Consteval-only Types

A design space overview

Wyatt Childers <wchilders@nvidia.com>

Foreword

This paper contains high level design points, points of potential future design direction, and an exploration of problems that need to be considered in the C++26 time frame (along with proposed resolutions). If you're coming to this paper to understand only the basics of consteval-only type motivation and the aforementioned issues and the proposed resolutions please read the sections:

- Introduction
- Uninstantiated Types
- Incomplete Types
- Other Options
- Conclusion

Introduction

It's been said of many things over the years that the question is not a question of if, but when. This committee has voted into C++26 the notion of a **std::meta::info** type. Nominally, this type is a scalar; however, it is immediately obvious that this type is distinct from int, long, unsigned long, and friends.

This paper looks to make the case that this distinction is meaningful in a way that should be captured formally in the type system. This is not about leaking meaningless values to runtime, and this is also not about the diagnostic problems that spurred the initial research into consteval-only types; it is instead about ensuring that C++26 sets a good foundation for everything we conceivably may want to do with interpreted types, empowering language evolution to solve the problems we'll have in the coming decades with increasingly complicated compile time programming.

It is through that solid foundation, that we are able to prevent leakage of meaningless scalar values to runtime (with a 100% success rate) and improve diagnostics using consteval-only types for C++26.

Invariants

If we take a step back and talk about types, a fundamental tenet is that types have invariants. These invariants are ultimately what drive many of the semantics of a C++ program. C++ would not be where it is today without RAII and RAII could not exist without a notion of invariants.

When we talk about invariants, we are sometimes referring to language facilitated invariants. Other times we are talking about invariants laid out by the author of the class and conveyed via comments or assertions in the function body.

As users of the language: we do our best to use the facilities provided by the language to allow the language to help us catch a mistake. For instance, we might delete the copy constructor if there is no reasonable way to express a copy operation for the given type. This empowers static analysis to tell users of our types that they have made a mistake and they need to correct it.

As designers of the language: we must look out for these opportunities to empower static analysis and not create holes in the type system that undermine its ability to catch our mistakes.

Informal Consteval-only Types

Code Generation

Consider the following:

```
struct type_pretty_printer {
  consteval type_pretty_printer() = default;
  consteval std::string format(meta::info type);
private:
  std::unordered_map<meta::info, std::string> cache;
};
```

This is a small type that conceptually would walk the members of a type, create a pretty rendering of the type (which in an interpreter is conceivably expensive enough to cache), cache it, and return the cached string.

Now the question is, does this type ever make sense at runtime? To which the – uncontestable – answer is no. So, we've created a type that's purely functional at compile time.

We can acknowledge that with a code comment and make this an informal invariant. We can have an informal notion within the community that this is a "consteval-only type" because all of its values only make sense at compile time.

Since this is an informal notion, if I write:

```
int main() {
  type_pretty_printer x;
  return 0;
}
```

There are no language rules that prevent me from doing that (assuming non-transient allocation worked for this map). We can simplify this a bit to remove the non-transient allocation from our train of thought:

```
struct rendered_type {
  meta::info type_reflection = ^^int;
  cont char* formatted_string = “int”;
};

int main() {
  rendered_type x;
  return 0;
}
```

Again, we have a type that is only whole when it’s at compile time; it is informally a consteval-only type. As this is an information notion, there are no language rules that prevent me from using it at runtime. That requires a layout to be defined by the language or the implementation because this needs to exist in the binary.

For ease of implementation, since we said **std::meta::info** is a scalar, we can just pick an integer size, say 64-bit (or perhaps whatever the bitwidth of int or long is on the given architecture), and generate a runtime layout. We can just leave all **std::meta::info** values uninitialized, zero them out, or assign a random number there.

Put another way, implementations will rephrase the above program as something like the following program:

```
struct rendered_type {
  long       type_reflection = 0;
  cont char* formatted_string = “int”;
};

int main() {
  rendered_type x;
```

```
  return 0;
}
```

which allows for a successful compilation. Notably this wasn't the design of our original type though, as a programmer, we intended for this type to be used at compile time only.

While the extra bits are trivial and likely harmless here (after all the optimizer should just remove this dead code), in a more complicated program, the impact might be more abstract; or in other words, it might be harder to spot your design flaw that resulted in this useless runtime object.

Diagnostics

A common problem was observed when developers first encountered reflection. They would write something conceptually like:

```
struct hello_world;

int main() {
  std::cout << identifier_of(^^hello_world) << '\n';
  return 0;
}
```

which works; but then when they'd rephrase this a bit and start adding more code suddenly it doesn't:

```
struct hello_world;

int main() {
  auto refl = ^^hello_world;
  std::cout << identifier_of(refl) << '\n';
  return 0;
}
```

They would then get a series of diagnostics like the following:

```
<source>:8:29: error: call to consteval function
'std::meta::identifier_of(refl)' is not a constant expression
   8 |   std::cout << identifier_of(refl) << '\n';
     |                ~~~~~~~~~~~~~^~~~~~
<source>:8:30: error: the value of 'refl' is not usable in a
constant expression
```

```
   8 |   std::cout << identifier_of(refl) << '\n';
     |                              ^~~~
<source>:7:8: note: 'refl' was not declared 'constexpr'
   7 |   auto refl = ^^hello_world;
     |        ^~~~
```

The very last diagnostic in the series tells the user what the actual problem is. Worse, these diagnostics must be repeated for every function call that uses this improperly declared variable. That is potentially a very large number of diagnostics for a very simple mistake. This is the kind of problem that C++ compilers are ridiculed for. The information to fix the problem is there, but it's in the least helpful form possible.

Formal Consteval-only Types

What if we instead said that, an informal notion of consteval-only types is not good enough? That has some favorable implications.

These consteval-only types allow us to derive notions like consteval-only variables. This then empowers the compiler to inform the user that their code is invalid. The above can now simply be diagnosed as:

```
<source>:7:17: error: consteval-only variable 'refl' not declared
'constexpr' used outside a constant-evaluated context
   7 |   auto refl = ^^hello_world;
     |                 ^~~~~~~~~~
<source>:7:8: note: add 'constexpr'
   7 |   auto refl = ^^hello_world;
     |        ^~~~
```

This is a notable improvement in diagnostic clarity and tells the user *exactly* what they need to do. It is then much easier as a matter of QOI to not diagnose on subsequent function calls (and not get the spurious indirect errors about our call to **identifier_of**).

We similarly no longer have any code generation obligations for a stray **std::meta::info** variable, as there is no longer a valid notion of a runtime **std::meta::info** variable.

Consteval-only Type Propagation

Continuing down that line of thought, as a matter of practicality it would not be very useful if this is invalid:

```
int main() {
```

```
  std::meta::info x;
  return 0;
}
```

however, this is not:

```
struct info_wrapper {
  std::meta::info value;
};

int main() {
  info_wrapper x;
  return 0;
}
```

If that were the case, there would still be a notion of a runtime **std::meta::info** variable (by virtue of it being a data member). By propagating the consteval-only type status to the enclosing class, we get the same semantics regardless of the level of indirection imposed by any encapsulating class(es).

That also means that if we expand our example:

```
struct hello_world;

struct info_wrapper {
  std::meta::info value;
};

int main() {
  info_wrapper refl{^^hello_world};
  std::cout << identifier_of(refl.value) << '\n';
  return 0;
}
```

we get the same good diagnostics:

```
<source>:11:34: error: consteval-only variable 'refl' not declared
'constexpr' used outside a constant-evaluated context
  11 |   info_wrapper refl{^^hello_world};
     |                                  ^
```

```
<source>:11:16: note: add 'constexpr'
  11 |   info_wrapper refl{^^hello_world};
     |                ^~~~
```

Type Erasure

At this point, the initial set of problems is solved. However there are further implications for the language. Notably, we cannot allow a consteval-only type to become erased. Thus this must be invalid:

```
int main() {
  constexpr std::meta::info refl = ^^int;
  void*                     refl_ptr = (void*)&refl;
}
```

This is not a problem that cannot be overcome. The notion of consteval-only value could actually serve as a means to allow some cases of erasure to be allowed. However, it is not something that is solved by consteval-only types.

Notably, it is not something that impedes any of the many practical examples outlined by P2996 or that have been thus far raised by the community. This is already addressed in P2996 and the current working draft.

Immediate Escalation

Consteval-only types play a role in the status quo immediate escalation design. The following rules in the working draft enable a number of cases of immediate escalation:

> A potentially-evaluated expression or conversion is immediate-escalating if it is neither initially in an immediate function context nor a subexpression of an immediate invocation, and
> …
> – it is of consteval-only type
>
> An immediate function is a function that is:
> …
> – an immediate-escalating function whose type is consteval-only
> …
> – an immediate-escalating function *F* whose function body contains either:
> …
> – a definition of a non-constexpr variable with consteval-only type

This in turn means that if you feed a consteval-only type to a template like:

```
template<typename T>
constexpr void foo(T x);

foo(^^int);
```

or have an expression (or variable) of consteval-only type, the function call to foo is guaranteed to be an immediate invocation. This in turn ensures that the invocation of **foo** occurs at compile time as opposed to (potentially) attempting to call a runtime branch of constexpr **foo**.

Consteval Virtual Functions

On a related note, constexpr exceptions were adopted for C++26 reflection (P3560R0) as an error handling mechanism for reflection. This is quite useful but it also had the effect of creating an exception hierarchy (considering the following promotion rule) like so:

```
struct exception {
  virtual const char* what() const noexcept;
};

struct meta_exception : exception {
  consteval meta_exception(u8string_view   what,
                           info            from,
                           source_location where);
  consteval virtual const char* what() const noexcept;
};
```

Examining this example, **meta_exception** almost certainly must be a consteval-only type (unless no **std::meta::info** is a data member of the exception type). This data member in turn would cause consteval-type propagation. That then enables **meta_exception** to override **exception::what** in a way that's observable at compile time only, and also mandates that what is consteval (even if originally spelled constexpr as its implicit "this" argument is consteval-only). The combined effect is that this example has well defined semantics.

The formal consteval-only type semantic ensures that we do not have a special case rule where a **meta_exception** can exist at runtime meanwhile its virtual function definition cannot.

Notably, the core language currently prescribes an additional rule based on this premise:

> A class with a consteval virtual function that overrides a virtual function that is not consteval shall have a consteval-only type.

This rule is somewhat backwards but in a way that's perhaps convenient; instead of requiring that **meta_exception** be consteval-only it *imposes* that **meta_exception** *is* consteval-only by virtue of its virtual member function. That means that the following is a consteval-only type:

```
struct const_what_exception : exception {
  consteval virtual const char* what() const noexcept;
};
```

even though there are no members to make **const_what_exception** consteval-only. This rule thus ensures that you get a working consteval-only type instead of an error.

Unions

The version of consteval-only types that is described by P2996 defines consteval-only type propagation as applying to unions (and thus their enclosing classes). This means that a type like:

```
struct refl_or_ti {
  union {
    info      refl;
    type_info ti;
  } value;
};
```

is considered to be a consteval-only type. This has been raised as a potential problem, possibly impacting the ability of C++ programmers to author high quality runtime reflection libraries based on compile-time reflection. However, no practical examples have been produced that hit this requirement.

In fact, Andrew Sutton gave a talk at CppCon 2019 (https://www.youtube.com/watch?v=ARxj3dfF_h0); as part of this talk, he presented "Nemesis: a library for runtime introspection" (starting at ~45:50). This library is designed following patterns that will be familiar to any programmers familiar with the common patterns of code generation. Namely, you examine some source material (in this case the source code of your C++ program via reflection) and then you use that to build out runtime state (in this case by instantiating templates that generate runtime algorithms and state). These runtime state objects do not contain the compile time state objects used to generate them; that would simply be unnecessary overhead for the runtime environment as the values are meaningless.

It is thus highly unlikely that outside of a toy program, you would ever see this phenomenon.

That said, if we get to C++29 and we say "this is actually a really useful case, we need to be able to have types that are – via a union – compile time only or runtime only" we can still do that.

The initial scope of consteval-only types is intentionally impermissive. This is because taking permissive approaches are much harder to walk back after you have more information. In C++29 we could easily say – having gained the insight and better understanding of the context – that the above **refl_or_ti** type is not consteval-only. The only thing we would be changing is that the following example would go from invalid to valid:

```
int main() {
  refl_or_ti x;
  return 0;
}
```

It is then conceivable, that rules could be written that limit access of consteval-only data members from non-consteval-only values:

```
int main() {
  refl_or_ti                x{0};
  constexpr std::meta::info x_refl = x.value.refl; // invalid
  constexpr refl_or_ti      y{^^int};
  constexpr std::meta::info y_refl = y.value.refl; // okay

  return 0;
}
```

These exact rules however, have not been thought through; attempting to come up with them at this stage would be irresponsible, especially given the extremely unlikely impact on any anticipated reflection use cases.

Uninstantiated Types

D4101R0 (https://isocpp.org/files/papers/D4101R0.html) raises an issue about the instantiation of pointers to template specializations:

```
template<typename T>
struct S;

void f(S<int>* x);
```

A perhaps more appropriate example would however be the following:

```
template<typename T>
```

```
struct pointee {
  static_assert(^^TT != ^^int);
};

struct wrapper {
  pointee<int>* member;
};
```

In the above code example, the assertion surprisingly will not trigger. This is because – strictly speaking – nothing has been done that mandates an instantiation.

When we consider type propagation and this problem, the issue becomes apparent:

```
template<typename T>
struct pointee {
  T pointee_member;
};

struct wrapper {
  pointee<std::meta::info>* member;
};
```

In this situation, **wrapper** should be a consteval-only type because **pointee<std::meta::info>** (once instantiated) is a consteval-only type. However, because the instantiation does not occur, **wrapper** will not be marked as a consteval-only type.

Unfortunately, the language has too much history to make this a mandated instantiation; doing so in the general case would break a large amount of real world code.

Eager Consteval-only Type Decision

This leaves us with a couple of reasonable options.

The first option (which is closer to the historic working model of consteval-only types) is to be eager. We can say that since we don't know whether **pointee<std::meta::info>** is consteval-only or not, we're going to assume it isn't. That then results in **wrapper** *not* being a consteval-only type.

Under this model, if we at any point later in the program instantiate **pointee<std::meta::info>** and that instantiation produces a consteval-only class type, the program is ill-formed.

If users wanted to prevent this outcome for the case above, they could explicitly instantiate **pointee<std::meta::info>** prior to the definition of **wrapper** as in either of the below examples:

```
// EXAMPLE A: makes this okay
constexpr pointee<int> x;

// EXAMPLE B: makes this okay
template struct pointee<int>;
```

This flows naturally with the existing accumulation of properties that occurs in a C++ program. As an example, consider the following:

```
struct A;
struct B;

B* foo(A* a)
{
   return static_cast<B*>(a);
}

struct B : A { };
```

if you want **foo** to be valid, the compiler must know more about the types **A** and **B**.

Template Argument Eager Instantiation

This approach can be further enhanced by requiring the instantiation of any specialization that uses template arguments that are consteval-only types or consteval-only values.

This allows the compiler to have the full picture when considering whether or not the wrapper type is consteval-only above. As there are no historical consteval-only types, this is also fully backwards compatible.

A caveat of this enhancement is that the template argument list does not always directly express the types being substituted. For instance, if the above example is changed like so:

```
template<typename T>
struct helper;

template<>
struct helper<int> {
  using type = std::meta::info;
};
```

```
template<typename T>
struct pointee {
  helper<T>::type pointee_member;
};

struct wrapper {
  pointee<int>* member;
};
```

The instantiation **pointee<int>** is still a consteval-only type, however we no longer have any hints in the **wrapper** type that we need to eagerly instantiate **pointee<int>** type.

Thus while the enhancement is useful for many common cases, it is not a universal solution and the user would have to (as they would without the enhancement) either use **pointee<int>** in a way that requires instantiation or explicitly instantiate **pointee<int>** so that wrapper is correctly marked as a consteval-only type.

Delayed Consteval-only Type Decision

An alternative model is to delay the decision on whether or not a type is consteval-only until storage is required.

Looking at the prior example again:

```
struct wrapper {
  pointee<int>* member;
};
```

the answer to “is **wrapper** a consteval-only type?” is at this point in the program “maybe” instead of “no.” More concretely, types are assigned one of three states “Yes”, “No”, or “Maybe”.

There are three possible paths for some **pointee<T>** to explore:

1. **pointee<T>** is a consteval-only type upon instantiation
    a. **pointee<T>** is instantiated before use
    b. **pointee<T>** is instantiated after use
2. **pointee<T>** is not a consteval-only type upon instantiation
    a. **pointee<T>** is instantiated before use
    b. **pointee<T>** is instantiated after use
3. **pointee<T>** is never instantiated

Consteval-only Upon Instantiation

We can use **pointee<int>** to continue and examine this first possibility for **pointee<T>**:

```
struct wrapper {
  pointee<int>* member;
};

template struct pointee<int>;
```

At this point **wrapper** is still “Maybe” a consteval-only type. Now we declare our initial variable:

```
constexpr wrapper k{nullptr};
```

We must now decide either “Yes” or “No” for all types involved in this variable’s type; “Maybe” is no longer an acceptable answer.

We use the exact same analysis that we used in the eager consteval-only types strategy however the instantiation is now allowed after definition and before use: **pointee<int>** is complete and it’s consteval-only, so **wrapper** is a consteval-only type.

If we remove the explicit instantiation from the equation … and suppose we have non-transient allocation for a second:

```
constexpr wrapper k{new pointee<int>(^^int)}
```

we realize a major benefit of the delayed model; the expression initializing the variable needs the complete type to initialize the variable and *that* provides the instantiation for the user in even the general case.

If we have no instantiation prior to the variable declaration even in the initializer this model also handles things quite eloquently. Consider the following:

```
constexpr wrapper k_ct{nullptr};
```

at this point there is no instantiation of **pointee<int>** in the program; **pointee<int>**’s specialization is marked as not consteval-only (since the type appeared in a variable its consteval-only status is frozen) and **wrapper** is not a consteval-only type.

Now let’s explicitly instantiate **pointee<int>**:

```
template struct pointee<int>;
```

At this point we instantiate **pointee<int>** resulting in a consteval-only type. However, we also at this point note that **pointee<int>** in its incomplete form was marked not consteval-only. This is a very quick and low overhead check for an implementation to perform.

Not Consteval-only Upon Instantiation

To explore the second possibility, consider the following:

```
template<>
struct helper<long> {
  using type = long;
};

struct wrapper {
  pointee<long>* member;
};

template struct pointee<long>;
```

As previously, at this point wrapper is still “Maybe” a consteval-only type. Now we declare our initial variable:

```
constexpr wrapper k{nullptr};
```

We must now decide either “Yes” or “No” for all types involved in this variable’s type. There’s nothing here’s that a consteval-only type, so this is not a consteval-only type.

Similarly, if we change this so our instantiation of `pointee<long>` occurs after our variable declaration:

```
constexpr wrapper k_ct{nullptr};
template struct pointee<long>;
```

absolutely nothing changes our assumption was that the specialization would not be a consteval-only type and that assumption matches the reality upon instantiation. The code is completely sound and compiles as expected; this is very boring and boring is good.

Never Instantiated

This case is covered in part by the consteval-only upon instantiation case as part of the exposition of the approach there. However, to reiterate, if we have no instantiation prior to the variable declaration:

```
constexpr wrapper k_ct{nullptr};
```

at this point there is no instantiation of **pointee<int>** in the program; **pointee<int>**'s specialization is marked as not consteval-only (since the type appeared in a variable its consteval-only status is frozen) and **wrapper** is not a consteval-only type.

Any attempts to instantiate **pointee<int>** (and thus potentially cause a problem) will be caught by the aforementioned mechanisms.

Incomplete Types

In a related vein, C++ has this notion we're all familiar with of incomplete types. This poses a problem for consteval-only types as it raises questions about how pointers to consteval-only types behave in propagation, e.g.:

```
struct pointer_wrapper {
  std::meta::info* value_ptr;
};
```

is **pointer_wrapper** a consteval-only type?

These issues are best described by P4101R0 and closely related to above problem with pointers to yet-to-be instantiated templates; however, the fundamental conclusion of P4101R0 is correct in that for this to work for C++26, **pointer_wrapper** *must* be consteval-only.

What is not examined in the aforementioned paper is the usage of incomplete types in C++; of which there are historically three (in C++26, four).

External Type

The first is to allow a type to be declared that's not visible in the translation unit, e.g.:

```
struct foo;
void act_on_foo(foo* x);
```

In this situation, the program is useful because some other translation unit will provide the definition of **foo** and **act_on_foo** and the linker will compose the two translation units into a coherent program.

This is not a problem for consteval-only types and their associated functions because the front end must be aware of them; they operate at a higher level.

You can of course have two translation units:

```
// translation unit - A
struct foo;
void act_on_foo(foo* x);

// translation unit - B
struct foo {
  std::meta::info value;
};
```

In translation unit A **foo** is not consteval-only, in translation unit B, **foo** is consteval-only. This is however, not a new problem, it is in the same family of errors as ODR violations.

Self Referentiation

The second use case is for self referentiation, e.g.:

```
struct meta_linked_list {
  meta_linked_list* next;
  std::meta::info   value;
};
```

In this case, consteval-only types can work just fine. The incomplete type member (being the class itself) does not factor into the conclusion of whether or not the type is consteval-only (only the other data members).

Cyclic Dependency

The third use case for incomplete types in C++ is when two types depend on each other. This case – similar to the union case – is highly unlikely to occur and has not manifested in real reflection programs:

```
struct a;
struct b;

struct a {
  b*              ref_to_b;
  std::meta::info value;
};

struct b {
  a*              ref_to_a;
};
```

Here a human can observe that **a** must be a consteval-only type because it contains a consteval-only type as a data member. Similarly, **b** must be consteval-only because it contains a reference to a which is consteval-only.

For C++26, this can be solved in the exact same way that uninstantiated types are solved using either the eager or delayed consteval-only type decision approach. In the former case this would be an error and in the latter case this would be valid code so long as the first variable using either **a** or **b** appears after the type definitions.

Notably P4101 notes that:

> *Note for interest that the Zig programming language has consteval-only types in the exact way that we want to define here (comptime-only in their parlance), but they don't have incomplete types in the same way, so they don't run into this issue.*

Define Aggregate

P2996 introduces **define_aggregate** which allows a forward declaration to be turned into a class definition programatically.

define_aggregate has many useful examples. However, it's worth noting that if we disallow a type to be forward declared and then later made consteval-only by its definition, **define_aggregate** has no means to declare a consteval-only type.

As **define_aggregate** is both a useful stopgap while we work on a more complete model for compile time code generation and primarily useful for generating traditional runtime (and optionally constexpr) code, the limitation that it cannot be used to make consteval-only types is actually fairly minimal. Of all of the examples presented in P2996, the only one that would be truly impacted is the tuple case. Of that case, the only impact is that you can't have a consteval-only tuple implemented via **define_aggregate**.

That said, if we take the delayed consteval-only type decision approach even this case of the consteval-only tuple could be implemented via **define_aggregate** in C++26.

Expressivity

The above incomplete type issues (cyclic dependencies and define aggregate) should not block P2996 and do not require any major late stage redesign of consteval-only types (or reflection). Like the union case, in the C++29 time frame, users that run into these cases can come back to the committee with real examples that explain the problem, why they've run into it, and what their ideal way to express their intent would be.

Fundamentally limitations to expressivity are what cause the problems with incomplete types.

Traditional forward declarations simply provide insufficient information to make an efficient and accurate judgement call on whether or not a type is consteval-only. This is largely an artifact of the design being rolled out incrementally rather than shipping a comprehensive consteval-only types model for C++26; users can only declare consteval-only types via propagation (from **std::meta::info**) and are not otherwise allowed to declare their own consteval-only types.

There is another similar problem that will almost certainly emerge with consteval-only types (formally or informally) in the future as constexpr usage increases. That is the need to sprinkle “constexpr” or “consteval” all over the program.

By increasing the expressivity of the class declaration, we can actually significantly reduce the verbosity. For instance, in C++26 we have to write things like the following:

```
struct my_compile_time_only_type {
  consteval my_compile_time_only_type();
  consteval int do_0();
  consteval int do_1();
  /* … “/
  consteval int do_n();
};
```

It is not hard to imagine that users would rather write:

```
struct consteval my_compile_time_only_type {
  my_compile_time_only_type();
  int do_0();
  int do_1();
  /* … “/
  int do_n();
};
```

The natural expansion of this syntax solves the forward declaration problem:

```
struct consteval a;
struct consteval b;
```

allowing the user to provide the compiler sufficient information to avoid any ambiguity in the cyclic dependency case (because it knows before it ever sees the definition the incomplete type is going to be consteval-only).

We can also then take a closer look at some of the other instances where we've forced users to write constexpr. By informing the type system of our intended semantics, we can actually revisit this example:

```
int main() {
  auto refl = ^^hello_world;
  std::cout << identifier_of(refl) << '\n';
  return 0;
}
```

While it is very much out of scope for C++26, it's not unreasonable to allow this in C++29. With consteval-only types we have sufficient context to know that the only valid interpretation of refl is that refl is a consteval-only variable (and thus "implicitly constexpr").

This is not necessarily something that we must do or will do. However, it demonstrates the value in formally acknowledging consteval-only types in the type system; if we so wish, we can use the extra information in the type system to stop repeating ourselves and write less verbose code that more seamlessly blends with runtime code.

User Defined Consteval-only Types

The current design goals are oriented around providing the fundamental machinery necessary for C++26 reflection to be successful. However, there is reason to believe that users will find consteval-only types useful outside of reflection.

consteval functions provide a means of performing an operation at compile time and ensuring that the algorithm does not escape to runtime (causing unnecessary binary bloat and potentially exposing a proprietary or otherwise secret algorithm that does not need to exist in the binary).

consteval functions do not however provide a means of ensuring similar guarantees for data; consteval-only types do. With a consteval-only type any variable of that type is consteval only. This means that large consteval-only data structures can be build up for the interpreter and guaranteed not to make it into the binary.

This also means that potentially sensitive information like a private key could be provided to the interpreter and used to encrypt or sign some data during compile time while having a strong guarantee (modulo implementation bugs) that the private key does not escape into your produced binary.

Compile Time Non-transient Allocation

Another long standing issue within the committee is that of non-transient allocation. The folks that would like to see non-transient allocation fall into two categories. Those that want

non-transient allocations for use at compile time (within the interpreter) and those that want non-transient allocations for use at runtime.

Consteval-only types provide a very clean and sound model for compile time only non-transient allocations as by definition all values with a given consteval-only type are consteval-only values. It is thus known without question that any constexpr variable with a consteval-only type will not have its value leak to runtime, and thus the entire issue of how to serialize the constexpr variable is avoided.

Metaprogramming

Another advantage of formally acknowledging consteval-only types is that C++26 reflection can tell us a type is consteval-only. This is a space that still needs more exploration; however, it seems reasonable to assume that as we begin to write metaprograms via templates (or future injection facilities) just as there are branches of logic that only make sense at runtime (or have runtime optimized implementations), there will be branches of logic that only make sense at compile time (or have compile time optimized dependencies).

It would be possible to recover this information by traversing the type and informally acknowledging that a given type is consteval-only. However, should this become necessary the traversal will certainly have more overhead than simply tracking this information in the compiler.

Interpreter Owned Types

Another interesting property of consteval-only types is that they do not need to meet the same requirements that C++ needs for traditional class types. They are fully owned by the front end, they have no linker impact, and thus they can potentially have novel behavior.

As an example, many scripting languages allow you to dynamically manipulate a type during evaluation, adding new data members and member functions. This is a powerful feature of scripting languages that is leveraged by REPLs interpreters for rapid prototyping, test mocking, and as a means of dynamically generating the type's API.

This also provides a carveout for the deployment of other magic built-in types. For instance if we wanted to have a **std::comp_map** that was only available at compile time and used a highly efficient hash map implementation hosted in the interpreter in native code (as opposed to being an interpreted map) this would become possible. The same is true of a **std::comp_vec** as a highly efficient compile time vector.

It is tempting to say that we could have these types without consteval-only types (perhaps as a category of types known as builtin interpreter types) and that only those types would be special. However, what we would quickly encounter is that all the problems that need to be solved for consteval-only types would be repeated for every builtin interpreter type. This would then result in a less cohesive model of compile time interpretation and compile time typing.

Runtime Non-transient Allocation

The struggle with runtime non-transient allocation is largely based around initialization; in other words, “how do you reconstruct the memory needed for your **std::vector<int>**?” Then there is a secondary design point of “is that runtime object then mutable or immutable?”

This has proven to be a hard problem to solve generally; meanwhile cases of common data structures like vectors and maps that are pain points. In a world where we’ve allowed C++ to have interpreter owned types, we can leverage that to solve some amount of the problem by defining a serialization from those types (e.g.) **std::comp_vec<int>** to a reasonable runtime equivalent.

Conceptually this could look like the following taking everything talked about previously in this paper and built on top of what’s shipping in C++26 into account:

```
std::comp_vec<int> vec;

consteval {
  /* populate the vector */
}

int              runtime_arr[] = vec;
std::vector<int> runtime_vec = vec;
```

In the case of **runtime_arr** this is ultimately little more than syntactic sugar facilitated by the type system to simplify creating a runtime array using the same methodology as **define_static_array**. In the latter case of **runtime_vec**, this syntactic sugar is further expanded to initialize a runtime vector; conceptually it’s a special case of aggregate initialization. This is of course not an example of profound elegance, it is not a general solution that works for any C++ data structure, it is however a very pragmatic solution that helps improve performance both at compile time and runtime while giving the user a very familiar syntax.

Module Non-transient Allocation

An additional type of non-transient allocation that has not been discussed is non-transient allocation between modules. As consteval-only types (and thus all values of that type) are owned by the compiler front end, conceptually non-transient allocations owned by values of consteval-only type can be serialized and deserialized across a module boundary. There will never be a runtime non-transient allocation aspect to consider for these values; which simplifies the problem.

Other Options

Consteval-only Types - “Consistency Check”

An earlier attempt to solve this problem used an algorithm that uses four states “Yes”, “No”, “Allowed”, and “Disallowed.” That algorithm would produce similar results to the delayed consteval-only types approach described above. However, the implementation of the algorithm does not establish decision points upon which a type either is or is not a consteval-only type.

That in turn greatly increases the complexity as it is effectively a consistency check over all uses of a type within the program. To elaborate, given an example like the following:

```
struct wrapper {
  pointee<int>* member;
};

wrapper k{nullptr};

template struct pointee<int>;
```

The algorithm prescribes that **wrapper** would be marked “allowed” at the point of its definition. The declaration of **k** would then convert wrapper from “allowed” to “disallowed.” A subsequent instantiation of **pointee<int>** would then require the implementation to detect that **wrapper::member** is now complete, causing **wrapper** to be necessarily consteval-only and thus in conflict with the declaration of **k**.

The most efficient implementation given a situation like the above would be to have a list of types to recheck upon instantiation stored with the specialization **pointee<int>**. This anticipated implementation would then result in considerable overhead even for code bases that make no usage of compile time types since *all substituted but uninstantiated types* must maintain a list of the types referencing them. Subsequently upon instantiation that list would have to be traversed to check for unpermitted pairings.

Consteval-only Values

P4101R0 prescribes consteval-only values as a solution that is superior to consteval-only types. P3603 expands upon the functionality proposed to allow for various forms of escalation from a constexpr variable to a consteval (immediate) variable.

These two ideas while being presented as mutually exclusive, are in fact complementary. If the committee at some point decides that consteval-only values (and the semantics associated with

them) are worth pursuing, consteval-only types merely strengthen the consteval-only values proposal by providing a foundation to let users define types that leverage this notion.

Non-transient Allocation Model

Without consteval-only types P3603's non-transient allocation model does not actually work generically. To explain this, consider the following:

```
template<typename T>
struct holder {
  constexpr static T value = T{};
};

struct my_type {
  int *x = new int(10);
};

holder<my_type>::value; // error, non-transient allocation
```

In this case, there is no typing information provided that would allow for **holder::value** to be promoted to a consteval (immediate) variable. That means there is necessarily a hole in the promotion rules. If **my_type** is allowed to express its intent to be used at compile time only (i.e., to be a consteval-only type) this problem disappears as the **holder::value** is promoted to a consteval-only variable by the type system.

If the model was changed so that any constexpr variable could be promoted to a consteval (immediate) variable, this could work without constexpr types. However, this would pose a problem for any potential future models of non-transient allocation that would prefer that constexpr variables have non-transient allocations that are available at runtime.

Immediate Escalation

Removing consteval-only types could have significant unforeseen impacts on immediate escalation that are not well understood. If we consider the following example:

```
template<typename T>
constexpr void a(T) {
  if consteval {
    comp_time_print("hello world");
  }
}
```

```
template<typename T>
constexpr void b(T t) {
  a(t);
}

b(^^int);
```

in the working draft, the instantiation of b with a consteval-only parameter type forces the instantiated **b(^^int)** to be an immediate (consteval) function. That immediate invocation semantic guarantees that **b** is evaluated at compile time and that **b** is not a runtime invocation.

It's conceivable to me that consteval-only values do cover these cases. However, with the considerably lower amount of attention they've received there could be unforeseen holes here.

We Already Have Them

There's been a sort of discussion point that we already have consteval-only values. This is true in a form and they're not great.

Back in the Lock3 days we implemented an innocent pattern where we wanted to specialize a type with a consteval function to implement our iterator type.

```
template<bool (*F)(bool)>
class iterator {};

consteval bool next_impl(bool);

consteval {
    iterator<next_impl> my_iter;
    /* use the iterator */
}
```

The type system has no way of expressing a consteval function pointer. So there is currently no way to specify that you're trying to specialize with a consteval-only function other than to change all of your consteval functions to consteval functors – which is exactly what we did.

The canonical reading is that it should not work because you cannot create a function pointer for a consteval function. Regardless, there is implementation divergence on whether this works.

This is not itself an impeachment of consteval-only values. However, it demonstrates that we can create new surprising problems that have to be fixed in multiple places within the language – particularly for generic code – when we do not pass along sufficient information in our type

system for templates.

Furthermore, reliance on values introduces its own complexity such as CWG2734 “Immediate forward-declared function templates.” It is conceivable that consteval-only values would cause similar issues.

Impact Summary

The major points to consider are that:

- consteval-only values have not been as thoroughly explored.
- consteval-only values are a piece of a solution but not a complete solution to compile time only values.
- consteval-only values do not allow for future improvements in expressitivity of types that are intended to be used only at compile time.
- consteval-only values do not provide a solution for user declared consteval-only types. The only way to get a consteval-only value is for the value to include a reflection or to declare a consteval variable.
- We would not only have to make a late stage design change, but this late stage design change would impacting the “Immediate Escalation” rules described above.

Conservative Consteval-only Types

The consteval-only types component of C++26 is already quite conservative. However, if we wanted to go further and ensure that consteval-only types can be removed wholesale from the language (should the committee come back with concrete evidence that they are truly a mistake) we can remove the rule described in “Immediate Escalation” and “Consteval Virtual Functions” leaving the **what()** function as a constexpr function override (instead of a consteval function override).

As noted above, removing these rules at this point would likely increase our risk of negatively impacting the user experience in a meaningful way as they would undermine any immediate escalation behavior that is coming from the type system, instead relying almost entirely on promotion resulting from a constant expression failing to evaluate.

We could remove the rules specifically related to the function type (or require that a consteval-only type appear in the function parameter types or return type) while leaving escalation based on consteval-only type expressions and variables in the function body. However, it would be very tricky to remove consteval-only types while preserving the escalation rules we shipped in C++26 if we do this.

In general, I do not believe we are solving real problems if we remove these escalation mechanism or take a more conservative approach with consteval-only types at this point.

Leak to Runtime

One possibility that's been proposed is removing consteval-only types from the working draft, not adding anything else, and allowing all values that are currently (status-quo) considered consteval-only to leak to runtime.

If we leak to runtime we have neither the triggers from the type system with consteval-only types nor the triggers from consteval-only values to feed the immediate escalation rules. This means that leaking to runtime almost certainly requires us to remove immediate escalation and this could have significant impact on the feature's applicability to generic code.

We would also then be unable to detect any suspicious **std::meta::info** uses to help catch typos, wasted runtime storage space, or guide users towards the code they're trying to write with diagnostics.

Diagnostics Fix via QOI

Particularly regarding diagnostic quality in general, we would almost certainly be in a worse place.

It's theoretically possible to fix the diagnostic issue described above as QOI (at least to some extent). However, as a general rule, you cannot in compiler diagnostics assume intent. With an informal notion of consteval-only types and the variable thus being fundamentally correct, it is at best a very awkward job (in every compiler diagnostic system – that I've seen) to retroactively mark the variable declaration problematic and then apply that to suppress further diagnostics. It is at worst, completely incorrect because of the ambiguity of intent. It is much easier if the language simply tells us the variable declaration is invalid.

Alternatively, we just live with some additional unnecessary diagnostics, but that seems like an at best unfortunate outcome.

Conclusion

Consteval-only types are a strong start towards expanding the language's understanding of compile-time only values incrementally (in a way that allows the committee and community to gain usage experience and thus make more informed decisions).

To address CWG3150 " Incomplete consteval-only class types", we should adopt the delayed consteval-only type decision approach. For both uninstantiated and incomplete types this completely solves the issue.

Overall, with this approach adopted, the issues raised have no anticipated impact on users of C++26 reflection and the suggested approach has trivial implementation implications. C++29

can then build on this strong foundation to address any defects with improvements to expressivity based on real use cases and user experiences.