

UE 코딩 표준

- 개요
 - Ground Rule
- 언리얼 엔진 코딩 표준
- 애셋 네이밍 컨벤션
- 코딩 일반
 - :: C++
 - auto 금지
 - 헤더 파일에서 변수 초기화 금지
 - 람다 함수 / 함수포인터 유의점
 - 언리얼 로그
 - 설명 주석
 - 코드 주석
 - 클래스 내 변수/함수 #pragma region 그룹핑
 - 논란 있던 코딩 컨벤션
 - Effective C++ 시리즈 따르기
 - 코드리뷰 체크리스트
- Visual Studio 설정
 - :: 템

개요

새로 짜는 코드의 생산성도 중요하겠으나, 프로젝트가 성장할수록 누적된 새로운 코드들로 유지보수 비용이 증가하기 때문에, 다른 사람이 짠 코드를 쉽게 해석할 수 있어야 합니다.

따라서 합의한 코딩 스타일을 가지도록 하여 코드의 가독성과 생산성을 높히는데 모두가 기여하기 위해, 코딩 표준을 정합니다.

Ground Rule

다만, 여기서 정하는 대부분의 룰들은 ground rule로, 오버데어라는 경기장에서의 느슨한 규칙입니다.

모든 룰은 다수의 생산성을 높히는 것에 목적을 두며, 코드리뷰의 근거이자 방향이 없는 분들에게 지침으로써 활용됩니다.

언제나 누구나 더 좋은 룰을 제시하실 수 있고, 근거가 타당하며, 구성원들의 동의가 있다면 바꿀 수 있습니다.

일반적으로 모든 법제화는 아래 과정을 거칩니다.

1. 소수의 사람들끼리 모여 삽
 - a. 암묵적인 룰이 생김
2. 적은 사람들이 합류하며 암묵적인 룰을 벗어나는 사람들이 생김
 - a. 기존 구성원이 말로써 암묵적인 룰을 전파함
 - b. 이 과정에서 암묵적인 룰이 와전되기도 함
3. 조금 더 많은 사람들이 합류하며 또 벗어나는 사람들이 생김
 - a. 말로 전파하는데 한계가 있고, 와전된 룰이 전파되기도 하며 혼돈이 시작됨
 - b. 가이드라인을 만들고 다 함께 지키자며 문화로써, 글로써 전파함
4. 매우 많은 사람들이 합류하며, 가이드에 관심도 없고 편함을 위해 의도적으로 지키지 않는 사람들이 생김
 - a. 더이상 가이드라인과 문화로써 룰이 존중받지 못함
 - b. 법제화하여 처벌과 보상으로 강제성을 지님
5. ~~무정부 상태로 법을 집행하는 곳이 없어졌고, 처음 그 룰을 만든 가치를 잘 아는 사람도 이제는 없음~~
 - a. 군사작전시작

이 중 우리는 3단계에 와있으며, 4단계로 가지 않으려면 모두의 노력이 필요합니다.

모두가 모든 것을 알고 빡세게 지켜야한다는 아니지만, 그 모두의 노력에는 적어도

1. 리뷰에서 우리가 만든 가이드라인에서 벗어나는 것이 보이면 의문을 가져야 하고,
2. 그 의문에 적절한 근거를 가지고 답변할 수 있어야 하며,
3. 그런 토론에서 결론지어진 방향으로 코드를 수정해야하고,
4. 토론에서 결정되지 않았다면 토론을 공론화 해야합니다.

모든 것에는 근거가 필요하고, 근거를 다시 고민하는데 시간을 쓰지 않기 위해 그라운드 룰을 둡니다.

언리얼 엔진 코딩 표준

기본적으로 [언리얼 엔진을 위한 에픽 C++ 코딩 표준](#) 문서를 따르며,

우리 프로젝트에 맞게 추가되는 내용을 이 문서에 정리.

애셋 네이밍 컨벤션

[Art 팀별 리소스 네이밍 규약](#)

[Gamemakin UE4 Style Guide](#) 에 기반함

코딩 일반

:: C++

auto 금지

- 람다 함수를 변수로 담을 때만 예외적으로 허용
- 리팩토링 시 auto 키워드 사용처는 Find All로 찾을 수 없어 코드 확인을 못함
- 동일한 변수명이 있는 struct/class 로 자동 변환 될 경우 런타임 이슈를 야기할 수 있음
- [언리얼 엔진 코딩 표준에서도 권장하지 않음](#)

헤더 파일에서 변수 초기화 금지

- 초기화값이 변경될 경우 헤더를 바꿔야하고, 인클루드하는 모든 cpp 파일을 리컴파일 함
- 헤더가 헤더를 인클루드하는 종속성 이슈로 인해 리컴파일 되는 cpp 수가 기하급수적으로 증가 할 수 있음
- 따라서 게임과 같이 코드베이스가 큰 경우 사용을 하지 않는 것이 생산성에 유리함

람다 함수 / 함수포인터 유의점

- 람다함수를 auto를 사용하여 변수로 저장함
- 꼭 필요한 경우가 아니라면 람다함수를 사용하지 않음
- 넘기는 변수의 생명 주기를 꼭 확인하고, 스코프 외부에서도 호출 될 수 있다면 TWeakObjectPtr, TWeakPtr로 매개변수 받아 사용
- 람다함수를 넘길 때 동일한 스레드에서 실행되는지 꼭 확인. 병렬처리 할거라면 스레드 세이프티를 보장해야 함.

언리얼 로그

- 무지성으로 Log category를 LogTemp로 사용하지 않는다.
 - LogTemp 카테고리는 빠르게 개발할 때만 사용하고, PR 전에는 꼭 적절한 카테고리를 만들어서 LogTemp를 모두 없앤다.
- 무지성으로 Log verbosity를 Log로 사용하지 않는다.
 - Log, Warning, Display, Error, Fatal 다섯가지 상세 수준을 상황에 맞게 붙여준다.

- 무지성으로 사용이 끝난 로그를 주석/제거 하지 않는다.
 - File IO 또는 초당 n회 등으로 많이 찍히는 로그이지만, 향후 디버깅을 위해 남겨둬야 할 때는, Log verbosity를 Verbose 또는 VeryVerbose로 바꿔서 올려둔다.
 - 동일한 빌드로 로그를 자세히 봐야 할 때 콘솔 변수로 Log Verbosity를 설정해서 새로 빌드 안뽑고 바로 분석할 수 있다.

설명 주석

- 임시처리 등으로 추가 작업이 필요한 코드의 주석은 다음과 같이 달아둔다.

```
1 // #todo_ovdr: message
```

- 코드 한 줄 한 줄의 동작을 주석하지 않는다. 코드가 가진 의도(또는 기획 의도)만을 주석으로 달아둔다.
 - 그 전에, 최대한 코드만 읽어도 의미가 파악되도록, 자연스럽게 읽히는 변수명, 함수명, 가독성을 신경써가며 코드를 작성한다.
 - 특히, 조건문에 수식을 바로 넣지 않고, bool 변수 또는 함수를 사용해 변수/함수명으로써 어떤 조건을 검사하려는지 명시한다. - [예시](#)
- 수학식을 옮겨온 경우 원본 수식 및 URL을 달아둔다.
- 순차 처리의 이해를 돋기 위한 코드는 다음과 같이 번호를 붙여 달아둔다.

```
1 // 1) 냉장고 문을 연다
2 ...
3 // 2) 쿠끼리를 냉장고에 넣는다
4 ...
5 // 3) 냉장고 문을 닫는다
6 ...
```

코드 주석

- 임시코드, 테스트코드 등 남겨둘 필요가 없는 코드는 커밋/푸시/PR 과정에서 제거한다.
- 남겨야하는 코드에 대해서 전처리기로 주석하기

```
1 #if 0 // 이 코드는 영국에서 시작되었으며...
2 ...
3 #endif
```

- 원본 코드가 보존되어야 할 때

```
1 #if 0 // 이 코드는 에픽에서 시작되었으며...
2 ...
3 #else // 이 코드는 오버데어에서...
4 ...
5 #endif
```

- on/off가 자주 일어날 때

```
1 #define _COMMENT_REASON_ 0
2 #if _COMMENT_REASON_
```

```
3 ...
4 #endif
```

클래스 내 변수/함수 #pragma region 그룹핑

[ALuaInstance 훑어보기 \(작업중\)](#) 를 따른다.

- Rule 0 : GENRATED_BODY(), friend class, 생성자 등은 pragma region으로 래핑하지 않는다.
- Rule 1 : pragma region은 최대 2단으로 사용한다.
- Rule 2 : pragma region 내부에 접근 제어 지시자를 포함시켜서 폴드했을때는 region 이름만 보이도록 한다.
- Rule 3 : region의 이름은 단어 별로 공백을 두고 첫 글자는 대문자로 한다.
- Rule 4 : 클래스 내 정의 순서는 super class overrides - class method - etc. (e.g. 로블록스 API) 순으로 정렬한다.
 - Rule 4-3 : 로블록스 API는 기타 메소드 - 이벤트 - 프로퍼티 순으로 진행한다.
- Rule 5 : 동일한 region에 메소드와 프로퍼티가 같이 들어가면 메소드를 우선한다.
- Rule 6 : 동일한 region에 메소드나 프로퍼티의 접근 제어 지시자가 여러 종류 선언 되는 경우, public - protected - private 순으로 진행한다.
- Rule 7 : 동일한 region에 메소드나 프로퍼티에 UFUNCTION(), UPROPERTY() 매크로를 선언한 필드를 그렇지 않은 경우보다 우선한다.

다만, 아래와 같은 예외를 가진다.

- 프로퍼티는 기능 단위가 아닌 하나의 region으로 묶는다.
- 처음 클래스를 만들 때 부터 모두 region으로 묶을 필요는 없으나, 가독성이 떨어지는 경우(기능 하나당 메소드가 3개 이상이 될 때) region으로 묶는다.
- 기획이 자주 바뀌는 컨텐츠 개발의 경우, region을 세세하게 나눠서 붙이지 않고, 큰 단위로 나눠서 붙인다. 기획이 결정되고 변동이 적어진 후에 세분화한다.

예시 헤더 [github link](#)

- [LuaInstance.h](#)
- [LuaBasePart.h](#)

논란 있던 코딩 컨벤션

- [읽기 좋은 코드가 좋은 코드다](#) (The Art of Readable Code) 의 [읽기 좋은 코드 철학](#)을 따름 (from [하드스킬을 같고 닦기 위한 안내서](#))
 - 코드 보고 한 번 더 생각하게 만들지 말고, 문학작품 읽듯이 자연스럽게 읽히도록 짭시다.
 - "자연스럽게 읽히도록"은 주관적이므로, 논쟁거리가 생기면 #unreal-engineer 채널에 올려 끝장토론 합시다.
 - 당장 책을 시간이 없다면 [요약 아티클](#)을 먼저 읽으시길

- 그래도 나중에 꼭 읽어보시길
- 헤더 선언부의 순서와 cpp 구현부의 순서를 맞춤
 - 코드 선언부(*.h)에 있는 순서대로 구현부(*.cpp)에 나열해 구현
 - 불필요한 위치 정보량을 줄여 코드를 찾기 쉽게 하기 위함
- 요다 컨디션은 사용하지 않음
 - ```
1 if (false == bWTF) // Bad :(
2
3 if (bWTF == false) // Good :)
```
  - `if (bWTF = false)` 와 같은 코딩 실수 때문에 썼는데, 요새는 컴파일러가 에러내줘서 안써도 됨
  - `false ==` 를 먼저 쓰면 주어가 뒤에 오므로 한 번 더 생각해야 함
- 조건문에 !(not) 연산자 사용하지 않음
  - ```
1 if (!bWTF) // Bad :(
2
3 if (bWTF == false) // Good :)
```
 - 가독성이 더 좋기도 하고, 경험상 런타임 버그 디버깅하다 보고 허탈하게 수정하는 경우가 숱함
 - `!` 을 먼저 쓰면 주어가 뒤에 오므로 한 번 더 생각해야 함
- Sandwich vs Compound inequalities
 - ```
1 if (0 < x && x < 10) // Bad :(
2
3 if (x > 0 && x < 10) // Good :)
```
  - `0 < x && x < 10` 과 같은 sandwich eq. 를 사용하는 목적은 `x && x` 를 `x` 로 치환하여 `0 < x < 10` 로 인지하는 생산성 증대가 근거였으나,
  - `0 < x && y < 10` 와 같은 조건이 있을 경우 `x && y` 를 `x` 로 치환해버려서 인지 실수를 할 수 있으므로, (조건문에서 `!`이 잘 인지되지 않는 것과 비슷)
  - `x > 0 && x < 10` 처럼 좌변에 비교하려는 변수가 오며 문장으로 치환할 수 있고 인지 실수가 발생하지 않는 compound 를 채택한다.
- guard clauses vs if statement with init
  - **guard clauses** 와 **if with init.** 둘 모두 사용하되, 아래의 룰을 따른다.
  - guard clauses는 해당 함수(또는 루프문)스코프 전역에서 사용되는 변수를 예외처리 하는데 사용한다.
    - 즉, 함수를 실행하는데 필요한 변수들의 sanity check 를 위해 사용하며, 이후에는 함수가 graceful하게 종료되도록 if with init. 등을 사용한다. - [예시](#)
  - ```
1 // Bad :(
2 AActor* AFee::Foo(UWorld* World)
```

```

3 {
4     if (IsValid(World)) // 이 메소드에서 없으면 아무것도 못하는 필수 변수
5     {
6         ...
7     }
8
9     return nullptr;
10}
11
12 // Good :
13 AActor* AFee::Foo(UWorld* World)
14 {
15     if (IsValid(World) == false) // 이 메소드에서 없으면 아무것도 못하는 필수 변수
16     {
17         return nullptr; // or continue;
18     }
19
20     ...
21 }

```

- if with init. 은 스코프 전역에서 사용되지는 않으나, 함수 내에서 여러 작업을 해야하는 경우 사용한다.

- 다만, 이럴 경우 (생산성을 해치지 않는 선에서) 새로운 함수로 빼 수 없을지 다시 한 번 고민해본다.

```

1 AActor* AFee::Foo(UWorld* World)
2 {
3     if (IsValid(World) == false) // 이 메소드에서 없으면 아무것도 못하는 필수 변수
4     {
5         return nullptr; // or continue;
6     }
7
8     AActor* FoundActor = nullptr;
9
10    // 이 로직이 이 메소드를 실행하는데 해도 되고 안해도 되는거라면 good
11    // 단, 이 메소드를 실행하는데 필수라면 위에처럼 guard clauses 사용해서 early out
12    if (ULevel* Level = World->GetCurrentLevel())
13    {
14        ...
15
16        // Bad :(
17        ALevelScriptActor* LevelScript = Level->GetLevelScriptActor();
18        if (IsValid(LevelScript))
19        {
20            return LevelScript;
21        }
22
23        // Good :)
24        // 함수 중간에 return 하게 되면, 동료가 추가한 로직이 skip 될 수 있으므로 가능한 graceful
25        if (ALevelScriptActor* LevelScript = Level->GetLevelScriptActor())
26        {
27            FoundActor = LevelScript; // good
28        }
29
30        // 또는, if with init. 을 사용한 블록 전체를 함수로 빼는 것을 고려
31
32        ...

```

```
33     }
34
35     ...
36
37     return FoundActor;
38 }
```

- 매크로 사용 시 세미콜론을 붙임

- UE_LOG(...) 등 언리얼 매크로를 쓸 때, 끝에 세미콜론이 없더라도 라이더에서는 개행할 때 문제가 없지만, VS에서는 코드 복붙이나 개행 시 아래처럼 임의로 인덴트를 추가
 - 라이더:

- ```
1 if (Soomin->GetIQLevel() == EIQLLevel::Babo)
2 {
3 UE_LOG(LogKing, Fatal, TEXT("????: That's bullshit."))
4 Soomin->GetIQLevel(EIQLLevel.Cheonjae);
5 }
```

- VS:

- ```
1 if (Soomin->GetIQLevel() == EIQLLevel::Babo)
2 {
3     UE_LOG(LogKing, Fatal, TEXT("????: That's bullshit.")) // <- 매크로 끝에 세미콜론이 없어
4         Soomin->GetIQLevel(EIQLLevel.Cheonjae); // VS에서는 코드 복붙하면 인덴트 생김
5 }
```

- UE_LOG(...) 등 매크로에 세미콜론이 포함되어있기 때문에 컴파일에는 문제가 없으나, VS 유저를 위해 매크로 사용 시 마지막에 세미콜론을 붙임.
 - 매크로를 열어보기 전까지는 세미콜론 여부를 알 수 없기 때문에, C++의 문법 상 명령줄의 끝임을 알리는 세미콜론을 붙이는 것이 이상하지는 않음.

Effective C++ 시리즈 따르기

책을 모두 읽는 것을 추천

시간이 없는 여러분들을 위해 정리본 링크 걸어둠. (그래도 책 읽어보시길)

- [Effective C++ 정리](#)
- [More Effective C++ 정리](#)
- [Effective Modern C++ 정리](#)
- Optimized C++ (정리본 없음)
- 번외 - [Effective Sandbox++ by 정수민](#)

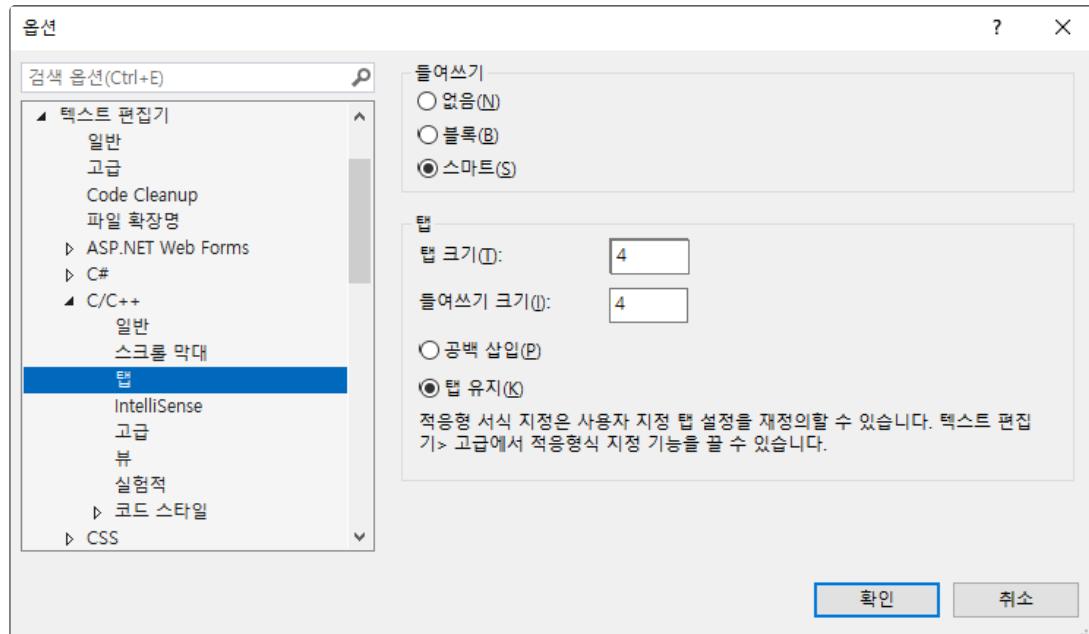
오래된 책이라 최근 트렌드/패러다임에 적용되지 않는 내용이 있을 수 있음
따르면 안될 것 같은 예외적인 내용은 따로 정리 예정

코드리뷰 체크리스트

[코드리뷰 체크리스트](#) 읽고 따르기

Visual Studio 설정

:: 템



탭을 주로 사용한 파일에서는 탭으로 통일하고, 스페이스를 주로 사용한 파일에서는 스페이스로 통일한다.

기본적으로 탭을 사용한다.