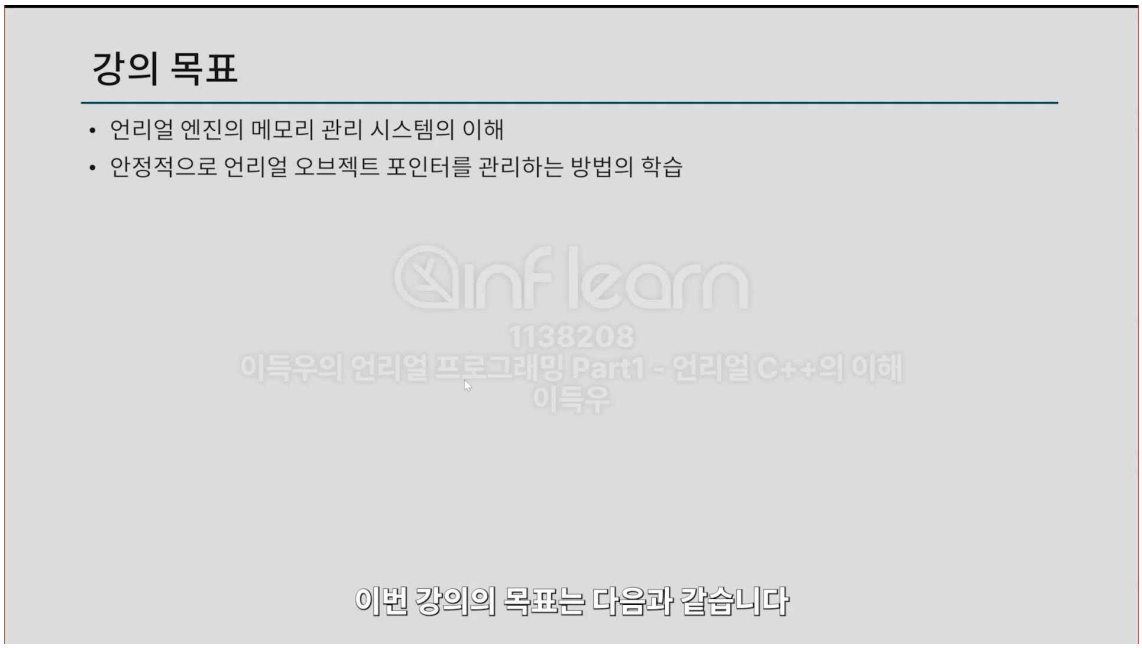
언리얼 프로그래밍 Part1-12

제목:언리얼 엔진의 메모리 관리

**강의 내용 : 언리얼 엔진의 메모리 관리 방식을 파악하고, 언리얼 오브젝트의 메모리를 관리하는 예제 실습

**강의 목표 :



***언리얼 엔진의 자동 메모리 관리

**C++언어 메모리 관리의 문제점

C++ 언어 메모리 관리의 문제점

- C++은 저수준으로 메모리 주소에 직접 접근하는 포인터를 사용해 오브젝트를 관리한다.
- 그러다보니 프로그래머가 직접 할당(new)과 해지(delete) 짝 맞추기를 해야 한다.
- 이를 잘 지키지 못하는 경우 다양한 문제가 발생할 수 있음.
- 잘못된 포인터 사용 예시
  - 메모리 누수(Leak) : new를 했는데 delete 짝을 맞추지 못함. 힙에 메모리가 그대로 남아있음.
  - 허상(Dangling) 포인터 : (다른 곳에서) 이미 해제해 무효화된 오브젝트의 주소를 가리키는 포인터
  - 와일드(Wild) 포인터 : 값이 초기화되지 않아 엉뚱한 주소를 가리키는 포인터.
- 잘못된 포인터 값은 다양한 문제를 일으키며, 한 번의 실수는 프로그램을 종료시킴.
- 게임 규모가 커지고 구조가 복잡해질수록 프로그래머가 실수할 확률은 크게 증가한다.

C++ 이후에 나온 언어 Java/C# 은 이런 고질적인 문제를 해결하기 위해

포인터를 버리고 대신 가비지 컬렉션 시스템을 도입함.

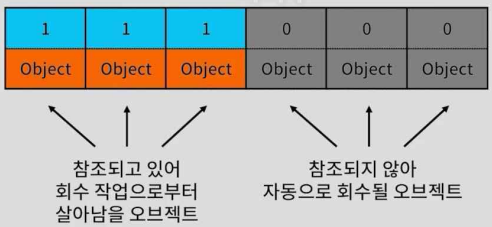
메모리 주소에 직접 접근하는 포인터를 사용해서

**가비지 컬렉션 시스템

가비지 컬렉션 시스템

- 프로그램에서 더 이상 사용하지 않는 오브젝트를 자동으로 감지해 메모리를 회수하는 시스템.
- 동적으로 생성된 모든 오브젝트 정보를 모아둔 저장소를 사용해 사용되지 않는 메모리를 추적
- 마크-스윕(Mark-Sweep) 방식의 가비지 컬렉션
  1. 저장소에서 최초 검색을 시작하는 루트 오브젝트를 표기한다.
  2. 루트 오브젝트가 참조하는 객체를 찾아 마크(Mark)한다.
  3. 마크된 객체로부터 다시 참조하는 객체를 찾아 마크하고 이를 계속 반복한다.
  4. 이제 저장소에는 마크된 객체와 마크되지 않은 객체의 두 그룹으로 나뉜다.
  5. 가비지 컬렉터가 저장소에서 마크되지 않은 객체(가비지)들의 메모리를 회수한다. (Sweep)

메모리를 회수하는 시스템을 의미합니다



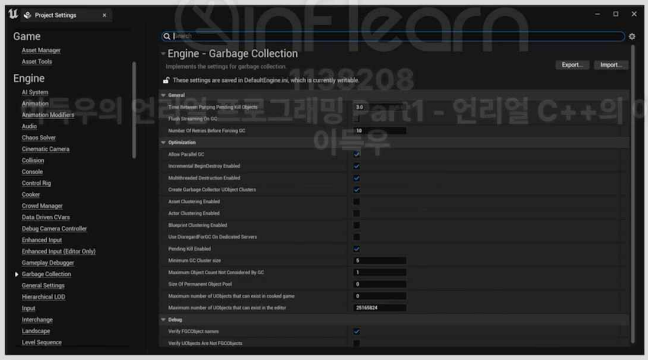
-루트 객체로부터 참조되는 객체는 1번이라 표시하고 참조되지 않은 객체는 기본값인 0번을 가짐

-모든 값을 다 파악하면 가비지 컬렉터가 동작할 때 0번인 객체들은 메모리로부터 회수함.

**언리얼 엔진의 가비지 컬렉션 시스템

언리얼 엔진의 가비지 컬렉션 시스템

- 마크-스윕 방식의 가비지컬렉션 시스템을 자체적으로 구축함.
- 지정된 주기마다 몰아서 없애도록 설정되어 있음. (GCCycle. 기본 값 60초)
- 성능 향상을 위해 병렬 처리, 클러스터링과 같은 기능을 탑재함.

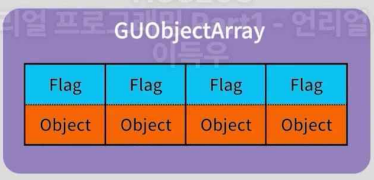


-언리얼엔진은 프로젝트 설정에 있는 주기마다 몰아서 마크가 안된 오브젝트들을 없앤다.
-가비지 컬렉터가 백그라운드에서 진행하는 작업이 부하가 작진 않지만, 언리얼 엔진의 경우엔 퍼포먼스를 높이기 위해 병렬처리, 클러스터링을 탑재함(옵션에서 다양한 설정 가능)

**가비지 컬렉션을 위한 객체 저장소

가비지 컬렉션을 위한 객체 저장소

- 관리되는 모든 언리얼 오브젝트의 정보를 저장하는 전역 변수 : GUObjectArray
- GUObjectArray의 각 요소에는 플래그(Flag)가 설정되어 있음.
- 가비지 컬렉터가 참고하는 주요 플래그
  - Garbage 플래그 : 다른 언리얼 오브젝트로부터의 참조가 없어 회수 예정인 오브젝트
  - RootSet 플래그 : 다른 언리얼 오브젝트로부터 참조가 없어도 회수하지 않는 특별한 오브젝트

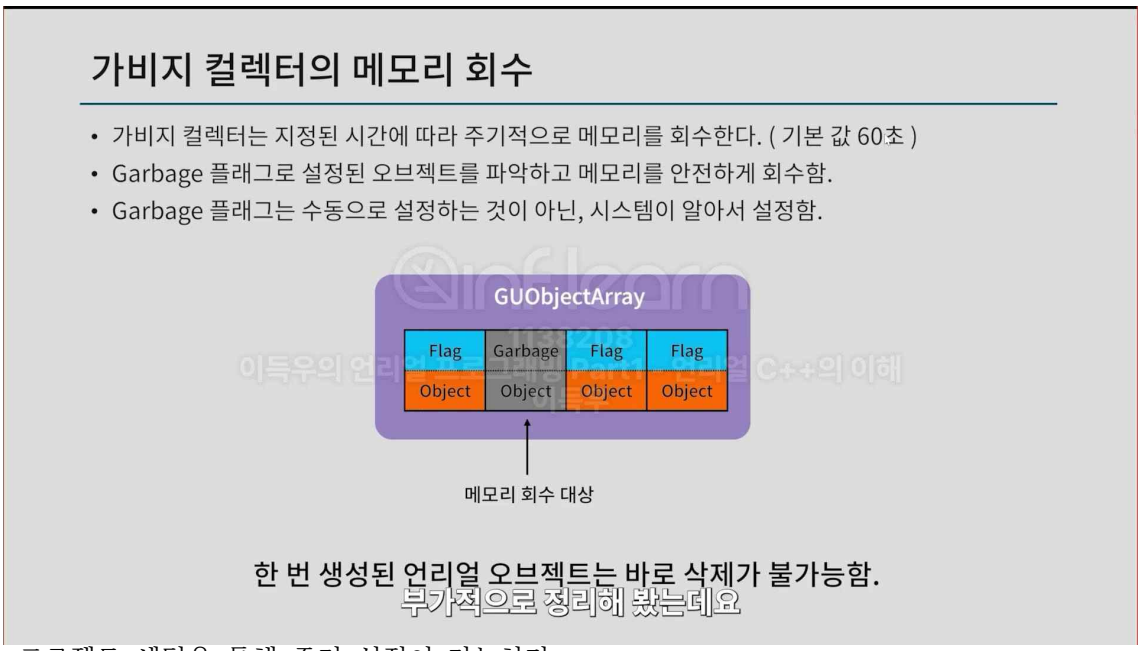


가비지 컬렉터는 GUObjectArray에 있는 플래그를 확인해
빠르게 회수해야 할 오브젝트를 파악하고 메모리에서 제거함

언리얼 엔진에서는 관리되는 모든 언리얼 오브젝트의

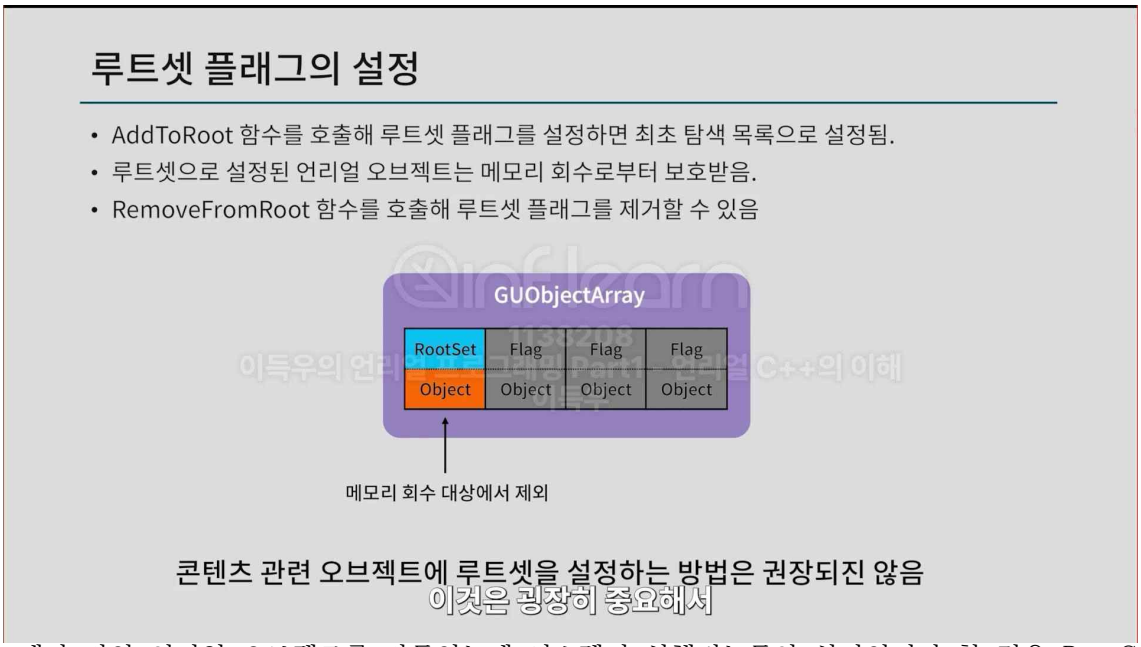
-언리얼 엔진이 활성화된 순간에 누구나 GUObjectArray에 접근할 수 있으며 GUObjectArray의 각 요소에는 플래그라는 정보가 설정되어 있고, 플래그엔 다양한 값 설정이 가능하다.(사진에 Garbage플래그와 RootSet플래그에 대해서 설명한다.)
-RootSet플래그가 있는 경우 다른 언리얼 오브젝트로부터 참조가 없어도 가비지 컬렉터는 회수 하지 않는다.

**가비지 컬렉터의 메모리 회수



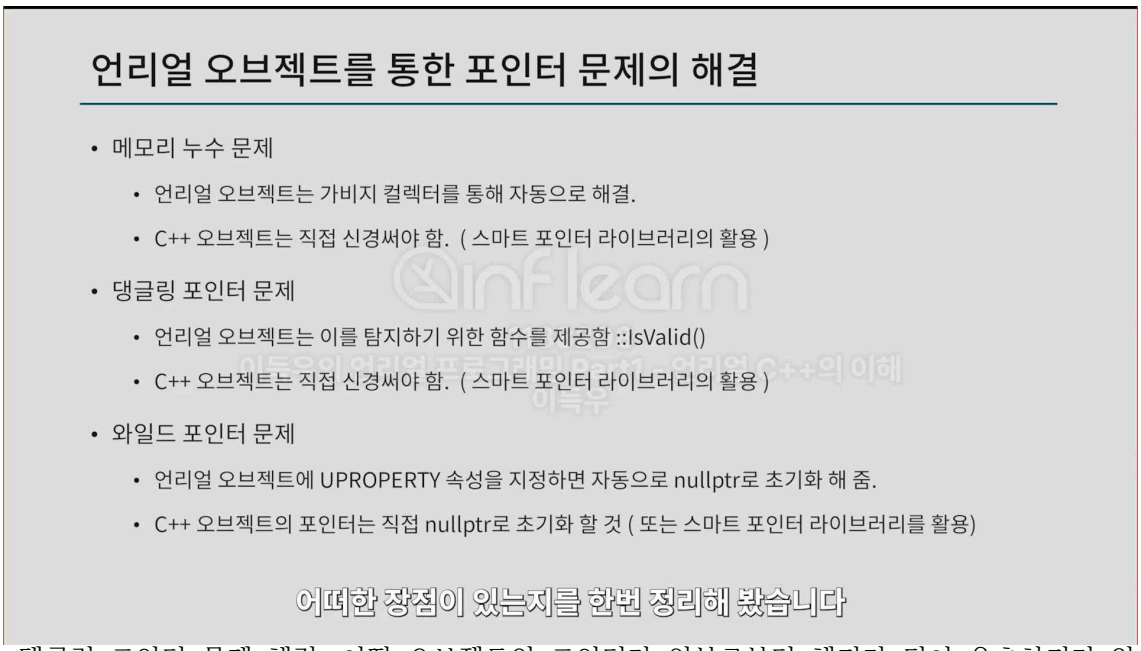
-프로젝트 세팅을 통해 주기 설정이 가능하다.
-가비지 플래그는 시스템이 알아서 설정한다.(우리가 GUObjectArray에 직접 접근하여 플래그 설정할 필요가 없다.)
-한번 생성된 오브젝트를 삭제하기 위해서는 delete 키워드를 사용하여 바로 삭제하는 것이 아니라 레퍼런스 정보를 없앰으로써 언리얼의 가비지 컬렉터가 자동으로 메모리를 회수하도록 설정하는 것이다.

**루트셋 플래그의 설정(컨텐츠를 만드는 도중에 사용하는 것은 추천하지 않음)



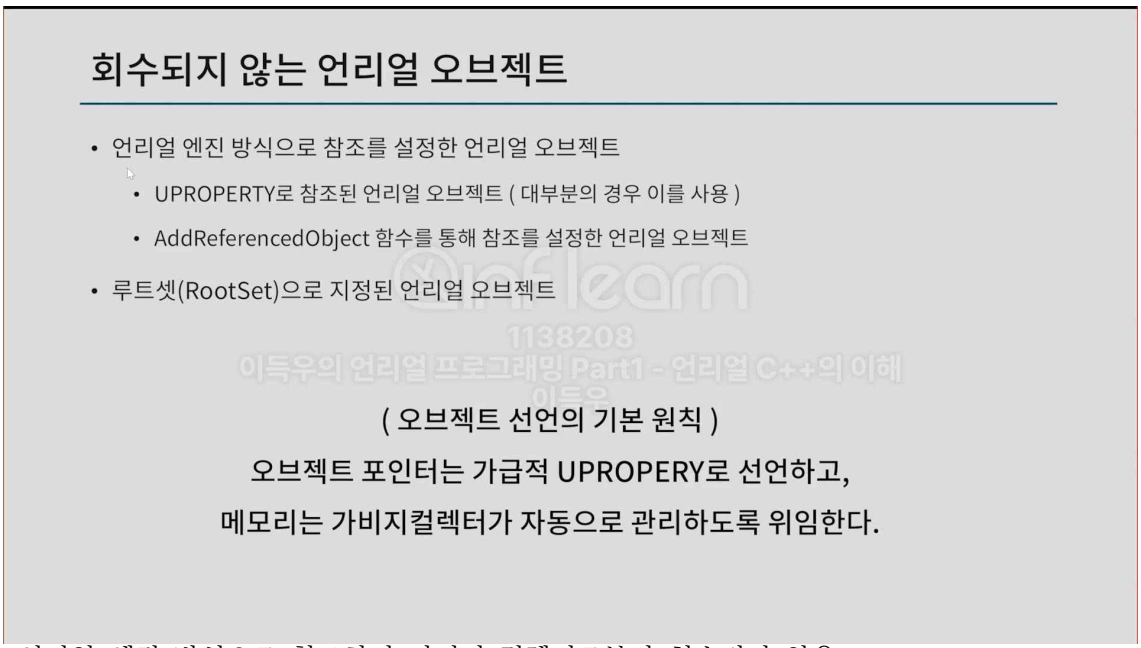
-내가 만약 언리얼 오브젝트를 만들었는데 시스템이 실행되는동안 살아있어야 할 경우 RootSet으로 등록하면 편해진다.(AddToRoot, RemoveFromRoot 함수를 사용하도록 하자.)

**언리얼 오브젝트를 통한 포인터 문제의 해결



-댕글링 포인터 문제 해결: 어떤 오브젝트의 포인터가 외부로부터 해지가 되어 유효하지가 않은 지를 파악할 수 있어야 하는데 언리얼 엔진은 이를 탐지하는 함수를 가지고 있다.

**회수되지 않는 언리얼 오브젝트

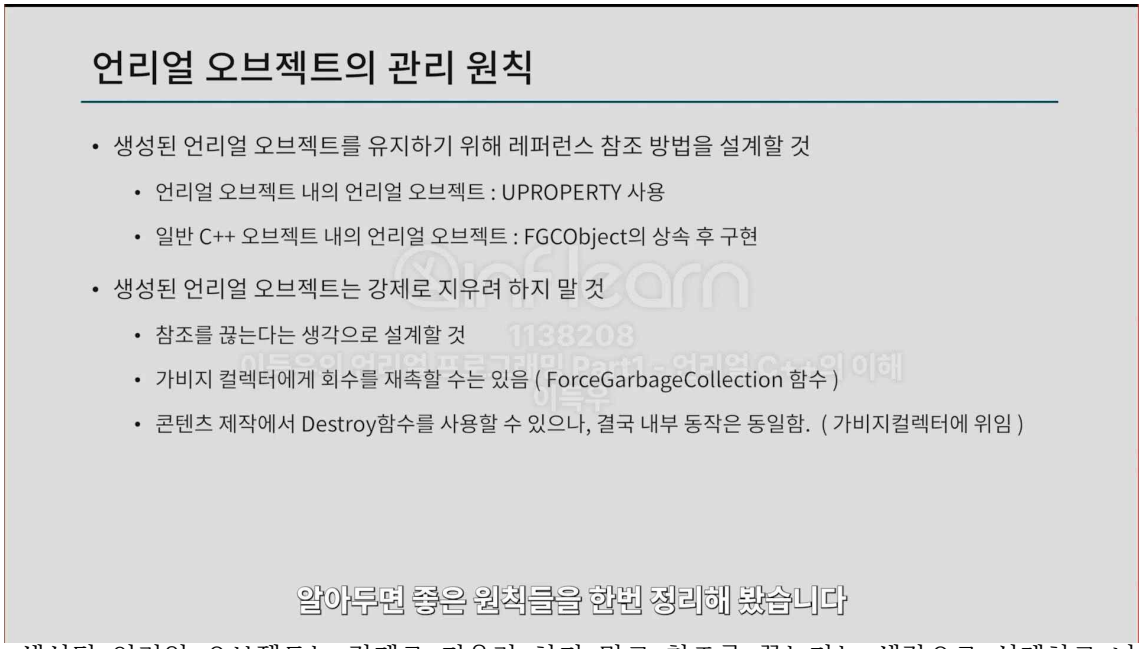


-언리얼 엔진 방식으로 참조하면 가비지 컬렉터로부터 회수되지 않음.
-루트셋으로 지정하는건 그 오브젝트가 굉장히 중요하다라고 지정해주는거라 많이 사용하지는 않는다.

**일반 클래스에서 언리얼 오브젝트를 관리하는 경우

-어떤 C++ 객체 내에 언리얼 오브젝트가 멤버변수로 들어가게 될 경우 FGC라는 오브젝트를 상속받은 다음에 AddReferencedObjects함수를 구현해야 한다.(이 부분은 콘텐츠 제작에서 자주 사용하는 방법은 아니다. [실습에서는 다뤄볼꺼임])

**언리얼 오브젝트의 관리 원칙

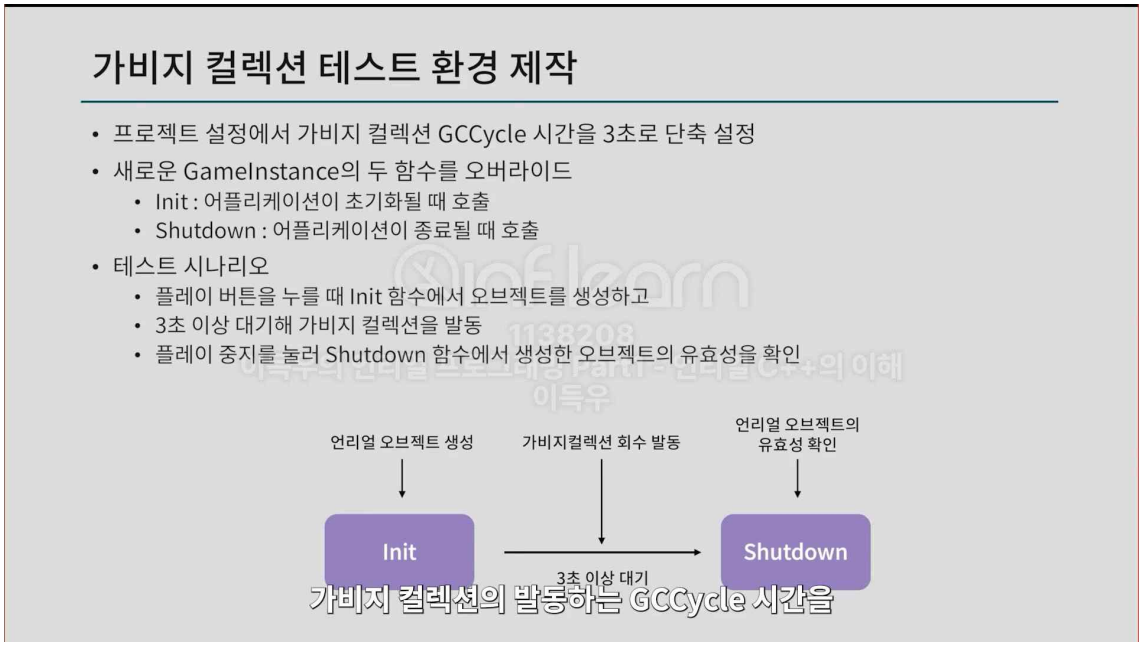


-생성된 언리얼 오브젝트는 강제로 지울려 하지 말고 참조를 끊는다는 생각으로 설계하고 나머지는 가비지 컬렉터에게 맡기면 된다.(60초가 길다면 회수를 재촉할수 있다.[ForceGarbage Collection 함수]) 또한 '콘텐츠 제작'에도 동일하게 사용이 가능한데 액터(Actor)라 불리는 오브젝트를 소멸하기 위해서 Destroy라는 함수를 제공한다. Destroy함수는 바로 오브젝트를 제거하는 것이 아닌 플래그를 설정해주고 가비지 컬렉터가 자동으로 회수하도록 설정해주는 것에 불과하다.

-본격적인 콘텐츠 제작하기 이전 언링러 엔진이 어떤식으로 메모리를 관리하는지 이해하는 것이 중요하다.

***실습

**가비지 컬렉션 테스트 환경 제작



**코드

-일반적으로 콘텐츠 제작에서는 IsValid라는 함수를 많이 사용해서 유효성 판단을 하지만 이번 예제에서는 내부적으로 깊숙이 들어갈거라 정교히 체크 가능한 함수를 사용할것임.

==>IsValidLowLevel()

-첫 번째 실행

```
LogTemp: [NonPropStudent] 널 포인터가 아닌 언리얼 오브젝트
LogTemp: [NonPropStudent] 유효하지 않은 언리얼 오브젝트
LogTemp: [PropStudent] 널 포인터가 아닌 언리얼 오브젝트
LogTemp: [PropStudent] 유효한 언리얼 오브젝트
```

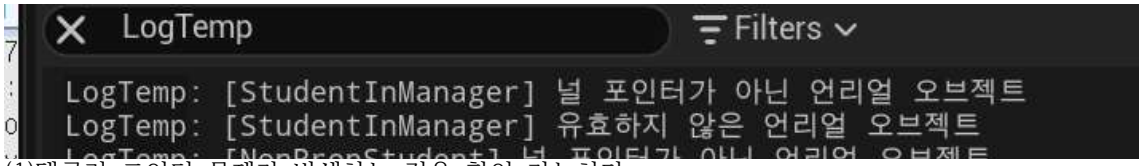
(1)NonPropStudent의 경우 우리가 널포인터인지 아닌지만 보고 유효하다, 유효하지 않다라고 진행하게 되면 댕글링 포인터 문제가 발생할 수 있게 된다.

(2)따라서 언리얼 오브젝트의 선언에서 언리얼 오브젝트의 클래스 멤버 변수를 선언할 때는 반드시 UPROPERTY를 붙여줘야지만 댕글링 포인터 문제에서 벗어날 수가 있다.

-자료구조 컨테이너 안에 언리얼 오브젝트 안전하게 관리해보기

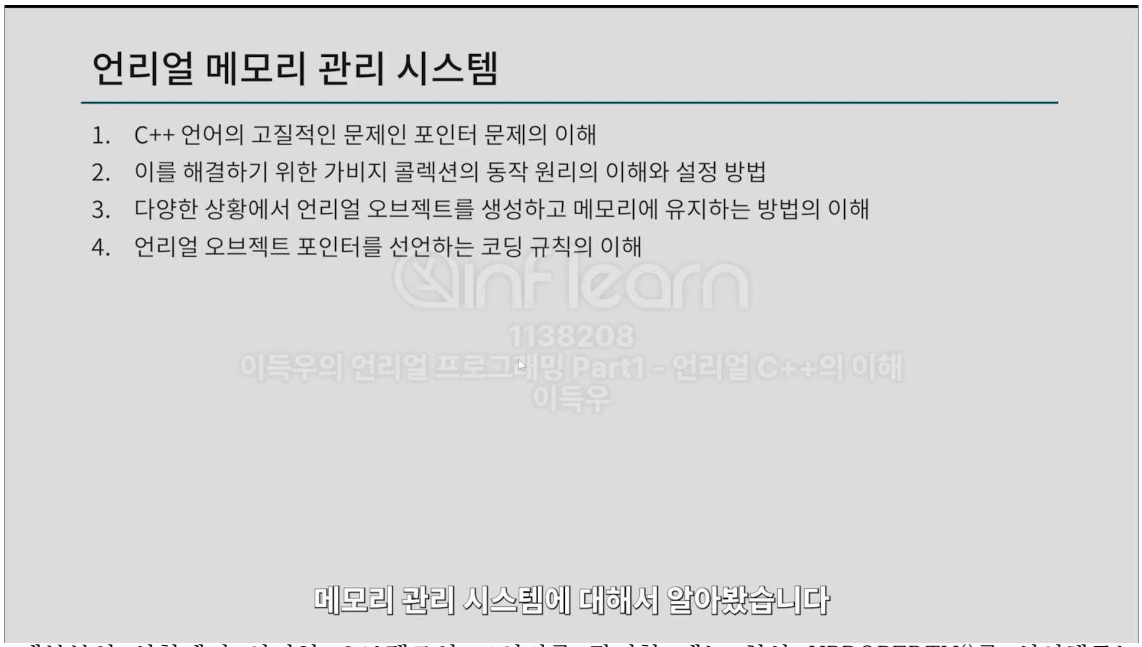
(1)TArray나 TSet이나 TMap같은 자료구조의 템플릿 인자, 즉 타입 인자로 언리얼 오브젝트 포인터가 들어가는 경우에는 반드시 UPROPERTY 매크로를 붙여줘야 안전하게 언리얼 오브젝트를 관리할 수 있게 된다.

-일반 c++ 오브젝트



(1)댕글리 포인터 문제가 발생하는 것을 확인 가능하다.
(2)이것을 해결할려면 일반 c++ 객체가 “나는 언리얼 오브젝트를 관리하겠다” 라고 알려줘야 한다.
(3)FGCObject를 상속 받으면 (2) 이 해결되는데 두가지 함수를 구현해줘야 한다.
(4)두가지 함수:AddReferencedObjects, GetReferencerName
(5) 변경후 사진

**정리



-대부분의 상황에서 언리얼 오브젝트의 포인터를 관리할 때는 항상 UPROPERTY()를 선언해주는 것으로 이해하자.