Document number: P3132
Authors: Gabriel Morin, Patrice Roy
Audience: SG14,
Date: 2024-02-15
Reply-to: Patrice Roy, patricer at gmail.com

Accept attributes with user-defined prefixes

Abstract

We want the compiler to accept attributes with user-defined prefixes such as [[myCompany::RegisterClass]], in order to introduce standardized syntax for custom code generation tags, and pave the way for future support for user-defined attributes.

Note: this paper is part of the wider effort described in P2966.

Introduction

Many C++ frameworks use custom code annotations to drive code generation. This is especially prevalent in game engines, that use them to:

- link class members to the level editor's authoring process
- in the case of networked games, identify variables whose values should be propagated to clients and how they should be packed to send across the wire.
- to register select classes with a custom RTTI system (since C++'s own RTTI system is seldom used in games - see FAQ).

Here's a syntax example from the well-known Unreal engine:

```
UPROPERTY([specifier, specifier, ...], [meta(key=value, key=value, ...)])
Type VariableName;
```

This kind of syntax is also present in in-house game engines such as the ones an author of this paper worked with at Ubisoft and Eidos:

```
class PatrolPath
{
    REGISTERCLASS
```

```
    Waypoints m_waypoints; PROPERTY(INIT() RUNTIMEEDITABLE)
}
```

In addition to making member variables visible and editable in the editor with the PROPERTY tag, we also use tags such as REGISTERCLASS on classes and enums that need to participate in our custom RTTI implementation.

Note that those tags can sometimes double as macros, but are usually defined to nothing:

```
#define PROPERTY
```

What gives meaning to the tags is a custom tool - let's call it the Property Parser - which is run before compilation. It generates a matching .property.cpp file for each .cpp file in our project.

Similar tools are used in the Unreal Engine and in other game studios' in-house engines. These tools **typically include a minimalist, hand-written parser** which often struggles to keep up with additions to the language. Limitations of the parser often include **limited or no support for templates in registered classes** (for example, only templates with a single argument might be supported).

Additionally, since the Property Parser runs before calling the compiler, and therefore before the preprocessor, **those custom tags cannot be added to the code via macros**.

Motivation

The situation

In-house game engines often lag multiple versions behind the C++ standard. The aforementioned tool limitations are one of the big reasons why. There is little incentive to upgrade if a large part of your code can't even use all C++11 features, let alone C++23, due to lack of support by the parser.

There would be a large benefit for the game development industry and the C++ ecosystem in general if we could offer a migration path away from these ad-hoc tools and towards something based on attributes and reflection. The current state of these tools, while necessary and having allowed the creation of beautiful games, tends to make developer lives miserable.

We hope this proposal can serve as a step towards such a future.

Clang to the rescue

The Clang compiler has been usable for a while for game development on all three major AAA game platforms (PC, Xbox and Playstation). It has a good reputation for performance, and while most studios still use MSVC on PC and Xbox platforms, the incentive to switch to Clang on all platforms to have a unified experience is pretty strong.

Furthermore, Clang offers a powerful plugin system and easy access to its syntax tree for manipulation at various stages of the compilation process. Therefore, along with the unified experience, the prospect of being able to migrate the aforementioned tools to Clang plugins is increasingly interesting. This would allow game studios' code generation tools to finally stop relying on "poor man's parsers" and easily keep up with the evolution of C++ syntax.

Thinking forward

When doing this migration, it might be tempting to keep the same non-standard tags and detect them as macros in the Clang preprocessor step.

However, as these StackOverflow questions illustrate, there is a desire to do this with custom attributes instead:
https://stackoverflow.com/questions/70610120/how-to-use-custom-c-attributes-with-clang-libtooling-without-modifying-clang-c
https://stackoverflow.com/questions/20180917/how-to-access-parsed-c11-attributes-via-clang-tooling

Instead of locking the functionality in some tool, *using user-defined attributes would open the door for reimplementing in standard C++ once custom attributes and reflection finally come around*.

Standardization

Therefore, we would like C++ to accept attributes with a custom prefix by default. Right now, compilers are only required to recognize the attributes defined in the standard [TODO citation needed]. And in practice, besides standard-mandated ones, they only recognize attributes that they define themselves:

MSVC
[[rpr::kernel]]
[[gsl::suppress(rules)]]
[[msvc::flatten]]

Clang
[[clang::opencl_private]]
[[clang::fallthrough]]

GCC

[[gnu::fallthrough]]

They might also ignore some attributes sporting the prefixes used by another major compiler they know about, but this is never guaranteed. This means cross-platform code (frequent in games) must isolate them with preprocessor directives.

Proposal

We propose that all compilers should *accept* [TODO: reword this in a more standardese way] any attribute sporting a prefix they don't recognize, such as:

[[myCompany::Property]]
[[myCompany::RegisterClass]]

Without Proposal	With Proposal

class PatrolPath
{
  REGISTERCLASS
 	
  Waypoints m_waypoints;
PROPERTY(INIT() RUNTIMEEDITABLE)
}
	class PatrolPath
	{
	  [[myCompany::RegisterClass]]
	
	  Waypoints m_waypoints;
	[[myCompany::Property(INIT() RUNTIMEEDITABLE)]]
	}

Cross-compatibility benefit

As a side-benefit, this means that compilers will by default be compatible with attributes introduced by other compilers, reducing the need for cross-platform code to isolate such attributes behind #if preprocessor statements.

Without Proposal	With Proposal
#if defined(__clang__) || defined(__GNUC__) || defined(__GNUG__) [[gnu::flatten]] #elif defined(_MSC_VER) [[msvc::flatten]] #endif void f();	[[gnu::flatten]] [[msvc::flatten]] void f();

Preprocessor compatibility benefit

By becoming attributes and being processed by a compiler plugin (and in the future, by standardized custom attributes), custom code generation tags would now be able to be added through a macro:

Without Proposal	With Proposal

#define BOILERPLATE \
class BoilerplateClass { \
  REGISTERCLASS **//won't be parsed!!!** \
  //(... boilerplate …) \
};
	#define BOILERPLATE \
	class BoilerplateClass { \
	  [[myCompany::RegisterClass]] **// ok!** \
	  //(... boilerplate …) \
	};

Accept, not ignore

Notice that we do not say that compilers should "ignore" said attributes. On the contrary, we would like to encourage implementers such as Clang to parse the attributes and make them available to plugins that read the syntax tree. Since this goes beyond the scope of the C++ standard, this encouragement could take the form of a footnote, or perhaps a separate SG15 tooling standard protocol for compilers that wish to provide this kind of functionality.

Relation with user-defined attributes

Our secondary goal is to get one step closer to custom attributes that are user-implemented in the source code, similar to what already exists in languages like C#: https://learn.microsoft.com/en-us/dotnet/standard/attributes/writing-custom-attributes

By letting users add attributes with a custom namespace to their code right now, we encourage the design of new syntaxes for code generation that are forward-compatible with future developments around attributes and reflection.

Other considerations

[[myCompany::RegisterClass]] is quite verbose and unsightly compared to just REGISTERCLASS. This might slow adoption of attributes with custom prefixes.

One solution that comes to mind is to provide for attributes the typical tools we use to make regular namespaces more palatable:

Equivalent

[[myCompany::RegisterClass]]	using namespace MyCompany;
	[[RegisterClass]]
		namespace myCompany
		{
		  [[RegisterClass]]
		}
			using [[RegisterClass]] =
			[[myCompany::RegisterClass]];
			[[RegisterClass]]

We see value in making the same namespaces usable for both attributes and regular code, but attribute-specific equivalents could be added if the committee deems it necessary (strawman syntax):

using [[namespace]] MyCompany;
[[namespace]] MyCompany { }

Survey of code generation tags across other industries

//TODO

Qt GUI framework for its signals and slots?

//TODO

Related Work

P2552R2 On the ignorability of standard attributes
https://open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2552r2.pdf

FAQ

Why do games typically prefer using a custom RTTI system?

- To reduce executable bloat by generating RTTI info only for a number of select classes
- To ensure a uniform implementation with more predictable performance across platforms and compilers