

# 언리얼 엔진을 통해 살펴보는 리플렉션과 가비지 컬렉션

NHN N스튜디오  
정수용

## C++에서 이런 편리함은 처음…

- 언리얼 엔진의 편리한 기능들을 소개

## 사용법보다는 작동 원리에 대한 고민을 전달하는 것이 목표

- 언리얼 엔진에 관심이 없는 C++ 개발자분들에게 영감을 드릴 수 있으면 좋겠다.
- 언리얼 C++로 개발 중이거나 공부하려는 분들께도 참고가 되면 좋겠다.
  - 컨셉과 작동 원리를 이해하면 사용상의 실수를 줄일 수 있음

## UObject 시스템? 언리얼 엔진의 핵심!

- UObject를 상속한 클래스들에게는 리플렉션, 가비지 컬렉션을 비롯해 여러 가지 편리한 기능들이 제공됨

	UObject 자손 클래스	일반 C++ 클래스
리플렉션	0	X
가비지 컬렉션	0	X
자동 참조 업데이트	0	X
자동 시리얼라이제이션	0	X
네트워크 리플리케이션	0	X
블루프린트 연동	0	X
언리얼 에디터 연동	0	X
기타 등등	0	X

# CONTENTS

1. 리플렉션이란? (C#의 리플렉션 객체 살펴보기)
2. 언리얼 엔진의 리플렉션
3. 가비지 컬렉션이란? (참조 카운터 방식 vs Mark-Sweep 방식)
4. C++에서의 Mark-Sweep GC (Precise vs Conservative)
5. 언리얼 엔진의 MarkSweep GC, 리플렉션과의 관계

# 리플렉션

## Reflection (computer programming)

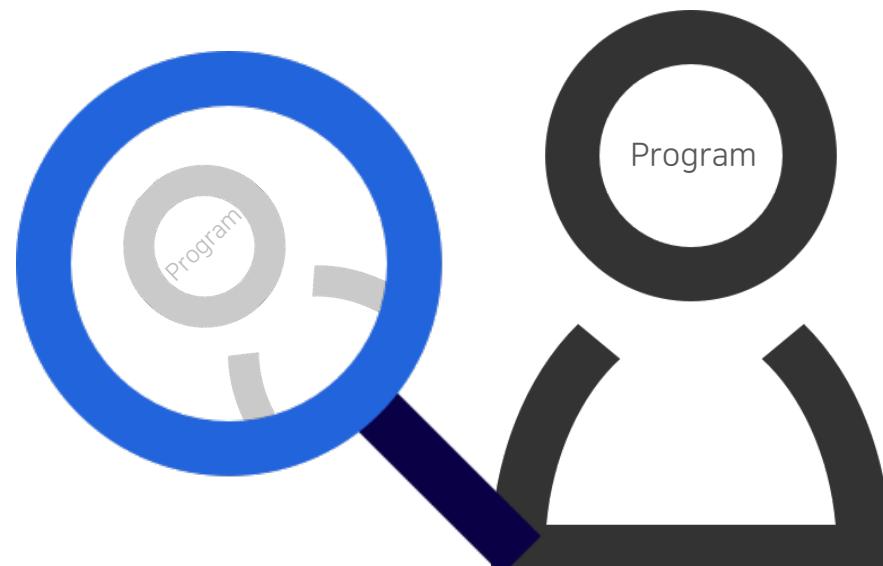
From Wikipedia, the free encyclopedia

*Not to be confused with Reflection (computer graphics).*

In computer science, **reflection** is the ability of a process to examine, introspect, and modify its own structure and behavior.<sup>[1]</sup>

오늘 얘기할 리플렉션의 의미는 type introspection에 한정

- C++에서 타입 정보는 컴파일 타임의 것
- 이것을 런타임에 들여다 볼 수 있게 해주는 기능

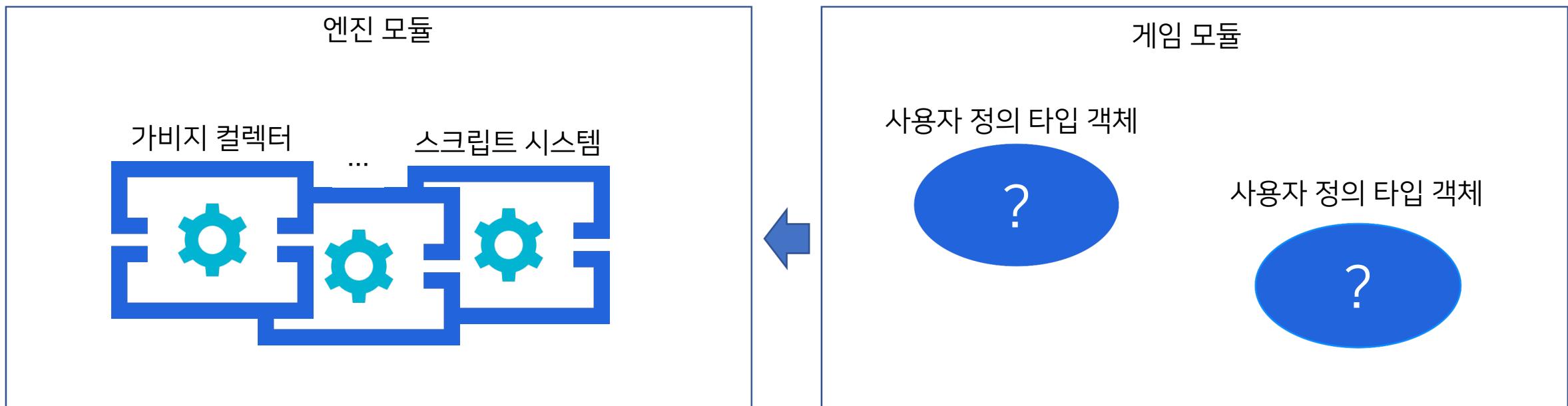


```
// 실제 타입을 모르는 어떤 객체가 주어졌을 때
Object obj = ... ;

// 그 타입에 DoSomething라는 이름을 가진 메서드가 존재한다면
MethodInfo Method = obj.GetType().GetMethod( " DoSomething " );
If (Method != null)
{
    // 호출!
    Object ret = Method.Invoke(obj, new Object[] {} );
    Console.WriteLine(ret.ToString());
}
```

## 게임 엔진에 런타임 리플렉션이 왜 필요할까?

- 다른 모듈들(동적으로 링크되는)의 사용자 정의 클래스 객체들도 엔진 내의 시스템들이 관리해야 함
- 컴파일 타임에는 해당 클래스들을 모르므로, 동적인 타입 정보가 필요
- 객체 생성 함수도 동적인 타입을 다룰 수 있어야 함

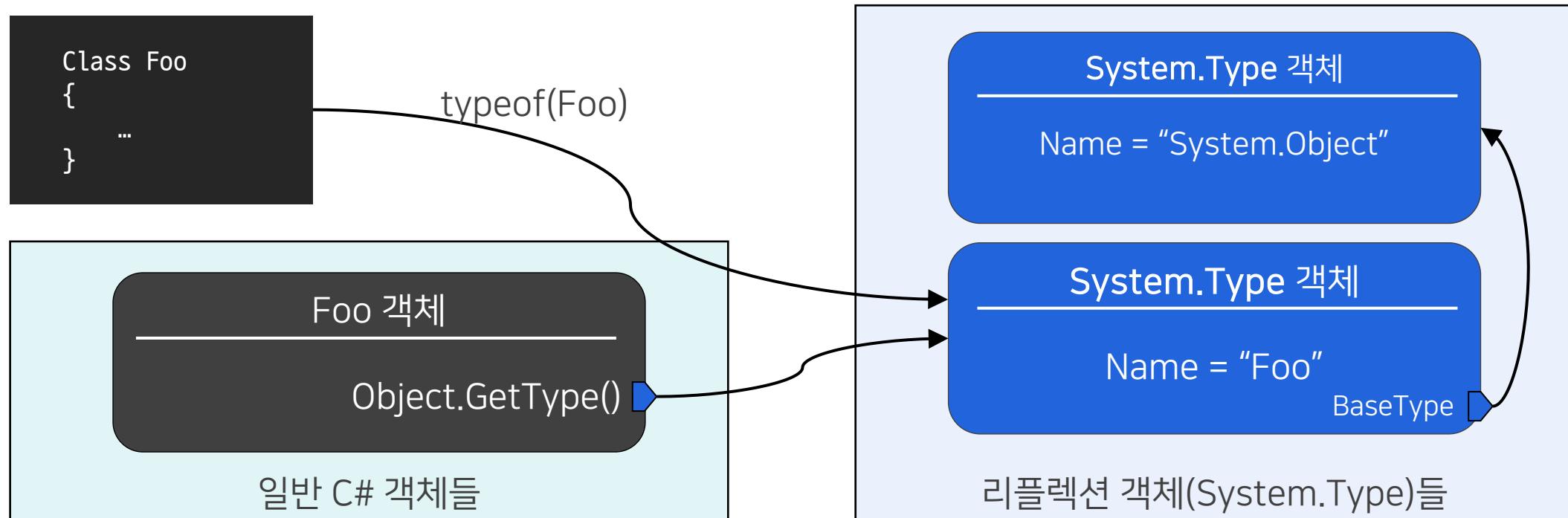


## 주요 런타임 리플렉션 객체 비교(C# vs 언리얼 엔진4)

	C#	언리얼 엔진4
클래스	System.Type	UClass
멤버 변수	System.Reflection.FieldInfo	UProperty
멤버 함수	System.Reflection.MethodInfo	UFunction

## System.Type: 각 타입을 설명하는 런타임 리플렉션 객체

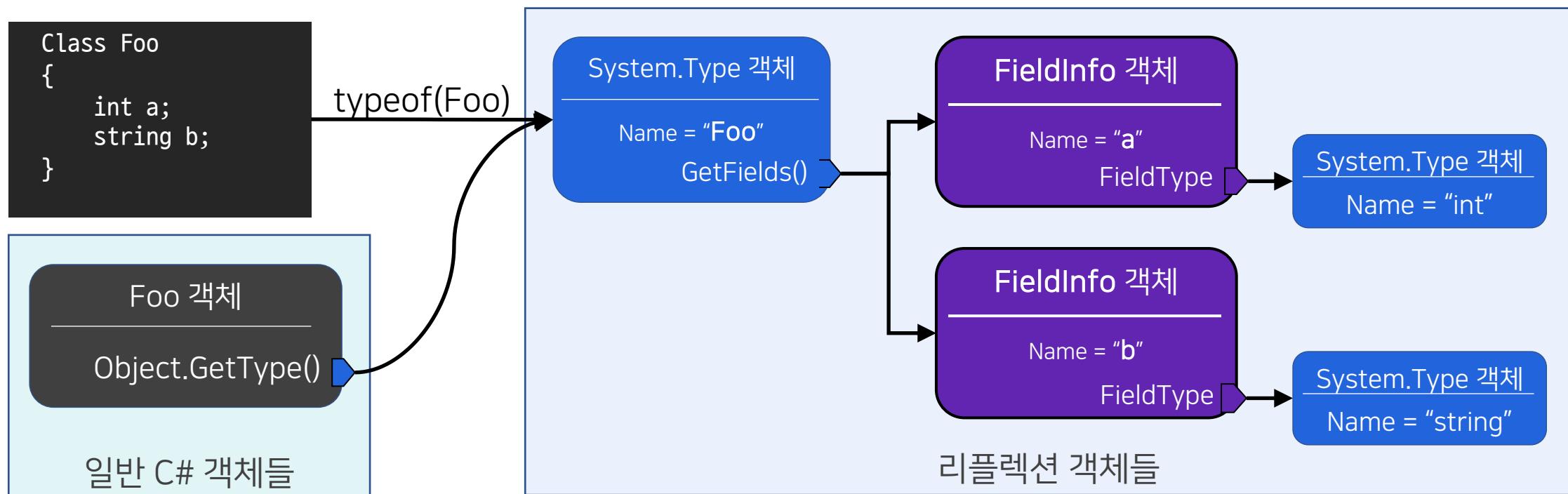
- C# 코드 상의 사용자 정의 클래스 각각마다 런타임에 System.Type 객체가 하나씩 생김
  - 정적으로(typeof(타입명)) 혹은
  - 동적으로(Object.GetType 메서드) 접근 가능



# 리플렉션이란 - C#의 리플렉션 살펴보기

NHN FORWARD ➤

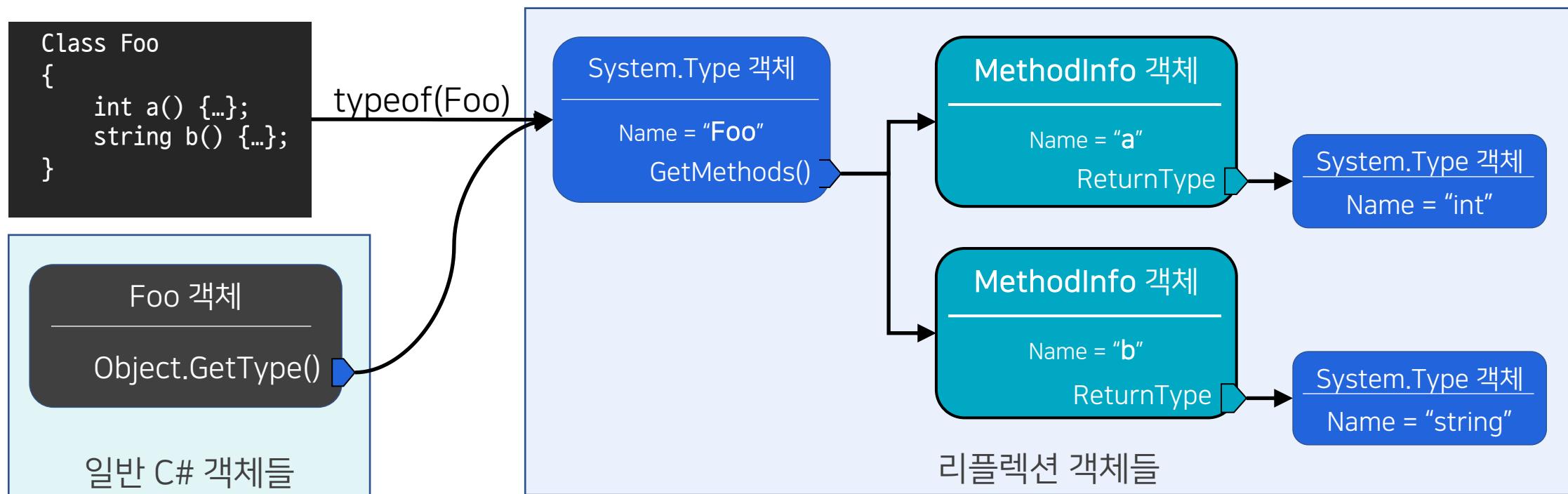
System.Reflection.FieldInfo: 각 멤버 변수를 설명하는 리플렉션 객체



# 리플렉션이란 - C#의 리플렉션 살펴보기

NHN FORWARD ➤

System.Reflection.MethodInfo: 각 멤버 함수를 설명하는 리플렉션 객체



## C#과 달리 C++은 언어 차원에서 런타임 리플렉션을 지원하지 않음

- 정적 다형성을 위한 도구는 많이 있으나 성격이 다름
- RTTI(런타임 타입 정보)가 있긴 하지만, 우리가 필요로 하는 수준에 미치지 못함



## 언리얼 엔진 외에도 리플렉션 라이브러리들이 있음

- 참고: [www.rttr.org](http://www.rttr.org)

클래스 정의  
액체에서 타입 얻어오기/  
전용 캐스팅 함수 사용을 위해 필요함  
리플렉션 정보 '등록'  
(어떤 멤버를 리플렉션 시킬 것인지 명시)

```
class MyObject : public MyBase {  
    MyObject() {};  
    void MyFunc(int, double) {};  
    int MyData;  
};  
  
RTTR_ENABLE(MyBase)  
  
RTTR_REGISTRATION  
{  
    registration::class_<MyObject>("MyObject")  
        .constructor<>()  
        .property("MyData", &MyObject::MyData)  
        .method("MyFunc", &MyObject::MyFunc);  
}
```

## UObject 류 클래스 정의

MyObject는 리플렉션 되는 클래스임을 의미

MyFunc는 리플렉션 되는 멤버 함수임을 의미

MyData는 리플렉션 되는 멤버 변수임을 의미

```
#include "MyObject.generated.h"

UCLASS()
class UMyObject : public UObject
{
    GENERATED_BODY()

    UFUNCTION()
    void MyFunc(int, double) {}

    UPROPERTY()
    int32 MyData;
};
```

## 구조체(USTRUCT)와 열거형 정의

MyEnum는 리플렉션 되는 열거형임을 의미

```
#include "MyObject.generated.h"
```

```
UENUM()
enum class EMyEnum
{
    ...
};
```

MyStruct는 리플렉션 되는 구조체임을 의미

```
USTRUCT()
struct FMyStruct
{
    GENERATED_BODY()
}
```

자동 생성된 코드가 여기에 삽입된다는 마커

```
UPROPERTY()
int32 MyData;
```

구조체의 멤버 함수는  
UFUNCTION으로 지정할 수 없음

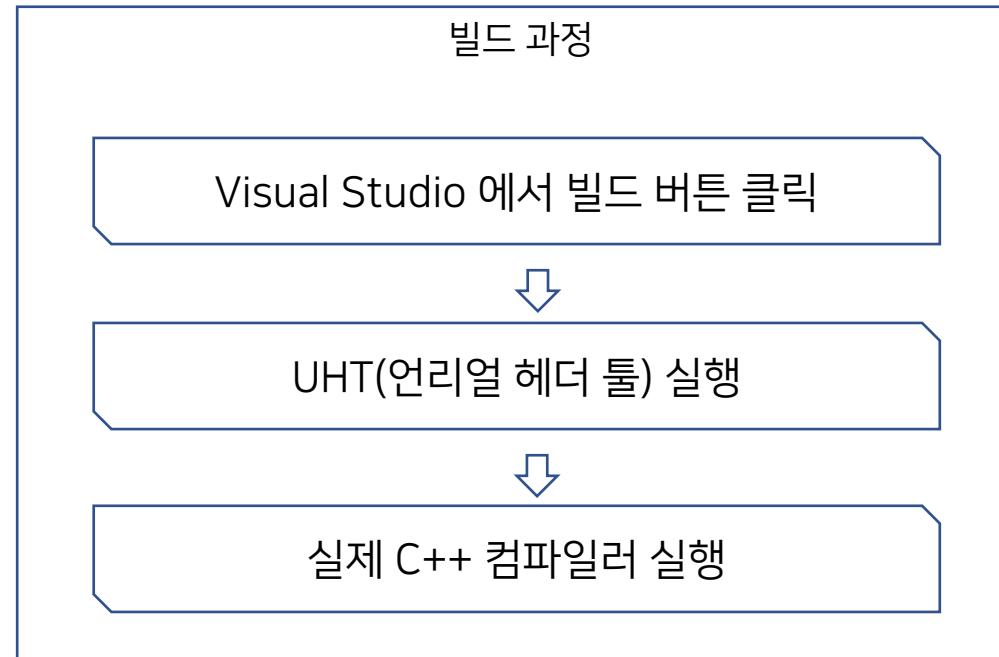
```
// UFUNCTION()
void MyFunc(double) {}
```

언리얼 C++에서는 클래스와 구조체가 명확히 구분됨

클래스(UCLASS)	구조체(USTRUCT)
주로 포인터로 다룸	주로 값으로 다룸
GC 대상	GC 대상이 아님
UObject의 자손이어야 함	부모 타입이 없어도 됨
타입 명이 U나 A로 시작해야 함	타입 명이 F로 시작해야 함

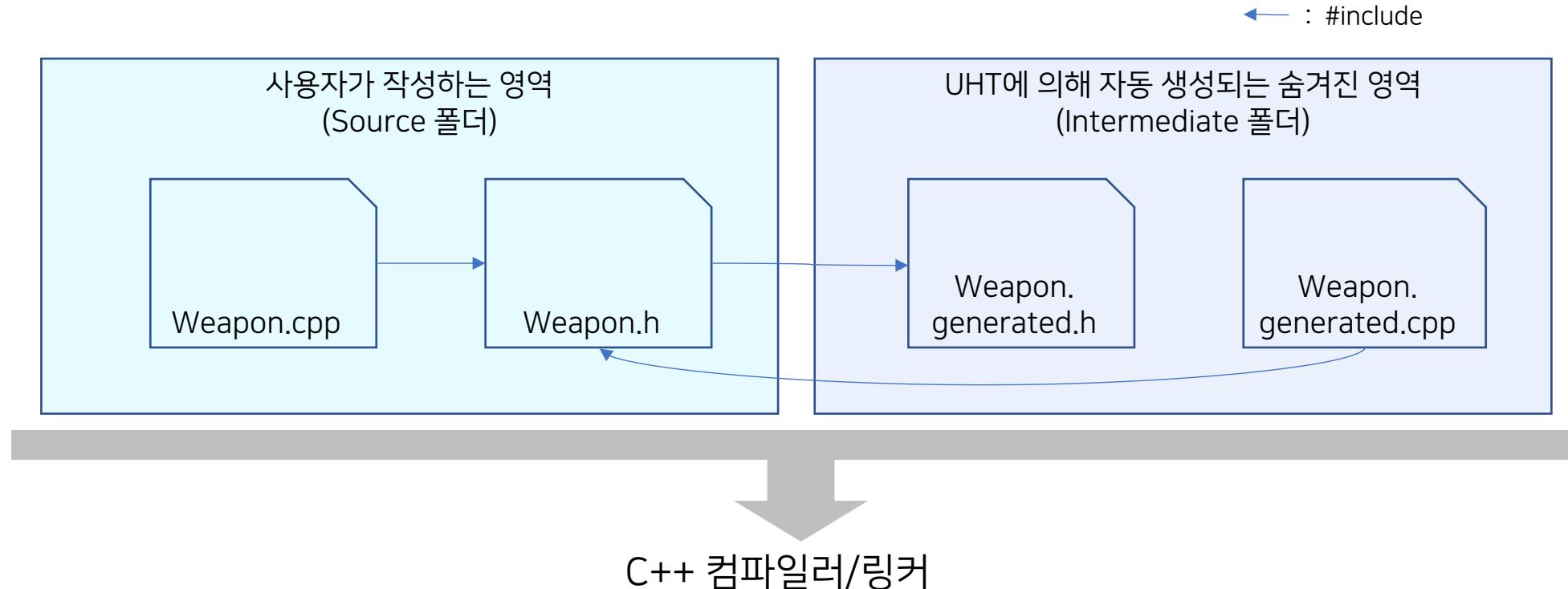
사용자가 작성한 코드는 UHT(언리얼 헤더 툴)에 의해 먼저 분석된다.

- UHT는 실제 컴파일러에 앞서서 실행되는 간이 C++ 구문 분석기
- 전체 C++ 소스코드 내에서 UClass 매크로가 붙은 클래스들을 찾아서 리플렉션 정보를 수집한다

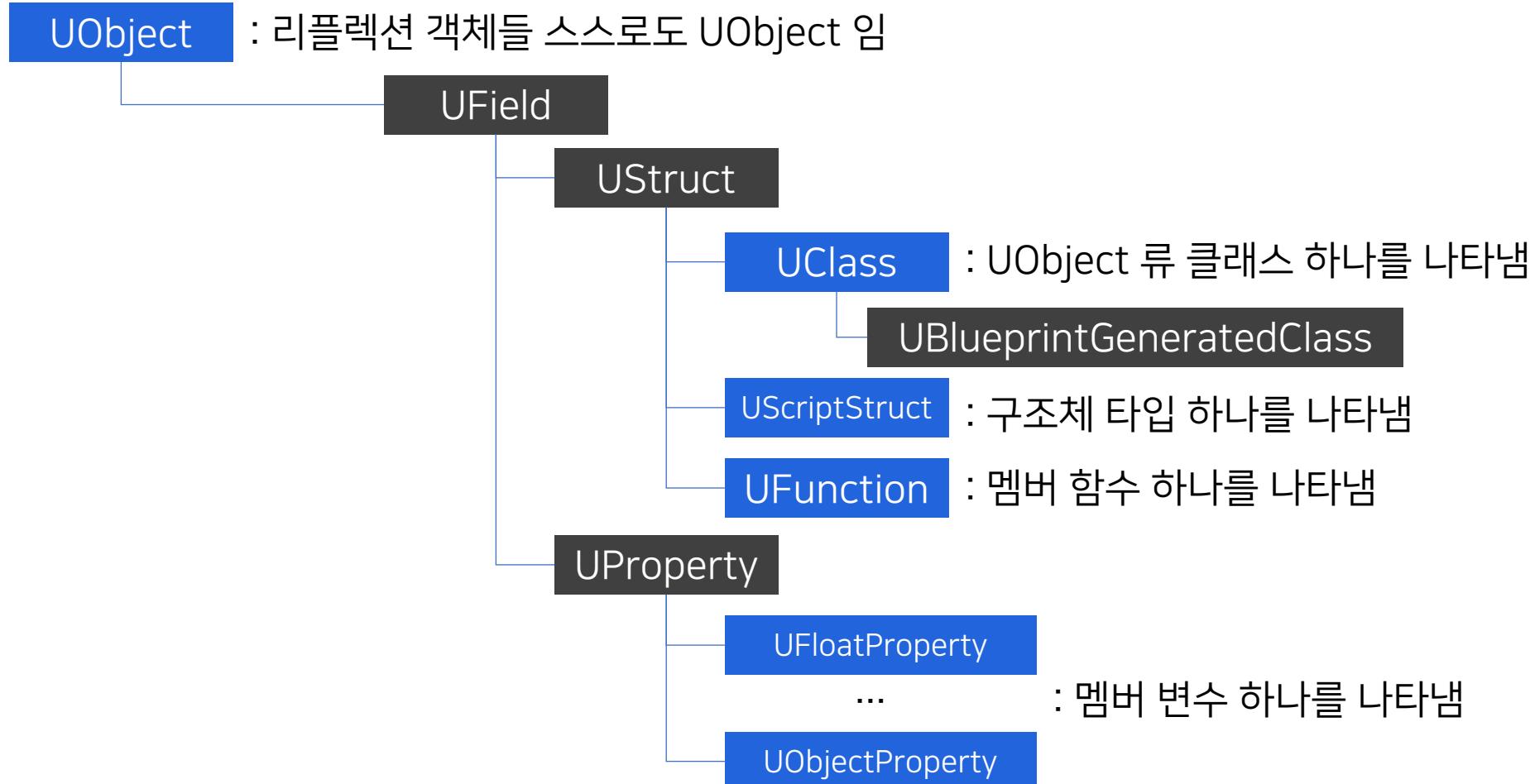


런타임에 리플렉션 객체들을 구성하기 위해 필요한 코드를 UHT가 자동 생성한다.

- 각 C++ 모듈이 로드됨과 동시에 모듈 내 클래스들에 대한 리플렉션 객체들도 적절히 초기화되어야 하는데
- 이 역할을 하는 C++ 코드는 UHT에 의해 자동 생성된다.

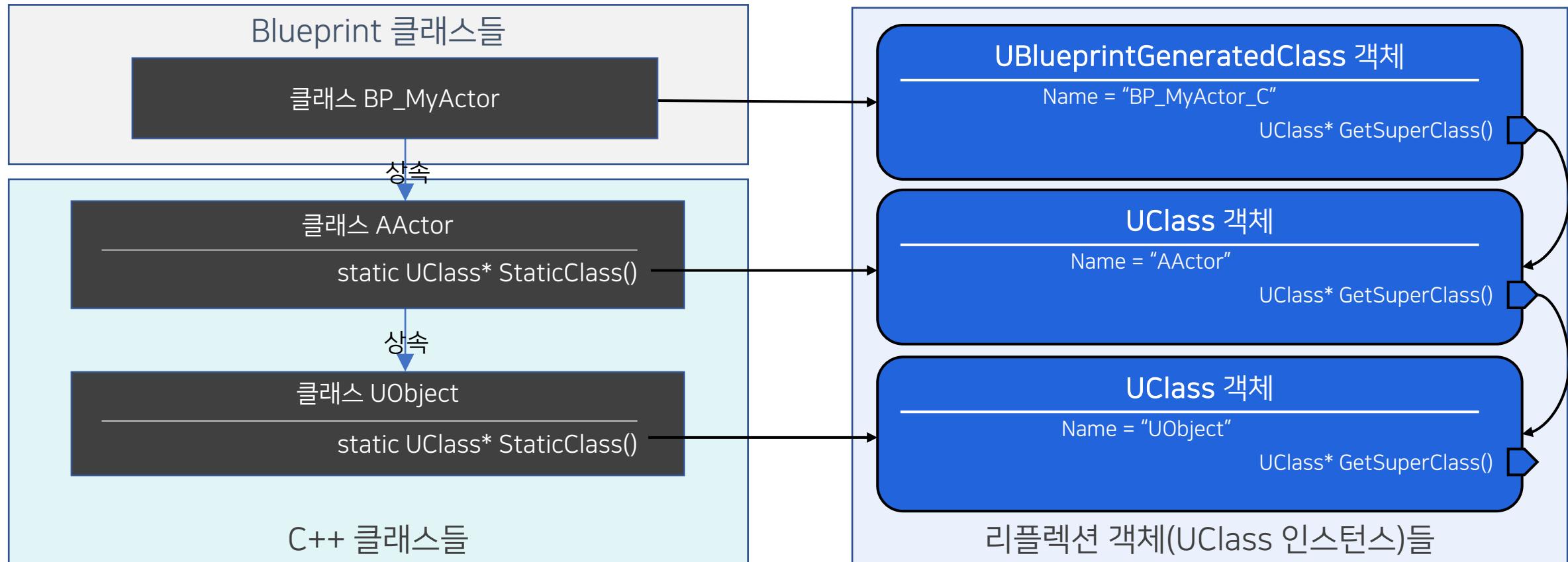


## 대표적인 언리얼 리플렉션 클래스들 간의 상속 구조



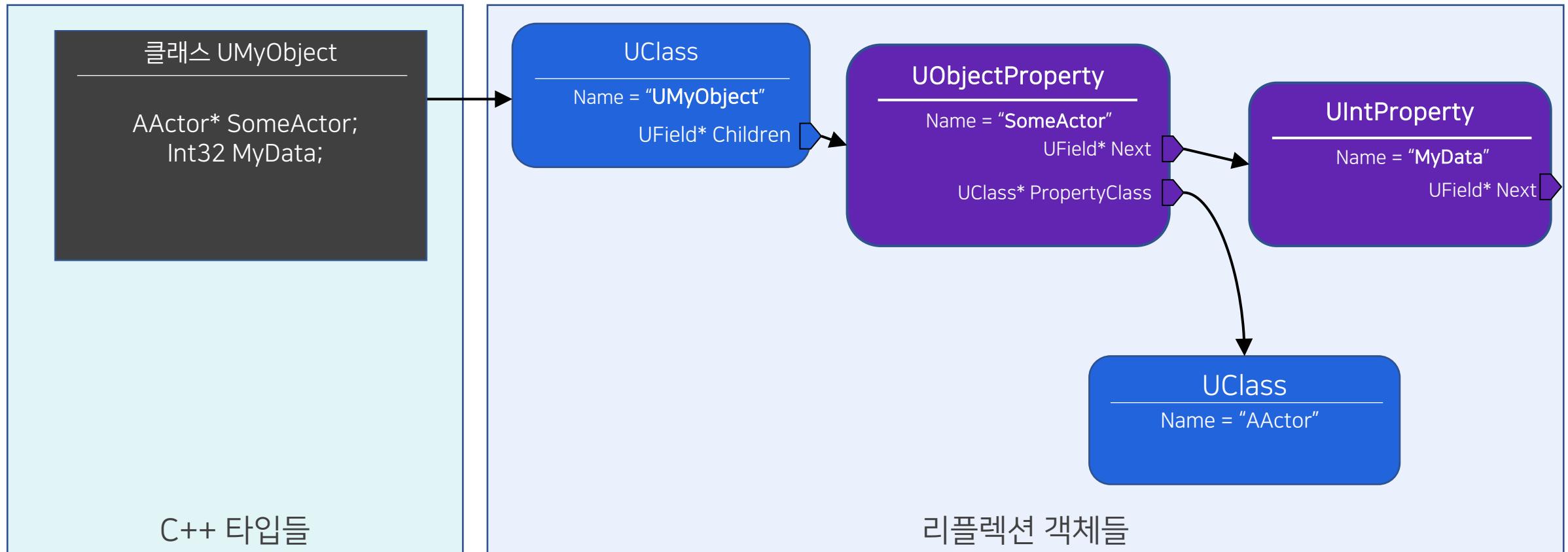
## C#의 System.Type과 같은 역할

- UObject 클래스 하나를 의미



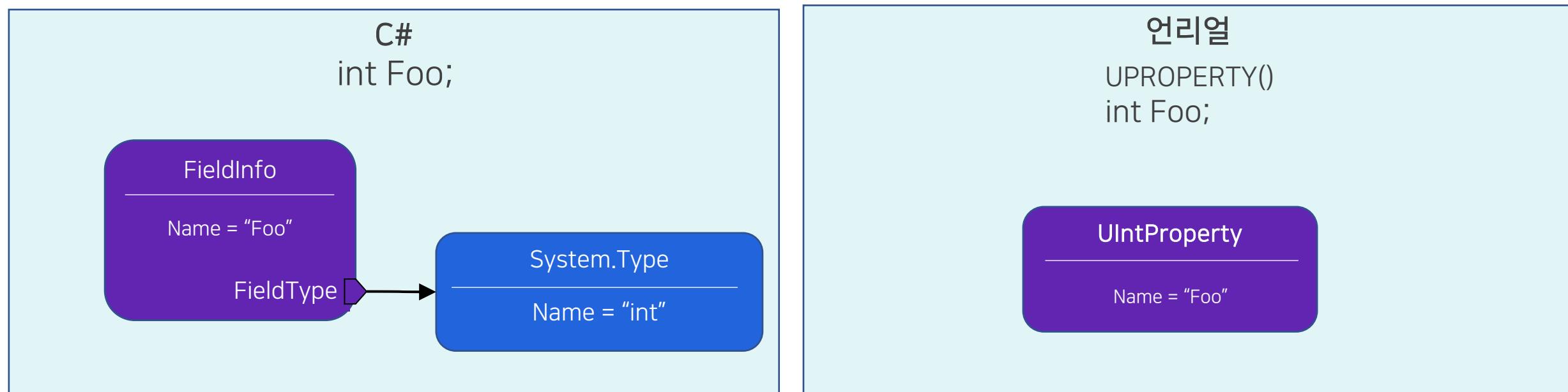
## UProperty - C#의 FieldInfo와 같은 역할

- UProperty 객체 하나가 필드 하나를 의미



## 왜 UProperty의 종류가 다양할까? - C#의 FieldInfo와 다른 점

- 변수 타입별로 UIntProperty, UObjectProperty, UMapProperty, 등등..
- 엔진의 각 시스템이 필드를 다루는 방식에 있어서 필드의 타입별로 상이한 부분을 정의
  - 두 객체가 가진 이 필드 값이 서로 동일한지 판단하는 방식
  - 에디터의 붙여넣기(Ctrl + V) - 주어진 텍스트로부터 필드의 값을 채우는 방식
  - 가비지 컬렉터가 이 필드 내에서 포인터를 찾는 방식 등등

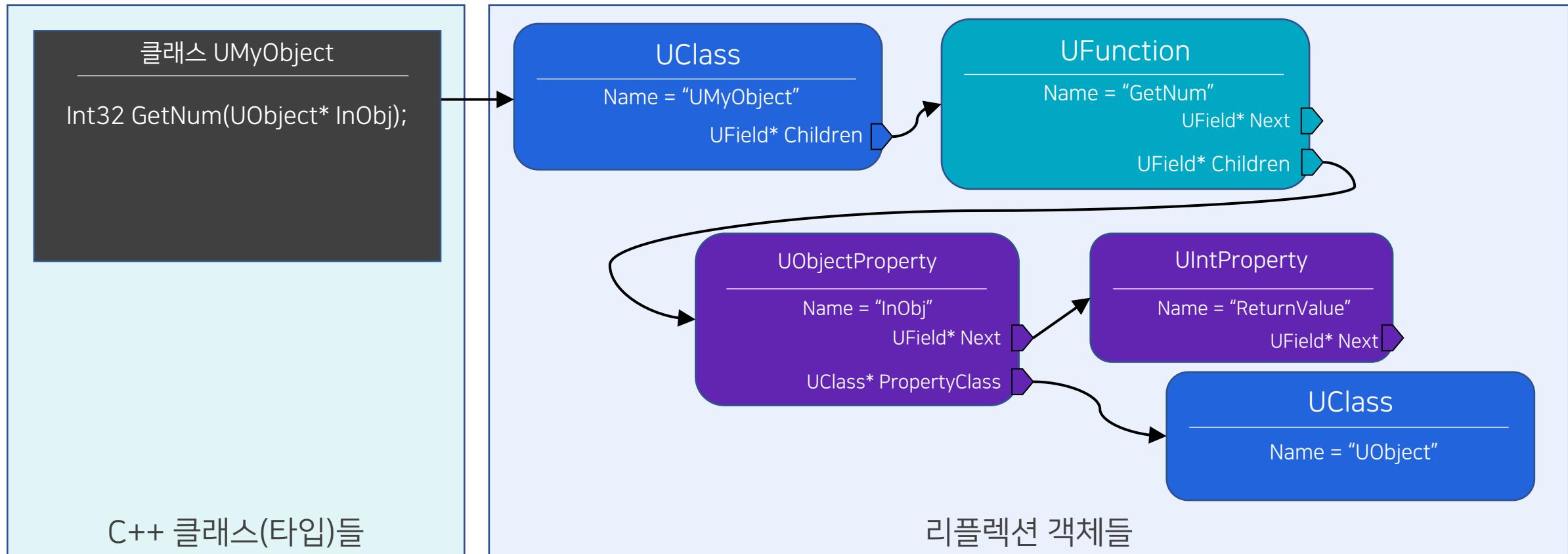


# 언리얼 리플렉션 객체, UFunction

NHN FORWARD ➤

## UFunction - C#의 MethodInfo와 같은 역할

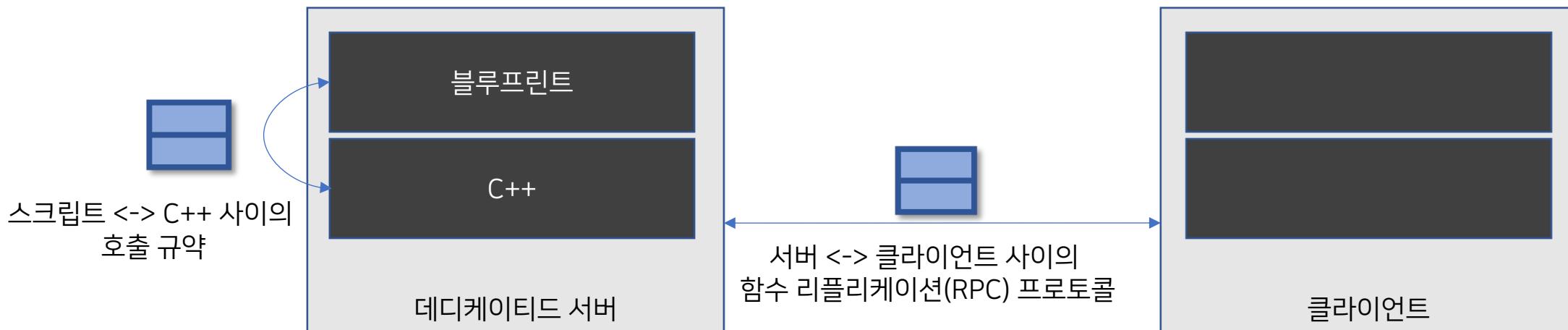
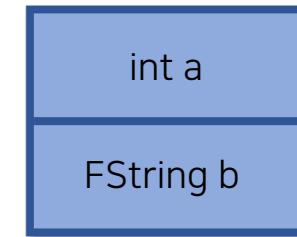
- UFunction 객체 하나가 멤버 함수 하나를 의미



## UFunction은 왜 UStruct 일까?

- 입/출력 파라미터들을 멤버(UProperty)로 갖는 구조체로 생각할 수 있음
- 구조체로서의 레이아웃 = 함수로서의 시그니처

```
UFUNCTION()
void Foo(int a, FString b)
```



언리얼 엔진 내의 여러가지 시스템들이 이 리플렉션 객체들에 의존

- 네트워크 리플리케이션
- 블루프린트 <-> C++ 연동
- 언리얼 에디터 연동
- 자동 시리얼라이제이션
- **가비지 컬렉션**

# 가비지 컬렉션

## new/delete 짹 맞추기는 프로그래머의 역할?

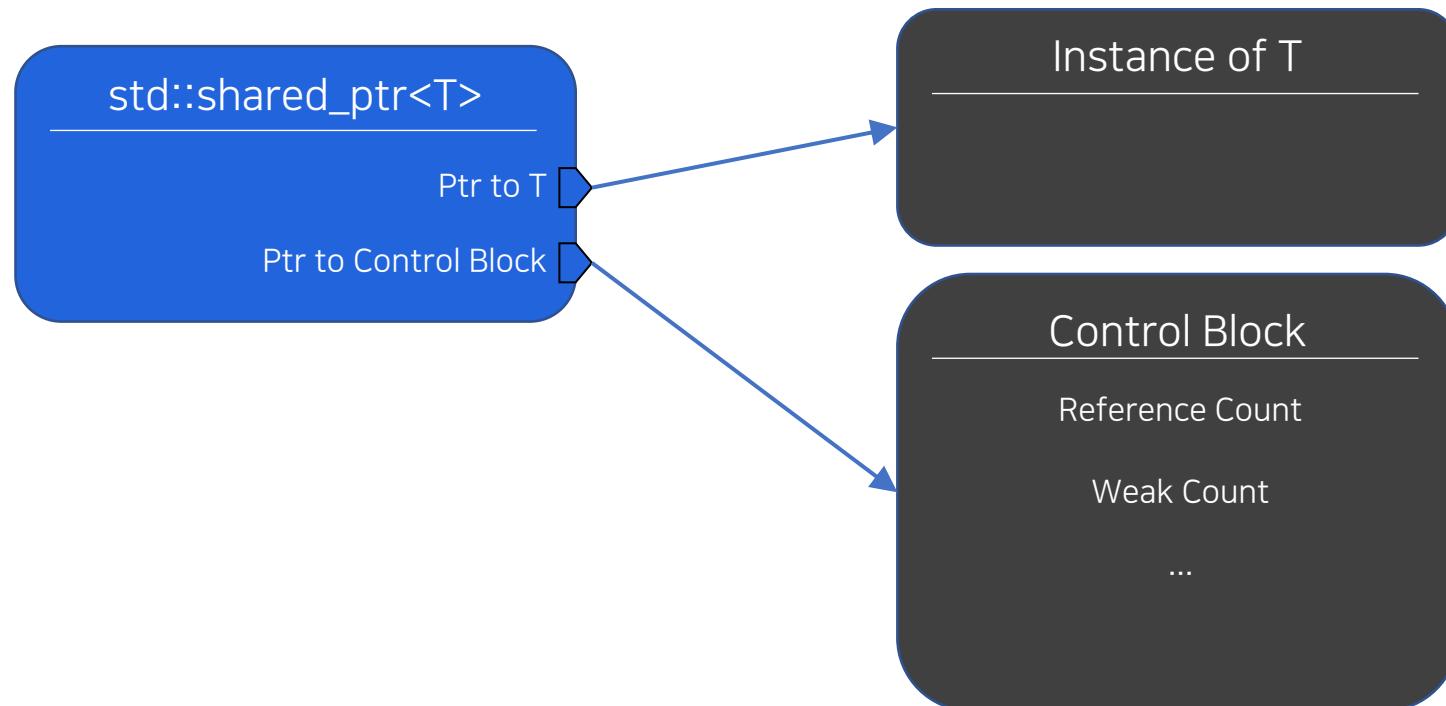
- 더 이상 사용되지 않는 객체인데 delete를 누락함(메모리 누수)
- 아직 사용 중인 객체인데 delete 해버림(허상 dangling 포인터)
- 할당되지 않은 영역에 대해 delete 해버림(미정의 동작)

필요 없어진 객체(garbage)는 알아서 delete 되면 좋겠다…

- 이러한 실수를 피할 수 있는 여러 가지 방법들 - 가비지 컬렉션

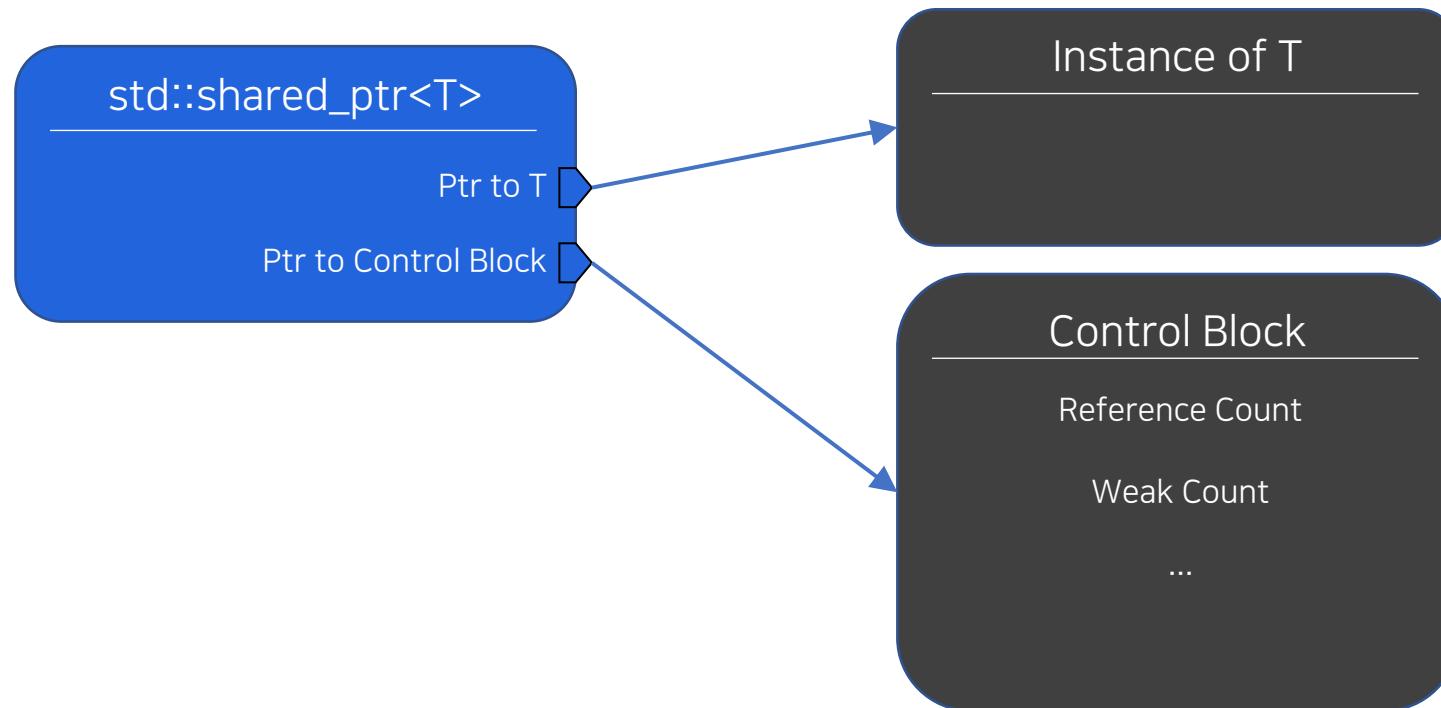
## 일반적인 C++ 프로그램에서 널리 쓰이는 단순한 방식

- Raw 포인터를 사용하는 대신 RAI 스타일의 포인터 객체를 사용(std::shared\_ptr, 언리얼의 TSharedPtr)
  - 별도로 할당된 메모리 블록에 대상 객체로의 참조 카운터를 업데이트하다가,  
참조 카운터가 0이 되면 delete를 대신 호출해주는 역할



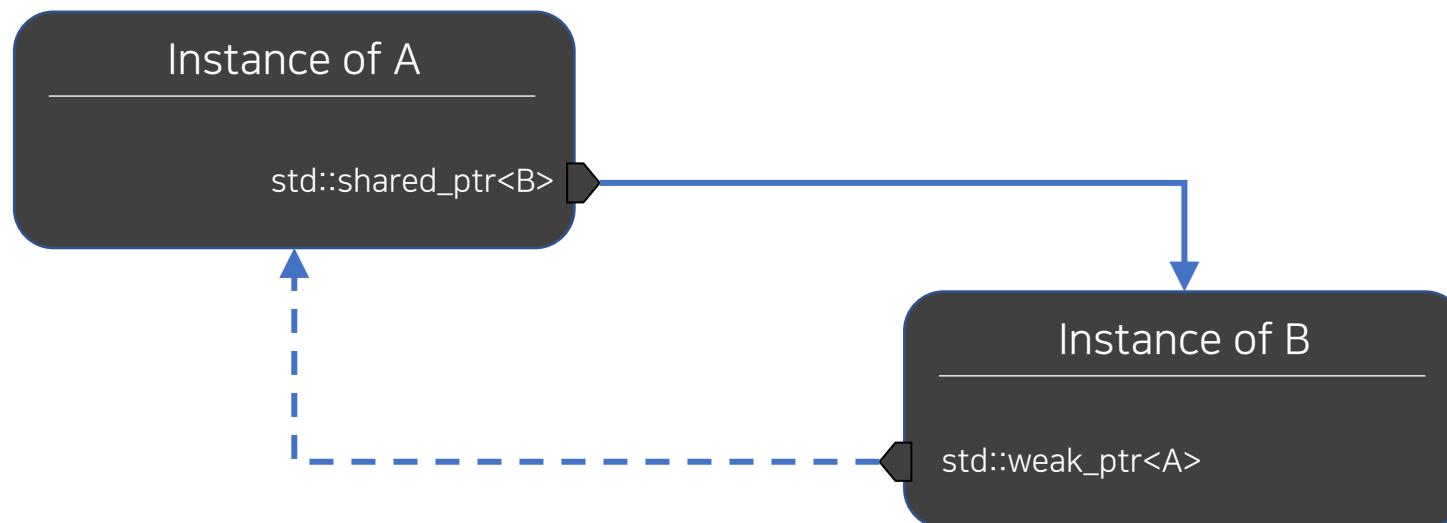
## 특징

- 개별 객체에 대한 '필요 없어짐'을 즉각적으로 감지할 수 있음
- 복사/대입/dereferencing 등의 포인터 기본 연산에 약간씩의 성능 비용이 추가됨



## 순환 참조에 주의해야 함

- 참조 카운터가 영영 0이 되지 못해 메모리 누수의 위험
- 소유권(ownership) 방향의 참조는 `shared_ptr`을 사용하고, 그 이외에는 `weak_ptr`을 사용

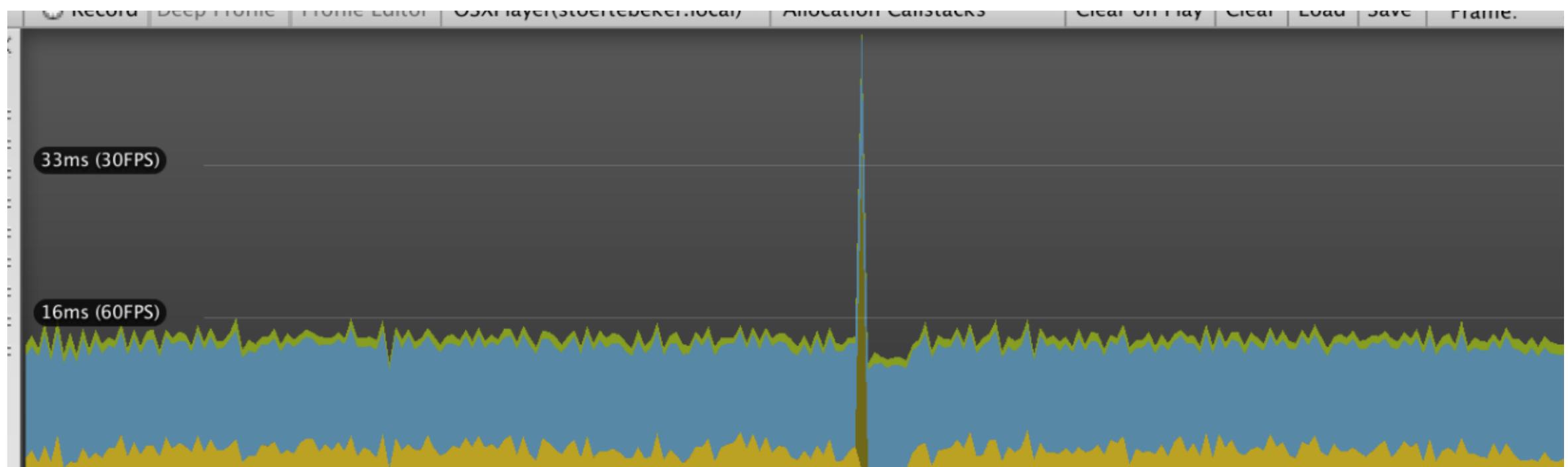


# Mark-Sweep 방식의 가비지 컬렉션

NHN FORWARD ➤

일단 해제하지 않고 놔뒀다가, 필요할 때 한 번에 정리한다.

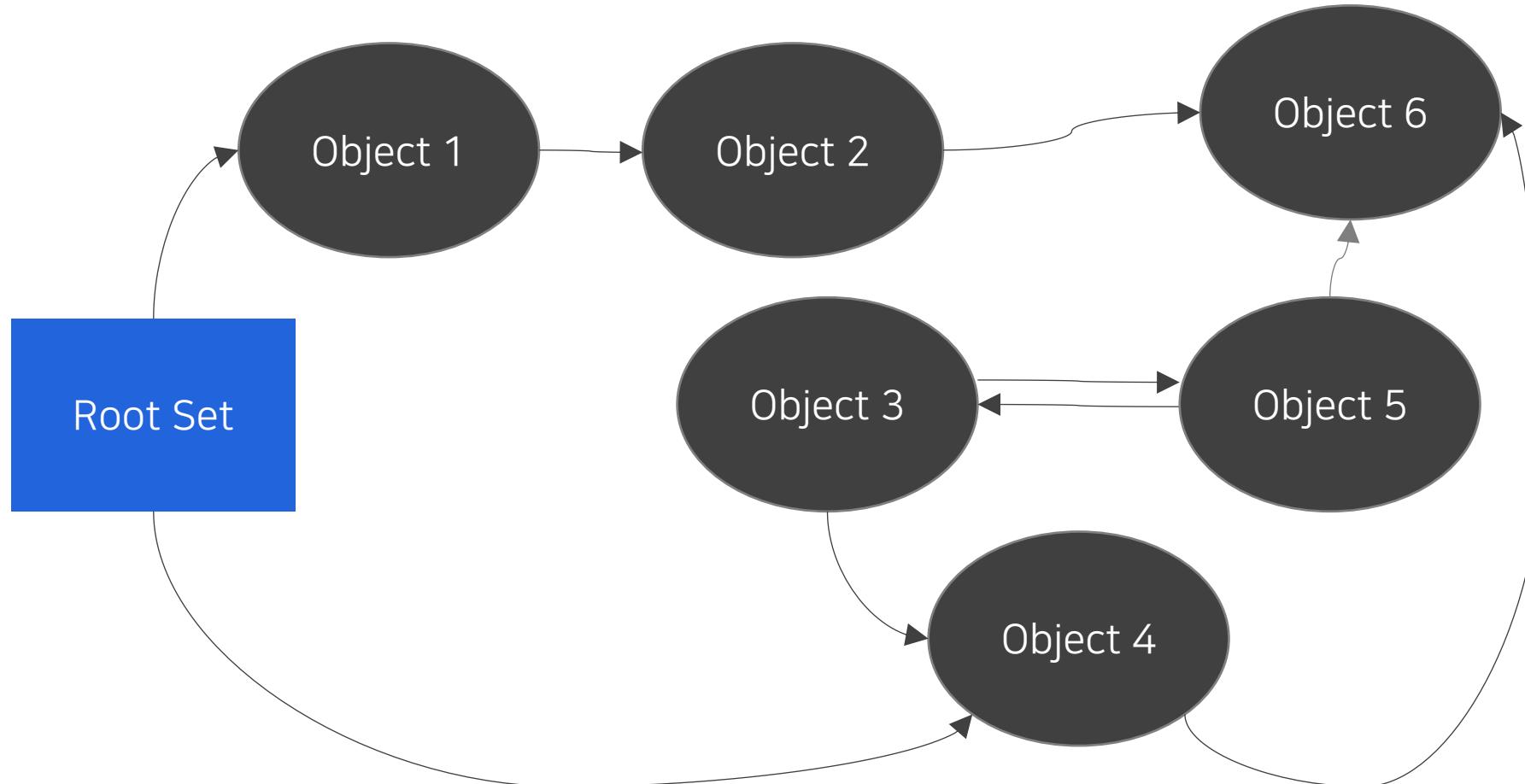
- Mark-Sweep 알고리즘의 한 사이클은 Mark 단계와 Sweep 단계로 나누어짐



# Mark-Sweep 방식의 가비지 컬렉션

NHN FORWARD ➤

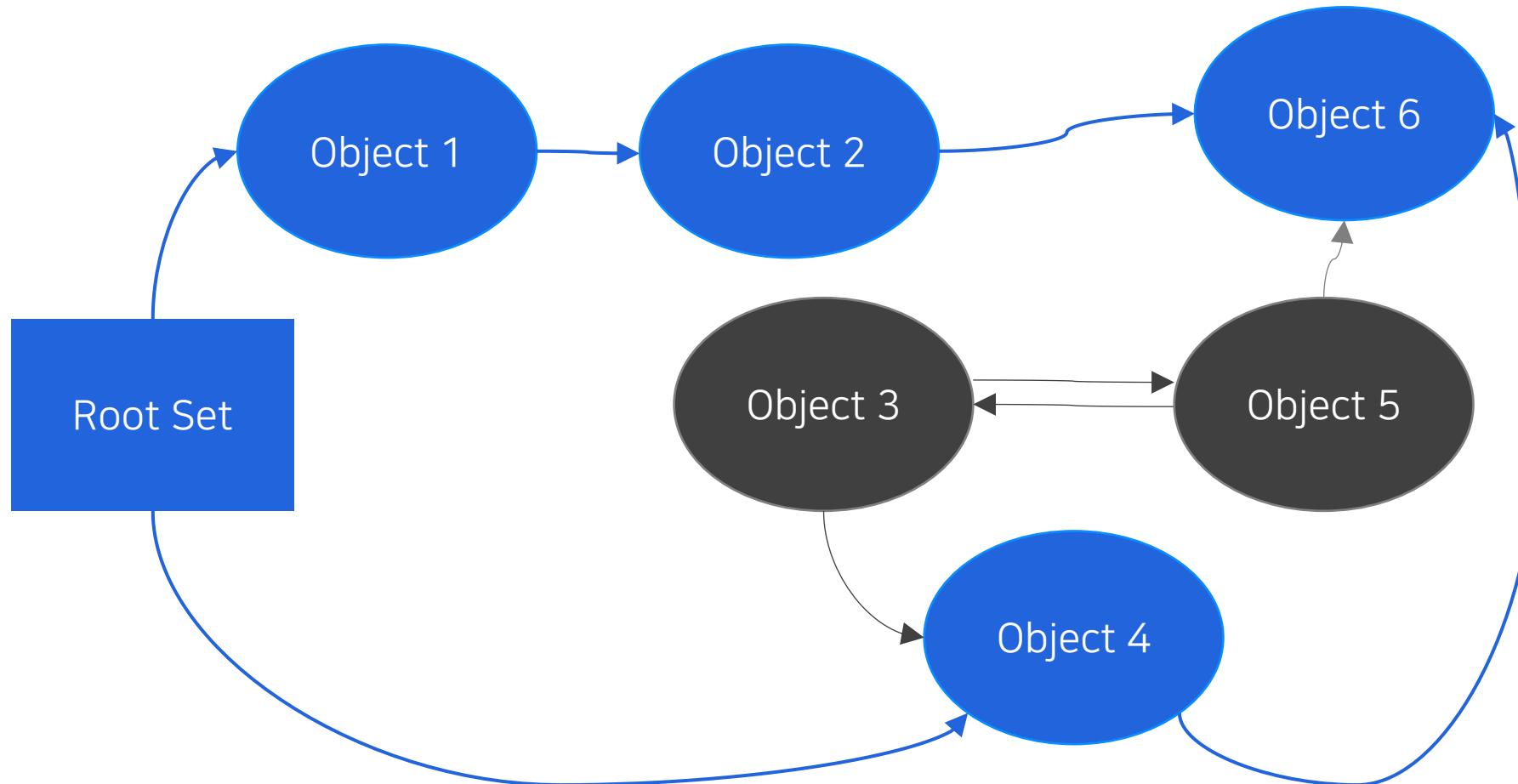
준비 단계 - 모든 객체의 도달 가능 마킹 해제



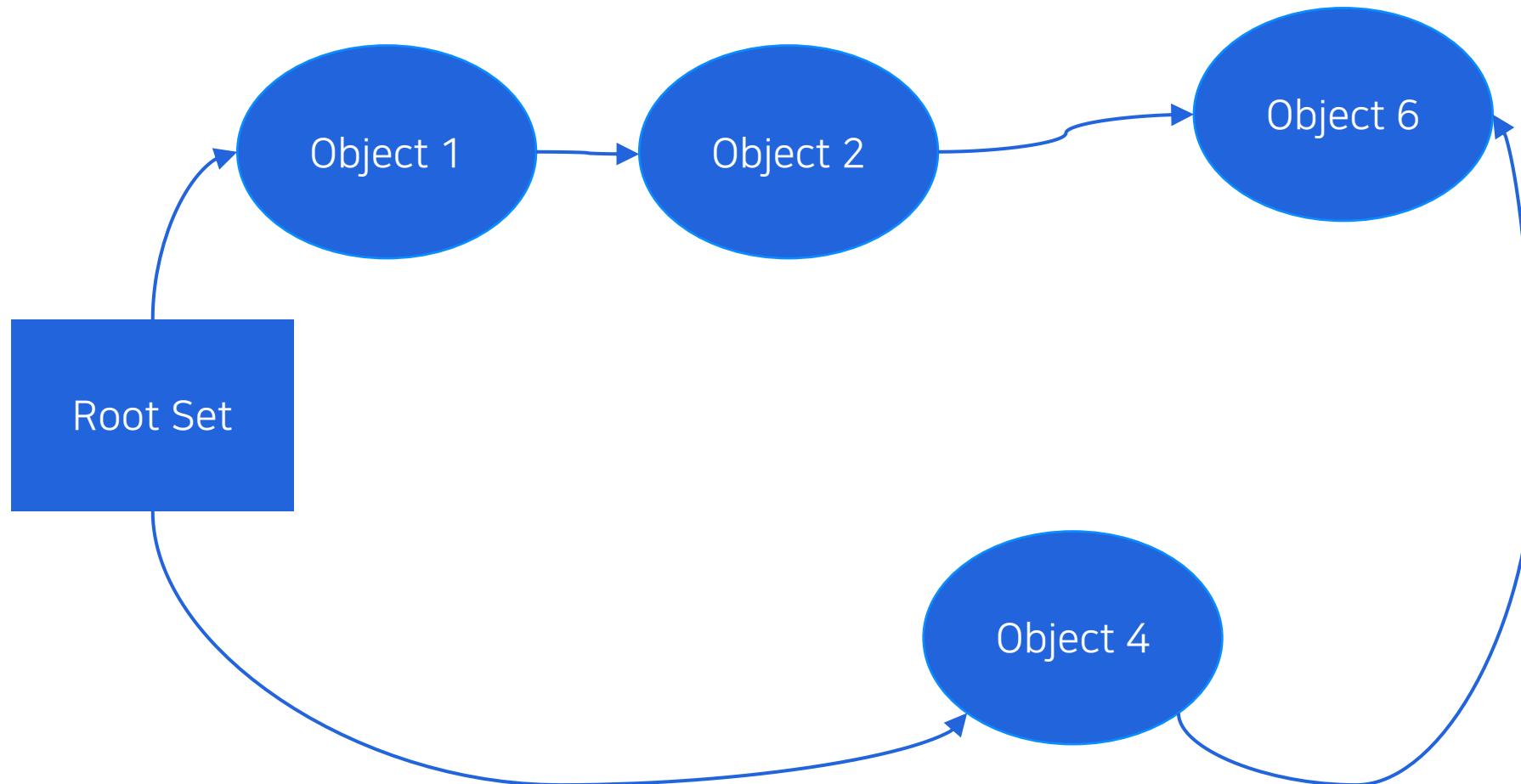
# Mark-Sweep 방식의 가비지 컬렉션

NHN FORWARD ➤

Mark 단계 - Root Set으로부터 도달 가능한 객체 마킹



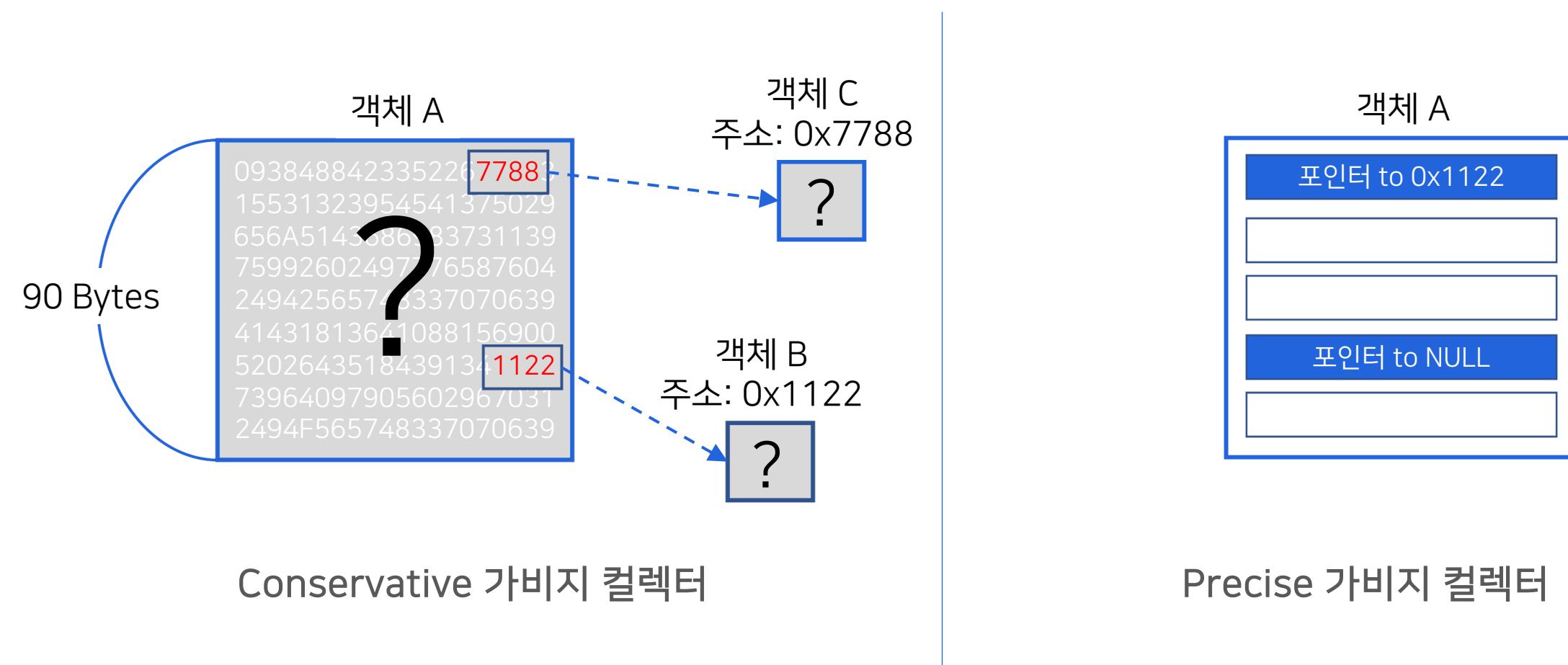
Sweep 단계 - 프로그램이 도달 불가능한(=마킹되지 않은) 객체 해제



# Mark-Sweep 방식의 가비지 컬렉션

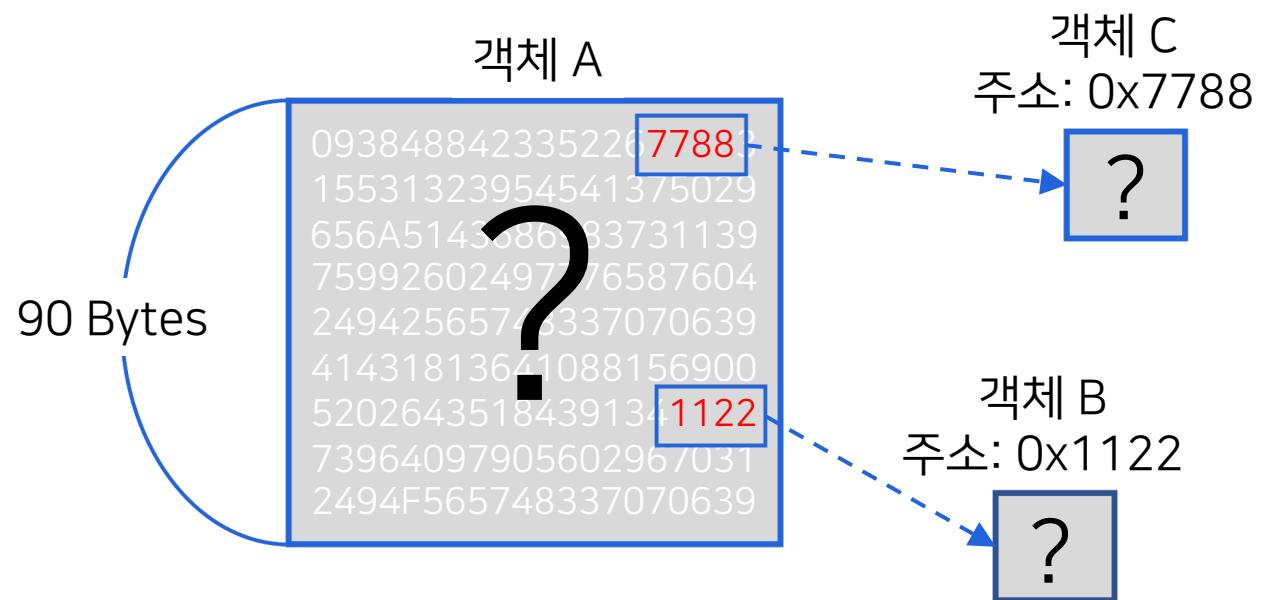
NHN FORWARD ➤

C++에서의 GC – 객체 내의 어느 부분이 포인터인가?



## Conservative 가비지 컬렉터

- 객체의 구체적인 생김새를 모르는 채로 동작
  - 크기만 주어진 메모리 블록으로 취급
- 메모리 블록 내의 어느 부분이든 잠재적으로 포인터일 가능성이 있음
  - '내가 할당해준 객체로의 주소' 가 등장하면 모두 포인터로 간주



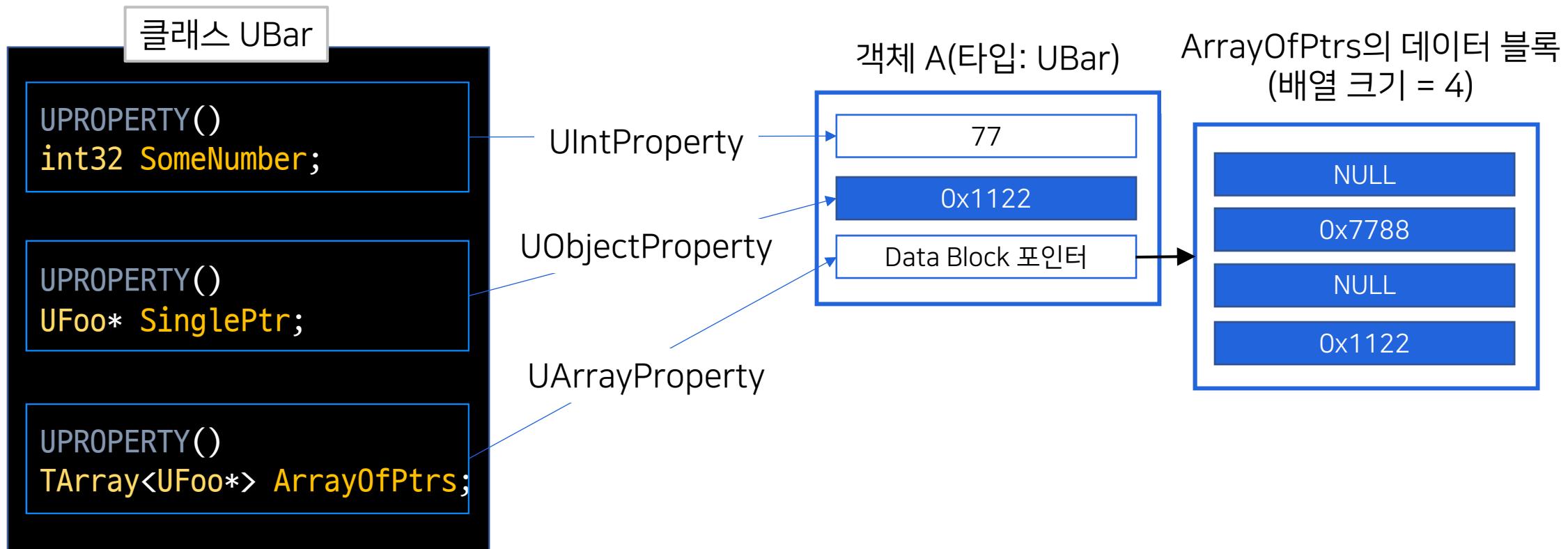
## Boehm-Demers-Weiser GC

- Root Set은 프로그램의 정적 데이터 영역, 각 스레드 별 레지스터와 스택 메모리
- Stop the world: GC를 제외한 모든 스레드를 멈춤
- Non-Moving
- Generational 및 Incremental 옵션
- 유니티 엔진에서 사용 중



언리얼은 리플렉션 정보를 활용해 객체 내에서 UObject 포인터의 위치를 알아냄

- 앞서 얘기한 기준에서 Precise 방식으로 분류할 수 있음
- UProperty 객체가 해당 필드 내의 모든 UObject 포인터 위치를 GC에게 알려줌



## 객체 내의 모든 포인터가 리플렉션 되는 건 아닐 텐데?

- UObject의 필드 중에도 리플렉션 되지 않은 UObject로의 포인터가 얼마든지 있을 수 있음
- 언리얼 GC는 사용자에게 이런 포인터들의 위치를 추가적으로 명시할 방법을 제공

클래스 UBar

```
UPROPERTY()           // OK
UFoo* SinglePtr;

UPROPERTY()           // OK
TArray<UFoo*> ArrayOfPtrs;

UPROPERTY()           // 빌드 실패 (UHT 에러: TCircularBuffer라는 자료구조는 알지 못합니다!)
TCircularBuffer<UFoo*> CircularBufferOfPtrs;

UPROPERTY()           // 빌드 실패 (UHT 에러: 중첩된 배열은 UPROPERTY로 지정할 수 없습니다!)
TArray< TArray<UFoo*> > ArrayOfArrayOfPtrs;
```

AddReferencedObjects - 리플렉션 정보가 없는 포인터에 대해 GC에게 알려주기 위한 통로

## 클래스 UBar

```
// UHT 에러로 인해 이 멤버 변수에는 UPROPERTY()를 붙일 수 없다
TArray< TArray<UFoo*> > ArrayOfArrayOfPtrs;

// 그럴 땐 이렇게!
static void AddReferencedObjects(UObject* InThis, FReferenceCollector& InCollector)
{
    UBar* This = Cast<UBar>(InThis);
    for (auto& ArrayOfPtrs : This->ArrayOfArrayOfPtrs)
    {
        InCollector.AddReferencedObjects(ArrayOfPtrs, This);
    }

    Super::AddReferencedObjects(InThis, InCollector);
}
```

## 대상 객체가 해제되어도 무방하다면?

- Raw 포인터(리플렉션 된)는 강한 참조이기 때문에, Raw 포인터가 가리키고 있는 동안은 대상 객체가 해제되지 않음
- 이를 원치 않는 경우 약한 참조 TWeakObjectPtr를 사용한다
  - IsValid()를 통해 대상 객체가 해제되었는지 확인 가능

TWeakPtr과 혼동하지 않도록 주의  
(순환 참조를 없애기 위한 용도가 아님)



## 네이티브 C++ 객체 내에 UObject 포인터가 있을 경우 FGCOBJECT를 상속

- 그러면 AddReferencedObjects를 override하여 UObject 포인터를 GC에게 알려줄 수 있음
- 이 네이티브 C++ 객체가 GC에 의해 해제된다는 의미는 아님!
  - 이 객체를 단순히 Root Set으로 지정하는 효과

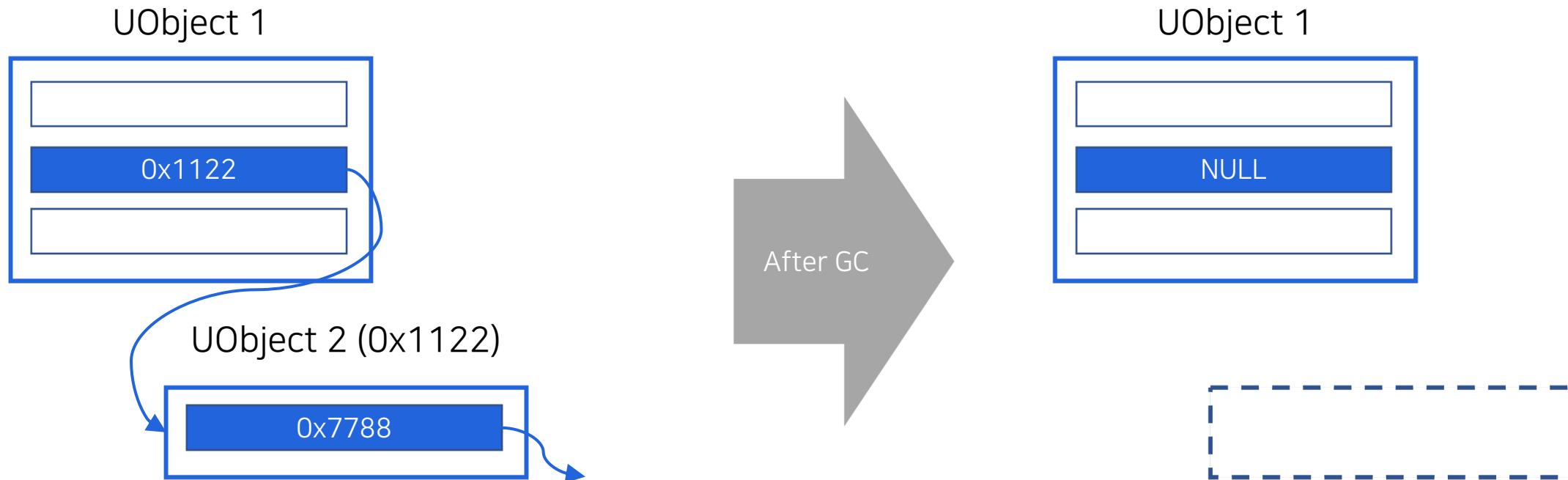
## FGCObject를 상속받지 못하는 경우에는?

- Raw 포인터 대신 TStrongObjectPtr 혹은 TWeakObjectPtr 포인터 객체를 사용
- 예: 타이머나 HTTP 요청 등의 비동기 콜백으로써 람다 객체를 넘기는 경우
  - TWeakObjectPtr을 지역 변수로 두고 이것을 캡처하자
  - 람다 함수 내에서 멤버 변수/함수에 접근하는 경우 this 포인터가 암시적으로 캡처되는 것에 주의

```
TWeakObjectPtr<UMyObject> WeakThis(this);
GetWorld()->GetTimerManager().SetTimer(TimerHandle, FTimerDelegate::CreateLambda(
    [WeakThis] {
        if (!WeakThis.IsValid()) return;
    } , 3.0f, false);
```

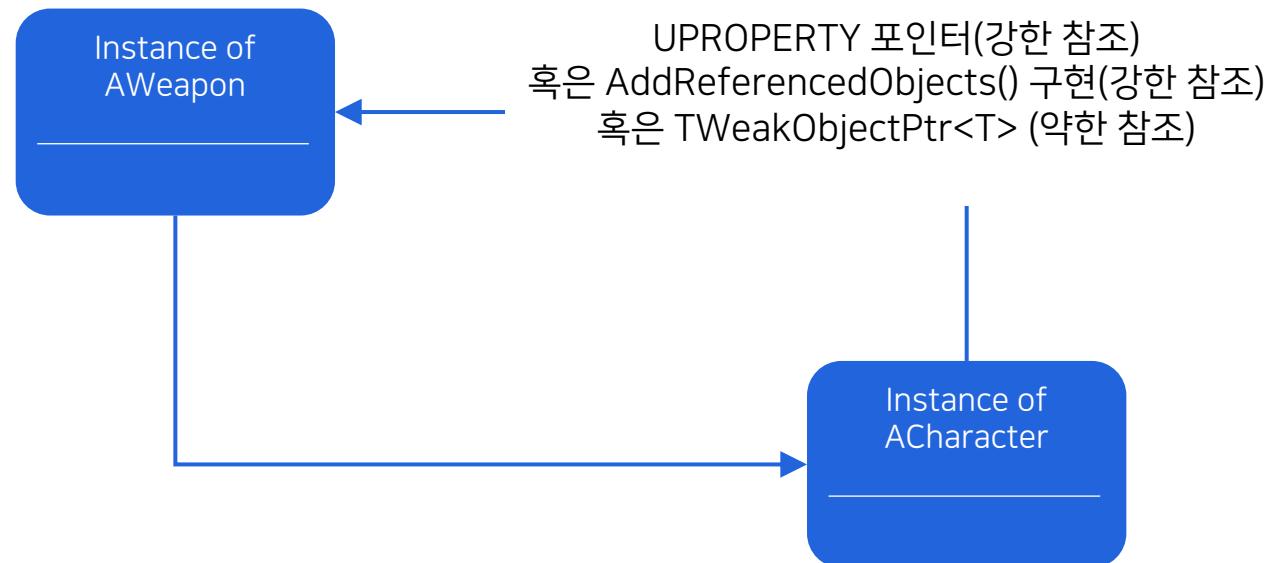
## 객체를 강제로 해제할 순 없나요?

- 도달 가능한 UObject 객체이더라도 MarkPendingKill 호출로 인해 강제 해제될 수 있음
- 가비지 컬렉터는 해당 객체를 가리키는 모든 포인터 변수의 값을 안전하게 NULL로 업데이트해줌



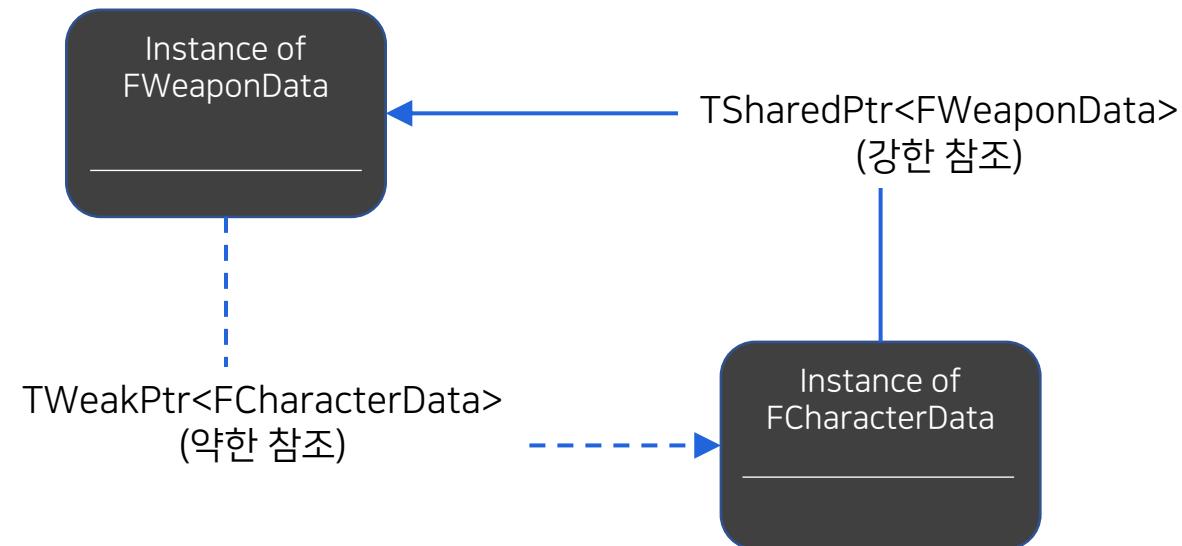
## UObject 객체 간의 참조

- 리플렉션 된 UPROPERTY 포인터로 참조



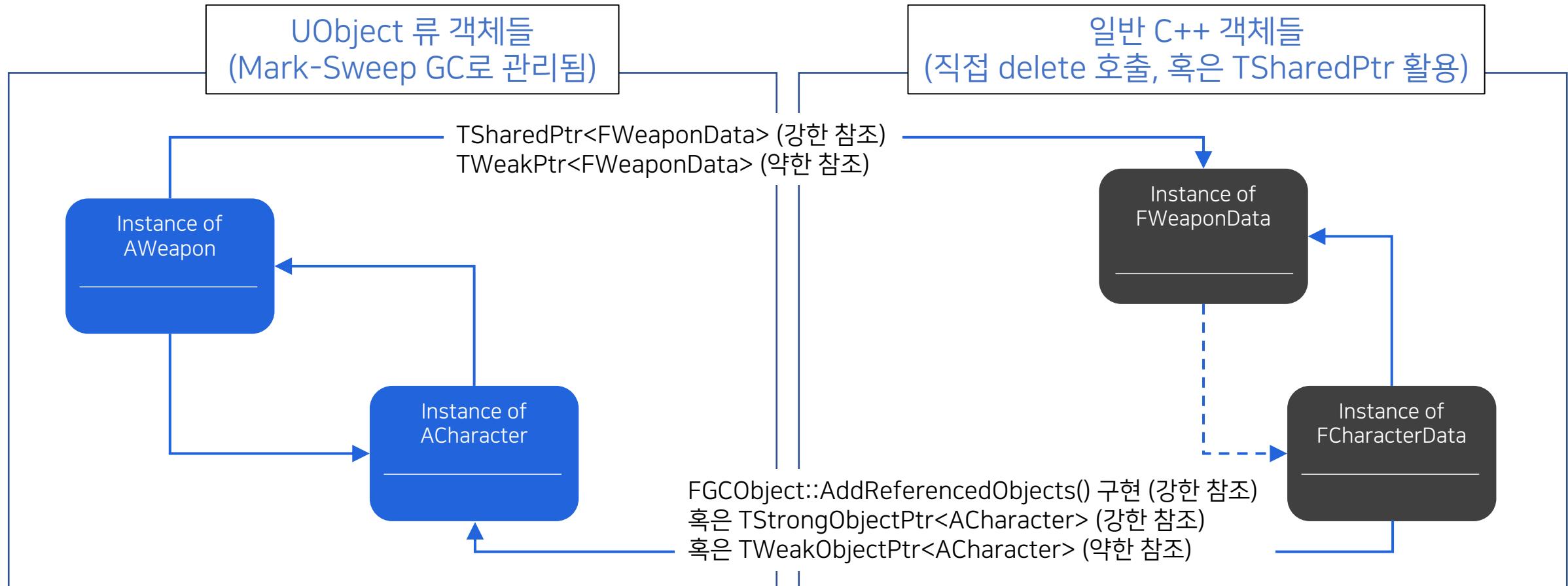
## 비 UObject 객체 간의 참조

- 직접 delete 호출, 혹은 TSharedPtr 활용
- 순환 참조에 주의 (TSharedPtr 은 소유권을 의미)



## 두 종류 객체의 메모리 관리 방식을 적절히 혼용

- 비 UObject 객체가 포함된 강한 참조 Cycle = 메모리 누수 위험!



## UObject 파생 클래스에는 런타임 리플렉션 정보가 제공된다

- UClass 와 UProperty 객체를 기억하자

## 가비지 컬렉터는 객체간의 참조관계를 파악하기 위해 리플렉션 정보를 최대한 활용한다

- 그래서 리플렉션 정보가 존재하는 UObject 포인터는 GC가 자동으로 인식한다
- 리플렉션 되지 않은 영역에 대해서 별도의 처리가 필요하다

# Q&A

# THANK YOU