What's Reflective?

Reflective is a C++ library providing detailed run-time code reflection, supporting:

- member-data or member-function based properties with getters and setters.
- method-based actions, with return type and any number of fully qualified parameters.
- vectorized construction\destruction\copy\move of objects, to allow efficient implementation of non-template generic collection.
- reflection of class templates, with fully qualified parameters information.
- indirection levels of pointers, and constness of each indirection level.
- safe up\down casts (often programmers forget that even an implicit cast may involve an offset or an indirection).
- self reflection: reflecting data is itself reflected.
- multiple\virtual base classes. Of course, you may not like them, but if they are used somewhere in your code, it's not a problem.

Reflection can have several applications, like:

- generic GUI
- generic serialization
- object oriented databases
- scripting
- remote procedure call

Reflection is not provided by the C++ language itself, so you should reflect your types somehow. There are 3 main modes have the reflection of types: implicit, non-intrusive and intrusive. These modes can be mixed in the same code: you choose the reflection definition mode on a per-type basis.

An example with implicit reflection

With implicit reflection you don't write any code to reflect the type. You rather let the compiler deduce information such size, alignment, name, presence of public *ctors. SFINAE (http://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error) and compiler extension are used to achieve this.

```
#include "reflective.h"

namespace Animals
{
	class Dog
	{
		float m_size;
	public:
		Dog() : m_size( 1.f ) { }
	};
}

int main()
{
	using namespace reflective;
	using namespace Animals;

	// get the type using reflective::get_type<T>() global function
```

```
	const Type & animal_type = get_type< Dog >();

	// get some info
	const SymbolName class_name = animal_type.name();
	const size_t class_size = animal_type.size();
	const SymbolName namespace_name = animal_type.parent_namespace().name();
	const SymbolName outer_namespace_name = animal_type.parent_namespace().parent()->name();
	const Type::Capabilities capabilities = animal_type.capabilities();
		// Type::Capabilities is an enum with OR support

	// create a ToStringBuffer on the stack
	char chars[ 1024 ];
	ToStringBuffer char_buffer( chars );

	// write some info on the buffer
	to_string( char_buffer, class_name ); char_buffer.append_literal( ", " );
	to_string( char_buffer, class_size ); char_buffer.append_literal( ", " );
	to_string( char_buffer, namespace_name ); char_buffer.append_literal( ", " );
	to_string( char_buffer, outer_namespace_name ); char_buffer.append_literal( ", " );
	to_string( char_buffer, capabilities ); char_buffer.append_literal( "\n" );

	// print on the ouput the text
	printf( chars );

	return 0;
}
```

The output of the above program is:

```
Dog, 4, Animals, global, HasDestructor | HasMover | HasCopier |
HasDefaultConstructor
```

In the sample the `Type` object is retrieved by the `reflective::get_type<T>()` global function, which is one way to do it. An other way to get the type from an object:

```
	Dog boby;
	DogSize size_of_boby = 5;
	const Type & dog_type = type_of( boby );
	const Type & height_type = type_of( size_of_boby );
```

`DogSize` may be a primitive type (of course they can be reflected too). `reflective::type_of()` can be used to resolve a derived type from a base pointer or reference, if the type supports type resolving.
A third way to get a type may be searching for it in a namespace:

```
	StaticConstString path( "Animals::Dog" );
	const reflective::Type * dog_type = Namespace::global().find_child_type( path );
```

`Type` is the base of some classes, such `Class` or `Enum`. You can query the type of a type:

```
	const bool is_class = is_instance_of<Class>( dog_type );
	const bool is_enum = type_of( dog_type ) == get_type<Enum>();
```

The capabilities of a type tells you what you can do with it.

```
enum Capabilities
{
	eCapabilitiesNone		= 0,
	eHasDefaultConstructor		= 1 << 0,
	eHasCopier			= 1 << 1,
	eHasMover			= 1 << 2,
	eHasDestructor			= 1 << 3,
	eHasEqualityComparer	= 1 << 4,
	eHasLessThanComparer	= 1 << 5,
	eHasToStringDumper		= 1 << 6,
	eHasFromStringAssigner		= 1 << 7,
	eHasTypeResolver		= 1 << 8,
	eHasCollectionHandler		= 1 << 9
};
```

You can use a `Type` object to construct, destroy, copy or move object, if the type has the capability. In the previous example, the global function `to_string()` is used to convert variables and object to text. This function can be used with any object which has the `eHasToStringDumper` capability.

Non-intrusive reflection

If you use implicit reflection, you don't have detailed information on the type. In the following example non-intrusive reflection is presented:

```
#include "reflective.h"

namespace Shapes
{
	struct Vector
	{
		float x, y;

		Vector() : x( 0.f ), y( 0.f ) { }
		Vector( float x_, float y_ ) : x( x_ ), y( y_ ) { }
	};

	enum Color
	{
		eColorWhite,
		eColorRed,
		eColorYellow,
		eColorMagenta
	};

	// Shape
	class Shape
	{
	protected:
		Shape();

	public:

		Color get_color() const;
		void set_color( Color color );

		virtual float perimeter() const = 0;
		virtual float area() const = 0;
		virtual void move_by( const Vector & offset ) = 0;

	private: // data members
		Color _color;
	};

	// Circle
	class Circle : public Shape
	{
	public:
		Circle();
		Circle( Vector center, float radius );

		// center
		const Vector & get_center() const;
		void set_center( const Vector & new_center );

		// radius
		float get_radius() const;
		void set_radius( float new_radius );
```

```
		// perimeter, area
		float perimeter() const;
		float area() const;

		void move_by( const Vector & offset );
		void scale_radius( float factor );

	private: // data members
		Vector _center;
		float _radius;
	};

}

namespace reflective_externals
{
	// reflection of Shapes::Shape
	reflective::Class * init_type(
		Shapes::Shape * null_pointer_1,
		Shapes::Shape * null_pointer_2 );

	// reflection of Shapes::Circle
	reflective::Class * init_type(
		Shapes::Circle * null_pointer_1,
		Shapes::Circle * null_pointer_2 );

	// reflection of Shapes::Color
	reflective::Enum * init_type(
		Shapes::Color * null_pointer_1,
		Shapes::Color * null_pointer_2 );

} // namespace reflective_externals
```

So, if you declare a function named `init_type` in the namespace `reflective_externals` for a type, it is used to link the type with that type. Here is the implementation of these functions:

```
namespace reflective_externals
{
	// reflection of Shapes::Shape
	reflective::Class * init_type(
		Shapes::Shape * null_pointer_1,
		Shapes::Shape * null_pointer_2 )
	{
		using namespace ::reflective;
		typedef Shapes::Shape ThisClass;

		// class object
		Class * class_object = new_class<ThisClass>();
		class_object->set_no_base_type();

		// properties
		const Property * properties[] =
		{
			new_property<ThisClass>( "color", &ThisClass::get_color,
&ThisClass::set_color ),
			new_property<ThisClass>( "perimeter", &ThisClass::perimeter ),
			new_property<ThisClass>( "area", &ThisClass::area ),
		};

		// actions
```

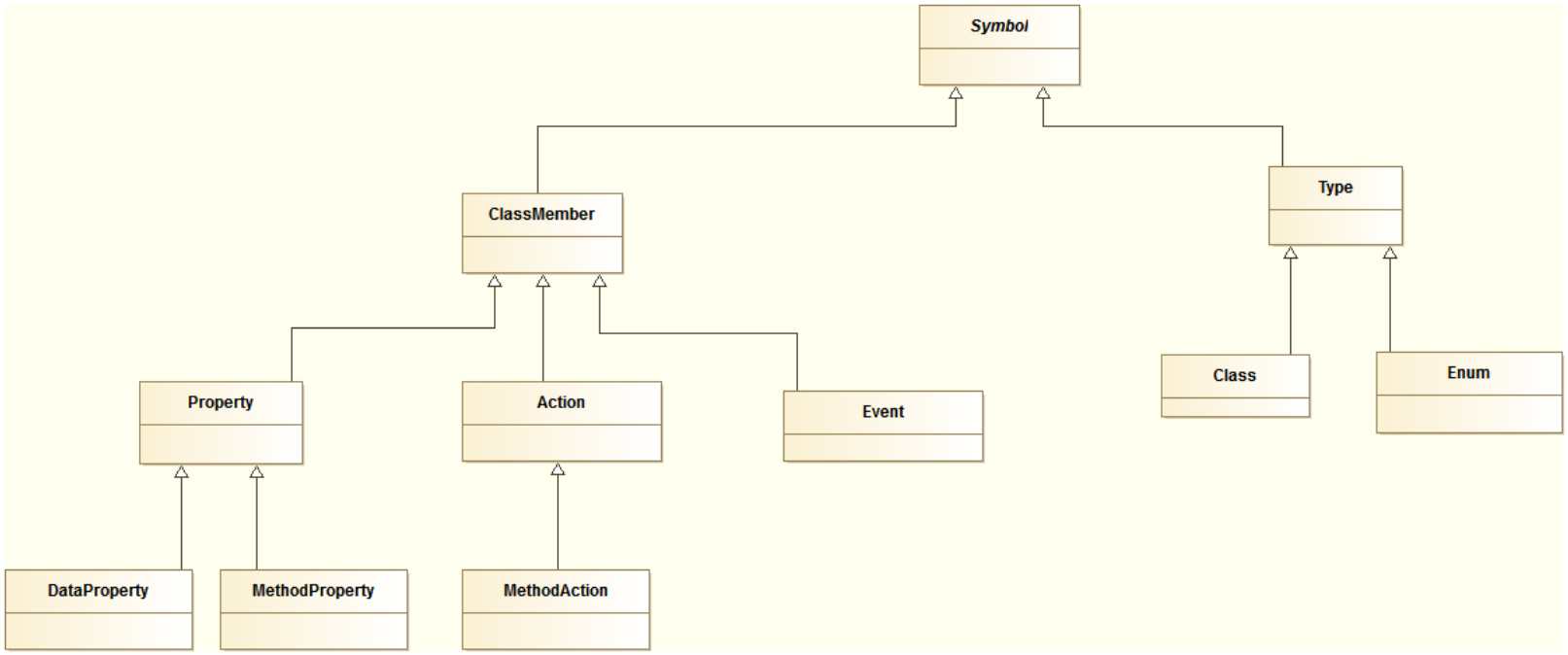
```
		const Action * actions[] =
		{
			new_action<ThisClass>( "move_by", &ThisClass::move_by, "offset"),
		};

		// assign members
		class_object->assign_properties( properties );
		class_object->assign_actions( actions );

		// return type
		return class_object;
	}

	// reflection of Shapes::Circle
	reflective::Class * init_type(
		Shapes::Circle * null_pointer_1,
		Shapes::Circle * null_pointer_2 )
	{
		using namespace ::reflective;
		typedef Shapes::Circle ThisClass;
		typedef Shapes::Shape BaseClass;

		// class object
		Class * class_object = new_class<ThisClass>();
		class_object->set_base_type( BaseType::from_types<ThisClass,BaseClass>() );

		// properties
		const Property * properties[] =
		{
			new_property<ThisClass>( "center", &ThisClass::get_center,
&ThisClass::set_center ),
			new_property<ThisClass>( "radius", &ThisClass::get_radius,
&ThisClass::set_radius ),
			new_property<ThisClass>( "perimeter", &ThisClass::perimeter ),
			new_property<ThisClass>( "area", &ThisClass::area ),
		};

		// actions
		const Action * actions[] =
		{
			new_action<ThisClass>( "move_by", &ThisClass::move_by, "offset"),
			new_action<ThisClass>( "scale_radius", &ThisClass::scale_radius, "factor"),
		};

		// assign members
		class_object->assign_properties( properties );
		class_object->assign_actions( actions );

		// return type
		return class_object;
	}

	// reflection of Shapes::Color
	reflective::Enum * init_type(
		Shapes::Color * null_pointer_1,
		Shapes::Color * null_pointer_2 )
	{
		using namespace ::reflective;

		const Enum::Member * members[] =
```

```
            {
                  new_enum_member( "White", Shapes::eColorWhite ),
                  new_enum_member( "Red", Shapes::eColorRed ),
                  new_enum_member( "Yellow", Shapes::eColorYellow ),
                  new_enum_member( "Magenta", Shapes::eColorMagenta ),
            };

            return new_enum( "Shapes", "Color", members );
      }

} // namespace reflective_externals
```

Classes have 3 kinds of members: properties, actions and events.

- Properties are “fields” of the class, and can be defined with:
  - const or non-const data members
  - getter-setters pairs. One of the two may be absent. The getter, if present, must be const and cannot have parameters. The setter, if present, must be non-const, must return void, and take one parameter. The return type of the getter must mach the parameter type of the setter.
  - Actions are the “methods” of the class, and can be defined by non-const methods with any return type, and zero, one or many parameters of any type. Actually the current implementation of `MethodAction` supports up to 6 parameters for in the definition of the action. Future versions may overcome this limitation exploiting variadic templates. The type of every parameter is deduced by the library, but the names are not. So, with the last parameter you can pass as parameter of the new_action a string with a comma-separated list of names.
- Events are occurrences observable from the external. Since the object is caller, you can think to events like “reverse methods”. An event has a type (usually a composed types), which carries additional information about the occurrence. The support of events in the current implementation is incomplete.

Here is a partial class diagram that show some basic relationships:



Of course, this hierarchy is not closed. You may have your own Property\Action implementation (for example to deal with remote objects), or to add extra information (for example hlsl semantic on vertex layouts).

Class Templates

Class templates are supported. A `Class` object is created for every template instance instantiated by the compiler. All these class objects reference a single `ClassTemplate` object. You may iterate the template parameters, and search for a specific template instance.

```
#include "reflective.h"

template < class COORD_TYPE, int DIM >
	class Vector
{
public:

	Vector()
	{
		// ...
	}

	void normalize()
	{
		// ...
	}

	COORD_TYPE get_module() const
	{
		return 0;
	}
```

```
private:
	COORD_TYPE _coord[ DIM ];
};


namespace reflective_externals
{
	template < class COORD_TYPE, int DIM >
		reflective::Class * init_type(
			Vector< COORD_TYPE, DIM > * null_pointer_1,
			Vector< COORD_TYPE, DIM > * null_pointer_2 )
	{
		using namespace ::reflective;
		typedef Vector< COORD_TYPE, DIM > ThisClass;

		// template parameters
		const TemplateParameter template_parameters[] =
		{
			TemplateParameter( "COORD_TYPE", safe_get_qualified_type< COORD_TYPE >() ),
			TemplateParameter( "DIM", DIM ),
		};

		// class object
		Class * class_object = new_class<ThisClass>( template_parameters );
		class_object->set_no_base_type();

		// properties
		const Property * properties[] =
		{
			new_property<ThisClass>( "module", &ThisClass::get_module ),
		};

		// actions
		const Action * actions[] =
		{
			new_action<ThisClass>( "normalize", &ThisClass::normalize ),
		};

		// assign members
		class_object->assign_properties( properties );
		class_object->assign_actions( actions );

		// return type
		return class_object;
	}

} // namespace reflective_externals

int main()
{
	using namespace reflective;

	typedef Vector< double, 3 > Vector3;
	const ParameterList & parameters = get_class< Vector3 >()->class_template()-
>template_arguments_list();

	// write the template formal parameters
	char chars[ 1024 ];
	ToStringBuffer char_buffer( chars );
	for( Iterator< const Parameter > parameter_iterator( parameters );
```

```
            parameter_iterator; parameter_iterator++ )
        {
            to_string( char_buffer, *parameter_iterator );
            char_buffer.append( '\n' );
        }

        // print on the ouput the text
        printf( chars );
}
```

The ouput of the program is

```
reflective::QualifiedType COORD_TYPE
int DIM
```

In the above exemple the `Iterator` class is used to iterate a type that does have the `eHasCollectionHandler` capability. Iterator can iterate collection of compile-time or run-time know type. The iteration may be filtered to include only types implicitly castable to a compile-time or run-time know type (in the above sample the filter type is `Parameter`).

Properties ad qualified types

The `Property` object can be used to get and set the value on a target object:

```
// Property
class Property : public ClassMember
{
public:

        // ...

        const Type & type() const;

        const QualifiedType & qualified_type() const;

        bool get_value( const void * object, void * dest_value ) const; /* gets a copy of
                the value of the property for the object pointed by 'object'.
                The value is copy-constructed in the buffer, which must be large enough.
                type().size() should be used to determine the required size.
                After using the value the caller must destroy it. type().destroy() method
                may be used.
                If the method returns false, the property could not be read (maybe it's a
                write-only property), and no operation has been performed. */

        bool set_value( void * object, const void * source_value ) const; /* sets the value
                of the property for the object pointed by 'object'. The buffer pointed
                by 'source_value' must contain a valid object of the type of the property.
                The implementation may use the type assignment operator (if any) to set the
value,
                or may destroy and copy-construct it.
                The object pointed by 'source_value' is not modified by set_value(), and the
                caller is responsible for destroying it.
                If the method returns false, the property could not be written (maybe it's a
                read-only property, or maybe the value was not suitable), and no operation
                has been performed. */

        // ...
```

Reflective is aware of pointer types, but it does not distinguish between references and const pointers. The indirection level (the number of stars in the declaration of the pointer) is deduced with templates. Often getter and setters pass objects by reference rather than by value, like in the “Shapes” sample:

```
// center
const Vector & get_center() const;
void set_center( const Vector & new_center );
```

One may think that the type of the property is Vector, but it isn't. If you inspect the `Property` object, you find that the type is Pointer: if you want to call `get_value()`, you have to provide a buffer sized and aligned like a pointer. So, what about “Vector”? You can find it in the qualified type. A qualified type is composed by:

a) a pointer to a const type
b) a type qualification

A type qualification is an object which tells you:

1) the number indirection levels
2) the constness of each of them (it's like an array of bools)
3) a pointer to a const “final type”, which is the type pointed to in the last indirection level (`Vector` in the above example).

Since 1) and 2) are compressed in a 32 bit word, the maximum supported indirection level is 15 (it should be enough...).

Using the `Property` object to set\get property values, or using the `Action` object to invoke methods, may be more difficult than one can think. If you deal with properties, you have to remember to align properly the buffer and to call the destructor on returned values. Invoking actions with parameters may be even more difficult. Furthermore, if you use properties or actions of base class with derived object, you have to down-cast the pointer to the object.
Reflective provides many utility classes to easy this operations:

- `TypeInspector` helps retrieving properties and their values on an object.
- `ActionInvoker` helps invoking actions. You can provide the value of parameters in a string.