior, that can be added to different actors.
- **USceneComponent:** An Actor Component, which has a transform (position and rotation) and support for attachements.
- **UPrimitiveComponent:** A SceneComponent which can show some kind of geometry, usable for rendering and/or collision. Examples for this type are *StaticMeshComponent*, *SkeletalMeshComponent*, or the *ShapeComponent*s.

3.2 Datastructures and Helpers

- **TArray:** The mostly used container in UE4. The objects in it have a well-defined order, and functions are provided to create, get, modify or sort the elements. Similar to C++'s std::vector. You can iterate over the element like this:

```
TArray<AActor> ActorArray;
//Add MyActor 3 times
ActorArray.Init(MyActor, 3);
ActorArray.Add(AnotherActor);
//Retrieve the first Actor from array
auto FirstActor = ActorArray[0]
//Iterate over all Actor in Array
for (AActor* Actor : ActorArray) {
    Actor->SomeFunc();
}
```

- **TMap:** A container, where every element has a key (of any type), via which you identify every element. Similar to std::map

```
TMap<int32, FString> StringMap;
StringMap.Add(4, TEXT("Foo"));
StringMap.Add(-1, TEXT("Bar"));
//Iterate over all Pairs
for (auto& pair : StringMap)
{
    pair.Key; //Gets the key of the pair
    *pair.Value; //Gets the value of pair
}
```

- **TSet:** A (fast) container to store unique elements without order. Similar to C++'s std::Set

```
TSet<int32> mySet;
mySet.Add(3); //mySet = [3]
mySet.Add(5); //mySet = [3,5]
mySet.Add(3); //mySet = [3,5]
//Only one 3 can be added to mySet
```

- **TSubclassOf:** When you define a UProperty with the type TSubclassOf<UMyObject>, the editor allows you only to select classes, which are derived from UMyObject.

```
UPROPERTY(EditAnywhere)
TSubclassOf<AActor> ActorType;
```

- **FName:** FNames provide a fast possibility to reference to things via a name. FNames are case-insensitive and can not be manipulated (they are immutable).
- **FText:** FText represents a string that can be displayed to user. It has a built in system for localization (so FTexts can be translated) and are immutable.
- **FString:** FString is the only class that allows manipulation. FStrings can be searched modified and compared, but this makes FStrings less performant than FText or FName.

3.3 Useful Functions and snippets

- **UE_LOG():** This functions allows to print message to the UE Log or the Output Log in the Editor. You can set a category (you can use LogTemp for temporal usage) and verbosity (like Error, Warning or Display). If you want to output a variable, you can use printf syntax. Usage Example:

```
//Print Test to console
UE_LOG(LogTemp, Warning, TEXT("Test"));
//Print the value of int n and a string
UE_LOG(LogTemp, Display, TEXT("n=%d"), n);
UE_LOG(LogTemp, Error, TEXT("%s"), MyString);
```

- **AddOnScreenDebugMessage():** If you want to print a debug message directly to the screen you can use AddOnScreenDebugMessage() from GEngine. You can specify a key, displaying time and display color. A message overrides an older message with the same key. Usage example:

```
GEngine->AddOnScreenDebugMessage(-1, 5.f,
     FColor::Red, TEXT("5 second Message"));
//Use FString, if you want to print vars
GEngine->AddOnScreenDebugMessage(-1, 5.f,
     FColor::Red,
     FString::Printf(TEXT("x: %f, y: %f"), x, y));
```

- **NewObject():** NewObject() creates a new UObject with the specific type. Objects created using NewObject() are not visible to the Editor, if you need that, use CreateDefaultSubObject() instead. Usage example:

```
auto RT = NewObject<UTextureRenderTarget2D>();
```

- **CreateDefaultSubobject():** This function creates a new named UObject with the specific type in the context of the current actor. Created objects are visible to the Editor, but this function can only be used in constructor. Usage example:

```
auto Mesh = CreateDefaultSubobject
    <UStaticMeshComponent>(FName("Mesh"));
```

- **LoadObject():** This function loads an objects from a specific asset. Usage example:

```
auto Mesh = LoadObject<UStaticMesh>(nullptr,
    TEXT("StaticMesh'/Asset/Path/Mesh.Mesh'");
```

- **Cast():** Casts an object to the given type. Returns nullptr if the object is not castable to this type. The object that should be casted, must be based on UObject, to work properly. Usage example:

```
AActor* Actor = Cast<AActor>(Other);
if(Actor != nullptr) {
/* do something */ }
```

- **Console Variables:** To define a variable that can be changed via editor (CVar), you can use TAutoConsoleVariable in any C++ file:

```
static TAutoConsoleVariable<int32> CVarMyVar(
TEXT("r.MyVar"),
2,//Default value
TEXT("CVar Description\n")
TEXT("  1: Infos about possible values \n"),
ECVF_Scalability | ECVF_RenderThreadSafe);
```

  The last parameter are some flags, that defines the behavior of the CVar. When you add ECVF_Cheat flag, the CVar can be only changed in cheat mode. If you want to access the CVar's value in C++, then use this:

```
// only needed if you are not in the same cpp file
extern TAutoConsoleVariable<int32> CVarMyVar;
// Retrieve the MyVar value via Game Thread
int32 MyVar = CVarMyVar.GetValueOnGameThread();
```

3.4 Assertions

Assertions can be used to ensure that specific conditions are fulfilled, before continue in the program flow. If the checks are not successful, the execution is halted. Assertions will only work when the `DO_CHECK` macro is set (and not zero) during compiling. There are different types of assertions:

- **check():** If the expression inside check() is false, the execution will be halted. The expression is only evaluated, when `DO_CHECK` is set. If you need that the expression is always evaluated, then use verify().

```
check(OneProperty == 1);
verify(ImportantCall() != nullptr);
```

- **checkf():** Behaves like check(), but additional debug info is printed. verifyf() works analogous.

- **checkNoEntry():** You can mark code path that should never be executed with this assertion. If it is still be called, the execution is halted.

```
switch(Property) {
    case EEnum:Val1:
        return 1;
    default:
        checkNoEntry();
}
```

- **unimplemented():** Use this assertions, on functions that are yet unimplemented or must be overridden to work properly.

```
void Function() {
    //This func must be overriden to work
    unimplemented();
}
```

3.5 Draw Debug Functions

To use the following functions you need to include `DrawDebugHelpers.h`.

- **DrawDebugPoint():** Draw a point in the world at a given location. You can choose Color and size of the point:

```
DrawDebugPoint(
   GetWorld(),
   MyLocation, //The location as FVector
   20,  //size of the point
   FColor(255,0,0),  //the color
   false, //Not persistant
   10.f //10s lifetime
);
```

- **DrawDebugLine():** Draw a line between to points in the world:

```
DrawDebugLine(
   GetWorld(),
   Start, //Start point
   End,  //End point
   FColor(0,255,0), //Line Color
   false, //Not persistant
   -1, //Infinite lifetime
   0,
   10 //Line Thickness
);
```