UE 4 함수 기초

# 액터(Actor)

|  |
| --- |
| #include "GameFramework/Actor.h"  #include "MyActor.generated.h"  UCLASS()  **class** AMyActor : **public** AActor  {  GENERATED\_BODY()  **public**:  // 이 액터의 프로퍼티에 기본값을 설정합니다.  AMyActor();  // 매 프레임 호출됩니다.  **virtual** **void** Tick( **float** DeltaSeconds ) **override**;  **protected**:  // 게임 시작 또는 스폰시 호출됩니다.  **virtual** **void** BeginPlay() **override**;  }; |
| 액터 클래스를 기본 생성한 경우 |

BeginPlay(): 액터가 플레이 가능한 상태로 게임에 들어옴을 알리는 이벤트

Tick(): 액터가 게임에 들어온 이후 경과된 시간만큼 프레임 당 한 번씩 호출된다.  
- 반복 로직도 가능함  
- 틱 함수 기능이 불필요한 경우 제거하는 것이 퍼포먼스 향상에 좋다  
- 제거 시 생성자에서도 틱이 일어난다고 지정한 줄도 제거한다.

|  |
| --- |
| AMyActor::AMyActor()  {  // 이 액터가 Tick() 을 매 프레임 호출하도록 설정합니다. 필요치 않은 경우 이 옵션을 끄면 퍼포먼스가 향상됩니다.  PrimaryActorTick.bCanEverTick = **true**;  } |
| PrimaryActorTick.bCanEverTick = true;를 제거해야 Tick이 완전히 제거되는 것 |

## UPROPERTY 지정자

프로퍼티를 에디터 상에 노출시키는 작업을 수행하는 지정자

|  |
| --- |
| UPROPERTY(**[1]:EditAnywhere**) int32 TotalDamage; |
| [1]: 에디터에서 값 편집 허용을 위한 기본 지정자 |

|  |
| --- |
| UPROPERTY(EditAnywhere, **[1]:Category="Damage"**) int32 TotalDamage; |
| Category=”카테고리명”: 다른 유사 프로퍼티와 한 섹션에 나타날 수 있게 묶어준다 |

|  |
| --- |
| UPROPERTY([1]EditAnywhere, **[2]:BlueprintReadWrite**, [3]Category="Damage") int32 TotalDamage; |
| [2]: 블루프린트 상에서 프로퍼티를 노출시키는 용도의 지정자  - BlueprintReadWrite: 블루프린트 상에서 읽고 쓰기가 가능하다  - BlueprintReadOnly: 블루프린트 상에서 프로퍼티를 Const 취급 |

- 추가 프로퍼티  
<https://docs.unrealengine.com/4.27/ko/ProgrammingAndScripting/GameplayArchitecture/Properties/Specifiers/>

|  |
| --- |
| UPROPERTY(BlueprintReadOnly, **[2]:VisibleAnywhere, [3]:Transient,** Category="Damage") **float** DamagePerSecond; |
| [2]: VisibleAnywhere: 프로퍼티가 보이기는 하지만 언리얼 상에서 수정이 불가능하도록 마크  [3]: Transient: 디스크에 저장 또는 로드되지 않는다(파생 값이라 저장이 불필요하다)고 마크 |

## 생성자: 기본값 설정

|  |
| --- |
| AMyActor::AMyActor()  {  TotalDamage = 200;  DamageTimeInSeconds = 1.0f;  }  AMyActor::AMyActor() :  TotalDamage(200),  DamageTimeInSeconds(1.0f)  {} |
| 생성자에서 프로퍼티의 기본 값을 설정하는 것은 C++ 클래스와 동일함  - 위는 생성 후 초기화 - 아래는 생성 전 초기화(초기화리스트 사용) |
| 프로퍼티에 기본 값이 지정되지 않으면 엔진이 자동으로 0 또는 NULL로 설정한다. |

## PostInitProperties() 콜 체인

|  |
| --- |
| **void** AMyActor::PostInitProperties()  {  Super::PostInitProperties();  DamagePerSecond = TotalDamage / DamageTimeInSeconds;  } |
| 주어진 오브젝트에 대해서 인스턴스 데이터에서 값을 로드할 수 있는 방법  - 디자이너 설정 값을 기반으로 기본 값을 만들 수 있음 - 생성자 이후 동작한다. |

## 파생된 값의 계산에 필요한 후크 추가

|  |
| --- |
| **void** AMyActor::PostInitProperties()  {  Super::PostInitProperties();  CalculateValues();  }  **void** AMyActor::CalculateValues()  {  DamagePerSecond = TotalDamage / DamageTimeInSeconds;  }  **#if WITH\_EDITOR**  **void** AMyActor::PostEditChangeProperty(FPropertyChangedEvent& PropertyChangedEvent)  {  CalculateValues();  Super::PostEditChangeProperty(PropertyChangedEvent);  }  **#endif** |
| 로딩 프로세스에 프로퍼티가 초기화된 이후의 초당 대미지 값만 계산한다 - 언리얼 에디터의 런타임 변경사항은 반영되지 않음 = 후크 추가 필요  - 게임에 필요한 코드로만 게임을 만들 수 있도록한다 |

## C++과 블루프린트 경계를 넘는 함수 호출

|  |
| --- |
| UFUNCTION(**[1]:BlueprintCallable**, Category="Damage") **void** CalculateValues(); |
| BlueprintCallable: 블루프린트 가상 머신 상에 노출시키는 매개변수 - 카테고리가 할당되어야 우클릭 컨텍스트 메뉴가 정상 동작한다 |

- 엔진의 많은 부분이 UFUNCTION() 매크로를 통해 블루프린트에 노출되어 있다  
- C++ 코드 없이 게임을 만들 수 있는 이유  
- 기반이 되는 게임 플레이 시스템과 성능이 중요한 코드는 C++로, C++ 기본 구성요소를 토대로 복합적인 작동방식을 만들거나 수정하는 것은 블루프린트로 하는 것이 최적의 접근법

|  |
| --- |
| UFUNCTION(**[1]:BlueprintImplementableEvent**, Category="Damage") **void** CalledFromCpp(); |
| [1]: C++ 코드에서 블루프린트에 정의된 함수를 호출할 수 있도록 해주는 매개변수 |

### THUNK(썽크)

언리얼 엔진이 블루프린트 가상 머신으로 호출해 들어가는 방법을 이해하는 베이스 C++ 함수를 구현하는 것  
- 해당 블루프린트가 이 메소드에 대해 함수 바디를 제공해주지 않으면 동작이 없는 C++ 함수인 것처럼 동작한다(아무것도 안하는 빈 함수)

블루프린트가 메소드를 덮어쓸 수 있도록 하면서 C++ 기본 구현을 제공하는 방법이 존재함

|  |
| --- |
| UFUNCTION(**[1]:BlueprintNativeEvent**, Category="Damage") **void** CalledFromCpp(); |
| [1]: C++ 코드에서 블루프린트에 정의된 함수를 호출할 수 있도록 해주는 매개변수 |
| **void** AMyActor::CalledFromCpp\_Implementation()  { // 여기서 어떤 작업을 해라 } |
| 위의 방법도 여전히 Thunk를 생성하는데 보이는 함수 선언을 새로 생성하는 툴이 제공된다 - 이 버전의 함수를 제공해주지 않으면 프로젝트가 링크하는데 실패한다 - 과거에는 함수명\_Implementation() 선언이 자동으로 생성되었으나 현재 버전에서는 안 되므로 명시적으로 생성해줘야 한다  - 이 버전의 함수가 호출되면 블루프린트는 메소드를 덮어쓰지 않고 동작한다. |

# 언리얼 오브젝트(UObject)

언리얼 엔진의 기본 구성요소  
- UClass와 함께 엔진의 가장 중요한 근간이 되는 서비스 제공

|  |
| --- |
| - 프로퍼티와 메서드의 리플렉션(반영)  - 프로퍼티의 시리얼라이제이션  - 가비지 컬렉션  - 이름으로 UObject 찾기  - 프로퍼티에 환경설정 가능 값  - 프로퍼티와 메서드에 네트워킹 지원 |

- UObject에서 파생되는 각 클래스에는 그에 대한 단독 개체인 UClass가 생성되어 클래스 인스턴스에 대한 모든 메타 데이터가 포함된다

UClass와 UObject의 차이점  
- UClass는 인스턴스가 어떤 모양인지, 어떤 프로퍼티가 시리얼라이제이션(어떤 환경의 데이터 구조를 다른 환경에 전송 및 저장하기 위해 나중에 재구성할 수 있는 바이트의 포맷으로 변환하는 과정), 네트워크 대상인지 등을 설명한다  
- 대부분의 개발 상황에서 UObject를 직접 파생하기 보다는 AActor, UActorComponent를 파생하게 된다.  
- 게임 플레이 코드를 작성하기 위해서 UClass/UObject 작동 방식을 알아야 할 필요는 없다  
- 이런 것이 있다는 정도

## AActor

AActor(액터)는 게임 플레이 경험의 일부로 쓸 오브젝트  
- 디자이너가 레벨에 배치  
- 게임 플레이 시스템을 통해 런타임에 생성 가능

**\* 레벨에 배치할 수 있는 오브젝트는 전부 이 클래스에서 확장된다**-> AStaticMeshActor, ACameraActor, APointLight 액터 등

게임플레이 코드(C++ 또는 블루프린트)를 통해서 명시적으로 소멸시키고 소유중인 레벨이 메모리에서 언로드 될 때 표준 가비지 컬렉션 메커니즘을 통해 소멸시킬 수 있다  
- 게임 오브젝트가 하이레벨에서 작동되는 방식을 담당  
- 네트워크 상황에서 리플리케이트(리플리케이션[Replication]: 서버와 클라이언트 사이에서 데이터와 명령을 주고 받는 프로세스)되는 유형

### 액터의 작동방식

- 상속을 통한 전문화

- 구성을 통한 전문화(계층 구조에 대한 컨테이너 역할)  
-> 액터의 루트 컴포넌트 멤버를 통해 이뤄지는데 여기에는 하나의 씬 컴포넌트가 들어가고 차례대로 다른 컴포넌트들이 담긴다  
-> 액터를 레벨에 배치하기 전, 반드시 최소 하나의 씬 컴포넌트가 들어있어야 한다  
-> 액터는 이 곳에서 이동, 회전, 스케일 정보를 얻는다

### 액터의 수명 도중 호출할 수 있는 이벤트

BeginPlay(): 게임 플레이 도중 오브젝트가 처음 등장했을 때 호출

Tick() 지속적으로 이뤄지는 작업을 하기 위해 프레임 당 한 번 호출

EndPlay(): 오브젝트가 게임 플에이 공간을 떠날 때 호출

### 런타임 수명

레벨에 배치된 액터의 경우 수명을 생각하기 쉽다  
- 액터가 등장하면 로드  
- 레벨이 언로드 되면 소멸

액터 스폰의 경우 게임에서 일반 오브젝트를 생성하는 것보다 좀 더 복잡하다  
- 액터는 다양한 런타임 시스템으로 등록해야 요구사항을 전부 충족시킬 수 있다  
- 액터의 초기 위치, 회전 설정(피직스 정보)  
- 액터에게 틱 요청을 담당하는 매니저도 알아야 한다

그래서 액터의 스폰을 전담하는 메소드를 따로 만듦  
- SpawnActor() 함수

액터 스폰에 성공하면 BeginPlay() 메소드가 호출되고 다음 프레임에 Tick이 잇따른다

## SpawnActor() 메소드

액터의 새로운 인스턴스를 생성하는 과정을 스폰이라고 한다  
- 액터의 스폰은 UWorld::SpawnActor() 함수를 이용한다  
- 함수는 지정된 클래스의 새 인스턴스를 생성한 다음 새로 생성된 액터로의 포인터를 반환한다  
- SpawnActor() 메소드는 클래스 계층 구조 안에 Actor 클래스를 상속하는 클래스 인스턴스를 생성할 때만 사용해야 한다.

|  |
| --- |
| AActor\* UWorld::SpawnActor  (  UClass\* Class,  FName InName,  FVector **const**\* Location,  FRotator **const**\* Rotation,  AActor\* Template,  **bool** bNoCollisionFail,  **bool** bRemoteOwned,  AActor\* Owner,  APawn\* Instigator,  **bool** bNoFail,  ULevel\* OverrideLevel,  **bool** bDeferConstruction  ) |
| 기본 구조 |

### 메소드 멤버 함수

|  |  |
| --- | --- |
| Class | 스폰시킬 액터의 클래스를 나타내는 UClass |
| InName | 스폰시킬 액터의 이름으로 할당시킬 FName(옵션)  - 지정된 값이 없다면 스폰되는 액터의 이름은 Class\_Number 형식으로 자동 생성됨 |
| Location | 액터를 스폰시킬 초기 위치를 제공하는 FVector(옵션) |
| Rotation | 액터를 스폰시킬 초기 방향을 제공하는 FRotator(옵션) |
| Template | 새 액터를 스폰할 때 템플릿으로 사용할 AActor(옵션)  - 스폰되는 액터는 템플릿 액터의 프로퍼티 값을 사용하여 초기화  - 템플릿 액터가 없는 경우 CDO를 이용하여 스폰되는 액터를 초기화 |
| bNoCollisionFail | 액터 스폰 시 콜리전 테스트를 할 지 결정하는 논리값(옵션)  - True면 액터 스폰 시 루트 컴포넌트나 템플릿 액터의 콜리전 세팅에 관계없이 콜리전 테스트를 하지 않는다 |
| bRemoteOwned | 논리 값(옵션) |
| Owner | 스폰된 액터를 소유하는 액터(옵션) |
| Instigator | 스폰된 액터가 입힌 피해를 담당하는 APawn(옵션) |
| bNoFail | 특정 조건을 충족하지 않을 때 스폰 실패 여부를 결정하는 논리 값 - True: 스폰되는 클래스가 bStatic=true, 템플릿 액터의 클래스가 스폰되는 액터의 클래스와 같지 않는 등의 이유로 스폰이 실패하지 않는다 |
| OverrideLevel | 액터를 어느 ULevel안에 스폰시킬 지 액터의 Outer 식  - 지정된 레벨의 없으면 Owner의 Outer가 사용된다  - 지정된 Outer가 없으면 퍼시스턴트 레벨이 사용된다 |
| bDeferConstruction | construction script를 실행시킬지 결정하는 논리 값  - True: 액터에서 construction script가 실행되지 않는다  - 액터가 블루프린트에서 스폰될 때만 적용 가능하다 |

### 메소드 반환 값

AActor 포인터 형태로 스폰되는 액터  
- 반환 값은 Class 파라미터로 지정된 파생형으로의 변환을 위해 형변환이 필요함

|  |
| --- |
| AKAsset\* SpawnedActor1 = (AKAsset\*) GetWorld()->  SpawnActor(AKAsset::StaticClass(), NAME\_None, &Location); |

### 스폰 함수 템플릿

액터 스폰을 쉽게 하기 위해 가장 흔히 사용되는 패턴에 대한 것  
- 여러가지 함수 템플릿이 제공되어 있다  
- 파라미터의 일부 부분집합만 필요하고 반환되는 액터 유형을 지정할 수 있어 액터 생성이 훨씬 단순해진다.

#### T 인스턴스 스폰과 T 포인터 반환

스폰 작업을 수행하는 액터의 루트 컴포넌트와 같은 위치, 같은 방향에 T 템플릿 클래스의 인스턴스를 스폰시키는 함수 템플릿으로 템플릿 클래스와 같은 유형의 해당 인스턴스 포인터(T\*)를 반환한다  
- 소유하는 액터, 선동하는 폰, 스폰되는 액터가 칩입할 때 스폰 작업 실패 여부, 이미 월드에 존재하는 다른 액터와의 충돌 여부 등을 지정할 수 있다.

|  |
| --- |
| /\*\* 클래스 T 를 스폰하고 반환, 루트 컴포넌트의 기본 방향과 위치를 따릅니다. \*/  **template**< **class** T >  T\* SpawnActor  (  AActor\* Owner=NULL,  APawn\* Instigator=NULL,  **bool** bNoCollisionFail=**false**  )  {  **return** (T\*)(GetWorld()->SpawnActor(T::StaticClass(), NAME\_None, NULL, NULL, NULL, bNoCollisionFail, **false**, Owner, Instigator));  } |
| 기본 형태 |

|  |
| --- |
| MyHUD = SpawnActor<AHUD>(**this**, Instigator); |
| 기본적인 사용 예시 |

#### 트랜스폼 정보로 T 인스턴스 스폰과 T 포인터 반환

지정된 Location과 지정된 Rotation에 템플릿 클래스 T의 인스턴스를 스폰시키는 함수 템플릿  
- 템플릿 클래스와 같은 유형의 해당 인스턴스로의 포인터(T\*)를 반환한다  
- 위치와 방향에 대해 소유중인 액터, 선동중인 폰, 스폰되는 액터 침입에 따른 스폰 작업 실패 여부, 월드에 이미 존재하는 다른 액터와의 충돌 여부 등을 지정할 수 있다.

|  |
| --- |
| /\*\* 클래스 T 스폰 후 반환, 월드 포지션은 강제 설정합니다. \*/  **template**< **class** T >  T\* SpawnActor  (  FVector **const**& Location,  FRotator **const**& Rotation,  AActor\* Owner=NULL,  APawn\* Instigator=NULL,  **bool** bNoCollisionFail=**false**  )  {  **return** (T\*)(GetWorld()->SpawnActor(T::StaticClass(), NAME\_None, &Location, &Rotation, NULL, bNoCollisionFail, **false**, Owner, Instigator));  } |
| 기본 형식 |

|  |
| --- |
| Controller = SpawnActor<AController>(GetLocation(), GetRotation(), NULL, Instigator, **true**); |
| 기본 사용 예시 |

#### 클래스 인스턴스 스폰과 T 포인터 반환

스폰 작업을 수행하는 액터의 루트 컴포넌트와 같은 위치, 같은 방향에 지정된 클래스의 인스턴스를 스폰시키는 함수 템플릿  
- 템플릿 클래스 유형으로 형변환 된 해당 인스턴스로의 포인터(T\*)를 반환한다  
- 여기에 지정된 클래스는 템플릿 클래스 T의 자손이어야 한다  
- 클래스에 추가로 소유중인 액터, 선동중인 폰, 스폰된 액터의 침범에 따른 스폰 작업 실패 여부, 월드에 이미 존재하는 다른 액터와의 충돌 여부 등을 지정할 수 있다.

|  |
| --- |
| /\*\* 주어진 클래스를 스폰하고 클래스 T 포인터를 반환, 루트 컴포넌트의 기본 방향과 위치를 따릅니다. \*/  **template**< **class** T >  T\* SpawnActor  (  UClass\* Class,  AActor\* Owner=NULL,  APawn\* Instigator=NULL,  **bool** bNoCollisionFail=**false**  )  {  **return** (Class != NULL) ? Cast<T>(GetWorld()->SpawnActor(Class, NAME\_None, NULL, NULL, NULL, bNoCollisionFail, **false**, Owner, Instigator)) : NULL;  } |
| 기본 형식 |

|  |
| --- |
| MyHUD = SpawnActor<AHUD>(NewHUDClass, **this**, Instigator); |
| 기본 사용 예시 |

#### 트랜스폼 정보로 클래스 인스턴스 스폰과 T 포인터 반환

지정된 Location과 Rotation에 지정된 클래스 인스턴스를 스폰시키는 함수 템플릿  
- 템플릿 클래스와 같은 유형의 해당 인스턴스의 포인터(T\*)을 반환한다  
- 지정된 클래스는 템플릿 클래스 T의 자손이어야 한다  
- 클래스, 위치, 방향에 추가로 소유중인 액터, 선동중인 폰, 스폰되는 액터 침입에 따른 스폰 작업 실패 여부, 월드에 이미 존재하는 다른 액터와의 충돌 여부 등을 지정할 수 있다.

|  |
| --- |
| /\*\* 주어진 클래스를 스폰하고 클래스 T 포인터를 반환, 월드 포지션은 강제 설정합니다. \*/  **template**< **class** T >  T\* SpawnActor  (  UClass\* Class,  FVector **const**& Location,  FRotator **const**& Rotation,  AActor\* Owner=NULL,  APawn\* Instigator=NULL,  **bool** bNoCollisionFail=**false**  )  {  **return** (Class != NULL) ? Cast<T>(GetWorld()->SpawnActor(Class, NAME\_None, &Location, &Rotation, NULL, bNoCollisionFail, **false**, Owner, Instigator)) : NULL;  } |
| 기본 형태 |

|  |
| --- |
| APawn\* ResultPawn = SpawnActor<APawn>(DefaultPawnClass, StartLocation, StartRotation, NULL, Instigator); |
| 기본 사용 예시 |

## UActorComponent

액터 컴포넌트는 별도의 작동 방식이 내장되어 있고 비주얼 메시, 파티클 이펙트, 카메라 시점, 피직스 상호작용 등 AActor의 여러 유형에 공유되는 기능을 담당한다  
- AActor는 보통 게임의 전반적인 규칙에 관계된 일반적인 수준의 목표가 주어진다  
- UActorComponent는 보통 일반적인 수준의 목표를 보조하는 개별 작업 수행

컴포넌트는 다른 컴포넌트에 붙일 수 있고 엑터의 루트 컴포넌트가 될 수도 있다.  
- 컴포넌트는 오직 하나의 부모 컴포넌트 또는 액터에만 붙일 수 있다  
- 자신에게는 다수의 자손 컴포넌트를 붙일 수 있다.  
- 트리 구조를 갖게 되는 것  
- 자손 컴포넌트는 부모 컴포넌트를 기준으로 한 위치, 회전, 스케일 값을 갖는다.

- 액터 = 이게 뭐지  
- 컴포넌트 = 이게 뭘로 만들어졌지

### 루트 컴포넌트

액터의 컴포넌트 트리 내 최상위 컴포넌트를 담는 AActor 멤버

### Ticking

- 소유한 액터의 Tick 함수 일부분으로 실행되는 컴포넌트  
- 자체 Tick 함수 작성 시 반드시 Super::Tick을 호출해야 한다

## UStruct

구조체 역할로 특정 클래스를 확장할 필요 없이 구조체인 UStruct() 마킹을 해주면 빌드 툴이 알아서 작업을 해준다  
- UObejct와는 다르게 가비지 컬렉션 대상이 아니다  
- 동적 인스턴스를 생성한 경우 그 수명 관리를 직접 해야 한다

언리얼 에디터 내 편집, 블루프린트 조작, 시리얼라이제이션, 네트워크 등 UObject 리플렉션이 지원되는 일반적인 이전 데이터 유형들을 모아놓는 용도로 쓴다.

# 델리게이트(Delegate)

함수를 바인딩하는 형태로 등록, Callback 함수와 함께 활용할 수 있게 된다.

델리게이트의 종류

|  |  |
| --- | --- |
| 싱글 | 가장 기본적인 델리게이트, 함수 1개를 바인드하여 사용 |
| 멀티 | 여러 함수를 바인드할 수 있다. |
| 이벤트 | 멀티 캐스트와 동일하지만 전역 설정이 불가 = 외부 클래스에서 추가 델리케이트 선언 불가 |
| 다이나믹 | 싱글, 멀티 둘 다 존재하여 직렬화되어 블루프린트에서 사용 가능 |

바인딩: 델리게이트에 특정 바인드 함수를 통해 CallBack 함수를 등록하는 것

|  |  |
| --- | --- |
| 함수 | 설명 |
| Bind() | 기존 델리게이트 오브젝트에 바인딩 |
| BindStatic() | Raw C++ 포인터 글로벌 함수 델리게이트를 바인딩 |
| BindRaw() | Raw C++ 포인터 델리게이트에 바인딩  - Raw 포인터는 어떠한 종류의 레퍼런스도 허용하지 않는다  - 오브젝트가 덜리게이트에서 삭제된 경우 호출하는 것이 불안정함  - 주의가 필요함 |
| BindSP() | 공유 포인터 기반 멤버 함수 델리게이트에 바인딩  - 공유 포인터 델리게이트는 오브젝트로의 약한 레퍼런스를 유지한다.  - ExecuteIfBound()로 호출 |
| BindUObject() | UObject 기반 멤버 함수 델리게이트를 바인딩  - UObject 델리게이트는 오브젝트로의 약한 레퍼런스 유지  - ExecuteIfBound()로 호출 |
| UnBind() | 해당 델리게이트 바인딩을 해제한다. |

## 싱글 캐스트 델리게이트

### 헤더

|  |
| --- |
| DECLARE\_DELEGATE(FDele\_Single);  DECLARE\_DELEGATE\_OneParam(FDele\_Single\_OneParam, int32);  UCLASS()  class 프로젝트 명\_API 클래스 명 : public AActor(또는 상속할 클래스)  {  GENERATED\_BODY()  public:  FDele\_Single Fuc\_DeleSingle;  FDele\_Single\_OneParam Fuc\_DeleSingle\_OneParam;  }; |

- 헤더 파일에서 Delegate 함수를 만들고 DECLARE\_DELEGATE() 매크로를 통해 델리게이트 화 시킨다.

- DECLARE\_DELEGATE: 인자 값이 없는 함수에 사용  
- DECLARE\_DELEGATE\_OneParam: 1개의 인자 값이 있는 함수에 사용  
- DECLARE\_DELEGATE\_TwoParam: 2개의 인자 값이 있는 함수에 사용

\* 델리게이트는 자료형 이름 앞에 F를 붙여야 한다  
- 안그러면 빌드 또는 컴파일 시에 에러가 난다.

### CPP

|  |
| --- |
| void 클래스명::함수명(~)  {  ~  Fuc\_DeleSingle.Unbind();  Fuc\_DeleSingle\_OneParam.Unbind();  } |

델리게이트 해제  
- Unbind()는 일종의 메모리 해제 함수로 생각할 수 있다  
- 해당 델리게이트에 바인드 된 함수를 제거한다

|  |
| --- |
| void 클래스명::함수명(~)  {  ~  if(Fuc\_DeleSingle.IsBound() == true)  Fuc\_DeleSingle.Execute();  if(Fuc\_DeleSingle\_OneParam.IsBound() == true)  Fuc\_DeleSingle\_OneParam.Execute(123);  } |

델리게이트에 등록된 Callback 함수 호출  
- IsBound()를 통해 바인드 된지 여부를 먼저 확인한다  
- 바인드 되어 있으면 Execute() 함수를 통해 호출한다.

### 함수 등록

#### 헤더

|  |
| --- |
| UCLASS()  class 프로젝트명 클래스명 : 상속받을 클래스명  {  GENERATED\_BODY()  public:  UFUNCTION() void CallDeleFunc\_Single();  UFUNCTION() void CallDeleFunc\_Single\_OneParam(int32 nValue);  } |

함수 등록 과정  
- Delegate에 등록할 함수는 UFUNCTION() 매크로를 붙여야 한다

#### CPP

|  |
| --- |
| void 클래스명::CallDeleFunc\_Single() (델리게이트 명)  { UE\_LOG(LogTemp, Warning, TEXT(“CallDeleFunc\_Single)); }  void 클래스명::CallDeleFunc\_Single\_One\_Param(int32 nValue) (델리게이트 명(인자))  { UE\_LOG(LogTemp, Warning, TEXT(“CallDeleFunc\_Single\_OneParam / %d”), nValue); } |

델리게이트로 함수 호출  
- 등록이 선행되어야 한다

|  |
| --- |
| void 클래스명::함수명(주로 BeginPlay() 등)  {  m\_pBoom->Fuc\_DeleSingle.BindUFunction(this, FName(“CallDeleFunc\_Single));  바인드할 변수->변수의 델리게이트.BindUFunction(this, FName(“현재 클래스의 델리게이트로 호출할 함수 명));  } |

함수를 델리게이트에 바인딩 하는 과정  
- 플레이어 클래스에서 무기 클래스에서 만든 변수에 접근하여 함수를 바인드하는 함수를 통해 함수를 등록하는 것 등이 가능함

## 멀티 캐스트 델리게이트

싱글 캐스트와의 차이점은 함수를 여러 개 바인드 할 수 있다는 것

### 헤더

|  |
| --- |
| DECLARE\_MULTICAST\_DELEGATE(FDele\_Multi);  DECLARE\_MULTICATE\_DELEGATE\_OneParam(FDele\_Multi\_OneParam, int32);  UCLASS()  class 프로젝트 명 클래스 명 : 상속받을 클래스 명  {  GENERATED\_BODY()  public:  FDele\_Multi Fuc\_DeleMulti;  FDele\_Multi\_OneParam Fuc\_DeleMulti\_OneParam;  }; |

- 멀티캐스트도 헤더 파일에서 작성한다

### CPP

|  |
| --- |
| void 클래스명::함수명(~)  {  ~  Fuc\_DeleMulti.Clear();  Fuc\_DeleMulti\_OneParam.Clear();  } |

- 델리게이트 해제  
- 싱글 캐스트는 Unbind() / 멀티 캐스트는 Clear()

|  |
| --- |
| void 클래스명::함수명(~)  {  if(Fuc\_DeleMulti.IsBound() == true) Fuc\_DeleMulti.Broadcast();  if(Fuc\_DeleMulti\_OneParam.IsBound() == true) Fuc\_DeleMulti\_OneParam.Boradcast(456);  } |

- 델리게이트 호출  
- 싱글 캐스트는 Execute() / 멀티 캐스트는 Broadcast()

### 함수 등록

#### 헤더

|  |
| --- |
| UCLASS()  class 프로젝트명 클래스명 : 상속받을 클래스명  {  GENERATED\_BODY()  public:  UFUNCTION() void CallDeleFunc\_Multi\_1();  UFUNCTION() void CallDeleFunc\_Multi\_2();  UFUNCTION() void CallDeleFunc\_Multi\_OneParam\_1(int32 nValue);  UFUNCTION() void CallDeleFunc\_Multi\_OneParam\_2(int32 nValue);  } |

- 멀티 캐스트는 함수를 여러 개 등록할 수 있다.

#### CPP

|  |
| --- |
| void 클래스 명::함수 명  {  ~  m\_pBoom->Fuc\_DeleMulti.AddFunction(this, FName(“CallDeleFunc\_Multi\_1));  m\_pBoom->Fuc\_DeleMulti.AddFunction(this, FName(“CallDeleFunc\_Multi\_2));  바인드할 변수 명->변수의 델리게이트.AddFuction(this, FName(“현재 클래스의 델리게이트 명));  } |

- 싱글 캐스트에서는 BindUFunction 함수를 사용하였다면 멀티 캐스트는 AddUFunction을 사용한다.

## 이벤트 델리게이트

멀티 캐스트와 유사한 개념이지만 전역으로 설정이 불가능함  
- 외부에서 추가 델리게이트 선언이 불가능함  
- 캡슐화와 유사한 개념으로 사용하면 된다고 함

|  |
| --- |
| UCLASS()  class 프로젝트 명 클래스 명 : 상속받을 클래스 명 {  GENERATED\_BODY()  public:  DECLARE\_EVENT(ABoom, FDele\_Event);  DECLARE\_EVENT\_OneParam(ABoom, FDele\_Event\_OneParam, int32);  public:  FDele\_Event Fuc\_DeleEvent;  FDele\_Event\_OneParam Fuc\_DeleEvent\_OneParam;  } |

- 싱글, 멀티 캐스트의 경우 델리게이트를 전역 위치에 작성했지만 이벤트는 클래스 내부에 작성한다  
- 인자 값으로 주인 클래스의 클래스를 입력하여 위치를 알린다  
- 나머지는 멀티캐스트와 동일하다

## 다이나믹 델리게이트

|  |
| --- |
| DECLARE\_DYNAMIC\_MULTICAST\_DELEGATE(FDele\_Dynamic);  DECLARE\_DYNAMIC\_MULTICATE\_DELEGATE\_OneParam(FDele\_Dynamic\_OneParam, int32, SomeParameter);  UCLASS()  class 프로젝트 명 클래스 명 : 상속받을 클래스 명  {  GENERATED\_BODY()  public:  UPROPERTY(BlueprintAssignable, VisibleAnywhere, BlueprintCallable, Category = “Event”)  FDele\_Dynamic Fuc\_Dynamic;  UPROPERTY(BlueprintAssignable, VisibleAnywhere, BlueprintCallable, Category = “Event”)  FDele\_Dynamic\_OneParam Fuc\_Dynamic\_OneParam;  } |

- BlueprintAssignable 등의 매크로를 추가로 작성해야 한다  
- 블루프린트에서 델리게이트 바인드가 가능해진다  
- 블루프린트에서 델리게이트를 사용할 시 많은 오류가 많이 나타난다  
- 수정하는 방법을 모르겠다.

# 인터페이스와 노티파이

서로 다른 클래스끼리 통신을 할 때 각 클래스에 인터페이스 클래스를 상속시켜 쉽게 클래스 사이의 통신을 구현할 수 있다.

# C++과 블루프린트 연동

## Reflection(리플렉션)

프로그램의 런타임 시점에서 객체의 정보, 구조를 관리하고 수정할 수 있게 액세스하는 것을 의미한다.  
- C++에는 없지만 언리얼 C++에는 자체적으로 구현되어 있다  
- 언리얼 프로퍼티 시스템이라고 하며 매크로 함수 형태로 되어 있다.  
- 이것을 통해 언리얼 C++과 블루프린트의 연동 등을 할 수 있다

## 언리얼 매크로 함수

UENUM(), UCLASS(), USTRUCT(), UFUNCTION(), UPROPERTY() 등 다양한 매크로 함수가 존재함  
- 클래스, 변수, 함수 등 상황에 맞게 리플렉션을 위한 매크로 함수로 작성한다

### UPROPERTY() = 변수

변수에 붙는 리플렉션 매크로

|  |
| --- |
| UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = “true”))  class UCameraComponent\* m\_pCamera; |

- 인자 값 순서는 특별하게 정해진 것이 없다

#### 1: 변수 공개와 수정 권한

|  |  |  |  |
| --- | --- | --- | --- |
|  | Edit | | Visible |
| DefaultsOnly | 블루프린트 에디터 창의 디테일 패널에서 값을 수정할 수 있다 | | 블루프린트 에디터 창의 디테일 패널에서 값을 보는 것이 가능하다 |
| InstanceOnly | 월드 상에 배치된 오브젝트의 디테일 패널에서 값을 수정할 수 있다 | | 월드 상에서 배치된 오브젝트의 디테일 패널에서 값을 보는 것이 가능하다 |
| Anywhere | 어디서든 수정이 가능하다 | | 어디서든 보는 것이 가능하다 |
| VisibleDefaultsOnly  VisibleInstanceOnly  VisibleAnywhere | | EditDefaultsOnly  EditInstanceOnly  EditAnywhere | |

포인터 변수에 사용할 경우  
- 포인터 변수를 만들고 수정할 때 Edit~ 매개변수를 사용할 경우 의도와는 다른 동작이 수행된다  
- 해당 클래스를 수정하는 것이 아닌 포인터 변수의 참조를 수정하게 되는 것  
- 블루프린트 에디터 상에서 참조를 변경하는 메뉴가 뜬다

= 포인터 변수를 만들고 수정할 경우 Visible로 지정하면 의도한 결과와 동일할 것임

#### 2: 블루프린트 공개와 수정 권한

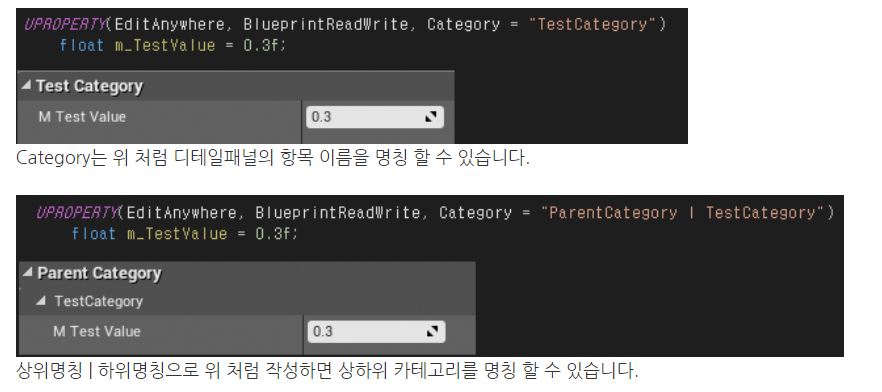
블루프린트와 직접적으로 연관된 부분

|  |  |
| --- | --- |
| BlueprintReadOnly | 블루프린트에서 해당 변수를 읽기가 가능 |
| BlueprintReadWrite | 블루프린트에서 해당 변수를 읽고 쓰는 것이 가능 |
| BlueprintGetter | 해당 변수에 접근할 수 있는 함수를 지정하고 블루프린트는 해당 함수를 통해 변수에 접근한다 |
| BlueprintSetter | 해당 변수에 수정할 수 있는 함수를 지정하고 블루프린트는 해당 함수를 통해 변수에 수정한다 |

- 보이지 않을 경우 상속된 변수 표시를 체크해야 한다

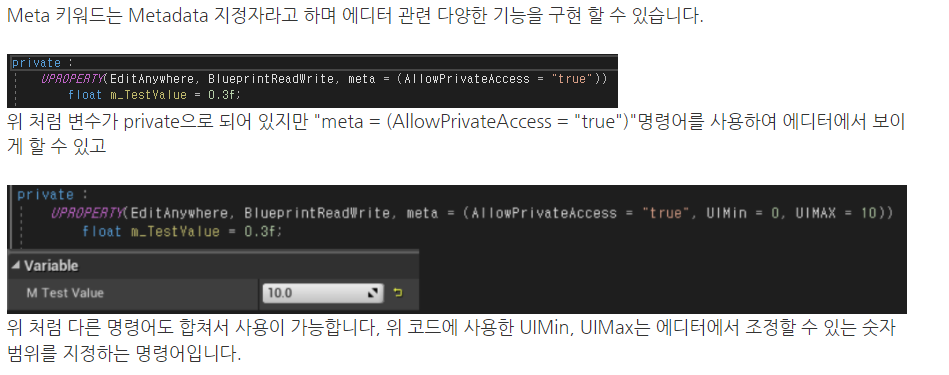
#### 3: 카테고리

블루프린트의 디테일 패널에서 카테고리를 지정하여 명칭할 수 있다.

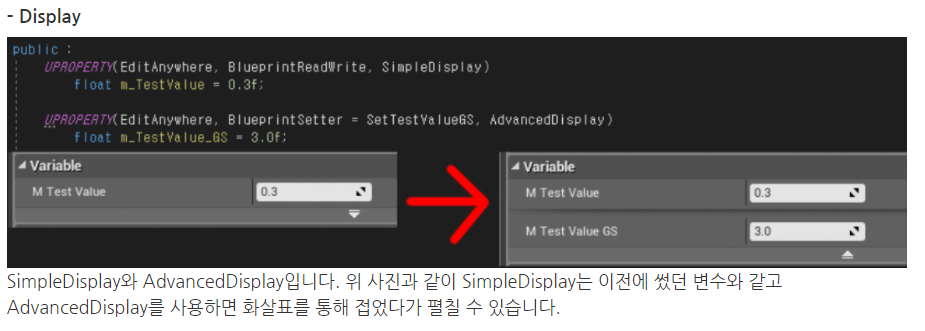


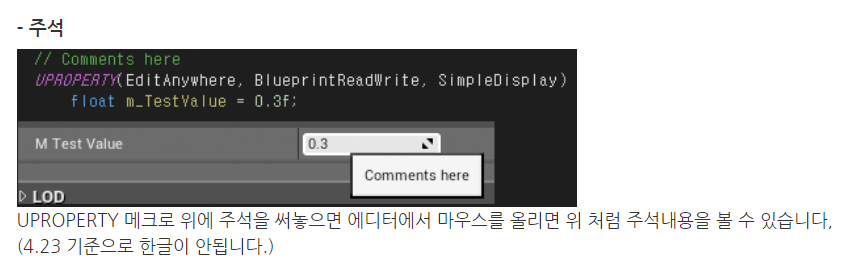
#### 4: 메타(Meta)

메타 데이터 지정자라고 하며 에디터에 관한 다양한 기능을 구현하는데 사용한다



#### 5: 기타





### UFUNCTION() = 함수

C++의 함수를 블루프린트와 연동하는 내용

#### BlueprintCallable

가장 기본적으로 C++에서 작성한 함수를 블루프린트에서 사용할 수 있도록 하는 키워드  
- 간단한 기능이라 블루프린트 내에서 재정의가 불가능함

#### BlueprintPure

매우 간단하게 값을 하나 반환하는 형태에서 사용하는 키워드  
- 블루프린트의 퓨어 함수로 만들어버리는 기능  
- 인자값을 넣는 것도 가능하다

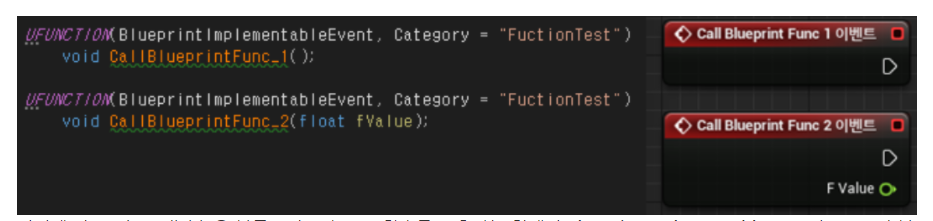
##### UPARAM()

인자값으로 래퍼런스 주소를 입력하려고 할 때 사용하는 매크로  
- 인자 값에 DisplayName 기능을 사용할 수 있다.



#### BlueprintImplementableEvent

C++이 블루프린트 함수를 호출하는 형태  
- C++에서 블루프린트를 사용할 함수 원형을 작성해야 한다



블루프린트에서 함수의 기능을 작성하고 C++ 코드에서 호출하면 동작한다.

##### Custom Event 호출

블루프린트에서 자체적으로 만든 커스텀 이벤트 함수를 C++에서 호출하는 것도 가능하다

|  |  |
| --- | --- |
| 매개변수 없음 | FOutputDeviceNull pAR;  this->CallFunctionByNameWithArguments(TEXT(“이벤트 명”), pAR, nulltpr, true); |
| 매개변수 존재 | FOutputDeviceNull pAR;  this->CallFunctionByNameWithArguments(\*FString::Printf(TEXT(“이벤트 명 %d”), 매개변수 값), pAR, nullptr, true); |
|  | |

#### BlueprintNativeEvent

C++에서 함수 원형을 만드는 것은 동일하지만 가상함수를 만든다는 개념

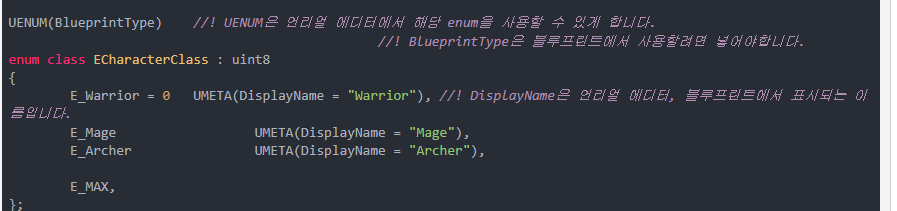
|  |
| --- |
|  |

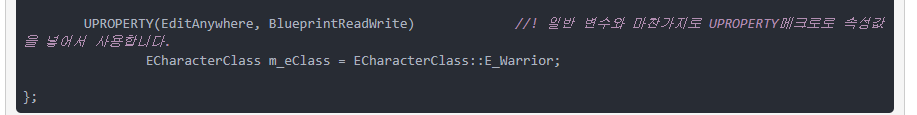
- void CallNativeFunction 함수를 만들고 C++ 에서 가상 함수로 void CallNativeFunction\_Implementation()이라는 함수를 추가로 만든 형태

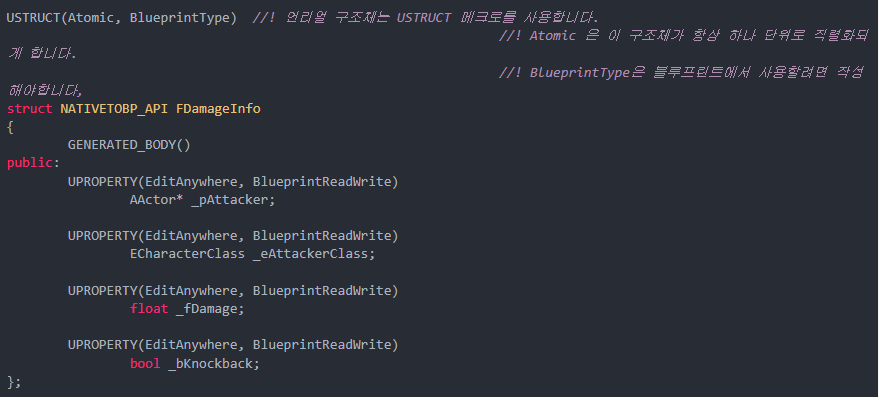
부모 함수 호출 방법은 이벤트 노드에서 우클릭 메뉴에서 생성 가능

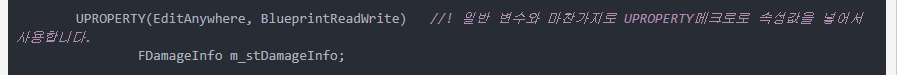
### UENUM(), USTRUCT()

BlueprintType: 에디터에서 해당 열거형을 사용할 수 있도록 한다.

  
- 열거형 내부의 변수에도 메타 데이터 지정을 할 수 있다  
- UMETA(DisplayName = “Warrior) 코드의 UMETA 지정자



  
- 열거형과 동일하게 USTURCT(Atomic, BlueprintType) 매크로를 선행하여 작성한다  
- Atomic: 이 구조체가 항상 하나 단위로 직렬화되도록한다  
- BlueprintType: 구조체를 블루프린트에서 사용하기 위해 작성

  
- 댕글링 포인터 문제(삭제된 오브젝트를 참조하는 문제)를 방지하기 위해 구조체의 이름 앞에 F를 붙여야 한다  
- 구조체에서 Atomic 키워드를 사용한 경우 UPROPERTY 매크로에서 EditAnywhere를 반드시 작성해야 한다.