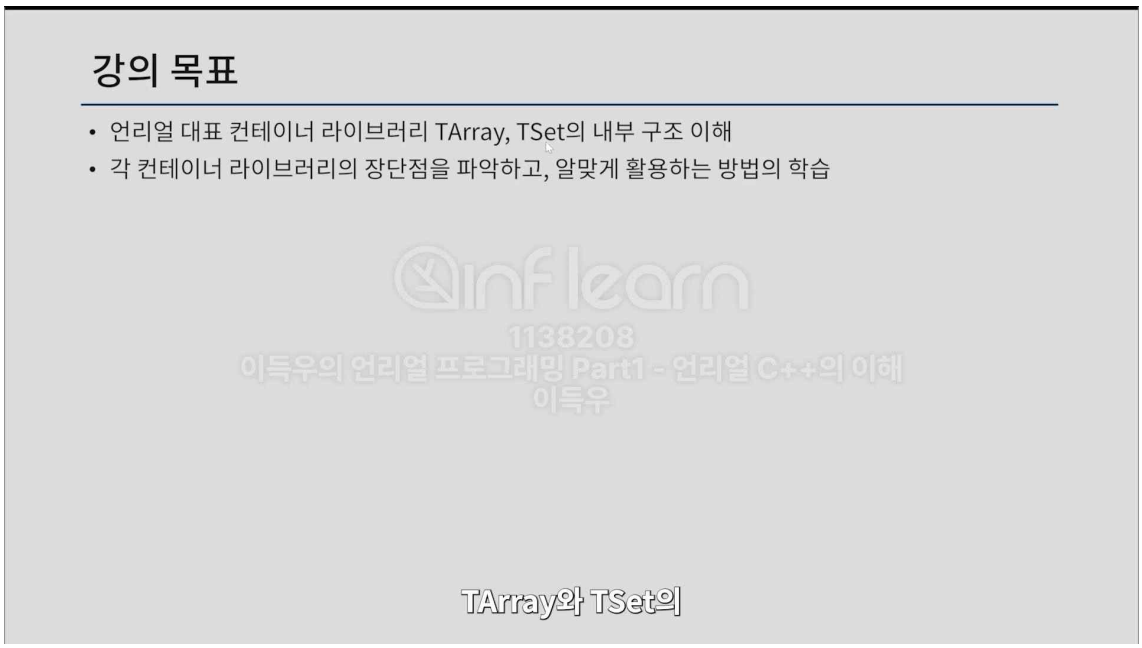
언리얼 프로그래밍 Part1-10

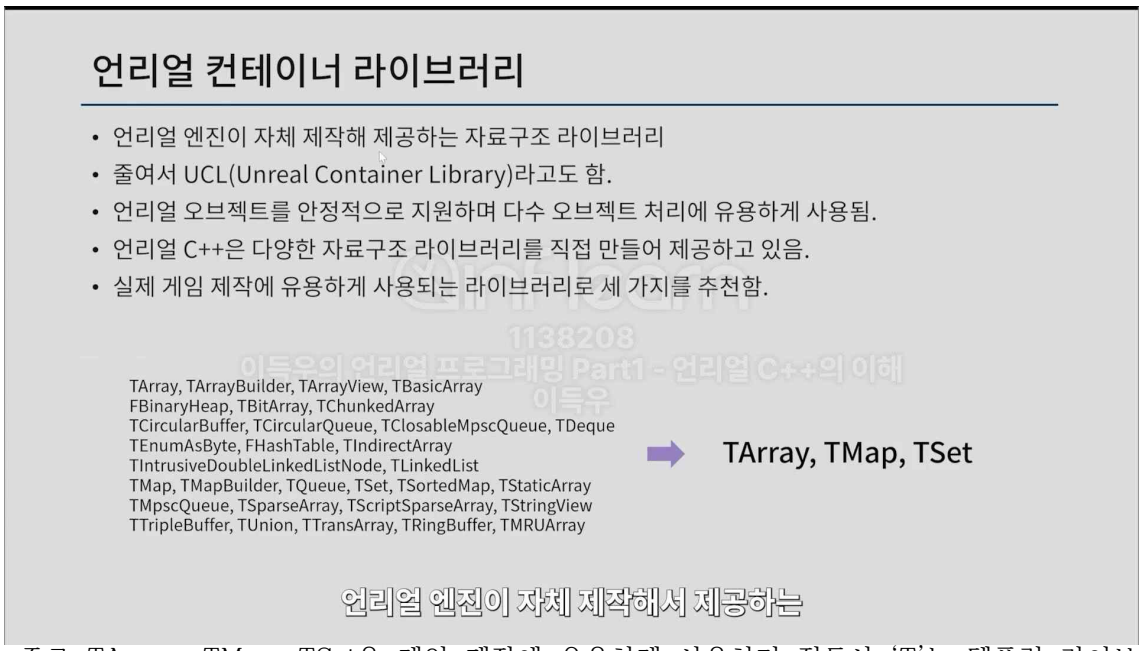
제목:언리얼 컨테이너 라이브러리 I - Array와 Set

**강의 내용 : 언리얼에서 제공하는 대표 컨테이너 라이브러리의 동작 원리와 활용 방법을 예제를 통해 살펴보기

**강의 목표

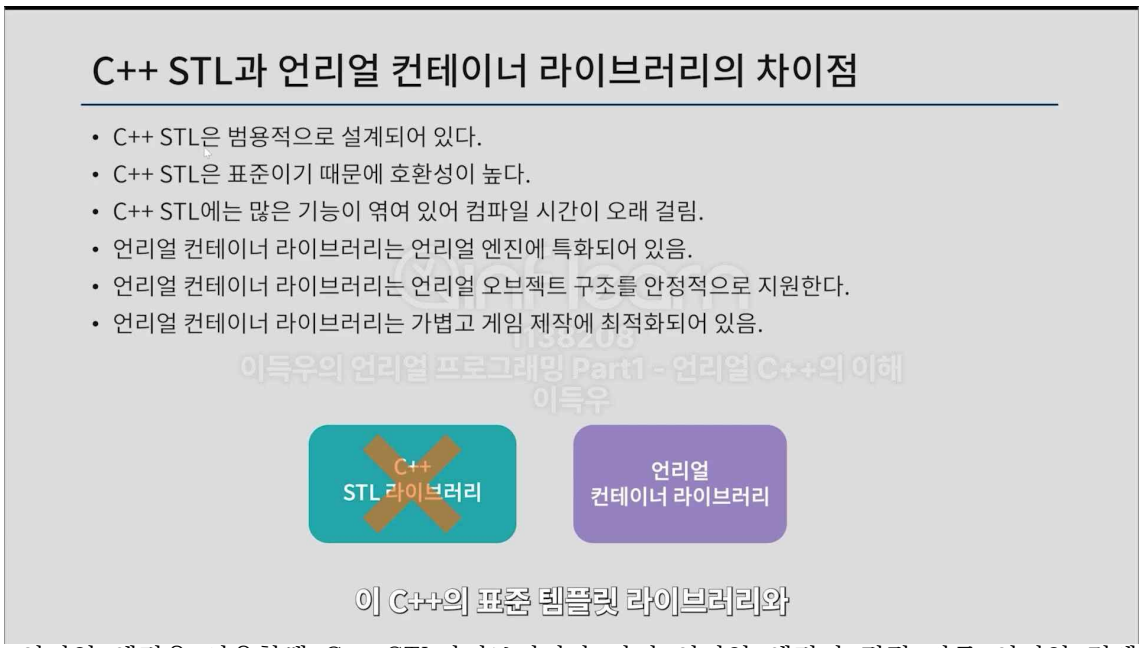


**언리얼 컨네티어 라이브러리



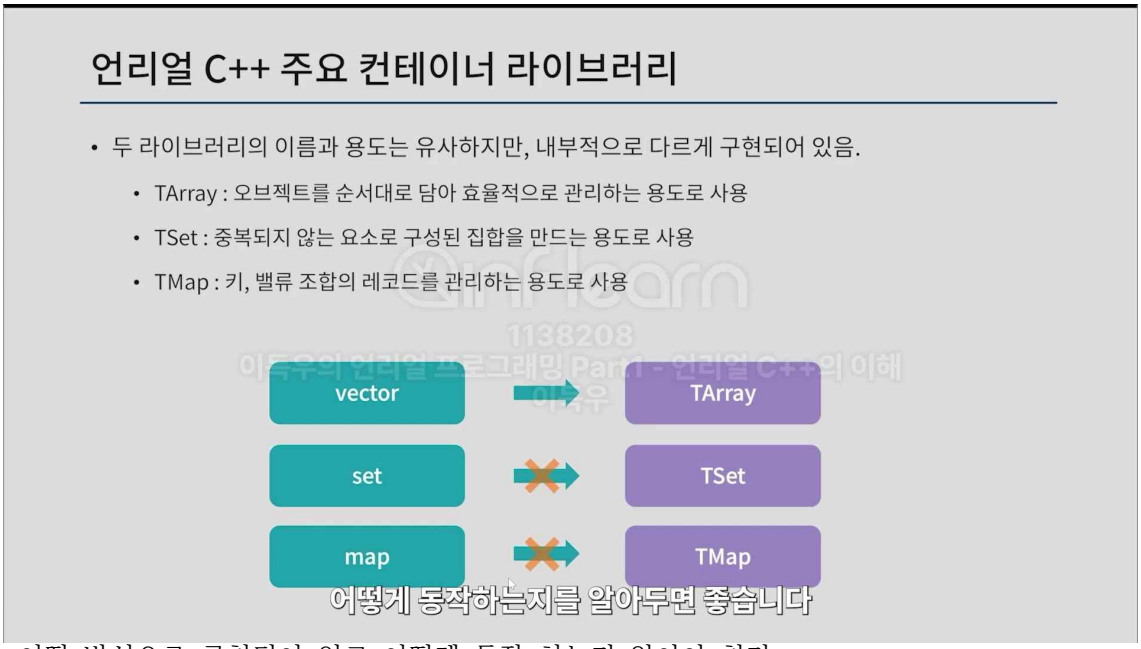
-주로 TArray, TMap, TSet을 게임 제작에 유용하게 사용하며 접두사 'T'는 템플릿 라이브러리를 의미한다.

**C++ STL과 언리얼 컨테니어 라이브러리의 차이점



-언리얼 엔진을 사용할땐 C++ STL라이브러리가 아닌 언리얼 엔진이 직접 만든 언리얼 컨테이너 라이브 러리를 사용해야 한다.

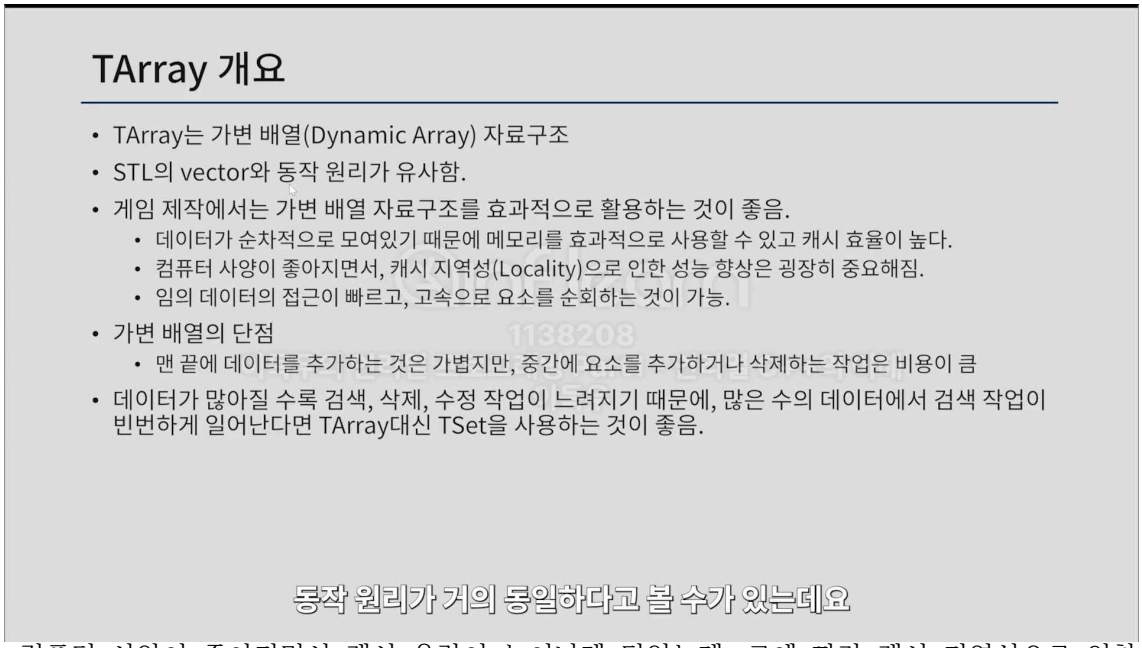
**언리얼 C++ 주요 컨테이너 라이브러리



-어떤 방식으로 구현되어 있고 어떻게 동작 하는지 알아야 한다.

TArray의 구조와 활용

**TArray개요



-컴퓨터 사양이 좋아지면서 캐시 용량이 늘어나게 되었는데, 그에 따라 캐시 지역성으로 인한 성능 향상은 매우 중요해졌다. 즉 데이터가 모여 있는 자료를 설계 하는 것이 중요하다.

**TArray의 내부 구조

-보여지는 것처럼 같은 규격을 가진 데이터들이 빈틈 없이 배열 되어 있다.

-배열의 시작 지점을 GetData()로 끝에 추가하는 함수로 Add/Emplace(), append()가 있다. 이러한 함수들은 그냥 바로 추가해줄 수 있다.

-중간에 추가하는 Insert()나 Remove함수의 경우 전체적인 메모리를 변경해야해서 비용이 많이 발생한다

-Index Operator ==> []와 같이 특정한 인덱스를 주어졌을 때 해당 인덱스를 빠르게 가져오는 작업은 균일한 데이터로 배열이 되어 있기 때문에 바로 주소를 알 수 있어서 한번에 가져올 수가 있다.

**TArray에 대한 공식 문서
-링크 : https://dev.epicgames.com/documentation/ko-kr/unreal-engine/array-containers-in-unreal-engine?application_version=5.1

-언리얼 엔진의 가장 간단한 컨테이너 클래스이며 동적 배열 자료구조로 TArray가 있다.
-항상 동일한 타입을 가져야 한다.

-TArray는 시퀀스는 연결되어 있기 때문에 잘 정의된 순서를 갖고 그 함수를 사용하여 해당 오브젝트와 순서를 결정론적으로 조작한다. 즉 같은 유형을 가지기 때문에 몇 번째의 요소를 가지고 올 경우 바로 해당 주소를 가져올 수 있다는 뜻이다.

-TArray는 가장 자주 쓰이는 컨테이너 클래스이다. 즉 신속성, 메모리 효율성, 안정성을 염두고 디자인 되었다

-TArray는 엘리먼트 유형[타입], 얼로케이터[메모리를 어떻게 관리 하는지에 대해 지정 하는 것] 총 두가지 프로퍼티로 정의되며 이중에서 얼로케이터는 일반 적으로 잘 사용하지 않는다.

-얼로케이터는 꽤 자주 생략 되는데 기본값은 1로 대부분 기본값을 사용하면 됨 근데 내가 특별히 메모리를 관리하고 싶으면 직접 작성해서 추가할 수 있다.

-TArray는 값 유형으로 동적 할당 하지 않는다. New나 Delete로 생성하거나 소멸시키는건 좋지 않고 자연스럽게 클래스 멤버 변수나 스택에서 소멸 시키면 된다. TArray소멸은 곧 거기 들어있는 엘리먼트의 소멸로 이어지게 된다.

-TArray는 힙기반의 얼로케이터를 사용한다. 즉 선언만 하면 할당된 메모리가 없게 된다.
==>TArray<int32> IntArray;

-요소는 Add나 Emplace함수로 끝에서부터 채울 수 있다.
```
==>     TArray<FString> StrArr;
        StrArr.Add (TEXT("Hello"));
        StrArr.Emplace(TEXT("World"));
        // StrArr == ["Hello","World"]
```

-Emplace의 경우 TArray자체에서 생성하기 때문에 좀더 효율적이다.

-Emplace가 Add보다 좋은 점은 복사하는 불필요한 절차를 피할 수 있다. 따라서 사소한 유형은 ADD 그 외에는 Emplace를 사용하자. (Add API는 가독성이 좋다.)

-구조체 같은 경우 Emplace를 고려하자.

-Append 함수는 한번에 다수의 개체를 집어 넣을 때 사용된다.

-AddUnique 는 검색이 들어가서 존재하지 않는 새 엘리먼트만 추가한다. 근데 이후에 나올 Set이 더 유용하기 때문에 효과적이진 않다.

-Insert같은 경우 주어진 인덱스에 추가하는 거기 때문에 전체적인 메모리 구조가 바뀐다 ==>비용이 크다.

-언리얼 컨테이너 라이브러리의 경우 컨테이너가 가지고 있는 요소의 수를 가져올땐 Count가 아닌 Num함수를 사용한다. 따라서 Num이라는 문자를 통해서 몇 개를 가지고 있는지 내가 몇 개를 설정할지에 대해서 지정할 수 있다. ==> SetNum을 사용하면 배열의 크기를 늘렸다가 줄였다가 할 수 있다.

-반복처리의 경우 Ranged for구문을 통해서 간편하게 순회할 수 있다. (일반 for문도 사용 가능)
==>

```
FString JoinedStr;
for (auto& Str : StrArr)
{
	JoinedStr += Str;
	JoinedStr += TEXT(" ");
}
```

-반복 처리에 대해서 읽기-쓰기 인지 읽기 전용인지에 대해서 즉 CreateIterator인지 아니면 CreateConstIterator를 사용할지 지정할 수도 있다.

-ranged for 구문에서도 앞에다가 const를 써주면 읽기 전용 순회가 된다.

-여러가지 Sort 옵션을 제공한다. (Sort, HeapSort, StableSort(병합 소트)) 등을 제공한다.

-쿼리를 제공한다
(1)Num함수를 통해 몇 개 있는지 알 수 있다.
(2)GetData를 통해 배열네 엘리먼트에 대한 포인터 반환이 가능하다.
(3)GetData를 통해 첫 번째 인자의 포인터를 얻어오면 Index Operator를 통해 개별 요소에 접근 할 수 있다.
(4)컨테이너가 const인 반환 데이터도 const이며 읽기 전용으로만 읽을 수가 있다.
(5)각각 엘리먼트의 사이즈를 물어볼 수 있다.
(6)Index Operator를 통해 원하는 엘리먼트를 바로 접근이 가능하다.
(7)유효 하지 않은 인덱스를 물어볼수 있기 때문에 Num을 통해 체크 하거나 IsValidIndex함수를 통해 물어볼 수도 있다.
(8)Operator의 경우 레퍼런스를 반환하는데 TArray가 const가 아닌 경우 Index Operator를 이용하여 배열 내 엘리먼트를 바로 바꿀 수가 있다.==> 수정이 가능하다
(9) Top과 Last를 통해 거꾸로 접근도 가능하다

(10) 특정 엘리먼트가 들어가 있는지 확인하는 Contains 함수도 있다.
(11) Find함수를 통해 우리가 지정한 조건에 맞는 엘리먼트가 있는지 물어볼 수 있다. 하지만 모든 요소를 순회 할 수도 있기 때문에 효율이 좋진 않다. 그리고 인덱스를 찾이 못했을 경우에는 특수 값인 INDEX_NONE이 반환된다.
(12) 검색 기능의 경우엔 빈번하게 빠르게 찾기 위해서는 이후에 배울 Set 함수를 사용해야 한다.

-제거
(1) RemoveAt을 통해 특정 요소를 제거할 수도 있고 RemoveAll을 통해 조건에 맞는 모든 요소를 지울 수도 있다.
(2)지우기의 경우 데이터가 재정렬 되기 때문에 빈틈이 없으며, 데이터 정리 과정에서 비용이 따른다.
(3)데이터가 어떤 순서로 남아있든 신경쓰이는게 아니면 교체 함으로써 비용을 줄일 순 있지만 꼭 지워야 하는 경우 지우기 함수들을 고려해보자.
(4) Empty 함수는 모든 것을 제거한다.

-연산자
(1) 할당 연산자를 통해서 우리가 값을 넣을 수가 있는데 이때 복사를 하기 때문에 Emplace 함수가 아닌 Add 함수처럼 동작한다.
(2) Append 함수 대신에 Operator 사용해서 바꿀 수 있다.
(3)MoveTemp를 사용해서 원래 자료구조가 가진 데이터를 이주 시키는 것도 가능하다.
(4) 두 배열이 동등한 경우==이퀄리티(equality)가 똑같은 경우에는 엘리먼트 수와 순서가 같을 경우만 이퀄리티 오퍼레이터가 true로 판정 된다.[FString의 비교 연산자가 대소문자를 구분 하지는 않는다 즉 대소문자만 다른 경우 같다고 판정 됨, 순번이 다른 경우에는 동등하지 않다고 판정함.]

-힙
(1) 배열을 사용할 때 이진 힙 구조를 효과적으로 관리하는 용도로도 사용할 수가 있다.

-슬랙
(1) 메모리를 확보해야 배열을 만들 수가 있는데 요소를 추가할 때마다 메모리를 하나씩 늘리게 되면 메모리 확보에 대한 오버헤드가 발생하기 때문에 보통 요청한 것보다 넉넉한 메모리를 확보해서 데이터를 넣는 방식으로 진행 한다. 따라서 엘리먼트를 삭제 한다고 메모리가 바로 해지되지는 않된다.
(2) 슬랙 == 여유분
(3) 이것이 계속해서 남아 있으면 불필요한 낭비이기 때문에 이것 조차 다 제거해서 깔끔하게 정리할 수 있다. 정리방법은 사이트 참고

-원시 메모리
(1) TArray는 궁극적으로 메모리를 둘러싼 포장 단위일 뿐이다.
(2) AddUninitialized 초기화 자체를 안 하기 때문에 빠르게 메모리만 딱 할당이 가능하다.
(3) 이러한 배열 메모리가 있을 때 memcpy를 통해 바로 복사해서 TArray를 생성할 수도 있다.
(4)이런 API들은 GetData를 통해 포인터를 얻어오고 c스타일로 메모리를 접근하여 데이터를 다룰 수도 있다. (내용은 사이트 참고)

-기타
(1) 마지막엔 TArray를 저장할 때 나오는 다양한 옵션들을 설명해준다. == 사이트 참고

**실습
-디버깅
디버깅을 위해서 여러가지 모드를 제공하지만 그냥 F5 하고 실행하면 실행 흐름은 추적이 가능하지만 내부 내용까지 추적이 가능한상세한 모드로 디버깅이 되지 않는다 따라서 빌드 모드를 바꿔주어야 한다. 따라서 상단의 Development Editor를 DebugGame Editor로 바꿔야 한다.

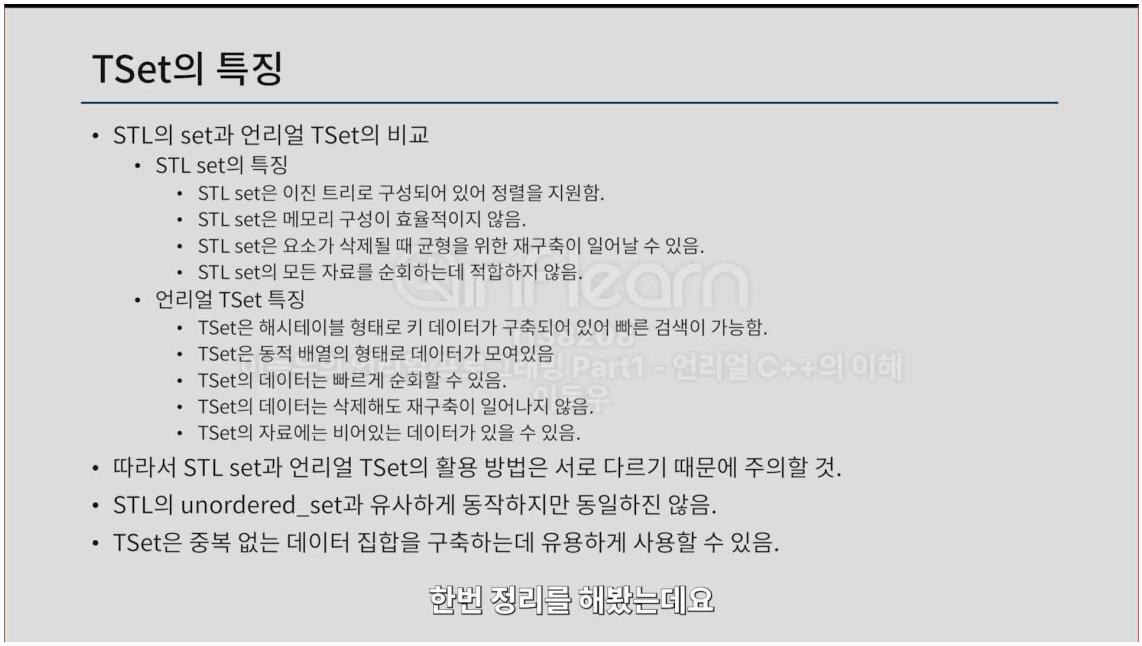
-언리얼 엔진은 여러 가지 알고리즘 라이브러리를 언리얼엔진 컨테이너에 맞게 제공하는데 대표적으로 합을 구하는 알고리즘이 있다.

-Accumulate 라이브러리
언리얼 엔진은 여러 가지 알고리즘 라이브러리를 언리얼엔진 컨테이너에 맞게 제공하는데 대표적으로 합을 구하는 알고리즘이 있다.

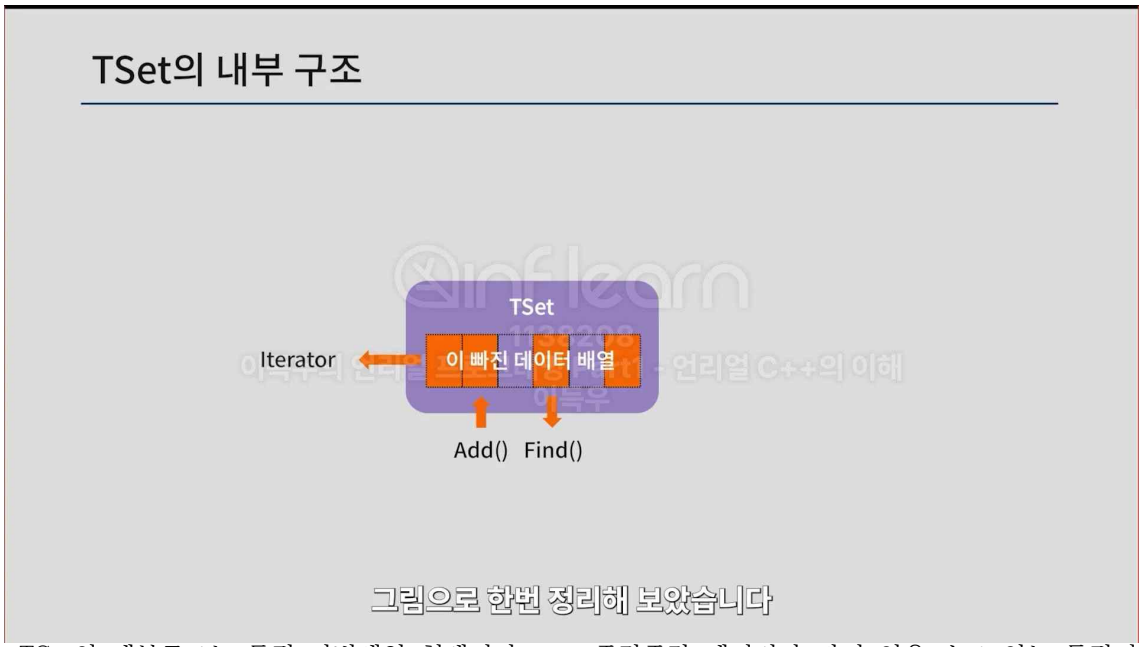
```
	// <보통  배열의 합 구하기>
	int32 Sum = 0;

	for (const int32& Int32Elem : Int32Array)
	{
		Sum += Int32Elem;
	}
	ensure(Sum == 55);

	//다음 위처럼 구문을 만들면 복잡한데 이때 accumulate 함수를 사용해보자

	int32 SumByAlgo = Algo::Accumulate(Int32Array, 0); //두번째 인자는 시작값으로 0으로 지정하자
	ensure(Sum == SumByAlgo);
}
```

TSet의 구조와 활용

**TSet의 특징



**TSet의 내부 구조



-TSet의 내부구조는 동적 가변배열 형태이다. ==> 중간중간 데이터가 빠져 있을 수도 있는 특징이 있다.
-내부적으로는 해시테이블이라서 빠른 검색이 가능하다.[Finde()]
-데이터를 추가할 때 비어있는 부분을 매꾸는 형태로 추가할 수 있다.
-중간에 데이터가 빠져 있을 지언정 데이터가 모여 있어 빠른 순회가 가능하다.

**TSet공식문서
-링크 : https://dev.epicgames.com/documentation/ko-kr/unreal-engine/set-containers-in-unreal-engine?application_version=5.1

-TSet
(1)TSet은 내부적으로 해시테이블로 구성되어 있고 데이터 자체가 키를 가지고 있 기 때문에 빠르게 검색할 수 있다. 따라서 엘리먼트 추가, 검색, 제거가 매우 빠르다.
(2)TSet은 기본적으로 중복키를 지원하지 않는다.
(3)TSet은 순서가 중요하지 않은 고유 엘리먼트를 저장하는데 사용되는 고속 컨테이너 클래스이다.
-대부분 딱 하나의 파라미터만 필요하다 하지만 TSet에 여러 가지 템플릿 파라미터로 구성하여 작동 방식을 변경하거나 다용도로 만들 수 있다.
(4)TArray와 동일하게 데이터 저장을 위해서 커스텀 메모리 얼로케이터를 제공할 수 있다. << 고급 문법이라 넘어가자
(5)TArray와 비슷하게 TSet은 같은 타입을 가지고 있는 균일한 데이터 타입이 모여 있다.
(6)TArray와 유사하게 TSet이 소멸되면 엘리먼트도 같이 소멸되도록 구성되어 있다.
(7)TSet은 내부적으로 해시를 사용하는데 이런것들은 커스텀마이징 가능한 기능을 제공한다.
(8)메모리 관리 방식을 여러 가지로 할 수 있다.
(9)TArray와 다르게 메모리 내 순서는 신뢰성이 있거나 안정적이진 않는다. 그리고 데이터가 추가할 때 비어있는 공간이 있으면 거기로 들어간다. 뚜라서 순서가 보장되지는 않는다. 즉 세트의 데이터 구조는 희소배열(Sparse Array)라는 자료구조를 기반으로 하고 있다.
(10)엘리먼트가 제거되면 간극(데이터 사이의 구멍)이 생기게 된다. 그리고 앞으로 추가되는 엘리먼트가 그 구멍을 매꾼다.

(11)TSet의 선언 방식
TSet<FString> FruitSet;
-비어 있는 자료구조가 생성 되며 '==' 오퍼레이터를 통해 엘리먼트를 비교 할 수 있고 타입에 대한 해시 값을 가져오기 위해서는 GetTypeHash 함수가 지정이 되어있어야 한다.
- 언리얼 엔진은 기본적으로 우리가 사용하는 FString이나 integer나 기본적인 UObject에 대한 해싱값을 가지고 있지만 새로운 형태의 구조체를 키셋으로 만들고 싶다면 해당 자료형에 대한 GetTypeHash를 직접 만들어 주어야 한다.

(12) 세트를 채우는 방식은 Add함수를 통해 하나씩 집어 넣는다.
(13) 기본적으로 중복을 허용하지 않기 때문에 중복 키를 시도하면 변화가 없어진다.
(14) TArray와 마찬가지로 Emplace를 사용하여 복사 없는 데이터 추가가 가능하고 Append를 사용하여 다른 세트와 병합이 가능하다.
(15) 'UPROPERTY TSet편집'은 에디터와 연동되는 키워드들을 설명하는데 Set이 가지는 특징이라고 설명하긴 어렵다.
(16) 이터레이션을 사용하여 ranged for문을 통해 쉽게 돌릴 수 있다.
(17) CreateIterator와 CreateConstIterator를 통해 읽기 전용인지 읽기-쓰기 인지 설정 가능하다.
(18) Ranged for문에서도 앞에 const를 붙이면 읽기 전용 순회가 된다.

-TSet쿼리
(1)쿼리도 비슷한데 Num함수를 지원하며 몇 개의 자료가 있는지 파악이 가능하다.
(2)contains로 자료가 있는지 미리 확인 가능하다 하지만 TArray와 다른점은 찾는 속도가 빠르다.

(3)내부적으로는 해시테이블로 찾은 아이디는 FSetElementId라는 구조체로 저장이 되어 있다. 우리가 내부 구조를 좀더 잘 알고 있다면 FSetElementId값을 직접 지정하여 찾을 수 있다.

(4)Set에 키가 있는지 없는지 확실하지 않을 겨웅 contains로 검색 후 operator[]를 사용하여 검사할 수 있다. 하지만 이 동작을 분석하면 사실 두 번 조회 하기 때문에 중복 작업이 된다. 보통 이러한 작업은 Find() 함수를 사용하여 한번에 처리하는 것이 효과적이다.

-Find() 예시:

```
FString* PtrBanana = FruitSet.Find(TEXT("Banana"));
FString* PtrLemon = FruitSet.Find(TEXT("Lemon"));
// *PtrBanana == "Banana"
// PtrLemon == nullptr
```

(5)동적 배열로 구성되어 있기 때문에 TArray를 Array함수를 통해 바로 반환이 가능하다. TArray는 빈틈이 없기 때문에 꽉찬 데이터를 얻을 수 있다.

-제거
(1) 제거의 경우 인덱스를 붙여서 제거가 가능하지만 순서를 보장하거나 순서를 파악하여 작업하기에는 매우 어렵다. 왜냐하면 데이터를 추가할 때 어디에 들어가는지 확실하지 않기 때문이다. 우리가 내부구조를 매우 잘 안다면 데이터 추가에 대한 예측이 가능하지만 일반적으로는 그렇게 사용하지 않기 때문에 인덱스를 통해 Set를 사용하는 것은 권장하지 않는다.
(2) 엘리먼트 제거는 데이터 간극을 남긴다.

-정렬
(1) 정렬 기능도 있다.

-연산자
(1) TSet<int32, FString> NewSet = FruitSet;처럼 '='와 같은 할당 연산자를 넣게 되면 복사를 통해 사본을 갖게 된다.

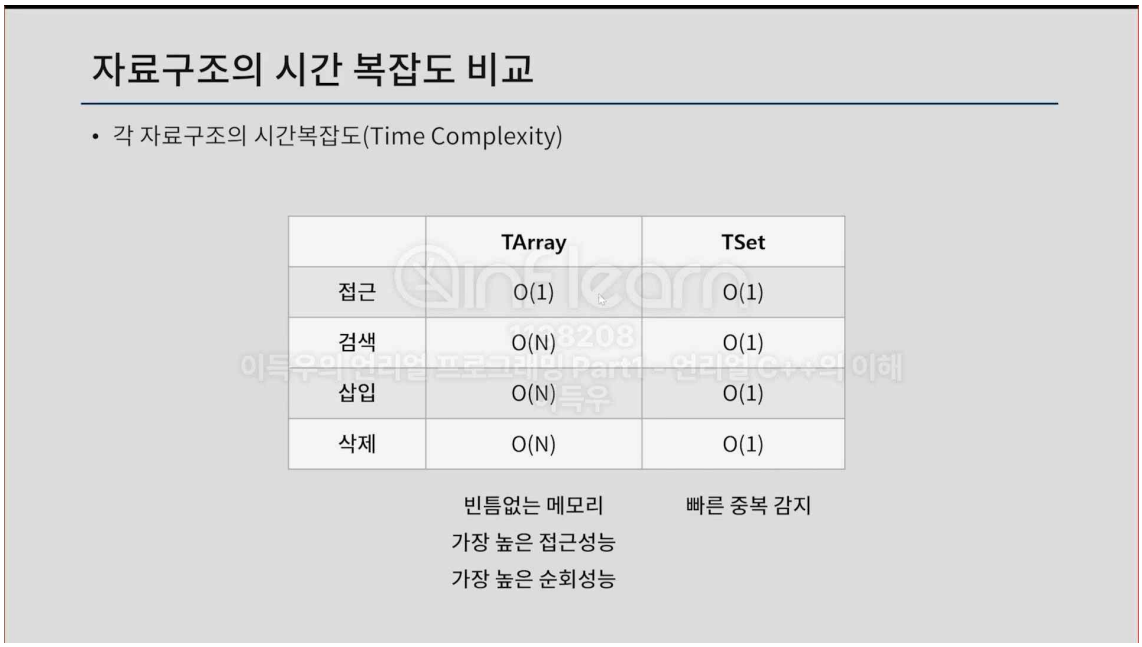
-슬랙
(1) 슬랙의 경우 TArray와 유사하지만 기본적으로 데이터를 제거 할 때 메모리를 완전히 제거가 아닌 비어 있는 상태로 놔두기 때문에 Invalid로 표시를 해준다. 따라서 데이터 뒷부분 또는 중간 중간에 생기는 슬랙이 있다.따라서 Shrink함수의 경우 뒤쪽에 생긴 슬랙을 제거해 준다.

-DefaultKetFuncs
(1) 커스텀 구조체를 통해 TSet을 만드는 경우에 'operator=='와 해당 타입에 대한 'GetTypeHash'함수를 직접 구현해주어야 한다. [이후 강좌에서 직접 해볼거임]
(2) 이러한 함수를 오버라이드하지 않는 경우에는 특이한 경우이기 때문에 별도의 함수를 제공해주어야 한다 혹시라도 Set에 대해 커스텀마이징을 해볼려면 이쪽을 파도록 하자. (강의에선 안함)

-기타
(1)CountBytes 및 GetAllocatedSize 함수를 통해 Set의 크기를 메모리 양을 측정 가능하다.

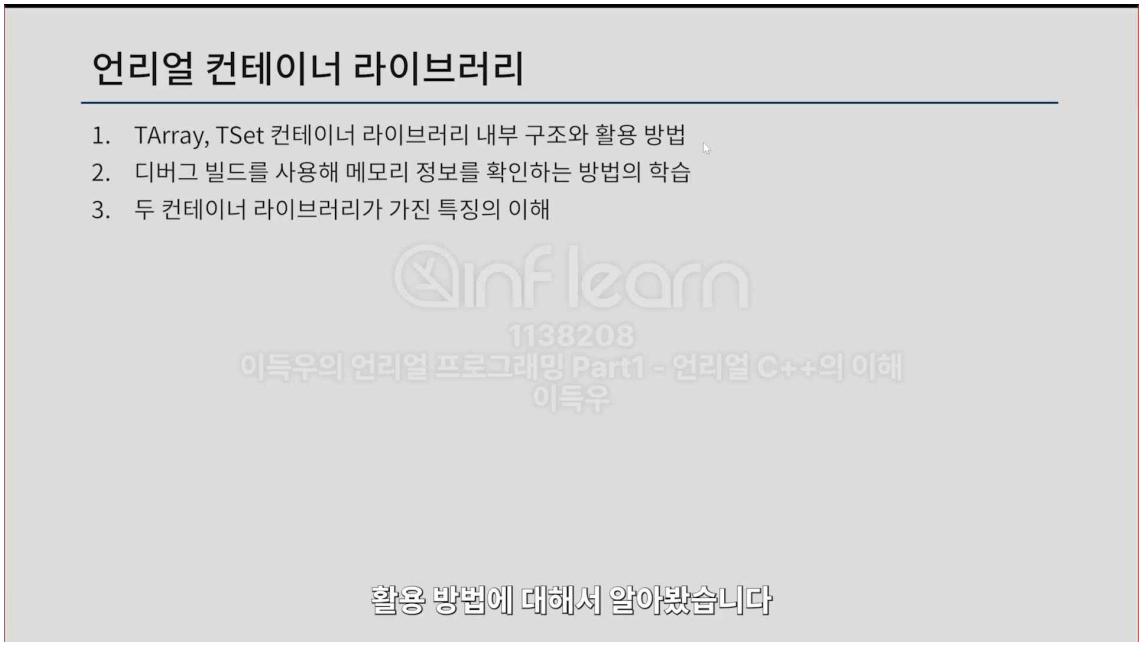
**실습
-TArray에서 10까지 넣고 2,4,6,8,10을 지운다면 다시 추가하면 1, 3, 5, 7, 9, 2, 4, 6, 8, 10 이렇게 저장되었다.
-TSet에서도 똑같이 하게된다면 지울 때 메모리가 사라지는게 아니라 Invalid로 표시된다 즉 메모리는 10개를 차지한다.
-Invalid가 있는 배열에 데이터를 추가할 때 가장 마지막에 빠진 요소에 대해서 빈칸을 채워 넣는 방식으로 데이터가 추가가 된다. 따라서 1, 10, 3, 8, 5, 6, 7, 4, 9, 2 이렇게 된다.
-따라서 Set의 경우 데이터가 어디에 들어갈지 예측이 복잡하기 때문에 순번을 생각하지 않고 그냥 무작위로 섞여 있는 집합처럼 다루는 것이 좋다.

**자료구조 시간 복잡도 비교



-TArray는 빈틈 없는 메모리를 제공하고 가장 높은 접근성능과 가장 높은 순회성능을 제공한다. 하지만 검색, 삽입, 삭제의 경우 최악일 때 개수 만큼 느려진다.
TSet의 경우 모든 경우에서 빠른 효율을 제공하고 빠른 중복을 감지하는데 유용하게 사용된다. 하지만 메모리의 형태에 빈틈이 존재할 수 있기 때문에 빈틈 없이 빠르게 접근할 땐 TSet보단 TArray가 더 좋은 선택이 될 수 있다.

**정리


언리얼 컨테이너 라이브러리
1. TArray, TSet 컨테이너 라이브러리 내부 구조와 활용 방법
2. 디버그 빌드를 사용해 메모리 정보를 확인하는 방법의 학습
3. 두 컨테이너 라이브러리가 가진 특징의 이해
inflearn
1138208
이득우의 언리얼 프로그래밍 Part1 - 언리얼 C++의 이해
이득우
활용 방법에 대해서 알아봤습니다