



Preshing on Programming

- [Twitter](#)
- [RSS](#)

Navigate... ▼

- [Blog](#)
- [Archives](#)
- [About](#)
- [Contact](#)

Jan 24, 2018

A Flexible Reflection System in C++: Part 2

In [the previous post](#), I presented a basic system for **runtime reflection** in C++11. The post included a sample project that created a **type descriptor** using a block of macros:

```
// Define Node's type descriptor
REFLECT_STRUCT_BEGIN(Node)
REFLECT_STRUCT_MEMBER(key)
REFLECT_STRUCT_MEMBER(value)
REFLECT_STRUCT_MEMBER(children)
REFLECT_STRUCT_END()
```

At runtime, the type descriptor was found by calling `reflect::TypeResolver<Node>::get()`.

This reflection system is small but very flexible. In this post, I'll extend it to support additional built-in types. You can clone the project [from GitHub](#) to follow along. At the end, I'll discuss other ways to extend the system.

 [preshing](#) / [FlexibleReflection](#)

Adding Support for double

In [Main.cpp](#), let's change the definition of `Node` so that it contains a `double` instead of an `int`:

```
struct Node {
    std::string key;
    double value;
    std::vector<Node> children;

    REFLECT() // Enable reflection for this type
};
```

Now, when we build the sample project, we get a link error:

```
error: unresolved external symbol "reflect::getPrimitiveDescriptor<double>()"
```

That's because the reflection system doesn't support `double` yet. To add support, add the following code near the bottom of [Primitives.cpp](#), inside the `reflect` namespace. The highlighted line defines the missing function that the linker complained about.

```
//-----
// A type descriptor for double
//-----

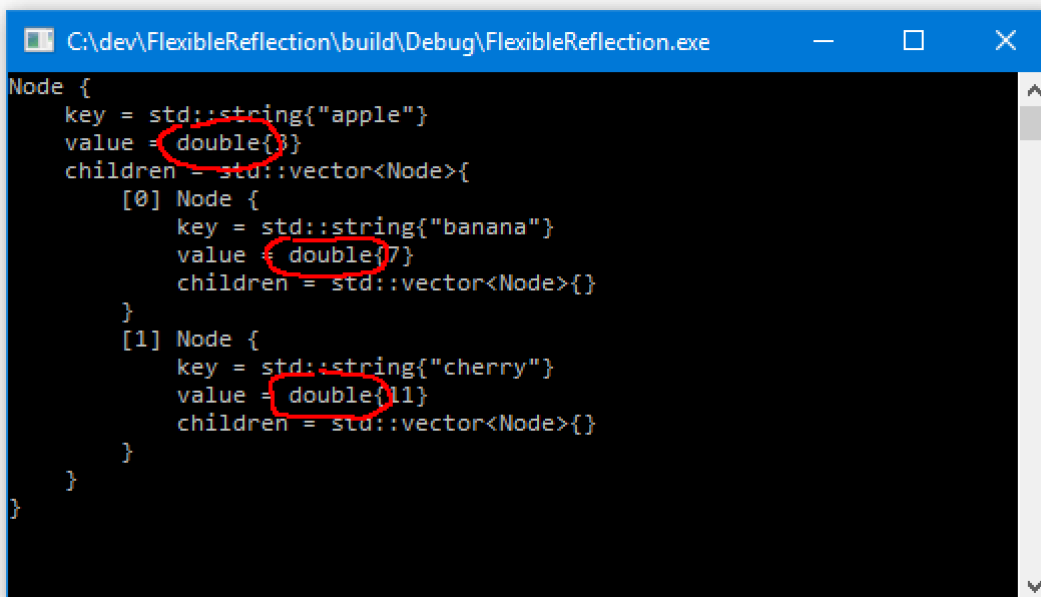
struct TypeDescriptor_Double : TypeDescriptor {
    TypeDescriptor_Double() : TypeDescriptor{"double", sizeof(double)} {}
}

virtual void dump(const void* obj, int /* unused */ const override {
    std::cout << "double{" << *(const double*) obj << "}";
}

};

template <>
TypeDescriptor* getPrimitiveDescriptor<double>() {
    static TypeDescriptor_Double typeDesc;
    return &typeDesc;
}
```

Now, when we run the program – which creates a `Node` object and dumps it to the console – we get the following output instead. As expected, members that were previously `int` are now `double`.



```
C:\dev\FlexibleReflection\build\Debug\FlexibleReflection.exe
Node {
  key = std::string{"apple"}
  value = double{3}
  children = std::vector<Node>{
    [0] Node {
      key = std::string{"banana"}
      value = double{7}
      children = std::vector<Node>{}
    }
    [1] Node {
      key = std::string{"cherry"}
      value = double{11}
      children = std::vector<Node>{}
    }
  }
}
```

How It Works

In this system, the type descriptor of every primitive “built-in” type – whether it's `int`, `double`, `std::string` or something else – is found using the `getPrimitiveDescriptor<>()` function template, declared in `Reflect.h`:

```
// Declare the function template that handles primitive types such as int, std::string, etc.:
template <typename T>
TypeDescriptor* getPrimitiveDescriptor();
```

That's the primary template. The primary template is not *defined* anywhere – only *declared*. When `Main.cpp` compiles, the compiler happily generates a call to `getPrimitiveDescriptor<double>()` without knowing its definition – just like it would for any

other external function. Of course, the linker expects to find the definition of this function at link time. In the above example, the function is defined in `Primitives.cpp`.

The nice thing about this approach is that `getPrimitiveDescriptor<>()` can be specialized for any C++ type, and those specializations can be placed in any `.cpp` file in the program. (They don't all have to go in `Primitives.cpp`!) For example, in my [custom game engine](#), the graphics library specializes it for `VertexBuffer`, a class that manages OpenGL vertex buffer objects. As far as the reflection system is concerned, `VertexBuffer` is a built-in type. It can be used as a member of any class/struct, and reflected just like any other member that class/struct.

Be aware, however, that when you specialize this template in an arbitrary `.cpp` file, there are [rules](#) that limit the things you're allowed to do in the same `.cpp` file – though the compiler [may or may not complain](#).

Adding Support for `std::unique_ptr<>`

Let's [change the definition of `Node`](#) again so that it contains a [`std::unique_ptr<>`](#) instead of a `std::vector<>`.

```
struct Node {
    std::string key;
    double value;
    std::unique_ptr<Node> next;

    REFLECT()           // Enable reflection for this type
};
```

This time, we'll have to [initialize the `Node` object](#) differently:

```
// Create an object of type Node
Node node = {
    "apple",
    5,
    std::unique_ptr<Node>{new Node{
        "banana",
        7,
        std::unique_ptr<Node>{new Node{
            "cherry",
            11,
            nullptr
        }}
    }}
};
```

The [block of macros](#) needs to be updated, too:

```
// Define Node's type descriptor
REFLECT_STRUCT_BEGIN(Node)
REFLECT_STRUCT_MEMBER(key)
REFLECT_STRUCT_MEMBER(value)
REFLECT_STRUCT_MEMBER(next)
REFLECT_STRUCT_END()
```

If we build the sample project at this point, we'll encounter a link error, as before.

```
error: unresolved external symbol "reflect::getPrimitiveDescriptor<std::unique_ptr<Node>>()"
```

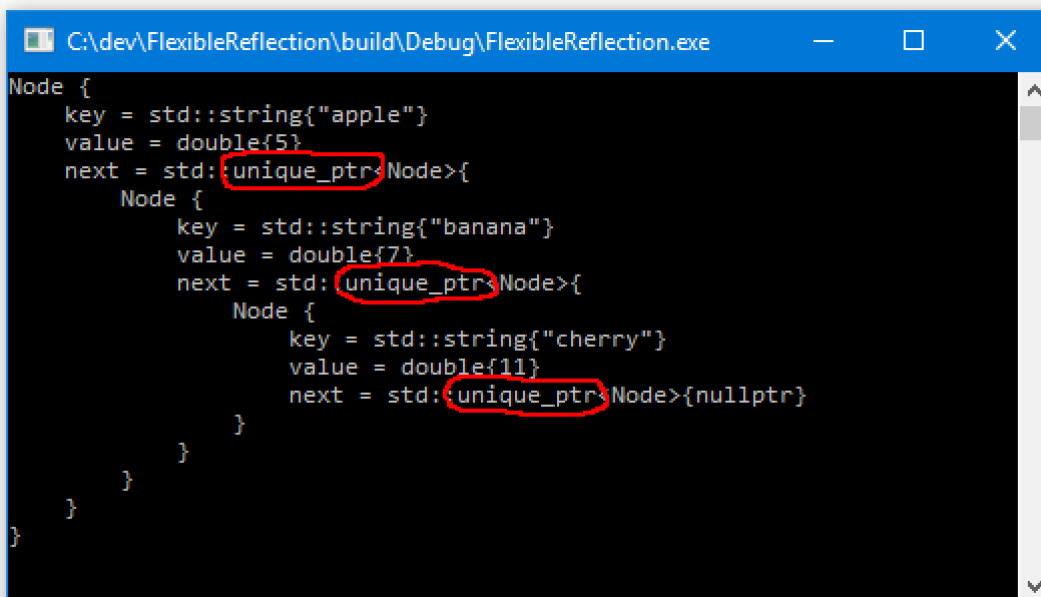
That's because the system doesn't support `std::unique_ptr<>` yet – no surprise there. We want the system to consider `std::unique_ptr<>` a built-in type. Unlike `double`, however, `std::unique_ptr<>` is not a primitive type; it's a template type. In this example, we've instantiated `std::unique_ptr<>` for `Node`, but it could be instantiated for an unlimited number of other types. Each instantiation should have its own type descriptor.

The system looks for `std::unique_ptr<Node>`'s type descriptor the same way it looks for every type descriptor: through the [TypeResolver<>](#) class template. By default, `TypeResolver<>::get()` tries to call `getPrimitiveDescriptor<>()`. We'll override that behavior by writing a [partial specialization](#) instead:

```
// Partially specialize TypeResolver<> for std::unique_ptr<>:
template <typename T>
class TypeResolver<std::unique_ptr<T>> {
public:
    static TypeDescriptor* get() {
        static TypeDescriptor_StdUniquePtr typeDesc{(T*) nullptr};
        return &typeDesc;
    }
};
```

In this partial specialization, `get()` constructs a new kind of type descriptor: `TypeDescriptor_StdUniquePtr`. Whenever the system looks for a type descriptor for `std::unique_ptr<T>` – for some type `T` – the compiler will instantiate a copy of the above `get()`. Each copy of `get()` will return a different type descriptor for each `T`, but the same type descriptor will always be returned for the same `T`, which is exactly what we want.

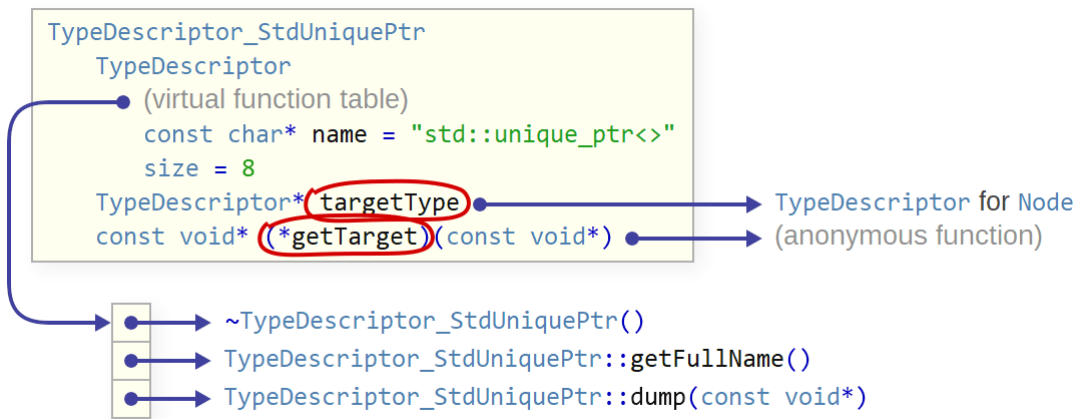
I've implemented full support for `std::unique_ptr<>` in a [separate branch on GitHub](#). The partial specialization is [located in Reflect.h](#) so that it's visible from every source file that needs it. With proper support in place, the sample project successfully dumps our updated `Node` object to the console.



```
C:\dev\FlexibleReflection\build\Debug\FlexibleReflection.exe
Node {
  key = std::string{"apple"}
  value = double{5}
  next = std::unique_ptr<Node>{
    Node {
      key = std::string{"banana"}
      value = double{7}
      next = std::unique_ptr<Node>{
        Node {
          key = std::string{"cherry"}
          value = double{11}
          next = std::unique_ptr<Node>{nullptr}
        }
      }
    }
  }
}
```

How It Works

In memory, the type descriptor for `std::unique_ptr<Node>` looks like this. It's an object of type `TypeDescriptor_StdUniquePtr`, a subclass of `TypeDescriptor` that holds two extra member variables:



Of those two member variables, the more mysterious one is `getTarget`. `getTarget` is a pointer to a kind of helper function. It points to an anonymous function that, at runtime, will dereference a particular specialization of `std::unique_ptr<>`. To understand it better, let's see how it gets initialized.

Here's the constructor for `TypeDescriptor_StdUniquePtr`. It's actually a [template constructor](#), which means that the compiler will instantiate a new copy of this constructor each time it's called with a different template parameter (specified via the dummy argument). It's called from the partial specialization of `TypeDescriptor` we saw earlier.

```
struct TypeDescriptor_StdUniquePtr : TypeDescriptor {
    TypeDescriptor* targetType;
    const void* (*getTarget)(const void*);

    // Template constructor:
    template <typename TargetType>
    TypeDescriptor_StdUniquePtr(TargetType* /* dummy argument */)
        : TypeDescriptor{"std::unique_ptr<>", sizeof(std::unique_ptr<TargetType>)},
          targetType{TypeResolver<TargetType>::get()} {
        getTarget = [] (const void* uniquePtrPtr) -> const void* {
            const auto& uniquePtr = *(const std::unique_ptr<TargetType>*) uniquePtrPtr;
            return uniquePtr.get();
        };
    }
    ...
}
```

Things get a little complex here, but as you can hopefully see, `getTarget` is initialized to a (captureless) [lambda expression](#). Basically, `getTarget` points to an anonymous function that casts its argument to a `std::unique_ptr<>` of the expected type, then dereferences it using [std::unique_ptr<>::get\(\)](#). The anonymous function takes a `const void*` argument because the struct `TypeDescriptor_StdUniquePtr` can be used to describe *any* specialization of `std::unique_ptr<>`. The function itself knows which specialization to expect.

Moreover, because the lambda expression is evaluated inside a template constructor, the compiler will generate a *different* anonymous function for each specialization of `std::unique_ptr<>`. That's important, because we don't know how `std::unique_ptr<>` is implemented by the standard library. All we can do is generate these anonymous functions to help us deal with every possible specialization.

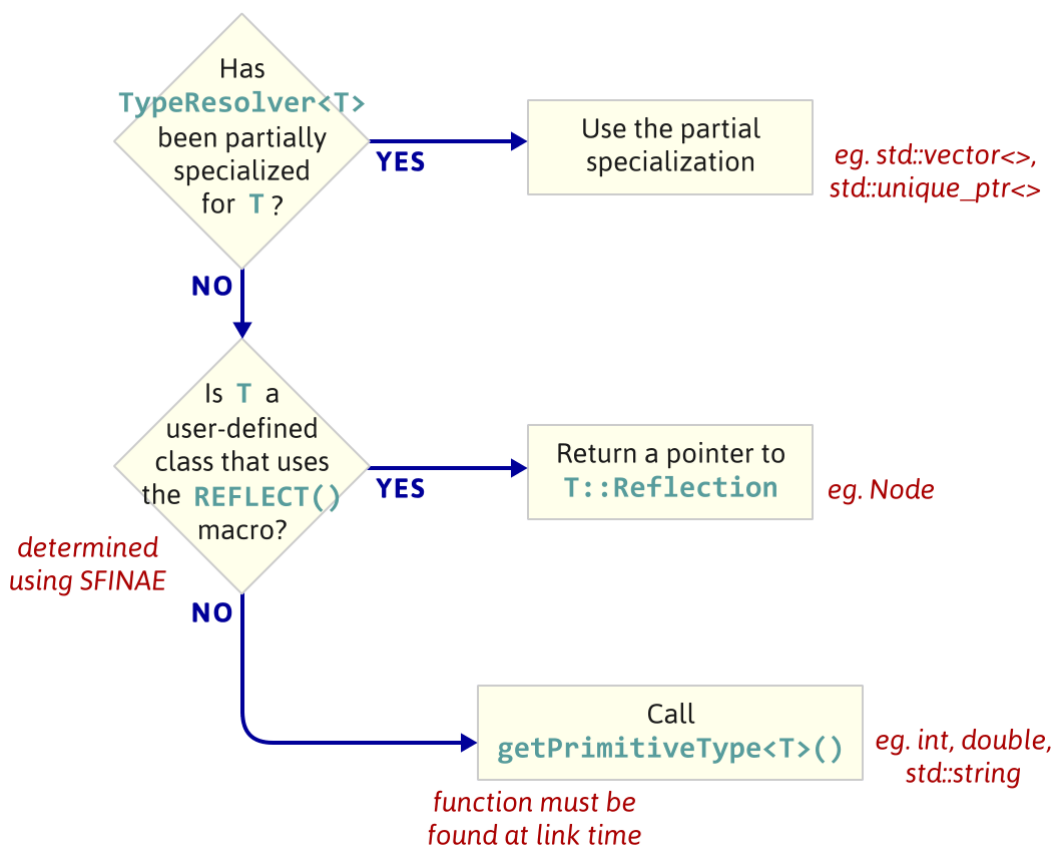
With all of that in place, the implementation of `TypeDescriptor_StdUniquePtr::dump()`, which helps dump the object to the console, is much more straightforward. You can [view the implementation here](#). It's written in a generic way: The same function is used by all

`std::unique_ptr<>` type descriptors, using `getTarget` to handle the differences between specializations.

Incidentally, `TypeDescriptor_StdVector` is implemented in much the same way as `TypeDescriptor_StdUniquePtr`. The main difference is that, instead of having one anonymous helper function, `TypeDescriptor_StdVector` has two: one that returns the number of elements in a `std::vector<>`, and another that returns a pointer to a specific element. You can see how both helper functions are initialized [here](#).

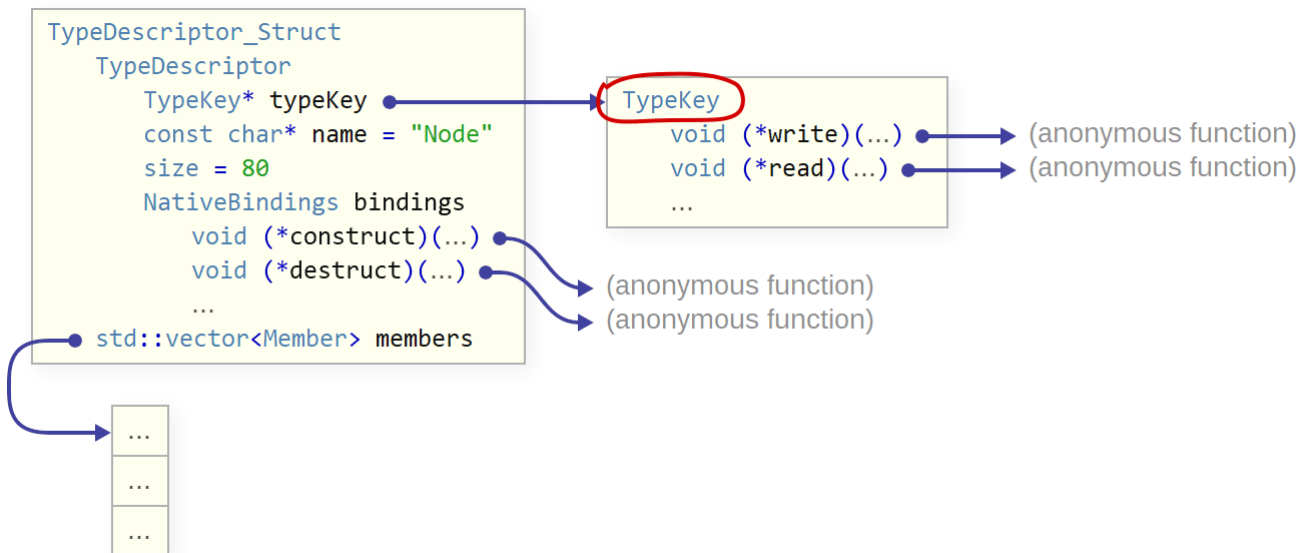
Summary of How Type Descriptors are Found

As we've seen, a call to `reflect::TypeResolver<T>::get()` will return a type descriptor for any reflected type `T`, whether it's a built-in primitive type, a built-in template type, or a user-defined class or struct. In summary, the compiler resolves the call as follows:



Further Improvements

In the [FlexibleReflection](#) sample project, type descriptors are useful because they implement virtual functions like `getFullName()` and `dump()`. In my real reflection system, however, type descriptors are mainly used to serialize to (and from) a custom binary format. Instead of virtual functions, the serialization API is exposed through a pointer to an explicit table of function pointers. I call this table the `TypeKey`. For example, the real `TypeDescriptor_Struct` looks something like this:



One benefit of the `TypeKey` object is that its address serves as an identifier for the *kind* of type descriptor it is. There' s no need to define a separate enum. For example, all `TypeDescriptor_Struct` objects point to the same `TypeKey`.

You can also see that the type descriptor has function pointers to help `construct` and `destruct` objects of the underlying type. These functions, too, are generated by lambda expressions inside a function template. The serializer uses them to create new objects at load time. You can even add helper functions to manipulate dynamic arrays and maps.

Perhaps the biggest weakness of this reflection system is that it relies on preprocessor macros. In particular, a block of `REFLECT_STRUCT_*` macros is needed to reflect the member variables of a class, and it' s easy to forget to keep this block of macros up-to-date. To prevent such mistakes, you could collect a list of class members automatically using [libclang](#), a custom header file parser (like [Unreal](#)), or a data definition language (like [Qt moc](#)). For my part, I' m using a simple approach: A small Python script reads class and member names from clearly-marked sections in each header file, then injects the corresponding `REFLECT_STRUCT_*` macros into the corresponding `.cpp` files. It took a single day to write this script, and it runs in a fraction of a second.

I developed this reflection system for [a custom game engine I' m working on](#), all to create a little game called **Hop Out**. The reflection system has proven invaluable so far. It' s used heavily in both my serializer and 3D renderer, and it' s currently reflecting 134 classes that are constantly in flux, with more being added all the time. I doubt I could manage all that data without this system!

[« A Flexible Reflection System in C++: Part 1 A New Cross-Platform Open Source C++ Framework »](#)

Comments (9)

Commenting Disabled

Further commenting on this page has been disabled by the blog admin.

 [Ben Hymers](#) · 203 weeks ago

Do you have a nice way to handle reference cycles? (relevant for save games probably)

Reply [2 replies](#) · active 203 weeks ago

 [Jeff Preshing](#) · 203 weeks ago

Well, this is only the reflection system -- it's agnostic as to whether there are cycles in the data it describes. The type descriptors themselves are allowed to have cycles, though, and it's demonstrated in the sample project (eg. Node contains a `std::unique_ptr<Node>`).

If you're wondering whether my serializer can handle cycles: I make a distinction between "strong" and "weak" pointers, and in fact, I don't even serialize weak pointers yet. When the time comes, I plan to allow cycles in the weak pointers but not the strong ones. Is this a problem you've had to solve?

Reply

 [Ben Hymers](#) · 203 weeks ago

Ah sorry, I mean in [de]serialisation rather than reflection - e.g. `TypeDescriptor::dump`. Like if you had two Nodes with pointers to each other, currently it would recurse forever. (not that you could do that when using `unique_ptr` though!)

This is something that I'm curious about how to implement in a nice simple way - it seems to end up adding a large amount of complexity to otherwise simple elegant solutions like this!

Reply

Nir · 203 weeks ago

I'm not sure why you need to have 3 levels of lookup: specialization, reflection member, and then primitives. Primitives can be handled by specialization, so it would simplify things to just have two steps. Specialization + member (or better, specialization + ADL lookup call; this allows users with access to a type's namespace but not the source to provide a function) is a common choice for allowing customization of something for types, e.g. `nlohman::json` uses this technique.

What's the advantage of having a special level of lookup when primitives can be handled by specialization?

Reply [1 reply](#) · active 203 weeks ago

 [Jeff Preshing](#) · 203 weeks ago

You're right. I experimented a little more and primitives can indeed be handled by leaving the primary `TypeResolver::get()` undefined (declaration only), then specializing it in arbitrary .cpp files. At the same time, the SFINAE stuff (detecting a class member) can be eliminated, too -- also by specializing `TypeResolver::get()`. It's definitely simpler. I'll try it out in my real system and see if I'm forgetting anything.

I made an experimental branch. If you can, check it out and let me know if you have any other feedback: <https://github.com/preshing/FlexibleReflection/co...>

[*** Edit: On second thought, this experimental branch is not a great idea. Eliminating the SFINAE will limit the places where it's [legally allowed](#) to put the `REFLECT_STRUCT_*` macros. But I don't know how to implement your suggestion otherwise. If you could fork the repo to demonstrate, that would be cool.]


Reply

Amit Karim · 203 weeks ago

Hi Jeff, it's a very cool post. I had a question:

How do you go about accessing your data through the type descriptors? `dump()` is neat but not terribly generic. What about for serialization, or building any kind of generic view onto this data (a property grid for example).

Reply [2 replies](#) · active 203 weeks ago

 [Jeff Preshing](#) · 203 weeks ago

That's a whole other topic. To make a long story short, I expose functionality through the type descriptors the same way that `getFullName()` and `dump()` are exposed here. All the information you need is in the type descriptors -- it's just a matter of exposing what you need, via a generic API (where objects, if passed around, are usually passed around as `void*`), to accomplish what you want.

Reply

Amit Karim · 203 weeks ago

I hope that's part 3!

Reply

Dihara Wijetunga · 193 weeks ago

Really inspiring post! Made me want to re-think the serialization system in my engine too. I have one question though. How do you handle dynamic arrays where the size of the array is in another variable of the same struct. Like for example:

```
struct Foo {  
    int size;  
    float* array;  
}
```

How would you retrieve the size of the array through the reflection API in a case like this? It's different from the `std::vector` case because you need access to the parent object in a case like that. How do you handle this?

Reply

[Check out Plywood, a cross-platform, open source C++ framework:](#)



Recent Posts

- [How C++ Resolves a Function Call](#)
- [Flap Hero Code Review](#)
- [A Small Open Source Game In C++](#)
- [Automatically Detecting Text Encodings in C++](#)
- [I/O in Plywood](#)
- [A New Cross-Platform Open Source C++ Framework](#)
- [A Flexible Reflection System in C++: Part 2](#)
- [A Flexible Reflection System in C++: Part 1](#)
- [How to Write Your Own C++ Game Engine](#)
- [Can Reordering of Release/Acquire Operations Introduce Deadlock?](#)
- [Here's a Standalone Cairo DLL for Windows](#)
- [Learn CMake's Scripting Language in 15 Minutes](#)
- [How to Build a CMake-Based Project](#)
- [Using Quiescent States to Reclaim Memory](#)
- [Leapfrog Probing](#)
- [A Resizable Concurrent Map](#)

- [New Concurrent Hash Maps for C++](#)
- [You Can Do Any Kind of Atomic Read-Modify-Write Operation](#)
- [Safe Bitfields in C++](#)
- [Semaphores are Surprisingly Versatile](#)

Copyright © 2021 Jeff Preshing - Powered by [Octopress](#)