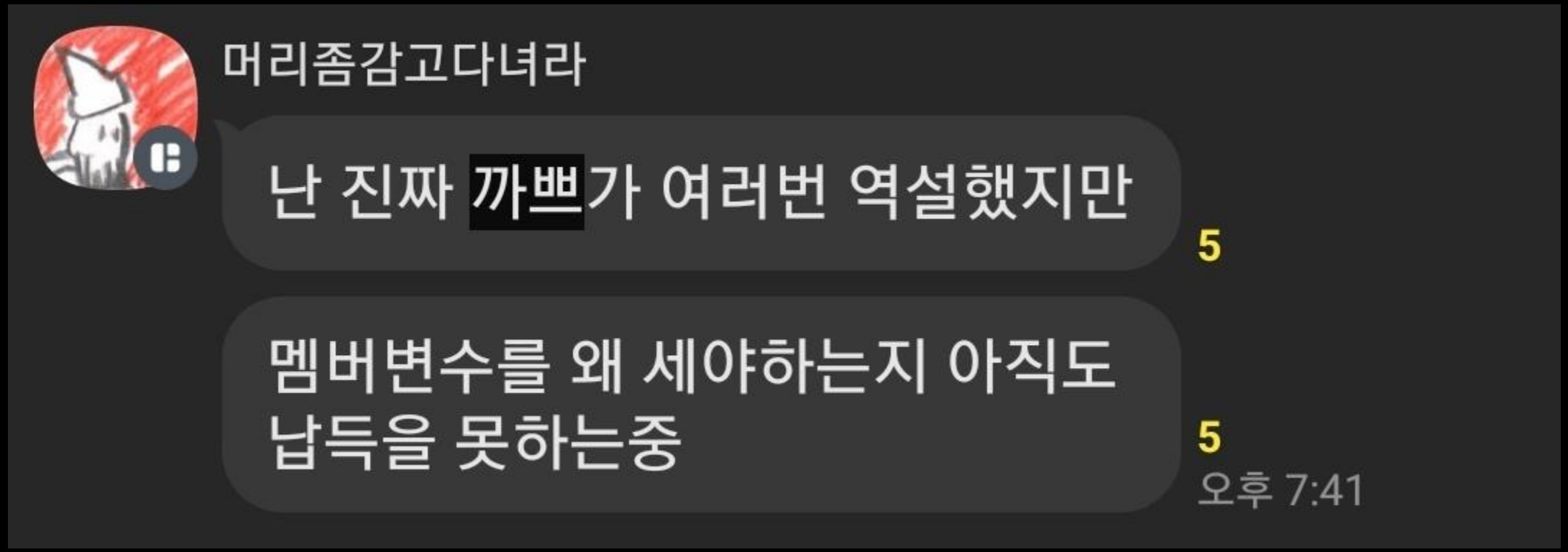
도대체 멤버변수 개수가 왜 필요한데?

- C++ 템플릿 메타프로그래밍을 이용한 정적분석-

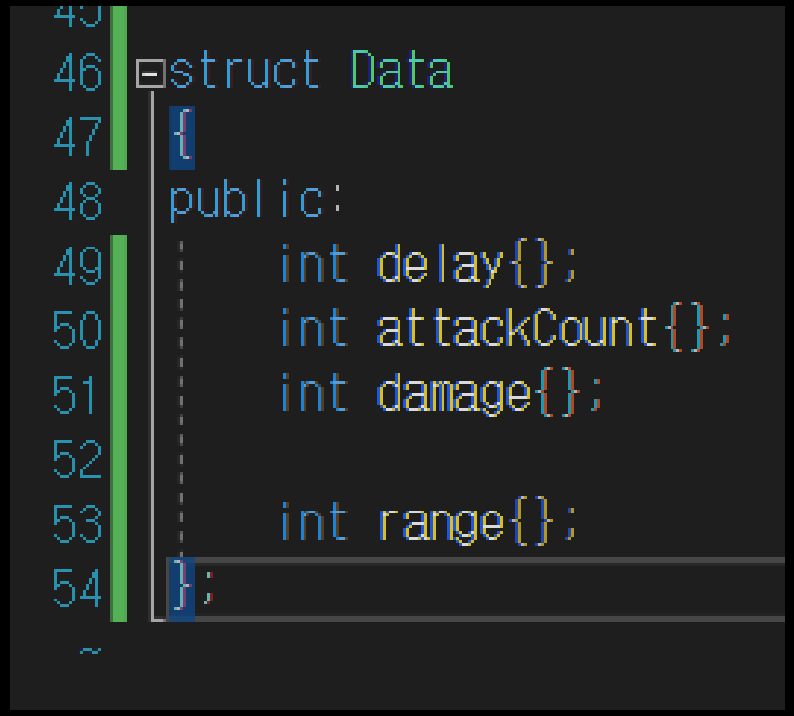
까쁘

이런 사람이 있었습니다...


머리좀감고다녀라
난 진짜 까쁘가 여러번 역설했지만
5
멤버변수를 왜 세야하는지 아직도
납득을 못하는중
5
오후 7:41

게임에는 핵이 존재한다.

- 핵이 존재하는 이유
  - 피격 판정을 클라에서 계산하기 때문
- 서버에서 하는 것
  - 데미지
- 피격 판정은 반응성을 이유로 클라이언트에서 하는 경우가 많음
- FPS에 유독 핵이 많은 이유도 이것



가장 간단하고 확실한 방법 해시값 비교

- 구조체 내에 있는 데이터를 이용하여 해시값을 만든다.

- 이를 특정 타이밍마다 서버와 비교하게 될 경우 데이터 변조는 사실상 완전히 막힘

```
struct Data
{
public:
    int delay{};
    int attackCount{};
    int damage{};

    int range{};

public:
    int CalcHash() const
    {
        int hash = 0;
        hash = GetHash( hash, delay );
        hash = GetHash( hash, attackCount );
        hash = GetHash( hash, damage );
        hash = GetHash( hash, range );
    }
};
```

아 맞다 까먹었다

변수를 추가 할 때 해시에 추가 안함

- 이러면 이 변수는 해시검사에서 제외되고 변조해도 검출 불가

```
24
25  struct Data
26  {
27  public:
28      int delay{};
29      int attackCount{};
30      int damage{};
31
32      int range{};
33      int range2{};
34
35  public:
36      int CalcHash() const
37      {
38          int hash = 0;
39          hash = GetHash( hash, delay );
40          hash = GetHash( hash, attackCount );
41          hash = GetHash( hash, damage );
42          hash = GetHash( hash, range );
43          // ???
44      }
45  };
```

멤버변수의 개수를 알 수 있다면...

- 어떻게든 기존과 달라졌는지를 체크해서 컴파일 에러를 띄울 수 있지 않을까
- 처음에는 구조체의 크기로 이 변화를 감지하고자 시도
- 구조체 패딩 문제 때문에 실패

유레카

```cpp
namespace impl
{
    struct any_type
    {
        template<typename T>
        constexpr operator T();
    };

    template <class T, class... Ts>
    decltype( void( T{ { std::declval<Ts>() }... } ), std::true_type{} ) test_is_braces_constructible( std::size_t );

    template <class, class...>
    std::false_type test_is_braces_constructible( ... );

    template<std::size_t N, class T, class... Ts>
    struct sizeof_struct
        : public std::conditional_t<decltype( test_is_braces_constructible<T, Ts...>( 0 ) )::value,
        sizeof_struct<N + 1, T, any_type, Ts...>,
        std::integral_constant<std::size_t, N>>
    {
    };
}

template<typename T, typename = std::enable_if_t<std::is_aggregate_v<T>>>
using sizeof_struct = typename impl::sizeof_struct<0, T, impl::any_type>;

template<typename T>
constexpr inline auto sizeof_struct_v = sizeof_struct<T>::value;
```


이.이게뭐노

any_type은 템플릿과 연산자 오버로딩으로 어떤 타입이든 가능하게 만든 구조체

```cpp
namespace impl
{
    struct any_type
    {
        template<typename T>
        constexpr operator T();
    };

    template <class T, class... Ts>
    decltype( void( T{ { std::declval<Ts>() }... } ), std::true_type{} ) test_is_braces_constructible( std::size_t );

    template <class, class...>
    std::false_type test_is_braces_constructible( ... );

    template<std::size_t N, class T, class... Ts>
    struct sizeof_struct
        : public std::conditional_t<decltype( test_is_braces_constructible<T, Ts...>( 0 ) )::value,
        sizeof_struct<N + 1, T, any_type, Ts...>,
        std::integral_constant<std::size_t, N>>
    {
    };
}

template<typename T, typename = std::enable_if_t<std::is_aggregate_v<T>>>
using sizeof_struct = typename impl::sizeof_struct<0, T, impl::any_type>;

template<typename T>
constexpr inline auto sizeof_struct_v = sizeof_struct<T>::value;
```

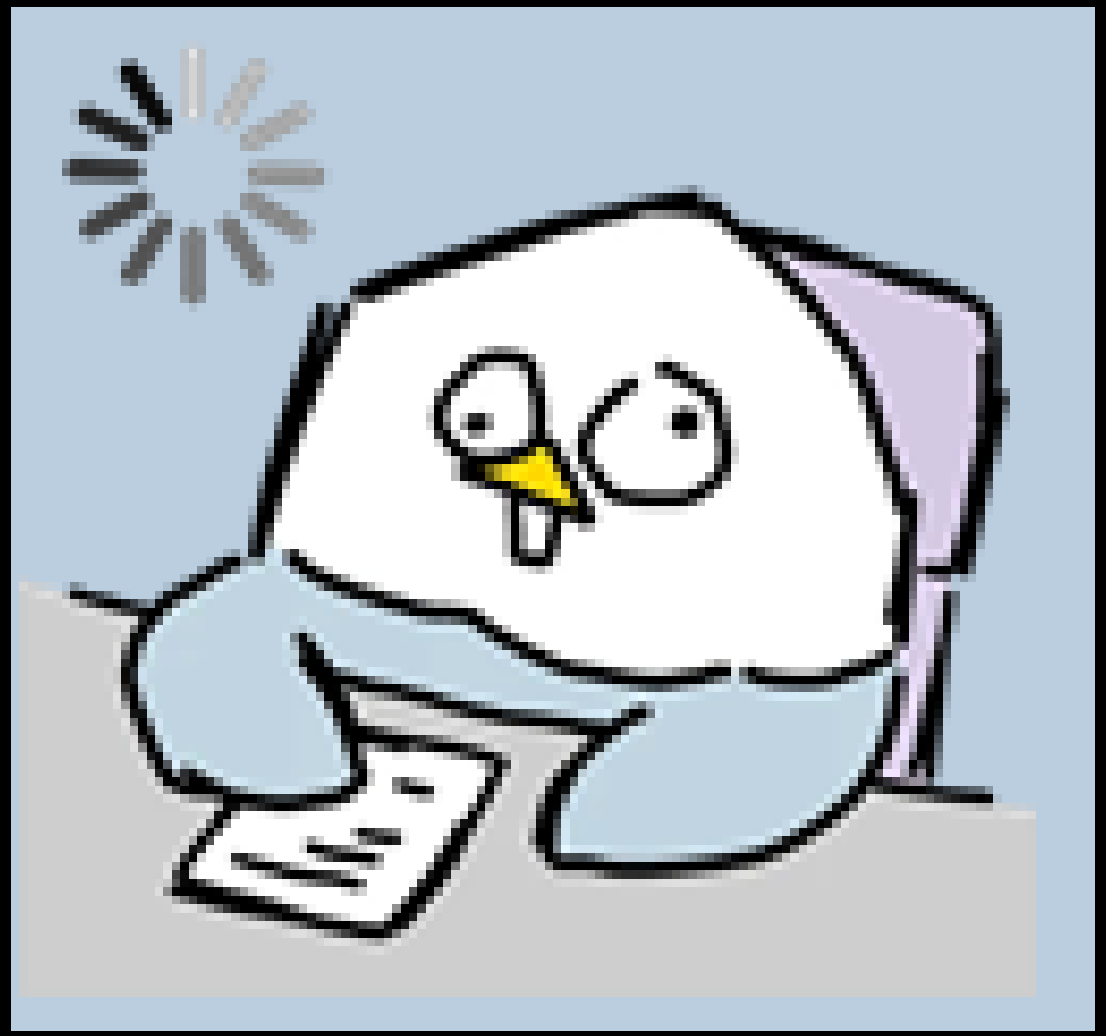
```cpp
struct any_type
{
    template<typename T>
    constexpr operator T();
};
```

```cpp
template<std::size_t N, class T, class... Ts>
struct sizeof_struct
    : public std::conditional_t<decltype( test_is_braces_constructible<T, Ts...>( 0 ) )::value,
    sizeof_struct<N + 1, T, any_type, Ts...>,
    std::integral_constant<std::size_t, N>>
{
};
```

```cpp
namespace impl
{
    struct any_type
    {
        template<typename T>
        constexpr operator T();
    };

    template <class T, class... Ts>
    decltype( void( T{ { std::declval<Ts>() }... } ), std::true_type{} ) test_is_braces_constructible( std::size_t );

    template <class, class...>
    std::false_type test_is_braces_constructible( ... );

    template<std::size_t N, class T, class... Ts>
    struct sizeof_struct
        : public std::conditional_t<decltype( test_is_braces_constructible<T, Ts...>( 0 ) )::value,
        sizeof_struct<N + 1, T, any_type, Ts...>,
        std::integral_constant<std::size_t, N>>
    {
    };
}

template<typename T, typename = std::enable_if_t<std::is_aggregate_v<T>>>
using sizeof_struct = typename impl::sizeof_struct<0, T, impl::any_type>;

template<typename T>
constexpr inline auto sizeof_struct_v = sizeof_struct<T>::value;
```

```cpp
template <class T, class... Ts>
decltype( void( T{ { std::declval<Ts>() }... } ), std::true_type{} ) test_is_braces_constructible( std::size_t );

template <class, class...>
std::false_type test_is_braces_constructible( ... );
```

```cpp
namespace impl
{
    struct any_type
    {
        template<typename T>
        constexpr operator T();
    };

    template <class T, class... Ts>
    decltype( void( T{ { std::declval<Ts>() }... } ), std::true_type{} ) test_is_braces_constructible( std::size_t );

    template <class, class...>
    std::false_type test_is_braces_constructible( ... );

    template<std::size_t N, class T, class... Ts>
    struct sizeof_struct
        : public std::conditional_t<decltype( test_is_braces_constructible<T, Ts...>( 0 ) )::value,
        sizeof_struct<N + 1, T, any_type, Ts...>,
        std::integral_constant<std::size_t, N>>
    {
    };
}

template<typename T, typename = std::enable_if_t<std::is_aggregate_v<T>>>
using sizeof_struct = typename impl::sizeof_struct<0, T, impl::any_type>;

template<typename T>
constexpr inline auto sizeof_struct_v = sizeof_struct<T>::value;
```

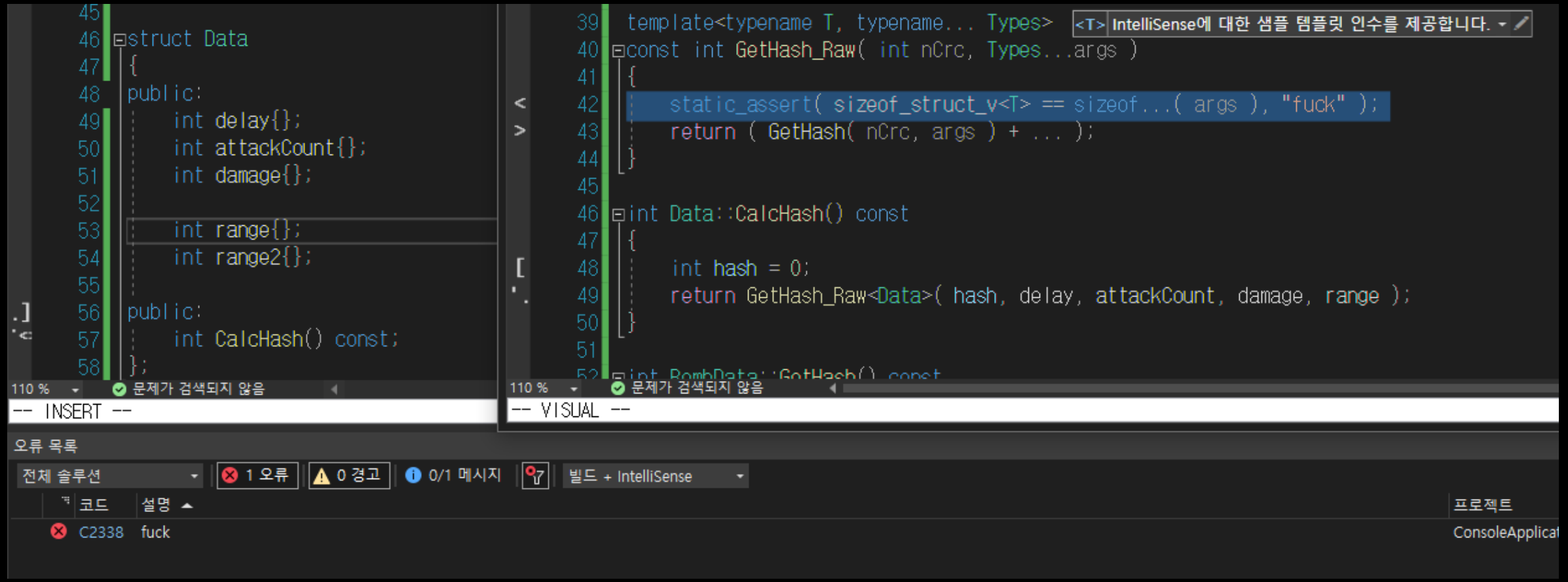


어떤 원리인가

- C++ 에는 Aggregate한 타입이라는 게 존재
- {}로 초기화 가능한 것
- 컴파일이 안될 때까지 any_type을 넣어서 체크
- 컴파일이 안될 때 아까 false를 리턴하고 std::conditional에 의해 상속개수가 저장된 std::integral_constant를 상속 받게 됨

```
59
60  Data{ impl::any_type() };                                                    // 컴파일 가능 N 1
61  Data{ impl::any_type(), impl::any_type() };                                  // 컴파일 가능 N 2
62  Data{ impl::any_type(), impl::any_type(), impl::any_type() };                // 컴파일 가능 N 3
63  Data{ impl::any_type(), impl::any_type(), impl::any_type(), impl::any_type() };  // 컴파일 가능 N 4
64  Data{ impl::any_type(), impl::any_type(), impl::any_type(), impl::any_type(), impl::any_type() };  // 컴파일 불가능가능 N 4인 상태로 std::integral_constant를 상속받고 끝
65
```

멤버변수를 추가하고 해시함수에 추가 안 하면 바로 컴파일 에러



누군가 구조체 타입의 멤버를 추가한다면

```
struct BombData
{
    int damage{};
    int range{};
    std::vector<int> preset;
};

struct Data
{
public:
    int delay{};
    int attackCount{};
    int damage{};

    int range{};
    BombData ball;

public:
    int CalcHash() const;
};
```

각 구조체에 GetHash를 추가적으로 구현

- Vector같은 애들때문에 구조체별로 따로 구현해줄 필요가 있음
- 근데 GetHash가 필요한 구조체인지 아닌지는 어떻게 구분해서 함수를 호출하지...?
- 그리고 이것도 누군가가 추가하는 걸 까먹으면 어떡하지??

아까 있었던 내용중 컴파일 에러를 이용한 오버로딩을 사용

```cpp
template<typename T, typename = std::enable_if_t<std::is_member_function_pointer< decltype( &T::GetHash ) >::value>>
int GetHash( int key, const T& t )
{
    return t.GetHash();
}

template<typename T, typename... Types>
int GetHash( int key, const T& t, const Types&... typs )
{
    static_assert( std::is_trivially_copyable_v<T>, "need GetHash" );
    static_assert( !std::is_pointer_v<T>, "fuckkkkk" );
    int value = key;
    for ( int i = 0; i < sizeof( T ); ++i )
    {
        char* ptr = ( char* )&t;
        value += ptr[ i ];
        value *= 29;
    }
    return value;
}

template<typename T>
int GetHash( int key, const std::vector<T>& vec )
{
    int value = key;
    for ( auto& t : vec )
    {
        value = GetHash( value, t );
    }
    return value;
}
```

```cpp
int Data::CalcHash() const
{
    int hash = 0;
    return GetHash_Raw<Data>( hash, delay, attackCount, damage, range, ball );
}

int main()
{
}

// 프로그램 실행: <Ctrl+F5> 또는 [디버그] > [디버깅하지 않고 시작] 메뉴
// 프로그램 디버그: <F5> 키 또는 [디버그] > [디버깅 시작] 메뉴
```

110 % 문제가 검색되지 않음

-- INSERT --

C2338 need GetHash

드디어 반드시 해시체크에 제대로 추가해야 컴파일이 되는 코드 완성

- 사람은 누구나 언제든 실수를 한다
- 이러한 실수를 미연에 방지할 수 있는 방어적 코딩을 하자
- 컴파일타임에 이런걸 분석해서 막을 수 있는 C++하실?