


1단계_03. 언리얼 C++ 기본타입/문자열

📅 수강일	@2023/03/19
👤 이름	 전영재
🔍 멘토	Min-Kang Song, 현웅 최
👥 멘티	
💡 작성 상태	In progress
🕒 단계	
☑ 강의 시청 여부	<input checked="" type="checkbox"/>
☑ 이수 여부	<input type="checkbox"/>

Contents



[언리얼 C++ 기본 타입/문자열](#)

[\[1\] 언리얼 C++ 기본 타입](#)

[\[2\] 캐릭터 인코딩](#)

[TCHAR와 FString](#)

[FString의 구조와 활용](#)

[FName의 활용](#)

[On. '언리얼 C++ 기본타입/문자열' 강의 과제](#)

[빅엔디언](#)

[리틀엔디언](#)

언리얼 C++ 기본 타입/문자열



강의 목표

- 언리얼 환경에서 알아두어야 할 기본 타입과 고려할 점
- 캐릭터 인코딩 시스템에 대한 이해
- 언리얼 C++이 제공하는 문자열 처리 방법과 내부 구성의 이해

[1] 언리얼 C++ 기본 타입

언리얼엔진은 기존 C++과는 다른 기본타입을 사용한다.

각 플랫폼마다 기본타입을 다르게 해석하는 경우가 있기에, 데이터 정보가 명확해야 하는 게임 제작의 특성 상 취약할 수 밖에 없었고 이것을 보완하기 위해 언리얼은 자신만의 데이터 타입을 사용하게 되었다.

ex) `int` 가 아닌 `int32` 를 사용한다.

포터블 C++ 코드

- `bool` - 부울 값(부울 크기 추정 금지). `BOOL` 은 컴파일되지 않습니다.
- `TCHAR` - character(문자) (TCHAR 크기 추정 금지)
- `uint8` - unsigned byte(부호 없는 바이트) (1바이트)
- `int8` - signed byte(부호 있는 바이트) (1바이트)
- `uint16` - unsigned 'shorts'(부호 없는 'short') (2바이트)
- `int16` - signed 'short'(부호 있는 'short')(2바이트)
- `uint32` - unsigned int(부호 없는 int) (4바이트)
- `int32` - signed int(부호 있는 int) (4바이트)
- `uint64` - unsigned 'quad word'(부호 없는 '쿼드 단어') (8바이트)
- `int64` - signed 'quad word'(부호 있는 '쿼드 단어') (8바이트)
- `float` - 단정밀도 부동 소수점(4바이트)
- `double` - 배정밀도 부동 소수점(8바이트)
- `PTRINT` - 포인터를 가질 수 있는 정수(PTRINT 크기 추정 금지)

<UE document에 명시된 각 타입과 크기>



`bool` 타입의 경우 크기가 명확하지 않으므로, 헤더에는 가급적 `bool` 대신 `uint8` 을 사용하되 BitField 오퍼레이터를 사용하자.

'일반 `uint8`'과 'bool표기를 위한 `uint8`'의 구분을 위해, b접두사를 사용한다.

[2] 캐릭터 인코딩

왜 언리얼 표준의 문자열이 따로 존재할까?

이유를 파악하기 위해서는 컴퓨터 역사를 살펴봐야 한다. 초기 컴퓨터는 서구권에서 개발되었기에 그들의 문자를 표시하는 것은 1바이트 데이터면 충분했다. 그러나 시간이 흐르면서 여러 나라의 문자를 표현할 방법이 필요했고 이를 위해 '유니코드'와 같은 문자열이 생겨났다.

문자열	보급 시기	종류
Single Byte	컴퓨터 초창기	ASCII : 7비트(128경우)로 문자를 표현 ANSI : 1바이트(8bit)로 문자를 표현
Multibyte	1990년대 초/중반 (보급기)	EUC-KR : 각기 완성된 문자에 코드를 부여 (한글을 위한 인코딩) CP949 : EUC-KR의 확장 버전, 코드를 조합하여 문자를 표현
Unicode	1990년대 후반 (정착기)	UTF-8: 가변 바이트 인코딩 방식(multibyte), 1~4바이트로 문자를 표현 UTF-16: 고정 바이트(2바이트 또는 4바이트) 인코딩 방식



정확히 Unicode는 '인코딩 방식'을 의미하는 것이 아닌, 문자와 코드의 '**매핑 방식**'을 의미한다. 그렇기 때문에 Unicode와 UTF를 동일시해서는 안 된다.

그러나 문제는 2020년대인 지금도 위 문자열들이 모두 사용되고 있다는 것이다. 앞서 강조한 대로 데이터 정보를 명확히 표현해야 하는 언리얼은 자체적으로 '**TCHAR**'이라는 고유 문자열 처리 방식을 만들어 해결했다.

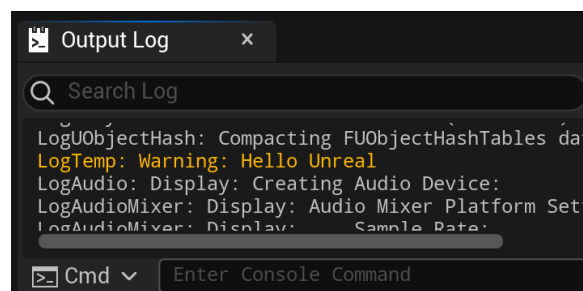
결과적으로 언리얼 엔진의 모든 스트링은 FStrings 혹은 TCHAR 정렬 상태로 UTF-16 방식으로 메모리에 저장된다. (소스코드 작성시 꼭 한글을 쓰고싶다면 UTF-8 방식을 사용하되 문제가 발생할 수 있다.)

TCHAR와 FString

언리얼은 표준 캐릭터 타입으로 **TCHAR** 를 사용한다. 그렇다면 **TCHAR** 를 사용해 간단한 로그를 출력해 보자.

```
void UMyGameInstance::Init()
{
    Super::Init();

    TCHAR LogCharArray[] = TEXT("Hello Unreal");
    UE_LOG(LogTemp, Warning, LogCharArray);
}
```

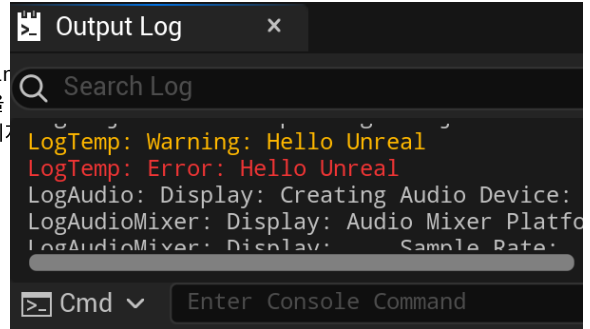


그러나 문자열을 조작하고 싶다면 **TCHAR** 보다는 **FString** 을 사용하는 것이 좋다.

```
void UMyGameInstance::Init()
{
    Super::Init();

    FString LogCharArray(TEXT("Hello Unreal"));
    UE_LOG(LogTemp, Warning, LogCharArray);
}
```

```
FString LogCharString = LogCharArray;
UE_LOG(LogTemp, Error, TEXT("%s"), *LogCharString);
//UE_LOG의 세번째 인자로는 배열만이 들어가기에 FString을
//또한 FString을 그대로 사용한다면 TCHAR 포인터가 반환되
```



FString은 TCHAR배열을 포함하는 헬퍼 클래스로 대소문자 변환, 부분 문자열 발췌, 역순 등 사용 가능한 여러 메서드를 가지고 있다. 아래는 숫자 및 기타 변수들을 스트링으로 변환하는 예시이다.

변수 유형	스트링으 변환	스트링 포맷
float	<code>FString::SanitizeFloat(FloatVariable);</code>	
int	<code>FString::FromInt(IntVariable);</code>	
bool	<code>InBool ? TEXT("true") : TEXT("false");</code>	'true' 거나 'false'
FVector	<code>VectorVariable.ToString();</code>	'X= Y= Z='
FVector2D	<code>Vector2DVariable.ToString();</code>	'X= Y='
FRotator	<code>RotatorVariable.ToString();</code>	'P= Y= R='
FLinearColor	<code>LinearColorVariable.ToString();</code>	'(R= G= B= A=)'
UObject	<code>(InObj != NULL) ? InObj ->GetName() : FString(TEXT("None"));</code>	UObject의 FName

C++에서 흔히 사용되는 `printf` 문의 경우 다음과 같이 사용할 수 있다.

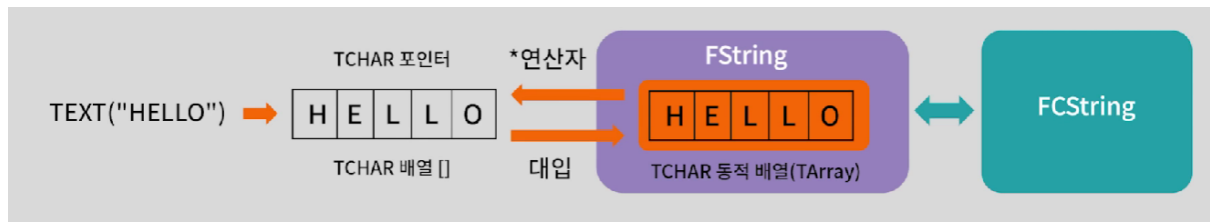
```
FString AShooterHUD::GetTimeString(float TimeSeconds)
{
    const int32 TotalSeconds = FMath::Max(0, FMath::TruncToInt(TimeSeconds) % 3600);
    const int32 NumMinutes = TotalSeconds / 60;
    const int32 NumSeconds = TotalSeconds % 60;

    const FString TimeDesc = FString::Printf(TEXT("%02d:%02d"), NumMinutes, NumSeconds);
    return TimeDesc;
}
```

또한, 언리얼 에디터 Viewport에 직접 로그를 띄우는 방법도 있으나 후에 챕터2에서 다시 다루기로 하겠다.

FString의 구조와 활용

FString의 구조에 대해서 알아보자.



TEXT매크로로 선언하게 되면 TCHAR의 배열이 생성된다. 이러한 배열을 FString에 대입하게 되는 순간, TArray라는 동적 배열 방식으로 해당 데이터를 보관한다. 동적 배열에 담긴 데이터를 사용하기 위해선 dereferencing을 해야하고 앞서 UE_LOG 출력 시에 포인터연산자(*)를 사용한 것이 이 부분이다.

나아가 FString의 내부는 FString 클래스로 구성되어, 실제 FString 헬퍼 함수들의 처리는 FString 클래스를 사용하여 처리된다.

```
FString LogCharString = TEXT("Hello Unreal");
TCHAR LogCharArrayWithSize[100];
FString::Strcpy(LogCharArrayWithSize, LogCharString.Len(), *LogCharString);
```

FName의 활용

- FName : 애셋 관리를 위해 사용되는 **초경량** 문자열 체계
 - 대소문자 구분 없음
 - 한번 선언되면 바꿀 수 없음 ('키'로 만들어지기에)
 - 가볍고 빠름
 - 문자 표현이 아닌 애셋 키를 지정하는 용도이며, 빌드 시 **해쉬값**으로 변환



입력된 문자열을 '키'와 '밸류' 구조로 만들어 문자열 처리나 활용은 어렵지만, 그만큼 가볍고 빠르기에 탐색이나 비교 등 **애셋 관리**에 주로 이용되며 뛰어난 성능을 보인다.

- FText : **다국어 지원**을 위한 문자열 관리 체계

- 일종의 키로 작용
- 별도의 문자열 테이블 정보가 요구됨
- 게임 빌드 시 자동으로 국가별 언어로 변환됨

FName은 문자열 대신 정수를 사용하여 문자열 처리 성능을 향상시키는 데에 유용하게 사용된다.

SanitizeFloat은 실수를 문자열로 변환하기 전에 문자열에 포함될 수 있는 잘못된 값(예: NaN, Infinity)을 처리하는 함수이다. (정화광선, 더러운값을 걸러준다.)



Summary

- 언리얼의 문자열 처리
 - 유니코드를 사용해 문자열 처리
 1. 2바이트로 사이즈가 균일한 UTF-16을 사용한다.
 2. 유니코드를 위한 언리얼 표준 캐릭터 타입 : TCHAR
 - 문자열은 언제나 TEXT 매크로를 사용해 지정한다. : TEXT("Hello Unreal")
TEXT매크로로 감싼 문자열은 TCHAR배열로 지정이 된다.
- FString의 구조
 - TCHAR배열 ↔ FString이라는 동적배열(TArray) ↔ FString
- FName의 구조
 - TCHAR배열 ↔ FName (Key + Value)

0n. '언리얼 C++ 기본타입/문자열' 강의 과제



Q1. ANSI, ASCII, EUC-KR, CP949, UTF-8 BOM, UTF-8, UTF16에 대해 정리하시오.

문자열	보급 시기	종류
Single Byte	컴퓨터 초창기	ASCII : 7비트(128경우)로 문자를 표현 ANSI : 1바이트(8bit)로 문자를 표현

문자열	보급 시기	종류
Multibyte	1990년대 초/중반 (보급기)	EUC-KR : 각기 완성된 문자에 코드를 부여 (한글을 위한 인코딩) CP949 : EUC-KR의 확장 버전, 코드를 조합하여 문자를 표현
Unicode	1990년대 후반 (정착기)	UTF-8 : 가변 바이트 인코딩 방식(multibyte), 1~4바이트로 문자를 표현 UTF-8 BOM : ByteOrderMark 로 처음 3byte를 통해 인코딩 방식을 표현한다. UTF-16 : 고정 바이트(2바이트 또는 4바이트) 인코딩 방식

? Q2. Little Endian, BigEndian에 대해서 정리하시오. 언제 그리고 어떤 경우에 두 방식이 사용되는지 조사하시오.

엔디언(Endianness)은 컴퓨터 메모리 같은 1차원 공간에 여러 개의 연속된 대상을 배열하는 방법을 의미한다.

일반적으로, 엔디언은 큰 단위부터 앞에 나오는 **빅엔디언**(Big-endian)과 작은 단위부터 나오는 **리틀엔디언**(Little-endian)으로 나뉘며, 두 경우에 모두 속하지 않거나 모두 지원하는 경우를 미들엔디언(Middle-endian), 바이 엔디언(Bi-endian)이라고 한다.

빅엔디언

인간이 숫자를 읽고 쓰는 방법과 유사하여, 메모리 값을 보기 쉽고 디버깅을 용이하게 한다.

리틀엔디언

메모리 값의 하위 바이트들만을 사용할 때 별도의 계산이 필요없어서 유용하다. 예를 들어, 0x2A라는 값을 참조할때, 빅엔디언의 경우는 00 00 00 2A 으로 표현하지만 리틀 엔디언의 경우는 2A 00 00 00 으로 표현하며, 앞의 바이트 일부만 떼어 내면 하위 비트들을 바로 얻을 수 있다.



위와 같은 특징들로 인해, 네트워크에서 데이터를 전송할 때는 직관적인 Big Endian을 사용하고, 컴퓨터 시스템과 소켓 통신을 할때는 효율성을 위해 Little Endian을 많이 사용한다.

? Q3. TCHAR 배열, FString, FString의 관계와 각각의 쓰임새에 대해 자신의 생각을 정리해보시오. (그림으로 설명하면 더 좋습니다)

간단히, TCHAR는 문자를 의미하며 FString은 그러한 문자의 집합인 문자열을 의미한다.

- **TCHAR** 배열의 경우, 문자열처럼 보이는 각 문자들의 연속된 집합으로, 각 문자에 해당하는 변수를 직접적으로 가지고 있다. 예를 들어, "Hello"라는 문자열을 "Hallo"라는 문자열로 바꾸고 싶다면, 간단히 'e'에 해당하는 메모리 값에 접근해 변수를 'a'로 바꿔주면 간단히 해결된다.
- **FString**은 TCHAR 배열을 래핑한 문자열 클래스이다. 정의를 살펴보면, 입력된 TCHAR의 시작과 끝 주소, 그리고 해당 데이터의 타입을 소유하는 구조체이며 TCHAR 배열과는 다르게 단순한 문자의 집합이 아닌 주소의 상관성을 이용한다는 것을 알 수 있다. 그렇기에 문자열을 수정하고 사용하는 과정에서, 각 데이터에 직접 접근해야하는 TCHAR배열보다 더 낮은 수행 비용을 기대할 수 있을거라 예상된다.

```

FString::FString(const ANSIOHP* Str)      { UE::String::Private::ConstructFromCString(Data, Str); }
FString::FString(const WIDEHP* Str)      { UE::String::Private::ConstructFromCString(Data, Str); }
FString::FString(const UTF8HP* Str)      { UE::String::Private::ConstructFromCString(Data, Str); }
FString::FString(const UCS2HP* Str)      { UE::String::Private::ConstructFromCString(Data, Str); }
FString::FString(int32 Len, const ANSIOHP* Str) { UE::String::Private::ConstructWithLength(Data, Len, Str); }
FString::FString(int32 Len, const WIDEHP* Str) { UE::String::Private::ConstructWithLength(Data, Len, Str); }
FString::FString(int32 Len, const UTF8HP* Str) { UE::String::Private::ConstructWithLength(Data, Len, Str); }
FString::FString(int32 Len, const UCS2HP* Str) { UE::String::Private::ConstructWithLength(Data, Len, Str); }
FString::FString(const ANSIOHP* Str, int32 ExtraSlack) { UE::String::Private::ConstructWithSlack(Data, Str, ExtraSlack); }
FString::FString(const WIDEHP* Str, int32 ExtraSlack) { UE::String::Private::ConstructWithSlack(Data, Str, ExtraSlack); }
FString::FString(const UTF8HP* Str, int32 ExtraSlack) { UE::String::Private::ConstructWithSlack(Data, Str, ExtraSlack); }
FString::FString(const UCS2HP* Str, int32 ExtraSlack) { UE::String::Private::ConstructWithSlack(Data, Str, ExtraSlack); }

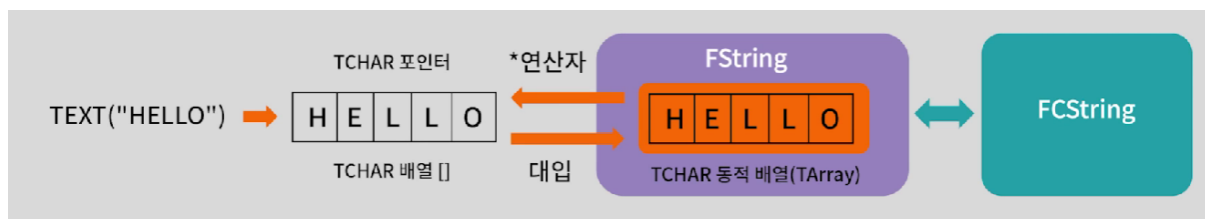
```

```

template<typename CharType>
FORCEINLINE void ConstructFromCString(/* Out */ TArray<TCHAR>& Data, const CharType* Src)
{
    if (Src && *Src)
    {
        int32 SrcLen = TQString<CharType>::Strlen(Src) + 1;
        int32 DestLen = FString::ConvertedLength<TCHAR>(Src, SrcLen);
        Data.Reserve(DestLen);
        FString::Convert(Data.GetData(), DestLen, Src, SrcLen);
    }
}

```

- **FCString**은 **FString**의 내부를 구성하는 클래스이다. FString의 문자열 처리 기능들은 FCString 클래스를 이용해 구현되며, FString은 FCString이 제공하는 기능들을 래핑해 사용자가 쉽게 헬퍼 함수를 사용 할 수 있도록 한다.



<언리얼 기본타입/문자열 강의 중 발췌>

? Q4. FName은 어떤 경우에 유용하게 쓰일지 자신의 생각을 정리해보시오.

우선 FName의 강점에 대해서 생각해보자. 해당 강의에서 FName은 애셋 관리를 위해 사용되는 초경량 문자열 체계라는 점을 강조했다. 그렇다면 실제로 다른 문자열 구조보다 얼마나 가벼울까?

검색 결과 FText는 40바이트, FString은 16바이트, FName은 8바이트로 다른 문자열 구조보다 훨씬 적은 용량을 소모한다는 것을 알 수 있었다. 또한 FName은 FString과 다르게 고유한 데이터로써 런타임 중에 변경할 수 없다는 것을 응용해, 애셋의 변경이 필요없거나 불변성을 요구하는 경우에 유용하게 사용할 수 있을 것이다.

즉, 위 두 특성을 고려한다면,

- 애셋 관리 (reference구조 관리)

- UObject의 이름 명명
- 액터의 Tag 관리



경량이라는 특성으로 인해 네트워크 통신에 사용되면 좋지 않을까 생각해봤지만, StringTable을 사용해 해쉬값을 사용하는 FName의 구조로 인해 로컬 프로세스를 넘어선 네트워크 간 통신이 생긴다면 확실성을 보장할 수 없다고 한다.



Q5. 실수를 문자열로 변환할 때 왜 FromFloat이 아닌 SanitizeFloat이라는 함수 명을 지었는지 F12키를 눌러 소스코드를 보면서 간략하게 설명하시오. (선택)



Reference

유니코드

유니코드(영어: Unicode)는 전 세계의 모든 문자를 컴퓨터에서 일관되게 표현하고 다룰 수 있도록 설계된 산업 표준이다. 유니코드는 유니코드 협회(Unicode Consortium)가 제정한다. 또한 이 표준에는 ISO 10646 문자 집합,
 w <https://ko.wikipedia.org/wiki/%EC%9C%A0%EB%8B%88%EC%BD%94%EB%93%9C>



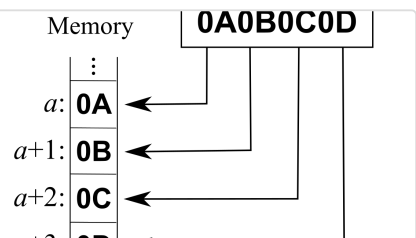
문자열 인코딩 개념 정리(ASCII/ANSI/EUC-KR/CP949/UTF-8/UNICODE)

지금껏 개발을 해오면서 ASCII와 ANSI의 차이에 대해 깊게 생각해 본 적이 없었다. UTF-8 기본으로 하여 개발을 해왔던 이유도 있거니와 ASCII=ANSI로 생각해도 사실 큰 문제는 없어왔다. 점 하나 그냥 찍어서는 안되는 개발에서 이렇게 기본기가 부족
 :: <https://onlywis.tistory.com/2>




엔디언

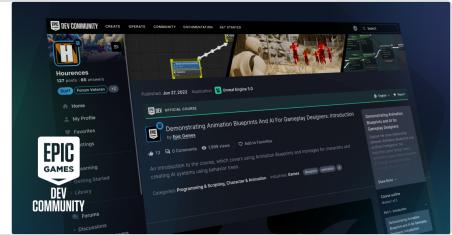
엔디언(Endianness)은 컴퓨터의 메모리와 같은 1차원의 공간에 여러 개의 연속된 대상을 배열하는 방법을 뜻하며, 바이트를 배열하는 방법을 특히 바이트 순서(Byte order)라 한다.
 w <https://ko.wikipedia.org/wiki/%EC%97%94%EB%94%94%EC%96%B8>



What is the general rule when using Text, String or Name?


Ok, so (unless this is deemed technically incorrect by UE staff) my crude rule-of thumb could be: Use 'Text' variables where localised alpha-numeric display is required Use 'Name' variables for alpha-

 <https://forums.unrealengine.com/t/what-is-the-general-rule-when-using-text-string-or-name/292348/3>



FNames under the hood

I guess this is a question for someone-who-knows at Epic. But if you're already using FNames like this and it works please let me know! If I declare FName variables in a project, and then load DataTables with

 <https://forums.unrealengine.com/t/fnames-under-the-hood/293511>

