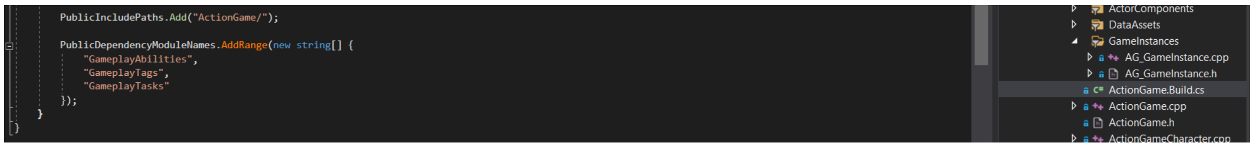
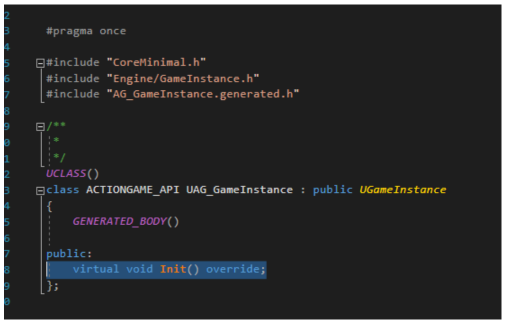
1. GAS Plugin, Ability, Ability System Component, Attribute Set, Character

1. **First enable Gameplay Abilities Plugin**.
2. **Add few base C++ classes:**
    - AbilitySystemComponent
    - AttributeSet
    - GameInstance
3. **Modify** *.Build.cs file on visual studio:
    - Use PublicIncludePaths.Add("*/"); to add our project to include directories
    - Add these modules:

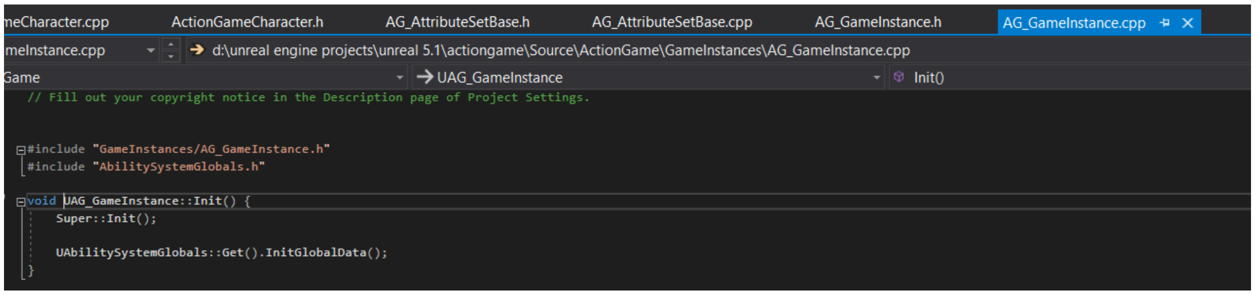


```
        PublicIncludePaths.Add("ActionGame/");

        PublicDependencyModuleNames.AddRange(new string[] {
            "GameplayAbilities",
            "GameplayTags",
            "GameplayTasks"
        });
    }
}
```

4. **We use created GameInstance to override the Init Function** and we use it as a point to initialize ability globals in GlobalData



```
#pragma once

#include "CoreMinimal.h"
#include "Engine/GameInstance.h"
#include "AG_GameInstance.generated.h"

/**
 *
 */
UCLASS()
class ACTIONGAME_API UAG_GameInstance : public UGameInstance
{
    GENERATED_BODY()

public:
    virtual void Init() override;
};
```



```
// Fill out your copyright notice in the Description page of Project Settings.


#include "GameInstances/AG_GameInstance.h"
#include "AbilitySystemGlobals.h"

void UAG_GameInstance::Init() {
    Super::Init();

    UAbilitySystemGlobals::Get().InitGlobalData();
}
```

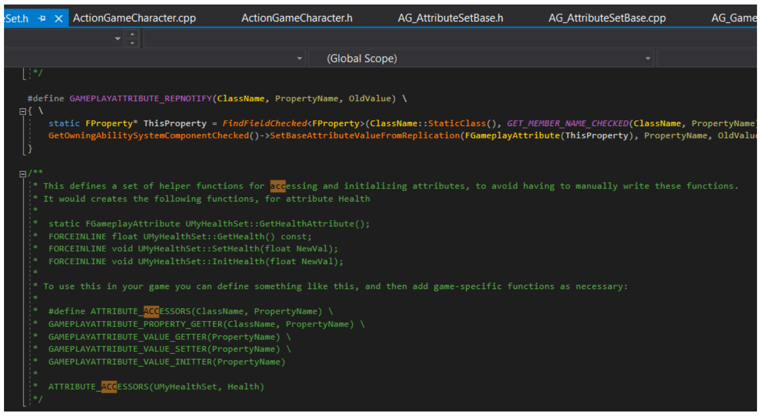
the code is invoking the InitGlobalData function of the UAbilitySystemGlobals class, which is part of the Ability System plugin in Unreal Engine. This function is responsible for **initializing global data related to the Ability System**.

By calling UAbilitySystemGlobals::Get().InitGlobalData(), the code **ensures** that the necessary global data for the Ability System is initialized during the initialization of the GameInstance. This allows the Ability System to function correctly throughout the game.

In summary, this code initializes the base GameInstance class and then initializes the global data for the Ability System in Unreal Engine.

---

5. **We set-up AttributeSet** and we can grab some premade code from AttributeSet.h file.

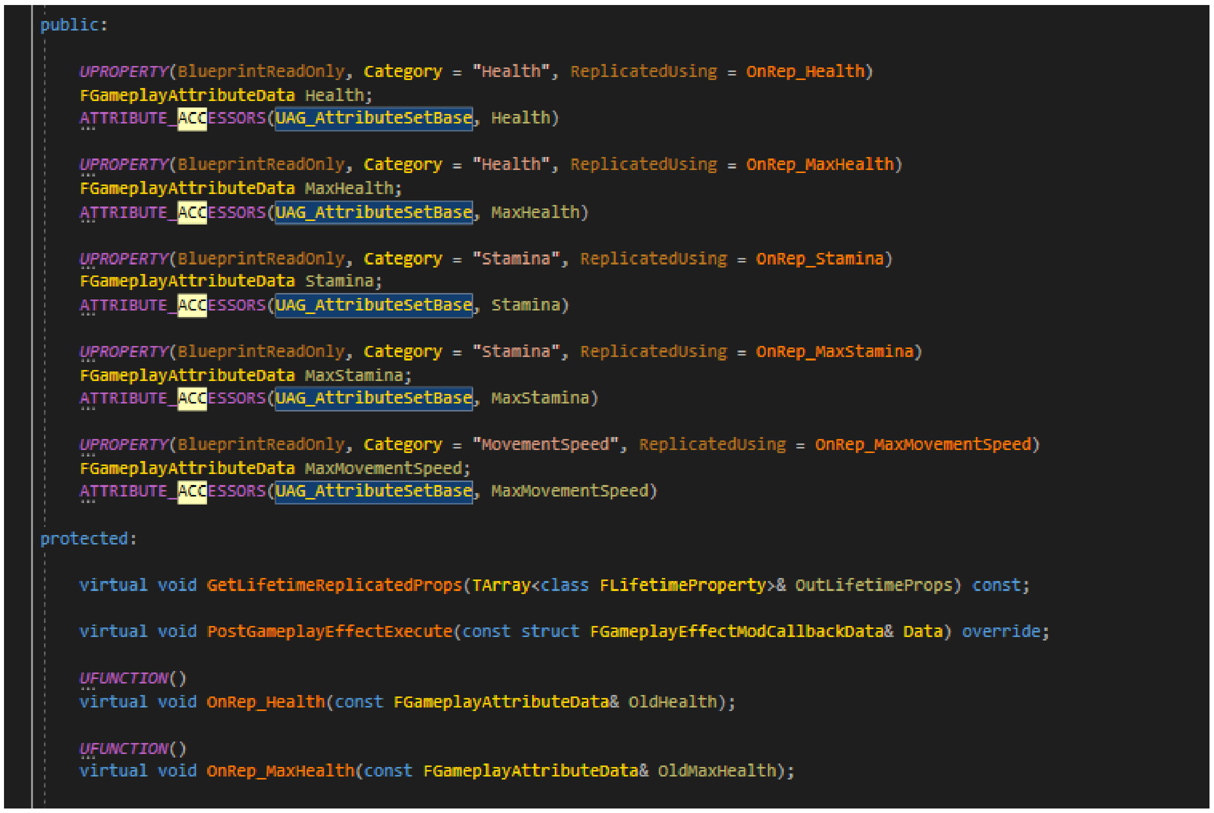


```
*/

#define GAMEPLAYATTRIBUTE_REPNOTIFY(ClassName, PropertyName, OldValue) \
{ \
    static FProperty* ThisProperty = FindFieldChecked<FProperty>(ClassName::StaticClass(), GET_MEMBER_NAME_CHECKED(ClassName, PropertyName
    GetOwningAbilitySystemComponentChecked()->SetBaseAttributeValueFromReplication(FGameplayAttribute(ThisProperty), PropertyName, OldValu
}

/**
 * This defines a set of helper functions for accessing and initializing attributes, to avoid having to manually write these functions.
 * It would creates the following functions, for attribute Health
 *
 *  static FGameplayAttribute UMyHealthSet::GetHealthAttribute();
 *  FORCEINLINE float UMyHealthSet::GetHealth() const;
 *  FORCEINLINE void UMyHealthSet::SetHealth(float NewVal);
 *  FORCEINLINE void UMyHealthSet::InitHealth(float NewVal);
 *
 * To use this in your game you can define something like this, and then add game-specific functions as necessary:
 *
 *  #define ATTRIBUTE_ACCESSORS(ClassName, PropertyName) \
 *  GAMEPLAYATTRIBUTE_PROPERTY_GETTER(ClassName, PropertyName) \
 *  GAMEPLAYATTRIBUTE_VALUE_GETTER(PropertyName) \
 *  GAMEPLAYATTRIBUTE_VALUE_SETTER(PropertyName) \
 *  GAMEPLAYATTRIBUTE_VALUE_INITTER(PropertyName)
 *
 *  ATTRIBUTE_ACCESSORS(UMyHealthSet, Health)
 */
```

```
 *      #define ATTRIBUTE_ACCESSORS(ClassName, PropertyName) \
 *      GAMEPLAYATTRIBUTE_PROPERTY_GETTER(ClassName, PropertyName) \
 *      GAMEPLAYATTRIBUTE_VALUE_GETTER(PropertyName) \
 *      GAMEPLAYATTRIBUTE_VALUE_SETTER(PropertyName) \
 *      GAMEPLAYATTRIBUTE_VALUE_INITTER(PropertyName)
```

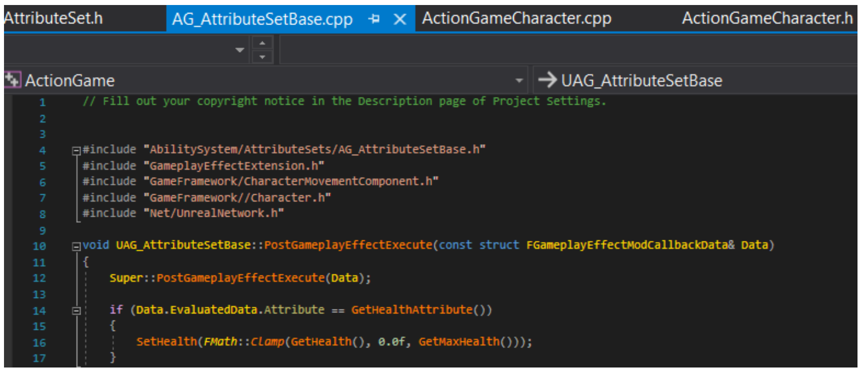
- We will create 2 attribute for now. Health and MaxHealth. We need the second one because we change the Health by a lot. The MaxHealth can also be changed and by making it an attribute we allowing it to be done by GameplayEffects.

- We also need to override PostGameplayEffectExecute function and write few On_Rep functions for our attributes in our class.



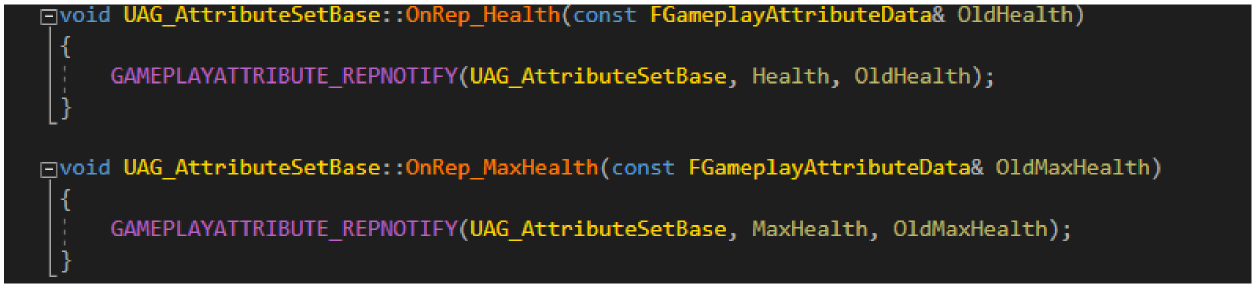
```
public:

    UPROPERTY(BlueprintReadOnly, Category = "Health", ReplicatedUsing = OnRep_Health)
    FGameplayAttributeData Health;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, Health)

    UPROPERTY(BlueprintReadOnly, Category = "Health", ReplicatedUsing = OnRep_MaxHealth)
    FGameplayAttributeData MaxHealth;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, MaxHealth)

    UPROPERTY(BlueprintReadOnly, Category = "Stamina", ReplicatedUsing = OnRep_Stamina)
    FGameplayAttributeData Stamina;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, Stamina)

    UPROPERTY(BlueprintReadOnly, Category = "Stamina", ReplicatedUsing = OnRep_MaxStamina)
    FGameplayAttributeData MaxStamina;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, MaxStamina)

    UPROPERTY(BlueprintReadOnly, Category = "MovementSpeed", ReplicatedUsing = OnRep_MaxMovementSpeed)
    FGameplayAttributeData MaxMovementSpeed;
    ATTRIBUTE_ACCESSORS(UAG_AttributeSetBase, MaxMovementSpeed)

protected:

    virtual void GetLifetimeReplicatedProps(TArray<class FLifetimeProperty>& OutLifetimeProps) const;

    virtual void PostGameplayEffectExecute(const struct FGameplayEffectModCallbackData& Data) override;

    UFUNCTION()
    virtual void OnRep_Health(const FGameplayAttributeData& OldHealth);

    UFUNCTION()
    virtual void OnRep_MaxHealth(const FGameplayAttributeData& OldMaxHealth);
```

For right now we need PostGameplayEffectExecute to handle Health Rang.



```
// Fill out your copyright notice in the Description page of Project Settings.


#include "AbilitySystem/AttributeSets/AG_AttributeSetBase.h"
#include "GameplayEffectExtension.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "GameFramework//Character.h"
#include "Net/UnrealNetwork.h"

void UAG_AttributeSetBase::PostGameplayEffectExecute(const struct FGameplayEffectModCallbackData& Data)
{
    Super::PostGameplayEffectExecute(Data);

    if (Data.EvaluatedData.Attribute == GetHealthAttribute())
    {
        SetHealth(FMath::Clamp(GetHealth(), 0.0f, GetMaxHealth()));
    }
```

And we add GamplayArreibute_RepNotify Macro:

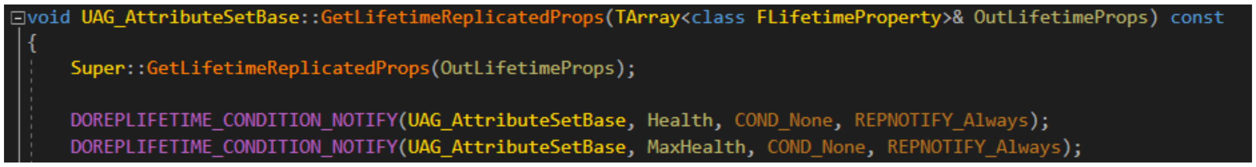


```
void UAG_AttributeSetBase::OnRep_Health(const FGameplayAttributeData& OldHealth)
{
    GAMEPLAYATTRIBUTE_REPNOTIFY(UAG_AttributeSetBase, Health, OldHealth);
}

void UAG_AttributeSetBase::OnRep_MaxHealth(const FGameplayAttributeData& OldMaxHealth)
{
    GAMEPLAYATTRIBUTE_REPNOTIFY(UAG_AttributeSetBase, MaxHealth, OldMaxHealth);
}
```

- And the last function is GetLifetimeReplicatedProps: (this is in the header and in the above pic too)

This Function allows us **to specify the properties that should be replicated and under which conditions**.

(we already saw it on the header in some pics above) in the cpp:



```
void UAG_AttributeSetBase::GetLifetimeReplicatedProps(TArray<class FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME_CONDITION_NOTIFY(UAG_AttributeSetBase, Health, COND_None, REPNOTIFY_Always);
    DOREPLIFETIME_CONDITION_NOTIFY(UAG_AttributeSetBase, MaxHealth, COND_None, REPNOTIFY_Always);
```

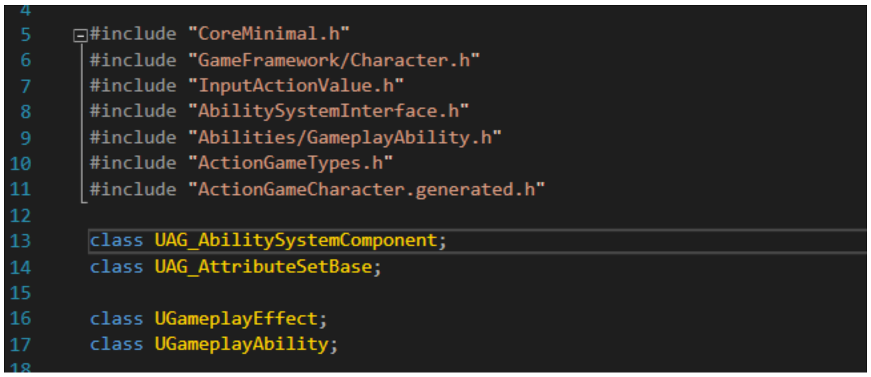
The parameters of the **DOREPLIFETIME_CONDITION_NOTIFY** macro specify the class type, the property name, the replication condition (COND_None means it will always replicate), and the replication notification (REPNOTIFY_Always means the property changes will be notified to clients whenever they occur).

---

6. Modify Character class:

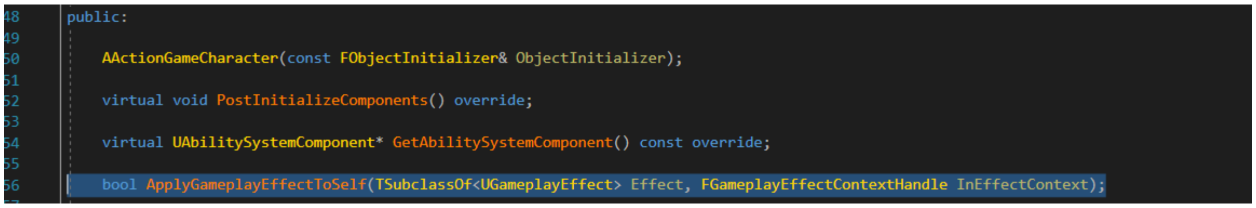
- for now just add And class definitions:

```
#include "Abilities/GameplayAbility.h"

class UAG_AbilitySystemComponent;

class UAG_AttributeSetBase;

class UGameplayEffect;

class UGameplayAbility;
```



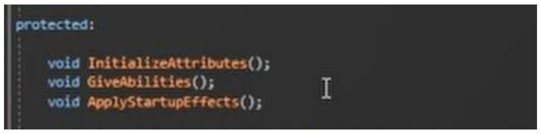
```cpp
#include "CoreMinimal.h"
#include "GameFramework/Character.h"
#include "InputActionValue.h"
#include "AbilitySystemInterface.h"
#include "Abilities/GameplayAbility.h"
#include "ActionGameTypes.h"
#include "ActionGameCharacter.generated.h"

class UAG_AbilitySystemComponent;
class UAG_AttributeSetBase;

class UGameplayEffect;
class UGameplayAbility;
```

- add a helper function that might be useful in future:

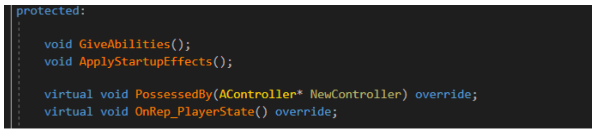


```cpp
public:

    AActionGameCharacter(const FObjectInitializer& ObjectInitializer);

    virtual void PostInitializeComponents() override;

    virtual UAbilitySystemComponent* GetAbilitySystemComponent() const override;

    bool ApplyGameplayEffectToSelf(TSubclassOf<UGameplayEffect> Effect, FGameplayEffectContextHandle InEffectContext);
```

- Then we need several initialization function for AtributeEffects and Abilities but in future we use data assets so this part will be deleted.



```cpp
protected:

    void InitializeAttributes();
    void GiveAbilities();
    void ApplyStartupEffects();
```



```cpp
protected:

    void GiveAbilities();
    void ApplyStartupEffects();

    virtual void PossessedBy(AController* NewController) override;
    virtual void OnRep_PlayerState() override;
```

The right one is what will be left

PossessedBy function is called when the character is possessed by a **controller**. Possession occurs when a player **or** an AI **controller** takes control of the character.

The PossessedBy function is an override, which means it replaces the base implementation provided by the ACharacter class.

Inside this function, you can implement any logic that needs to be executed when the character is possessed. For example, you may want to initialize certain attributes, set up gameplay-related components, or perform any other necessary setup specific to the possessed character.

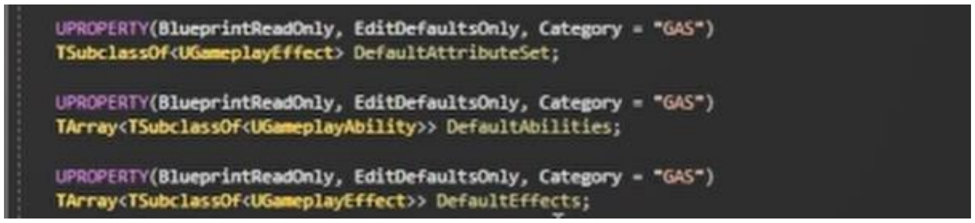
OnRep_PlayerState function is called when the character's PlayerState property is replicated across the network and its value changes.

Inside this function, you can implement any logic that needs to be executed when the PlayerState **property changes**. For example, you may want to update the character's appearance, adjust gameplay behavior based on the new player state, or trigger any relevant events or effects.

The OnRep_PlayerState function is typically **used to react to changes in the player's state and update the character's behavior accordingly**.

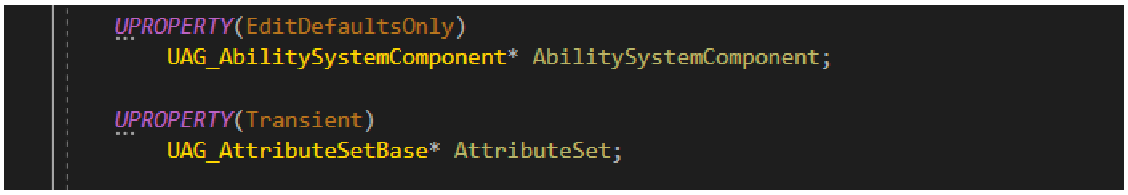
- Next is setting-up Properties: first is GameplayEffect to set up initial attribute because we should not modify our attributes directly.

We will change this in future so not going so deep in to it.



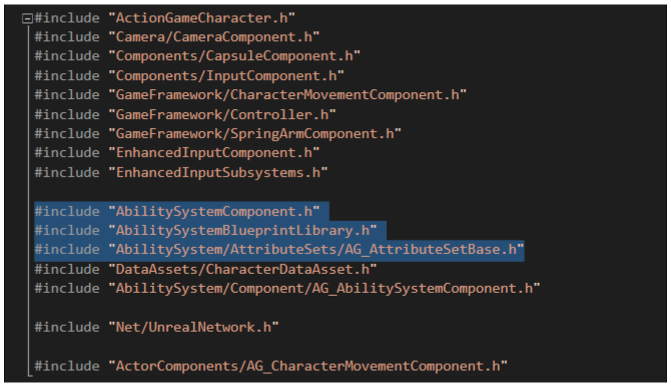
```
UPROPERTY(BlueprintReadOnly, EditDefaultsOnly, Category = "GAS")
TSubclassOf<UGameplayEffect> DefaultAttributeSet;

UPROPERTY(BlueprintReadOnly, EditDefaultsOnly, Category = "GAS")
TArray<TSubclassOf<UGameplayAbility>> DefaultAbilities;

UPROPERTY(BlueprintReadOnly, EditDefaultsOnly, Category = "GAS")
TArray<TSubclassOf<UGameplayEffect>> DefaultEffects;
```

- Finally we add our AbilitySystemComponent and our AttributeSet.



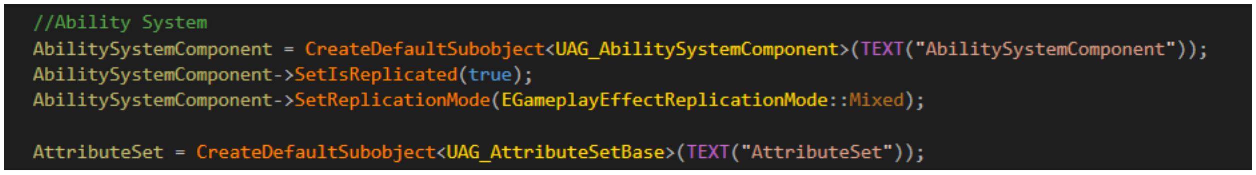
```
UPROPERTY(EditDefaultsOnly)
    UAG_AbilitySystemComponent* AbilitySystemComponent;

UPROPERTY(Transient)
    UAG_AttributeSetBase* AttributeSet;
```

CPP side:



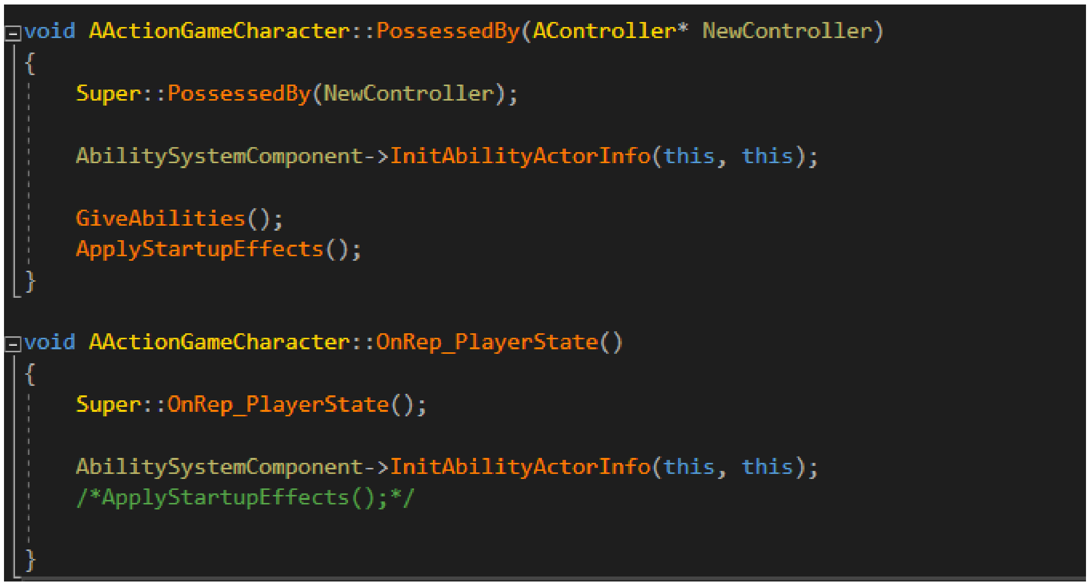
```
#include "ActionGameCharacter.h"
#include "Camera/CameraComponent.h"
#include "Components/CapsuleComponent.h"
#include "Components/InputComponent.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "GameFramework/Controller.h"
#include "GameFramework/SpringArmComponent.h"
#include "EnhancedInputComponent.h"
#include "EnhancedInputSubsystems.h"

#include "AbilitySystemComponent.h"
#include "AbilitySystemBlueprintLibrary.h"
#include "AbilitySystem/AttributeSets/AG_AttributeSetBase.h"
#include "DataAssets/CharacterDataAsset.h"
#include "AbilitySystem/Component/AG_AbilitySystemComponent.h"

#include "Net/UnrealNetwork.h"

#include "ActorComponents/AG_CharacterMovementComponent.h"
```

- We add our AbilitySystemComponent and AttributeSet to our constructor

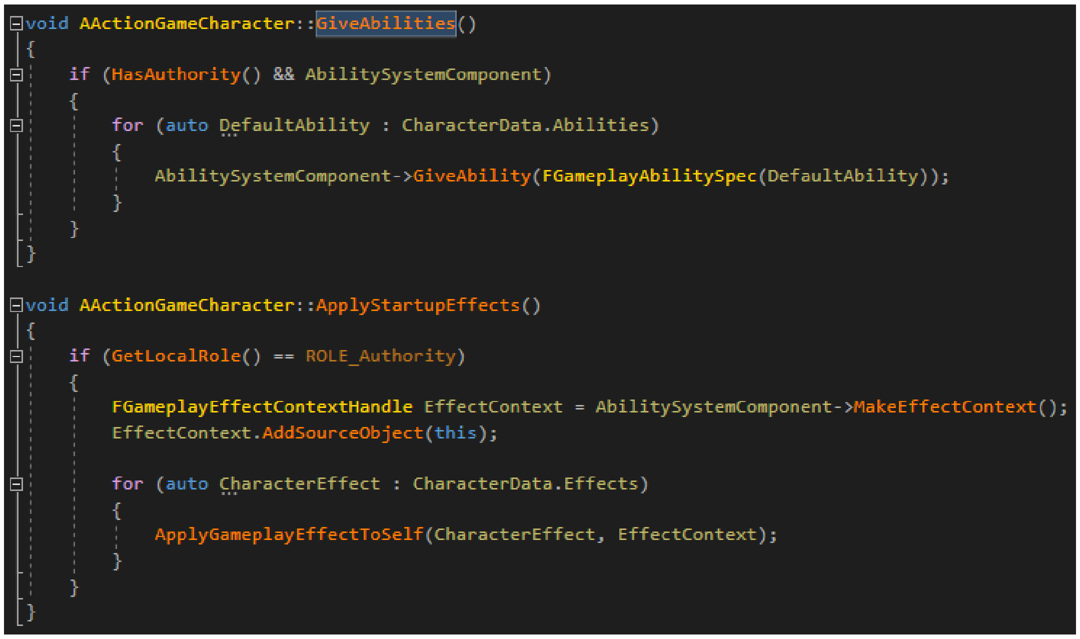


```cpp
//Ability System
AbilitySystemComponent = CreateDefaultSubobject<UAG_AbilitySystemComponent>(TEXT("AbilitySystemComponent"));
AbilitySystemComponent->SetIsReplicated(true);
AbilitySystemComponent->SetReplicationMode(EGameplayEffectReplicationMode::Mixed);

AttributeSet = CreateDefaultSubobject<UAG_AttributeSetBase>(TEXT("AttributeSet"));
```

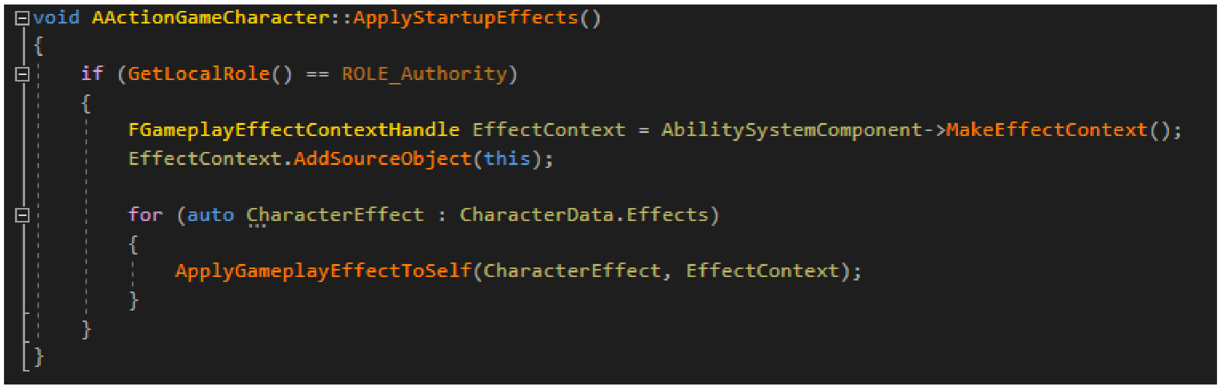
PossessedBy is for **initializing** the server side and OnRep_PlayerState for client side.



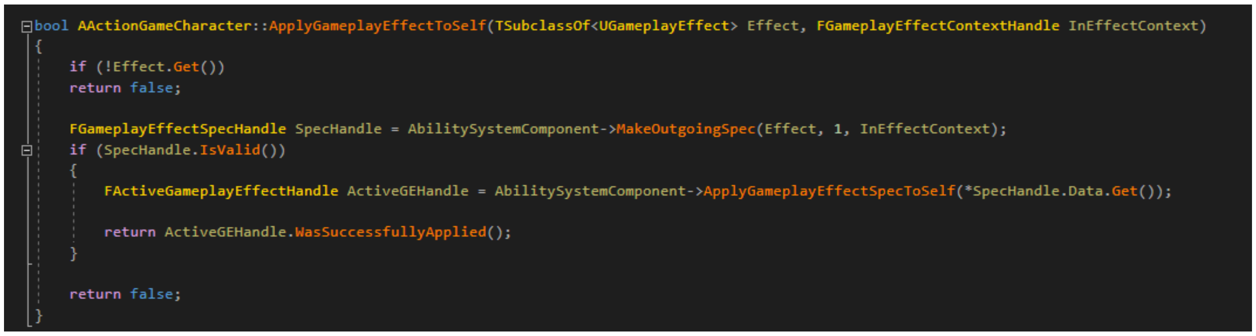
```cpp
void AActionGameCharacter::PossessedBy(AController* NewController)
{
    Super::PossessedBy(NewController);

    AbilitySystemComponent->InitAbilityActorInfo(this, this);

    GiveAbilities();
    ApplyStartupEffects();
}

void AActionGameCharacter::OnRep_PlayerState()
{
    Super::OnRep_PlayerState();

    AbilitySystemComponent->InitAbilityActorInfo(this, this);
    /*ApplyStartupEffects();*/

}
```



```cpp
void AActionGameCharacter::GiveAbilities()
{
    if (HasAuthority() && AbilitySystemComponent)
    {
        for (auto DefaultAbility : CharacterData.Abilities)
        {
            AbilitySystemComponent->GiveAbility(FGameplayAbilitySpec(DefaultAbility));
        }
    }
}

void AActionGameCharacter::ApplyStartupEffects()
{
    if (GetLocalRole() == ROLE_Authority)
    {
        FGameplayEffectContextHandle EffectContext = AbilitySystemComponent->MakeEffectContext();
        EffectContext.AddSourceObject(this);

        for (auto CharacterEffect : CharacterData.Effects)
        {
            ApplyGameplayEffectToSelf(CharacterEffect, EffectContext);
        }
    }
}
```



```cpp
void AActionGameCharacter::ApplyStartupEffects()
{
    if (GetLocalRole() == ROLE_Authority)
    {
        FGameplayEffectContextHandle EffectContext = AbilitySystemComponent->MakeEffectContext();
        EffectContext.AddSourceObject(this);

        for (auto CharacterEffect : CharacterData.Effects)
        {
            ApplyGameplayEffectToSelf(CharacterEffect, EffectContext);
        }
    }
}
```

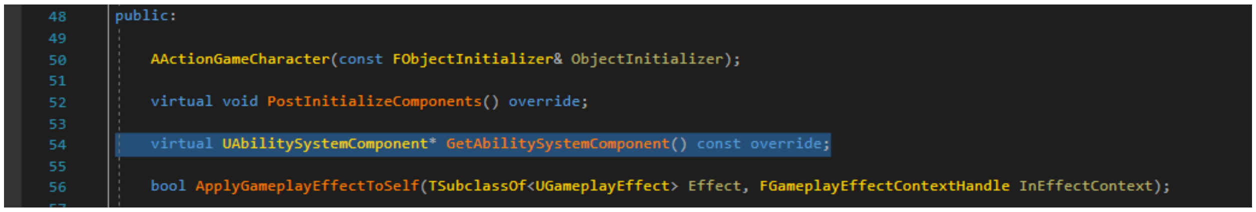


```cpp
bool AActionGameCharacter::ApplyGameplayEffectToSelf(TSubclassOf<UGameplayEffect> Effect, FGameplayEffectContextHandle InEffectContext)
{
    if (!Effect.Get())
    return false;

    FGameplayEffectSpecHandle SpecHandle = AbilitySystemComponent->MakeOutgoingSpec(Effect, 1, InEffectContext);
    if (SpecHandle.IsValid())
    {
        FActiveGameplayEffectHandle ActiveGEHandle = AbilitySystemComponent->ApplyGameplayEffectSpecToSelf(*SpecHandle.Data.Get());

        return ActiveGEHandle.WasSuccessfullyApplied();
    }

    return false;
}
```

---

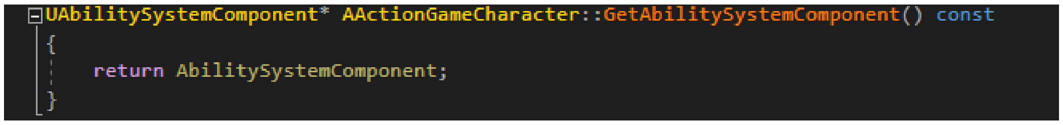
7. One important thing in our header is IAbilitySystemInterface, this is important for the global function that can get the AbilitySystemComponent through the interface.

#include "AbilitySystemInterface.h"



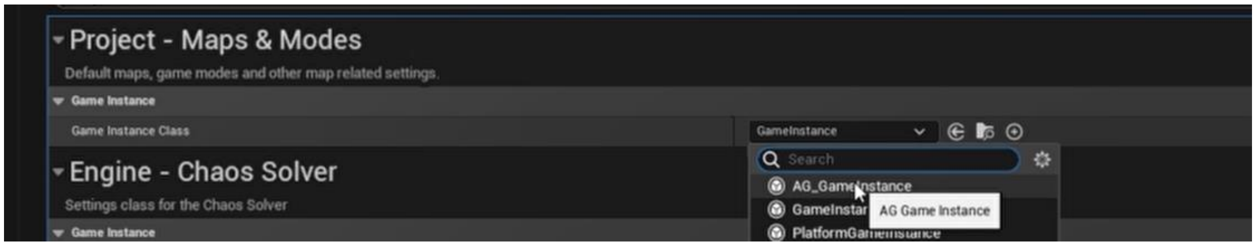
```cpp
public:

    AActionGameCharacter(const FObjectInitializer& ObjectInitializer);

    virtual void PostInitializeComponents() override;

    virtual UAbilitySystemComponent* GetAbilitySystemComponent() const override;

    bool ApplyGameplayEffectToSelf(TSubclassOf<UGameplayEffect> Effect, FGameplayEffectContextHandle InEffectContext);
```

We add the highlighted one to our header and the below on is the implantation:



```cpp
UAbilitySystemComponent* AActionGameCharacter::GetAbilitySystemComponent() const
{
    return AbilitySystemComponent;
}
```

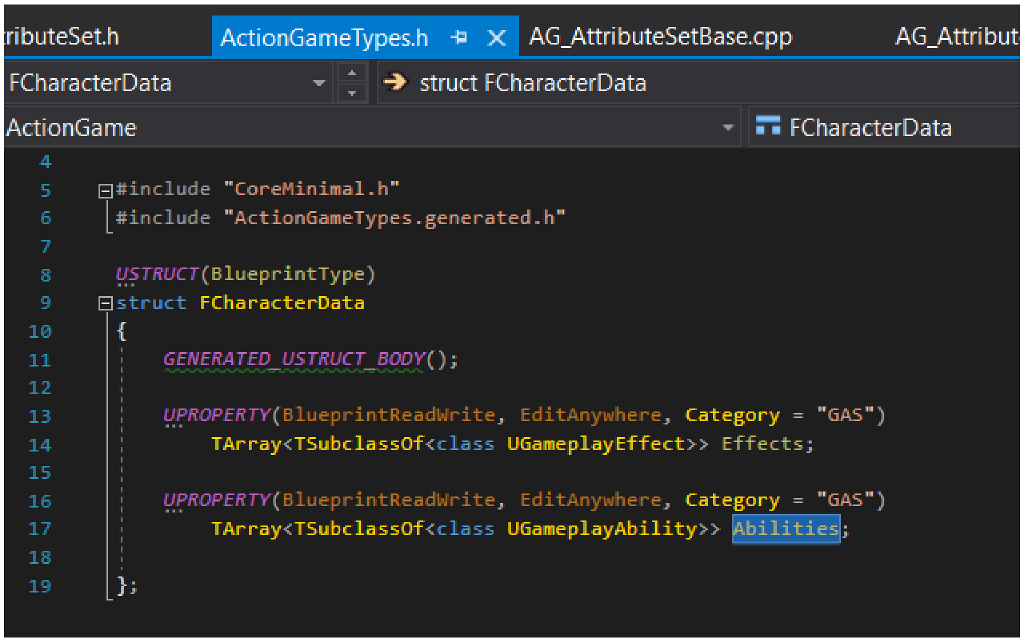
8. We should set our GameInstance as the default on



3. Character Data, Character Data Asset

1. Create one single c++ file (don't choose any parent for that class. We use it to place all our common data types.

- Create an UStruct (unreal struct) that will contain important data about character and it's BluprintType. And for now we just add it's Effects and Abilities.

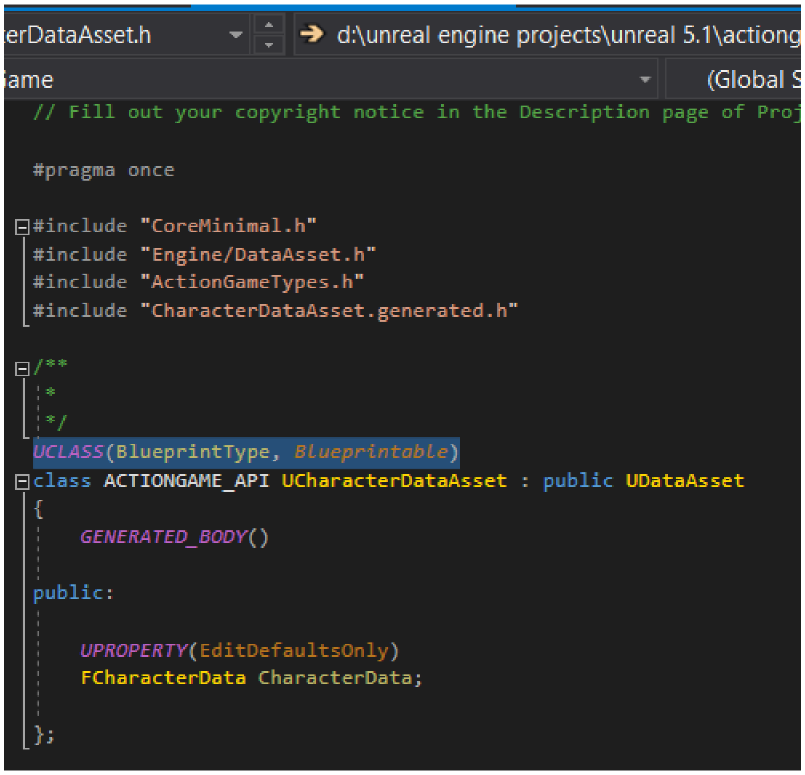


```
#include "CoreMinimal.h"
#include "ActionGameTypes.generated.h"

USTRUCT(BlueprintType)
struct FCharacterData
{
	GENERATED_USTRUCT_BODY();

	UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
		TArray<TSubclassOf<class UGameplayEffect>> Effects;

	UPROPERTY(BlueprintReadWrite, EditAnywhere, Category = "GAS")
		TArray<TSubclassOf<class UGameplayAbility>> Abilities;

};
```

- We also remove it's .cpp file and regenerate projects files.

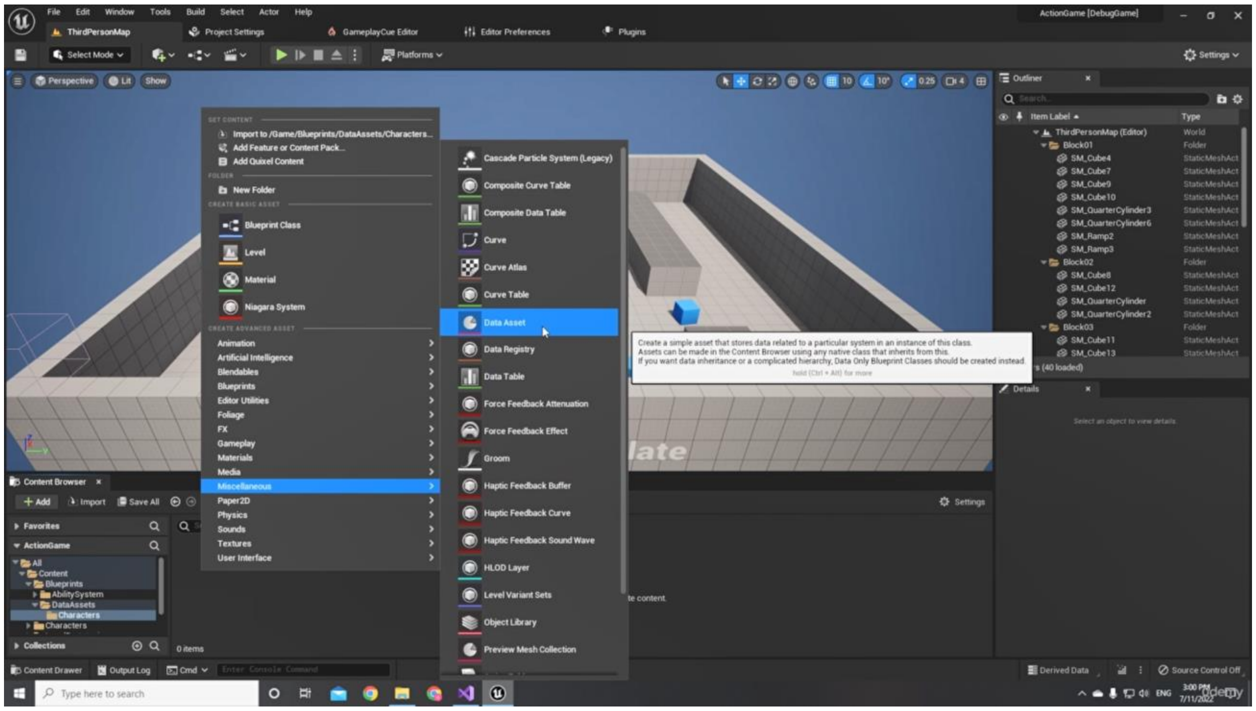
2. Next thing we need is a place to set-up the data and for this we **create a DataAsset class**. Each DataAsset then represent 1 character and could be possibly reused by many at once.

Pay attention that we make that class BueprintType



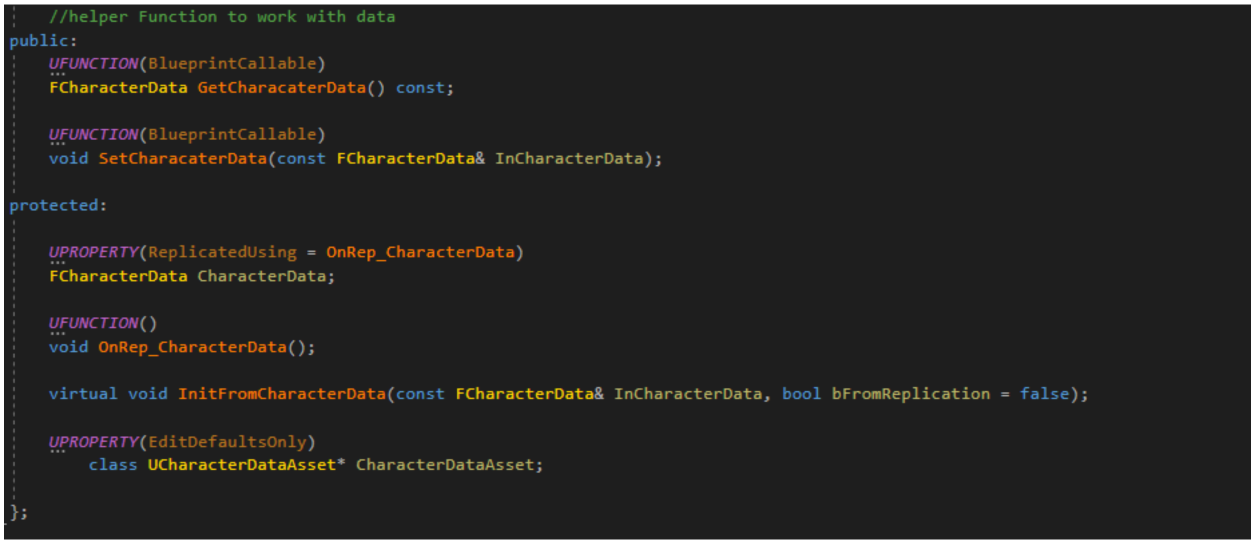
```
erDataAsset.h          d:\unreal engine projects\unreal 5.1\actiong
ame                    (Global S
// Fill out your copyright notice in the Description page of Proj

#pragma once

#include "CoreMinimal.h"
#include "Engine/DataAsset.h"
#include "ActionGameTypes.h"
#include "CharacterDataAsset.generated.h"

/**
 *
 */
UCLASS(BlueprintType, Blueprintable)
class ACTIONGAME_API UCharacterDataAsset : public UDataAsset
{
    GENERATED_BODY()

public:

    UPROPERTY(EditDefaultsOnly)
    FCharacterData CharacterData;

};
```

3. Then we create a DataAsset Blueprint and choose the class that we just created.



Create a simple asset that stores data related to a particular system in an instance of this class.
Assets can be made in the Content Browser using any native class that inherits from this.
If you want data inheritance or a complicated hierarchy, Data Only Blueprint Classes should be created instead.
hold (Ctrl + Alt) for more

4. Inside of our **character** we create some helper functions



```
    //helper Function to work with data
public:
    UFUNCTION(BlueprintCallable)
    FCharacterData GetCharacaterData() const;

    UFUNCTION(BlueprintCallable)
    void SetCharacaterData(const FCharacterData& InCharacterData);

protected:

    UPROPERTY(ReplicatedUsing = OnRep_CharacterData)
    FCharacterData CharacterData;

    UFUNCTION()
    void OnRep_CharacterData();

    virtual void InitFromCharacterData(const FCharacterData& InCharacterData, bool bFromReplication = false);

    UPROPERTY(EditDefaultsOnly)
        class UCharacterDataAsset* CharacterDataAsset;

};
```

The character data itself will be replicated and we'll have a Notify Function (OnRep_CharacterData) where we can track it's network update.

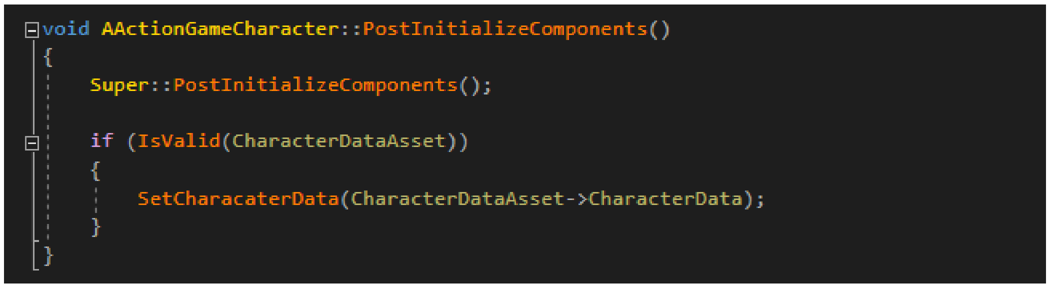
The InitFromCharacterData will be the exact place where we'll do the required modification based on the data.(based what I did figured it will be triggered when we are sure that it did replicated with OnRep_CharacterData and we can get the DataAsset Blueprint in it with a bool that tells us if it has been replicated or not)

The CharacterDataAsset is the reference to our DataAsset.

- We can also do initialization inside of PostInitializeComponents.

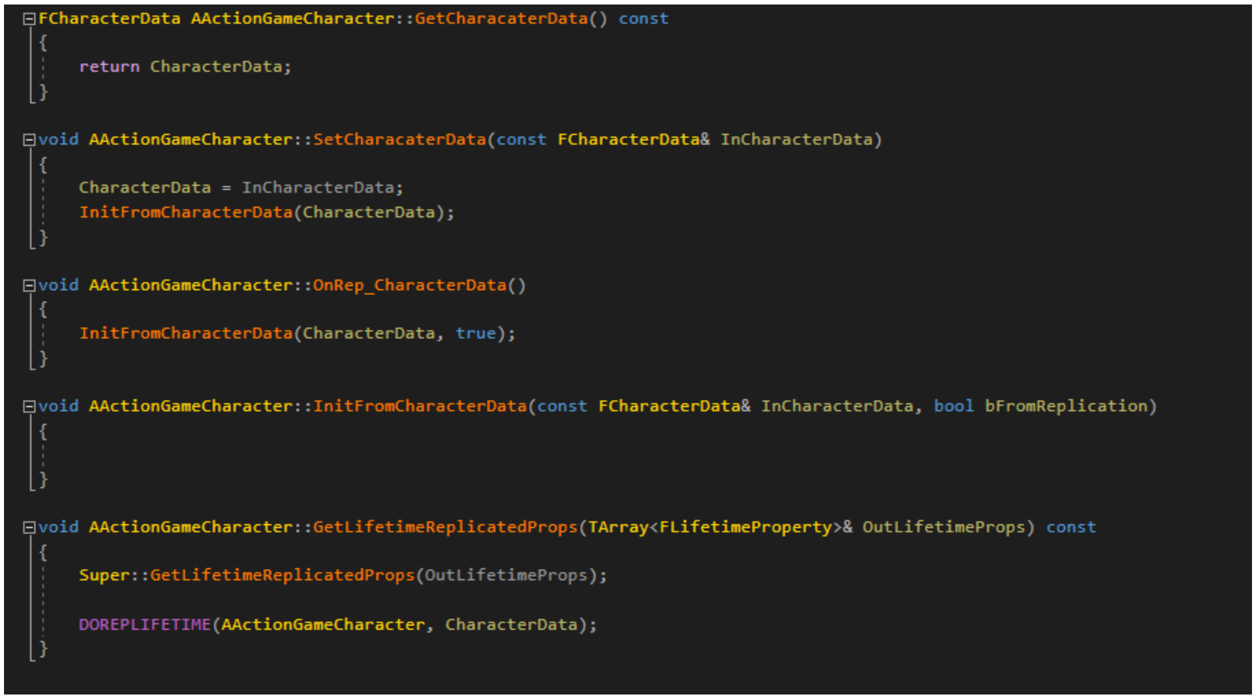
UCharacterDataAsset is a data asset class used to define and store character data that can be edited in the Unreal Editor, while FCharacterData is a **runtime data structure** used to hold and manipulate character-specific data during gameplay.

UCharacterDataAsset is for static, configurable data, while FCharacterData is for dynamic, runtime data specific to an individual character instance.



```cpp
void AActionGameCharacter::PostInitializeComponents()
{
    Super::PostInitializeComponents();

    if (IsValid(CharacterDataAsset))
    {
        SetCharacaterData(CharacterDataAsset->CharacterData);
    }
}
```

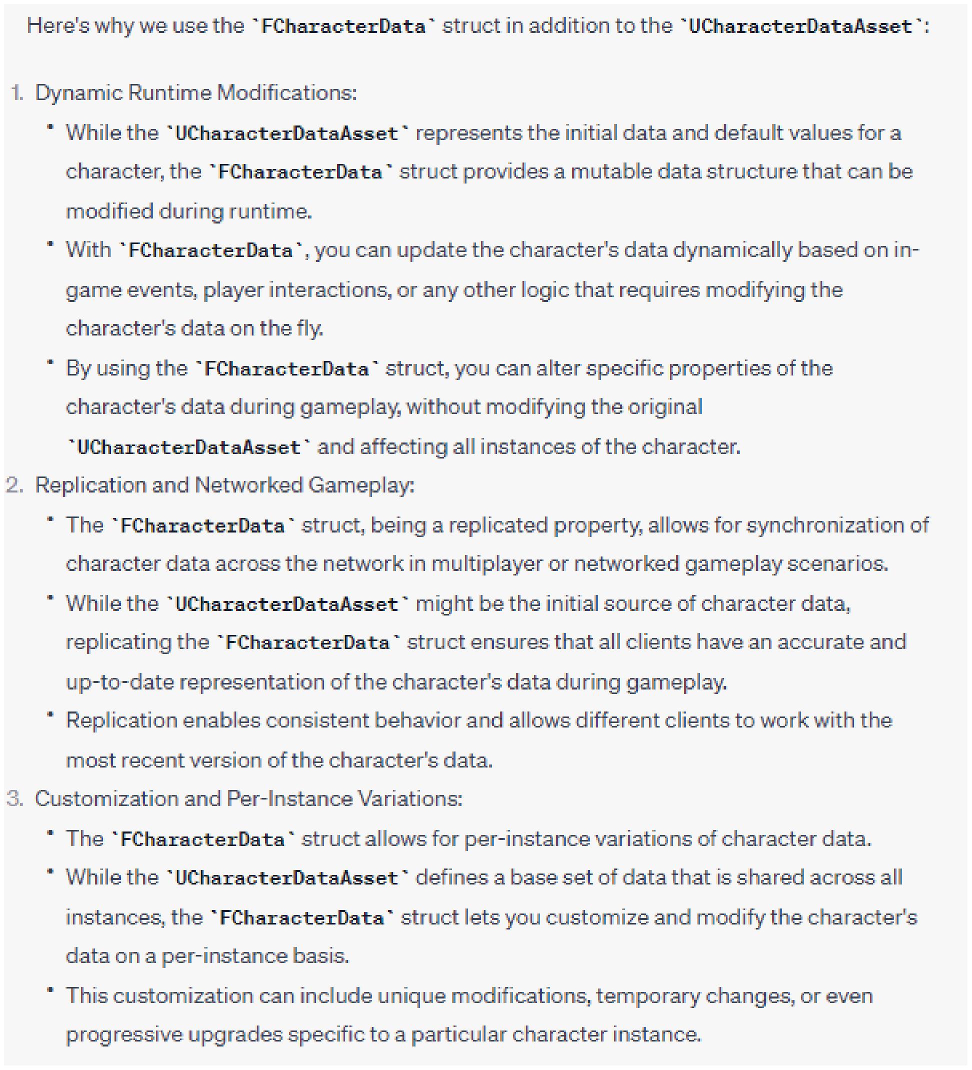
During the PostInitializeComponents phase, the code checks if the CharacterDataAsset is valid and, if so, retrieves the CharacterData from the asset and assigns it to the CharacterData property of the character instance. This setup allows for character-specific data to be defined in a data asset and assigned to individual character instances during initialization.



```cpp
FCharacterData AActionGameCharacter::GetCharacaterData() const
{
    return CharacterData;
}

void AActionGameCharacter::SetCharacaterData(const FCharacterData& InCharacterData)
{
    CharacterData = InCharacterData;
    InitFromCharacterData(CharacterData);
}

void AActionGameCharacter::OnRep_CharacterData()
{
    InitFromCharacterData(CharacterData, true);
}

void AActionGameCharacter::InitFromCharacterData(const FCharacterData& InCharacterData, bool bFromReplication)
{

}

void AActionGameCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME(AActionGameCharacter, CharacterData);
}
```

5. One important function is GetLifetimeReplicatedProps, if you have a replicated property you have to add it you have to add it inside of this function with specify macro

By using the DOREPLIFETIME **macro** for the CharacterData property, Unreal Engine's replication system is informed that this property should be replicated. When the property changes on the server, the updated value is automatically sent to all connected clients, ensuring that all instances of the **AActionGameCharacter** class have the same value for CharacterData. This allows for consistent gameplay and networked interactions involving the CharacterData property.



Here's why we use the `FCharacterData` struct in addition to the `UCharacterDataAsset`:

1. Dynamic Runtime Modifications:
   - While the `UCharacterDataAsset` represents the initial data and default values for a character, the `FCharacterData` struct provides a mutable data structure that can be modified during runtime.
   - With `FCharacterData`, you can update the character's data dynamically based on in-game events, player interactions, or any other logic that requires modifying the character's data on the fly.
   - By using the `FCharacterData` struct, you can alter specific properties of the character's data during gameplay, without modifying the original `UCharacterDataAsset` and affecting all instances of the character.
2. Replication and Networked Gameplay:
   - The `FCharacterData` struct, being a replicated property, allows for synchronization of character data across the network in multiplayer or networked gameplay scenarios.
   - While the `UCharacterDataAsset` might be the initial source of character data, replicating the `FCharacterData` struct ensures that all clients have an accurate and up-to-date representation of the character's data during gameplay.
   - Replication enables consistent behavior and allows different clients to work with the most recent version of the character's data.
3. Customization and Per-Instance Variations:
   - The `FCharacterData` struct allows for per-instance variations of character data.
   - While the `UCharacterDataAsset` defines a base set of data that is shared across all instances, the `FCharacterData` struct lets you customize and modify the character's data on a per-instance basis.
   - This customization can include unique modifications, temporary changes, or even progressive upgrades specific to a particular character instance.