



Preshing on Programming

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

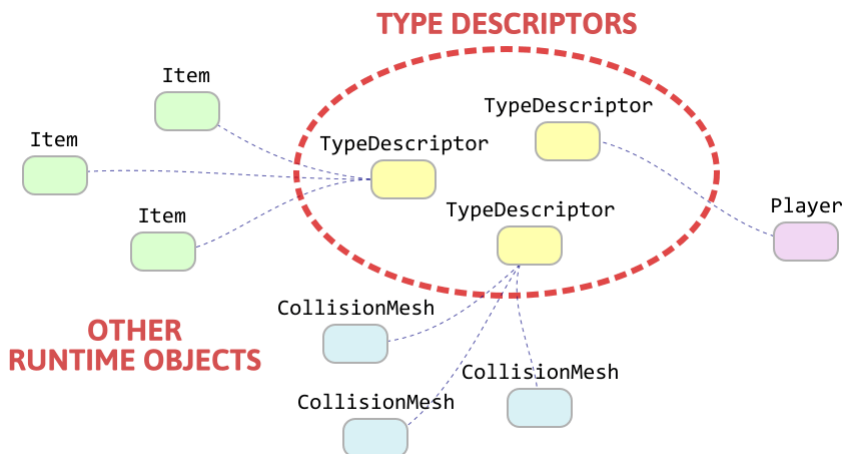
Navigate... ▼

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

Jan 16, 2018

A Flexible Reflection System in C++: Part 1

In this post, I'll present a small, flexible system for **runtime reflection** using C++11 language features. This is a system to generate [metadata](#) for C++ types. The metadata takes the form of `TypeDescriptor` objects, created at runtime, that describe the structure of other runtime objects.



I'll call these objects **type descriptors**. My initial motivation for writing a reflection system was to support **serialization** in my [custom C++ game engine](#), since I have very specific needs. Once that worked, I began to use runtime reflection for other engine features, too:

- **3D rendering:** Every time the game engine draws something using OpenGL ES, it uses reflection to pass uniform parameters and describe vertex formats to the API. It makes graphics programming much more productive!
- **Importing JSON:** The engine's asset pipeline has a generic routine to synthesize a C++ object from a JSON file and a type descriptor. It's used to import 3D models, level definitions and other assets.

This reflection system is based on preprocessor macros and templates. C++, at least in its current form, was not designed to make runtime reflection easy. As anyone who's written one knows, it's tough to design a reflection system that's easy to use, easily extended, and that actually works. I was burned many times by obscure language rules, order-of-initialization bugs and corner cases before settling on the system I have today.

To illustrate how it works, I've published a sample project [on GitHub](#):

 [preshing / FlexibleReflection](#)

This sample doesn't actually use my game engine's reflection system. It uses a tiny reflection system of its own, but the most interesting part – the way type descriptors are **created**, **structured** and **found** – is almost identical. That's the part I'll focus on in this post. In the next post, I'll discuss how the system can be extended.

This post is meant for programmers who are interested in how to *develop* a runtime reflection system, not just use one. It touches on many advanced features of C++, but the sample project is only 242 lines of code, so hopefully, with some persistence, any determined C++ programmer can follow along. If you're more interested in using an existing solution, take a look at [RTTR](#).

Demonstration

In `Main.cpp`, the sample project defines a struct named `Node`. The `REFLECT()` macro tells the system to enable reflection for this type.

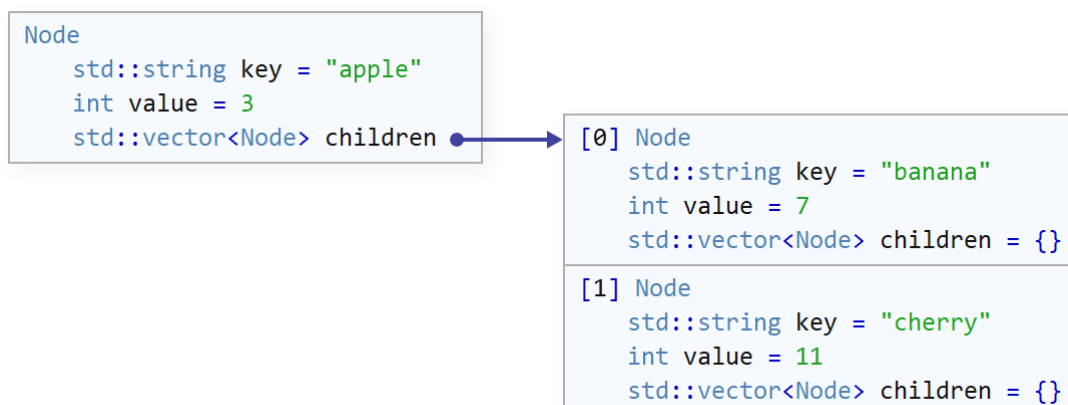
```
struct Node {
    std::string key;
    int value;
    std::vector<Node> children;

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

At runtime, the sample creates an object of type `Node`.

```
// Create an object of type Node
Node node = {"apple", 3, {"banana", 7, {}}, {"cherry", 11, {}}};
```

In memory, the `Node` object looks something like this:



Next, the sample finds `Node`'s type descriptor. For this to work, the following macros must be placed in a `.cpp` file somewhere. I put them in `Main.cpp`, but they could be placed in any file from which the definition of `Node` is visible.

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

`Node`'s member variables are now said to be **reflected**.

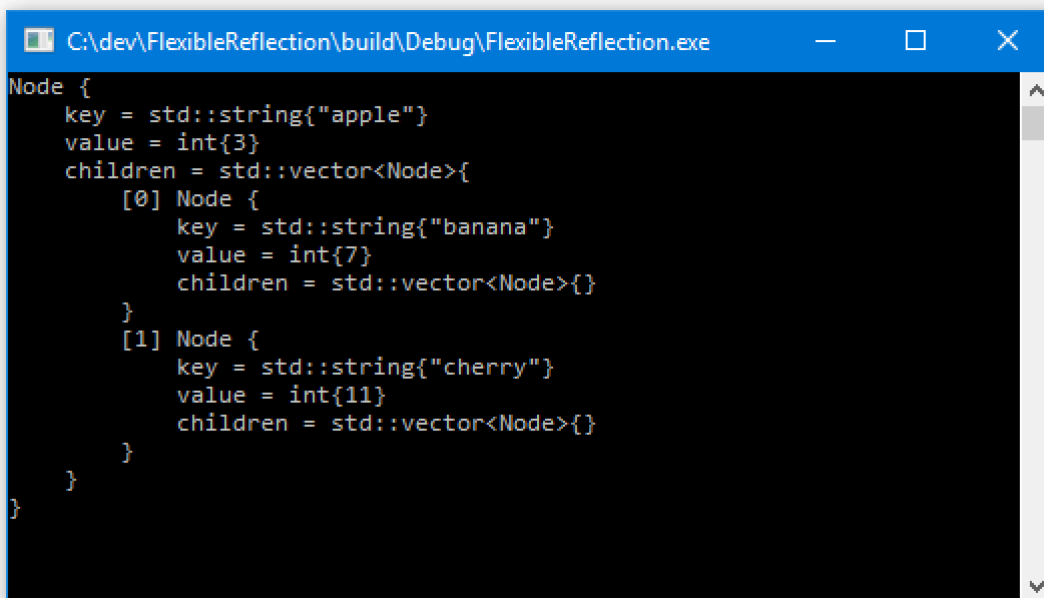
A pointer to `Node`'s type descriptor is obtained by calling `reflect::TypeResolver<Node>::get()`:

```
// Find Node's type descriptor
reflect::TypeDescriptor* typeDesc = reflect::TypeResolver<Node>::get();
```

Having found the type descriptor, the sample uses it to dump a description of the `Node` object to the console.

```
// Dump a description of the Node object to the console
typeDesc->dump(&node);
```

This produces the following output:



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

How the Macros Are Implemented

When you add the `REFLECT()` macro to a struct or a class, it declares two additional static members: `Reflection`, the struct's type descriptor, and `initReflection`, a function to initialize it. Effectively, when the macro is expanded, the complete `Node` struct looks like this:

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

  // Declare the struct's type descriptor:
  static reflect::TypeDescriptor_Struct Reflection;
```

```
// Declare a function to initialize it:
static void initReflection(reflect::TypeDescriptor_Struct*);
};
```

Similarly, the block of `REFLECT_STRUCT_*` macros in `Main.cpp` look like this when expanded:

```
// Definition of the struct's type descriptor:
reflect::TypeDescriptor_Struct Node::Reflection{Node::initReflection};

// Definition of the function that initializes it:
void Node::initReflection(reflect::TypeDescriptor_Struct* typeDesc) {
    using T = Node;
    typeDesc->name = "Node";
    typeDesc->size = sizeof(T);
    typeDesc->members = {
        {"key", offsetof(T, key), reflect::TypeResolver<decltype(T::key)>::get()},
        {"value", offsetof(T, value), reflect::TypeResolver<decltype(T::value)>::get()},
        {"children", offsetof(T, children), reflect::TypeResolver<decltype(T::children)>::get()},
    };
}
```

Now, because `Node::Reflection` is a static member variable, its constructor, which accepts a pointer to `initReflection()`, is automatically called at program startup. You might be wondering: Why pass a function pointer to the constructor? Why not pass an [initializer list](#) instead? The answer is because the body of the function gives us a place to declare a C++11 [type alias](#): `using T = Node`. Without the type alias, we'd have to pass the identifier `Node` as an extra argument to every `REFLECT_STRUCT_MEMBER()` macro. The macros wouldn't be as easy to use.

As you can see, inside the function, there are three additional calls to `reflect::TypeResolver<>::get()`. Each one finds the type descriptor for a reflected member of `Node`. These calls use C++11's [decltype specifier](#) to automatically pass the correct type to the `TypeResolver` template.

Finding TypeDescriptors

(Note that everything in this section is defined in the `reflect` namespace.)

`TypeResolver` is a **class template**. When you call `TypeResolver<T>::get()` for a particular type `T`, the compiler instantiates a function that returns the corresponding `TypeDescriptor` for `T`. It works for reflected structs as well as for every reflected member of those structs. By default, this happens through the primary template, highlighted below.

By default, if `T` is a struct (or a class) that contains the `REFLECT()` macro, like `Node`, `get()` will return a pointer to that struct's `Reflection` member – which is what we want. For every other type `T`, `get()` instead calls `getPrimitiveDescriptor<T>` – a **function template** that handles primitive types such as `int` or `std::string`.

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

// A helper class to find TypeDescriptors in different ways:
struct DefaultResolver {
    ...

    // This version is called if T has a static member variable named "Reflection":
    template <typename T, /* SFINAE stuff here */>
```

```

static TypeDescriptor* get() {
    return &T::Reflection;
}

// This version is called otherwise:
template <typename T, /* SFINAE stuff here */>
static TypeDescriptor* get() {
    return getPrimitiveDescriptor<T>();
}
};

// This is the primary class template for finding all TypeDescriptors:
template <typename T>
struct TypeResolver {
    static TypeDescriptor* get() {
        return DefaultResolver::get<T>();
    }
};

```

This bit of compile-time logic – generating different code depending on whether a static member variable is present in `T` – is achieved using [SFINAE](#). I omitted the SFINAE code from the above snippet because, quite frankly, it's ugly. You can check the actual implementation [in the source code](#). Part of it could be rewritten more elegantly using [if constexpr](#), but I'm targeting C++11. Even then, the part that detects whether `T` has a specific member variable will remain ugly, at least until C++ adopts [static reflection](#). In the meantime, however – it works!

The Structure of TypeDescriptors

In the sample project, every `TypeDescriptor` has a name, a size, and a couple of virtual functions:

```

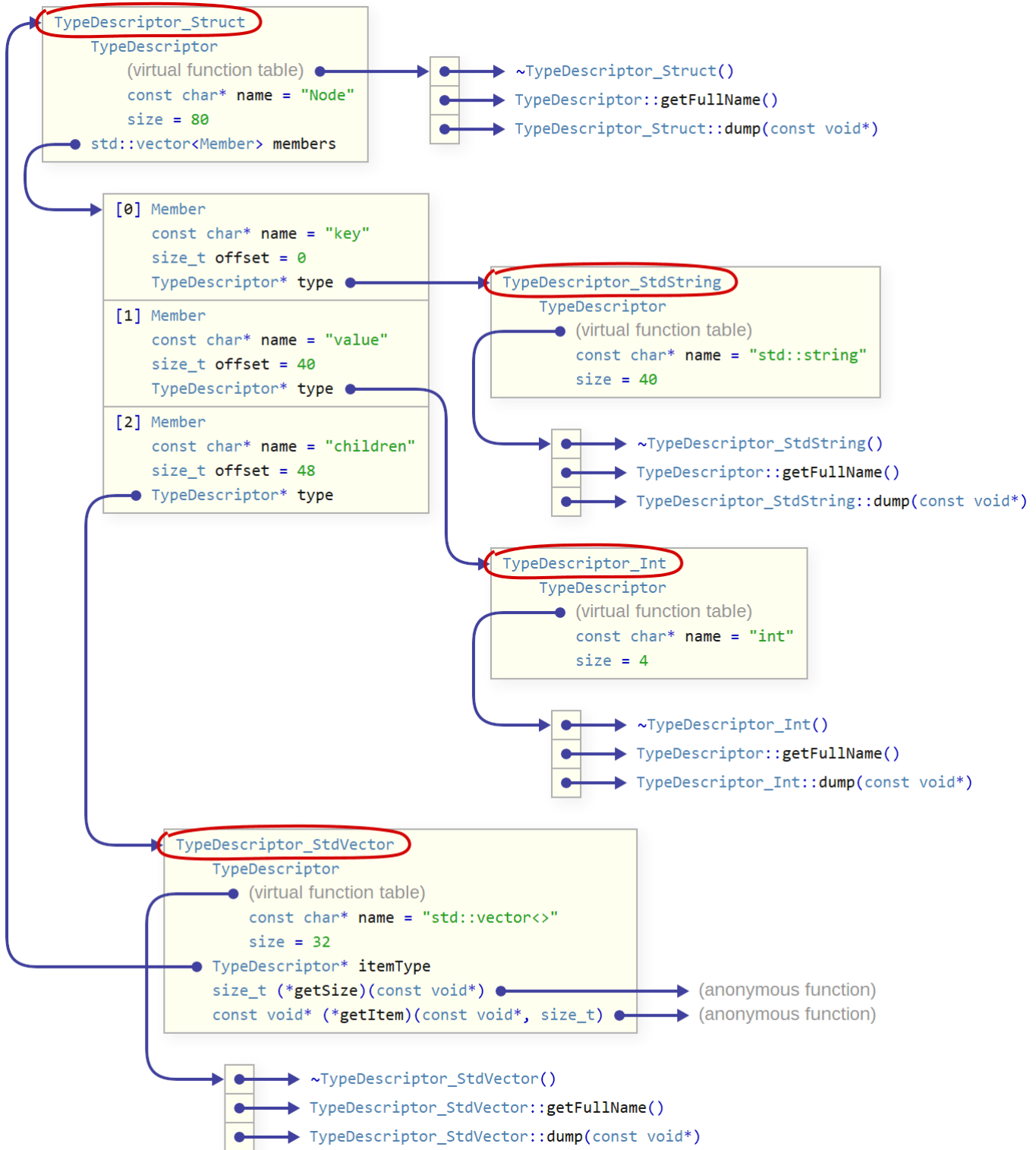
struct TypeDescriptor {
    const char* name;
    size_t size;

    TypeDescriptor(const char* name, size_t size) : name{name}, size{size} {}
    virtual ~TypeDescriptor() {}
    virtual std::string getFullName() const { return name; }
    virtual void dump(const void* obj, int indentLevel = 0) const = 0;
};

```

The sample project never creates `TypeDescriptor` objects directly. Instead, the system creates objects of types derived from `TypeDescriptor`. That way, every type descriptor can hold extra information depending on, well, the *kind* of type descriptor it is.

For example, the actual type of the object returned by `TypeResolver<Node>::get()` is `TypeDescriptor_Struct`. It has one additional member variable, `members`, that holds information about every reflected member of `Node`. For each reflected member, there's a pointer to another `TypeDescriptor`. Here's what the whole thing looks like in memory. I've circled the various `TypeDescriptor` subclasses in red:



At runtime, you can get the full name of any type by calling `getFullName()` on its type descriptor. Most subclasses simply use the base class implementation of `getFullName()`, which returns `TypeDescriptor::name`. The only exception, in this example, is `TypeDescriptor_StdVector`, a subclass that describes `std::vector<>` specializations. In order to return a full type name, such as `"std::vector<Node>"`, it keeps a pointer to the type descriptor of its item type. You can see this in the above memory diagram: There's a `TypeDescriptor_StdVector` object whose `itemType` member points all the way back to the type descriptor for `Node`.

Of course, type descriptors only describe *types*. For a complete description of a runtime object, we need both a type descriptor and a pointer to the object itself.

Note that `TypeDescriptor::dump()` accepts a pointer to the object as `const void*`. That's because the abstract `TypeDescriptor` interface is meant to deal with *any* type of object. The subclassed implementation knows what type to expect. For example, here's the implementation of `TypeDescriptor_StdString::dump()`. It casts the `const void*` to `const std::string*`.

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

You might wonder whether it's safe to cast `void` pointers in this way. Clearly, if an invalid pointer is passed in, the program is likely to crash. That's why, in my game engine, objects represented by `void` pointers always travel around with their type descriptors in pairs. By representing objects this way, it's possible to write many kinds of generic algorithms.

In the sample project, dumping objects to the console is the only functionality implemented, but you can imagine how type descriptors could serve as a framework for serializing to a binary format instead.

In the next post, I'll explain how to add built-in types to the reflection system, and what the "anonymous functions" are for in the above diagram. I'll also discuss other ways to extend the system.

[« How to Write Your Own C++ Game Engine A Flexible Reflection System in C++: Part 2 »](#)

Comments (10)

Commenting Disabled

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



Nir · 204 weeks ago

If you're already using macros, it is IMHO a real shame a huge shame to repeat the fields, this is a maintenance burden and can easily lead to silent errors. You should be able to do something like: `DECLARE_STRUCT(Node, (std::string, key), ...)`. Or, `class Node { DECLARE_MEMBERS((std::string, key), ...);`. If you are fine repeating the fields, then you can just omit macros entirely. Instead write a free function `auto reflect(Node&)`, which returns a tuple of `std::pair<const char&, T&>` (where T is the type of each field in turn). You can do everything you described here with such a function (unless I am missing something). Such a system implements compile time reflection, which avoids costs incurred by `TypeDescriptor`. If you need to, you can also implement `TypeDescriptor` on top of such a thing (with no additional per-type boilerplate).

Reply [4 replies](#) · active 204 weeks ago



Jeff Preshing · 204 weeks ago

Yeah, that's always been the problem with macro-based systems. The advantage is they're easy to integrate. I've been hit by those silent errors, but it hasn't been *that* much of a maintenance burden. Mistakes are easy enough to discover and fix. Still, I'm about to add reflection to a whole bunch of classes that don't currently use it, so I'll likely implement a build script that generates the macros in a way that keeps things in sync.

For your second suggestion, doesn't that just move the boilerplate from the macros to a function? The nice thing about macros is you can use the stringify `#` token.

Reply



Nir · 204 weeks ago

Sorry, re your first point I don't think I understand. When you say "that's always been the problem with macro based systems", it seems like you are saying that you always have to repeat the fields with macro based reflection. My point is that it actually is not necessary, you can easily write a macro that gives you reflection without ever having to repeat the field names. Obviously, it would be intrusive, but this is mostly for classes you are writing yourself so this is ok, plus it is always possible to design the reflection system so that you can fall back to repetition for classes you don't control.

You can restrict yourself to only using the macro to get the stringized name and field without repetition, but otherwise do everything in functions. Working example: <http://coliru.stacked-crooked.com/a/25638f2ebc642...>. This approach has quite a few advantages in terms of type safety, extensibility, and performance.

Obviously this is missing some stuff that you have, but I can't see any feature you're needing here that can't be easily provided in this style; I'm curious if I'm missing some trade-off here. (I use a polymorphic lambda, which is a C++14 feature, but obviously you could just write the same thing out by hand to implement e.g. your dump function).

Reply



[Jeff Preshing](#) · 204 weeks ago

But you still repeat the names of the fields: on line 20. Cool example though.

Reply



Nir's · 204 weeks ago

My point is that you can either almost entirely avoid macros (the link I sent), or you can avoid field repetition. In your solution, you have both lots of macros and field repetition. I don't have reflective struct code that I can post handy, but here's an example of a reflective enum: https://github.com/quicknir/wise_enum. You use a macro to declare it, but you never have to repeat the names of the enumerators, even though the enum can be converted to/from string.

Reply



Sarfaraz Nawaz · 204 weeks ago

Here is my approach to define objects on the fly, while having ability to introspect about its members, names, types and so on.

<https://github.com/snawaz/cson>

Once we have such introspectable objects, we can write many generic algorithms for it, such as equals(), serialize(), deserialize() and so on. It is just a proof-of-concept project.

Reply



Ethan · 203 weeks ago

Have you considered using <https://github.com/foonathan/cppast> to automatically generate type data?

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



[Jeff Preshing](#) · 203 weeks ago

Nope, hadn't heard of it before, thanks! For now I'm using a simple Python script to scrape headers, as mentioned in part 2.

Reply



Beautiful · 203 weeks ago

Could you please tell me what software you're using to generate these diagrams?

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



[Jeff Preshing](#) · 203 weeks ago

The memory diagrams are drawn using a custom layout software I wrote using [Cairo](#). The rest is Inkscape.

Reply

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



Plywood

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](#)