

Metaclasses: Generative C++

Document Number: **P0707 R2**
Date: 2017-10-15
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: SG7

Contents

1	Overview.....	2
2	Language: Metaclasses.....	5
3	Library: Example metaclasses.....	18
4	Applying metaclasses: Qt moc and C++/WinRT	35
5	Alternatives for source→definition transform	41
6	Alternatives for applying the transform.....	44
7	Tooling	48
8	Revision history	50

Major additions in R2: Expanded §2.5 (composability), added §5, §6, and §7 (design alternatives, tooling).

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:

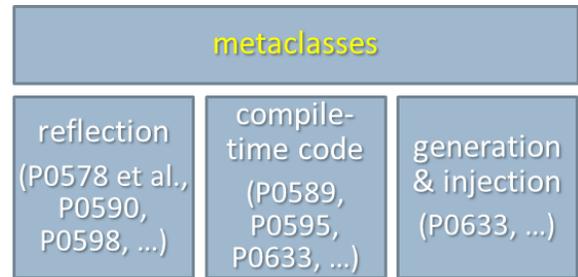
Elevate coding patterns/idioms into new abstractions built into the language. For example, in current C++, `range-for` lets programmers directly declare “for each” loops with compiler support and enforcement.

(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, `$class` metaclasses enable defining categories of `classes` that have common defaults and generated functions, and formally expand C++’s type abstraction vocabulary beyond `class/struct/union/enum`.

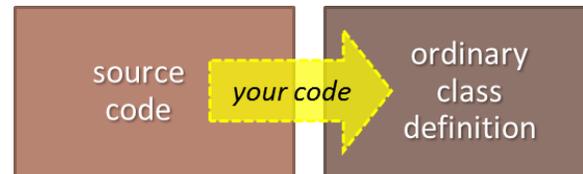
Also, §3 shows a set of common metaclasses, many of which are common enough to consider for `std::`. This paper begins by demonstrating how to implement Java/C# `interface` as a 10-line C++ `std::` metaclass – with the same usability, expressiveness, diagnostic quality, and performance of the built-in feature in such languages, where it is specified as ~20 pages of “standardese” text specification.

1 Overview

This paper assumes that C++ adds support for static reflection and compile-time programming to C++ along the lines of [P0578](#) and [P0633](#), and focuses on the next-level layer of abstraction we could build on top of that. This paper will not extensively describe those proposals, which are still evolving; see those papers for details. 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 metaclasses.



Metaclasses (provisional name) let programmers write a new kind of efficient abstraction: a user-defined named subset of `classes` that share common characteristics – including user-defined rules, defaults, and generated functions – by writing a custom transformation from normal C++ source code to a normal C++ class definition. 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 “just code” so they 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 \Rightarrow easier evolution, higher quality proposals. Common metaclasses (like common classes) can be standardized as `std::` libraries.
- For C++ implementations: Fewer new language features \Rightarrow less new compiler work and more capacity to improve tooling and quality for existing features. Over time, can deprecate and eventually remove many nonstandard extensions.

A Clang-based prototype is available at github.com/asutton/clang (source) and cppx.godbolt.org (live compiler). Some examples in this paper have links to the online compiler where you can see and experiment with the example.

1.1 Design principles

Note These principles apply to all design efforts and aren't specific to this paper. Please steal and reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability. – For example, in metaclasses we use normal class declaration syntax instead of inventing novel syntax.
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination. – For example, in these papers `for` can be used to process a reflected collection of items (e.g., all the member functions of a class), without having a distinct special-purpose `for_each<>` on a reflected collection.
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features. – For example, this paper prefers to avoid creating a special-purpose syntax to define metaclasses, and instead lets programmers use normal class scope declaration syntax plus the general features of reflection and compile-time programming. Also, metaclasses are just code, that can appear wherever code can appear – written inside namespaces to avoid name collisions (including putting common ones in `std::`), and shared via `#include` headers or via modules.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

1.2 Synopsis of P0578 & P0633-based syntax (expect this to evolve)

This paper assumes concepts, general compile-time programming along the lines proposed in P0633 and related papers, and underlying reflection facilities along the lines in P0194, P0385, P0578 and related papers. This paper is tracking the evolution of those compile-time facilities, whose syntax is still undergoing change; here is a “cheat sheet” synopsis of current draft syntax for the main features of those papers that will be used for this paper's examples, but the higher-level metaclass facility proposed herein is not affected by the syntactic details and the intent of this proposal is to build on whatever syntax ends up being adopted.

The strawman syntax for reflection is prefix `$`. For example:

```
$T           // reflect type T
$expr       // reflect expression expr
```

The strawman syntax for a compile-time code block, which can appear at any scope, is a `constexpr { }` block. Within a `constexpr` block, `-> { }` injects code into the enclosing scope. For example:

```
constexpr {           // execute this at compile time
    for (auto m : $T.variables()) // examine each member variable m in T
        if (m.name() == "xyzyz") // if there is one with name "xyzyz"
            -> { int plugh; }     // then inject also an int named "plugh"
}
```

For further details, see P0633 and the other cited papers.

Quick cheat sheet

Reflection

```
$T, $expr
```

Compile-time programming

```
constexpr {
    for (auto m : $T.variables())
        if (m.name() == "xyzyz")
            -> { int plugh; }
}
```

In addition, this paper proposes compiler-integrated diagnostics, where `compiler.error("message", source_location)` directs the compiler to emit the diagnostic message, which is intended to be integrated with the compiler's native diagnostics, including in visual style and control options. For example:

```
constexpr {
    for (auto f : $T.functions())           // for each member function f in T
        if (f.is_copy() || f.is_move())    // let's say we want to disallow copy/move
            compiler.error("this type may not have a copy or move function", f);
}
```

// note: passing f will use f.source_location() for this diagnostic message

For convenience, `compiler.require(cond, "message", source_location)` is equivalent to `if constexpr(!cond) compiler.error("message", source_location);`. So this is equivalent to the above:

```
constexpr {
    for (auto f : $T.functions())
        compiler.require(!f.is_copy() && !f.is_move()),
            "this type may not have a copy or move function", f);
}
```

Notes The current prototype implementation will change “for” to “for” per EWG direction in Kona, but in the meantime this paper still uses “for” to stay in closer sync with the compiler.

The current prototype implementation does not yet allow a `source_location`, so that has been temporarily removed from this paper's examples to make it easier to cut-and-paste examples from here into the prototype compiler. The `source_location` will be added so that diagnostics can have precise source line and column information.

1.3 Acknowledgments

Special thanks to Andrew Sutton and Bjarne Stroustrup for their review feedback on several drafts of this paper and other major contributions to C++. They are two of the primary designers of the current Concepts TS. Andrew Sutton is also the first implementer of the Concepts TS (in GCC 6), and the first implementer of this proposal (in a Clang-based prototype). This paper would be poorer without their insightful feedback.

Thanks also to the ACCU 2017 attendees for their enthusiastic reception and feedback after the talk on this topic at this spring's conference, and to the organizers for holding the video until we could also report the results of the initial presentation to the ISO C++ committee in July and produce the post-Toronto R1 revision of this paper.

Thanks also to the following experts for their comments in discussions and/or on pre-R0 drafts of this paper: Louis Brandy, Chandler Carruth, Casey Carter, Matúš Chochlík, Marshall Clow, Lawrence Crowl, Pavel Curtis, Louis Dionne, Gabriel Dos Reis, Joe Duffy, Thomas Heller, Howard Hinnant, Kenny Kerr, Nicolai Josuttis, Aaron Lahman, Scott Meyers, Axel Naumann, Gor Nishanov, Stephan T. Lavavej, Andrew Pardoe, Sean Parent, Jared Parsons, David Sankel, Richard Smith, Jeff Snyder, Mike Spertus, Mads Torgersen, Daveed Vandevoorde, Tony Van Eerd, JC van Winkel, Ville Voutilainen, and Titus Winters.

Thanks also to the following for further discussion, corrections, and other feedback since the R0 draft: Andras Agocs, Jonathan Boccara, Marco Foco, Alexandre Folle de Menezes, Barry Revzin.

2 Language: Metaclasses

“Classes can represent almost all the concepts we need... Only if the library route is genuinely infeasible should the language extension route be followed.” — B. Stroustrup (D&E, p. 181)

This paper relies on C++ classes’ already being general and unified. Stroustrup resisted all attempts to bifurcate the type system, such as to have `struct` and `class` be different kinds of types. The result is that the C++ `class` can express virtually every kind of type. – The goal of metaclasses is to fully preserve that, while also being able to define different kinds of types as reusable code by providing a narrow targeted hook: the ability to write compile-time code that participates in how the compiler interprets source code and turns it into a class definition.

Today’s language has rules to interpret source code and applies defaults and generates special member functions (SMFs). Here is a pseudocode example to illustrate how the compiler interprets `class` and `struct`:

Source code

```
class Point {
    int x, y;
};

struct MyClass : Base {
    void f() { /*...*/ }
    // ...
};
```

Compiler

```
for (m : members)
    if (!v.has_access())
        if (is_class())
            v.make_private();
        else // is_struct()
            v.make_public();

for (f : functions) {
    if (f.is_virtual_in_base_class()
        && !f.is_virtual())
        f.make_virtual();

    if (!f.is_virtual_in_base_class()
        && f.specified_override())
        ERROR("does not override");

    if (f.is_destructor())
        if (members_dtors_noexcept())
            f.make_noexcept();
}
```

Definition

```
class Point {
private:
    int x, y;
public:
    Point() =default;
    ~Point() noexcept =default;
    Point(const Point&) =default;
    Point& operator=(const Point&) =default;
    Point(Point&&) =default;
    Point& operator=(const Point&&) =default;
};

class MyClass : public Base {
public:
    virtual void f() { /*...*/ }
    // ...
};
```

Today, the contents of the “compiler” box is specified in English-like standardese and hardwired into compiler implementations. The generalization in this paper is to ask one narrowly targeted question:

Source code

```
class Point {
    int x, y;
};

struct MyClass : Base {
    void f() { /*...*/ }
    // ...
};
```

Compiler

Q: What if you could write your own code here, and give a name to a group of defaults & behaviors?

(treat it as ordinary code, share it as a library, etc.)

Definition

```
class Point {
private:
    int x, y;
public:
    Point() =default;
    ~Point() noexcept =default;
    Point(const Point&) =default;
    Point& operator=(const Point&) =default;
    Point(Point&&) =default;
    Point& operator=(const Point&&) =default;
};

class MyClass : public Base {
public:
    virtual void f() { /*...*/ }
    // ...
};
```

The intent is to “view `struct` and `class` as the first two metaclasses,”¹ except that today their semantics are baked into the language and written inside C++ compiler implementations, instead of being an extensibility point that can be written as ordinary C++ code.

This hook helps to solve a number of existing problems caused by the fact that “different kinds of types” are not supported by the language itself. For example, today we rely on coding patterns such as abstract base classes (“ABCs”) and “regular types” instead of giving names to language-supported features like “interface” or “value” that would let users easily name their design intent and get the right defaults, constraints, and generated functions for *that* kind of type. And the fact that there is only one kind of “class” means that the language’s defaults (e.g., all members private by default for classes and public for structs, functions that are virtual in a base class are virtual by default in the derived class) and generated special member functions (SMFs) (e.g., generate move assignment under these conditions) must be specified using a single heuristic for all conceivable types, which guarantees that they will be wrong for many types, and so when the heuristic fails we need tools like `=delete` to suppress an incorrectly generated SMF and `=default` to opt back in to a desired incorrectly suppressed SMF.

A **metaclass** allows programmers to write compile-time code that executes while processing the definition of class. In a nutshell, the goal is to:

- **name a subset** of the universe of C++ classes whose members share **common characteristics**;
- express that subset and its characteristics using **compile-time code** (which can be unit-tested, put in namespaces, shared in libraries, etc. like any other code); and
- **make classes easier to write** by letting class authors **use the name as a single-word “generalized opt-in”** to get that whole package of characteristics.

The goal is to elevate idiomatic conventions into the type system as compilable and testable code, and in particular to write all of the same diverse kinds of class types we already write today, but more cleanly and directly.

Metaclasses complement (and rely on) concepts and reflection, which are about *querying* capabilities – based on “does this expression compile” and “does this member/signature exist,” respectively. Metaclasses are about *defining* types – participating in interpreting the meaning of source code to generate the class definition.

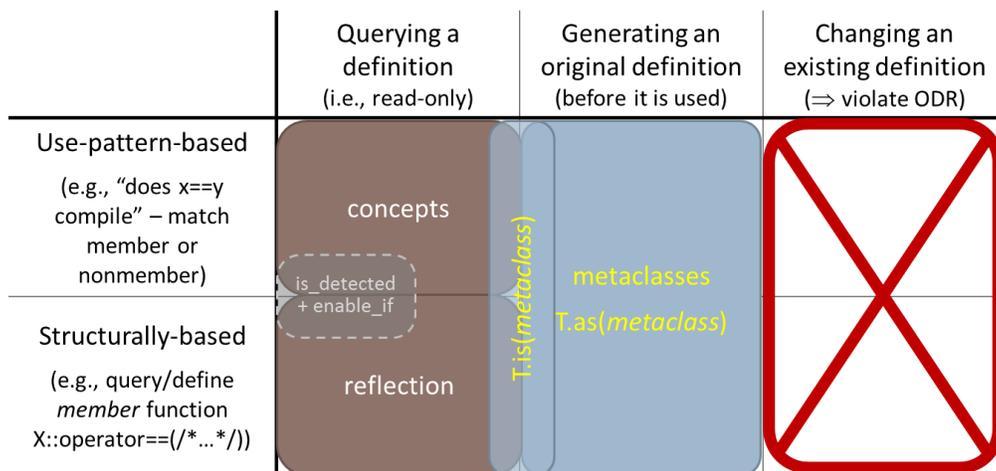


Figure 1: How the pieces fit

¹ And `union` and `enum` as the next two, though the latter has slightly different syntax than a `class`.

2.1 What and how: “Constructive” concepts

A metaclass is defined using `$class`, and can express constraints, defaults, and more using compile-time code. A metaclass is just code; it can be put in a namespace, and shared in a header or a module, in the same ways as other compile-time code we have today (in particular, templates). For example:

```
namespace std::experimental {
    $class interface {
        // we will describe how to write code to:
        // - apply “public” and “virtual” to member functions by default
        // - require all member functions be public and virtual
        // - require no data members, copy functions, or move functions
        // - generate a pure virtual destructor (if not user-supplied)
    };
}
```

A metaclass name can be written in place of `class` to more specifically define a type in terms of “what it is.” The compile-time code is run when instantiating the metaclass by using it to define an ordinary class:

```
interface Shape { // Shape is-a interface
    int area() const; // metacode in $class interface runs on
    void scale_by(double factor); // the contents in this proclass body
};
```

Here:

- Metaclass `interface` is used in place of the unspecialized keyword `class` to state that the characteristics associated with `interface` apply to `Shape`.
- The code the user writes as the body of `Shape` is the source **proclass**. Its AST is passed as input to the metaclass `interface`. The contents are available via reflection; the functions can be reflected as `$interface.functions()`, the data members as `$interface.variables()`, etc.
- At the opening brace of `interface`, `Shape` is **open** and its definition can be used by code in the body of metaclass `interface`, for reflection and other purposes. While a class is open (and only then), reflection on itself returns non-`const` information that can be modified.
- At the closing brace of `interface`, metaclass finalization runs (see below), after which `Shape` is **complete** a normal fully defined class type. This is the point of definition of `Shape`. When a class is fully defined, reflection returns `const` information.

Note Unlike in Java/C#, the type system is not bifurcated; there is still only one kind of `class`, and every interface is still a `class`. A metaclass simply gives a name to a subset of classes that share common characteristics and makes them easier to write correctly.

A metaclass’ code is fully general and so can express anything computable. There are four common uses:

- **Provide defaults:** Implicit meanings, such as “an interface’s functions are `public` and `virtual` by default” without the author of a particular interface type having to specify the default.
- **Generate members:** Default declarations and implementations for members that all classes conforming to the metaclass must have, such as “a `value` always has copy and move, and memberwise definitions are generated by default if copy and move are not explicitly written by hand.”

- **Enforce rules:** Constraints, such as “an `interface` contains only public virtual functions and is not copyable.” Use concepts to express usage-based patterns, and use reflection to query specific entities; together these enable a constraint to express anything computable about a type.
- **Perform transformations:** Changes to declared entities, such as “an `rt_interface` must have an `HRESULT` return type, and a non-`void` return type must be changed to an additional `[[out, retval]]` parameter instead,” or “a `variant` type replaces all of the data members declared in the protocol class with an opaque buffer in the fully defined class.”

Notes One result is that metaclasses provide “generalized opt-in” for generated functions. A metaclass replaces the built-in `class` special member function generation rules because the metaclass is taking over responsibility for all generation.

C++ provides only a few “special” generated functions for all classes, and more are desirable (e.g., comparisons). They are difficult to manage and extend because today C++ has only a monolithic universe of all classes, with no way to name subsets of classes. So, each compiler-generated “special member function” has to be generated based on a general heuristic that must work well enough *for all conceivable classes* to decide whether the function would likely be desired. But no heuristic is correct for all types, so this led to bugs when a special function was generated or omitted inappropriately (the heuristic failed), which led to the need for ways to “opt back out” and turn off a generated function when not desired (`=delete`) or to “opt back in” and use the default function semantics when the heuristic did not generate them (manual declaration followed by `=default`). Any new generated functions, such as comparisons, would need their own heuristics and face the same problems if the same rule is forced to apply to all possible classes.

Metaclasses provide a way to name a group of classes (a subset of the universe of all classes), and an extensible way to give that subset appropriate generated functions. Because the generated functions are provided by the metaclass, the metaclass name is the natural “opt-in” to get everything it provides. In turn, because generated functions are provided exactly and only when asked for, metaclasses remove the need to reinstate/suppress them – because we opted in, the functions the metaclass generates cannot logically be suppressed because if we didn’t want them we wouldn’t have opted into the metaclass (thus no need for `=delete` for generated functions), and because they are never suppressed by a heuristic we never need to reinstate them (thus no need to `=default` them).

Of course, `=default` and `=delete` are still useful for other things, such as a convenient way to get default bodies (see P0515) or to manage overload sets, respectively. The point here is only that, when using metaclasses, they are no longer needed to override an overly general heuristic that guesses wrong.

In a metaclass the following defaults apply, and are applied in metaclass finalization:

- Functions are public by default, and data members are private by default (if not already specified).
- The only implicitly generated function is a public nonvirtual default destructor (if not declared).

These are applied by the default metaclass program that runs the following at the end of the class definition after all other compile-time metaclass code (using `__` because this is in the language implementation of `$class`):

```
constexpr {
    for (auto o : $thisclass.variables())
        if (!o.has_access()) o.make_private();    // make data members private by default

    bool __has_declared_dtor = false;
    for (auto f : $thisclass.functions()) {
        if (!f.has_access()) f.make_public();    // make functions public by default
        __has_declared_dtor |= f.is_destructor(); // and find the destructor
    }

    if (!__has_declared_dtor)                    // if no dtor was declared, then
        -> { public: ~$thisclass.name$() { } }    // make it public nonvirtual by default
}
```

2.2 Metaclass bird’s-eye overview: Usage and definition examples

To illustrate, here is an overview of some equivalent code side by side. In each case, the code on the right is just a more convenient way to write exactly the code on the left and so has identical performance, but the code on the right offers stronger abstraction and so eliminates classes of errors and is more robust under maintenance.

C++17 style	This paper (proposed)
Applying a reusable abstraction with custom defaults and constraints = Medium improvement	
<pre>class Shape { public: virtual int area() const =0; virtual void scale_by(double factor) =0; // ... etc. virtual ~Shape() noexcept { }; // be careful not to write nonpublic/nonvirtual function }; // or copy/move function or data member; no enforcement</pre>	<pre>interface Shape { // see §3.1 int area() const; void scale_by(double factor); // ... etc. }; // see below in this table for t // definition of \$class interface</pre>
Applying a reusable abstraction that additionally has custom generated functions = Large improvement	
<pre>class Point { int x = 0; int y = 0; public: // ... behavior functions ... Point() = default; friend bool operator==(const Point& a, const Point& b) { return a.x == b.x && a.y == b.y; } friend bool operator< (const Point& a, const Point& b) { return a.x < b.x (a.x == b.x && a.y < b.y); } friend bool operator!=(const Point& a, const Point& b) { return !(a == b); } friend bool operator> (const Point& a, const Point& b) { return b < a; } friend bool operator>=(const Point& a, const Point& b) { return !(a < b); } friend bool operator<=(const Point& a, const Point& b) { return !(b < a); } };</pre>	<pre>value Point { // see §3.5 int x = 0; int y = 0; // ... behavior functions ... };</pre>

Applying a reusable abstraction with defaults, generated functions, and **custom semantics** = XL improvement

```
template <class T1, class T2>
struct pair {
    using first_type = T1;
    using second_type = T2;
    T1 first;
    T2 second;
    template <class... Args1, class... Args2>
        pair(piecewise_construct_t,
            tuple<Args1...> args1,
            tuple<Args2...> args2);
    constexpr pair();
    pair(const pair&) = default;
    pair(pair&&) = default;
    pair& operator=(const pair& p);
    pair& operator=(pair&& p) noexcept(see below);
    void swap(pair& p) noexcept(see below);
    explicit constexpr pair(const T1& x, const T2& y);
    template<class U, class V>
        explicit constexpr pair(U&& x, V&& y);
    template<class U, class V>
        explicit constexpr pair(const pair<U, V>& p);
    template<class U, class V>
        explicit constexpr pair(pair<U, V>&& p);
    template<class U, class V>
        pair& operator=(const pair<U, V>& p);
};
```

```
template<class U, class V>
pair& operator=(pair<U, V>&& p);
};
template <class T1, class T2>
constexpr bool operator==(
    const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
constexpr bool operator<
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
constexpr bool operator!=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
constexpr bool operator>
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
constexpr bool operator>=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
constexpr bool operator<=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template<class T1, class T2>
void swap(pair<T1, T2>& x, pair<T1, T2>& y)
    noexcept(noexcept(x.swap(y)));
template <class T1, class T2>
constexpr pair<T1, T2>
make_pair(T1&& x, T2&& y);
```

```
template<class T1, class T2>
literal_value pair {
    T1 first;
    T2 second;
};
// note: section 3 shows code for
// all metaclasses mentioned in the
// paper except for literal_value
```

Writing as-if a new ‘language’ feature using compile-time code + adding expressive power = XXL improvement

// C# language spec: ~20 pages of nontestable English



```
// User code (today’s Java or C#)
interface Shape {
    int area();
    void scale_by(double factor);
}
```

// (Proposed) C++ library impl: ~10 lines of testable code

```
$class interface { // see §3.1
constexpr {
    compiler.require($interface.variables().empty(),
        "interfaces may not contain data");
    for (auto f : $interface.functions()) {
        compiler.require(!f.is_copy() && !f.is_move(),
            "interfaces may not copy or move; consider a"
            " virtual clone() instead");
        if (!f.has_access()) f.make_public();
        compiler.require(f.is_public(),
            "interface functions must be public");
        f.make_pure_virtual();
    }
}
virtual ~interface() noexcept { }
};
```

```
// User code (proposed C++)
interface Shape {
    int area() const;
    void scale_by(double factor);
};
```

Notes Re “interface”: C++ has always been able to express “interfaces” in a manual ad-hoc manner and even gave the idiomatic convention a name (ABCs, for abstract base classes). There should be a way for class authors to express their intent more directly with a name that is actual code.

Re “pair”: Specifying the “simple” type `std::pair` has been embarrassingly complex. For years, I have been asking the world’s most experienced C++ language and library experts to describe what is missing from C++ to enable expressing `std::pair` as simply as

```
template <class T1, class T2> struct pair { T1 first; T2 second; };
```

but I never received an answer. As far as I know, this is the first proposal that achieves that goal, changing “struct” to a metaclass name (herein I call it “literal_value”) that can then be reused directly to just as simply define other similar types (e.g., `std::tuple`, users’ own literal value types).

2.3 Example: interface

The previous page shows the code for an example, `$class interface`, that could be a candidate for the standard library, and that has the same expressiveness, efficiency and usability as the same feature hardwired into other languages.

Note The concept of an “interface” exists in many languages as a built-in feature, specified in all those languages as pages of human-language specification and implemented in a compiler. I believe that the above specification and implementation is as good (and sometimes better) in every respect, including in strength of abstraction, expressiveness, error diagnostic quality, testability, debuggability, run-time performance, and (to be proven) compile-time performance.

`$interface.functions()` includes all functions in the current class `interface` is being applied to, including functions it inherited from any of its base classes. The `interface` metaclass:

- **Implicitly generates** a pure virtual destructor. In this case we can just implicitly declare the pure virtual destructor without any additional checks to see whether the user declared it the same way explicitly, because if the user did declare it explicitly then this declaration is just redundant. (In other cases, we’ll first check to see what the user declared, and then supply generated functions only if the user did not.)
- **Applies defaults** via compile-time code to make all functions public and pure virtual. This applies to all functions in the type including the required function that it declares itself (the destructor).
- **Applies constraints:** If the author of the type applying `interface` explicitly declared any nonpublic or nonvirtual function, copy/move function, or data member, they get a compile-time error message.

2.3.1 Applying interface

So now we can use `interface` in place of `class` when defining a new type, to get its defaults and generated functions, and to apply its requirements at compile time.

```
// see §3.1
interface drawable {           // this is an interface
    int draw(canvas& c);       // draw now defaults to public pure virtual
    // ...
};
```

And user code gets high-quality diagnostics when it violates constraints. For example, if this class is modified during maintenance by a programmer who forgets that it should consist of only public pure virtual functions, today the code could silently compile, but with `interface` the compiler helps robustly maintain the class author’s declared intent:

```

interface drawable {           // attempted modification during maintenance...
    int draw(canvas& c);       // ok
private:
    void scale(double factor); // ERROR: "interface functions must be public"
    string data;               // ERROR: "interfaces may not contain data"
};

```

Of course, if the maintainer really wants to add a nonpublic function or data member, they can still do that – they just need to change `interface` to a more suitable metaclass name, or just `class`, to document that this is no longer an `interface`. The change is simple, but not silent (it wouldn't be silent for class users in any event!), so that the maintainer cannot violate the original class author's intent by accident.

2.4 Declarations

The contents of a metaclass consist of:

- Declarations of members to appear the completed class, using ordinary class scope declaration syntax.
- Compile-time code to reflect, and modify protoclass members in-place or compute new declarations.

At metaclass scope, a member declaration can appear unadorned using all ordinary syntax. If in a `constexpr` block, it can be injected into the metaclass scope using `-> { }`.

Alternatively, a part of the declaration can be provided by compatible `meta::` values. For example, in a function declaration, the function name can be provided by any compile-time `meta::` value that has a `.name`, or a `meta::string`, and the parameter list can be provided by any compile-time `meta::` value that has `.parameters`:

```

$class x {
    // for each function, create a no-op overload with an extra "int" parameter
    constexpr {
        for (auto f : $x.functions())
            -> { void (f$)( f$, int ) { } }
    }
};

```

2.5 Composition / composability

There are two aspects to composability: of the generated classes, and of metaclasses themselves.

A metaclass-generated class is just as composable as any class, because metaclasses always generate just "an ordinary class" that could be exactly written by hand. Metaclasses just provide a more convenient way to write the classes we already write today where the common class patterns are expressed by manual convention.

Metaclasses themselves are as composable as concepts. Given two metaclasses `M1` and `M2`, and we want to apply both in that order to define a class `C`, consider the four major things a metaclass can do and how they compose:

- **Provide defaults:** `M1` runs first and applies its defaults. Then `M2` runs and applies any defaults not provided by `M1`.
- **Generate members:** `M1` runs first and generates its members. Then `M2` runs and generates its members (seeing those generated by `M1`).

- **Enforce rules:** `C` needs to follow the union of `M1`'s and `M2`'s constraints. For example, if `M1` requires copyability and `M2` requires a `hash()` function, then `C` must both be copyable and supply a `hash` function. Only if there are *mutually exclusive rules* that conflict (e.g., `M1` requires copying exist, `M2` requires no copying exist) then those two metaclasses cannot be used to define the same class. This is the same as with concepts once concepts can express constraints; for example, to extend examples from the concepts papers, with the addition of `constexpr{}` blocks we anticipate being able to express a concept `Regular` (a copyable value type that requires there be no protected or virtual members, just like this paper's `value` metaclass but query-only) and a concept `Interface` (an abstract base class that requires all members be virtual and that is not copyable, just like this paper's `interface` metaclass but query-only), and the same class cannot be both concept `Regular` and concept `Interface`, just as with the metaclasses in this paper the same class cannot be both metaclass `value` and metaclass `interface`.

For example, today we already write many kinds of classes by hand: regular types, iterators, POD types, abstract base classes, COM interfaces, and many more “kinds.” If each is now expressed as a metaclass, exactly and only those that are compatible compose elegantly, and this is a major improvement over status quo:

- **(as today) Correct combinations ‘Just Work’:** “Regular” and “iterator” compose easily. So do “regular” and “POD.” So do “abstract base class” and “COM interface.” This is no different from today; we can already write a type that is both “regular” and an “iterator,” but this proposal just makes them simpler to write initially and keep correct under maintenance. (See examples in the next subsection, and throughout this paper. See also §2.6.1 on `.is` which does not require foreknowledge to opt into a metaclass.)
- **(major improvement) Incorrect combinations are diagnosed at first attempted use:** “Regular” and “abstract base class” can never compose in principle because they are conceptually incompatible: one requires copying, one forbids it. A programmer already cannot write such a type without violating widely published C++ guidance: the *C++ Core Guidelines*, my and Alexandrescu's *C++ Coding Standards* and my other books, Stroustrup's books, Meyers' books, etc. – but if the programmer ignores the English guidance and does it anyway by mistake, today by default their monster class silently compiles and we can only hope they run a linter or that an experienced programmer catches it in code review. With metaclasses, the bad code cannot compile, and will get “a regular type must have a copy constructor” (and when they add one to quiet that message, “a base class must not have a copy constructor, consider a virtual clone() instead”) so they cannot compile without declaring actually compatible concepts – this is a major improvement over status quo.

In short, when we enable the programmer to express their intent in code, we can check it. And our limited experience with metaclasses so far bears out that this works as designed in practice.

Note The same will apply to concepts, and does already.

As with the first bullet above, concepts as described in the Concepts TS match positive must-have usage patterns, and so should all compose: requiring two concepts requires the union of the required usage patterns.

As with the second bullet above, in draft C++20 we already have concepts and `is_detected` in the same language, which should naturally enable concepts that additionally can include negative must-not-have requirements by using `!is_detected`, and so now concepts (like metaclasses) will be composable if and only if their requirements are compatible. This will continue when we add reflection and `constexpr{}` blocks to the language, because concepts too will be able to express the same

computed requirements as metaclasses, including “must not have copying” and “must not have virtual functions.” For example, something like:

```
concept Regular_type = /*today's requirements*/
    && constexpr { /*has no protected or virtual members*/ };
```

2.5.1 Syntax

The current strawman syntax lets metaclasses be written in terms of existing ones using an inheritance-like syntax. As with base class constructors, each metaclass is executed in left-to-right depth-first order. Here is an example from §3.5:

```
$class io_and_comparable : iostreamable, comparable { // both streams, and compares
    // ... with additional defaults/constraints/generation/etc. ...
};
$class mapped_serializable : mapped, serializable { // both maps, and serializes
    // ... with additional defaults/constraints/generation/etc. ...
};
$class value : basic_value, ordered { // a value is-an ordered basic_value
    // ... with additional defaults/constraints/generation/etc. ...
};
```

Note This is just using the current class-like strawman syntax with “inheritance”-like composition. Alternatively (see §5) if we choose a function-like syntax we would naturally use “call”-like composition:

```
// using one of the alternative metaclass definition syntaxes in §7
void io_and_comparable(meta::type source, meta::type& target) constexpr {
    // both streams, and compares
    iostreamable(source, target);
    comparable(source, target);
    // ... with additional defaults/constraints/generation/etc. ...
}

void value(meta::type source, meta::type& target) constexpr {
    // a value is-an ordered basic_value
    basic_value(source, target);
    ordered(source, target);
    // ... with additional defaults/constraints/generation/etc. ...
}
```

This model has several advantages and we are actively investigating it. However, the actual content of each metaclass is basically the same structurally; see §5 for more discussion.)

A metaclass can also compose concepts, with the semantics that the resulting class is required to satisfy the concept. For example, given a concept `Regular`, we can add it to the requirements list:

```
$class value : ordered, basic_value, Regular { // include a Concept
    // etc.
};
```

and behaves as a convenience shorthand for:

```

$class value : ordered, basic_value {
    // etc.                                // run metaprogram first: defaults/generation/...
    requires Regular<value>;               // then put this at the end, when we have
};                                        // the complete generated type

```

2.6 .is and .as

2.6.1 .is to match

We can perform ad-hoc duck typing to test whether a class implicitly satisfies the requirements of a metaclass M . In this proposal, $\$T.is(M)$ evaluates to `true` iff:

- applying M to T (as-if the definition of T had specified M) succeeds; and
- the resulting type has no new members not already present in T .

For example, this test uses the `copyable_pointer` metaclass defined in §3.6:

```
static_assert ($shared_ptr<widget>.is(copyable_pointer<widget>));
```

For example, consider `Shape` written equivalently by hand vs. using the `interface` metaclass:

```

class Shape1 {                                // written by hand as in C++17
public:
    virtual void draw() = 0;
    virtual ~Shape1() noexcept = 0;
};

interface Shape2 {                            // same written using a metaclass
    void draw();
};

```

Both types `.is(interface)`:

```

static_assert ($Shape1.is(interface));
static_assert ($Shape2.is(interface));

```

This applies transitively to base metaclasses. For example, if `interface` had been refactored as shown in §2.5 to be written in terms of a `has_pure_virtual_dtor` “base” metaclass, the following would also hold:

```

static_assert ($Shape1.is(has_pure_virtual_dtor));
static_assert ($Shape2.is(has_pure_virtual_dtor));

```

This loop prints the names of all interfaces in namespace N :

```

constexpr {
    for (auto t : $N.types())
        if (t.is(interface))
            cout << t.name() << endl;
}

```

2.6.2 `.as` to apply

Additionally, we can use a class as-if it had been declared with a metaclass, including to apply defaults and generated functions. To express that, use `$T.as(M)`, which generates a type that is identical to `T` but is additionally defined using the named metaclass `M`. Here is an example using a metaclass `ordered` (see §3.4):

```
struct legacy_point { int x; int y; }; // in C++17 this is not comparable...
set<legacy_point> s; // and so this is an error

using ordered_point = $legacy_point.as(ordered); // ... but this is ordered
set<ordered_point> s; // and so this is ok
```

Interestingly, the above example illustrates how strong typedefs fall out naturally from `.as ...`

2.6.3 Bonus: strong typedefs via `using ... as`

To enable general strong typedefs via `using ... as`, we first define an empty metaclass, which requires and adds nothing to the type. Let's call it `new_type` because that's how programmers will use it:

```
$class new_type { }; // no-op metaclass
```

Then the following is a spelling for “strong typedef of `T`”:

```
using my_T = $T.as(new_type);
```

[Common motivating cases](#) are new `int` and `string` types that work the same as the originals but are distinct types for overloading and do not implicitly convert to/from the original type by default.

```
using handle = $int.as(new_type); // better than “enum class handle : int { };”
using score = $unsigned.as(new_type);
using player = $string.as(new_type);
```

2.7 Concepts + metaclasses

Concepts and metaclasses are complementary. The key is that metaclasses are “constructive concepts” in that they go beyond concepts to define new types, but metaclass implementations *use* both concepts and reflection:

- Metaclasses use concepts to ask “can class `T` be used this way” via use-pattern constraints.
- Metaclasses use reflection-based computation to ask “does class `T` have these contents” via inspection.

Because both concepts and metaclasses have requirements and constraints, we should allow the complementary applications, which both involve replacing the keyword `class`.

First, concepts allow class *uses* to be constrained by replacing `class` with a concept name:

```
template <class T> // unconstrained - any type will do
template <Sequence S> // constrained - requires Sequence<S>
```

So we propose that a metaclass also be allowed to replace `class` here with `.is` meaning:

```
template <interface I> // constrained - requires $I.is(interface)
```

Second, metaclasses allow class *definitions* to be constrained by replacing `class` with a metaclass name:

```

class X { /*...*/ }; // unconstrained - “just some type”
interface I { /*...*/ }; // constrained - is-an interface

```

So we propose that a concept also be allowed to replace `class` here with the meaning of checking that the complete type must satisfy the concept:

```

Sequence S { /*...*/ }; // constrained - requires Sequence<S>

```

Note Casey Carter has asked for this feature in the past, and reports that this capability would be used widely in the Ranges TS implementation.

There is currently no way to enforce these conditions for specializations of a template. Here is the essence of the problem:

```

template<typename T>
struct S {
    // ...
    static_assert(Regular<S>); // always fails, S is incomplete
};

static_assert(Regular<S<??>>); // what goes in ???

```

The above proposal provides a way to express an annotation in `S` that can be extracted and applied after instantiation:

```

template<typename T>
Regular S {
    // ...
};

```

Alternatively, writing an explicit `requires` is useful in combination with conditional compile-time programming. For example:

```

template<typename T>
struct vector {
    // ...

    constexpr {
        if (Copyable<T>) // if T is Copyable, then
            -> { requires Copyable<vector>; } // vector<T> is also Copyable
    }
};

```

However, note that this is just a requirement check; it does not make `vector` model `Copyable`. This is a minor extension of modern Concepts TS concepts; it is not moving towards C++0x concepts, Haskell typeclasses, Rust traits, etc. by injecting anything into the class.

3 Library: Example metaclasses

This section shows how to use metaclasses to define powerful abstractions as libraries, often only in a few lines, without loss of efficiency, expressiveness, usability, diagnostics, or debuggability compared to languages that support them as language features baked into their compilers.

This paper proposes considering the following subset as `std::` standard libraries:

- `interface`, an abstract base `class` with all public virtual functions and no copy/move or data members;
- `base_class`, a `class` designed to be inherited from with no copy/move or data members;
- `ordered` et al., each a `class` that supports a comparison category (e.g., total ordering, equality comparison);
- `value`, a `class` that is a “regular” type with default construction, destruction, copy/move, and comparison (memberwise by default), and no virtual functions or protected members;
- `plain_struct` (what we usually mean when we write “struct”), and `flag_enum`.

3.1 interface

“... an abstract base class defines an interface...”—Stroustrup (D&E § 12.3.1)

An `interface` is a `class` where all functions are public and pure virtual, both by requirement and by default, and there is a virtual destructor and no data or copying. The definition is as we saw earlier.

// LIVE, click here for live example: <https://godbolt.org/g/Uzw5iJ>

```
$class interface {
    constexpr {
        compiler.require($interface.variables().empty(),
            "interfaces may not contain data");

        for (auto f : $interface.functions()) {
            compiler.require(!f.is_copy() && !f.is_move(),
                "interfaces may not copy or move; consider a"
                " virtual clone() instead");

            if (!f.has_access()) f.make_public();
            compiler.require(f.is_public(),
                "interface functions must be public");

            f.make_pure_virtual();
        }
    }
    virtual ~interface() noexcept { }
};
```

We can then use this to define classes, including to use access/virtual defaults and enforce rules:

```
interface Shape {
    int area() const;
    void scale_by(double factor);
    // int x; // would be error, no data allowed
```

```

    // private: void g();           // would be error, no private functions allowed
    // Shape(const Shape&);        // would be error, no copying allowed
};

```

In this interface, `area` and `scale_by` are implicitly public and pure virtual because nothing else is allowed. Trying to make a function explicitly public or virtual would be fine but redundant. Trying to make a function explicitly nonpublic or nonvirtual would be an error, as would adding copy/move functions or data members.

3.2 `base_class`

A pure `base_class` is a `class` that has no instance data, is not copyable, and whose a destructor is either public and virtual or protected and nonvirtual. Unlike an `interface`, it can have nonpublic and nonvirtual functions. Also, implemented interfaces are public by default.

// LIVE, click here for live example: <https://godbolt.org/g/sXmhkN>

```

$class base_class {
    constexpr {
        for (auto f : $base_class.functions()) {
            if (f.is_destructor() &&
                !((f.is_public() && f.is_virtual())
                 || (f.is_protected() && !f.is_virtual()))
                compiler.error("base class destructors must be public and"
                               " virtual, or protected and nonvirtual");
            if (f.is_copy() || f.is_move())
                compiler.error("base classes may not copy or move;"
                               " consider a virtual clone() instead");
            if (!f.has_access()) f.make_public();
        }
        for (auto b : $base_class.bases())
            if (!b.has_access()) b.make_public();
        if (!$base_class.variables().empty())
            compiler.error("pure base classes may not contain data");
    }
};

```

These can be used to write types that match that metaclass:

```

base_class Rectangle : Shape {
    int area() const override { /*...*/ }
    void scale_by(double factor) override { /*...*/ }
};

```

3.3 `final`

A `final` type is a `class` that cannot be further included in another type (aka derived from).

```

$class final {
    final.can_derive = false;           // can't derive from this
};

```

For example:

```
final circle : shape {
    override void draw(canvas& c) { /*...*/ }
};
```

3.4 ordered

Notes Up to this point, we have only used metaclasses (a) to apply defaults to declared functions and variables, and (b) to enforce requirements. Now we’re going to take another step: additionally using them to implement custom default-generated functions. C++17 already does this for the special member functions; the difference here is that no functions are “special” (this works for any function we want to both require to exist and generate a suitable default implementation for) and it’s not hardwired into the language. In this section and the next, we’ll cover the most familiar generated functions—default construction, copy construction, copy assignment, move construction, and move assignment—and comparisons which is where we’ll begin.

This section is written in terms of C++17 and does not depend on my parallel paper [P0515 Consistent Comparison](#). However, P0515 makes comparisons much better, and if that paper is adopted then this section is easily updated to refer to the features added by that paper including $\lt\Rightarrow$ three-way comparison and all five comparison categories. Nearly all of P0515 can be implemented as a library in this way, except only the automatic generation of comparison functions for fundamental types and for existing class types defined without metaclasses.

This section illustrates how opting in to default comparisons is easy and efficient using metaclasses, by demonstrating a single comparison category (total ordering) implemented as a library with full opt-in semantics.

An `ordered` type is a `class` that requires operators `<`, `>`, `<=`, `>=`, `==`, and `!=`. If the functions are not user-written, lexicographical memberwise implementations are generated by default.

```
$class ordered {
    constexpr {
        if (! requires(ordered a) { a == a; }) -> {
            friend bool operator == (const ordered& a, const ordered& b) {
                constexpr {
                    for (auto o : ordered.variables()) // for each member
                        -> { if (!(a.o.name$ == b.(o.name)$)) return false; }
                }
                return true;
            }
        }
    }

    if (! requires(ordered a) { a < a; }) -> {
        friend bool operator < (const ordered& a, const ordered& b) {
            for (auto o : ordered.variables()) -> { // for each member
                if (a.o.name$ < b.(o.name)$) return true; // (disclaimer: inefficient; P0515
                if (b.(o.name)$ < a.o.name$) return false; // with 3-way comparison is better)
            }
            return false;
        }
    }
}
```

```

    }
    if (! requires(ordered a) { a != a; })
        -> { friend bool operator != (const ordered& a, const ordered& b) { return !(a == b); } }
    if (! requires(ordered a) { a > a; })
        -> { friend bool operator > (const ordered& a, const ordered& b) { return b < a; } }
    if (! requires(ordered a) { a <= a; })
        -> { friend bool operator <= (const ordered& a, const ordered& b) { return !(b < a); } }
    if (! requires(ordered a) { a >= a; })
        -> { friend bool operator >= (const ordered& a, const ordered& b) { return !(a < b); } }
    }
};

```

Note This example (and quite a few others) will not compile in the prototype compiler yet for a number of incompleteness reasons, but here in particular because this code uses concepts and Clang does not yet have concepts.

This example shows how using concepts is convenient in metaclasses, especially when we just care whether a given operation (here comparison) is provided already at all, regardless of the manner in which it's provided (as a member, nonmember friend, etc.). So this code just writes:

```
if (! requires(ordered a) { a == a; })
```

Alternatively, we could also have written the following (assuming `constexpr` range-based `find_if`), but it's more tedious and less general:

```
if (find_if($ordered.functions(),
          [](auto x){ return x.name == "operator=="; })
    != ordered.functions().end())
```

The author of a totally ordered type can just apply `ordered` to get all comparisons with memberwise semantics:

```

// using ordered (but prefer "value", see §3.5 -- this is for illustration)
ordered Point { int x; int y; /*copying etc. */ }; // no user-written comparison

Point p1{0,0}, p2{1,1};
assert (p1 == p1);           // ok, == works
assert (p1 != p2);           // ok, != works

set<Point> s;                 // ok, less<> works
s.insert({1,2});             // ok, < works

```

However, most code will not use `ordered` directly because it's an intermediate metaclass. Which brings us to **value**, an important workhorse...

3.5 value types (regular types)

A **value** is a `class` that is a totally ordered regular type. It must have all public default construction, copy/move construction/assignment, and destruction, all of which are generated by default if not user-written; and it must not have any protected or virtual functions (including the destructor).

`basic_value` carries the common defaults and constraints that apply to regular value types:

// LIVE, click here for live example: <https://godbolt.org/g/59LSSZ>

```
$class basic_value {
    constexpr {
        if (find_if(basic_value.functions(), [](auto x){ return x.is_default_ctor(); }) != basic_value.functions().end())
            -> { basic_value() = default; }

        if (find_if(basic_value.functions(), [](auto x){ return x.is_copy_ctor(); }) != basic_value.functions().end())
            -> { basic_value(const basic_value& that) = default; }

        if (find_if(basic_value.functions(), [](auto x){ return x.is_move_ctor(); }) != basic_value.functions().end())
            -> { basic_value(basic_value&& that) = default; }

        if (find_if(basic_value.functions(), [](auto x){ return x.is_copy_assignment(); }) != basic_value.functions().end())
            -> { basic_value& operator=(const basic_value& that) = default; }

        if (find_if(basic_value.functions(), [](auto x){ return x.is_move_assignment(); }) != basic_value.functions().end())
            -> { basic_value& operator=(basic_value&& that) = default; }

        for (auto f : basic_value.functions()) {
            compiler.require(!f.is_protected() && !f.is_virtual(),
                "a value type must not have a protected or virtual function");
            compiler.require(!f.is_destructor() || !f.is_public(), "a value type must have a public destructor");
        }
    }
};
```

A value is a totally ordered regular type:

```
$class value : ordered, basic_value { };
```

Note If P0515 is accepted, we would naturally expand this to provide other convenient opt-ins here, and because “total ordering” and “equality comparable” are the most commonly used and the default to be encouraged, they get the nice names:

```
$class weakly_ordered_value    : weakly_ordered    , basic_value { };
$class partially_ordered_value : partially_ordered , basic_value { };
$class equal_value            : equal              , basic_value { };
$class weakly_equal_value     : weakly_equal       , basic_value { };
```

Example:

```
value Point { int x; int y; }; // note: that's it, convenient and fully opt-in
Point p1;                      // ok, default construction works
Point p2 = p1;                 // ok, copy construction works

assert (p1 == p1);             // ok, == works
assert (p1 >= p2);             // ok, >= works

set<Point> s;                   // ok, less<> works
s.insert({1,2});
```

3.6 plain_struct

“By definition, a struct is a class in which members are by default public; that is,

```
struct s { ...
```

is simply shorthand for

```
class s { public: ...
```

*... Which style you use depends on circumstances and taste. I usually prefer to use **struct** for classes that have all data public.” — B. Stroustrup (C++PL3e, p. 234)*

A `plain_struct` is a `basic_value` with only public objects and functions, no virtual functions, no user-defined constructors (i.e., no invariants) or assignment or destructors, and the most powerful comparison supported by all of its members (including none if there is no common comparison category).

Notes Up to this point, we’ve seen (a) applying defaults, (b) enforcing requirements, (c) combining metaclasses. Now we’ll look at reflecting on members, evaluating whether they meet a metaclass, and selectively combining metaclasses.

The full 5-way comparison category computation below assumes we’ve gone ahead with P0515, so they’re stronger than the simple extract shown in §3.3.

// LIVE, click here for live example: <https://godbolt.org/g/2uMpF5>

```
$class plain_struct : basic_value {
  constexpr {
    for (auto f : $plain_struct.functions()) {
      compiler.require(f.is_public() && !f.is_virtual(),
        "a plain_struct function must be public and nonvirtual");

      compiler.require(!(f.is_constructor() || f.is_destructor()
        || f.is_copy() || f.is_move()
        || f.is_defaulted()),
        "a plain_struct can't have a user-defined "
        "constructor, destructor, or copy/move");
    }

    bool all_ordered      = true,    // to compute common comparability
         all_weakly_ordered = true,
         all_partially_ordered = true,
         all_equal         = true,
         all_weakly_equal   = true;

    for (auto o : plain_struct.variables()) {
      if (!o.has_access()) o.make_public();
      compiler.require(o.is_public(), "plain_struct members must be public");

      all_ordered      &= o.type.is(ordered);
      all_weakly_ordered &= o.type.is(weakly_ordered);
      all_partially_ordered &= o.type.is(partially_ordered);
      all_equal         &= o.type.is(equal);
      all_weakly_equal   &= o.type.is(weakly_equal);
    }
  }
}
```

```

    }
    if (all_ordered)           // generate greatest common comparability
        plain_struct = plain_struct.as(ordered);
    else if (all_equal)
        plain_struct = plain_struct.as(equal);
    else if (all_weakly_ordered)
        plain_struct = plain_struct.as(weakly_ordered);
    else if (all_weakly_equal)
        plain_struct = plain_struct.as(weakly_equal);
    else if (all_partially_ordered)
        plain_struct = plain_struct.as(partially_ordered);
}
};

```

Now we can use `plain_struct` to have this meaning strictly, without relying on it being just a personal convention. To write a type that self-documents this intent, we can write for example:

```

plain_struct group_o_stuff {
    int i;           // implicitly public
    string s;
};
group_o_stuff a, b, c;           // ok, because values are default-constructible
if (a == b && c > a) { }       // ok, ordered because all members are ordered

```

3.7 copyable_pointer

A `copyable_pointer` is a value that has at least one type parameter and overloads `*` to return an lvalue of that parameter and `->` to return a pointer to that parameter.

```

template<class T>
class copyable_pointer : value {
    T& operator* () const;           // require * and -> operators
    T* operator->() const;
};

```

Now we can use `copyable_pointer` both to tell if a type is a smart pointer, and to write new smart pointers (unlike concepts).

```

static_assert ($shared_ptr<widget>.type.is(copyable_pointer<widget>));
copyable_pointer<gadget> my_ptr {
    // ... can't forget to write copying and both indirection operators ...
};

```

3.8 enum_class and flag_enum

“C enumerations constitute a curiously half-baked concept. ... the cleanest way out was to deem each enumeration a separate type.”—[Stroustrup, D&E §11.7]

“An enumeration is a distinct type (3.9.2) with named constants”—[ISO C++ standard]

An `enum_class` is a totally ordered `value` type that stores a value of its enumerators’ type, and otherwise has only `public` member variables of its enumerators’ type, all of which are naturally scoped because they are members of a type.

Note Up to this point, we’ve seen (a) applying defaults, (b) enforcing requirements, (c) combining metaclasses, (d) reflecting on members and computing characteristics such as selectively combining metaclasses. Now, we’ll generate an additional data member.

```
$class basic_enum : value {
  constexpr {
    compiler.require(basic_enum.variables().size() > 0,
                    "an enum cannot be empty");

    if ($basic_enum.variables().front().type().is_auto())
      -> { using U = int; } // underlying type
    else -> { using U = $basic_enum.variables().front().type(); }

    for (auto o : $basic_enum.variables) {
      if (!o.has_access()) o.make_public();
      if (!o.has_storage()) o.make_constexpr();
      if (o.has_auto_type()) o.set_type(U);
      compiler.require(o.is_public(), "enumerators must be public");
      compiler.require(o.is_constexpr(), "enumerators must be constexpr");
      compiler.require(o.type() == U, "enumerators must use same type");
    }
    -> { U$ value; } // the instance value
  }
};
```

Note A very common request is to be able to get string names of enums (e.g., [StackOverflow example](#)). It is tempting to provide that as a function on `basic_enum` that is always available, which would be easy to provide. But we need to be careful not to violate C++’s zero-overhead principle; we must not impose overhead (here in the object/executable image) by default on programs that don’t use it. Making this available always or by default, such as always generating string names for the members of a `basic_enum`, would be a baby step down the slippery slope toward always-on or default-on runtime metadata.

However, making it opt-in would be fine. One way would be to have a specific metaclass that adds the desired information. A better way would be to write a general constrained function template:

```
template<basic_enum E> // constrained to enum types
string_to_string(E e) {
  switch (value) {
    constexpr {
      for (const auto o : $E.variables())
```

```

        if (!o.default_value.empty())
            -> { case o.default_value(): return E::(o.name()); }
    }
}

```

Because templates are only instantiated when used, this way the information is generated (a) on demand at compile time, (b) only in the calling code (and only those calling programs) that actually use it, and (c) only for those enum types for which it is actually used.

There are two common uses of enumerations. First, `enum` expresses an enumeration that stores exactly one of the enumerators. The enumerators can have any distinct values; if the first enumerator does not provide a value, its value defaults to 0; any subsequent enumerator that does not provide a value, its value defaults to the previous enumerator's value plus 1. Multiple enumerators can have the same value.

```

$class enum_class : basic_enum {
    constexpr {
        U next_value = 0;
        for (auto o : $enum_class.variables()) {
            if (!o.has_default_value())
                o.set_default_value(next_value);
            next_value = o.get_default_value()++;
        }
    }
};

```

Here is a `state` enumeration that starts at value 1 and counts up:

```

enum_class state {
    auto started = 1, waiting, stopped;    // type is int
};

state s = state::started;
while (s != state::waiting) {
    // ...
}

```

Here is a different enumeration using a different value type and setting some values while using incremented values where those are useful:

```

enum_class skat_games {
    char diamonds = 9, hearts /*10*/, spades /*11*/, clubs /*12*/, grand = 24;
};

```

Second, `flag_enum` expresses an enumeration that stores values corresponding to bitwise-or'd enumerators. The enumerators must be powers of two, and are automatically generated; explicit values are not allowed. A `none` value is provided, with an explicit conversion to `bool` as a convenience test for "not none." Operators `|` and `&` are provided to combine and extract values.

```

$class flag_enum : basic_enum {
    flag_enum operator& (const flag_enum& that) { return value & that.value; }
    flag_enum& operator&= (const flag_enum& that) { value &= that.value; return *this; }
}

```

```

flag_enum operator| (const flag_enum& that) { return value | that.value; }
flag_enum& operator|= (const flag_enum& that) { value |= that.value; return *this; }
flag_enum operator^ (const flag_enum& that) { return value ^ that.value; }
flag_enum& operator^= (const flag_enum& that) { value ^= that.value; return *this; }

flag_enum()          { value = none; } // default initialization
explicit operator bool() { value != none; } // test against no-flags-set

constexpr {
    compiler.require(objects.size() <= 8*sizeof(U),
                    "there are " + objects.size() + " enumerators but only room for " +
                    to_string(8*sizeof(U)) + " bits in value type " + $U.name());

    compiler.require(!numeric_limits<U>.is_signed,
                    "a flag_enum value type must be unsigned");

    U next_value = 1; // generate powers-of-two values
    for (auto o : $flag_enum.variables()) {
        compiler.require(!o.has_default_value(),
                        "flag_enum enumerator values are generated and cannot be specified explicitly");
        o.set_default_value(next_value);
        next_value *= 2;
    }
}

U none = 0; // add name for no-flags-set value
};

```

Here is an `ios_mode` enumeration that starts at value 1 and increments by powers of two:

```

flag_enum openmode {
    auto in, out, binary, ate, app, trunc; // values 1 2 4 8 16 32
};

openmode mode = openmode::in | openmode::out;
assert (mode != openmode::none); // comparison comes from 'value'
assert (mode & openmode::out); // exercise explicit conversion to bool

```

Note There is a recurring need for a “flag enum” type, and writing it in C++17 is awkward. After I wrote this implementation, [Overload 132](#) (April 2016) came out with Anthony Williams’ article on “Using Enum Classes as Bitfields.” That is a high-quality C++17 library implementation, and illustrates the limitations of authoring not-the-usual-class types in C++: Compared to this approach, the C++17 design is harder to implement because it relies on TMP and SFINAE; it is harder to use because it requires flag-enum type authors to opt into a common trait to enable bitmask operations; and it is more brittle because the flag-enum type authors must still set the bitmask values manually instead of having them be generated. In C++17, there is therefore a compelling argument to add this type because of its repeated rediscovery and usefulness—but to be robust and usable it would need to be added to the core language, with all of the core language integration and wordsmithing that implies including to account for feature interactions and cross-referencing; in a future C++ that had the capabilities in this proposal, it could be added as a small library with no interactions and no language wording.

3.9 bitfield

A `bitfield` is an object that allows treating a sequence of contiguous bits as a sequence of values of trivially copyable types. Each value can be get or set by copy, which the implementation reads from or writes to the value bits. To signify padding bits, set the type to `void` or leave the name empty. It supports equality comparison.

Note Also, treating a bitfield as an object is truer to the C++ memory model. The core language already says (though in standardese English) that a sequence of bitfield variables is treated as a single object for memory model purposes. That special case falls out naturally when we model a sequence of bits containing multiple values as a single object.

A `bitfield` metaclass could pass each member's size as an attribute (e.g., `int member [[3]];`) – but since we already have the bitfield-specific C grammar available, let's use it:

```
bitfield game_stats {
    int      score_difference : 3;
    void     _                : 2;    // padding
    unsigned counter         : 6;
} example;
```

Note Up to this point, we've seen (a) applying defaults, (b) enforcing requirements, (c) combining metaclasses, (d) reflecting on members and computing characteristics such as selectively combining metaclasses, and (e) generating additional data members. Now we'll go further and not just generate new data members, but actually remove the existing declared data members and replace them.

Here is the code:

```
$class bitfield : final, comparable_value {    // no derivation
    constexpr {
        auto objects = bitfield.variables();    // take a copy of the class's objects
        size_t size = 0;                        // first, calculate the required size

        for (auto o : objects) {
            size += (o.bit_length == default ? o.type.size*CHAR_BITS : o.bit_length);

            if (!o.has_storage()) o.make_member();
            compiler.require(o.is_member(), "bitfield members must not be static");

            compiler.require(is_trivially_copyable_v<o.T>,
                "bitfield members must be trivially copyable");

            compiler.require(!o.name.empty() || o.T == $void,
                "unnamed bitfield members must have type void");

            compiler.require(o.type != $void || o.name.empty(),
                "void bitfield members must have an empty name");

            if (o.type != $void) -> {           // generate accessors for non-empty members
                o.T$ o.name$ () { return /*bits of this member cast to T*/; }
                set_(o.name)$ (const o.T&& val) { /*bits of this value*/ = val; }
            }
        }
    }
}
```

```

    $bitfield.variables().clear();           // replace the previous instance vars
    -> { byte data[ (size/CHAR_BITS)+1 ]; } // now allocate that much storage
}

bitfield() {                               // default ctor inits each non-pad member
    constexpr {
        for (auto o : objects)
            if (o.type != $void)
                -> { /*set bits of each value to its default value*/ };
    }
}

~bitfield() {                               // cleanup goes here
    constexpr {
        for (auto o : objects)
            if (o.type != $void)
                -> { o.name$.~(o.type.name$)( ); }
    }
}

bitfield(const bitfield& that) : bitfield() { // copy constructor
    *this = that;                            // just delegate to default ctor + copy =
} // you could also directly init each member by generating a mem-init-list

bitfield& operator=(const bitfield& that) { // copy assignment operator
    constexpr {
        for (auto o : objects)               // copy each non-pad member
            if (o.type != $void)            // via its accessor
                -> { set_(o.name$)( that.(o.name)$() ); }
    }
}

bool operator==(const bitfield& that) const {
    constexpr {                               // (we'll get != from 'comparable_value')
        for (auto o : objects)               // just compare each member
            -> { if (o.name$() != that.(o.name)$()) return false; }
        return true;
    }
}
};

```

For example, this bitfield fits in two bytes, and holds two integers separated by two bits of padding:

```

bitfield game_stats {
    int      score_difference : 3;
    void     _                : 2;    // padding
    unsigned counter          : 6;
} example;

example.set_score_difference(-3); // sadly, the home team is behind
unsigned val = example.counter(); // read value back out

```

Note that in computing the size, the metaclass defaults to the natural size if the number of bits is not explicitly specified. For example, the following two are the same on systems where `int` is 32 bits:

```
bitfield sample { char c : 7; int i : 32; };
bitfield sample { char c : 7; int i; };
```

And here is a 7-bit character as an anonymous `bitfield` type:

```
bitfield { char value : 7 } char_7;
char_7.set_value('a');
```

Of course, if we can transform the declared members to lay them out successively, we could also transform the declared members to overlap them in suitably aligned storage, which brings us to `Union` with similar code...

Note Unlike C and C++17, special language support is not necessary, packing is guaranteed, and because a value's bits are not exposed there is no need to specially ban attempting to take its address.

When adding the concurrency memory model to C++11, we realized that we had to invent a language rule that “a set of contiguous bitfields is treated as one object” for the purposes of the machine memory model. That doesn't need saying here; contiguous bitfield values *are* one object. Further, in C++11 we had to add the wart of a special “:0” syntax (added in C++11) to demarcate a division in a series of bitfields to denote that this was the location to start a new byte and break a series of successive bitfields into groups each so that each group could be treated as its own object in the memory model. Again, that doesn't need saying here; each `bitfield` variable is already an object, so if you want two groups of them to be two objects, just do it: use two `bitfield` objects.

3.10 `safe_union`

A `safe_union` is a `class` where at most one data member is active at a time, and let's just say equality comparison is supported. The metaclass demonstrates how to replace the declared data members with an `active` discriminant and a `data` buffer of sufficient size and alignment to store any of the types. There is no restriction on the number or types of members, except that the type must be copy constructible and copy assignable.

For simpler exposition only (not as a statement on how a variant type should behave), this sample `safe_union` follows the model of having a default empty state and the semantics that if setting the union to a different type throws then the state is empty. A `safe_union` with exactly the C++17 `std::variant` semantics is equally implementable.

```
$class safe_union : final, comparable_value { // no derivation
    constexpr {
        auto objects = safe_union.variables(); // take a copy of the class's objects
        size_t size = 1; // first, calculate the required size
        size_t align = 1; // and alignment for the data buffer
        for (auto o : $safe_union.variables()) {
            size = max(size, sizeof(o.type));
            align = max(align, alignof(o.type));
            if (o.storage.has_default()) o.make_member();
            compiler.require(o.is_member(), "safe_union members must not be static");
            compiler.require(is_copy_constructible_v<o.type>);
        }
    }
};
```

```

        && is_copy_assignable_v<o.type$>,
        "safe_union members must be copy "
        "constructible and copy assignable");
    }

    safe_union.variables().clear(); // now replace the previous instance vars
    -> { alignas(align) byte data[size]; } // with a data buffer
}

int active; // and a discriminant
safe_union() { active = 0; } // default constructor
void clear() { // cleanup goes here
    switch (active) {
        constexpr {
            for (auto o : objects) // destroy the active object
                -> { case o.num$: o.name$.~(o.type.name$)(); }
        }
        active = 0;
    }
}

~safe_union() { clear(); } // destructor just invokes cleanup
safe_union(const safe_union& that) // copy construction
    : active{that.active}
{
    switch (that.active) {
        constexpr {
            for (auto o : objects) // just copy the active member
                -> { case o.num$: o.name$() = that.(o.name)$(); }
        } // via its accessor, defined next below
    }
}

safe_union& operator=(const safe_union& that) { // copy assignment
    clear(); // to keep the code simple for now,
    active = that.active; // destroy-and-construct even if the
    switch (that.active) { // same member is active
        constexpr {
            for (auto o : objects) // just copy the active member
                -> { case o.num$: o.name$() = that.(o.name)$(); }
        } // via its accessor, defined next below
    }
}

constexpr {
for (auto o : objects) -> { // for each original member
    auto o.name$() { // generate an accessor function
        assert (active==o.num); // assert that the member is active
        return (o.type$&)data;
    } // and cast data to the appropriate type&
}
}

```

```

    void operator=(o.type$ value){ // generate a value-set function
        if (active==o.num)
            o.name$() = value; // if the member is active, just set it
        else {
            clear(); // otherwise, clean up the active member
            active = o.num; // and construct a new one
            try { new (&data[0]) o.type.name$(value); }
            catch { active = 0; } // failure to construct implies empty
        }
    }
}
}

bool is_(o.name)$() { // generate an is-active query function
    return (active==o.num);
}

}
}

bool operator==(const safe_union& that) const {
    // (we'll get != from 'comparable_value')
    if (active != that.active) // different active members => not equal
        return false;
    if (active == 0) // both empty => equal
        return true;
    switch (that.active) {
        constexpr {
            for (auto o : objects) // else just compare the active member
                -> { case o.num$: return o.name$() == that.(o.name)$(); }
        }
    }
}

bool is_empty() { return active == 0; }
};

```

Here is code that defines and uses a sample `safe_union`. The usage syntax is identical to C and C++17.

```

safe_union U {
    int i;
    string s;
    map<string, vector<document>> document_map;
};

```

Notes I would be interested in expressing `variant` in this syntax, because I think it's better than writing `variant<int, string, map<string, vector<document>>>` for several reasons, including:

it's easier to read, using the same syntax as built-in unions;

we can give `U` a type that is distinct from the type of other unions even if their members are of the same type;

we get to give nice names to the members, including to access them (instead of `get<0>`).

That we can implement `union` as a library and even get the same union definition syntax for members is only possible because of Dennis Ritchie’s consistent design choice: When he designed C, he wisely used the same syntax for writing the members of a `struct` and a `union`. He could instead have gratuitously used a different syntax just because they were (then) different things, but he didn’t, and we continue to benefit from that design consistency. Thanks again, Dr. Ritchie.

```
U u;
u = "xyzyzy"; // constructs a string
assert (u.is_s());
cout << u.s() << endl; // ok
```

Note I love today’s `std::variant`, but I wouldn’t miss writing the anonymous and pointy `get<0>`.

```
u = map<string, vector<document>>; // destroys string, moves in map
assert (u.is_document_map());
use(u.document_map()); // ok
u.clear(); // destroys the map
assert (u.is_empty());
```

3.11 namespace_class

“In this respect, namespaces behave exactly like classes.”—[Stroustrup, D&E §17.4.2]

“It has been suggested that a namespace should be a kind of class. I don’t think that is a good idea because many class facilities exist exclusively to support the notion of a class being a user-defined type. For example, facilities for defining the creation and manipulation of objects of that type has little to do with scope issues. The opposite, that a class is a kind of namespace, seems almost obviously true. A class is a namespace in the sense that all operations supported for namespaces can be applied with the same meaning to a class unless the operation is explicitly prohibited for classes. This implies simplicity and generality, while minimizing implementation effort.”—[Stroustrup, D&E §17.5]

“Functions not intended for use by applications are in `boost::math::detail`.”—[[Boost.Math](#)]

A `namespace_class` is a `class` with only static members, and `static public` members by default.

First, let’s define a separately useful `reopenable` metaclass – any type that does not define nonstatic data members can be treated as incomplete and `reopenable` so that a subsequent declaration can add new things to the type members:

```
$class reopenable {
    constexpr {
        compiler.require($reopenable.member_variables().empty(),
            "a reopenable type cannot have member variables");
        $reopenable.make_reopenable();
    }
};
```

A `namespace_class` is reopenable:

```
$class namespace_class : reopenable {
    constexpr {
        for (auto m : $reopenable.members()) {
            if (!m.has_access ()) m.make_public();
            if (!m.has_storage()) m.make_static();
            compiler.require(m.is_static(), "namespace_class members must be static");
        }
    }
};
```

These can be used to write types that match that metaclass. Using Boost's Math library as an example:

C++17 style	Using a metaclass
<pre>namespace boost { namespace math { // public contents of boost::math namespace detail { // implementation details of boost::math // go here; function call chains go in/out // of this nested namespace, and calls to // detail:: must be using'd or qualified } } }</pre>	<pre>namespace_class boost { namespace_class math { // public contents of boost::math private: // implementation details of boost::math // go here and can be called normally }; };</pre>

Notes In C++11, we wanted to add a more class-like `enum` into the language, and called it `enum class`. This has been a success, and we encourage people to use it. Now we have an opportunity to give a similar upgrade to namespaces, but this time without having to hardwire a new `enum class`-like type into the core language and plumb it through the core standardese.

This implementation of the `namespace` concept applies generality to enable greater expressiveness without loss of functionality or usability. Note that this intentionally allows a `namespace_class` to naturally have `private` members, which can replace today's hand-coded `namespace detail` idiom.

4 Applying metaclasses: Qt moc and C++/WinRT

Today, C++ framework vendors are forced resort to language extensions that require side compilers/languages and/or extended C++ compilers/languages (in essence, tightly or loosely integrated code generators) only because C++ cannot express everything they need. Some prominent current examples are:

- **Qt moc (meta-object compiler)** (see **Figure 1**): One of Qt’s most common FAQs is “why do you have a meta-object compiler instead of just using C++?”² This issue is contentious and divisive; it has caused spawning forks like [CopperSpice](#) and creating projects like [Verdigris](#), which are largely motivated by trying to eliminating the moc extensions and compiler (Verdigris was created by the Qt moc maintainer).
- **Multiple attempts at Windows COM or WinRT bindings, lately C++/CX (of which I led the design) and its in-progress replacement C++/WinRT** (see **Figures 2 and 3**): The most common FAQ about C++/CX was “why all these language extensions instead of just using C++?”³ Again the issue is contentious and divisive: C++/WinRT exists because its designer disliked C++/CX’s reliance on language extensions and set out to show it could be done as just a C++ library; he created an approach that works for consuming WinRT types, but still has to resort to extensions to be able to express (author) the types, only the extensions are in a separate .IDL file instead of inline in the C++ source.

The side/extended languages and compilers exist to express things that C++ cannot express sufficiently today:

- Qt has to express **signals/slots, properties, and run-time metadata** baked into the executable.
- C++/CX and C++/WinRT has to express **delegates/events, properties, and run-time metadata** in a separate .winmd file.

Note The C++ static reflection proposal by itself helps the run-time metadata issue, but not the others. For example, see [“Can Qt’s moc be replaced by C++ reflection?”](#) in 2014 by the Qt moc maintainer.

There are two aspects, illustrated in Figures 1-3:

- **Side/extended language:** The extra information has to go into source code somewhere. The two main choices are: (1) Nonportable extensions in the C++ source code; this is what Qt and C++/CX do, using macros and compiler extensions respectively. (2) A side language and source file, which requires a more complex build model with a second compiler and requires users to maintain parallel source files consistently (by writing in the extended language as the primary language and generating C++ code, or by hand synchronization); this is what classic COM and C++/WinRT do.
- **Side/extended compiler:** The extra processing has to go into a compiler somewhere. The same choices are: (1) Put it in nonportable extensions in each C++ compiler; this is what C++/CX does. (2) Put it in a side compiler and use a more complex build model; this is what Qt and classic COM and C++/WinRT do.

² The Qt site devotes multiple pages to this. For example, see:

- [“Moc myths debunked / ... you are not writing real C++”](#)
- [“Why Does Qt Use Moc for Signals and Slots”](#)
- [“Why Doesn’t Qt Use Templates for Signals and Slots?”](#)
- [“Can Qt’s moc be replaced by C++ reflection?”](#)

³ C++/CX ended up largely following the design of [C++/CLI](#), not by intention (in fact, we consciously tried not to follow it) but because both had very similar design constraints and forces in their bindings to COM and .NET respectively, which led to similar design solutions. We would have loved nothing better than to do it all in C++, but could not. Still, the “all these language extensions” issue with C++/CLI was contentious enough that I had to write [“A Design Rationale for C++/CLI”](#) in 2006 to document the rationale, which is about the C++/CLI binding to CLI (.NET) but applies essentially point-for-point to the C++/CX binding to COM and WinRT.

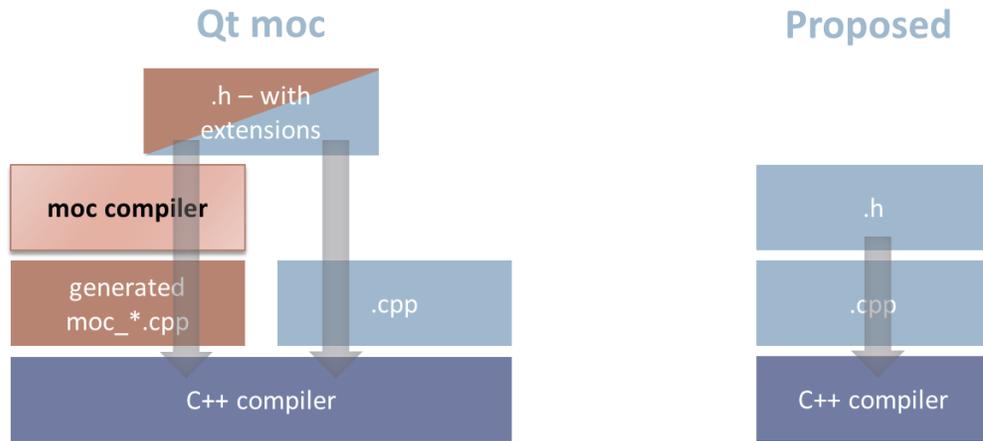


Figure 2: Qt *extended* language + *side* compiler – build model vs. this proposal

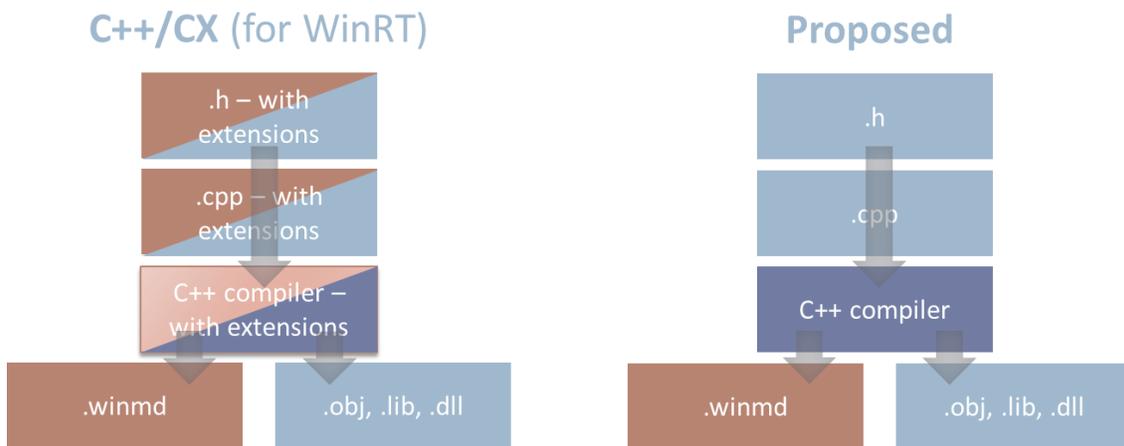


Figure 3: C++/CX *extended* language + *extended* compiler – build model vs. this proposal

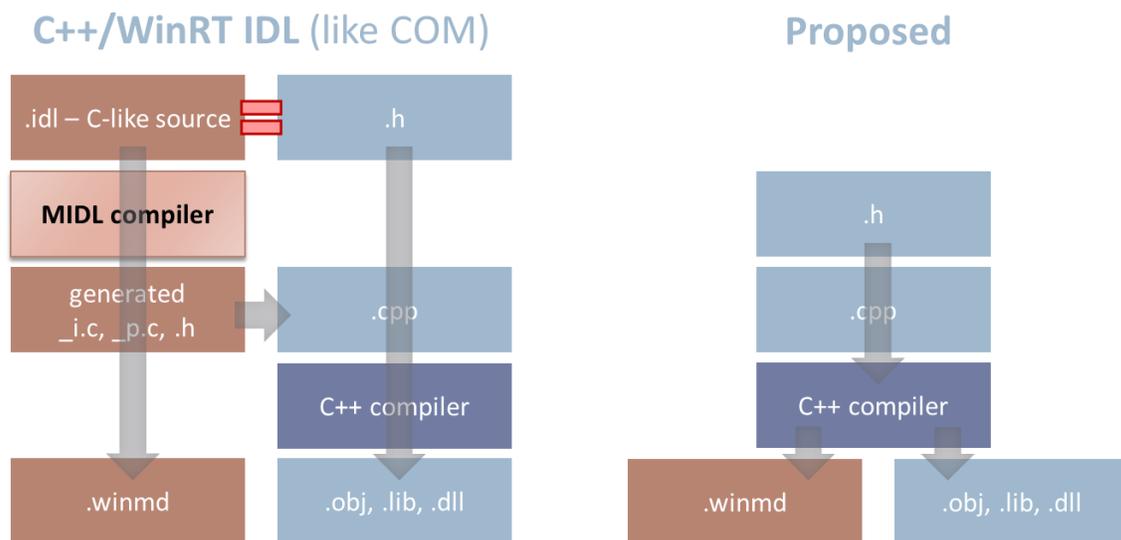


Figure 4: C++/WinRT *side* language + *side* compiler – build model vs. this proposal

4.1 Qt moc → metaclasses (sketch)

This section describes how Qt moc could be replaced by metaclasses.

The approach centers on writing metaclasses to encapsulate Qt conventions. In particular:

Feature	Qt moc style	Proposed
Qt class	<code>: public QObject</code> <code>Q_OBJECT</code> macro	<code>QClass</code> metaclass
Signals and slots	<code>signals:</code> access specifier <code>slots:</code> access specifier Both are grammar extensions	<code>qt::signal</code> type <code>qt::slot</code> type No grammar extensions
Properties	<code>Q_PROPERTY</code> macro	<code>property<></code> metaclass (note: not necessarily specific to Qt)
Metadata	Generated by moc compiler	Generated in <code>QClass</code> metaclass code, or separately by reflection

Consider this example, which uses a simple property for which it's easy to provide a default (as do C# and other languages), and a simple signal (outbound event notification) and slot (inbound event notification):

Qt moc style	This paper (proposed)
<pre>class MyClass : public QObject { Q_OBJECT public: MyClass(QObject* parent = 0); Q_PROPERTY(int value READ get_value WRITE set_value) int get_value() const { return value; } void set_value(int v) { value = v; } private: int value; signals: void mySignal(); public slots: void mySlot(); };</pre>	<pre>QClass MyClass { property<int> value { }; signal mySignal(); slot mySlot(); };</pre>

4.2 QClass metaclass

`QClass` is a metaclass that implements the following requirements and defaults:

- Implicitly inherits publicly from `QObject`.
- Generates a constructor that takes `QObject*` with a default value of `nullptr`.
- Performs all the processing currently performed by the `Q_OBJECT` macro.
- For each nested type declared `property<T>` (see below), “inline” the nested type by moving its data member(s) and function(s) into the scope of this class.

- For each function whose return type is `qt::signal<T>` (see below), change its return type to `T` and treat it as a signal function.
- For each function whose return type is `qt::slot<T>` (see below), change its return type to `T` and treat it as a slot function.
- Performs all the processing currently performed by the `Q_ENUMS` macro to every nested `enum` type.
- (etc. for other `Q_` macros)
- Apply any Qt class rules (e.g., on accessibility of signals and slots).

Note These techniques allow adding “language extensions” that don’t change the C++ grammar:

(1) Using a well-known marker class type as a contextual keyword. By using a well-known type such as `signal` or `slot` as a marker type (for a variable, or a function parameter or return type), a metaclass like `QClass` can assign special semantics and processing to that type when it encounters it in the specially recognized position, essentially turning the type into a contextual keyword but without disturbing the C++ grammar. (The same can be done with variable and function names.)

(2) Using a well-known marker metaclass as a contextual keyword and abstraction. For `property`, we need a little more because it is intended to be an abstraction encapsulating multiple components. Because the C++ grammar already allows nested abstractions (classes), and we are now adding metaclasses, we can simply use a well-known metaclass such as `property` to define a nested class that represents the abstraction. (Processing that is reusable in other places the nested type’s metaclass (e.g., `property`) is useful can be done inside that metaclass, and the combining or post-processing to integrate it into the enclosing `QClass` can be done in `QClass`.)

4.2.1 signal and slot types

The types `qt::signal` and `qt::slot` are ordinary empty types that do nothing on their own, but are used as markers recognized by the `QClass` metaclass.

```
template<class Ret = void> class signal { };
template<class Ret = void> class slot { };
```

These are templates because Qt has some support for non-`void` signal and slot return types. A non-`void` return type can be specified by the template parameter:

```
signal<int> mySignalThatReturnsInt();
slot<Priority> mySlotThatReturnsPriority();
```

Otherwise, a C++17 deduction guide offers nice default syntax without `< >` brackets, as in this section’s example:

```
signal mySignal();           // signal<void>
slot mySlot();              // signal<void>
```

Note Qt itself rarely makes use of non-`void` return types in signal-slot calls. However, slots can also be called like normal functions, so they can return values. For now I’ll leave in this generality of using a template for the return type intact for both signals and slots as it helps to underscore the flexibility that is available with metaclasses; if the generality is not needed for signals, it’s easily removed.

4.2.2 property metaclass

A Qt “property” is modeled as a nested class defined using the metaclass template `qt::property`:

```
template<class T>
$class property<T> {
    // ...
};
```

This metaclass implements the following requirements and defaults (note: strawman that follows the [published Qt rules](#)):

- We recognize as a “get” any function that is `const` and returns `T` or `T const&`.
- We recognize as a “set” any function that is non-`const` and takes exactly one parameter of type `T`, `T&`, or `T const&`.
- We recognize as a “notify” any function declaration that is a `signal` function in the same class.
- Apply any other Qt property rules.

Note We could design a more general “property” that could be standardized and used both here and in the following C++/WinRT section. For now this just illustrating how to create a binding to Qt.

For convenience, an empty `property` that has no user-declared data member or functions:

```
property<T> xxx { };
```

generates the following if `T` is default-constructible:

- a data member named `xxx` of type `T`;
- a “get” function `T get_xxx() { return value; }`; and
- if `T` is not `const`, a “set” function `void set_xxx(const T& value) { xxx = value; }`.

A property can have customizable contents, for example have a different internal type (if Qt allows this):

```
property<string> blob {
    DBQuery q;
    string get_blob() const          { return q.run(“SELECT blob_field FROM /*...*/”); }
    void set_blob(const string& s) { q.run(“UPDATE blob_field /*... using s ...*/”); }
};
```

After the `property` metaclass has been run to define the property’s data and functions as a nested class, the `QClass` metaclass then “inlines” the nested class into the main class so that its data and functions can be used normally by other class members and users.

Note The above shows how to support the basic `Q_PROPERTY` options of `MEMBER`, `READ`, and `WRITE`. To fully support `Q_PROPERTY` semantics, `qt::property` should also support the other options – `RESET`, `NOTIFY`, `DESIGNABLE`, etc.

4.2.3 Generating metadata

Finally, generating metadata is largely enabled by just the reflection proposal on its own, but aided in accuracy by metaclasses. Because we are going to automate Qt conventions using metaclasses such as `QClass`, the source code directly identifies exactly which types are Qt types.

- As each such type is defined by applying the metaclass, the metaclass’s code can use reflection at the time each `QClass` is processed to generate compile-time data structures for metadata.

- Alternatively, a `generate_metadata` function could reflect over the whole program to identify and inspect Qt types and generate metadata only for those; that function can be built and invoked as a separate executable. This keeps the metadata generator code outside the metaclass code, if that is desirable.

In both cases, all processing is done inside the C++ program and C++ compiler.

5 Alternatives for source→definition transform

This section explores some alternative ways to express the source→definition transformation. Note that the code within the metaclass is structurally the same under these alternatives. In this section:

- the *source class* (input) means the class as written by the user in source code interpreted without any special rules being applied, not even the usual rules for `class` and `struct` (e.g., the default accessibility of all members is “none,” neither private nor public); and
- the *defined class* (output) means the class that is generated as a result of applying the metaclass’s logic to the source class.

Note Any metaclass can still inject additional output classes, free functions, etc. under any option.

5.1 Metaclass, with modify-in-place semantics

Summary: The members are as defined in the source, and we can then modify them in-place and build the defined class once which is read-only after the metaclass logic finishes. (This is the style in R0 and R1 of this paper.)

Semantics:

- The initial contents of the metaclass are exactly contents those of the source class.
- The name of the metaclass (e.g., `interface`) refers to both the source class members (initially) and to the defined class as it is generated.
- Any injections made in a `constexpr` block become visible upon exit of the `constexpr` block. For example, a `constexpr` block that injects members cannot reflect injected members within the same block.

Drawbacks:

- Composes using inheritance-like structure.
- We lack distinct names to refer to the source and defined classes, and cannot refer back to the source class state after changes have begun (without explicitly taking a copy at the start).
- After we process the metaclass the source class is no longer available (unless an explicit copy is made).

Example:

```
$class has_virtual_destructor { virtual ~has_virtual_destructor() noexcept {} };

$class interface : has_virtual_destructor {
    constexpr {
        compiler.require($interface.variables().empty(), "interfaces may not contain data");
        for (auto f : $interface.functions()) {
            compiler.require(!f.is_copy() && !f.is_move(),
                "interfaces may not copy or move; consider a virtual clone()");
            if (!f.has_access()) f.make_public();
            compiler.require(f.is_public(),
                "interface functions must be public");
            f.make_pure_virtual();
        }
    }
};
```

5.2 Metaclass, with read-only `prototype` + define-once semantics

Summary: Have distinct names for the source `prototype` (which is read-only) and the generated class definition which is being built (once, after which it too is read-only).

Semantics:

- The source class is stored somewhere unspecified. An implementation may put it in a nested helper namespace, a nested class, or elsewhere.
- The initial contents of the metaclass is an implementation-generated using-alias named `prototype` which refers to the source class, which gives the program a way to refer to it (a permanent way, unless the metaclass's logic goes out of its way to explicitly remove this alias which would be weird).
- The name `prototype` refers only to the source class, and is read-only.
- The name of the metaclass (e.g., `interface`) refers only to the defined class.
- Any injections made in a `constexpr` block become visible upon exit of the `constexpr` block. For example, a `constexpr` block that loops over members and also injects members cannot see the injected members within the same block.

Advantages:

- Distinct names to refer to the source and defined classes.
- Source class is read-only and always available.

Drawbacks:

- Composes using inheritance-like structure.
- The code needs to inject each item into the destination class as it goes, whereas with the prior approach all items are in the destination class by default.

Example:

```
$class has_virtual_destructor { virtual ~has_virtual_destructor() noexcept {} };
$class interface : has_virtual_destructor {
    constexpr {
        compiler.require($prototype.variables().empty(), "interfaces may not contain data");
        for (auto f : $prototype.functions()) {
            compiler.require(!f.is_copy() && !f.is_move(),
                "interfaces may not copy or move; consider a virtual clone()");
            if (!f.has_access()) f.make_public();
            compiler.require(f.is_public(),
                "interface functions must be public");
            f.make_pure_virtual();
        }
        -> f;
    }
};
```

5.3 Functional style: Pass and fill `meta::type` parameters

Summary: Similar to previous, but expressed as a compile-time function.

Semantics:

- The source is a read-only input parameter of type `meta::type`.
- The target generated class definition is an in-out parameter of type `meta::type`. (Note: The reason not to make the target just a return value is that we want to compose calls to delegate work to other meta-class functions to incrementally build types, which requires the ability to pass to that function the `type` generated so far so it can apply further work and make those results visible to its caller.)
- Inside injection blocks in particular, we use `this_class` to refer to the name of the type itself, for example to mention the name as a function name for the special member functions, or as a parameter type.
- Still create a `prototype` alias in the generated class so that the source class can be referred to later.

Advantages:

- Composes using function-call structure, including natural support for conditional composition (`if(something) other_metaclass(source, target);`).
- Distinct names to refer to the source and defined classes.
- Source class is read-only.

Drawbacks:

- The code needs to inject each item into the target as it goes.
- We additionally need to invent a way to “inject” a member (e.g., a function) to a compile-time `meta::type` variable, rather than a scope. (We may need a way to do this anyway though.)

Example:

```
void has_virtual_destructor(const meta::type source, meta::type& target) constexpr {
    ->(target) { virtual ~has_virtual_destructor() noexcept { } }
}

void interface(const meta::type source, meta::type& target) constexpr {
    has_virtual_destructor(source, target);

    compiler.require($source.variables().empty(), "interfaces may not contain data");
    for (auto f : $source.functions()) {
        compiler.require(!f.is_copy() && !f.is_move(),
            "interfaces may not copy or move; consider a virtual clone()");
        if (!f.has_access()) f.make_public();
        compiler.require(f.is_public(),
            "interface functions must be public");

        f.make_pure_virtual();
        ->(target) f;
    }
    ->(target) { virtual ~interface() noexcept { } };
}
```

6 Alternatives for applying the transform

This section explores some alternative ways to apply the source→definition transformation. Note that the code within the class being defined is structurally the same under these alternatives.

In this section “metaclass” means the name given to the transformation.

Unlike the previous section, these alternatives are not mutually exclusive.

6.1 Terse/natural syntax: In place of `class`

Summary: The metaclass name appears in place of `class` when defining classes and in constrained template arguments (as with concepts).

Note: The rest of this section assumes that this syntax is supported regardless of which of the other syntaxes that follow in are also pursued. Those other syntaxes should be in addition to, not instead of, this syntax.

Advantages:

- Clarity for code authors and readers: This is the “terse syntax” for applying metaclasses, and important for all the reasons the terse syntax is important for applying concepts.
- No parsing ambiguity.
- Symmetry with applying concepts as constraints.

Limitations:

- Allows exactly one metaclass name to be applied to a class definition. If this were the only style supported, a class that wants to apply multiple unrelated metaclasses must define a new metaclass to give a name to a combination of the metaclasses. Personally, I do not view this as an important limitation because it is normally both self-documenting and reuse-promoting to give a name to the combined metaclass; naming it captures the intent of the combiner, and promotes using the name again.
- (Of course, this limitation goes away if other styles in this section are supported as well.)

Example:

```
// to apply one metaclass named interface
interface myclass {
    // ... etc. ...
};

// to apply multiple metaclasses M1 and M2
$class M1M2 : M1, M2 { };
M1M2 myclass {
    // ... etc. ...
};

// to constrain a template parameter using one metaclass named interface
template<interface T> void f(T);

// to constrain a template parameter using multiple metaclasses M1 and M2
$class M1M2 : M1, M2 { };
template<M1M2 T> void f(T);
```

6.2 As adjectives before `class`

Summary: A whitespace-delimited list of metaclass names appear as adjectives before `class` when defining classes, and as adjectives before `class` or `typename` in constrained template arguments.

Semantics:

- The terse syntax `M myclass{};` becomes a shorthand for `M class myclass{};`.

Advantages:

- Clarity for code authors and readers: Preserves the “terse syntax” of keeping the more-specialized word of power up front, albeit with a bit of “syntax boilerplate.”
- Allows multiple metaclasses to be listed.
- Extends the naming pattern of C++11’s own `enum class`. Secondly, we have experience that commercial nonstandard extensions like C++/CLI’s and C++/CX’s `interface class` and `ref class` are adoptable by users, and that users like them (except for their nonstandardness, but not as far as we know because of their naming convention).

Drawbacks:

- If we seriously want to explore this, we should do a UX study to see how users react to the “redundant boilerplate,” because we know C++ developers actively complain about the boilerplate the language already requires.
- For constraints, a novel syntax that is not consistent with how we apply concept constraints. If we were to support this syntax for metaclass constraints, we should support it for concept constraints as well.

Example:

```
// to apply one metaclass named interface
interface class myclass {
    // ... etc. ...
};

// to apply multiple metaclasses M1 and M2
M1 M2 class myclass {
    // ... etc. ...
};

// to constrain a template parameter using one metaclass named interface
template<interface class T> void f(T);
template<interface typename T> void g(T);

// to constrain a template parameter using multiple metaclasses M1 and M2
template<M1 M2 class T> void f(T);
template<M1 M2 typename T> void g(T);
```

6.3 As “specializations” of `class<>`

Summary: The metaclass name appears as a `<>`-enclosed comma-delimited list after `class` when defining classes, and after `class` or `typename` in constrained template arguments.

Advantages:

- Allows multiple metaclasses to be listed.
- Symmetry with specialization syntax, which suggests that the metaclass name(s) are specializations of the general “class” concept which is true in that the generated class is “still just a class.”

Drawbacks:

- Symmetry with specialization syntax, which suggests that the metaclass name(s) are specializations of the general “class” concept, which is untrue in the detail that the default “metaprogram for class” that applies rules like “default private,” “implicit virtual,” etc. is not run, we are running this metaclasses’ rules instead.
- For constraints, a novel syntax that is not consistent with how we apply concept constraints. If we were to support this syntax for metaclass constraints, we should support it for concept constraints as well.

Example:

```
// to apply one metaclass named interface
class<interface> myclass {
    // ... etc. ...
};

// to apply multiple metaclasses M1 and M2
class<M1,M2> myclass {
    // ... etc. ...
};

// to constrain a template parameter using one metaclass named interface
template<class<interface> T> void f(T);
template<typename<interface> T> void g(T);

// to constrain a template parameter using multiple metaclasses M1 and M2
template<class<M1, M2> T> void f(T);
template<typename<M1, M2> T> void g(T);
```

6.4 In the class body (primarily motivated by transitional uses)

Summary: The metaclass name can be applied under some syntax within the class body.

Advantages:

- Allows multiple metaclasses to be listed.
- Allows existing macro-based language extensions to (e.g., Qt macros) to change their existing macros to apply metaclasses to existing code as a transitional tool (e.g., within `Q_OBJECT`). That permits code written using the existing macros targeting a separate proprietary compiler to be recompiled without source changes in a metaclass-based implementation.

Drawbacks:

- Does not extend symmetrically to applying a metaclass as a constraint.
- Naturally supports for conditional composition (`if(something) $other_metaclass`).
- To be useful in migration of existing code such as Qt macros, which uses macros like `Q_OBJECT` typically at the top, we would probably be forced to make the position of the directive not matter and apply to all declarations including those following the directive.

Example:

```
// to apply one metaclass named interface
class myclass {
    $interface;                // placeholder for some other syntax
    // ... etc. ...
};

// to apply multiple metaclasses M1 and M2
class myclass {
    $M1; $M2;                // placeholder for some other syntax
    // ... etc. ...
};
```

7 Tooling

Every abstraction that C and C++ have ever added works without tooling, and also benefits from tooling:

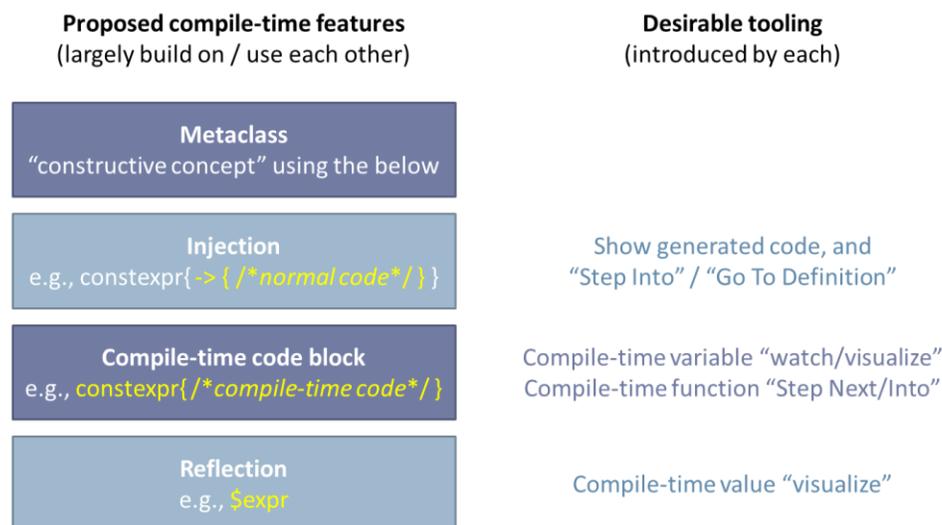
Abstractions are **hid**ers ⇒ benefit from tool support

C	Variables: hide values ⇒ watch windows (debug) Functions: hide code ⇒ Go To Definition (IDE) / Step Into (debug) Pointers: hide indirection ⇒ visualizers (debug) #includes: hide dependencies ⇒ file “touch”-aware build (build)
C++98	Classes: hide code/data, encapsulate behavior ⇒ most of the above Overloads: hide static polymorphism ⇒ better warning/error msgs Virtuals: hide dynamic polymorphism ⇒ dynamic debug support
C++17	constexpr functions: hide computations ⇒ compile-time debug if constexpr: hide whether code even has to compile ⇒ colorizers
Proposed	Modules: hide dependencies ⇒ extract DAG + module “touch”-aware build Compile-time variables: hide values ⇒ compile-time watch Compile-time code/functions: hide computation ⇒ compile-time debug Injection: generate entities ⇒ visualize to enable the usual above

In each case above:

- *The feature is usable before tooling.* For example, absent other tool support, C++ programmers use `printf`-style debugging to see variable values, we figure out overload candidates by inspection to debug why we can’t call an overloaded function, and we manually inspect and imagine specialization instantiations to figure out the outcome of a template metaprogram.
- *The feature, no matter how basic, benefits from tools to “look inside the abstraction.”* For example, C++ debuggers now routinely offer watch windows to see variable values, and compilers routinely show overload candidates when we can’t call an overloaded function. (TMP remains hard to write, read, and tool; so we should replace indirect TMP with direct compile-time `constexpr` code that’s much easier to write, read, and tool... and then apply the tooling we have for ordinary code to that compile-time code.)

Metaclasses build on injection, which builds on compile-time code blocks, which uses reflection. The bottom three of those layers will benefit from tooling:



Importantly, note that metaclasses themselves do not add a major new tooling requirement. The three layers they depend on, and which we should adopt into C++ anyway in isolation, do – and once we have them, there is no primary new kind of tooling required by metaclasses.

As an example of tooling for metaclasses, when the user writes this source class:

```
value Point {
    int x;
    int y;
    Point(int, int);
};
```

and the metaclass program generates this class definition:

```
class Point {
private:
    int x = 0;
    int y = 0;
public:
    Point(int, int);
    Point() = default;
    Point(const Point&) = default;
    Point(Point&&) = default;
    Point& operator=(const Point&) = default;
    Point& operator=(Point&&) = default;
    auto operator<=>(const Point&) = default;
};
```

then how do we show (visualize, “see”) the defined class?

Without any special tooling, this proposal provides the minimum guaranteed level of “printf-style debugging”:

```
constexpr{ compiler.debug($Point); } // we can always print what’s generated
```

Additionally, an IDE could for example offer a button beside to switch between viewing the source class (editable) and the defined class (noneditable), and additionally use the latter for its existing Step Into behavior.

Note Any IDE that does this should immediately work better for existing C++17 code. For example, doing this enables Step Into for today’s special member functions, which is already something most (>50%) C++ developers wish they had but most (>95%) do not have (source: poll of audiences, N ≈ 2,000).

If the source class is a template, such as

```
template<class T> customized_type MyClass { /*...*/ };
```

so that applying a metaclass could in general generate different things in each instantiation depending on the properties of type `T`, the IDE can still allow the same view-switching on at least each instantiation `MyClass<T>`. For example, for a given variable `var` whose type is an instantiation `MyClass<SpecificType>`, performing Step Into a call `var.func()` goes to the defined type for `MyClass<SpecificType>` which is concrete and unique.

8 Revision history

R2 (pre-Albuquerque, 2017-10):

- Expanded section 2.5, “Composition,” to discuss composability.
- Added new sections 5, 6, and 7 in response to Toronto feedback and for discussion in Albuquerque.

R1 (post-Toronto, 2017-07):

- Minor tweaks from Toronto.

R0 (pre-Toronto, 2017-06): Initial revision.