

Reflection

목차

- 시작하며...
- 타입 정보 생성
- 변수 정보 생성
- 함수 정보 생성
- 마치며...
- 더 자세한 코드를 보고 싶으시다면...
- 참고자료

시작하며...

이 ppt에서는 C++에서 Reflection을 어떻게 구현하였는지 다루고자 합니다.

여기서 이야기하는 'Reflection'(이하 리플렉션)은 런타임 중에 자신의 구조와 행위를 검사할 수 있는 기술을 의미하는 것으로 컴퓨터 그래픽스의 반사 기법을 의미하는 것이 아님을 미리 밝힙니다.

시작하며...

구현 방식을 자세히 살펴보기 전에 리플렉션을 통해서 무엇을 하고 싶었는지 이야기하고 싶습니다.

개인 프로젝트의 코드 정리를 하면서 게임 오브젝트의 기능 확장 방식을 상속을 통한 방식에서 언리얼 엔진과 같은 컴포넌트를 이용한 확장으로 변경하면서 게임 오브젝트가 컴포넌트의 집합을 가지고 있도록 변경하였습니다.

```
1 std::vector<Component*> m_components;
```

시작하며...

이에 따라 다양한 타입의 컴포넌트를 일괄적으로 보관하기 위해서 부모타입으로 변경(Upcasting)했고 게임 오브젝트에서 컴포넌트를 사용하기 위해서 실제타입으로 다시 변경(Downcasting)하는 과정이 필요했습니다.

실제타입으로의 변경하는 과정은 잘 못된 변경으로 인한 오류를 막기 위해서 다음과 같이 `dynamic_cast`을 사용하여 안전하게 변경되도록 처리하였습니다.

```
1 template <typename T>
2 T* GetComponent( )
3 {
4     for ( Component* component : m_components )
5     {
6         if ( auto concrete = dynamic_cast<T*>( component ) )
7         {
8             return concrete;
9         }
10    }
11
12    return nullptr;
13 }
```

시작하며...

dynamic_cast은 추가적인 런타임 비용을 지불해야 하는 캐스팅 방식으로 안전한 캐스팅을 위해 사용한 것은 어쩔 수 없는 선택이었지만 언리얼 엔진의 Cast 연산자와 같이 리플렉션을 이용하여 이 비용을 줄여보고 싶었습니다.

즉 dynamic_cast 보다 빠르면서 안전한 Downcasting을 하는 것을 목적으로 구현하였습니다.

그래서 성능상 이득은 있었을까요?

시작하며...

성능 측정에는 아래와 같은 사양의 PC를 사용하였으며

CPU : Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz

RAM : 16 GB

Visual Studio 2022 64 bit 빌드

다음과 같은 코드로 10회 측정하였습니다.

시작하며...

```
1 #include "Reflection.h"
2
3 #include <chrono>
4 #include <iostream>
5
6 class A
7 {
8     GENERATE_CLASS_TYPE_INFO( A )
9
10 public:
11     virtual ~A() = default;
12 };
13
14 class B : public A
15 {
16     GENERATE_CLASS_TYPE_INFO( B )
17 };
18
19 class C : public B
20 {
21     GENERATE_CLASS_TYPE_INFO( C )
22 };
23
24 class D : public C
25 {
26     GENERATE_CLASS_TYPE_INFO( D )
27 };
28
29 class E : public A
30 {
31     GENERATE_CLASS_TYPE_INFO( E )
32 };
```


시작하며...

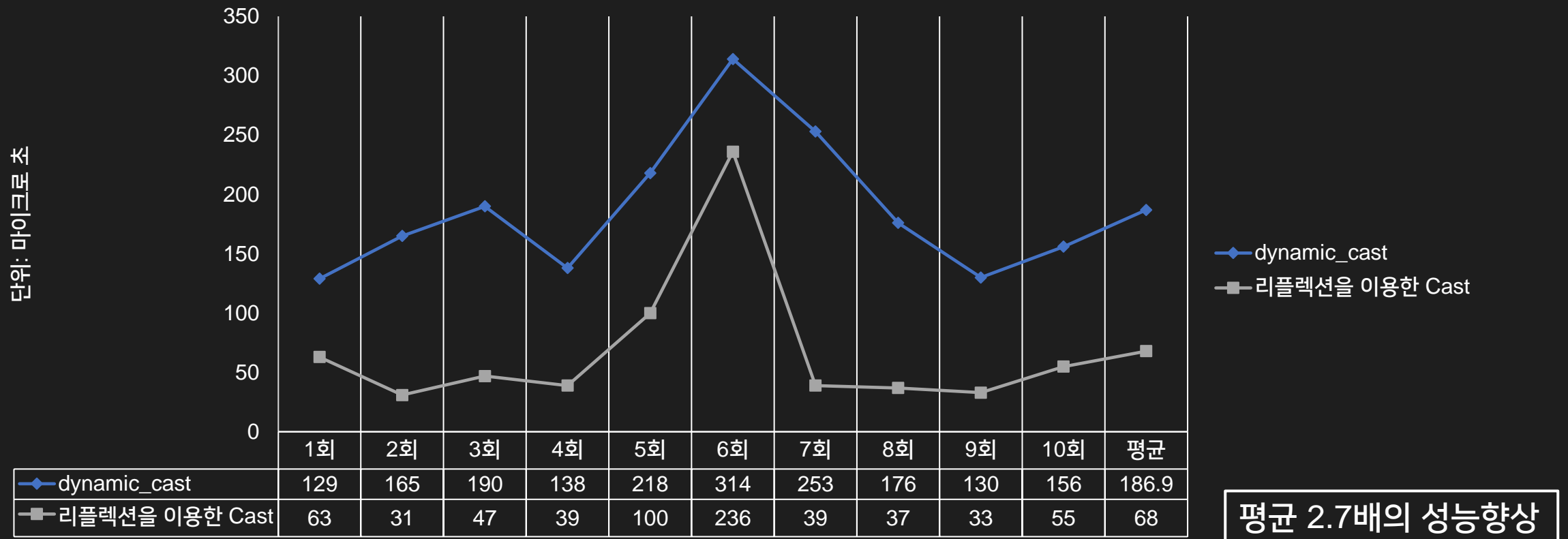
```
33
34 int main()
35 {
36     A* test[10000] = {};
37     for ( int i = 0; i < 10000; ++i )
38     {
39         if ( i % 3 == 0 )
40         {
41             test[i] = new D;
42         }
43         else
44         {
45             test[i] = new E;
46         }
47     }
48
49     auto start = std::chrono::system_clock::now();
50
51     int count1 = 0;
52     for ( int i = 0; i < 10000; ++i )
53     {
54         if ( B* b = dynamic_cast<B*>( test[i] ) )
55         {
56             ++count1;
57         }
58     }
59
60     auto end = std::chrono::system_clock::now();
61
62     auto d1 = std::chrono::duration_cast<std::chrono::microseconds>( end - start );
63     std::cout << "count : " << count1 << std::endl;
64     std::cout << d1.count() << std::endl;
65
```

시작하며...

```
66     start = std::chrono::system_clock::now();
67
68     int count2 = 0;
69     for ( int i = 0; i < 10000; ++i )
70     {
71         if ( B* b = Cast<B>( test[i] ) )
72         {
73             ++count2;
74         }
75     }
76
77     end = std::chrono::system_clock::now();
78
79     auto d2 = std::chrono::duration_cast<std::chrono::microseconds>( end - start );
80     std::cout << "count : " << count1 << std::endl;
81     std::cout << d2.count() << std::endl;
82 }
```

시작하며...

DOWNCASTING 속도 비교



타입 정보 생성

클래스의 타입 정보는 다음과 같은 매크로를 클래스 선언에 추가하는 것으로 생성됩니다.

```
1 class A
2 {
3     GENERATE_CLASS_TYPE_INFO( A )
4
5 public:
6     virtual ~A() = default;
7 };
```

GENERATE_CLASS_TYPE_INFO 매크로는 크게 2가지 역할을 하는데

1. 클래스에 자신의 타입과 부모 타입의 **별칭을 일관된 이름으로** 선언하고
2. 클래스의 타입 정보를 담고 있는 **TypeInfo 객체를 생성**합니다.

타입 정보 생성

이제부터 매크로의 내용을 보면서 어떻게 각 기능을 제공하는지 보겠습니다.
매크로는 다음과 같이 작성되었습니다.

```
#define GENERATE_CLASS_TYPE_INFO( TypeName )    \
private:    \
    friend SuperClassTypeDeduction; \
    friend TypeInfoInitializer; \
\
public: \
    using Super = typename SuperClassTypeDeduction<TypeName>::Type; \
    using ThisType = TypeName; \
\
    static TypeInfo& StaticTypeInfo()    \
    {    \
        static TypeInfo typeInfo{ TypeInfoInitializer<ThisType>( #TypeName ) }; \
        return typeInfo; \
    } \
\
    virtual const TypeInfo& GetTypeInfo() const \
    {    \
        return m_typeInfo; \
    } \
\
private: \
    inline static TypeInfo& m_typeInfo = StaticTypeInfo(); \
\
private:    \
```

타입 정보 생성

```
#define GENERATE_CLASS_TYPE_INFO( TypeName ) \
private: \
    friend SuperClassTypeDeduction; \
    friend TypeInfoInitializer; \
\
public: \
    using Super = typename SuperClassTypeDeduction<TypeName>::Type; \
    using ThisType = TypeName; \
\
    static TypeInfo& StaticTypeInfo() \
    { \
        static TypeInfo typeInfo{ TypeInfoInitializer<ThisType>( #TypeName ) }; \
        return typeInfo; \
    } \
\
    virtual const TypeInfo& GetTypeInfo() const \
    { \
        return m_typeInfo; \
    } \
\
private: \
    inline static TypeInfo& m_typeInfo = StaticTypeInfo(); \
\
private: \
```

클래스는 매크로를 통해서 첫번째로 Super라는 부모 클래스 타입의 별칭과 ThisType이라는 자신 타입의 별칭을 선언합니다.

ThisType은 매크로의 TypeName을 그대로 이용할 수 있으나 부모 클래스의 타입은 부모 클래스가 없는 경우로 인해 추가적인 처리가 필요합니다.

타입 정보 생성

```
#define GENERATE_CLASS_TYPE_INFO( TypeName )    \
private:    \
    friend SuperClassTypeDeduction; \
    friend TypeInfoInitializer; \
\
public: \
    using Super = typename SuperClassTypeDeduction<TypeName>::Type; \
    using ThisType = TypeName; \
\
    static TypeInfo& StaticTypeInfo() \
```

```
1 template <typename T, typename U = void>
2 struct SuperClassTypeDeduction
3 {
4     using Type = void;
5 };
6
7 template <typename T>
8 struct SuperClassTypeDeduction<T, std::void_t<typename T::ThisType>>
9 {
10     using Type = T::ThisType;
11 };
```

SuperClassTypeDeduction 은 부모 클래스의 타입을 가져오기 위해 사용되는 구조체입니다.

std::void_t를 이용한 SFINAE를 통해 해당 타입에 ThisType 별칭이 있다면 그 타입을 부모 클래스의 타입으로 하고 없다면 void를 사용하도록 합니다.

타입 정보 생성

```
#define GENERATE_CLASS_TYPE_INFO( TypeName ) \
private: \
    friend SuperClassTypeDeduction; \
    friend TypeInfoInitializer; \
\
public: \
    using Super = typename SuperClassTypeDeduction<TypeName>::Type; \
    using ThisType = TypeName; \
\
    static TypeInfo& StaticTypeInfo() \
    { \
        static TypeInfo typeInfo{ TypeInfoInitializer<ThisType>( #TypeName ) }; \
        return typeInfo; \
    } \
\
    virtual const TypeInfo& GetTypeInfo() const \
    { \
        return m_typeInfo; \
    } \
\
private: \
    inline static TypeInfo& m_typeInfo = StaticTypeInfo(); \
\
private: \
```

이 선언 전까지 ThisType은 부모 클래스의 타입.

자신의 타입을 나타내는 ThisType을 부모 타입으로 할 수 있는 이유는 자신의 클래스 내에서 ThisType을 선언하기 전까지는 부모로 부터 상속 받은 ThisType이 사용되기 때문입니다.

즉 부모의 ThisType이 사용되기에 자식 입장에서는 Super가 됩니다.

타입 정보 생성

```
#define GENERATE_CLASS_TYPE_INFO( TypeName ) \
private: \
    friend SuperClassTypeDeduction; \
    friend TypeInfoInitializer; \
\
public: \
    using Super = typename SuperClassTypeDeduction<TypeName>::Type; \
    using ThisType = TypeName; \
\
    static TypeInfo& StaticTypeInfo() \
    { \
        static TypeInfo typeInfo{ TypeInfoInitializer<ThisType>( #TypeName ) }; \
        return typeInfo; \
    } \
\
    virtual const TypeInfo& GetTypeInfo() const \
    { \
        return m_typeInfo; \
    } \
\
private: \
    inline static TypeInfo& m_typeInfo = StaticTypeInfo(); \
\
private: \
```

두번째로 매크로는 클래스에 대한 TypeInfo 객체를 정적 멤버 변수로 선언합니다.

정적 멤버 변수로 선언했기 때문에 프로그램 시작 시 생성되고 같은 dll 내에서는 중복 생성되지 않습니다.

타입 정보 생성

```
1 class TypeInfo
2 {
3 public:
4     template <typename T>
5     explicit TypeInfo( const TypeInfoInitializer<T>& initializer ) :
6         m_typeHash( typeid( T ).hash_code() ),
7         m_name( initializer.m_name ),
8         m_fullName( typeid( T ).name() ),
9         m_super( initializer.m_super )
10    {
11    }
12
13    /* 일부 코드 생략 */
14
15 private:
16     friend Method;
17     friend Property;
18     using MethodMap = std::map<std::string_view, const Method*>;
19     using PropertyMap = std::map<std::string_view, const Property*>;
20
21    /* 일부 코드 생략 */
22
23    size_t m_typeHash;
24    const char* m_name = nullptr;
25    std::string m_fullName;
26    const TypeInfo* m_super = nullptr;
27
28    std::vector<const Method*> m_methods;
29    MethodMap m_methodMap;
30
31    std::vector<const Property*> m_properties;
32    PropertyMap m_propertyMap;
33 };
```

매크로는 다 보았으니 실제 타입 정보를 담는 TypeInfo 클래스를 보겠습니다.

가장 주목할 부분은 부모의 TypeInfo에 대한 주소를 가지고 있는 부분입니다.

이를 통해서 TypeInfo는 타입의 상속 관계를 표현합니다. 다만 다중 상속을 지원하지 않는다는 제약이 있습니다.

타입 정보 생성

```
1 template <typename T>
2 concept HasSuper = requires
3 {
4     typename T::Super;
5 } && !std::same_as<typename T::Super, void>;
6
7 template <typename T>
8 struct TypeInfoInitializer
9 {
10     TypeInfoInitializer( const char* name ) :
11         m_name( name )
12     {
13         if constexpr ( HasSuper<T> )
14         {
15             m_super = &( typename T::Super::StaticTypeInfo() );
16         }
17     }
18
19     const char* m_name = nullptr;
20     const TypeInfo* m_super = nullptr;
21 };
```

GENERATE_CLASS_TYPE_INFO 매크로와 같이 TypeInfo도 부모 클래스가 없는 경우에 대한 처리가 필요합니다.

해당 처리는 TypeInfo 초기화용으로 사용되는 TypeInfoInitializer 구조체에서 처리합니다.

타입 정보 생성

```
1 template <typename T>
2 concept HasSuper = requires
3 {
4     typename T::Super;
5 } && !std::same_as<typename T::Super, void>;
6
7 template <typename T>
8 struct TypeInfoInitializer
9 {
10     TypeInfoInitializer( const char* name ) :
11         m_name( name )
12     {
13         if constexpr ( HasSuper<T> )
14         {
15             m_super = &( typename T::Super::StaticTypeInfo() );
16         }
17     }
18
19     const char* m_name = nullptr;
20     const TypeInfo* m_super = nullptr;
21 };
```

C++20의 Concept과 constexpr if를 통해서 부모 타입의 TypeInfo를 얻어 오도록 되어 있습니다.

HasSuper 은 Super 별칭이 있고 void가 아닐 것을 요구합니다.

이를 통해 Super 타입 별칭이 없거나 void이면 부모 타입의 TypeInfo는 nullptr이 됩니다.

타입 정보 생성

```
1 bool IsA( const TypeInfo& other ) const
2 {
3     if ( this == &other )
4     {
5         return true;
6     }
7     return m_typeHash == other.m_typeHash;
8 }
9
10
11 bool IsChildOf( const TypeInfo& other ) const
12 {
13     if ( IsA( other ) )
14     {
15         return true;
16     }
17     for ( const TypeInfo* super = m_super; super != nullptr; super = super->GetSuper() )
18     {
19         if ( super->IsA( other ) )
20         {
21             return true;
22         }
23     }
24     return false;
25 }
26
27
28
29 template <typename T>
30 bool IsA() const
31 {
32     return IsA( GetStaticTypeInfo<T>() );
33 }
34
35 template <typename T>
36 bool IsChildOf() const
37 {
38     return IsChildOf( GetStaticTypeInfo<T>() );
39 }
```

이제 TypeInfo의 상속 관계 정보를 통해 클래스간 상속 관계를 런타임에 조사할 수 있습니다.

IsA()는 TypeInfo가 동일한 타입인지 검사합니다. TypeInfo는 static 변수이기 때문에 대부분 주소검사로 확인이 가능한데 다른 데의 같은 타입의 경우 처리가 안되기 때문에 typeid로 얻은 해시 값도 비교합니다.

타입 정보 생성

```
1 bool IsA( const TypeInfo& other ) const
2 {
3     if ( this == &other )
4     {
5         return true;
6     }
7     return m_typeHash == other.m_typeHash;
8 }
9
10
11 bool IsChildOf( const TypeInfo& other ) const
12 {
13     if ( IsA( other ) )
14     {
15         return true;
16     }
17
18     for ( const TypeInfo* super = m_super; super != nullptr; super = super->GetSuper() )
19     {
20         if ( super->IsA( other ) )
21         {
22             return true;
23         }
24     }
25
26     return false;
27 }
28
29 template <typename T>
30 bool IsA() const
31 {
32     return IsA( GetStaticTypeInfo<T>() );
33 }
34
35 template <typename T>
36 bool IsChildOf() const
37 {
38     return IsChildOf( GetStaticTypeInfo<T>() );
39 }
```

IsChildOf()는 현재 TypeInfo가 other를 상속받은 타입인지를 런타임에 검사합니다.

TypeInfo 생성시 얻어온 부모 클래스의 TypeInfo를 순회하면서 IsA()로 동일한 타입인지를 확인합니다.

타입 정보 생성

```
1 template <typename To, typename From>
2 To* Cast( From* src )
3 {
4     return src && src->GetTypeInfo().IsChildOf<To>() ? reinterpret_cast<To*>( src ) : nullptr;
5 }
```

이제 IsChildOf 함수를 이용해서 dynamic_cast를 대체하는 새로운 Cast 함수를 정의할 수 있습니다.

```
1 template <typename T>
2 T* GetComponent( )
3 {
4     for ( Component* component : m_components )
5     {
6         if ( auto concrete = Cast<T>( component ) )
7         {
8             return concrete;
9         }
10    }
11
12    return nullptr;
13 }
```

변수 정보 생성

여기 까지만 해도 `dynamic_cast`를 대체하려고 한 목적을 어느정도 달성 할 수 있었습니다.

이제부터 다룰 내용은 이 기능을 구현하려고 한 계기는 아니지만 리플렉션의 주요기능 중 하나인 멤버 변수의 질의를 위한 변수 정보 생성입니다.

변수의 정보는 다음과 같은 `PROPERTY` 매크로를 통해 생성합니다.

```
1 PROPERTY( member )  
2 int m_member = 0;  
3  
4 PROPERTY( staticMember )  
5 static bool m_staticMember;
```


변수 정보 생성

리플렉션에 의해서 관리되는 변수는 TypeInfo를 통해서 얻어 올 수 있고 값을 가져오거나(Get) 대입(Set)할 수 있습니다.

```
1 B b;  
2 // 실제 타입을 모르는 상황을 가정  
3 A* a = &b;  
4 const auto& typeInfo = a->GetTypeInfo();  
5 if ( auto p = typeInfo.GetProperty( "member" ) ) // member라는 멤버 변수가 있는지 검사  
6 {  
7     int value = p->Get<int>( a );  
8     std::cout << "before : m_member : " << value << "\n";  
9  
10    p->Set( a, 125 );  
11    value = p->Get<int>( a );  
12    std::cout << "after : m_member : " << value << "\n";  
13 }  
14  
15 if ( auto p = typeInfo.GetProperty( "staticMember" ) ) // staticMember라는 멤버 변수가 있는지 검사  
16 {  
17 {  
18     bool value = p->Get<bool>( a );  
19     std::cout << "before : m_staticMember : " << value << "\n";  
20  
21     p->Set( a, true );  
22     value = p->Get<bool>( a );  
23     std::cout << "after : m_staticMember : " << value << "\n";  
24 }
```

Microsoft Visual Studio 디버그 콘솔

```
before : m_member : 0  
after : m_member : 125  
before : m_staticMemeber : 0  
after : m_staticMemeber : 1
```

b	{m_member=125 }
A	{...}
m_member	125
B::m_staticMember	true

변수 정보 생성

```
#define PROPERTY( Name ) \
    inline static struct RegistPropertyExecutor_##Name \
    { \
        RegistPropertyExecutor_##Name() \
        { \
            static PropertyRegister<ThisType, decltype(m_##Name), decltype \
                (&ThisType::m_##Name), &ThisType::m_##Name> property_register_##Name \
                { #Name, ThisType::StaticTypeInfo() }; \
        } \
    } \
    } regist_##Name;
```

그럼 이제 타입에 변수 정보를 생성하는 PROPERTY 매크로를 살펴보도록 하겠습니다.

변수 정보는 각 변수마다 정의된 등록 구조체의 생성자에 선언된 PropertyRegister 객체를 통해 생성됩니다.

변수 정보 생성

```
#define PROPERTY( Name ) \
    inline static struct RegistPropertyExecutor_##Name \
    { \
        RegistPropertyExecutor_##Name() \
        { \
            static PropertyRegister<ThisType, decltype(m_##Name), decltype\n            (&ThisType::m_##Name), &ThisType::m_##Name> property_register_##Name\n            { #Name, ThisType::StaticTypeInfo() }; \
        } \
    } \
    } regist_##Name;
```

① ② ③ ④

```
<ThisType, decltype(m_##Name), decltype(&ThisType::m_##Name), &ThisType::m_##Name>
```

템플릿 클래스 PropertyRegister는 아래와 같은 정보를 담고 있어 **각각의 변수마다 유일한 타입**을 갖게 합니다.

1. 소유하고 있는 객체의 타입
2. 변수의 타입
3. 변수 포인터 타입(static or 멤버)
4. 변수의 주소

변수 정보 생성

```
1 template <typename TClass, typename T, typename TPtr, TPtr ptr>
2 class PropertyRegister
3 {
4 public:
5     PropertyRegister( const char* name, TypeInfo& typeInfo )
6     {
7         if constexpr ( std::is_member_pointer_v<TPtr> )
8         {
9             static PropertyHandler<TClass, T> handler( ptr );
10            static PropertyInitializer initializer = {
11                .m_name = name,
12                .m_type = TypeInfo::GetStaticTypeInfo<T>(),
13                .m_handler = handler
14            };
15            static Property property( typeInfo, initializer );
16        }
17        else
18        {
19            static StaticPropertyHandler<TClass, T> handler( ptr );
20            static PropertyInitializer initializer = {
21                .m_name = name,
22                .m_type = TypeInfo::GetStaticTypeInfo<T>(),
23                .m_handler = handler
24            };
25            static Property property( typeInfo, initializer );
26        }
27    }
28 };
```

이제 PropertyRegister의 생성자는 어떤 일을 하는지 보겠습니다.

Property 객체 생성에 필요한 객체들을 static으로 생성하고 있습니다.

타입이 아닌 템플릿 파라미터로 해당 변수의 포인터를 저장하고 있기 때문에 동일 클래스의 동일한 타입의 변수라도 구분이 가능해집니다.

변수 정보 생성

```
1 template <typename TClass, typename T, typename TPtr, TPtr ptr>
2 class PropertyRegister
3 {
4 public:
5     PropertyRegister( const char* name, TypeInfo& typeInfo )
6     {
7         if constexpr ( std::is_member_pointer_v<TPtr> )
8         {
9             static PropertyHandler<TClass, T> handler( ptr );
10            static PropertyInitializer initializer = {
11                .m_name = name,
12                .m_type = TypeInfo::GetStaticTypeInfo<T>(),
13                .m_handler = handler
14            };
15            static Property property( typeInfo, initializer );
16        }
17        else
18        {
19            static StaticPropertyHandler<TClass, T> handler( ptr );
20            static PropertyInitializer initializer = {
21                .m_name = name,
22                .m_type = TypeInfo::GetStaticTypeInfo<T>(),
23                .m_handler = handler
24            };
25            static Property property( typeInfo, initializer );
26        }
27    }
28 };
```

여기서도 constexpr if 가 사용되었는데 static 여부에 따라서 다른 handler를 사용하기 때문입니다.

handler는 변수의 값을 얻거나 대입하는 일을 담당하는 객체로 포인터의 타입에 따라 다른 표현식을 사용하기에 분리되어야 합니다.

변수 정보 생성

```
1 template <typename TClass, typename T>
2 class PropertyHandler : public IPropertyHandler<T>
3 {
4     GENERATE_CLASS_TYPE_INFO( PropertyHandler )
5     using MemberPtr = T TClass::*;
6
7 public:
8     virtual T& Get( void* object ) const override
9     {
10         return static_cast<TClass*>( object )->*m_ptr;
11     }
12
13     virtual void Set( void* object, const T& value ) const override
14     {
15         static_cast<TClass*>( object )->*m_ptr = value;
16     }
17
18     explicit PropertyHandler( MemberPtr ptr ) :
19         m_ptr( ptr ) {}
20
21 private:
22     MemberPtr m_ptr = nullptr;
23 };
```

멤버 변수를 위한 PropertyHandler
는 다음과 같이 작성하였습니다.

변수 정보 생성

```
1 template <typename TClass, typename T>
2 class StaticPropertyHandler : public IPropertyHandler<T>
3 {
4     GENERATE_CLASS_TYPE_INFO( StaticPropertyHandler )
5
6 public:
7     virtual T& Get( [[maybe_unused]] void* object ) const override
8     {
9         return *m_ptr;
10    }
11
12    virtual void Set( [[maybe_unused]] void* object, const T& value ) const override
13    {
14        *m_ptr = value;
15    }
16
17    explicit StaticPropertyHandler( T* ptr ) :
18        m_ptr( ptr ) {}
19
20 private:
21     T* m_ptr = nullptr;
22 };
```

StaticPropertyHandler는 정적 변수를 담당하고 다음과 같이 작성하였습니다.

변수 정보 생성

```
1 class Property
2 {
3 public:
4     /* 일부 생략 */
5     template <typename T>
6     ReturnValueWrapper<T> Get( void* object ) const
7     {
8         if ( m_handler.GetTypeInfo().IsChildOf<IPropertyHandler<T>>() )
9         {
10             auto concreteHandler = static_cast<const IPropertyHandler<T>*>( &m_handler );
11             return ReturnValueWrapper( concreteHandler->Get( object ) );
12         }
13         else
14         {
15             assert( false && "Property::Get<T> - Invalid casting" );
16             return {};
17         }
18     }
19     template <typename T>
20     void Set( void* object, const T& value ) const
21     {
22         if ( m_handler.GetTypeInfo().IsChildOf<IPropertyHandler<T>>() )
23         {
24             auto concreteHandler = static_cast<const IPropertyHandler<T>*>( &m_handler );
25             concreteHandler->Set( object, value );
26         }
27         else
28         {
29             assert( false && "Property::Set<T> - Invalid casting" );
30         }
31     }
32     /* 일부 생략 */
33     Property( TypeInfo& owner, const PropertyInitializer& initializer ) :
34         m_name( initializer.m_name ),
35         m_type( initializer.m_type ),
36         m_handler( initializer.m_handler )
37     {
38         owner.AddProperty( this );
39     }
40
41 private:
42     const char* m_name = nullptr;
43     const TypeInfo& m_type;
44     const PropertyHandlerBase& m_handler;
45 };
```

이제 Property 클래스를 살펴볼 수 있을 것 같습니다.

Property 클래스의 생성자에서 해당 변수를 소유하고 있는 TypeInfo에 Property를 추가하고 있습니다.

이제 해당 타입은 변수에 대한 정보를 관리하게 되었습니다.

변수 정보 생성

```
1 class Property
2 {
3 public:
4     /* 일부 생략 */
5     template <typename T>
6     ReturnValueWrapper<T> Get( void* object ) const
7     {
8         if ( m_handler.GetTypeInfo().IsChildOf<IPropertyHandler<T>>() )
9         {
10             auto concreteHandler = static_cast<const IPropertyHandler<T>*>( &m_handler );
11             return ReturnValueWrapper( concreteHandler->Get( object ) );
12         }
13         else
14         {
15             assert( false && "Property::Get<T> - Invalid casting" );
16             return {};
17         }
18     }
19     template <typename T>
20     void Set( void* object, const T& value ) const
21     {
22         if ( m_handler.GetTypeInfo().IsChildOf<IPropertyHandler<T>>() )
23         {
24             auto concreteHandler = static_cast<const IPropertyHandler<T>*>( &m_handler );
25             concreteHandler->Set( object, value );
26         }
27         else
28         {
29             assert( false && "Property::Set<T> - Invalid casting" );
30         }
31     }
32     /* 일부 생략 */
33     Property( TypeInfo& owner, const PropertyInitializer& initializer ) :
34         m_name( initializer.m_name ),
35         m_type( initializer.m_type ),
36         m_handler( initializer.m_handler )
37     {
38         owner.AddProperty( this );
39     }
40
41 private:
42     const char* m_name = nullptr;
43     const TypeInfo& m_type;
44     const PropertyHandlerBase& m_handler;
45 };
```

이제 Property 클래스의 전부라 할 수 있는 Get, Set 함수를 보겠습니다.

handler 클래스의 선언을 주의 깊게 보셨다면 handler도 리플렉션으로 관리하고 있다는 걸 보셨을 거라 생각합니다.

변수 정보 생성

```
1 class Property
2 {
3 public:
4     /* 일부 생략 */
5     template <typename T>
6     ReturnValueWrapper<T> Get( void* object ) const
7     {
8         if ( m_handler.GetTypeInfo().IsChildOf<IPropertyHandler<T>>() )
9         {
10             auto concreteHandler = static_cast<const IPropertyHandler<T>*>( &m_handler );
11             return ReturnValueWrapper( concreteHandler->Get( object ) );
12         }
13         else
14         {
15             assert( false && "Property::Get<T> - Invalid casting" );
16             return {};
17         }
18     }
19     template <typename T>
20     void Set( void* object, const T& value ) const
21     {
22         if ( m_handler.GetTypeInfo().IsChildOf<IPropertyHandler<T>>() )
23         {
24             auto concreteHandler = static_cast<const IPropertyHandler<T>*>( &m_handler );
25             concreteHandler->Set( object, value );
26         }
27         else
28         {
29             assert( false && "Property::Set<T> - Invalid casting" );
30         }
31     }
32     /* 일부 생략 */
33     Property( TypeInfo& owner, const PropertyInitializer& initializer ) :
34         m_name( initializer.m_name ),
35         m_type( initializer.m_type ),
36         m_handler( initializer.m_handler )
37     {
38         owner.AddProperty( this );
39     }
40
41 private:
42     const char* m_name = nullptr;
43     const TypeInfo& m_type;
44     const PropertyHandlerBase& m_handler;
45 };
```

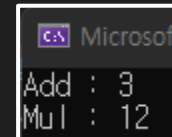
타입 정보를 가지고 있기 때문에 Get / Set 시 템플릿 인자로 변수의 형을 받아 알맞은 handle로 형 변환 하는데 이를 이용합니다.

여기서는 생략하였지만 소유자의 타입도 검사하는 Get / Set 함수도 제공되고 있으니 좀 더 안전한 변환을 위해서는 해당 방법을 사용할 수도 있습니다.

함수 정보 생성

마지막으로 함수 정보를 생성하는 과정을 살펴보겠습니다. 다음은 함수 정보를 생성하고 사용하는 예시입니다.

```
1 class B : public A
2 {
3     GENERATE_CLASS_TYPE_INFO( B )
4
5 public:
6     METHOD( Add )
7     int Add( int a, int b )
8     {
9         return a + b;
10    }
11
12    METHOD( Mul )
13    static int Mul( int a, int b )
14    {
15        return a * b;
16    }
17 };
18
19 B b;
20 A* a = &b;
21 const auto& typeInfo = a->GetTypeInfo();
22 if ( auto m = typeInfo.GetMethod( "Add" ) )
23 {
24     std::cout << "Add : " << m->Invoke<int>( a, 1, 2 ) << "\n";
25 }
26
27 if ( auto m = typeInfo.GetMethod( "Mul" ) )
28 {
29     std::cout << "Mul : " << m->Invoke<int>( a, 4, 3 ) << "\n";
30 }
```



```
Microsoft
Add : 3
Mul : 12
```

함수 정보 생성

함수 정보를 생성해서 TypeInfo에 등록하는 과정은 변수 정보와 크게 차이가 없습니다.

따라서 해당 부분은 생략하도록 하고 함수의 호출을 담당하는 Callable클래스와 함수 정보를 담고 있는 Method 클래스를 어떻게 작성했는지 보겠습니다.

변수의 Set / Get 을 담당하는 handler클래스와 유사하게 Callable클래스도 정적 함수와 멤버 함수 간의 표현식의 차이로 서로 다른 클래스를 사용하게 작성하였습니다.

함수 정보 생성

```
1 template <typename TClass, typename TRet, typename... TArgs>
2 class Callable : public ICallable<TRet, TArgs...>
3 {
4     GENERATE_CLASS_TYPE_INFO( Callable )
5     using FuncPtr = TRet( TClass::* )( TArgs... );
6
7 public:
8     virtual TRet Invoke( void* caller, TArgs&&... args ) const override
9     {
10         if constexpr ( std::same_as<TRet, void> )
11         {
12             ( static_cast<TClass*>( caller )->*m_ptr )( std::forward<TArgs>( args )... );
13         }
14         else
15         {
16             return ( static_cast<TClass*>( caller )->*m_ptr )( std::forward<TArgs>( args )... );
17         }
18     }
19
20     Callable( FuncPtr ptr ) :
21         m_ptr( ptr ) {}
22
23 private:
24     FuncPtr m_ptr = nullptr;
25 };
```

멤버 함수 호출을 담당하는 Callable 클래스입니다.

함수의 반환 타입이 void일 경우를 처리하기 위해서 constexpr if 와 same_as concept을 사용했습니다.

함수 정보 생성

```
1 template <typename TClass, typename TRet, typename... TArgs>
2 class StaticCallable : public ICallable<TRet, TArgs...>
3 {
4     GENERATE_CLASS_TYPE_INFO( StaticCallable )
5     using FuncPtr = TRet( * )( TArgs... );
6
7 public:
8     virtual TRet Invoke( [[maybe_unused]] void* caller, TArgs&&... args ) const override
9     {
10         if constexpr ( std::same_as<TRet, void> )
11         {
12             ( *m_ptr )( std::forward<TArgs>( args )... );
13         }
14         else
15         {
16             return ( *m_ptr )( std::forward<TArgs>( args )... );
17         }
18     }
19
20     StaticCallable( [[maybe_unused]] TClass* owner, FuncPtr ptr ) :
21         m_ptr( ptr ) {}
22
23 private:
24     FuncPtr m_ptr = nullptr;
25 };
```

정적 멤버 함수 호출을 담당하는 StaticCallable 클래스입니다.

void 반환에 대해 멤버 함수와 같은 처리를 하였습니다.

함수 정보 생성

```
1 class Method
2 {
3 public:
4     /* 일부 생략 */
5     template <typename TRet, typename... TArgs>
6     TRet Invoke( void* owner, TArgs&&... args ) const
7     {
8         if ( m_callable.GetTypeInfo().IsChildOf<ICallable<TRet, TArgs...>>() )
9         {
10             auto concreteCallable = static_cast<const ICallable<TRet, TArgs...>>( &m_callable );
11             if constexpr ( std::same_as<TRet, void> )
12             {
13                 concreteCallable->Invoke( owner, std::forward<TArgs>( args )... );
14             }
15             else
16             {
17                 return concreteCallable->Invoke( owner, std::forward<TArgs>( args )... );
18             }
19         }
20         /* 예외 상황 생략 */
21     }
22     /* 일부 생략 */
23     template <typename TClass, typename TRet, typename... TArgs>
24     Method( TypeInfo& owner, [[maybe_unused]] TRet( TClass::* ptr )( TArgs... ), const char* name,
25     const CallableBase& callable ) :
26     m_name( name ),
27     m_callable( callable )
28     {
29         CollectFunctionSignature<TRet, TArgs...>();
30         owner.AddMethod( this );
31     }
32 private:
33     template <typename TRet, typename... Args>
34     void CollectFunctionSignature()
35     {
36         m_returnType = &TypeInfo::GetStaticTypeInfo<TRet>();
37         m_parameterTypes.reserve( sizeof...( Args ) );
38         ( m_parameterTypes.emplace_back( &TypeInfo::GetStaticTypeInfo<Args>() ), ... );
39     }
40     const TypeInfo* m_returnType = nullptr;
41     std::vector<const TypeInfo*> m_parameterTypes;
42     const char* m_name = nullptr;
43     const CallableBase& m_callable;
44 };
```

함수 정보를 담고 있는 Method 클래스입니다.

기본적으로 함수의 반환 타입이 무엇인지 인자들의 타입이 무엇인지를 담고 있습니다.

CollectFunctionSignature()에서 템플릿 파라미터를 통해 타입 정보를 수집하도록 작성하였습니다.

함수 정보 생성

```
1 class Method
2 {
3 public:
4     /* 일부 생략 */
5     template <typename TRet, typename... TArgs>
6     TRet Invoke( void* owner, TArgs&&... args ) const
7     {
8         if ( m_callable.GetTypeInfo().IsChildOf<ICallable<TRet, TArgs...>() )
9         {
10             auto concreateCallable = static_cast<const ICallable<TRet, TArgs...>>( &m_callable );
11             if constexpr ( std::same_as<TRet, void> )
12             {
13                 concreateCallable->Invoke( owner, std::forward<TArgs>( args )... );
14             }
15             else
16             {
17                 return concreateCallable->Invoke( owner, std::forward<TArgs>( args )... );
18             }
19         }
20         /* 예외 상황 생략 */
21     }
22     /* 일부 생략 */
23     template <typename TClass, typename TRet, typename... TArgs>
24     Method( TypeInfo& owner, [[maybe_unused]] TRet( TClass::* ptr )( TArgs... ), const char* name,
25     const CallableBase& callable ) :
26     m_name( name ),
27     m_callable( callable )
28     {
29         CollectFunctionSignature<TRet, TArgs...>();
30         owner.AddMethod( this );
31     }
32 private:
33     template <typename TRet, typename... Args>
34     void CollectFunctionSignature()
35     {
36         m_returnType = &TypeInfo::GetStaticTypeInfo<TRet>();
37         m_parameterTypes.reserve( sizeof...( Args ) );
38         ( m_parameterTypes.emplace_back( &TypeInfo::GetStaticTypeInfo<Args>() ), ... );
39     }
40     const TypeInfo* m_returnType = nullptr;
41     std::vector<const TypeInfo*> m_parameterTypes;
42     const char* m_name = nullptr;
43     const CallableBase& m_callable;
44 };
```

함수를 소유하고 있는 TypeInfo에 등록하는 과정은 보시다시피 변수와 차이가 없습니다.

Invoke 함수에서도 Callable 클래스가 리플렉션으로 관리되고 있다는 점을 이용하여 변수의 경우와 유사하게 처리하도록 작성하였습니다.

차이점은 void에 대한 처리 뿐입니다.

마치며...

준비한 내용은 여기까지 입니다. 이 내용을 통해 매크로와 템플릿을 사용한 여러 기법에 대한 아이디어를 얻으셨다면 좋을 것 같습니다.

마무리하면서 리플렉션으로 관리할 정보를 등록하는 매크로의 문법으로 인한 제약점을 하나 소개하려고 합니다.

이 제약은 다음과 같은 문법 때문인데요.

```
1 METHOD( Add )  
2 int Add( int a, int b )  
3 {  
4     return a + b;  
5 }
```

마치며...

매크로에 함수의 이름을 넘기고 이를 통해서 함수 포인터 타입을 얻어옵니다.
문제는 함수가 동일한 이름으로 오버로딩 된 경우에는 함수 포인터 타입을 유추하는데 실패하게 됩니다.

이는 decltype이 실패하기 때문이며 지금과 같은 매크로 문법에서는 이름을 바꾸는 것 외에는 해결 하기 어려운 제약입니다.

```
1 METHOD( Add )
2 int Add( int a, int b )
3 {
4     return a + b;
5 }
6
7 int Add( int a, int b, int c )
8 {
9     return a + b;
10 }
```

더 자세한 코드를 보고 싶으시다면...

- <https://github.com/xtozero/SSR/tree/multi-thread/Source/Core/Public/Reflection>
- <https://github.com/xtozero/SSR/tree/multi-thread/Source/Core/Private/Reflection>

참고자료

- <https://github.com/Aumoa/CPP.REF>
- <https://www.codeproject.com/Articles/8712/AGM-LibReflection-A-reflection-library-for-C>