

Unreal Engine 델리게이트 완벽 가이드 (1차 완본)

📌 델리게이트란?

델리게이트(Delegate)는 C++ 함수를 가리키고 실행할 수 있는 데이터 타입입니다.

쉽게 말하면, **"나중에 실행할 함수를 저장해두는 상자"**라고 생각하면 됩니다.

왜 델리게이트를 사용할까요?

게임을 만들 때 이런 상황이 자주 있습니다:

- 버튼을 클릭했을 때 → 여러 시스템이 반응해야 함
- 플레이어가 죽었을 때 → UI 업데이트, 사운드 재생, 게임 종료 등
- 체력이 변했을 때 → 체력바 업데이트, 경고 표시 등

이럴 때 델리게이트를 사용하면, 어떤 함수가 실행될지 미리 정하지 않고도 나중에 연결할 수 있습니다.

🎯 델리게이트의 종류

Unreal Engine의 델리게이트는 크게 4가지 특성으로 구분됩니다:

1 싱글캐스트 vs 멀티캐스트

싱글캐스트	멀티캐스트
하나의 함수만 바인딩	여러 함수를 동시에 바인딩
반환값 사용 가능	반환값 사용 불가 (void만)
콜백에 적합	이벤트 시스템에 적합

싱글캐스트 예시:

```
cpp
// 선언
DECLARE_DELEGATE(FSingleDelegate)

// 사용
MySingleDelegate.BindUObject(this, &AMyClass::Function1);
MySingleDelegate.BindUObject(this, &AMyClass::Function2); // Function1은 사라짐!
MySingleDelegate.Execute(); // Function2만 실행
```

멀티캐스트 예시:

```
cpp
```

```
// 선언
DECLARE_MULTICAST_DELEGATE(FMultiDelegate)

// 사용
MyMultiDelegate.AddUObject(this, &AMyClass::Function1);
MyMultiDelegate.AddUObject(this, &AMyClass::Function2);
MyMultiDelegate.AddUObject(this, &AMyClass::Function3);
MyMultiDelegate.Broadcast(); // 3개 함수 모두 순차 실행!
```

2 Static vs Dynamic

Static 델리게이트	Dynamic 델리게이트
C++ 전용	블루프린트 연동 가능 ★
빠른 속도	약간 느림
일반 함수 바인딩 가능	UFUNCTION만 바인딩 가능
저장 불가	저장 가능

Static 델리게이트:

```
cpp

DECLARE_DELEGATE(FMyDelegate)
DECLARE_MULTICAST_DELEGATE(FMyMultiDelegate)
```

Dynamic 델리게이트 (블루프린트 노출):

```
cpp

DECLARE_DYNAMIC_DELEGATE(FMyDynamicDelegate)
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FMyDynamicMultiDelegate)

// 블루프린트에 노출
UPROPERTY(BlueprintAssignable)
FMyDynamicMultiDelegate OnPlayerDied;
```

델리게이트 선언하기

델리게이트를 사용하려면 먼저 **매크로로 선언해야 합니다.**

기본 선언 패턴

```
cpp
```

```
// 파라미터 없음  
DECLARE_DELEGATE(FMyDelegate)  
  
// 파라미터 1개  
DECLARE_DELEGATE_OneParam(FMyDelegate, int32)  
  
// 파라미터 2개  
DECLARE_DELEGATE_TwoParams(FMyDelegate, int32, FString)  
  
// 파라미터 3개 이상  
DECLARE_DELEGATE_ThreeParams(FMyDelegate, int32, FString, bool)  
// FourParams, FiveParams ... 최대 8개까지
```

 **TIP:** 공식 문서에 따르면 파라미터는 **최대 8개까지** 지원됩니다.

반환값이 있는 델리게이트

cpp

```
// 반환값만  
DECLARE_DELEGATE_RetVal(int32, FCalculateDelegate)  
  
// 반환값 + 파라미터 1개  
DECLARE_DELEGATE_RetVal_OneParam(bool, FCheckDelegate, FString)  
  
// 반환값 + 파라미터 2개  
DECLARE_DELEGATE_RetVal_TwoParams(float, FComputeDelegate, int32, int32)
```

 **주의:** 멀티캐스트 델리게이트는 반환값을 가질 수 없습니다!

멀티캐스트 델리게이트 선언

cpp

```
// Static 멀티캐스트  
DECLARE_MULTICAST_DELEGATE(FMyMultiDelegate)  
DECLARE_MULTICAST_DELEGATE_OneParam(FMyMultiDelegate, int32)  
DECLARE_MULTICAST_DELEGATE_TwoParams(FMyMultiDelegate, int32, FString)  
  
// Dynamic 멀티캐스트  
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FMyDynamicMultiDelegate)  
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FMyDynamicMultiDelegate, int32, NewValue)  
DECLARE_DYNAMIC_MULTICAST_DELEGATE_TwoParams(FMyDynamicMultiDelegate, int32, Value1, FString, \
```

실전 예제

cpp

```

// .h 파일
class AMyActor : public AActor
{
    GENERATED_BODY()

public:
    // 싱글캐스트 - 계산 결과를 반환받을 때
    DECLARE_DELEGATE_RetVal_OneParam(int32, FCalculateDelegate, int32)
    FCalculateDelegate OnCalculate;

    // 멀티캐스트 - 체력 변화 이벤트
    DECLARE_MULTICAST_DELEGATE_OneParam(FOnHealthChanged, float)
    FOnHealthChanged OnHealthChanged;

    // Dynamic 멀티캐스트 - 블루프린트 노출
    DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FOnScoreChanged, int32, NewScore);

    UPROPERTY(BlueprintAssignable, Category = "Events")
    FOnScoreChanged OnScoreChanged;
};

```

🔗 델리게이트 바인딩하기

함수를 델리게이트에 "연결"하는 것을 바인딩이라고 합니다.

주요 바인딩 방법

1. BindUObject / AddUObject ★ (가장 많이 사용)

UObject 클래스의 멤버 함수를 바인딩할 때 사용합니다.

cpp

```

// 싱글캐스트
MyDelegate.BindUObject(this, &AMyActor::MyFunction);

// 멀티캐스트
MyMultiDelegate.AddUObject(this, &AMyActor::Function1);
MyMultiDelegate.AddUObject(this, &AMyActor::Function2);

```

특징:

- 객체가 삭제되면 자동으로 안전하게 처리됨 (약한 참조)
- UObject 파생 클래스 전용
- 가장 안전하고 권장되는 방식

2. BindDynamic / AddDynamic (블루프린트 연동)

Dynamic 델리게이트를 바인딩할 때 사용합니다.

cpp

```
// 함수는 반드시 UFUNCTION으로 선언되어야 함
UFUNCTION()
void OnPlayerDied();

// 바인딩
OnPlayerDiedDelegate.AddDynamic(this, &AMyActor::OnPlayerDied);

// 중복 방지 바인딩 (같은 함수가 이미 바인딩되어 있으면 추가 안 함)
OnPlayerDiedDelegate.AddUniqueDynamic(this, &AMyActor::OnPlayerDied);
```

특징:

- Dynamic 델리게이트 전용
- UFUNCTION 매크로 필수
- AddUniqueDynamic은 중복 바인딩 방지

3. BindLambda / AddLambda (간단한 로직)

람다 함수(익명 함수)를 바인딩할 때 사용합니다.

cpp

```
MyDelegate.BindLambda([]() {
    UE_LOG(LogTemp, Warning, TEXT("람다 실행!"));
});

// 변수 캡처
int32 Score = 100;
MyDelegate.BindLambda([Score]() {
    UE_LOG(LogTemp, Warning, TEXT("점수: %d"), Score);
});
```

4. BindWeakLambda / AddWeakLambda (안전한 람다)

객체가 삭제되면 자동으로 실행되지 않는 안전한 람다입니다.

cpp

```
MyDelegate.BindWeakLambda(this, [this]() {
    // this가 유효할 때만 실행됨
    ... Health -= 10;
});
```

5. BindRaw / AddRaw (일반 C++ 클래스)

UObject가 아닌 일반 C++ 클래스를 바인딩할 때 사용합니다.

cpp

```
MyNonUObjectClass* MyObject = new MyNonUObjectClass();
MyDelegate.BindRaw(MyObject, &MyNonUObjectClass::Function);

// ⚠ 주의: 객체 삭제를 직접 관리해야 함!
```

6. BindSP / AddSP (스마트 포인터)

TSharedPtr 스마트 포인터로 관리되는 객체를 바인딩할 때 사용합니다.

cpp

```
// TSharedPtr 객체 바인딩
TSharedPtr<MyClass> SharedObj = MakeShared<MyClass>();
MyDelegate.BindSP(SharedObj, &MyClass::Function);

// 멀티캐스트
MyMultiDelegate.AddSP(SharedObj, &MyClass::Function);
```

특징:

- 스마트 포인터(TSharedPtr)와 함께 사용
- 자동 수명 관리 (참조 카운트)
- 약한 참조 유지로 안전함

7. BindThreadSafeSP / AddThreadSafeSP (멀티스레드 안전)

멀티스레드 환경에서 안전한 스마트 포인터 바인딩입니다.

cpp

```
// ThreadSafe TSharedRef 생성
TSharedRef<MyClass, ESPMode::ThreadSafe> ThreadSafeObj(new MyClass());
MyDelegate.BindThreadSafeSP(ThreadSafeObj, &MyClass::Function);

// 멀티캐스트
MyMultiDelegate.AddThreadSafeSP(ThreadSafeObj, &MyClass::Function);
```

특징:

- ESPMode::ThreadSafe 모드의 TSharedPtr/TSharedRef 전용
- 멀티스레드에서 안전하게 사용 가능

- 일반 BindSP보다 약간 느림

8. BindStatic / AddStatic (전역/정적 함수)

static 함수나 전역 함수를 바인딩할 때 사용합니다.

cpp

```
static void GlobalFunction()
{
    // 전역 함수
}
```

```
MyDelegate.BindStatic(&GlobalFunction);
```

9. BindUFunction (문자열로 바인딩)

함수 이름을 문자열로 지정하여 바인딩합니다.

cpp

```
UFUNCTION()
void MyFunction();

// 함수 이름으로 바인딩
MyDelegate.BindUFunction(this, FName("MyFunction"));
```

특징:

- 동적으로 함수 이름을 결정할 때 유용
- 오타 위험이 있으므로 주의

바인딩 방식 비교표

바인딩 방식	사용 대상	안전성	멀티스레드	추천도
BindUObject / AddUObject	UObject 멤버 함수	✓ 높음	⚠ 주의	★★★★★
BindDynamic / AddDynamic	UFUNCTION	✓ 높음	⚠ 주의	★★★★★
AddUniqueDynamic	UFUNCTION (중복 방지)	✓ 높음	⚠ 주의	★★★★★
BindWeakLambda / AddWeakLambda	안전한 람다	✓ 높음	⚠ 주의	★★★★★
BindSP / AddSP	TSharedPtr	✓ 높음	✗ 불가	★★★★★
BindThreadSafeSP / AddThreadSafeSP	ThreadSafe TSharedPtr	✓ 높음	✓ 가능	★★★★★
BindLambda / AddLambda	람다 함수	⚠ 주의	⚠ 주의	★★★★
BindRaw / AddRaw	일반 C++ 클래스	✗ 낮음	⚠ 주의	★★★
BindStatic / AddStatic	정적 함수	✓ 높음	✓ 가능	★★★★
BindUFunction	UFUNCTION (문자열)	✓ 높음	⚠ 주의	★★★★

▶ 델리게이트 실행하기

싱글캐스트 델리게이트 실행

Execute()

cpp

```
MyDelegate.Execute();  
MyDelegate.Execute(Param1, Param2); // 파라미터 전달
```

⚠ 주의: 바인딩이 없으면 크래시 발생! (공식 문서: "메모리에 낙서할 수 있음")

ExecutelfBound() ★ (권장)

cpp

```
MyDelegate.ExecutelfBound();  
MyDelegate.ExecutelfBound(Param1, Param2);
```

✓ 바인딩 여부를 자동으로 확인하고 실행합니다. 안전한 방식!

IsBound() (수동 확인)

cpp

```
if (MyDelegate.IsBound())  
{  
    ...MyDelegate.Execute();  
}
```

반환값 받기

cpp

```
DECLARE_DELEGATE_RetVal(int32, FCalculateDelegate)  
  
FCalculateDelegate MyDelegate;  
MyDelegate.BindUObject(this, &AMyActor::Calculate);  
  
int32 Result = MyDelegate.Execute(); // 반환값 받기
```

멀티캐스트 델리게이트 실행

Broadcast()

cpp

```
MyMultiDelegate.Broadcast();
MyMultiDelegate.Broadcast(Param1, Param2); // 파라미터 전달
```

- 바인딩된 모든 함수를 순차적으로 실행합니다. 바인딩이 없어도 안전합니다 (아무 일도 일어나지 않음).

실행 방식 비교

방식	델리게이트 타입	안전성	사용 시기
Execute()	싱글캐스트	⚠️ 크래시 위험	바인딩이 확실할 때
ExecutelfBound()	싱글캐스트	<input checked="" type="checkbox"/> 안전	일반적인 경우 (권장)
Broadcast()	멀티캐스트	<input checked="" type="checkbox"/> 항상 안전	이벤트 발생 시

▣ 바인딩 해제하기

싱글캐스트

cpp

```
MyDelegate.Unbind(); // 바인딩 해제
MyDelegate.Clear(); // 같은 기능
```

멀티캐스트

cpp

```
// 특정 함수 제거 (핸들 사용)
FDelegateHandle Handle = MyMultiDelegate.AddUObject(this, &AMyActor::Function);
MyMultiDelegate.Remove(Handle);

// 특정 객체의 모든 바인딩 제거
MyMultiDelegate.RemoveAll(this);

// Dynamic 델리게이트 - 특정 함수 제거
OnScoreChanged.RemoveDynamic(this, &AMyActor::OnScoreUpdate);

// 전체 제거
MyMultiDelegate.Clear();
```

▣ 실전 예제

예제 1: 체력 시스템 (멀티캐스트)

cpp

```

// MyCharacter.h
class AMyCharacter : public ACharacter
{
    GENERATED_BODY()

public:
    // 체력 변화 이벤트 (여러 시스템이 구독 가능)
    DECLARE_MULTICAST_DELEGATE_TwoParams(FOnHealthChanged, float, float)
    FOnHealthChanged OnHealthChanged;

    void TakeDamage(float Damage)
    {
        float OldHealth = Health;
        Health -= Damage;

        // 모든 구독자에게 알림
        OnHealthChanged.Broadcast(OldHealth, Health);
    }

private:
    float Health = 100.0f;
};

// HealthBarWidget.cpp
void UHealthBarWidget::Initialize()
{
    // 체력 변화 구독
    Player->OnHealthChanged.AddUObject(this, &UHealthBarWidget::UpdateHealthBar);
}

void UHealthBarWidget::UpdateHealthBar(float OldHealth, float NewHealth)
{
    // UI 업데이트
    HealthBar->SetPercent(NewHealth / 100.0f);
}

// SoundManager.cpp
void USoundManager::Initialize()
{
    // 같은 이벤트를 여러 곳에서 구독 가능!
    Player->OnHealthChanged.AddUObject(this, &USoundManager::PlayHurtSound);
}

void USoundManager::PlayHurtSound(float OldHealth, float NewHealth)
{
    if (NewHealth < OldHealth)

```

```
{  
    PlaySound(HurtSoundCue);  
}  
}
```

예제 2: 버튼 클릭 (Dynamic + 블루프린트)

cpp

```

// MyButton.h
UCLASS()
class UMyButton : public UUserWidget
{
    GENERATED_BODY()

public:
    // 블루프린트에서 바인딩 가능한 이벤트
    DECLARE_DYNAMIC_MULTICAST_DELEGATE(FOnButtonClicked);

    UPROPERTY(BlueprintAssignable, Category = "Events")
    FOnButtonClicked OnClicked;

    UFUNCTION()
    void HandleClick()
    {
        // 클릭 이벤트 발생
        OnClicked.Broadcast();
    }
};

// GameMode.cpp
void AMyGameMode::BeginPlay()
{
    Super::BeginPlay();

    // C++에서 바인딩
    StartButton->OnClicked.AddDynamic(this, &AMyGameMode::StartGame);

    // 중복 방지 바인딩 (이미 바인딩되어 있으면 무시)
    StartButton->OnClicked.AddUniqueDynamic(this, &AMyGameMode::StartGame);
}

UFUNCTION()
void AMyGameMode::StartGame()
{
    // 게임 시작 로직
    UE_LOG(LogTemp, Warning, TEXT("게임 시작!"));
}

```

블루프린트에서:

1. MyButton 위젯을 배치
2. OnClicked 이벤트 노드를 추가
3. 원하는 블루프린트 로직 연결

예제 3: 비동기 작업 완료 콜백 (싱글캐스트)

cpp

```
// AsyncLoader.h
class UAsyncLoader : public UObject
{
public:
    // 로딩 완료 콜백 (반환값 있는 멤리게이트)
    DECLARE_DELEGATE_OneParam(FOnLoadComplete, bool)

    void LoadAssetAsync(FOnLoadComplete Callback)
    {
        .... OnLoadCompleteCallback = Callback;
        // 비동기 로딩 시작...
    }

private:
    void OnAssetLoaded(bool bSuccess)
    {
        // 로딩 완료 시 콜백 실행
        OnLoadCompleteCallback.ExecuteIfBound(bSuccess);
    }

    FOnLoadComplete OnLoadCompleteCallback;
};

// 사용
void AMyActor::LoadSomething()
{
    ... Loader->LoadAssetAsync(
        .... FAsyncLoader::FOnLoadComplete::CreateUObject(
            this, &AMyActor::OnLoadFinished
        )
    );
}

void AMyActor::OnLoadFinished(bool bSuccess)
{
    if (bSuccess)
    {
        UE_LOG(LogTemp, Warning, TEXT("로딩 성공!"));
    }
}
```

예제 4: 람다를 활용한 간단한 타이머

cpp

```
void AMyActor::StartCountdown()
{
    GetWorld()->GetTimerManager().SetTimer(
        TimerHandle,
        FTimerDelegate::CreateWeakLambda(this, [this]()
    {
        CountdownTime--;
        UE_LOG(LogTemp, Warning, TEXT("남은 시간: %d"), CountdownTime);

        if (CountdownTime <= 0)
        {
            GetWorld()->GetTimerManager().ClearTimer(TimerHandle);
            OnCountdownFinished.Broadcast();
        }
    }),
    1.0f, // 1초마다
    true // 반복
);
}
```

예제 5: 멀티스레드 안전 멸리게이트

cpp

```

// ThreadSafeManager.h
class FThreadSafeManager : public TSharedFromThis<FThreadSafeManager, ESPMode::ThreadSafe>
{
public:
    DECLARE_DELEGATE(FOnTaskComplete)

    void StartAsyncTask(FOnTaskComplete Callback)
    {
        // 멀티스레드 환경에서 안전한 바인딩
        TSharedRef<FThreadSafeManager, ESPMode::ThreadSafe> SharedThis = AsShared();

        Async(EAsyncExecution::ThreadPool, [SharedThis, Callback]()
        {
            // 백그라운드 작업
            FPlatformProcess::Sleep(2.0f);

            // 완료 후 콜백 실행 (메인 스레드에서)
            AsyncTask(ENamedThreads::GameThread, [Callback]()
            {
                Callback.ExecuteIfBound();
            });
        });
    }
};

// 사용
void AMyActor::BeginPlay()
{
    Super::BeginPlay();

    TSharedRef<FThreadSafeManager, ESPMode::ThreadSafe> Manager =
        MakeShared<FThreadSafeManager, ESPMode::ThreadSafe>();

    FThreadSafeManager::FOnTaskComplete Callback;
    Callback.BindThreadSafeSP(Manager, &FThreadSafeManager::OnComplete);

    Manager->StartAsyncTask(Callback);
}

```

🎓 선택 가이드

어떤 멀리게이트를 사용해야 할까?

블루프린트 연동이 필요한가?

|— YES → Dynamic 멀리게이트 사용

```
| ..... DECLARE_DYNAMIC_MULTICAST_DELEGATE(...)  
| ..... UPROPERTY(BlueprintAssignable)
```

└ NO → Static 델리게이트 사용

여러 함수를 동시에 실행해야 하는가?

└ YES → 멀티캐스트

```
| ..... DECLARE_MULTICAST_DELEGATE(...)
```

| 예: 이벤트 시스템, UI 업데이트

└ NO → 싱글캐스트

```
DECLARE_DELEGATE(...)
```

예: 콜백, 반환값이 필요한 경우

어떤 바인딩 방식을 사용해야 할까?

UObject 파생 클래스인가?

└ YES → BindUObject / AddUObject (권장)

└ NO → 일반 C++ 클래스인가?

└ YES → 스마트 포인터 사용?

└ YES → 멀티스레드 환경?

```
| ..... | ..... YES → BindThreadSafeSP / AddThreadSafeSP
```

| | NO → BindSP / AddSP

| NO → BindRaw / AddRaw (주의!)

└ 정적 함수인가?

└ YES → BindStatic / AddStatic

간단한 로직만 필요한가?

└ YES → BindWeakLambda / AddWeakLambda (안전)

또는 BindLambda / AddLambda

⚠ 주의사항 및 베스트 프랙티스

1. Execute vs ExecutefBound

cpp

// ✗ 위험한 코드

```
MyDelegate.Execute(); // 바인딩이 없으면 크래시!
```

// ✓ 안전한 코드

```
MyDelegate.ExecutefBound(); // 바인딩 확인 후 실행
```

2. 객체 수명 관리

cpp

```
// ✗ 위험: 객체가 삭제될 수 있음  
MyDelegate.BindRaw(new MyClass(), &MyClass::Function);  
  
// ✓ 안전: UObject는 자동 관리됨 (약한 참조)  
MyDelegate.BindUObject(this, &AMyActor::Function);  
  
// ✓ 안전: 스마트 포인터 사용  
TSharedPtr<MyClass> MyObj = MakeShared<MyClass>();  
MyDelegate.BindSP(MyObj, &MyClass::Function);
```

3. Dynamic 델리게이트는 UFUNCTION 필수

cpp

```
// ✗ 컴파일 에러  
void NormalFunction(); // UFUNCTION 없음  
MyDynamicDelegate.AddDynamic(this, &AMyActor::NormalFunction);  
  
// ✓ 올바른 사용  
UFUNCTION()  
void ProperFunction();  
MyDynamicDelegate.AddDynamic(this, &AMyActor::ProperFunction);
```

4. 멀티캐스트는 반환값 불가

cpp

```
// ✗ 불가능 - 컴파일 에러  
DECLARE_MULTICAST_DELEGATE_RetVal(int32, FMyDelegate)  
  
// ✓ 멀티캐스트는 void만 가능  
DECLARE_MULTICAST_DELEGATE(FMyDelegate)
```

5. 중복 바인딩 방지

cpp

```
// ✓ 중복 방지 - 이미 바인딩되어 있으면 추가 안 함  
OnEvent.AddUniqueDynamic(this, &AMyActor::OnEventHandler);  
  
// 또는 수동으로 제거 후 추가  
OnEvent.RemoveDynamic(this, &AMyActor::OnEventHandler);  
OnEvent.AddDynamic(this, &AMyActor::OnEventHandler);
```

6. 페이로드 데이터 활용

cpp

```
// 바인딩 시 추가 데이터 전달 (최대 4개)
MyDelegate.BindUObject(this, &AMyActor::OnComplete, 100, TEXT("ExtraData"));

void AMyActor::OnComplete(int32 ExtraValue, FString ExtraString)
{
    // 바인딩 시 전달한 데이터 사용
}
```

 **TIP:** 페이로드 변수는 최대 4개까지 지원되며, Dynamic 델리게이트는 페이로드를 지원하지 않습니다.

7. 멀티스레드 환경

cpp

```
// ❌ 위험: 일반 델리게이트는 멀티스레드 안전하지 않음
MyDelegate.BindSP(SharedPtr, &MyClass::Function);

// ✅ 안전: ThreadSafe 버전 사용
TSharedRef<MyClass, ESPMode::ThreadSafe> ThreadSafeObj(new MyClass());
MyDelegate.BindThreadSafeSP(ThreadSafeObj, &MyClass::Function);
```

💡 팁: 직렬화(Serialization)

직렬화란? 데이터를 파일에 저장하거나 네트워크로 전송할 수 있는 형태로 변환하는 것입니다.

Dynamic 델리게이트만 직렬화가 가능합니다.

cpp

```
// ✅ 저장 가능 - 블루프린트에서 설정한 바인딩이 저장됨
UPROPERTY(BlueprintAssignable)
FMyDynamicMultiDelegate OnEvent;

// ❌ 저장 불가 - 재시작 시 바인딩 사라짐
FMyStaticMultiDelegate OnEvent;
```

용도:

- 게임 세이브/로드 시스템
- 네트워크 리플리케이션
- 에디터에서 설정한 이벤트 유지

요약

델리게이트 핵심 정리

특징	싱글캐스트	멀티캐스트	Static	Dynamic
바인딩 개수	1개	여러 개	-	-
반환값	가능	불가 (void)	가능	불가
블루프린트	-	-	불가	가능 ★
저장	-	-	불가	가능
속도	-	-	빠름	약간 느림
사용 예	콜백	이벤트	C++ 전용	BP 연동

주요 함수 정리

바인딩

- BindUObject / AddUObject - UObject 멤버 함수 (권장) ★
- BindDynamic / AddDynamic - UFUNCTION (블루프린트) ★
- AddUniqueDynamic - 중복 방지 바인딩 ★
- BindWeakLambda / AddWeakLambda - 안전한 람다
- BindSP / AddSP - 스마트 포인터
- BindThreadSafeSP / AddThreadSafeSP - 멀티스레드 안전

실행

- ExecuteIfBound() - 싱글캐스트 안전 실행 (권장) ★
- Broadcast() - 멀티캐스트 실행 ★
- Execute() - 직접 실행 (주의!)

해제

- Unbind() / Clear() - 싱글캐스트
- Remove() / RemoveAll() / Clear() - 멀티캐스트
- RemoveDynamic() - Dynamic 델리게이트

제한 사항

- 파라미터: 최대 8개
- 페이로드: 최대 4개 (Dynamic은 불가)
- 반환값: 싱글캐스트만 가능

다음 단계

델리게이트를 마스터했다면:

1. 이벤트 시스템 구축하기

- 게임플레이 이벤트
- UI 업데이트 시스템
- 사운드 관리 시스템

2. Observer 패턴 구현하기

- 멀티캐스트 델리게이트 활용
- 느슨한 결합 시스템 설계

3. 비동기 작업 처리하기

- 로딩 시스템
- 네트워크 통신
- 타이머 시스템

4. 멀티스레드 프로그래밍

- ThreadSafe 델리게이트 활용
- 백그라운드 작업 관리

참고 자료

- [Unreal Engine 공식 델리게이트 문서](#)
- [Dynamic Delegates 공식 문서](#)
- [Multicast Delegates 공식 문서](#)

Happy Coding! 🎮

버전: 1.0 (2025) 최종 검증: Unreal Engine 5.6 공식 문서 기준