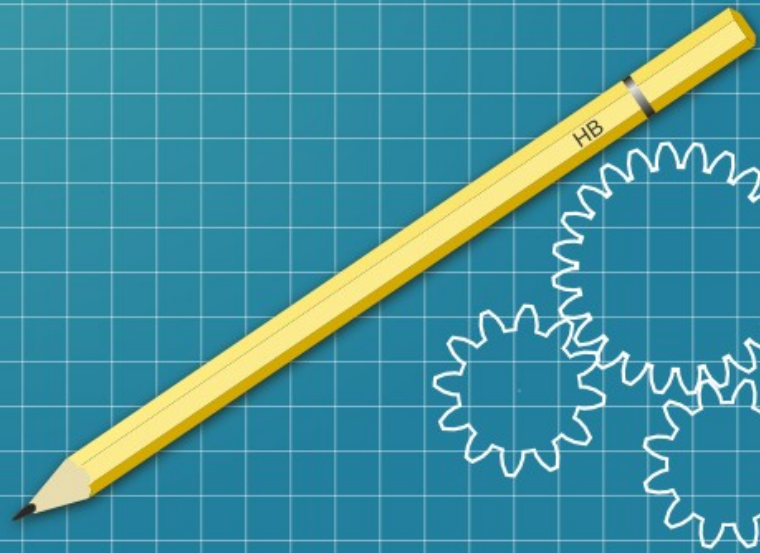
Consteval-only Types

Wyatt Childers



Who are you?

- Minecraft Modder (Java Runtime Reflection)
- Student of Andrew Sutton
- Former Ruby on Rails Software Engineer
- Lock3 Metaprogramming Compiler Engineer
- Edison Design Group
  - Compiler Front End Engineer
  - Tooling Engineer (Python, Codegen)
  - Operations (Servers, Networking, etc)

Why bother?

- Are you just wasting everyone’s time?
- Discouraging core poll
  - Consetval-only values are easier to specify
  - Consteval-only types just make consteval-only values
  - No runtime reflections leaking
- Alternative didn’t require me to implement anything
- Alternative didn’t require conflict with dear friends
- I could’ve enjoyed my weekend!

Why bother?

- I DIDN’T WANT TO

What if I don’t at least try?

- Can I live with the outcome?
- I’m “the guy”, who else is going to do it?
  - Unique mix of reflection + metaprogramming + compilers + dynamically typed languages + static typed languages
- WG21 should make an informed choice

Historical Context

How It Started

- Working on experimental reflection for C++!!
- Hey friend, check this out!!!

Const-wha?

```
struct hello_world;

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

Const-wha?

```
struct hello_world;

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

Const-wha?

```
<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;
     |        ^~~~
```

I Love Making Simple Work

- Asked to restrict users to one login per account
  - Terminate current session upon login to another device
- Seems like that would be really annoying switching between Phone and Computer
- Counter offer: What if we use WebSockets to limit concurrent connections but not sessions?
  - Simpler to implement? NO!
  - Simpler to describe? NOOO!
  - Simpler for the user? YES!!!

Properties of Novel Simplicity

- The right thing is easy
- The wrong thing is hard
- It’s general / flexible
- It helps in ways you didn’t expect
- Sacrifices made in context

False Signals

- Everything is a rainbow
- Fewer lines of code
- “It sounds simple enough”

Wrong is Wrong

```
struct hello_world;

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

Wrong is Wrong

```
struct hello_world;

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

Propagation of Wrong

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

Propagation of Wrong

```
int main() {
  // this doesn’t make sense either
  rendered_type refl;
  return 0;
}
```

What do we have?

- If I write code that doesn’t make sense; it doesn’t compile 100% of the time!
- I can never waste runtime storage; truly 0 cost!
- Good compiler diagnostics are easy!

What do we have?

```
struct hello_world;

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

What do we have?

```
"<source>", line 7: error: variable with
consteval-only type "std::meta::info" declared
outside of immediate context
                add constexpr to the variable
                                        declaration
    auto refl = ^^hello_world;
                ^
```

What do we have?

```
struct hello_world;

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

What do we have?

```
struct hello_world;

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

Thanks, I Hate It

- When I first showed this to Andrew Sutton, he was not a fan.
- Things like “unions” and “variant” did come up.
- This is the guttural reaction for a C++ programmer, but it is incorrect.
- Andrew eventually came around; especially after implementing a runtime reflection library on P2996-style reflection.
- I’m dismayed we’re having that same conversation now.

What can’t it do?

```
auto v = std::variant<std::meta::info,
                     int>(42);


struct C { std::meta::info const* p; };
auto c = C{.p = nullptr};
```

Code Review

```
struct plausible_thingy {
  union {
    long  x;
    void* y;
  } variant;
};
```

Change it Later?

```
auto v = std::variant<std::meta::info,
                     int>(42);

// error, add constexpr
```

Metaprogramming

```
template<typename T>
struct foo {
  consteval {
    if (is_consteval_only_type(^^T)) {
      → fragment struct { /* ... */ };
    } else {
      → fragment struct { /* ... */ };
    }
  }
};
```

New Information

Promises Made

- We can fix variant visitation with consteval-only values.
- We can use consteval functions as constant template arguments.
- We can probably get to a good non-transient allocation model with consteval-only values and variables.

Code is Code

```
dispatch_map = {}
for name, handler in find_handlers():
  dispatch_map[name] = handler

def rate_pies(next_rating):
  best_pies = {}
  for name, rating in next_rating():
    best_pies[name] = rating
```

C++ Today

```
namespace epic_dispatch {
  using map = epic_internal_map<str, fn>;
}

constexpr map dispatch_map = build_it();
```

C++ Consteval

```
consteval epic_dispatch::map dispatch_map;

consteval {
  for (/* */) {
    dispatch_map[key] = value;
  }
}
```

C++ Consteval

```
epic_dispatch::map dispatch_map;

consteval {
  for (/* */) {
    dispatch_map[key] = value;
  }
}
```

Consteval Expressivity

```
template<typename K, typename V>
struct epic_internal_map {
  consteval size_t hash_code(K);
  consteval V*     get_or_null(K);
  // ...
  consteval V*     remove(K);
};
```

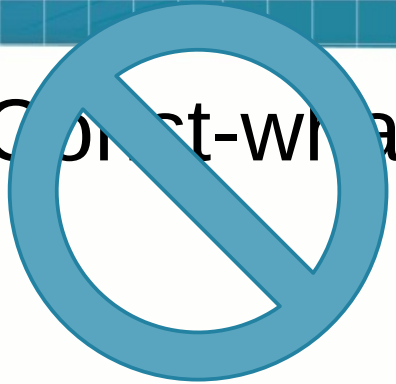
Consteval Expressivity

```
template<typename K, typename V>
struct consteval epic_internal_map {
            size_t hash_code(K);
            V*     get_or_null(K);
  // ...
            V*     remove(K);
};
```

User Consteval Types

```
struct consteval comp_crypto {
  using str_type = const char*;
  comp_crypto(str_type private_key);
  template<typename T>
  str_type generate_signature(T t);
  template<typename T>
  str_type encrypt(T t);
  template<typename T>
  str_type decrypt(T t);
};
```

C++ct-wh a?



```
struct hello_world;

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

Non-transient Allocation

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

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

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

Non-transient Allocation

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

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

// always runtime
holder<my_type>::value;
```

Non-transient Allocation

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

// it just works
// it’s general
// it’s unambiguously compile time
holder<epic_internal_map>::value;
```

Avoiding Past Mistakes

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

consteval bool next_impl(bool);

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

Avoiding Past Mistakes

```
using fn_type = bool(bool);

template<typename T>
struct callee {
  constexpr auto operator()() { return (*this->x)(true); }
  fn_type *x;
};

          bool next_impl(bool);

int main() {
  auto x = callee<fn_type>{&next_impl};
  x();
}
```

Avoiding Past Mistakes

```
using fn_type = bool(bool);

template<typename T>
struct callee {
  constexpr auto operator()() { return (*this->x)(true); }
  fn_type *x;
};

consteval bool next_impl(bool);

int main() {
  auto x = callee<fn_type>{&next_impl};
  x();
}
```

Magic Compiler Types

- When you’re in Python, JavaScript, etc lists, maps, and tuples are all intrinsic to your interpreter …
- Why? It’s nice syntax for one.
- … but also? It’s typically faster.
- Anyone that tries to implement their own “vector” in Python is going to quickly learn they’re going to lose against **[]**

Magic Compiler Types

- Maybe we want these for C++?
  - std::comp_map<K, V>
  - std::comp_vec<T>
- Isn’t it easier if we can just say “they’re consteval-only types”?

Runtime Non-transient Allocation

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

consteval {
  /* populate the vector */
}

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

Why Are We Here?

Incomplete Types

- They hadn’t come up … in *almost* 10 years!
- (Per Barry) Zig doesn’t have them
- Most “metaprogramming” languages are scripting languages … so they don’t
- For compile time metaprogramming, it’s an edge case; oops!!

Incomplete Types

```
struct A { info a = ^^int; };

struct B { A *p; constexpr B (A *x) : p (x) {} };
```

Incomplete Types

```
struct A;

consteval {
  define_aggregate(^^A, /*...*/);
}

struct B { A *p; constexpr B (A *x) : p (x) {} };
```

Incomplete Types

```
struct A;

struct B { A *p; constexpr B (A *x) : p (x) {} };

struct A { info a = ^^int; };
```

Incomplete Types

- We fixed it
  - https://godbolt.org/z/YvbofjfPc
  - https://godbolt.org/z/sWKdfj7P8
- No code breaks
- Core believes it’s implementable
- **I implemented it over the last two days from scratch**
- **It passes over 120,000 EDG QA tests (even when forced on in older language modes)**
  - It crashes in some reflection cases … because our implementation is incomplete

Incomplete Types – Original Fix

```
struct A;

struct B { A *p; constexpr B (A *x) : p (x) {} };

// error A is not a consteval-only type
struct A { info a = ^^int; };
```

Incomplete Types – Original Fix

```
struct consteval A;

struct B { A *p; constexpr B (A *x) : p (x) {} };

struct A { info a = ^^int; };
```

Incomplete Types – C++26 Fix

```
struct A;

struct B { A *p; constexpr B (A *x) : p (x) {} };

B b; // A is no longer permitted
     // to be consteval-only

struct A { info a = ^^int; };
```

Incomplete Types – C++26 Fix

```
"<source>", line 9: error: previous use as an
incomplete class type prevents "A" from becoming
a consteval-only type
            type assumed not consteval-only at
line 7
  struct A { info a = ^^int; };
         ^
```

Incomplete Types – C++26 Fix

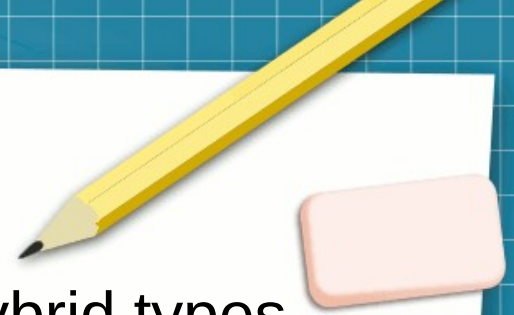
```
struct A;

struct B { A *p; constexpr B (A *x) : p (x) {} };

struct A { info a = ^^int; };

B b; // error, B is a consteval-only type
     // add constexpr to the variable
```

Procedural

- This is **HIGHLY** unusual
- Consteval-only values are NOT well understood
- Consteval-only values have been rejected for C++26 by EWG previously
- Consteval-only types have been around for **YEARS** with consensus
- The core issue has been addressed without changing the feature
- Yet we’re still considering changing the feature???
- On the last week???

Types vs (Only) Values

- Give us better diagnostics
- Catch more real user errors
- Can improve the expressivity of the language
- Can impower library authors
- Work well with templates
- Work well for non-transient allocation
- Work well as a model for interpreter builtin types
- Have other potential use cases outside of reflection
- Are forward compatible with the consteval-only value fixes towards variant visitation and template parameter objects
- Are implementable **and in the working draft!**



- Can declare hybrid types that contain useless values at runtime (just in case)
  - If this is really important we *almost certainly* will be able to fix it in C++29, even with consteval-only types
- Are a bit easier to specify and implement