UE4 C++

# Assert 매크로

주어진 코드 조각이 가정하는 상황을 검증하는 도구

런타임 Assert 매크로는 실행 중지, 디버그 빌드에서 실행 중지, 실행을 중지하지 않고 오류 보고 3가지 카테고리에 들어간다.  
- 실행 중지, 실행을 중지하지 않고 오류 보고는 DO\_CHECK define에 따라서 컴파일  
- 디버그 빌드에서 실행 중지 유형은 DO\_GUARD\_SLOW define을 사용하여 컴파일 된다.  
- 해당 define 중 하나가 0으로 설정되면 매크로는 비활성화, 실행에 영향을 미치지 않는다.

check(표현식)

표현식을 실행한 후 Assert 결과가 false이면 실행을 중지시킨다  
- 표현식은 매크로가 빌드에 컴파일 되는 경우에만 실행된다  
- DO\_CHECK = 1  
- 가장 간단한 형태의 매크로

|  |
| --- |
| check(GEngine); // 엔진 초기화를 확인  check(GetWorld() != nullptr); // 월드가 제대로 로드되었는지 확인 |

verify(표현식)

DO\_CHECK가 켜져 있으면 Check 매크로와 동일한 역할을 한다  
- DO\_CHECK가 꺼져 있어도 실행된다  
- 변수 할당이 가정한 대로 되어 있는지 등을 검증하는데 사용할 수 있다.

|  |
| --- |
| verify((World = GetWorld()) != nullptr); |

checkf(표현식, 표현식, …)

표현식이 true가 아니면 디버깅에 도움이 되는 추가 정보를 출력하는 것이 가능하다  
- 컴파일 측면에서 check 매크로와 동일

|  |
| --- |
| checkf(bActorDestroyed, TEXT("Failed to destroy Actor %s", \*Actor->GetClass()->GetName()); |

verifyf(표현식, … )

verify 매크로와 동일하게 항상 실행된다  
- checkf 매크로처럼 실행을 중지시키면서 디버깅에 도움이 되는 추가적인 디버그 메시지를 남길 수 있다.

Assert 매크로 정보

<https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/Assertions/>

<https://jhtop0419.tistory.com/24?category=944635>

# 비동기 애셋 로딩

실행 시간에 애셋을 로드/언로드 하는 메소드를 의미한다.

개발 도중, 디바이스 상에서 쿠킹된 데이터로 실행할 때나 동일하게 작동하므로 필요에 따라 데이터를 로딩하기 위한 별도의 코드 패스를 유지할 필요가 없다

필요에 따라 데이터를 로드 및 참조하는 것에 사용되는 방법은 일반적으로 2가지 존재한다

FSoftObjectPath와 TSoftObjectPtr

FSofObjectPath는 애셋의 전체 이름으로 된 스트링이 들어있는 단순한 구조체로 클래스에 이 유형의 프로퍼티를 만들면 에디터에 UObejct\* 프로퍼티처럼 나타난다  
- 쿠킹과 리디렉터도 제대로 처리되므로 FSoftObjectPath가 있다면 디바이스에서의 정상 작동도 보장된다

FSoftObjectPtr은 기본적으로 FSoftObjectPath를 감싸는 TWeakObjectPtr로 참조된 애셋이 메모리에 존재한다면 TSoftObjectPtr.Get()은 애셋을 반환한다

FSoftObjectPath와 TSoftObjectPtr은 아티스트, 디자이너가 래퍼런스를 수동으로 셋업하는 경우에 좋다

애셋 레지스트리와 오브젝트 라이브러리

애셋 레지스트리는 애셋에 대한 메타데이터를 저장하여 해당 애셋에 대한 검색 및 질의를 가능하게 해주는 시스템  
- 애셋에 대한 데이터를 검색 가능하게 만들려면, 프로퍼티에 "AssetRegistrySearchable" 태그를 추가해야 한다  
- 애셋 레지스트리에 대한 질의는 FAssetData 유형의 오브젝트를 반환하는데 여기에는 오브젝트에 대한 정보, 검색 가능한 것으로 마킹된 프로퍼티가 들어있는 (키->값) 쌍의 Map도 포함된다.

오브젝트 라이브러리는 로드되지 않은 애셋 그룹을 가지고 작업하는 가장 쉬운 방법이다  
- ObjectLibrary는 로드된 오브젝트와 로드되지 않은 오브젝트의 경우 FAssetData를 합친 목록이 들어있는 오브젝트로, 공유 베이스 클래스를 상속한다  
- ObjectLibrary에 검색할 경로를 주는 것으로 로드하면 그 경로에 존재하는 모든 애셋이 추가된다  
- 컨텐츠 폴더 일부분을 각기 다른 유형으로 지정, 아티스트와 디자이너가 마스터 목록을 수동 편집할 필요 없이 새 애셋을 추가하는 것이 가능함

ObjectLibrary를 사용하여 AssetData를 디스크에서 로드하는 방법

|  |
| --- |
| if(!ObjectLibrary)  {  ObjectLibrary = UObjectLibrary::CreateLibrary(베이스 클래스 명, false, GlsEditor);  ObjectLibrary->AddToRoot();  }  ObjectLibrary->LoadAssetDataFromData(TEXT("/Game/PathWithAllObjectsOfSameType");  if(bFullyLoad)  {  ObjectLibrary->LoadAssetsFromAssetData();  } |

- 새로운 오브젝트 라이브러리를 생성하고, 베이스 클래스를 할당하면 주어진 경로의 모든 애셋 데이터를 로드한다  
- 옵션을 통해 실제 애셋을 로드하는 것도 가능  
- 애셋이 작은 경우 애셋을 전체 로드할 수도 있고 쿠킹 중인 경우 모두 쿠킹되도록 할 수 도 있다.  
- 쿠킹 도중 애셋 레지스트리 질의를 하고서 반환된 애셋을 로드하는 한 ObjectLibrary는 개발 중인 데이터든 디바이스에서 쿠킹된 데이터든 똑같이 작동한다.

ObjectLibrary에 애셋 데이터가 들어있다면 질의를 통해서 특정 애셋만을 선택적으로 로드하는 것도 가능하다

|  |
| --- |
| TArray<FAssetData> AssetDatas;  ObjectLibrary->GetAssetDataList(AssetDatas);  for(int32 i = 0; i < AssetDatas.Num(); i++)  {  FAssetData& AssetData = AssetDatas[i];  const FString\* FoundTypeNameString = AssetData.TagsAndValues.Find(GET\_MEMBER\_NAME\_CHECKED(UAssetObject, 타입));  if(FoundTypeNameString && FoundTypeNameString->Contains(Text("Type)))  {  return AssetData;  }  } |

- ObjectLibrary에서 타입으로 지정한 type이 들어있는 것들을 검색하여 처음 찾은 것을 반환하는 코드

StreamableManager와 비동기 로딩

ObjectLibrary에서 반환된 AssetData를 가지고 ToStringReference()를 호출하여 FSoftObjectPath로 변환하고 나면 StreamableManager를 가지고 비동기 로드를 하는 것이 가능하다

먼저 FStreamableManager를 일종의 싱글톤 오브젝트로 생성하는 것이 좋다  
- DefaultEngine.ini에서 GameSingletonClassName에 지정된 오브젝트에 넣는 것이 좋다  
- FSoftObjectPath를 전달한 다음 로드를 시작한다  
- SynchronousLoad를 사용하면 단순한 로드 블록 후 오브젝트를 반환, 이럴 경우 오브젝트의 크기가 클 경우 상당히 비효율적이다.  
- RequestAsyncLoad를 사용하여 애셋 그룹을 비동기 로드한 다음 완료되면 델리게이트를 호출한다.

|  |
| --- |
| void UGameCheatManager::GrantItems()  {  TArray<FSoftObjectPath> ItemsToStream;  FStreamableManager& Streamable = UGameGlobals::Get().StreamableManager;  for(int32 i = 0; i < ItemList.Num(); ++i)  {  ItemsToStream.AddUnique(ItemList[i].ToStringReference());  }  Streamable.RequestAsyncLoad(ItemsToStream, FStreamableDelegate::CreateUObject(this, &UGameCheatManager::GrantItemsDeferred));  }  void UGameCheatManager::GrantItemsDeferred()  {  for(int32 i = 0; i < ItemList.Num(); ++i)  {  UGameItemData\* ItemData = ItemList[i].Get();  if(ItemData)  {  MyPC->GrantItem(ItemData);  }  }  } |

# 언리얼 오브젝트

언리얼 엔진에서 게임 오브젝트를 처리하는 견고한 시스템  
- 언리얼 엔진 오브젝트의 베이스 클래스인 UObject 클래스  
- 파생되는 클래스에 UCLASS 매크로를 사용하여 태그하면 UObject를 처리하는 시스템에서 인식

## 1: UCLASS 매크로

UObject에게 자신의 언리얼에서 기반으로 삼은 유형에 대해 설명해주는 UClass로의 레퍼런스를 넘겨준다  
- 각각의 UClass는 CDO(Class Default Object)라 불리는 오브젝트를 하나 유지한다  
- CDO는 본질적으로 기본 템플릿 오브젝트로 클래스 생성자에 의해 생성된 후 변경되지 않는다  
- 오브젝트 인스턴스에 대해서 UClass와 CDO 둘 다 접근 가능하지만 일반적으로 읽기 전용으로 간주되어야 한다  
- 오브젝트 인스턴스에 대한 UClass는 GetClass() 함수를 사용하여 언제든 접근 가능

## 2: 프로퍼티와 함수 유형

UObject는 멤버 변수와 어떤 유형의 함수도 가질 수 있다. 하지만 언리얼 엔진에서 이러한 변수나 함수를 인식하고 조작할 수 있도록 하기 위해서는 특수한 매크로로 마킹하여 특정 유형 표준에 맞춰야 한다.

### 프로퍼티 선언

|  |
| --- |
| UPROPERTY([specifier, specifier, …], [meta = (key = value, …)])  Type VariableName; |

- 프로퍼티는 URPOPERTY 매크로를 사용하여 프로퍼티 메타 데이터와 변수 지정자를 붙여서 선언한다

#### 프로퍼티 코어 데이터 유형

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 정수형 | 정수 데이터 유형의 규칙은 int, uint 뒤에 크기를 비트 단위로 붙이는 것   |  |  | | --- | --- | | uint8 | 부호 없는 8비트 정수형 | | uint16 | 부호 없는 16비트 정수형 | | uint32 | 부호 없는 32비트 정수형 | | uint64 | 부호 없는 64비트 정수형 | | int8 | 부호 있는 8비트 정수형 | | int16 | 부호 있는 16비트 정수형 | | int32 | 부호 있는 32비트 정수형 | | int64 | 부호 있는 64비트 정수형 | |
| 부동소수점 형 | 언리얼 엔진에서는 표준 C++ 부동 소수점 유형인 float와 double을 사용 |
| 부울형 | C++ bool 키워드 또는 비트 필드로 나타내는 것이 가능   |  | | --- | | uint32 bIsHungry : 1;  bool bIsThirsty; | |
| 스트링 | 언리얼 엔진에서는 3가지 유형의 스트링이 지원된다   |  |  | | --- | --- | | FString | 고정적인 Char 타입 동적 배열 스트링 유형 | | FName | 글로벌 스트링 테이블로 된 변경이 불가능한, 대소문자 구분이 없는 스트링에 대한 레퍼런스  - FString보다 작아 전송에 효율적  - FString보다 조작하기 더 어렵다 | | FText | 현지화 처리를 위해 고안된 보다 단단한 스트링 표현 |   대부분의 경우 언리얼은 문자에 대해 TCHAR 유형에 의존한다 - TEXT() 매크로는 TCHAR 상수를 나타내는 데 사용할 수 있다.   |  | | --- | | MyDogPtr->DogName = FName(TEXT("Samson Aloysius")); | |

#### 프로퍼티 지정자

프로퍼티 선언 시 프로퍼티 지정자를 붙여 프로퍼티가 엔진과 에디터의 다양한 부분과 어떻게 작동하는지를 제어할 수 있다.

|  |  |
| --- | --- |
| 프로퍼티 지정자 | 기능 |
| AdvancedDisplay | 디테일 패널의 고급 섹션에 들어간다 |
| AssetRegistrySearchable | 애셋 레지스트리 검색 가능 지정자  - 프로퍼티와 그 값이 이것을 멤버 변수로 포함하는 클래스 인스턴스에 대한 애셋 레지스트리에 자동으로 추가된다  - 구조체 프로퍼티나 파라미터에 사용하는 것은 적법하지 않다. |
| BlueprintAssignable | 멀티캐스트 델리게이트 전용  프로퍼티를 블루프린트에서 할당할 수 있도록 노출시킨다 |
| BlueprintAuthorityOnly | 프로퍼티가 멀티캐스트 델리게이트여야 사용 가능  - BlueprintAuthorityOnly 지정자가 붙은 이벤트만 받는다 |
| BlueprintCallable | 멀티캐스트 델리게이트 전용  - 프로퍼티를 노출시켜야 블루프린트 코드에서 호출할 수 있다. |
| BlueprintGetter  = GetterFunctionName | 커스텀 액세서 함수를 지정한다  - BlueprintSetter 또는 BlueprintReadWrite지정자가 지정되지 않으면 BlueprintReadOnly로 간주된다. |
| BlueprintReadOnly | 블루프린트에서 읽기는 가능하지만 쓰기는 불가능하다  - BlueprintReadWrite 지정자와 호환되지 않는다 |
| BlueprintReadWrite | 블루프린트에서 읽고 쓰기가 가능  - BlueprintReadOnly 지정자와 호환되지 않는다 |
| BlueprintSetter  = SetterFunctionName | 묵시적으로 BlueprintReadWrite 지정자가 붙으며 커스텀 뮤테이터 함수가 있다.  - 뮤테이터 함수는 "같은 클래스명 으로 그 일부를 따서 지어야 한다.  \* 뮤테이터 = Setter 또는 수정자 |
| Config | 환경 설정이 가능해진다  - 현재 값은 클래스와 연결된 .ini 파일에 저장  - 생성 시 로드된다  - 디폴트 프로퍼티에 값을 줄 수 없다  - BlueprintReadOnly로 간주된다 |
| DuplicateTransient | 프로퍼티를 어떤 식으로든 복제할 때 클래스 기본 값으로 리셋시킬 지 나타낸다 |
| Export | 오브젝트 프로퍼티 전용  - 이 프로퍼티에 할당된 오브젝트는 오브젝트를 복사, T3D로 익스포트 할 때 오브젝트 레퍼런스 자체만 출력하는 것이 아니라 서브 오브젝트 블록 전체로 익스포트 해야함을 의미 |
| GlobalConfig | Config와 비슷하지만 서브 클래스에서 덮어쓸 수 없다  - 디폴트 프로퍼티에 값을 줄 수 없다  - BlueprintReadOnly로 간주된다 |
| Instanced | 오브젝트(UCLASS) 프로퍼티 전용  - 이 클래스의 인스턴스가 생성될 때 이 프로퍼티에 기본으로 할당된 오브젝트 고유 사본을 보내준다.  - 클래스 디폴트 프로퍼티에 정의된 서브 오브젝트 인스턴싱에 사용된다.  - EditIiline 및 Export로 간주됨 |
| Interp | 마티네의 트랙으로 시간에 따라 구동시킬 수 있다는 것을 나타낸다. |
| Localized | 정의된 현지화 값이 있음  - 거의 스트링 처리에 사용된다.  - ReadOnly로 간주됨 |
| Native | 프로퍼티가 네이티브임을 나타냄  - C++ 코드가 직렬화, 가비지 컬렉션으로의 노출을 담당한다. |
| NoClear | 에디어테이서 이 오브젝트 레퍼런스를 없음으로 설정하지 못하게 막는다  - 에디터에서 clear, browse 버튼을 숨겨버린다. |
| NoExport | 네이티브 클래스 전용  - 프로퍼티를 자동 생성 클래스 선언에 포함시키면 안된다. |
| NonPIEDuplicateTransient | PIE 세션 도중이 아닌 경우 복제 도중 기본 값으로 리셋된다. |
| NonTransactional | 프로퍼티에 대한 변경은 에디터의 되돌리기, 다시하기 히스토리에 포함되지 않는다 |
| NotReplicated | 리플리케이션을 생략  - 서비스 요청 함수의 구조체 멤버와 파라미터에만 적용됨 |
| Replicated | 네트워크를 통해 리플리케이트된다. |
| ReplicatedUsing  = FunctionName | 네트워크를 통해 프로퍼티를 업데이트 할 때 실행되는 콜백 함수를 지정한다 |
| RepRetry | 구조체 프로퍼티 전용  - 프로퍼티를 완전히 전송하는 데 실패한 경우 그 리플리케이션을 재시도한다.  - 단순 레퍼런스의 경우 디폴트 값  - 구조체의 경우 대역폭 비용으로 인해 바람직하지 않아 기본적으로 비활성화 되어 있음 |
| SaveGame | 프로퍼티 레벨에서 체크포인트/저장 시스템 용 필드를 명시적으로 포함시키기 위한 단순한 방법  - 옵션은 저장된 게임의 일부로 포함시키려는 모든 필드에 설정해 두면, 프록시 아카이버를 사용하여 읽고 쓰기가 가능하다 |
| SerializeText | 네이티브 프로퍼티를 텍스트로 직렬화 |
| SkipSerialization | 직렬화되지 않지만 텍스트 포맷으로 익스포트가 가능 |
| SimpleDisplay | Visble 또는 Editable 프로퍼티는 디테일 패널에 표시  - Advanced 섹션을 열지 않아도 보인다. |
| TextExportTransient | 프로퍼티가 텍스트 포맷으로 익스포트 되지 못하게 한다 |
| Transient | 프로퍼티를 휘발성으로 만든다  - 로드 시간에 0으로 채워진다. |

### 함수 유형 선언

|  |
| --- |
| UFUNCTION([specifier, specifier, …], [meta = (key=value, …)])  ReturnType FunctionName([Parameter, Parameter, …]) |

- 프로퍼티 선언과 비슷한 방식으로, 함수 유형은 UFUNCTION 매크로를 사용하여 선언한다.

## 3: 오브젝트 생성

|  |  |
| --- | --- |
| NewObject<class>() | 자동 생성되는 이름으로 새 인스턴스를 만든다  - 단순 클래스를 생성할 때 자주 사용 |
| new | 생성자가 인수를 필요로 할 때와 같이 특정 로우 레벨 환경에서 오브젝트를 생성할 때 사용한다. |

- 게임 플레이용 UObject 클래스 인스턴스 생성을 위한 함수들

### NewObject

UObjectGlobals.h 파일에 다양한 형태로 오버로딩 되어 있다.

|  |  |
| --- | --- |
| template<class T>  T\* NewObject  (  UObject\* Outer=(UObject\*)GetTransientPackage(),  UClass\* Class=T::StaticClass()  ) | template<class T>  T\* NewObject  (  UObject\* Outer,  FName Name,  EObjectFlags Flags = RF\_NoFlags,  UObject const\* Template=NULL  ) |
| 기본형 | 일반적으로 많이 사용되는 함수 구현부 |

일반적으로 많이 사용되는 함수 구현부에서는 새 인스턴스 이름, 오브젝트 플래그, 템플릿 오브젝트를 인수로 지정할 수도 있다.  
- 파라미터에서 Outer와 Name은 필수로 넘겨줘야 하며 플래그와 템플릿 오브젝트에 대한 정보는 선택 사항

|  |  |
| --- | --- |
| Outer | 생성될 오브젝트를 소유하는 UObject객체 |
| Name | 새 오브젝트에 설정할 이름 |
| Flags | 새 오브젝트를 설명하는 FObjectFlagsEnum 값 |
| Template | 새 오브젝트 생성 시 템플릿으로 사용할 UObject |

### 플래그

오브젝트 플래그는 FObjectFlagsEnum 값을 사용하여 표현한다

|  |  |  |
| --- | --- | --- |
| 플래그 유형 | 플래그 이름 | 설명 |
| 오브젝트 유형 | RF\_Public | 이 오브젝트는 포함된 패키지 밖에서도 보인다 |
|  | RF\_Standalone | 이 오브젝트는 레퍼런스 되지 않아도 편집 가능한 상태로 유지된다. |
| 가비지 컬렉션 | RF\_MarkAsRootSet | 이 오브젝트는 레퍼런스 되지 않아도 가비지 콜렉팅이 되지 않는다 |
|  | RF\_TagGarbageTemp | 이 오브젝트는 가비지 컬렉션을 사용하는 다양한 유틸리티가 사용할 수 있도록 마킹된다.  - 이 플래그는 가비지 콜렉터 자체적으로는 해석하지 않는다 |
| 오브젝트 수명 | RF\_AsyncLoading | 이 오브젝트는 비동기 로딩 중이다 |
|  | RF\_NeedPostLoad | 이 오브젝트는 사후 로딩이 필요하다 |

### 사용 예시

|  |
| --- |
| UItemData\* ItemData = NewObject<UItemData>(this, TEXT("ItemData")); |
| UItemData\* ItemData = NewObject<UItemData>(this, TEXT("ItemData"), RF\_MarkAsRootSet); |

Outer와 새 오브젝트에 지정할 이름만을 넘겨줘 생성도 가능하며 플래그도 함께 넘겨서 생성해줄 수도 있다.  
- RF\_MarkAsRootSet 플래그를 사용할 경우 해당 오브젝트가 가비지 콜렉팅이 되지 않응므로 사용이 끝나면 수동으로 메모리 해제가 필요함

## 4: UObject 클래스 상속의 장점

생성한 클래스를 UObject 처리 시스템이 인식할 수 있도록 만들어 주는 것  
- 모든 경우에 이 시스템을 사용하는 것이 필수적이지 않음  
- 오히려 적절하지 않은 경우도 존재함

장점  
- 가비지 컬렉션  
- 레퍼런스 업데이트  
- 직렬화  
- 리플렉션  
- 디폴트 프로퍼티 변경사항 자동 업데이트  
- 자동 프로퍼티 초기화  
- 자동 에디터 통합  
- 런타임에 유형 정보 사용가능  
- 네트워크 리플리케이션

# 스트링 처리

언리얼 엔진에서 사용 가능한 스트링 클래스는 FName, FText, FString으로 3가지가 존재한다

## FName

콘텐츠 브라우저에서 새 애셋 이름을 지을 때, 다이나믹 머테리얼 인스턴스의 파라미터를 변경할 때, 스켈레탈 메시에서 본에 접근할 때, 모두 FName을 사용한다.

FName은 문자열 사용에 있어서 초경량 시스템을 제공한다  
- 주어진 문자열이 사용되도 데이터 테이블에 한 번만 저장된다.

FName은 대소문자를 구분하지 않고 변경, 조작도 불가능하다

FName의 정적인 속성과 저장 시스템 덕에 키 값으로 FName에 접근하는 속도가 굉장히 빠르다.

스트링에서 FName 변환이 해시 테이블을 사용하여 빠르다.

|  |
| --- |
| FName TestName = FName(TEXT("MyTestFName")); |
| FName 생성 |

- FName 클래스로의 타입 변환만으로 FName 타입의 객체 생성 가능

|  |
| --- |
| FName TestName = FName(TEXT("MyTestFName"));  FName OtherName = FName(TEXT("MyTestFName"));  FName AnotherName = FName(TEXT("AnotherTestFName"));  bool bCompare1 = (TestName == OtherName); // 결과는 true  bool bCompare2 = (TestName == AnotherName); // 결과는 false |
| == 연산자를 사용해서 두 FName 객체가 같은지 다른지 비교 가능 |
| FName TestName1 = FName(TEXT("A"));  FName TestName2 = FName(TEXT("B"));  FName TestName3 = FName(TEXT("B"));  FName TestName4 = FName(TEXT("C"));  float CompareFloat1 = TestName2.Compare(TestName1); // 양수 값  float CompareFloat2 = TestName2.Compare(TestName3); // 0  float CompareFloat3 = TestName2.Compare(TestName4); // 음수 값 |
| FName::Compare 함수를 사용하여 객체 비교도 가능  - 현재 객체가 파라미터로 넘기는 객체의 값보다 크면 양수, 같으면 0, 작으면 음수를 반환한다 |

## FText

언리얼 엔진에서 텍스트 지역화의 주요 구성요소 클래스

- 지역화 된 텍스트 상수 생성  
- 텍스트 서식(자리 표시자 패턴에서 텍스트)를 생성  
- 숫자에서 텍스트를 생성  
- 날짜와 시간에서 텍스트를 생성  
- 텍스트를 상위 또는 소문자로 만드는 것과 같은 파생된 텍스트를 생성

모든 사용자 대면 텍스트는 위의 기능을 제공하여 텍스트 지역화를 지원하여 FText 클래스를 사용해야 한다.

|  |
| --- |
| FText text = FText::Format(LOCTEXT("ExampleFText", "You currently have {0} health left."), CurrentHealth); |
| FText 생성 |

- FText::Format 함수를 사용하여 생성할 수 있다.  
- 변수를 추가할 수 있는데 {0}, {1}, {2}, … 로 위치를 표시하고 이후 변수들을 뒤쪽에 추가하면 된다.

FText 클래스는 단순한 문자열보다 복잡하므로 오버로드된 연산자 비교가 지원되지 않는다  
- 대신 포함된 미묘한 데이터를 인식하는 비교를 수행하는 여러 기능들이 제공된다.

|  |  |
| --- | --- |
| 함수 | 기능 |
| EqualTo | 두 FText 객체가 같은지 다른지 비교한다  - 같으면 true, 다르면 false |
| EqualToCaseIgnored | 이 함수는 대소문자를 구부하지 않고 두 FText 객체가 같은지 다른지 비교한다  - 같으면 true, 다르면 false |
| CompareTo | FText 객체가 다른 FText 객체보다 크다면 양수, 같으면 0, 작으면 음수 값을 반환한다. |
| CompareToCaseIgnored | 대소문자를 구분하지 않고 FText 객체가 다른 FText 객체보다 크다면 양수, 같으면 0, 작다면 음수를 반환한다. |

예시

|  |
| --- |
| bool bCompare1 = TestName.EqualTo(OtherName); // 대소문자 구분  bool bCompare2 = TestName.EqualToCaseIgnored(OtherName); // 대소문자 미구분  int32 iCompare1 = TestName.CompareTo(OtherName); // 대소문자 구분  int32 iCompare2 = TestName.CompareToCaseIgnored(OtherName); // 대소문자 미구분 |

## FString

FName이나 FText와는 달리 조작이 가능한 유일한 스트링 클래스  
- 대소문자 변환, 부분 문자열 추출, 스트링 역순 등의 함수가 지원됨  
- 검색, 변경 및 다른 스트링과의 비교 또한 가능

|  |
| --- |
| FString TestString = FString(TEXT("TestString")); |

- 간단한 방식으로 생성이 가능하다(FName과 같은 방식)

### FString 클래스 비교

두 FString 객체를 비교하는 방법은 여러가지 존재함  
- 오버로딩 된 == 연산자 사용  
- FString 변수와 TCHAR\* 배열 비교  
- FString::Equals() 함수 사용

FString::Equals() 함수 사용 시에는 ESearchCase Enum 값을 파라미터로 넘겨 대소문자 비교 유무를 설정해야 한다  
- ESearchCase::IgnoreCase = 대소문자 미구분  
- ESearchCase::CaseSensitive = 대소문자 구분

|  |
| --- |
| bool bCompare1 = TestString.Equals(OtherString, ESearchCase::IgnoreCase);  bool bCompare2 = TestString.Equals(OtherString, ESearchCase::CaseSensitive); |

### FString 클래스 검색

FString::Contains() 함수

찾고자 하는 문자열이 FString 내에 존재한다면 true, 존재하지 않으면 false를 반환  
- 검색 대상으로 FString, TCHAR\* 배열 사용 가능  
- ESearchCase Enum 값으로 대소문자 비교 유무 설정  
- ESearchDir Enum 값으로 검색 방향 지정 가능

|  |
| --- |
| bool bCompare = TestString.Contains(OtherString, ESearchCase::CaseSensitive, ESearchDir::FromEnd); |
| TestString에서 OtherString을 찾는다  - 대소문자 구분, 뒤에서부터 검색 |

FString::Find() 함수

찾고자 하는 문자열의 처음 인덱스를 반환, 찾는 값이 없다면 -1을 반환  
- FString::Contains() 함수와 동일하게 대소문자 비교 유모와 검색 방향 지정 가능  
- 검색 시작 인덱스를 지정할 수 있다.

|  |
| --- |
| int32 iCompare = TestString.Find(OtherString, ESearchCase::CaseSensitive, ESearchDir::FromStart, 10); |
| TestString 값에서 OtherString의 값을 찾는다  - 대소문자 구분, 인덱스 10의 위치부터 앞에서 뒤로 검색하는 코드 |

### FString 클래스 연결

FString 객체에 FString 객체를 이어 붙이는 것도 가능하다

|  |
| --- |
| TestString += OtherString;  FString AnotherString = TestString + OtherString; |

## 스트링 변환

각 클래스 타입에서 다른 클래스 타입의 스트링으로 변환하는 방법

|  |  |  |
| --- | --- | --- |
| 시작 클래스 | 변환할 클래스 | 예시 |
| FName | FString | TestString = TestName.ToString(); |
| FName | FText | TestText = FText::FromName(TestName); |
| FString | FName | TestName = FName(\*TestString);  \* FString->FName은 손실성 변환으로 위험하다  \* FName은 대소문자를 구분하지 않음 |
| FString | FText | TestText = FText::FromString(TestString); |
| FText | FString | TestString = TestText.ToString();  \* FText->FString은 일부 언어에서 변환 시 손실 위험이 있음 |
| FText | FName | 직접 변환방법이 없다  FText->FString->FName으로 변환하는 방법은 존재  \* 역시 손실성 변환으로 위험하다 |

# 언리얼 컨테이너

## TArray

언리얼 엔진의 가장 간단한 컨테이너 클래스, 템플릿 클래스  
- 타입이 같은 오브젝트들을 담을 수 있는 클래스

언리얼 엔진에서 가장 자주 쓰이는 컨테이너 클래스로 신속성, 메모리 효율성, 안전성을 위주로 디자인되었다.

원소 TArray  
- 배열에 저장되는 오브젝트 타입  
- TArray 컨테이너는 동질성 컨테이너로, 속하는 원소들이 모두 같은 타입이어야 한다.  
- 타입이 다른 원소를 하나의 TArray 컨테이너 저장할 수는 없다

TArray 컨테이너는 값의 타입으로 int32, float 같은 내장된 타입과 비슷하게 취급해야 한다  
- 확장을 고려하지 않았기 때문에 TArray 인스턴스를 new, delete로 생성, 소멸 시키는 것은 좋지 않다.  
- 원소는 값의 타입이도 하며 배열이 소유한다  
- TArray 컨테이너의 소멸은 컨테이너에 속한 원소의 소멸로 이어진다.

### 컨테이너 함수

|  |  |
| --- | --- |
| TArray<int32> IntArray; | TArray 컨테이너 생성 |

|  |  |
| --- | --- |
| IntArray.Init(10, 5); | Init 함수로 컨테이너에 값을 여러 개 채워넣는 방법  = {10, 10, 10, 10, 10} |
| TArray<FString> StrArray;  StrArray.Add(TEXT("Hello"));  StrArray.Emplace(TEXT("World")); | Add 또는 Emplace 함수  - 컨테이너 끝에 새 오브젝트를 추가한다.  - Add: 원소 타입의 인스턴스를 컨테이너에 복사  - Emplace: 지정한 파라미터를 사용하여 원소 타입의 인스턴스를 새로 생성한다  최종 결과는 같지만 Emplace 함수는 임시 변수를 생성하지 않는다  - 일반적으로 Emplace 함수가 Add 함수보다 좋은 점은 호출 되는 곳에서 임시 생성 후 컨테이너에 복사하는 불필요한 동작을 피할 수 있다.  - 단순한 타입에는 Add, 복잡한 타입에는 Emplace 함수를 사용한다.  - Add는 가시성이 좋기 때문 |

|  |
| --- |
| FString Arr[] = {TEXT("of"), TEXT("Tomorrow")};  StrArray.Append(Arr, ARRAY\_COUNT(Arr)); |
| 결과 = {"Hello", "World", "of", "Tomorrow"} |

- Append 함수는 다른 TArray 또는 일반 C 배열로의 포인터 및 해당 배열의 크기에 대수의 원소를 한번에 넣을 수 있다.

|  |
| --- |
| StrArray.AddUnique(Text("!")); // {"Hello", "World", "of", "Tomorrow", "!"}  StrArray.AddUnique(Text("!")); // 이미 존재하므로 ! 원소가 추가되지 않음 |

- AddUnique 함수는 기존 컨테이너에 동일한 원소가 없을 경우 새 원소를 추가한다.

|  |
| --- |
| StrArray.Insert(TEXT("Brave"), 1); {"Hello", "Brave", "World", "of", "Tomorrow", "!"} |

- Insert 함수는 단일 원소나 배열 사본을 주어진 인덱스에 추가할 수 있다.

|  |
| --- |
| StrArray.SetNum(8); // {"Hello", "Brave", "World", "of", "Tomorrow", "!", "", ""}  StrArray.SetNum(6); // {"Hello", "Brave", "World", "of", "Tomorrow", "!"} |

- SetNum 함수를 사용하여 컨테이너의 사이즈를 변경할 수 있다.  
- 설정된 숫자가 현재 컨테이너의 원소 수보다 큰 경우 기본 생성자의 원소 타입을 사용하여 새 원소를 만들어 추가한다  
- 작은 경우에는 원소들을 제거하여 크기를 줄인다.

### 컨테이너 반복처리

TArray 컨테이너의 엘리먼트에 대한 반복처리에 대한 부분

|  |
| --- |
| FString ResultStr;  for(auto& Str : StrArray)  {  ResultStr += Str;  ResultStr += TEXT(" ");  } |

- C++의 범위 for 기능을 사용하여 반복처리를 하는 구문

|  |
| --- |
| for(int32 Index = 0; Index != StrArray.Num(); Index++)  {  ResultStr += StrArray[Index];  ResultStr += TEXT(" ");  } |

- 일반적으로 인덱스 기반 반복 처리도 가능하다

|  |
| --- |
| for(auto Iter = StrArray.CreateConstIterator(); Iter; Iter++)  {  ResultStr += \*Iter;  ResultStr += TEXT(" ");  } |

반복자(이터레이터)를 사용하여 반복처리도 가능하다  
- CreateIterator와 CreateConstIterator로 반복자를 생성 가능하다  
- CreateIterator로 만든 반복자는 읽고 쓰는 것 모두 가능하다  
- CreateConstIterator로 만든 반복자는 읽기만 가능하다.

### 컨테이너 정렬

|  |
| --- |
| StrArray.Sort() |

- Sort() 함수를 호출하는 것으로도 간단하게 정렬이 가능하다.

|  |
| --- |
| StrArray.Sort([](const FString& A, const FString& B)  {  return A.Len() < B.Len();  } |

- 연산자 <를 사용해서 원소 값을 원하는 조건대로 정렬햘 수 있다.  
- FString의 경우 대소문자 구분 없이 사전식으로 비교한다  
- Sort() 함수는 기존 순서를 보장하지 않는 비안정 정렬 함수

|  |
| --- |
| StrArray.HeapSort([] (const FString& A, const FString& B)  {  return A.Len() < B.Len();  } |

- HeapSort() 함수는 힙 정렬을 수행하는 정렬 함수  
- 기존 순서를 보장하지 않는 비안정 정렬 함수

|  |
| --- |
| StrArray.StableSort([] (const FString& A, const FString& B)  {  return A.Len() < B.Len();  } |

- StableSort() 함수는 기존 순서를 유지시키는 안정 정렬 함수  
- 합병 정렬로 구현되어 있다.

### 컨테이너 쿼리

|  |
| --- |
| int32 Count = StrArray.Num(); |

Num() 함수를 사용하여 컨테이너의 원소의 개수를 확인할 수 있다.

|  |
| --- |
| FString\* StrPtr = StrArray.GetData(); |

- GetData() 함수를 사용하여 컨테이너의 원소에 대한 포인터를 반환받을 수 있다.  
- 반환 받은 포인터는 컨테이너가 존재할 때만, 컨테이너에 대한 변형이 적용되기 전에만 유효하다  
- 컨테이너가 const라면 반환되는 포인터도 const

|  |
| --- |
| uint32 ElementSize = StrArray.GetTypeSize(); |

- 컨테이너 원소의 크기에 대해서도 확인할 수 있다.

|  |
| --- |
| FString Element1 = StrArray[1]; |

- 원소 값을 operator[] 인덱싱을 통해서 얻어낼 수 있다

|  |
| --- |
| bool bValidM1 = StrArray.IsValidIndex(-1);  bool bValidM0 = StrArray.IsValidIndex(0);  bool bValidM5 = StrArray.IsValidIndex(5);  bool bValidM6 = StrArray.IsValidIndex(6); |

- IsValidIndex() 함수를 통해 컨테이너의 유효한 인덱스를 확인할 수 있다  
- 0 미만이거나 Num() 이상의 값을 전해주면 런타임 오류

|  |
| --- |
| StrArray[3] = StrArray[3].ToUpper(); |

- operator[]가 원소의 레퍼런스를 반환하므로 컨테이너가 const가 아니라면 []를 통해서 원소를 변형시키는 것도 가능하다  
- GetData() 함수와 동일하게 컨테이너가 const인 경우 const 레퍼런스를 반환한다.

|  |
| --- |
| FString ElementEnd = StrArray.Last();  FString ElementEnd0 = StrArray.Last(0);  FString ElementEnd1 = StrArray.Last(1);  FString ElementTop = StrArray.Top(); |

- Last() 함수를 통해 컨테이너 끝에서 역순으로 값을 가져올 수 있다.  
- Last() 함수의 기본 값은 0으로 Last(0)와 Last() 함수는 같은 값을 반환한다  
- Top() 함수는 Last(0) 함수와 동일한 값을 반환하며 인덱스를 받지 않는다

|  |
| --- |
| bool bHello = StrArray.Contains(TEXT("Hello"));  bool bGoodBye = StrArray.Contains(TEXT("GoodBye")); |

- Contains() 함수로 컨테이너에 특정 원소가 존재하는지 확인할 수 있다.

|  |
| --- |
| bool bLen5 = StrArray.ContainsByPredicate([] (const FString& Str)  {  return Str.Len() == 5;  });  bool bLen6 = StrArray.ContainsByPredicate([] (const FString& Str)  {  return Str.Len() == 6;  }); |

- ContainsByPredicate() 함수로 지정된 술부와 일치하는 원소가 존재하는지 확인도 가능하다.

|  |
| --- |
| int32 Index;  if(StrArray.Find(TEXT("Hello"), Index)) |

- Find() 함수를 사용하여 원소를 찾을 수 있다.  
- 원소가 존재하는 검사하고 존재할 때 해당 원소의 인덱스 값을 받고 싶다면 Find() 함수를 사용하고 파라미터에 인덱스를 받을 변수를 집어넣으면 된다.

|  |
| --- |
| int32 IndexLast;  if(StrArray.FindLast(TEXT("Hello"), IndexLast)) |

- 중복된 원소가 있는 상태에서 맨 마지막 원소의 인덱스를 찾으려는 경우 FindLast() 함수를 사용하면 된다  
- Find(), FindLast() 모두 원소 발견 여부를 bool 값으로 반환하고 존재할 경우 파라미터에 입력한 변수에 인덱스 값을 저장한다.

|  |
| --- |
| int32 Index2 = StrArray.Find(TEXT("Hello"));  int32 IndexLast2 = StrArray.FindLast(TEXT("Hello"));  int32 IndexNone = StrArray.Find(TEXT("None")); |

- Find(), FindLast() 함수로 원소를 직접 반환하는 것도 가능하다  
- 파라미터에 인덱스를 넘겨주지 않을 경우 정수형을 반환하는 식으로 동작한다.  
- 원소를 찾지 못한 경우 INDEX\_NONE 값이 반환된다.

|  |
| --- |
| int32 Index = StrArray.IndexOfByKey(TEXT("Hello")); |

- IndexOfByKey() 함수는 operator==(ElementType, KeyType)이 존재하는 키 유형에 대해 동작한다.  
- IndexOfByKey() 함수는 처음 찾은 원소의 인덱스를 반환한다  
- 원소를 찾지 못한다면 INDEX\_NONE 값이 반환된다.

|  |
| --- |
| int32 Index = StrArray.IndexOfByPredicate([] (const FString& Str)  {  return Str.Contains(TEXT("r"));  }); |

- IndexOfByPredicate() 함수는 지정도니 술부에 일치하는 첫 원소의 인덱스를 찾는데 사용한다.  
- 찾은 것이 없다면 INDEX\_NONE을 반환한다.

|  |
| --- |
| auto\* Len5Ptr = StrArray.FindByPredicate([] (const FString& Str)  {  return Str.Len() == 5;  });  auto\* Len5Ptr = StrArray.FindByPredicate([] (const FString& Str)  {  return Str.Len() == 6;  }); |

- FindByPredicate() 함수도 IndexOfByPredicate() 함수처럼 사용된다  
- 차이점은 FindByPredicate() 인덱스의 포인터를 반환한다.

|  |
| --- |
| auto Filter = StrArray.FilterByPredicate([[] (const FString& Str)  {  return (!Str.IsEmpty()) && (Str[0] < TEXT('M'));  }); |

- FilterByPredicate() 함수는 특정 술부에 일치하는 원소의 컨테이너를 가져올 수 있다.

# Tset

# TSubclassOf 템플릿 클래스

UCLASS 타입의 안전성을 보장해주는 템플릿 클래스

|  |
| --- |
| UPROPERTY(EditDefaultsOnly, Category = Particle)  UClass\* ParticleType; |
| 그냥 UPROPERTY타입의 UClass를 만들고 디자이너에게 파생 클래스만 할당하라고 가이드할 수 있음  - 실수로 다른 클래스를 할당할 가능성이 존재함  - 이러한 상황을 막기 위해 TSubclassOf 템플릿이 존재함  - 클래스 할당의 선택지를 제한하는 것 |
| UPROPERTY(EditDefaultsOnly, Category = Particle)  TSubclassOf<UParticleSystemComponent> ParticleType; |
| TSubclassOf 템플릿을 사용하여 선언하면 에디터의 프로퍼티 창에서 UParticleSystemComponent의 파생 클래스만 선택되도록 제한할 수 있다.  - UPROPERTY() 안전성에 추가로 C++ 수준에서의 타입 안전성도 확보 가능 |

- 호환되지 않는 TSubclassOf 타입을 서로에게 할당하려고 한다면 컴파일 오류가 발생한다  
- 범용 UClass를 할당하려고 하는 경우 할당이 가능한지 검증하는 런타임 검사를 수행  
- 런타임 검사에 실패한다면 결과 값으로 nullptr를 반환

|  |
| --- |
| UClass\* ClassA = UParticleSystemComponent::StaticClass();  TSubclassOf<UParticleSystemComponent> ClassB;  ClassB = ClassA; |
| 이 경우에는 런타임 검사를 수행하게 된다  - 수행 완료시 할당  - 런타임 검사를 실패하면 nullptr이 반환 |
| TSubclassOf<UParticleSystemComponent\_Splash> ClassC;  ClassB = ClassC; |
| 컴파일 에러 |

# 언리얼 오브젝트 처리

언리얼 엔진에는 내부적으로 게임 오브젝트를 처리하는 견고한 시스템이 존재함  
- 클래스는 UCLASS() 매크로  
- 프로퍼티는 UPROPERTY() 매크로  
- 함수는 UFUNCTINO() 매크로

이러한 매크로를 사용하여 선언을 하면 오브젝트 처리 시스템에서 인식하여 접근할 수 있게 된다.

## 1: 자동 프로퍼티 초기화

UObject는 생성자 호출 전 초기화 시 자동으로 0으로 채워진다  
- 클래스, UProperty, 네이티브 멤버 모두 전체적으로 동일  
- 이후 멤버는 클래스 생성자의 커스텀 값으로 초기화가 가능

## 2: 레퍼런스 자동 업데이트

AActor 또는 UActorComponent가 소멸되거나 다른 식으로 플레이에서 제거되면, 리플렉션 시스템이 보이고 있는 그에 대한 모든 레퍼런스가 알아서 null로 변한다  
- 허상 참조를 방지하여 문제의 소지를 줄이는 장점이 있다  
- 코드 부분에서 AActor와 UActorComponent 포인터를 소멸시키는 경우 null이 된다는 뜻

\* 최고의 장점은 null 검사 안전성이 높다는 것  
- 일반적인 null 포인터의 경우 null이 아닌 포인터가 삭제된 메모리를 가리키는 경우 둘 다 감지해내기 때문이다.

레퍼런스 자동 업데이트는 UPROPERTY로 마킹되거나 언리얼 엔진 컨테이너 클래스에 저장된 AActor 또는 UActorComponent 레퍼런스에만 적용된다  
- raw 포인터에 저장된 오브젝트 레퍼런스는 언리얼 엔진이 알지 못하기 때문에 null이 되거나 가비지 컬렉션이 방지되지 않는다

모든 UObject\* 변수가 UPROPERTY로 처리되야 하는 것은 아니다  
- UProperty가 아닌 오브젝트 포인터가 필요한 경우 TWeakObjectPtr의 사용을 고려  
- 약 포인터로 가비지 컬렉션이 방지되지는 않지만 접근 전 질의를 통해 유효성 검사가 가능  
- 가리키는 오브젝트가 소멸되는 경우 null 설정도 가능함

참조된 UObject UPROPERTY가 자동으로 null이 되는 다른 경우는 에디터에서 애셋을 강제로 삭제한 경우  
- 이로 인해 애셋인 UObject에 대한 모든 코드 작업 시 이 포인터가 null이 되도록 처리해야 한다.

## 3: 직렬화

UObject가 직렬화될 때 모든 UPROPERTY 값은 명시적으로 휘발성 마킹 또는 생성자 이후 기본 값에서 미변경 상태가 아닌 이상 자동으로 읽고 쓰기가 가능하다

UPROPERTY가 추가 또는 제거될 때, 기존 컨텐츠 로드는 매끄럽게 처리된다.  
- 새 프로퍼티는 새 CDO에서 기본 값을 복사해온다.  
- 제거된 프로퍼티는 말 없이 무시된다.

커스텀 동작 방식이 필요한 경우 UObject::Serialize 함수로 오버로딩 가능  
- 데이터 오류, 버전 번호 검사, 데이터 포맷 변경 시 자동 변환 또는 업데이트 수행 등에 유용함

### 직렬화에 대해서

객체를 저장하거나 메모리, 데이터베이스 혹은 파일로 옮기려고 할 때 필요환 과정  
- 객체를 바이트 스트림으로 바꾸는 것  
- 객체에 저장된 데이터를 스트림에 쓰기 위해 연속적엔 데이터로 변환하는 과정

주된 목적은 객체를 상태 그대로 저장하고 필요할 때 다시 생성하여 사용하는 것이다.

역직렬화: 직렬화의 반대말로 네트워크, 영구저장소 등에서 바이트 스트림을 가져와서 객체가 저장되었던 그 상태로 변환하는 것이다.

직렬화를 통해 생긴 바이트 스트림은 플랫폼에 독립적이다  
= 직렬화된 객체는 다른 플랫폼에서 역직렬화를 할 수 있다

## 4: 프로퍼티 값 업데이트

UClass의 CDO가 변경되면 엔진은 그 클래스의 모든 인스턴스 로드 시 알아서 변경사항을 적용한다.  
- 주어진 오브젝트 인스턴스에 대해서 업데잉트 된 변수 값이 이전 CDO 값과 일치하면 새로운 CDO에 저장된 CDO 값으로 업데이트 된다.  
- 변수 값이 다른 경우, 그 변수가 의도적으로 설정되었다 가정하여 그 변경사항을 보존한다

## 5: 에디터 통합

UObject와 UProperty는 에디터에서 인식되고, 에디터는 별도의 코드를 작성할 필요 없이 해당 값들을 자동으로 노출시킬 수 있다.  
- 블루프린트 비주얼 스크립팅 시스템으로의 통합이 가능함

변수와 함수의 노출 및 접근 여부를 제어할 수 있는 옵션이 존재함

|  |  |
| --- | --- |
| 프로퍼티 지정자 | 효과 |
| BlueprintCallable | 멀티 캐스트 델리게이트 전용.  - 프로퍼티를 노출시켜야 블루프린트 코드에서 호출할 수 있다. |
| BlueprintReadOnly | 이 프로퍼티는 블루프린트에서 읽을 수는 있지만 변경은 안된다  - BlueprintReadWrite와 호환 안됨 |
| BlueprintReadWrite | 이 프로퍼티는 블루프린트에서 읽고 쓸 수 있다.  - BlueprintReadOnly 지정자와 호환 안됨 |
| EditAnywhere | 이 프로퍼티는 아키타입, 인스턴스 양쪽의 프로퍼티 창에서 편집 가능  - 어떤 Visible 지정자와도 호환되지 않는다. |

- 자주 사용되는 프로퍼티 지정자들만 추려놓은 것

프로퍼티: <https://docs.unrealengine.com/4.26/ko/ProgrammingAndScripting/GameplayArchitecture/Properties/>

## 6: 런타임 타입 정보 및 형변환

UObject는 언리얼 엔진 리플렉션 시스템의 일부로 어떠한 UClass인지 알고 있어 형변환을 실시간으로 할 수 있다.  
- 네이티브 코드에서 모든 UObject 클래스에는 그 부모 클래스로 설정된 커스텀 Super typedef가 존재하여 오버로딩에 대한 제어가 쉽게 가능하다

|  |
| --- |
| class AEnemy : public ACharacter  {  virtual void Speak()  {  Say("Time to Fight!");  }  };  class AMegaBoss : public AEnemy  {  virtual void Speak()  {  Say("Powering up!");  Super::Speak();  }  }; |
| AMegaBoss 인스턴스의 Speak 함수를 호출할 경우  "Powering up! Time to Fight!"를 반환할 것이다. |

템플릿 Cast 함수를 사용하여 베이스 클래스에서의 오브제그를 좀 더 파생된 클래스로 안전하게 형변환하거나 IsA를 사용하여 오브젝트가 특정 클래스의 것인지 질의하는 것도 가능하다

|  |
| --- |
| class ALegendaryWeapon : public AWeapon  {  void SlayMegaBoss()  {  TArray<AEnemy> EnemyList = GetEnemyListFromSomewhere();  for(AEnemy Enemy : EnemyList)  {  AMegaBoss\* MegaBoss = Cast<AMegaBoss>(Enemy);  if(MegaBoss)  {  Incinerate(MegaBoss);  }  }  }  }; |
| Cast<>() 함수를 통해 AEnemy를 이용하여 AMegaBoss로 형변환 시도 - 형변환을 시도한 오브젝트가 AMegaBoss가 아닌 경우 Cast<>() 함수는 nullptr을 반환하므로 적절한 대응이 가능하다. |

## 7: 가비지 컬렉션

언리얼 엔진에서는 더 이상 참조되지 않거나 명시적으로 소멸을 예약한 UObject를 주기적으로 정리하는 가비지 컬렉션 스키마를 사용한다  
- 엔진에서는 레퍼런스 그래프를 만들어 어느 오브젝트가 사용중이고 어느 것이 사용중이지 않는 지 알아낸다  
- 이 그래프에는 "루트 세트"라 지정된 오브젝트 세트가 있어 어떤 오브젝트라도 루트 세트에 추가할 수 있다.  
- 가비지 컬레겻ㄴ이 발생하면, 엔진은 루트 세트부터 시작하여 알려진 UObject 레퍼런스 트리를 검색하여 참조된 오브젝트를 전부 추적할 수 있다.  
- 참조되지 않은 오브젝트(트리 검색에서 찾지 못한 것)들은 더 이상 필요하지 않다고 가정, 제거한다.

일반적으로 살려두고자 하는 오브젝트에 UPROEPRTY 레퍼런스를 유지하거나 그에 대한 포인터를 TArray또는 다른 언리얼 엔진 컨테이너 클래스에 저장해야 한다.  
- 액터와 그 컴포넌트는 예외가 되는 경우도 있다.  
- 액터는 보통 자신이 속한 레벨처럼 루트 세트로 다시 링크되는 오브젝트에 레퍼런싱된다  
- 액터의 컴포넌트는 액터 자체에 레퍼런싱 된다.  
- 액터는 자신의 Destroy() 함수를 호출하여 명시적으로 소멸을 마킹할 수 있고 이것은 진행중인 게임에서 액터를 제거하기 위한 표준 방식이다  
- 컴포넌트는 DestroyComponent() 함수로 명시적으로 소멸시킬 수 있고 보통은 소유 액터가 게임에서 제거될 때 소멸된다.

언리얼 엔진의 가비지 컬렉션을 효율적으로, 부하를 최소화시키기 위한 기능이 다수 존재한다

|  |  |
| --- | --- |
| 세팅 | 기능 설명 |
| Create Garbage  Collector UObject  Clusters | 프로젝트 세팅에서 키거나 끌 수 있다(디폴트 값 = 켜짐)  - 켜져 있을 경우 관련된 오브젝트들을 하나의 가비지 컬렉션 클러스터에 묶어 클러스터 하나만 검사할 수 있도록 만든다 - 전체 클러스터를 하나의 오브젝트로 취급할 수 있어 도달 가능성 파악이 빨라지지만, 클러스터의 개별 항목 전부를 같은 프레임에 언해시 및 삭제 준비를 한다 = 클러스터가 너무 크면 버벅인다  - 일반적으로 클러스터를 만들면 가비지 컬렉션 퍼포먼스가 향상되어 도달 가능성 분석에 소요되는 시간이 줄어든다. |
| Merge GC Clusters | 클러스터 병합을 켜변 한 클러스터의 오브젝트가 다른 클러스터의 오브젝트를 참조할 대 클러스터를 합치도록 한다.  - 병합을 유발시킨 레퍼런스를 지워도 새로 병합된 클러스터는 어떠한 방식으로든 분해되거나 나뉘지 않는다  \* 기능의 정상 작동을 위해 Create Garbage Collector UObject Clusters 옵션도 켜져 있어야 한다.  - 가비지 컬렉터의 언해시 및 오브젝트 소멸 작업 빈도를 낮추지만 더욱 많은 수의 오브젝트를 한 번에 언해시 및 소멸시킨다.  - 클러스터 병합을 하지 않았다면 가비지 컬렉션이 일어나지 않을 수 있다  - 클러스터 내의 오브젝트로의 레퍼런스가 존재하면 전체 클러스터가 가비지 컬렉션 대상에서 제외되기 때문 |
| Actor Clustering  Enabled | 프로젝트 세팅에서 옵션을 켜고 bCanBeInCluster 함수가 true를 반환하도록 덮어쓰면 액터를 클러스터에 넣을 수 있다.  - 기본적으로 스태틱 메시 액터와 리플렉션 캡처 컴포넌트를 제외한 액터 및 컴포넌트에는 이 기능이 꺼져있다.  - 이 기능은 한 번에 소멸시킬 것으로 예상되는 액터 그룹을 만드는 데 좋다  - 일반적으로 레벨에 배치한 스태틱 메시가 속한 서브 레벨을 언로드하지 않고서는 소멸되지 않도록 하는 것 |
| Blueprint Clustering  Enabled | 블루프린트의 UBlueprintGeneratedClass 및 관련 데이터(예를 들어 공유 UPROPERTY 및 UFUNCTION 데이터 등)는 이 세팅을 켜서 클러스터로 묶을 수 있다.  - 이렇게 생킨 클러스터는 블루프린트의 개별 인스턴스가 아닌 Blueprint Generated Class 자체를 참조한다 |
| Time Between Purging Pending Kill Objects | 프로젝트 세팅에서 가비지 컬렉션 발동 빈도를 조절할 수 있다.  - 특히 버벅임 방지에 좋다  - 컬렉션 발동 간격을 줄여 다음 도달 가능성 분석 패스에 걸릴 도달 불가 오브젝트 발생 가능성을 낮추고 동시에 너무 많은 액터를 정리하느라 발생할 수 있는 버벅임도 피할 수 있다. |

가비지 컬렉터 퍼포먼스 튜닝에 흔히 사용되는 세팅

## 8: 네트워크 리플리케이션

오브젝트 처리 시스템에는 네트워크 통신과 멀티플레이어 게임을 원활하게 만들기 위한 함수성 세트가 포함되어 있다.

<https://docs.unrealengine.com/4.26/ko/InteractiveExperiences/Networking/>

UPROPERTY에는 태그를 붙여 네트워크 플레이 도중 데이터의 리플리케이트 여부를 엔진에게 알릴 수 있다.  
- 흔히 쓰이는 모델은 서버에서 변수가 변경되면 엔진에서 그 변화를 감지하여 모든 클라이언트에 신뢰성있게 전송하는 것  
- 클라이언트에서는 리플리케이션을 통해 변수가 변할 때, 옵션을 통해 콜백 함수를 받을 수 있다.

UFUCNTION 역시 태그를 붙여 원격 머신에서 실행시킬 수 있다.  
- server 함수는 클라이언트 머신에서 호출 시 실제로는 서버 머신에서 해당 액터의 서버 버전이 실행되도록 한다.  
- client 함수는 서버에서 호출될 수는 있지만 해당 액터의 소유중인 클라이언트 버전이 실행된다.

# 스마트 포인터 라이브러리

## 스마트 포인터(Smart Pointer)

C++ 프로그램 상에서 new 키워드를 사용하여 동적으로 할당한 메모리는 반드시 delete 키워드를 사용하여 해제해야 한다  
- C++ 에서는 메모리 누수로부터 프로그램의 안전성을 보장하기 위해서 스마트 포인터가 제공된다  
- 스마트 포인터는 포인터처럼 동작하는 클래스 템플릿으로 사용이 끝난 메모리를 자동으로 해제해준다.

스마트 포인터의 동작

보통 new 키워드를 사용해 기본 포인터(raw pointer)가 실제 메모리를 가리키도록 초기화하고 기본 포인터를 스마트 포인터에 대입하여 사용한다  
- 이렇게 정의된 스마트 포인터의 수명이 다하면 소멸자는 delete 키워드를 사용하여 할당된 메모리를 자동으로 해제한다  
- new 키워드가 반환하는 주소 값을 스마트 포인터에 대입하면 따로 메모리를 해제할 필요가 없어진다.

스마트 포인터의 종류

C++ 11 이후부터 3종류의 스마트 포인터가 제공된다

1: unique\_ptr(유니크 포인터)

2: shared\_ptr(쉐어드 포인터)

3: weak\_ptr(위크 포인터)

### 1: Unique\_ptr(유니크 포인터)

하나의 스마트 포인터만이 특정 객체를 소유할 수 있도록 객체에 소유권 개념을 도입한 스마트 포인터  
- 스마트 포인터는 해당 객체의 소유권을 가지고 있을 때만 소멸자가 해당 객체를 삭제할 수 있다.  
- move() 멤버 함수를 통해 소유권을 이전할 수는 있지만 복사는 불가능하다  
- 소유권이 이전되면 소유권을 이전한 인스턴스는 해당 객체를 소유하지 않게 되는 것

|  |  |
| --- | --- |
| unique\_ptr<int> ptr01(new int(5)); | int 형 유니크 포인터인 ptr01을 선언 및 초기화 |
| auto ptr02 = move(ptr01); | ptr01에서 ptr02로 소유권을 이전함 |
| unique\_ptr<int> ptr03 = ptr01; | 대입 연산자를 이용한 복사는 불가능 |
| ptr02.reset(); | ptr02가 가리키고 있는 메모리 영역 삭제 |
| ptr01.reset(); | ptr01이 가리키고 있는 메모리 영역 삭제 |

C++14 이후부터는 make\_unique() 함수를 사용하면 유니크 포인터를 안전하게 생성할 수 있다  
- make\_unique() 함수는 전달받은 인수를 사용해 지정된 타입의 객체를 생성한다  
- 생성된 객체를 가리키는 unique\_ptr을 반환함  
- 이 함수를 통해 예외 발생에 대해서 안전하게 대처할 수 있다.

일반 포인터와는 달리 delete 키워드를 통해 명시적으로 메모리를 해제할 필요가 없다는 것이 장점이다.

### 2: Shared\_ptr(쉐어드 포인터)

하나의 특정 객체를 참조하는 스마트 포인터가 총 몇 개인지를 참조하는 스마트 포인터

이렇게 참조하고 있는 스마트 포인터의 개수를 참조 횟수(레퍼런스 카운트)라고 한다  
- 참조 횟수는 특정 객체에 새로운 쉐어드 포인터가 추가될 때마다 1씩 증가한다  
- 쉐어드 포인터의 수명이 다 할 때마다 1씩 감소한다  
- 마지막 쉐어드 포인터의 수명이 다하여 참조 횟수가 0이 되면 delete 키워드를 사용하여 메모리를 자동으로 해제한다.

|  |  |
| --- | --- |
| shared\_ptr<int> ptr01(new int(5)); | int형 쉐어드 포인터 ptr01을 선언하고 초기화 |
| cout << ptr01.use\_count() << '\n'; | 참조 횟수 = 1 |
| auto ptr02(ptr01); | 복사 생성자를 이용한 초기화도 가능하다 |
| cout << ptr01.use\_count() << '\n; | 참조 횟수 = 2 |
| auto ptr03 = ptr01; | 대입을 통한 초기화도 가능하다 |
| cout << ptr01.use\_count() << '\n'; | 참조 횟수 = 3 |

- use\_count() 함수는 쉐어드 포인터 객체가 현재 가리키고 있는 리소스를 참조 중인 소유자의 수를 반환해준다.

make\_shared() 함수를 사용하면 쉐어드 포인터 인스턴스를 안전하게 생성할 수 있다  
- make\_shared() 함수는 전달 받은 인수를 사용하여 지정된 타입의 객체를 생성하고, 생성된 객체를 가리키는 쉐어드 포인터를 반환한다  
- 역시 예외 발생에 안전하게 대처할 수 있게 해주는 함수

|  |
| --- |
| shared\_ptr<Person> hong = make\_shared<Person>("길동", 29); // 참조 횟수 = 1  auto han = hong; // 참조 횟수 = 2 |
| 프로그램이 종료되면 자동으로 메모리가 해제된다. |

### 3: weak\_ptr(위크 포인터)

하나 이상의 쉐어드 포인터 인스턴스가 소유하는 객체에 대한 접근을 제공하지만 소유자의 수에 포함되지 않는 스마트 포인터  
- 쉐어드 포인터는 참조 횟수를 기반으로 동작하는 스마트 포인터  
- 서로가 상대방을 가리키는 쉐어드 포인터를 갖게 된다면 참조 횟수는 절대 0이 될 수 없고 영원히 메모리가 해제되지 않는다(순환 참조; circular reference)  
- 위크 포인터는 쉐어드 포인터 사이의 순환 참조를 제거하기 위해서 사용된다.

## 언리얼 스마트 포인터 라이브러리

언리얼 스마트 포인터 라이브러리는 메모리 할당과 추적의 부담을 해소하도록 설계된 C++11 스마트 포인터의 커스텀 구현  
- 업계 표준인 공유 포인터, 위크 포인터, 유니크 포인터가 포함되어 있다  
- null이 불가능한 공유 포인터와 같은 역할을 하는 공유 레퍼런스도 추가됨  
- 언리얼 오브젝트는 게임 코드에 더 최적화된 별도의 메모리 추적 시스템을 사용하여 이러한 클래스들은 UObject 시스템과 사용할 수 없다

### 스마트 포인터 타입

|  |  |
| --- | --- |
| 타입 | 설명 |
| Shared Pointers  (TSharedPtr) | 쉐어드 포인터는 참조하는 오브젝트를 소유하며, 무기한으로 오브젝트의 소멸을 방지한다  - 참조하는 쉐어드 포인터 또는 쉐어드 레퍼런스가 없을 경우에는 궁극적으로 오브젝트를 소멸시킨다  - 웨어드 포인터는 어느 오브젝트도 참조하지 않는 빈 상태일 수 있다  - 모든 null이 불가능한 쉐어드 포인터는 참조하는 오브젝트에 쉐어드 레퍼런스를 생성할 수 있다. |
| Shared References  (TSharedRef) | 쉐어드 레퍼런스는 참조하는 오브젝트를 소유하는 측면에서 쉐어드 포인터와 같은 역할을 한다.  쉐어드 레퍼런스는 항상 null이 불가능한 오브젝트를 참조해야 한다  - 쉐어드 포인터에는 이러한 제약이 없어 쉐어드 레퍼런스는 언제나 쉐어드 포인터로 변환될 수 있어 변환된 쉐어드 포인터는 유효한 오브젝트를 참조한다는 것을 보장할 수 있다.  - 참조한 오브젝트가 null이 불가능한 오브젝트인 것을 보장하길 원하거나 공유된 오브젝트 소유권을 보여주길 원할 경우, 쉐어드 레퍼런스를 사용하면 된다. |
| Weak Pointer  (TWeakPtr) | 위크 포인터는 쉐어드 포인터와 유사하지만 참조하는 오브젝트를 소유하지 않아 생명 주기에 영향을 주지 않는다  - 참조 주기에 영향을 주지 않기 때문에 매우 유용하지만 사전 경고 없이 null이 될 수 있다는 뜻이다  - 위크 포인터는 참조하는 오브젝트에 쉐어드 포인터를 생성할 수 있고, 프로그래머들에게 일시적으로 오브젝트에 대한 안전한 접근을 보장한다. |
| Unique Pointers  (TUniquePtr) | 유니크 포인터는 참조하는 오브젝트를 유일하고 명시적으로 소유한다 - 특정 자원에 대해서 하나의 유니크 포인터만 존재한다 - 유니크 포인터는 소유권을 이전할 수 있지만 공유할 수는 없다  - 유니크 포인터를 복사하면 컴파일 에러가 나온다  - 유니크 포인터가 스코프(영역)을 벗어나면 참조하는 오브젝트가 자동으로 소멸한다. |

각 스마트 포인터들은 참조하는 오브젝트의 생명 주기에 영향을 줄 수 있으며 스마트 포인터 타입마다 오브젝트에 주는 제한과 효과도 달라지기 때문에 적절한 것을 골라 사용해야 한다

유니크 포인터가 참조하는 오브젝트에 쉐어드 포인터, 쉐어드 레퍼런스를 생성하면 위험하다  
- 유니크 포인터가 소멸하면 해당 오브젝트도 함께 소멸한다  
- 쉐어드 포인터, 쉐어드 레퍼런스가 참조하는 오브젝트에 유니크 포인터를 생성하는 것도 위험하다

### 스마트 포인트의 장점

|  |  |
| --- | --- |
| 장점 | 설명 |
| 메모리 누수 방지 | 위크 포인터를 제외한 스마트 포인터들은 더 이상 공유된 레퍼런스가 없다면 오브젝트가 자동으로 소멸된다 |
| 위크 레퍼런싱 | 위크 포인터는 참조 주기에 영향을 주지 않는다  - 삭제된 오브젝트를 참조하는 포인터를 방지함 |
| 선택적인 스레드 안전 | 언리얼 스마트 포인터 라이브러리에는 멀티 스레드에 걸쳐 참조 카운팅을 관리하는 코드인 스레드 세이프 코드가 포함됨  - 스레드 안정성이 필요하지 않다면 그 대신 햐상된 퍼포먼스를 구현할 수 있다. |
| 런타임 안전성 | 쉐어드 레퍼런스는 절대 null일 수 없으며 언제든 참조 해제가 가능하다 |
| 명확한 의도 | 관찰자 중에서 오브젝트의 소유자를 쉡게 분별할 수 있다. |
| 메모리 | 스마트 포인터는 64비트의 c++ 포인터 크기의 반  - 유니크 포인터만 C++ 포인터와 크기가 동일함 |

### 스마트 포인터 사용

#### 1: 헬퍼 클래스와 함수

언리얼 스마트 포인터 라이브러리는 스마트 포인터를 쉽고 직관적으로 사용할 수 있도록 다양한 헬퍼 클래스와 함수를 제공한다

|  |  |
| --- | --- |
| 헬퍼 클래스 | 설명 |
| TSharedFromThis | TSharedFromThis에서 클래스를 파생시키면 AsShared 또는 SharedThis 함수가 추가된다  - 이러한 함수들을 통해 오브젝트에 대한 TSharedRef를 구할 수 있다 |
| 헬퍼 함수 | 설명 |
| MakeShared  MakeShareable | 일반 C++ 포인터로 쉐어드 포인터를 생성한다  Make Shared 함수는 새 오브젝트 인스턴스와 레퍼런스 컨트롤러를 한 메모리 블록에 할당하지만 오브젝트가 public 생성자를 제공해야 한다  Make Shareable 함수는 덜 효율적이지만 오브젝트의 생성자가 private 여도 접근이 가능하여 직접 생성하지 않은 오브젝트에 대한 소유권을 가질 수 있으며 오브젝트를 소멸시킬 경우에는 커스텀 비헤이비어가 지원된다. |
| StaticCastSharedRef  StaticCastSharedPtr | 정적인 형변환 유틸리티 함수  주로 파생된 타입으로 다운캐스트 하는데 사용된다 |
| ConstCastShareRef  ConstCastSharedPtr | const 스마트 레퍼런스 또는 스마트 포인터를 mutable 스마트 레퍼런스 또는 스마트 포인터로 각각 변환한다. |

#### 2: 스마트 포인터 구현 세부사항

언리얼 스마트 포인터 라이브러리의 모든 스마트 포인터는 기능성, 효율성 측면에서 일반적인 특징을 공유한다.

속도  
- 스마트 포인터를 사용할지 고려할 때에는 항상 퍼포먼스에 대해서 생각해야 한다  
- 스마트 포인터는 특정 고수준 시스템이나 자원 관리 또는 툴 프로그램에 매우 적합하지만 일부 스마트 포인터 타입은 C++ 기본 포인터보다 더 느리다  
- 이런 오버헤드로 인해 렌더링 같은 로우 레벨 엔진 코드에는 덜 유용하다

스마트 포인터의 일반적인 퍼포먼스 이점

|  |
| --- |
| 모든 연산이 고정 비용 |
| 빌드를 출시할 때, 대부분의 스마트 포인터들을 참조 해제하는 속도가 C++ 기본 포인터만큼 빠르다 |
| 스마트 포인터들을 복사해도 절대 메모리가 할당되지 않는다 |
| 스레드 세이프 스마트 포인터는 lockless(lockFree) 구조 |

스마트 포인터의 일반적인 퍼포먼스 문제점

|  |
| --- |
| 스마트 포인터의 생성 및 복사는 C++ 기본 포인터의 생성 및 복사보다 더 많은 오버헤드가 발생한다 |
| 참조 카운트를 유지하면 기본 연산에 주기가 추가된다 |
| 일부 스마트 포인터는 C++ 기본 포인터보다 메모리 사용량이 높다 |
| 레퍼런스 컨트롤러에는 2번의 힙 할당량이 있다  - MakeShareable 함수 대신에 MakeShared 함수를 사용하면 2번째 할당을 피할 수 있으며, 퍼포먼스를 개선할 수 있다. |

침범형 접근자

쉐어드 포인터의 경우 비침범형  
- 오브젝트가 스마트 포인터의 소유 하에 있는지 알 수 없다  
- 이런 속성은 일반적으로 문제가 없지만 오브젝트를 쉐어드 레퍼런스 또는 쉐어드 포인터로 접근하려는 경우가 있을 수 있다  
- 이러한 경우에는 오브젝트의 클래스를 템플릿 매개변수로 사용하여 TSharedFromThis에서 오브젝트의 클래스를 파생시켜야 한다

TSharedFromThis는 두 가지 함수 AsShared 및 SharedThis를 제공하며, 두 함수로 오브젝트를 쉐어드 레퍼런스로 변환하고, 쉐어드 레퍼런스를 또 쉐어드 포인터로 변환할 수 있다.  
- 항상 쉐어드 레퍼런스를 반환하는 클래스 팩토리나 쉐어드 레퍼런스 또는 쉐어드 포인터를 요구하는 시스템 오브젝트를 넣을 때 특히 유용하다.

AsShared 함수는 호출되는 오브젝트의 부모 타입일 수 있는 TSharedFromThis에 템플릿 인자로서 전달된 본래 타입의 클래스를 반환한다  
- SharedThis 함수는 this에서 타입을 직접 파생시켜 해당 타입의 오브젝트를 참조하는 스마트 포인터를 반환한다.

|  |
| --- |
| class FRegistryObject;  class FMyBaseClass : public TSharedFromThis<FMyBaseClass>  {  virtual void RegisterAsBaseClass(FRegistryObject\* RegistryObject)  {  TSharedRef<FMyBaseClass> ThisAsSharedRef = AsShared();  RegistryObject->Register(ThisAsSharedRef);  }  }; |
| class FMyDerivedClass : public FMyBaseClass  {  virtual void Register(FRegistryObject\* RegistryObject) override  {  TSharedRef<FMyDerivedClass> AsSharedRef = SharedThis(this);  RegistryObject->Register(AsSharedRef);  }  }; |
| class FRegistryObject  { void Register(TSharedRef<FMyBaseClass>); }; |

\* AsShared 함수나 SharedThis 함수를 생성자로 호출하면 안된다  
- 쉐어드 레퍼런스가 선언되지 않은 상태로 충돌 또는 Assert 가 발생한다

형변환

쉐어드 포인터와 쉐어드 레퍼런스는 언리얼 스마트 포인터 라이브러리에 포함되어 잇는 여러가지 지원 함수를 통해 형변환할 수 있다.  
- 업 캐스팅은 C++포인터와 마찬가지로 묵시적  
- ConstCastSharedPtr 함수로 const cast 연산자를 사용할 수 있으며 StaticCastSharedPtr 함수로 static cast 연산자를 사용할 수 있다.  
- 런타임 타입 정보가 없어서 동적 형변환은 지원되지 않는다

|  |
| --- |
| TSharedPtr<FDragDropOperation> Operation = DragDropEvent.GetOperation();  TSharePtr<FAsseDragDropOp> DRagDropOp = StaticCastSharedPtr<FAssetDragDropOp>(Operation); |

스레드 안정성

기본적으로 스마트 포인터는 싱글 스레드가 접근하는 것이 안전하다  
- 멀티 스레드가 접근해야 한다면 스마트 포인터 클래스의 스레드 세이프 버전을 사용해야 한다

|  |
| --- |
| TSharedPtr<T, ESPMode::ThreadSafe>  TSharedRef<T, ESPMode::ThreadSafe>  TWeakPtr<T, ESPMode::ThreadSafe>  TSharedFromThis<T, ESPMode::ThreadSafe> |

- 이러한 스레드 세이프 버전은 원자적 참조 카운팅으로 인해 기본보다 느리지만 행동은 C++ 포인터와 동일하다  
- 두 개 이상의 스레드가 포인터에 접근하지 않는 것이 확실하지 않다면 스레드 세이프 버전을 사용하지 않아도 되고 퍼포먼스를 향상시킬 수 있다.

#### 3: 제한 사항

함수에 데이터를 TSharedRef 또는 TSharedPtr 매개변수로 넣지 않는 것을 권장한다  
- 이러한 데이터의 해제와 참조 카운팅으로 인해 오버헤드가 발생하게 된다  
- 레퍼런스된 오브젝트를 const&로 넣는 것이 권장된다

쉐어드 포인터를 불완전한 타입이나 형식으로 미리 선언할 수 있다.

쉐어드 포인터는 언리얼 오브젝트(UObject)와 호환되지 않는다  
- UObject 관리를 위한 별도의 메모리 관리 시스템이 존재하며 스마트 포인터 시스템과는 서로 완전히 다른 시스템이다.