Reflection Is Not Contemplation

Andrei Alexandrescu | CppCon 2024

“

Ba-dum tsss...

”

I’ll Start With the Punchline

- Static reflection without code generation is incomplete
- The “reading” part of reflection generally agreed upon
- The “generation” part of reflection suffered of neglect
  - P2996 very gingerly sneaks in a foot in the door (`define_class`)
  - P3294 finally blows the door off its hinges
- The two facets of reflection are equally important
- Where do AI tools fit within this craze?

The Reflection Circularity Problem

Without generation, we're chasing our tails

- Recall `^^x` reflects on `x`, `[:y:]` unreflects (splices) `y`; `[:^^x:]` is `x`
- Large consensus on introspection query: contemplation is great
- Fear and loathing about code generation
  - Expansion of existing introspection objects deemed acceptable
- Consequence:
  - Severely limited: can't do 3D with 2D abilities
  - No consensus on "how much" generation is just enough
  - No clarity on necessary primitives

Where Do We Want To Be?

Reflection is ultimate code reuse

- Procedural: *I can write functions then call them*
- OOP: *I can call functions that haven't been written yet*
- Templates: *I can design with types that haven't been written yet*

- Reflection: *I can customize types that will not have been meant for customization*

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

How to Generate a Property

```
struct Book {
  consteval {
    property("author", ^^std::string);
    property("title", ^^std::string);
  }
};
```

- A property introduces a data member, a getter, and a setter
- We’d like to be able to define such metafunctions

How to Generate a Property

```
// Equivalent hand-written code
struct Book {
  std::string m_author;
  const std::string& get_author() const { return m_author; }
  void set_author(std::string s) { m_author = std::move(s); }

  std::string m_title;
  const std::string& get_title() const { return m_title; }
  void set_title(std::string s) { m_title = std::move(s); }
};
```

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42;typename[:^char:]  = '*' // Same as: char c = '*';
```

Early Observation

- We create identifiers from strings
  - Plus ability to concatenate “`m_`”, “`get_`”, “`set_`” with parameter
- Identifiers may be:
  - introduced (`get_title`); or
  - part of generated code (`return m_title;`)

- Ability to fluidly translate strings to identifiers *crucial*

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

Comparison of Splicing Models

- **The Spec API:** function calls create a “spec” of a type that is spliced
  - Complex; problematic; death of a thousand cuts
- **The CodeReckons API:** OOP interface for AST building
  - Verbose, loquacious, garrulous, prolix, long-winded, circumlocutory, and unceasingly inclined toward linguistic superfluity.
- **String Injection:** offer a primitive that splices CT strings into code
  - Horribly unstructured. Also… can’t use macros?!?
- **Fragments:** C++ fragments of stylized code
  - Thorough early checking makes complexity explode

> “There’s a representation of C++ code
> that is well-defined, complete, and easy
> to understand: C++ code.”
> – Daveed Vandevoorde

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

The Game Changeth: Token Sequences

P3294

- At this point complexity is a huge liability
- Adding yet another sublanguage to C++ deemed undesirable
- String-based generation best; *let's eliminate its disadvantages*

- Strings opaque/unwieldy? Use *token sequences* instead of strings
  - Cost: one added literal kind
- Injection risks and dangers? Restrict string expansion
  - Carefully controlled escapes, interpolation-style

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42;typename[:^char:]  = '*' // Same as: char c = '*';
```

Token Sequences

Status: prototype implementation

```
constexpr auto t1 = ^^{ a + /* hi! */ b }; // three tokens
static_assert(std::is_same_v<decltype(t1), >);
constexpr auto t2 = ^^{ a += ( };
constexpr auto t3 = ^^{ abc { def }; // Error, unpaired brace
```

- Concatenation, comparison, comments etc are token-level, not textual
    - Eliminates string and preprocessor dirt
- Macros work

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

Quoting & Splicing a Token Sequence

Status: prototype implementation

- Escapes inside a token sequence:
  - `\( expr )` evaluates expr eagerly during creation of the token sequence
  - `\id( e1, e2, ... )` concatenates strings and integrals, creates an identifier
  - `\tokens( expr )` expands another token sequence
- Inside any `consteval` function:
  - `queue_injection(` *`tokens_expr`* `)` injects a token sequence into the *current declaration context*

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

Quoting in a Token Sequence

Status: prototype implementation

```
consteval void property(std::meta::info type, std::string_view name) {
  queue_injection(^^{
    [:\(type):] \id("m_", name);
    [:\(type):] const& \id("get_", name)() const {
      return \id("m_", name);
    }
    void \id("set_", name)([:\(type):] x) {
      \id("m_", name) = std::move(x);
    }
  });
}
```

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

For Comparison: `property` Using The Spec API

```
template <class T> using getter_type = auto() const -> T const&;
template <class T> using setter_type = auto(T const&) -> void;

consteval auto property(string_view name, meta::info type) -> void {

auto member = inject(data_member_spec{
 .name=std::format("m_{}", name), .type=type
});

inject(function_member_spec{
 .name=std::format("get_{}", name),
 .signature=substitute(^getter_type, {^type}),
 .body=defer(member, ^[]<std::meta::info M>(auto const& self) -> auto const& {
   return self.[:M:];
   })
 });

inject(function_member_spec{
 .name=std::format("set_{}", name),
 .signature=substitute(^setter_type, {^type}),
 .body=defer(member, ^[]<std::meta::info M>(auto& self, typename [:type_of(M):] const& x) -> void {
   self.[:M:] = x;
   })
 });

}
```

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

For Comparison: `property` Using The CodeReckons API

```
consteval auto property(class_builder& b, type type, std::string name) -> void {
auto member_name = identifier{("m_" + name).c_str()};
append_field(b, member_name, type);

method_prototype mp;
object_type(mp, make_const(decl_of(b)));
return_type(mp, make_lvalue_reference(make_const(type)));
append_method(b, identifier{("get_" + name).c_str()}, mp,
[member_name](method_builder& b){
  append_return(b,
    make_field_expr(
      make_deref_expr(make_this_expr(b)),
      member_name));
    });

method_prototype mp1;
append_parameter(mp1, "x", make_lvalue_reference(make_const(type)));
object_type(mp1, decl_of(b));
return_type(mp1, ^void);
append_method(b, identifier{("set_" + name).c_str()}, mp1,
  [member_name](method_builder& b){
    append_expr(b,
      make_operator_expr(
        operator_kind::assign,
        make_field_expr(make_deref_expr(make_this_expr(b)), member_name),
        make_decl_ref_expr(parameters(decl_of(b))[1])
        ));

    });

}
```

> “You can observe a lot by just implementing **`identity`**”
> – Yogi Berra

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42;typename[:^char:]  = '*' // Same as: char c = '*';
```

Notion of Identity/Copying/Cloning is Crucial

Allow me a little soapbox

- Identity fundamental concept in Computer Science
  - Different languages define it differently
- Philosophical underpinnings
  - Are objects fungible?
  - What is “same” object vs. “a copy”?
  - See Ship of Theseus Paradox/Hobbes variation
- Identifying, aliasing, copying, and moving objects central topic in C++
  - Alias analysis, self-assignment, self-move, double deletion, …
- “Identity Function:” can insert in any (sub)expression without changing meaning

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

Identity Function Example

Allow me a little soapbox

FROM
ACCU
2008

- We had to add && because C++98 couldn’t implement identity
- The year was 2008 and this was C++0x:

```
template <typename T>
T&& identity(T&& x) { return x; }
```

- Done! Or are we?

NVIDIA.

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

Identity Function Example

```
const int& a = identity(42); // dangling
int&& b = identity(42);      // dangling
auto&& c = identity(42);     // dangling
```

- When input is rvalue or rvalue reference, output should be a “true” rvalue
- When input is a reference or lvalue, output should be (the same) reference

- Early feedback:
  - *“I don’t understand his obsession with identity”*
  - *“Identity has no use in the real world”*
  - *“What’s the applicability here?”*

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42;typename[:^char:]  = '*' // Same as: char c = '*';
```

A Two-Body Problem

```
template <class T>
T& identity(T& x) { return x; }

template <class T>
T identity(T&& x) { return std::move(x); }
```

- All rvalue (reference)s go through the second overload, others through the first
- Key problem: scalability
  - Two definitions
  - Scales poorly to multiple arguments (T mentioned in the returned type)

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

One-Body `identity()`

```
template <class T>
T identity(T&& x) {
  return T(std::forward<T>(x));
}
```

- `T` morphs into either a value or reference (as before)
- `std::forward<T>(x)` makes sure a move happens if needed
- The additional `T()` is either a move ctor call OR a no-op, as needed
- Scalability problem remains
  - `T` still mentioned in the returned type
  - We can’t return arbitrary expressions with fidelity

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

C++14 Fixes “Last” Problem: `decltype(auto)`

```
template <class T>
decltype(auto) identity(T&& x) {
  return T(std::forward<T>(x));
}
```

- `decltype(auto)` *teleports either a value type or reference type out*
- Turns out it was important – see Howard Hinnant’s N2199
  - Rids `std::m(in|ax)` of dangling references with C++11 tech
  - Valiant effort: 210 lines, 21 struct definitions (7 distinct names)
  - Plagued by… `identity()` not implementable scalably
  - Rejected as too complex for what it does
- `min()` with `decltype(auto)` to N2199 spec in 8 lines, no helpers
  - `godbolt.org/z/bGPnjM76e`

std::min
SHERIFF
C++ Safety
imgflip.com

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

Just For Kicks

```
template <class A, class B>
decltype(auto)
min(A&& a, B&& b) {
  static_assert(... no dangerous comparisons ...);
  return b < a ? B(std::forward<B>(b)) : A(std::forward<A>(a));
}
```

- See also: P1179, different take (also 12 years later)

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

`identity` Appears in the Darndest Places

Not Only for Functions!

- “The functionality [the identity metafunction] provides is both fundamental and surprisingly useful.” — Timur Doumler, P0887

- `std::identity` (C++20)
  - Default projection in constrained algorithms
  - (Intentionally does not create copies so it’s different from the one above)
- `thrust::identity`
  - Templated, works on CPU and GPU, does not accept rvalues
- `std::type_identity`, `std::type_identity_t` (C++20)
  - Introduce non-deduced contexts in template argument deduction

So What Is the Identity of Reflection?

Assembling a Type From Parts of Another is the Core Business of Reflection

IKEA Job Interview
Please have a seat
CANARY PETE

> “Identity of reflection is being able to deconstruct an entity and construct a new, identical one piecewise.”

> Please note: that entails (a) complete deconstruction of the entity; (b) construction from scratch of the entity.

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

Once You Have Identity of Reflection…

Everything becomes easy!

- Herb keynote’s `interface` example?
  - “Deconstruct type, clone inserting `virtual` and `=0` for each memfun”
  - “Complain if you find data members or other suspect items”
- `polymorphic_base`:
  - “Deconstruct type, reassemble making sure no copying allowed”
  - “Ensure the dtor is public/virtual or protected/nonvirtual”
- `ordered`:
  - “Deconstruct type, reassemble and add `operator<=>`”

- *The point is (re)assembly from small, replaceable pieces*

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42; typename[:^char:]  = '*' // Same as: char c = '*';
```

The Real Challenge

- Cloning a class template is *much more difficult* than cloning a class
- `clone<MyStruct>` easy, `clone<std::vector>` absolutely crazy
  - Reflection must preserve order of declaration
  - Introspect function templates (e.g. `emplace_back`)
    - Signatures have constraints, `noexcept` clauses, attributes…
  - Clone inner classes: `iterator`, `const_iterator`
  - Manipulate (template) function signatures, e.g. `erase()`
  - Probably a million things we haven’t thought of
- We can’t avoid it! `clone<std::vector<int>>` equally hard

- Important part: *We know where the destination is*

```
constexpr auto  = ^int typename[: :]  = 42 // Same as: int x = 42;typename[:^char:]  = '*' // Same as: char c = '*';
```

Angle of Attack

Tokens, tokens everywhere...

- P3294 (Injection with Token Sequences) a game changer for generation
  - Lennon/McCartney of Daveed Vandevoorde, Barry Revzin, and yours truly
  - Flexible, loosely-structured “atoms” for code generation
  - Perfect currency for querying and writing code
- Repurposing P3157 (Generative Extensions for Reflection)
  - Retrieve/set template signature via multiple token strings:
    - `info get_template parameters(info);`
    - `info get_template_constraints(info);`
    - `info get_parameters(info);`
    - `info get_constraints(info);`
    - `info get_attributes(info);`
    - `info get_body(info); // (!)`

> Suddenly, an AI

We Can’t Not Talk About This

- How is generative AI changing language (features) design?
- `jippity.pro`: 350 lines of working code written by a 9 yo in 3h
- Entire project passed to an LLM alongside a change request

- Will we really need in the future:
  - Boilerplate generation? AI could do that
  - Read all that boilerplate? AI editors can fold that away
  - Refactor/change large swaths of code? AI could read/modify/write code

> Query (o1): *Please write a functionally equivalent clone of std::vector that uses a private member of type std::vector and forwards all methods to it. It should count calls to each method separately.*

> “(After 35 seconds) Generated a fragment, gave me homework.”

“

Me: *I noticed you wrote this:*
// Additional methods forwarded similarly...
Can you please generate the FULL code?
This is very important for my career.

”

“

(9 seconds later) Generated 447 lines of working code.

”

“

Me: *Can you please do the same for* `std::unordered_map`*? Otherwise, the Taliban will have a word with me.*

”

> “(6 seconds later) Generated 470 lines of working code. No uncomfortable questions asked.”

“

Don’t forget: today’s AI coding tools are absolutely the dumbest, most laughable, least capable they’ll ever be. Everything now is at best alpha quality.

”

Where Is this Going?

- We need to assess how AI will influence programming language design
  - We now assume human production and consumption; that is being challenged
- We need to figure:
  - What is the spec? Is it the code? Is it some initial code and a set of queries?
  - Will we look at source code, stylized summaries, or something else?
    - “This is a proxy, this is a memoized fetcher, this is a façade…”
- How do we define technical debt?
  - Today: entropy created in order to deliver something quickly
  - Tomorrow: entropy *that is difficult for humans+AI to reduce*

However

- *Current LLM technology akin to an extra engineer, not a language feature*
  - 100 lines/s way faster than human, but 100x slower than reflection
  - Not as reliable as vetted code that generates code
- Reflection >10x faster than TMP
  - 100x slower than compiled code
  - Close to parity with JIT or DLL tech
- Caching of reflection output will be a major focus
  - Dramatic improvement of compilation times

Don’t Forget

- Procedural: *I can write functions then call them*
- OOP: *I can call functions that haven’t been written yet*
- Templates: *I can design with types that haven’t been written yet*

- Reflection: *I can customize types that will not have been meant for customization!*

Thank You!