Declarative class authoring using `consteval` functions + reflection + generation (aka: Metaclasses for generative C++)

Document Number: **P0707 R5**
Date: 2024-10-12
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: SG7, EWG

Contents

1 What's new: Why this paper is now very short....2
2 Background motivation (largely repeated from R4)....3
3 Proposal: `class(xxx)` syntactic sugar on top of [P2996R5] and [P3294R1] 4
4 References....6

Abstract

The only way to make a language more powerful, but also make its programs simpler, is by *abstraction*: adding well-chosen abstractions that let programmers replace manual code patterns with saying directly what they mean. There are two major categories:

Elevate coding patterns/idioms into new abstractions built into the language. For example, in current C++, range-`for` lets programmers directly declare "for each" loops with compiler support and enforcement.

(major, this paper) Provide a new abstraction authoring mechanism so programmers can write new kinds of user-defined abstractions that encapsulate behavior. In current C++, the function and the `class` are the two mechanisms that encapsulate user-defined behavior. In this paper, type metafunctions (aka metaclass functions) enable defining categories of `class`es that have common defaults and generated functions, and formally expand C++'s type abstraction vocabulary beyond `class`/`struct`/`union`/`enum`.

Also, [cppfront] demonstrates working implementations of a set of common type metafunctions, many of which are common enough to consider for `std::`. This paper begins by demonstrating how to implement Java/C# `interface` as a 10-line C++ `std::` type metafunction – with the same usability, expressiveness, diagnostic quality, and performance of the built-in feature in such languages, where it is specified as ~20 pages of "standardese" text specification.

1 What's new: Why this paper is now very short

Previous versions of this paper have been 50+ pages long because previously it had to provide deep detail on:

- **Motivation for why we should add reflection and generation to C++.** However, that is now well in progress with [P2996R5] and [P3294R1] and similar papers, so convincing is no longer needed and that previous material can be replaced with a link to those papers.

- **Precise pseudocode examples for how reflection and generation would work**, in great detail so as to be convincing including because previously there was only a partial implementation (Lock3's Clang-based implementation) that could only compile a subset of the examples. However, now [cppfront] is available which implements all of the metafunctions proposed in earlier versions of this paper (and more) as code that works on all recent versions of MSVC, GCC, and Clang, so nearly all of the rest of the page count of this paper can be replaced with a link to cppfront's reflect.h2 header (note: as of this writing, the first ~600 lines are the reflection+generation API, and the metafunctions appear after that) — for the ordinary ISO C++ reflection+generation code, see the reflect.h generated header.

"Metaclass" type metafunctions currently implemented and working in cppfront

`interface`	An abstract class having only pure virtual functions
`polymorphic_base`	A pure polymorphic base type that is not copyable or movable, and whose destructor is either public+virtual or protected+nonvirtual
`ordered`	A totally ordered type with `operator<=>` that implements `strong_ordering`. Also: `weakly_ordered`, `partially_ordered`
`copyable`	A type that has copy/move construction/assignment
`basic_value`	A `copyable` type that has public default construction and destruction (generated if not user-written) and no protected or virtual functions
`value`	An `ordered basic_value`. Also: `weakly_ordered_value`, `partially_ordered_value`
`struct`	A `basic_value` with all public members, no virtuals, no custom assignment
`enum`	An `ordered basic_value` with all public values
`flag_enum`	An `ordered basic_value` with all public values, and bitwise sets/tests
`union`	A safe (tagged) union with names (unlike `std::variant`)
`regex`	A CRTE-style compile time regex, but using reflection+generation (Max Sagebaum)
`print`	Print the reflection as source code at compile time

All that's left of P0707R4 and earlier is just two things:

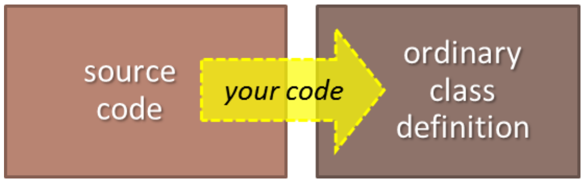
- **(this R5 paper) "`class(xxx)`" 2-line language sugar.** Proposing the `class(xxx,yyy)` syntax as a two-line syntactic sugar for what is already possible with [P2996R5] and [P3294R1]. This is the syntax that SG7 previously gave guidance to pursue (instead of the original proposed syntax that used a `$`).

- **(future) Library proposal paper for the `consteval` functions in the above box (a future version of this paper, or a separate library proposal paper).** Proposing the set of `consteval` functions (the type metafunctions, aka metaclasses) already implemented and that are general enough to add to the standard (pretty much everything in the box above). Even that will be a short paper because it will just propose a few more `consteval` library interfaces (as usual, only the interfaces and not their code implementations).

2 Background motivation (largely repeated from R4)

This paper assumes that C++ adds support for static reflection and compile-time programming to C++, and focuses on the next-level layer of abstraction we could build on top of that. This paper hopes to provide "what we want to be able to write" use cases for using features in the related work, and this paper's prototype implementation also implements most of those other proposals since they are necessary for metaclass functions.

Type metafunctions (aka metaclass functions) let programmers write a new kind of efficient abstraction: a user-defined named subset of `class`es that share common characteristics, typically (but not limited to):



- defaults,
- generated functions, and
- constraints and other rules

by writing a custom transformation from normal C++ source code to a normal C++ class definition. Importantly, there is no type system bifurcation; the generated class is a normal `class`.

Primary goals:

- Expand C++'s abstraction vocabulary beyond `class`/`struct`/`union`/`enum` which are the type categories hardwired into the language.
- Enable providing longstanding best practices as reusable libraries instead of English guides/books, to have an easily adopted vocabulary (e.g., `interface`, `value`) instead of lists of rules to be memorized (e.g., remember this coding pattern to write an abstract base class or value type, relying on tools to find mistakes).
- Enable writing compiler-enforced patterns for any purpose: coding standards (e.g., many Core Guidelines "enforce" rules), API requirements (e.g., rules a class must follow to work with a hardware interface library, a browser extension, a callback mechanism), and any other pattern for classes.
- Enable writing many new "specialized types" features (e.g., as we did in C++11 with `enum class`) as ordinary library code instead of pseudo-English standardese, with equal usability and efficiency, so that they can be unit-tested and debugged using normal tools, developed/distributed without updating/shipping a new compiler, and go through LEWG/LWG as code instead of EWG/CWG as standardese. As a consequence, enable standardizing valuable extensions that we'd likely never standardize in the core language because they are too narrow (e.g., `interface`), but could readily standardize as a small self-contained library.
- Eliminate the need to invent non-C++ "side languages" and special compilers, such as Qt moc, COM MIDL, and C++/CX, to express the information their systems need but cannot be expressed in today's C++ (such as specialized types for properties, event callbacks, and similar abstractions).

Primary intended benefits:

- For users: Don't have to wait for a new compiler ⇒ can write "new class features" as ordinary libraries, that can be put in namespaces, shared as libraries and on GitHub, and so on like any other code.
- For standardization: More features as testable libraries ⇒ easier evolution, higher quality proposals. Common metaclasses (like common classes) can be standardized as `std::` libraries.
- For C++ implementations: Fewer new language features ⇒ less new compiler work and more capacity to improve tooling and quality for existing features. Over time, I hope we can deprecate and eventually remove many nonstandard extensions.

3 Proposal: `class(xxx)` syntactic sugar on top of [P2996R5] and [P3294R1]

[P2996R5] and [P3294R1] already support authoring a "prototype" type and then running a `consteval` function to reflect on that type and use the result to generate another version of the type.

```
//  Possible with [P2996R5] and [P3294R1]
namespace __prototype { class widget { /*...*/ }; }
consteval{ metafunc( ^^__prototype::widget ); }
```

The point here is that the programmer is authoring class `widget` to also enjoy the defaults, generated functions, and constraints and other rules provided by the compile-time function `metafunc`.

This paper proposes that the following be syntactic sugar for the above, so the user can say it directly:

```
//  Proposed in this paper to be sugar identical to the above
class(metafunc) widget{ /*...*/ };
```

It's "just" sugar, but has major benefits:

- **It's cleaner** and more directly expresses the intent that "I'm not just writing any kind of class, I'm writing *this particular* kind of class" without distraction.
- **It's more efficient** because by construction it's clear the prototype class will not be used after the `consteval` block ends, so the compiler can discard it.

3.1 Example

The usual starter example is `interface`.

Today, to write an "interface type" `IFoo` by hand, I have to write something like this, where the highlighted text is pure boilerplate because `class` doesn't give me the appropriate defaults for an "interface" type:

```
//  Today's C++: How to write an IFoo interface by hand
class IFoo {
public:
  virtual int f() = 0;
  virtual void g(std::string) = 0;
  virtual ~IFoo() = default;
  IFoo() = default;
  IFoo(IFoo const&) = delete;
  void operator=(IFoo const&) = delete;
};
```

Of course this has all the usual drawbacks: It's tedious because there's so much boilerplate, and it's error-prone because interface-specific rules aren't enforced (e.g., if I accidentally write a copy constructor, which makes no sense for this kind of abstract base class, the code still silently compiles).

Using [P2996R5] and [P3294R1], I can directly express my intent and write this much more simply as follows using the pattern I showed above **(Godbolt: godbolt.org/z/rvdabTb5M)**:

```
//  With [P2996R5] and [P3294R1] - godbolt.org/z/rvdabTb5M
namespace __proto {
  class IFoo {
    int  f();
    void g(std::string);
  };
}
consteval { interface(^^__proto::IFoo); }
```

With this paper, I can equivalently write this:

```
//  Same, with this paper
class(interface) IFoo {
    int  f();
    void g(std::string);
};
```

and it's even better, because:

- **It's cleaner.**
- **It's declarative** because I've declared my intent up front — the metaclass name is a Word of Power, a single name denoting a bundle of defaults, constraints, and generated functions I opt into (so I never need to `=delete` what is generated).
- **It's more efficient** because the compiler knows by construction that `__proto::IFoo` will not be needed anymore after this single use, so its AST and other information can be discarded.

3.1.1 For completeness: `interface` implementation

This is working code for P0707's original `interface`, in the EDG prototype of [P2996R5] and [P3294R1]:

```
// godbolt.org/z/rvdabTb5M
consteval auto make_interface_functions(info proto) -> info {
    info ret = ^^{};
    for (info mem : members_of(proto)) {
        if (is_nonspecial_member_function(mem)) {
            ret = ^^{
                \tokens(ret)
                virtual [:\(return_type_of(mem)):]
                    \id(identifier_of(mem)) (\tokens(parameter_list_of(mem))) = 0;
            };
        }
        else if (is_variable(mem)) {
           // --- reporting compile time errors not yet implemented ---
           // print_error( "interfaces may not contain data members" );
        }
        // etc. for other kinds of interface constraint checks
    }
    return ret;
}
```

```
consteval void interface(std::meta::info proto) {
    std::string_view name = identifier_of(proto);
    queue_injection(^^{
        class \id(name) {
        public:
            \tokens(make_interface_functions(proto))
            virtual ~\id(name)() = default;
            \id(name)() = default;
            \id(name)(\id(name) const&) = delete;
            void operator=(\id(name) const&) = delete;
        };
    });
}
```

3.2 Applying multiple functions

This paper proposes that `class(/*...*/)` be able to contain a comma-separated list of metafunctions to apply, which are applied in order.

For example, this:

```
class(xxx, yyy, zzz) Widget { /*...*/ };
```

would be syntactic sugar for this:

```
namespace __proto { Widget { /*...*/ };
  namespace __proto2 { consteval { xxx(^^Widget); }
    namespace __proto3 { consteval { yyy(^^Widget); }
  }
}
consteval { yyy(^^::__proto::__proto2::__proto3::Widget); }
```

4 References

[cppfront] H. Sutter. Cppfront compiler (GitHub, 2022-2024).

[P2996R5] W. Childers, P. Dimov, D. Katz, B. Revzin, A. Sutton, F. Vali, D. Vandevoorde. "Reflection for C++26" (WG21 paper, August 2024).

[P3294R1] A. Alexandrescu, B. Revzin, D. Vandevoorde. "Code injection with token sequences" (WG21 paper, July 2024).