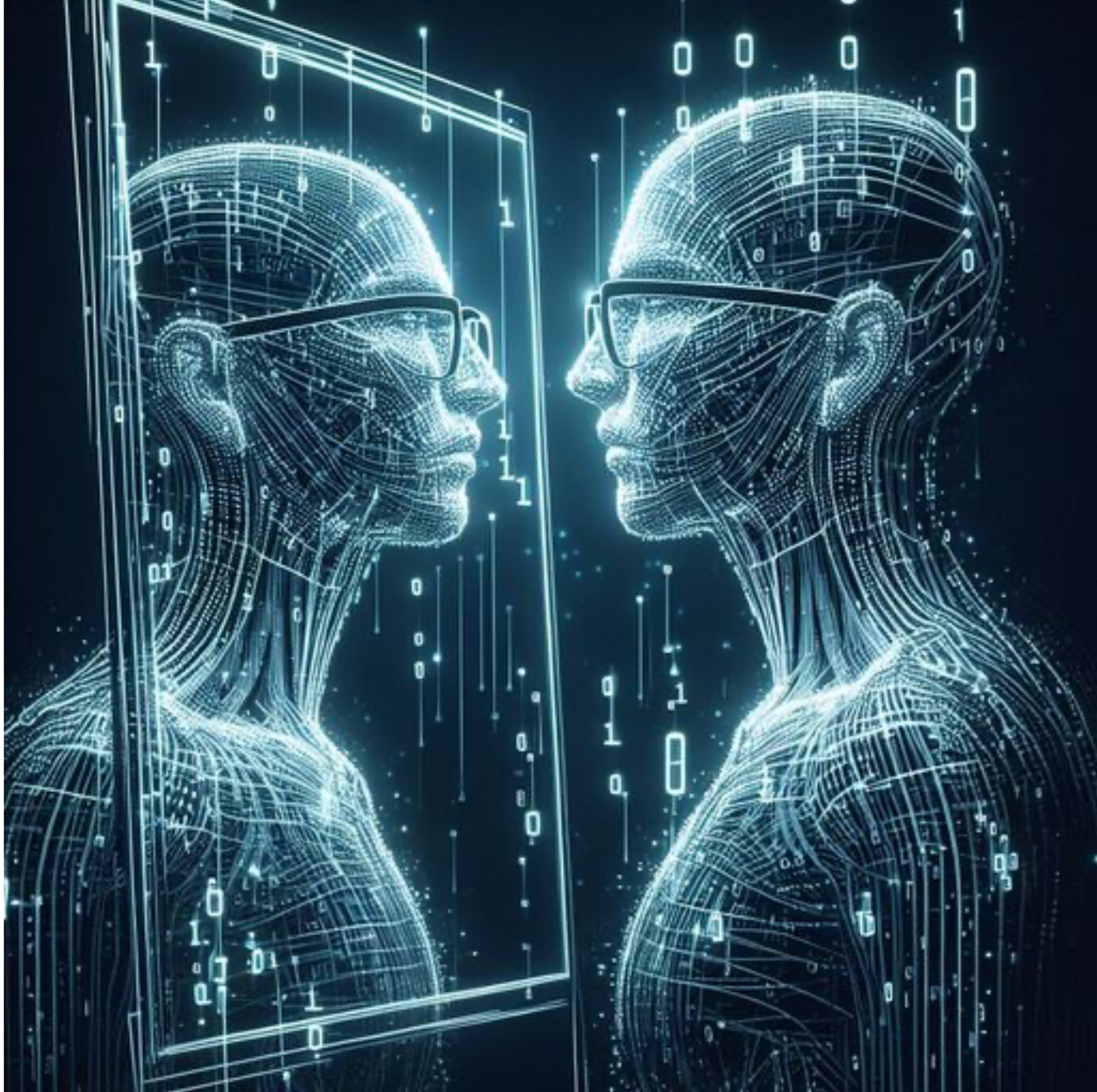
Gazing Beyond Reflection for C++26

Daveed Vandevoorde

Edison Design Group



P2996: Reflection for C++26

Wyatt Childers

Barry Revzin

Peter Dimov

Andrew Sutton

Dan Katz

Faisal Vali

P1240: Scalable Reflection in C++

Wyatt Childers **Andrew** Sutton **Faisal** Vali

P2996: In a slide...

Source construct
(syntactic domain)

`std::meta::info` *Value*
(computational domain)

Source construct
(syntactic domain)

```
^^std::vector<int>
^^int(*)()
^^std::cout
^^std::vector
^^std
^^::
```

```
#include <meta>
auto r = ^^Vec;
... is_template(r) ...
r = substitute(r, {^^int});
r = doTheThing(members_of(r));
Mega<^^Vec> device;
```

```
[:^^int:] x = 42;
typename[:r:] y;
template[:q:]<int> z;
[:r:]::X x = [:v:];
```

```cpp
#include <experimental/meta>
#include <iostream>

struct Entry {
  int key:24;
  int flags:8;
  double value;
} e = { 100, 0x0d, 42.0 };

int main() {
  constexpr std::meta::info  r = ^^Entry;
  std::cout << identifier_of(r) << '\n';
  constexpr std::meta::info  dm = nonstatic_data_members_of(r)[2];
  std::cout << identifier_of(dm) << '\n';
  std::cout << e.[:dm:] << '\n';
}
```

```cpp
struct I;
int main() {
  constexpr info r =
    define_class(^^I, {
      data_member_spec(^^int, {.name="index"}),
      data_member_spec(^^bool, {.name="flag", .width=1})
    });
  I x = { .index = 42, .flag = true };
  static_assert(std::is_same_v<[:r:], I>);
}
```

```
template<typename... Ts> struct Tuple {
  struct storage;
  [:
    define_class(^^storage,
                 {data_member_spec(^^Ts, {.no_unique_address=true})...});
  :] data;

  ...
};
```

```cpp
template<typename... Ts> struct Tuple {
  struct storage;
  [:
    define_class(^^storage,
                 {data_member_spec(^^Ts, {.no_unique_address=true})...});
  :] data;
  Tuple(): data{} {}
  Tuple(Ts const& ...vs): data{ vs... } {}

  ...
};


template<std::size_t I, typename... Ts>
  struct std::tuple_element<I, Tuple<Ts...>> {
    using type = [: std::array{^^Ts...}[I] :];
  };
```

```
template<typename... Ts> struct Tuple {
  struct storage;
  [:
    define_class(^^storage,
                 {data_member_spec(^^Ts, {.no_unique_address=true})...});
  :] data;
  Tuple(): data{} {}
  Tuple(Ts const& ...vs): data{ vs... } {}
  static consteval std::meta::info nth_nsdm(std::size_t n) {
    return nonstatic_data_members_of(^^storage)[n];
  }
};

template<std::size_t I, typename... Ts>
  constexpr auto get(Tuple<Ts...> &t) noexcept
                    -> std::tuple_element_t<I, Tuple<Ts...>>& {
    return t.data.[:t.nth_nsdm(I):];
  }


...
```

P3294: Code Injection with Token Sequences

Andrei Alexandrescu

Barry Revzin

Semantic generation

E.g., P2996's define_class, Swift macros, Code Reckons API

Code injection

Grammatical code injection

E.g., modern templates, Lock3 fragments

Token injection

E.g., original templates, Rust macros

We pick this!

String injection

E.g., D, CppFront

```cpp
#include <experimental/meta>
include <iostream>

consteval std::meta::info make_output_stmt() {
    return ^^{ std::cout << "Hello, World!"; };
}

int main() {
  queue_injection(make_output_stmt());
}
```

```cpp
#include <experimental/meta>

template<bool B, typename T = void> struct enable_if {
  consteval {
    if (B) queue_injection(^^{ using type = T; });
  }
};

enable_if<true, int*>::type p = nullptr;
// enable_if<false, int>::type i = 42;  // Would be error.
```

```
consteval auto make_field(info type, string_view name, int val) {
  return ^^{ [:\(type):] \id(name) = \(val*2); };
}

consteval auto make_func(info type, string_view name, info body) {
  return ^^{ [:\(type):] \id(name)() \tokens(body) };
}

struct S {
  consteval {
    queue_injection(make_field(^^int, "x", 21));
    queue_injection(make_func(^^int, "f", ^^{ { return 42; } }));
  }
};

int main() {
  return S{}.x != S{}.f();
}
```

Automatic Type Erasure

Barry Revzin

```
template<typename> class Dyn { ... };
struct Interface {
  void draw(std::ostream&) const;
};
int main() {
  struct Hello {
    void draw(std::ostream &os) const { os << "Hello\n"; }
  } hello;
  struct Number {
    int i;
    void draw(std::ostream &os) const { os << "Number{" << i << "}\n"; }
  } one{1}, two{2};
  std::vector<Dyn<Interface>> v = {one, h, two};
  for (auto &dyn : v) {
    dyn.draw(std::cout);
  }
}
```

```
struct Interface {
  void draw(ostream&) const;
};
```

```
class Dyn<Interface> {
  void *data;

  struct VTable {
    void (*draw)(void*, ostream&);
  } const *vtable;

  template<typename T> static constexpr VTable vtable_for = {
      +[](void *data, ostream &p0) -> void {
         { return static_cast<T const*>(data)->draw(p0); }
    };

public:
  void draw(ostream& p0) const
    { return vtable->draw(data, p0); }

  template<typename T>
    Dyn(T&& t): vtable(&vtable_for<remove_cvref_t<T>>)
              , data(&t)
      {}

  Dyn(Dyn&) = default;
  Dyn(Dyn const&) = default;
};
```

```
template<typename Interface> class Dyn {
  void* data;

  consteval {
    inject_Vtable(^^Interface);
  };

  consteval {
    inject_vtable_for(^^Interface);
  };


public:
  consteval {
    inject_interface(^^Interface);
  };

  consteval {
    inject_erasing_ctor();
  };

  Dyn(Dyn&) = default;
  Dyn(Dyn const&) = default;
};
```

```
class Dyn<Interface> {
  void *data;

  struct VTable {
    void (*draw)(void*, ostream&);
  } const *vtable;

  template<typename T> static constexpr VTable vtable_for =
      +[](void *data, ostream &p0) -> void {
         { return static_cast<T const*>(data)->draw(p0); }
    };

public:
  void draw(ostream& p0) const
    { return vtable->draw(data, p0); }

  template<typename T>
    Dyn(T&& t): vtable(&vtable_for<remove_cvref_t<T>>)
              , data(&t)
      {}

  Dyn(Dyn&) = default;
  Dyn(Dyn const&) = default;
};
```

```
consteval void inject_Vtable(info  interface) {
  std::meta::list_builder  vtable_members(^^{});
  for (info mem : members_of(interface)) {
    if (is_function(mem) && !is_special_member(mem) &&
        !is_static_member(mem)) {
      info  r = return_type_of(mem);
      auto  name = identifier_of(mem);
      std::meta::list_builder  params(^^{,});
      params += ^^{ void* };
      params += param_tokens(parameters_of(mem));
      vtable_members += ^^{
        [:\(r):] (*\id(name))(\tokens(params));
      };
    }
  }
  queue_injection(^^{
    struct VTable {
      \tokens(vtable_members)
    } const *vtable;
  });
}
```

```
class Dyn<Interface> {
  void *data;

  struct VTable {
    void (*draw)(void*, ostream&);
  } const *vtable;

  template<typename T> static constex
      +[](void *data, ostream &p0) ->
        { return static_cast<T const
  };

public:
  void draw(ostream& p0) const
    { return vtable->draw(data, p0);

  template<typename T>
    Dyn(T&& t): vtable(&vtable_for<re
              , data(&t)
    {}

  Dyn(Dyn&) = default;
  Dyn(Dyn const&) = default;
};
```

```
consteval info param_tokens(vector<info>  params,
                            string_view   prefix = "") {
  list_builder  result(^^{,});
  for (int k = 0; info p : params) {
    p = type_of(p);
    if (prefix.size() != 0) {
      result += ^^{ typename[:\(p):] \id(prefix, k++) };
    } else {
      result += ^^{ typename[:\(p):] };
    }
  }
  return result;
}
```

```
class Dyn<Interface> {
  void *data;

  struct VTable {
    void (*draw)(void*, ostream&);
  } const *vtable;

  template<typename T> static constex
      +[](void *data, ostream &p0) ->
        { return static_cast<T const
    };

public:
  void draw(ostream& p0) const
    { return vtable->draw(data, p0);

  template<typename T>
    Dyn(T&& t): vtable(&vtable_for<re
              , data(&t)
      {}

  Dyn(Dyn&) = default;
  Dyn(Dyn const&) = default;
};
```

```
template<typename> class Dyn { ... };
struct Interface {
  void draw(std::ostream&) const;
};
int main() {
  struct Hello {
    void draw(std::ostream &os) const { os << "Hello\n"; }
  } hello;
  struct Number {
    int i;
    void draw(std::ostream &os) const { os << "Number{" << i << "}\n"; }
  } one{1}, two{2};
  std::vector<Dyn<Interface>> v = {one, h, two};
  for (auto &dyn : v) {
    dyn.draw(std::cout);
  }
}
```

P3394: Annotations for Reflection

(aka. "custom attributes")

Wyatt Childers **Dan** Katz **Barry** Revzin

```cpp
template<typename T>
unsigned long hash(T const &obj) {
  unsigned long result = 17;
  [:expand(nonstatic_data_members_of(^^T)):] >>
  [&]<info dm> {
    // ... delicious meta stuff with obj.[:dm:]
  };
  return result;
}
```

```
namespace __impl {
  template<auto... vals>
  struct replicator_type {
    template<typename F>
      constexpr void operator>>(F body) const {
        (body.template operator()<vals>(), ...);
      }
  };

  template<auto... vals>
  replicator_type<vals...> replicator = {};
}

template<typename R>
consteval auto expand(R range) {
  std::vector<std::meta::info> args;
  for (auto r : range) {
    args.push_back(std::meta::reflect_value(r));
  }
  return substitute(^^__impl::replicator, args);
}
```

```cpp
template<typename T>
unsigned long hash(T const &obj) {
  unsigned long result = 17;
  [:expand(nonstatic_data_members_of(^^T)):] >>
  [&]<info dm> {
    // ... delicious meta stuff with obj.[:dm:]
  };
  return result;
}
```

```cpp
struct Ultra {
  float data[3];
  Cache cache;
};
void tada(Ultra const &ultra) {
    ... hash(ultra) ...
}
```

```
enum class HashNotes { ignore };
```

```
struct Ultra {
  float data[3];
  Cache cache [[=HashNotes::ignore]];
};
void tada(Ultra const &ultra) {
    ... hash(ultra) ...
}
```

```
enum class HashNotes { ignore };
template<typename T>
unsigned long hash(T const &obj) {
  unsigned long result = 17;
  expand[:nonstatic_data_members_of(^^T):] >>
  [&]<info dm> {
    if (annotation_of_type<HashNotes>(dm) != HashNotes::ignore) {
      // ... delicious meta stuff with obj.[:dm:]
    }
  };
  return result;
}
struct Ultra {
  float data[3];
  Cache cache [[=HashNotes::ignore]];
};
void tada(Ultra const &ultra) {
    ... hash(ultra) ...
}
```

Command-Line Argument Parsing

Barry Revzin

```cpp
// ...

struct Args: clap::Clap {
    [[=Help("Name to greet")]]
    [[=Short, =Long]]
    std::string name;

    [[=Help("Number of times to greet")]]
    [[=Long("repeat")]]
    int count = 1;
};

int main(int argc, char** argv) {
    Args args;
    args.parse(argc, argv);
    for (int i = 0; i < args.count; ++i) {
        std::cout << "Hello " << args.name << "!\n";
    }
}
```

```cpp
#include <lyra/lyra.hpp>

namespace clap {

struct ShortArg { ... };
struct LongArg { ... };
struct HelpArg { ... };

struct Clap {
  static constexpr auto Short = ShortArg();
  static constexpr auto Long = LongArg();
  static constexpr auto Help(char const *msg) {
    return HelpArg{msg};
  };
  template<typename Args>
    void parse(this Args &args, int argc, char **argv);
};
// ...
}
```

```
template<typename Args>
void Clap::parse(this Args &args, int argc, char **argv) {
  bool show_help = false;
  auto cli_parser = lyra::cli() | lyra::help(show_help);
  configure(&args, &cli_parser);
  auto result = cli_parser.parse({argc, argv});
  if (not result) {
    std::cerr << "Error in command line: " << result.message() << '\n';
  }
  if (show_help || not result) {
    std::cerr << cli_parser << '\n';
    exit(not result);
  }
}
```

```
static consteval auto clap_annotations_of(info dm) {
  auto notes = annotations_of(dm);
  std::erase_if(notes, [](info  ann) { return parent_of(type_of(ann)) != ^^clap; });
  return notes;
}

template<typename Args, typename Parser>
void configure(Args *args, Parser *parser) {
  [:expand(nonstatic_data_members_of(^^Args)):] >> [&]<info mem> {
    std::string id(identifier_of(mem));
    lyra::opt opt_parser(args->[:mem:], id);
    [:expand(clap_annotations_of(mem)):] >> [&]<info ann> {
      using Ann = [: type_of(ann) :];
      extract<Ann>(ann).apply(opt_parser, id);
    };
    parser->add_argument(opt_parser);
  };
}
```

```
struct LongArg {
  bool engaged = false;
  char const* value = "";
  constexpr LongArg operator()(char const *s) const {
    return {.engaged=true, .value=s};
  };
  void apply(lyra::opt &opt, string const &id) const {
    opt[string("--") + string(engaged ? value : id)];
  }
};

struct HelpArg {
  char const* msg;
  void apply(lyra::opt &opt, string const &id) const {
    opt.help(msg);
  }
};
```

Thank You!

SG 7 **Chandler** Carruth **Hana** Dusíková

Matús Chochlík **Axel** Naumann **David** Sankel

Jagrut Dave **Tomasz** Kamiński **Saksham** Sharma

SG 16 **EWG** LEWG **CWG** LWG

Adam Lach **Walter** Genovese **Herb** Sutter

Nvidia **Bloomberg** **Jump** Trading

EDG