

언리얼 엔진 스마트 포인터 가이드

🎯 스마트 포인터란?

스마트 포인터는 메모리를 자동으로 관리해주는 특별한 포인터입니다. 일반 포인터(Raw Pointer)를 사용하면 개발자가 직접 `new`로 메모리를 할당하고 `delete`로 해제해야 하지만, 스마트 포인터는 이 과정을 자동으로 처리해줘서 메모리 누수나 땅글링 포인터(이미 삭제된 메모리를 가리키는 포인터) 같은 오류를 방지할 수 있습니다.

언리얼 엔진의 스마트 포인터 라이브러리는 C++11 표준 스마트 포인터를 기반으로 커스텀 구현한 것으로, 메모리 할당과 추적의 부담을 줄여주도록 설계되었습니다.

⚠️ 중요한 제한사항

언리얼의 스마트 포인터는 UObject와 함께 사용할 수 없습니다!

- AActor, UActorComponent, UObject 등 언리얼의 기본 클래스들은 이미 자체적인 메모리 관리 시스템(가비지 컬렉션)을 갖고 있습니다
- 엔진에는 UObject 관리를 위한 별도의 메모리 관리 시스템이 있으며, 두 시스템은 호환되지 않습니다
- UObject 파생 클래스들은 `UPROPERTY()` 매크로를 사용해야 합니다
- 스마트 포인터는 일반 C++ 클래스(UObject 시스템 외부의 데이터 객체)에만 사용하세요

📋 스마트 포인터 종류 한눈에 보기

| 타입 | 설명 | Null 가능 여부 | 사용 시점 |
|------------|--------------------------|------------|----------------------------------|
| TSharedRef | 여러 곳에서 공유 가능, 절대 비어있지 않음 | ✗ | 여러 객체가 공유하되 null이 아님을 보장할 때 (권장) |
| TSharedPtr | 여러 곳에서 공유 가능, 비어있을 수 있음 | ✓ | TSharedRef와 같지만 null 상태가 필요할 때 |
| TWeakPtr | 약한 참조, 순환 참조 방지 | ✓ | 객체를 관찰만 하고 소유하지 않을 때 |
| TUniquePtr | 단 하나의 소유자만 가능 | ✓ | 한 곳에서만 독점적으로 사용할 때 |

중요: TSharedRef를 TSharedPtr보다 우선적으로 사용하는 것이 권장됩니다. 비어있거나 null 가능한 객체가 필요한 경우에만 TSharedPtr를 사용하세요.

위험: Unique Pointer가 참조하는 객체에 대해 Shared Pointer나 Shared Reference를 만드는 것은 위험합니다!

1 TSharedRef (공유 레퍼런스) - 권장!

언제 사용하나요?

여러 객체가 하나의 데이터를 함께 사용하되, 절대 null이 되지 않음을 보장하고 싶을 때 사용합니다. TSharedPtr과 TSharedRef 중 선택해야 한다면, null 상태가 필요하지 않은 한 **TSharedRef**가 권장됩니다.

주요 특징

- **Non-nullable:** 절대 비어있을 수 없습니다
- **공유 소유권:** 여러 개의 TSharedRef가 같은 객체를 가리킬 수 있습니다
- **참조 카운팅:** 몇 개가 객체를 가리키는지 자동으로 세어줍니다

- **자동 삭제**: 마지막 참조가 사라지면 객체도 자동으로 삭제됩니다
- **IsValid()** 없음: 항상 유효하므로 유효성 검사 함수가 없습니다
- **재할당 가능**: 일반 C++ 레퍼런스와 달리, 생성 후 다른 객체로 재할당 가능합니다

코드 예제



cpp

```
// 반드시 객체와 함께 생성해야 함
TSharedRef<FMyClass> MyRef = MakeShared<FMyClass>();

// Null 체크 필요 없음 - 항상 유효함
MyRef->DoSomething(); // 안전!

// 다른 객체로 재할당 가능
MyRef = MakeShared<FMyClass>();

// TSharedPtr로 암시적 변환 가능
TSharedPtr<FMyClass> MyPtr = MyRef; // 자동 변환
```

TSharedPtr에서 TSharedRef로 변환



cpp

```
TSharedPtr<FMyClass> MyPtr = MakeShared<FMyClass>();

// ToSharedRef()로 변환 (null이면 assertion 발생!)
if (MyPtr.IsValid())
{
    TSharedRef<FMyClass> MyRef = MyPtr.ToSharedRef();
}
```

2 TSharedPtr (공유 포인터)

언제 사용하나요?

여러 객체가 하나의 데이터를 함께 사용하고, **null 상태가 필요한 경우** 사용합니다. TSharedRef와 비슷하지만 비어있을 수 있습니다.

주요 특징

- **Nullable**: 비어있는 상태(null)가 가능합니다
- **공유 소유권**: 여러 개의 TSharedPtr가 같은 객체를 가리킬 수 있습니다

- **참조 카운팅**: 몇 개의 포인터가 객체를 가리키는지 자동으로 세어줍니다
- **자동 삭제**: 마지막 포인터가 사라지면 객체도 자동으로 삭제됩니다
- **자동 무효화**: 휘발성 객체를 데그링 포인터 걱정 없이 안전하게 참조할 수 있습니다
- **Non-intrusive**: 객체는 스마트 포인터가 자신을 소유하는지 알지 못합니다

추가 장점

- **메모리 누수 방지**: 자동 메모리 관리
- **데그링 포인터 방지**: 삭제된 메모리 접근 방지
- **초기화되지 않은 메모리 방지**: 항상 유효하거나 null 상태
- **약한 참조 지원**: TWeakPtr로 순환 참조 해결
- **의도 표현**: 소유자와 관찰자를 명확히 구분

코드 예제



cpp

```
// 빈 포인터 생성 가능
TSharedPtr<FMyClass> MyPtr;

// 또는 객체와 함께 생성
TSharedPtr<FMyClass> MyPtr = MakeShared<FMyClass>();

// 다른 포인터와 공유
TSharedPtr<FMyClass> MyPtr2 = MyPtr; // 참조 카운트: 2

// 값 접근
MyPtr->DoSomething();

// 유효성 검사 (null 체크)
if (MyPtr.IsValid())
{
    // 안전하게 사용
}

// null로 리셋
MyPtr.Reset();
MyPtr = nullptr;

// MyPtr과 MyPtr2가 모두 스코프를 벗어나면 객체 자동 삭제
```

커스텀 삭제자 (Custom Deleter)

특별한 삭제 로직이 필요한 경우 람다 함수를 제공할 수 있습니다:



```

void DestroyMyObjectType(FMyObjectType* ObjectAboutToDelete)
{
    // 커스텀 삭제 코드
    UE_LOG(LogTemp, Warning, TEXT("객체 삭제 중..."));
}

// 커스텀 삭제자를 가진 스마트 포인터 생성
TSharedRef<FMyObjectType> NewReference(
    new FMyObjectType(),
    [] (FMyObjectType* Obj){ DestroyMyObjectType(Obj); }
);

TSharedPtr<FMyObjectType> NewPointer(
    new FMyObjectType(),
    [] (FMyObjectType* Obj){ DestroyMyObjectType(Obj); }
);

```

3 TWeakPtr (약한 포인터)

언제 사용하나요?

객체를 "관찰"만 하고 싶고, 그 객체의 생명주기에 영향을 주고 싶지 않을 때 사용합니다. 순환 참조 문제를 해결할 때 필수적입니다.

주요 특징

- **약한 참조**: 참조 카운트에 영향을 주지 않습니다
- **소유권 없음**: 객체의 생명주기를 연장하지 않습니다
- **순환 참조 방지**: 강한 참조 사이클을 끊을 수 있습니다
- **안전한 접근**: Pin()을 통해 안전하게 접근합니다
- **의도 표현**: "이 객체를 관찰만 하고 소유하지 않음"을 명확히 표현

중요한 동작 방식

주의: Weak Pointer는 참조하는 객체가 삭제되어도 **자동으로 null이 되지 않습니다**. 대신, 가비지 컬렉션 시스템에 객체가 아직 유효한지 물어보는 추가 작업을 수행합니다. 따라서 Weak Pointer는 거의 항상 Hard Pointer(TSharedPtr/TSharedRef)보다 런타임 비용이 더 큽니다.

순환 참조 문제란?



```
// 문제 상황: A가 B를 가리키고, B가 A를 가리킴
class A {
    ... TSharedPtr<B> BPtr; // A가 B를 소유
};

class B {
    ... TSharedPtr<A> APtr; // B가 A를 소유
};

// 결과: 둘 다 서로를 잡고 있어서 영원히 삭제되지 않음! (메모리 누수)
```

해결 방법



cpp

```
class A {
    ... TSharedPtr<B> BPtr; // A가 B를 소유
};

class B {
    ... TWeakPtr<A> APtr; // B는 A를 관찰만 함
};

// 결과: 순환 참조 해결!
```

코드 예제



cpp

```

TSharedPtr<FMyClass> SharedPtr = MakeShared<FMyClass>();
TWeakPtr<FMyClass> WeakPtr = SharedPtr; // 약한 참조 생성

// 사용하려면 먼저 Pin()으로 TSharedPtr로 변환
TSharedPtr<FMyClass> TempPtr = WeakPtr.Pin();
if (TempPtr.IsValid())
{
    // 객체가 아직 살아있음
    ... TempPtr->DoSomething();
}
else
{
    // 객체가 이미 삭제됨
}

// 또는 nullptr과 비교
if (WeakPtr.Pin() != nullptr)
{
    // 유효함
}

// 또는 IsValid() 사용
if (WeakPtr.IsValid())
{
    // 유효함
}

// WeakPtr 복사는 안전함 (객체 유효성 무관)
TWeakPtr<FMyClass> AnotherWeakPtr = WeakPtr;

// 리셋
WeakPtr = nullptr;
WeakPtr.Reset();

```

Pin 함수의 중요성



cpp

```
// Pin은 Weak Pointer를 Shared Pointer로 승격시킵니다
// Shared Pointer가 스코프 내에 있는 동안 객체는 유효함이 보장됩니다
if (TSharedPtr<FMyClass> PinnedPtr = WeakPtr.Pin())
{
    // bool 조건문에서 true = 유효한 객체
    .. PinnedPtr->DoSomething();
    // PinnedPtr이 스코프를 벗어나기 전까지 객체는 안전
}
```

4 TUniquePtr (고유 포인터)

언제 사용하나요?

하나의 객체가 다른 객체를 **독점적으로 소유**하고 싶을 때 사용합니다. 소유권을 공유할 필요가 없을 때 가장 효율적입니다.

주요 특징

- **단독 소유**: 하나의 TUniquePtr만 객체를 가리킬 수 있습니다
- **복사 불가**: 다른 포인터로 복사할 수 없습니다 (이동만 가능)
- **가장 가벼움**: 참조 카운팅이 없어서 오버헤드가 가장 적습니다
- **소유권 이전**: Move를 통해 소유권을 다른 포인터로 넘길 수 있습니다

코드 예제



cpp

```
// 생성
TUniquePtr<FMyClass> MyPtr = MakeUnique<FMyClass>();

// 복사 불가 - 컴파일 에러!
// TUniquePtr<FMyClass> MyPtr2 = MyPtr; ✗

// 이동은 가능 (소유권 이전)
TUniquePtr<FMyClass> MyPtr2 = MoveTemp(MyPtr); // MyPtr은 이제 비어있음

// 사용
MyPtr2->DoSomething();

// 스코프를 벗어나면 자동 삭제
```

TSharedFromThis

무엇인가요?

클래스 내부에서 자기 자신을 TSharedPtr이나 TSharedRef로 만들고 싶을 때 사용하는 특별한 기능입니다. TSharedFromThis를 상속받으면 AsShared()와 SharedThis() 함수를 사용할 수 있습니다.

언제 필요한가요?



cpp

```
class FMyClass
{
public:
    void RegisterSelf()
    {
        // 문제: 나 자신을 TSharedPtr로 어떻게 만들지? 🤔
        // TSharedPtr<FMyClass> Self = ????
    }
};
```

해결 방법



cpp

```
// TSharedFromThis를 상속받음
class FMyClass : public TSharedFromThis<FMyClass>
{
public:
    void RegisterSelf()
    {
        // AsShared()로 자기 자신의 TSharedRef를 얻을 수 있음
        TSharedRef<FMyClass> Self = AsShared();

        // 또는 SharedThis()
        TSharedPtr<FMyClass> SelfPtr = SharedThis(this);
    }
};

// 사용 - 반드시 Shared Pointer로 생성해야 함!
TSharedRef<FMyClass> MyObject = MakeShared<FMyClass>();
MyObject->RegisterSelf(); // 이제 작동함!
```

AsShared vs SharedThis

- **AsShared():** TSharedFromThis의 템플릿 인자로 전달된 타입을 반환 (부모 타입일 수 있음)
- **SharedThis(this):** 호출한 객체의 실제 타입을 자동으로 추론하여 반환

이 기능은 클래스 팩토리가 항상 Shared Reference를 반환하거나, Shared Reference/Pointer를 요구하는 시스템에 객체를 전달할 때 유용합니다.

실전 사용 예제

참고: 아래 예제들은 이해를 돋기 위해 추가한 것으로 원문에는 없습니다.

예제 1: 게임 매니저 시스템



cpp

```
// 게임 데이터를 여러 곳에서 공유
class FGameData
{
public:
    int32 Score;
    FString PlayerName;
};

class FGameManager
{
private:
    TSharedRef<FGameData> GameData; // TSharedRef 권장!

public:
    FGameManager()
        : GameData(MakeShared<FGameData>()) // 초기화 필수
    {
    }

// 다른 시스템에 데이터 공유
TSharedRef<FGameData> GetGameData()
{
    return GameData;
}

class FUIManager
{
private:
    TSharedRef<FGameData> GameData;

public:
    void SetGameData(TSharedRef<FGameData> InData)
    {
        GameData = InData; // 같은 데이터를 공유
    }

    void UpdateUI()
    {
        // TSharedRef는 항상 유효하므로 검사 불필요
        UE_LOG(LogTemp, Log, TEXT("Score: %d"), GameData->Score);
    }
};
```

예제 2: 순환 참조 방지



cpp

```
class FPlayer;

// 무기는 플레이어를 관찰만 함
class FWeapon
{
private:
    ... TWeakPtr<FPlayer> OwnerPlayer; // 약한 참조

public:
    void SetOwner(TSharedRef<FPlayer> Player)
    {
        OwnerPlayer = Player;
    }

    void Fire()
    {
        TSharedPtr<FPlayer> Player = OwnerPlayer.Pin();
        if (Player.IsValid())
        {
            // 플레이어가 아직 살아있음
            Player->UseAmmo();
        }
        else
        {
            // 플레이어가 삭제됨
            UE_LOG(LogTemp, Warning, TEXT("소유자 없음!"));
        }
    }
};

// 플레이어는 무기를 소유함
class FPlayer : public TSharedFromThis<FPlayer>
{
private:
    ... TSharedRef<FWeapon> Weapon; // 강한 참조

public:
    FPlayer()
        : Weapon(MakeShared<FWeapon>())
    {
    }

    void EquipWeapon()
    {
        // 자기 자신을 무기에 전달
        Weapon->SetOwner(AsShared());
    }
};
```

```
void UseAmmo()
{
    UE_LOG(LogTemp, Log, TEXT("탄약 사용!"));
}
};
```

예제 3: 리소스 관리



cpp

```
// 텍스처 데이터를 독점 관리
class FTextureLoader
{
private:
    TUniquePtr<FRawImageData> ImageData;

public:
    void LoadTexture(const FString& Path)
    {
        // 새 데이터 로드 (이전 데이터는 자동 삭제됨)
        ImageData = MakeUnique<FRawImageData>();
        ImageData->LoadFromFile(Path);
    }

    // 소유권 이전
    TUniquePtr<FRawImageData> TakeData()
    {
        return MoveTemp(ImageData); // 소유권을 넘김
    }

    bool HasData() const
    {
        return ImageData.IsValid();
    }
};
```

⚡ 성능 고려사항

언제 스마트 포인터를 사용하면 좋을까요?

적합한 경우:

- 상위 레벨 시스템 (게임 모드, 매니저 클래스 등)
- 리소스 관리 (파일, 네트워크 연결 등)

- 툴 프로그래밍
- 복잡한 소유권 관계가 있는 경우

✖ 피해야 하는 경우:

- 렌더링 코드 같은 성능이 중요한 저수준 엔진 코드
- 매 프레임 수백~수천 번 생성/삭제되는 객체
- 단순한 임시 데이터

성능 특성

장점:

- ✓ 모든 연산이 상수 시간 (Constant Time)
- ✓ 대부분의 스마트 포인터 역참조는 일반 포인터만큼 빠름 (Shipping 빌드)
- ✓ 스마트 포인터 복사 시 메모리 재할당 없음
- ✓ Thread-safe 스마트 포인터는 Lockless (잠금 없음)

단점:

- ⚠ 생성/복사 시 일반 포인터보다 오버헤드 발생
- ⚠ 참조 카운팅 유지로 인한 사이클 추가

🔒 멀티스레드 환경

여러 스레드에서 동시에 접근해야 한다면 **스레드 안전 버전**을 사용하세요:



cpp

```
// 스레드 안전 버전
TSharedPtr<FMyClass, ESPMode::ThreadSafe> ThreadSafePtr;
TSharedRef<FMyClass, ESPMode::ThreadSafe> ThreadSafeRef;
TWeakPtr<FMyClass, ESPMode::ThreadSafe> ThreadSafeWeakPtr;
```

스레드 안전 특성

- **읽기/복사:** 항상 안전합니다
- **쓰기/리셋:** 동기화가 필요합니다
- **성능:** 원자적 참조 카운팅으로 인해 기본 버전보다 약간 느립니다
- **권장사항:** 단일 스레드만 사용한다면 일반 버전이 더 빠릅니다

📌 함수 파라미터로 전달하기

✖ 비효율적인 방법



cpp

```
void ProcessData(TSharedPtr<FMyData> Data) // 참조 카운팅 오버헤드!
{
    ... Data->DoSomething();
}
```

✓ 효율적인 방법



cpp

```
// 객체를 const 레퍼런스로 직접 전달 (더 빠름)
void ProcessData(const FMyData& Data)
{
    ... Data.DoSomething();
}

// 호출
TSharedPtr<FMyData> MyData = MakeShared<FMyData>();
ProcessData(*MyData); // 역참조해서 전달
```

원칙: 가능한 한 TSharedRef나 TSharedPtr를 함수 파라미터로 전달하지 마세요. 역참조와 참조 카운팅으로 인한 오버헤드가 발생합니다. 대신 참조하는 객체를 직접 전달하되, 가급적 const &로 전달하세요.

예외: 함수가 소유권에 영향을 주는 경우(예: 포인터를 저장하거나 수명을 연장하는 경우)에만 스마트 포인터로 전달하세요.

🔧 추가 기능

Forward Declaration (전방 선언)

불완전한 타입에 대해 Shared Pointer를 전방 선언할 수 있습니다. 이는 헤더 포함을 줄이는 데 도움이 됩니다:



cpp

```
// MyClass.h
class FOtherClass; // 전방 선언

class FMyClass
{
private:
    TSharedPtr<FOtherClass> OtherObject; // 작동함!
};
```

MakeSharable

이미 존재하는 raw pointer를 Shared Pointer로 감쌀 때 사용합니다:



cpp

```
FMyClass* RawPtr = new FMyClass();
```

```
// MakeSharable로 감싸기
```

```
TSharedPtr<FMyClass> SharedPtr = MakeSharable(RawPtr);
```

```
// 주의: RawPtr을 직접 delete하면 안 됨! SharedPtr이 관리함
```

어떤 포인터를 선택할까?

참고: 이 플로우차트는 이해를 돋기 위해 추가한 것입니다.



시작

```
|  
|   └─ UObject 파생 클래스인가?  
|       └─ YES → UPROPERTY() 사용 (스마트 포인터 사용 불가)  
|  
└─ NO → 일반 C++ 클래스  
  
|   └─ 여러 곳에서 공유해야 하나?  
|       └─ YES → Shared Pointer/Reference  
|           |  
|           └─ Null이 될 수 있나?  
|               └─ NO → TSharedRef (권장!)  
|               └─ YES → TSharedPtr  
|  
|   └─ 관찰만 하고 소유는 안 하나?  
|       └─ YES → TWeakPtr  
|  
└─ 독점적으로 소유해야 하나?  
    └─ YES → TUniquePtr
```



핵심 요약

1. **UObject는 스마트 포인터 사용 금지** - 대신 UPROPERTY() 사용
2. **TSharedRef를 우선 사용** - null이 필요한 경우에만 TSharedPtr
3. **관찰만 하려면 TWeakPtr (순환 참조 방지)**
4. **독점 소유하려면 TUniquePtr (가장 가벼움)**
5. **함수 파라미터는 가급적 객체 직접 전달 (const &)**
6. **성능이 중요한 곳에서는 신중하게 사용**
7. **멀티스레드에서는 ThreadSafe 버전 사용**
8. **TWeakPtr는 자동으로 null이 되지 않음** - Pin()으로 유효성 확인 필요
9. **Non-intrusive** - 객체는 스마트 포인터의 존재를 모름
10. **Forward declaration 가능** - 헤더 포함 최소화



추가 학습 자료

- [언리얼 엔진 공식 문서 - Smart Pointers](#)
- [Shared Pointers](#)
- [Shared References](#)
- [Weak Pointers](#)

팁: TSharedRef부터 시작하세요. 가장 안전하고 권장되는 방식입니다. null이 필요한 경우에만 TSharedPtr를 사용하세요!