Contents

- Metadata, metaobjects, reflection, “un-reflection”
- Enumeration/string conversions
- Serialization and deserialization
- Parsing of application options
  - from command-line arguments,
  - from configuration files
- Remote procedure call stubs and skeletons
- Generic ReST API adapter
- Scripting language bindings
- Generating UML diagrams
- Fetching of structured data from a RDB
- Generating SQL queries
- . . .

Crash course

metadata, metaobjects, reflection

Metadata – describing program declarations

- type of a variable,
- data members of a `struct`,
- constructors or member functions of a `class`,
- base classes of a `class`,
- return type or parameters of a function,
- enumerators in an `enum` type,
- name of a namespace, type, function, data member, parameter, etc.
- address of a variable, data member or member function,
- specifier like `virtual`, `constexpr`, `static`, etc.
- source location,
- ...

Metaobject

- A *meta-level* representation of a *base-level* entity[1]
- On the *meta-level*[2] all of the above are "reified"
- We say a metaobject "reflects" the *base-level* entity
- Provides access to *metadata* describing the reflected *base-level* entity
- A compile-time constant value of a type satisfying the following concept:

```
template <typename X>
concept metaobject = not-really-important-here;
```

[1] namespace, type, class, function, constructor, destructor, variable, enumerator, expression, ...

[2] unlike the base-level (namespaces, constructors, etc)

Reflection

- The process of obtaining metadata or metaobjects which provide metadata indirectly
- Done through a dedicated operator or language expression
- For example

```
auto meta_string =
  mirror(string);
```

. . . and "un-reflection" [3]

- Getting back to the base-level entity reflected by a metaobject
- . . . through an operation on a metaobject reflecting that *base-level* entity
- "Splicing" can also mean emitting a snippet of code involving base-level entities, reflected by metaobjects

[3] a.k.a. ""splicing""

Reflection API

Set of compile-time[4] functions operating on *metaobjects*

- classification
  - `reflects_type`, `reflects_callable`, ...
- primitive operations
  - `get_name`, `get_scope`, `get_type`, `get_aliased`, `get_enumerators`, `is_constexpr`, `is_scoped_enum`, `invoke`, `apply`, ...
- sequence operations
  - `is_empty`, `get_size`, `get_element`, `concat`, `flatten`, ...
- algorithms
  - `for_each`, `fold`, `join`, `transform`, `filter`, `remove_if`, `sort_by`, `group_by`, `find_if`, `find_if_not`, `find_ranking`, ...
- comparators
  - `reflect_same`, ...
- syntax sugar, placeholder expressions
  - `_1`, `_2`, `get_type(_1)`, `reflect_same(_1, _2)`, ...

[4] `consteval`, `constexpr` assumed, but omitted here to save space on slides

“hello world of reflection!”

```
enum class greeting {
  hello, world, of, reflection
};
```

```
cout << join[5](
        get_enumerators[6](mirror[7](greeting)[8]),
        to_string(get_name[9](_1[10]))[11],
        string(" "))
     << "!\n";
```

[5] algorithm
[6] metaobject sequence getter
[7] reflection operator
[8] reflection in action – returns a metaobject
[9] metadata (name) getter
[10] placeholder
[11] placeholder expression

Extractable – concepts

- Types which can optionally refer to or store a value
- Unifies usage of
  - raw pointers, smart pointers,
  - `optional`, `expected`,
  - ...
- ... in some specific cases

```
template <typename T>
concept extractable = requires(T v) {
  { declval<extracted_type_t<T>>() };
  { has_value(v) } -> convertible_to<bool>;
  extract(v);
};
```

Extractable – operations

- `has_value_type` – indicates if the extracted value has a specific type
- `has_value` – indicates if an extractable has a value
- `extract` – provides access to a value in an extractable

```
template <typename V>
consteval auto has_value_type(
  const extractable auto& v)
  noexcept -> bool;
```

```
auto has_value(
  extractable auto &) noexcept -> bool;
```

```
auto extract(extractable auto&) -> auto&;
auto extract(extractable auto&&) -> const auto&&;
auto extract(extractable const auto&) -> const auto&;
```

Extractable – the idiom

This is quite common...

```
auto get_opt_val(auto... params)
  -> extractable;
```

```
if(const auto opt{get_opt_val(args...)};
   has_value(opt)) {

  do_something(extract(opt));
} else {
  do_something_else();
}
```

Conversion from string

```
template <typename T>
auto from_string(
  const string_view src,
  type_identity<T> = {})
    -> extractable;
```

```
template <typename T>
auto from_extractable_string(
  const extractable auto src,
  type_identity<T> tid = {})
    -> extractable
  requires(has_value_type<string_view>(src));
```

Conversion from string – common combo with "un-reflection"

```
auto get_reflected_type(metaobject auto mo)
    -> type_identity<unspecified>
  requires(reflects_type(mo));
```

```
auto mo = mirror(some_type);
```

```
auto opt_val = from_string(
  get_some_string(),
  get_reflected_type(mo));
```

Command-line arguments

```
class program_arg {
public:
  auto next() -> program_arg;

  auto is_short_tag(
    string_view) -> bool;
  auto is_long_tag(
    string_view) -> bool;
  // ...
  operator string_view();
};
```

```
class program_args {
public:
  program_args(int, const char**);
  auto begin();
  auto end();
  auto command() -> string_view;
  // ...
  auto find(string_view)
    -> program_arg;
};
```

- `program_arg` – represents a single program argument.
  - get previous and next argument,
  - check for `-o` and `--long-opt` options,
  - starts-with, ends-with,
  - ...
- `program_args` – represents all program arguments
  - iteration, search,
  - command, first, last,
  - ...

Enumeration

conversion utilities[12]

[12] let's start easy. . .

Enumeration conversions – enum-to-string

```
template <typename E>
auto enum_to_string(E e) noexcept
  -> string_view {

  return choose(
    string_view{},
    get_enumerators(mirror(E)),
    has_value(_1, e),
    get_name(_1));
}
```

Enumeration conversions – string-to-enum

```
template <typename E>
auto string_to_enum(string_view s) noexcept
  -> optional<E> {

  return choose(
    optional<E>{},
    get_enumerators(mirror(E)),
    has_name(_1, s),
    get_value(_1));
}
```

Enumeration conversions – example

```
enum class weekdays : int { monday, tuesday, /*...*/ };
```

```
weekdays next_day(weekdays d);
```

```
void print_next_day(string_view name) {
  if(auto opt_day{string_to_enum<weekdays>(name)};
     has_value(opt_day)) {
    cout << name << " -> " << enum_to_string(
              next_day(extract(opt_day)))
         << endl;
  }
}
```

```
for_each(
  get_enumerators(mirror(weekdays)),
  [](auto mo) {
    print_next_day(get_name(mo));
});
```

```
monday -> tuesday
...
```

Serialization
and deserialization[13]

[13] the *cliché* of reflection use-cases

Serialization – read-backend concept

```
template <typename T>
concept read_backend =
  requires(T v) {

  { v.enum_as_string(
      declval<T::context&>())
  } -> convertible_to<bool>;

  { v.begin(declval<T::context&>())
  } -> extractable;

  { v.read(
      declval<read_driver>(),
      declval<T::context&>(),
      declval<unspecified&>())
  } -> same_as<read_errors>;

  { v.begin_list(
      declval<T::context&>(),
      declval<size_t&>())
  } -> extractable;
```

...

```
  { v.begin_element(
      declval<T::context&>(),
      declval<size_t&>())
  } -> extractable;
```

...

```
  { v.separate_element(
      declval<T::context&>())
  } -> same_as<read_errors>;

  { v.finish_element(
      declval<T::context&>(),
      declval<size_t>())
  } -> same_as<read_errors>;

  { v.finish_list(
      declval<T::context&>())
  } -> same_as<read_errors>;

  { v.begin_record(
      declval<T::context&>(),
      declval<size_t&>())
  } -> extractable;

  { v.begin_attribute(
      declval<T::context&>(),
      declval<string_view>())
  } -> extractable;
```

...

```
    { v.finish(
        declval<T::context&>())
    } -> same_as<read_errors>;
};
```

Serialization – write-backend concept

```
template <typename T>
concept write_backend =
  requires(T v) {

  { v.enum_as_string(
      declval<T::context&>())
  } -> convertible_to<bool>;

  { v.begin(declval<T::context&>())
  } -> extractable;

  { v.write(
      declval<write_driver>(),
      declval<T::context&>(),
      declval<const unspecified&>())
  } -> same_as<write_errors>;

  { v.begin_list(
      declval<T::context&>(),
      declval<size_t&>())
  } -> extractable;
```

...

```
  { v.begin_element(
      declval<T::context&>(),
      declval<size_t&>())
  } -> extractable;
```

...

```
  { v.separate_element(
      declval<T::context&>())
  } -> same_as<write_errors>;

  { v.finish_element(
      declval<T::context&>(),
      declval<size_t>())
  } -> same_as<write_errors>;

  { v.finish_list(
      declval<T::context&>())
  } -> same_as<write_errors>;

  { v.begin_record(
      declval<T::context&>(),
      declval<size_t&>())
  } -> extractable;

  { v.begin_attribute(
      declval<T::context&>(),
      declval<string_view>())
  } -> extractable;
```

...

```
    { v.finish(
        declval<T::context&>())
    } -> same_as<write_errors>;
};
```

Serialization – `deserializer`

Default implementation for types that can be read directly by the *backend*

```
template <typename T>
struct deserializer {
  template <read_backend Backend>
  static auto read(
    const read_driver& driver,
    Backend& backend,
    typename Backend::context_param ctx,
    T& value) noexcept {
      // delegate the work to the backend
      return backend.read(driver, ctx, value);
  }
};
```

Serialization – deserializer – enum types

```
static auto read(
  const read_driver& driver,
  read_backend auto& backend, /*...*/ ctx,
  T& value) noexcept {
    read_errors errors{};

    if(backend.enum_as_string(ctx)) {
        string name;
        errors |= driver.read(backend, ctx, name);

        if(const auto conv{string_to_enum<T>(name)};
           has_value(conv)) {
            value = extract(conv);
        } else {
            errors |= read_error_code::invalid_format;
        }
    } else {
        underlying_type_t<T> temp{};
        errors |= driver.read(backend, ctx, temp);
        value = static_cast<T>(temp);
    }
    return errors;
}
```

Serialization – deserializer – classes

```
static auto read(
  const read_driver& driver,
  read_backend auto& backend, /*...*/ ctx,
  T& value) noexcept {
    read_errors errors{};
```

```
    const auto mdms{filter(get_data_members(mt), not_(is_static(_1)))};
    size_t count{get_size(mdms)};
    auto subctx{backend.begin_record(ctx, count)};

    if(has_value(subctx)) {
      bool first = true;
      for_each(mdms, [&](auto mdm) {
          errors |= backend.separate_attribute(extract(subctx));

          const auto name{get_name(mdm)};
          auto subsubctx{backend.begin_attribute(extract(subctx), name)};
          if(has_value(subsubctx)) {
            errors |= driver.read(
              backend, extract(subsubctx), get_reference(mdm, value));
            errors |= backend.finish_attribute(extract(subsubctx), name);
          } else {
            errors |= get_error(subsubctx);
          }
      });
      errors |= backend.finish_record(extract(subctx));
    } else {
      errors |= get_error(subctx);
    }
    return errors;
}
```

Serialization – `read_driver`

Used by the *backend* and some `deserializer` specializations.
Creates appropriate nested `deserializer` and uses it.

```
struct read_driver {
  template <typename T, read_backend Backend>
  auto read(
    Backend& backend,
    typename Backend::context_param ctx,
    T& value) const -> read_errors {
      deserializer<remove_cv_t<T>> reader;
      return reader.read(
        *this,
        backend,
        ctx,
        value);
  }
};
```

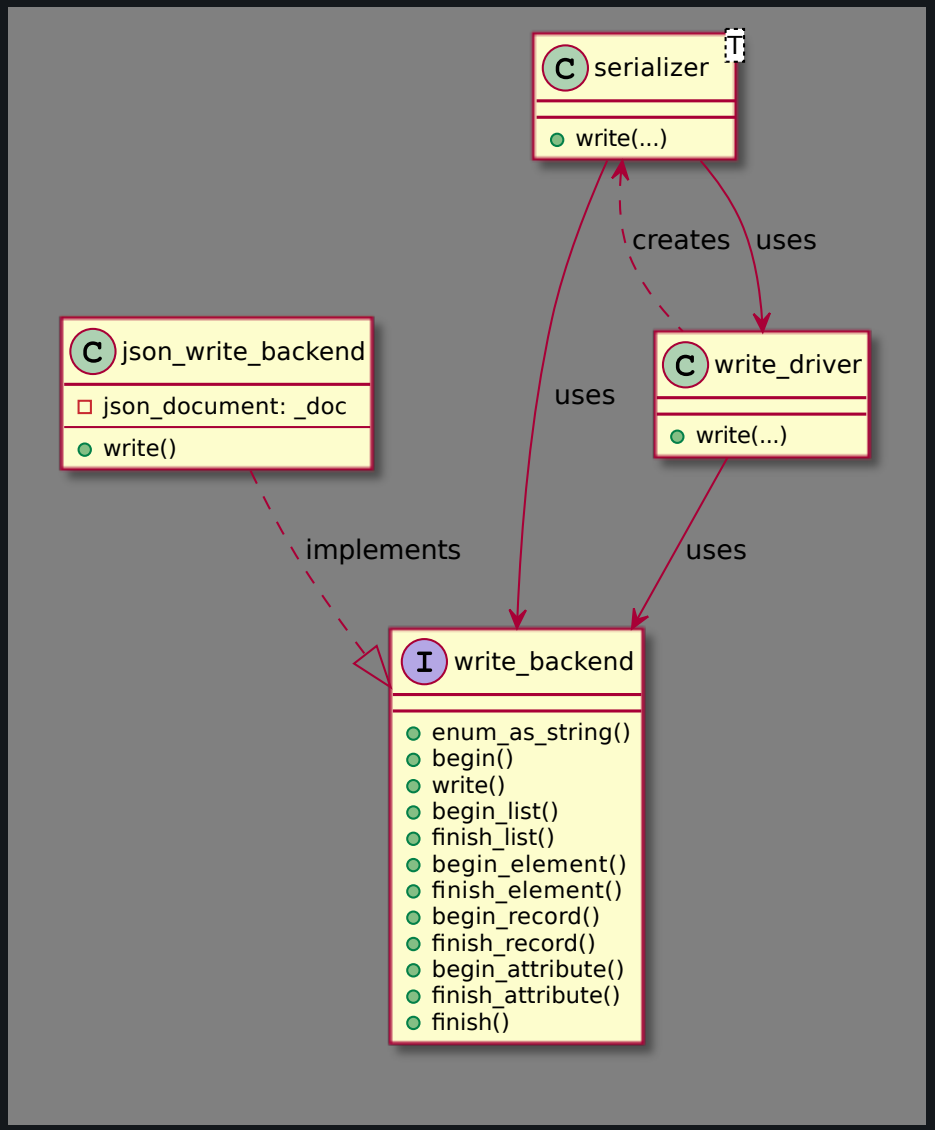
Serialization

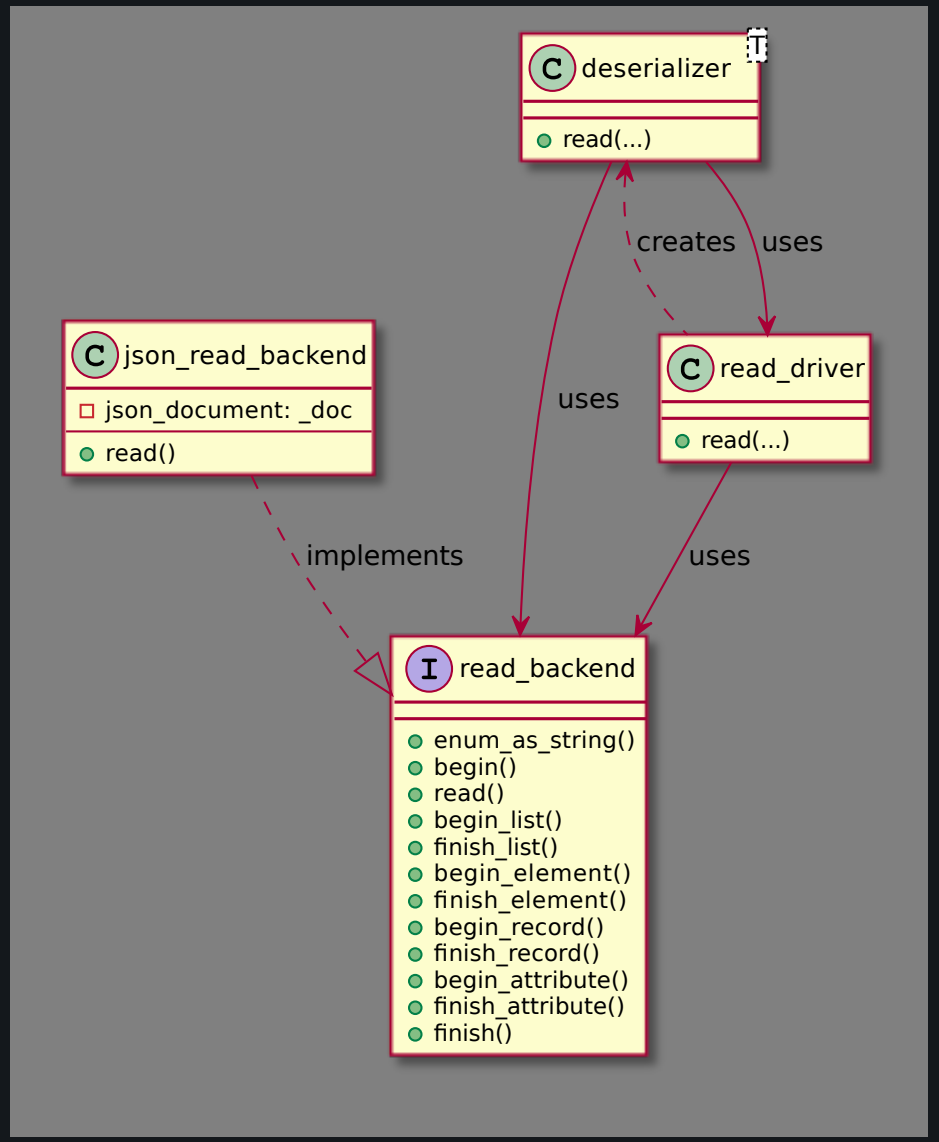
Writing is done analogously:

```
template <typename T>
struct serializer {
  template <write_backend Backend>
  static auto write(
    const write_driver& driver,
    Backend& backend,
    typename Backend::context_param ctx,
    T& value) noexcept;
};
```

```
struct write_driver {
  template <typename T, write_backend Backend>
  auto write(
    Backend& backend,
    typename Backend::context_param ctx,
    T& value) const -> write_errors;
};
```

Serialization – how it fits together. . .

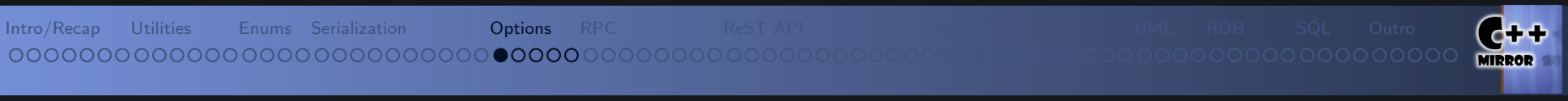




Serialization – the generic API functions

```
template <typename T, write_backend Backend>
auto write(
  const T& value,
  Backend& backend,
  typename Backend::context_param ctx) noexcept
    -> write_errors;
```

```
template <typename T, read_backend Backend>
auto read(
  T& value,
  Backend& backend,
  typename Backend::context_param ctx) noexcept
    -> read_errors;
```



Application options

parsing from command-line arguments
and external configuration files

Parsing command-line arguments – into a structure

- `options` – application-specific data structure storing options
- `parse_args` – generic function that parses and stores command-line argument values into a structure
  - can be implemented using reflection

```
struct options {
  int count{3};
  string message{"Hello, world!"};
  chrono::milliseconds interval{500};
};
```

```
template <typename T>
bool parse_options(T& opts, const program_args&);
```

Parsing command-line arguments – usage

```
int main(int argc, const char** argv) {

    const program_args args{argc, argv};
    options opts;

    if(parse_options(opts, args)) {
      const auto repeats{
        ranges::views::iota(1, opts.count + 1)};

      for(auto i : repeats) {
          cout << i << ": "
               << opts.message << endl;

          this_thread::sleep_for(opts.interval);
      }
    }
}
```

Parsing arguments – how to implement *generic* parse_options?

```
template <typename T>
bool parse_options(T& opts, const program_args& args) {
  bool parsed = true;

  for(const auto& arg : args) {
    for_each(get_data_members(mirror(T)), [&](auto mdm) {
      if(arg.is_long_tag(get_name(mdm))) {
        if(const auto opt{from_string(
          arg.next(), get_reflected_type(get_type(mdm)))}
        ) {
          get_reference(mdm, opts) = extract(opt);
        } else {
          cerr << "invalid value '" << arg.next()
               << "' for option " << arg
               << "!" << endl;
          parsed = false;
        }
      }
    });
  }
  return parsed;
}
```

Loading options from file – good thing we implemented serialization!

```
auto parse(options& opts, istream& cfg_in) -> bool {
  const auto errors =
    read_rapidjson_stream(opts, cfg_in);
  return !errors;
}
```

```
{
  "message": "Hello reflection!",
  "interval": "250ms",
  "count": 4
}
```

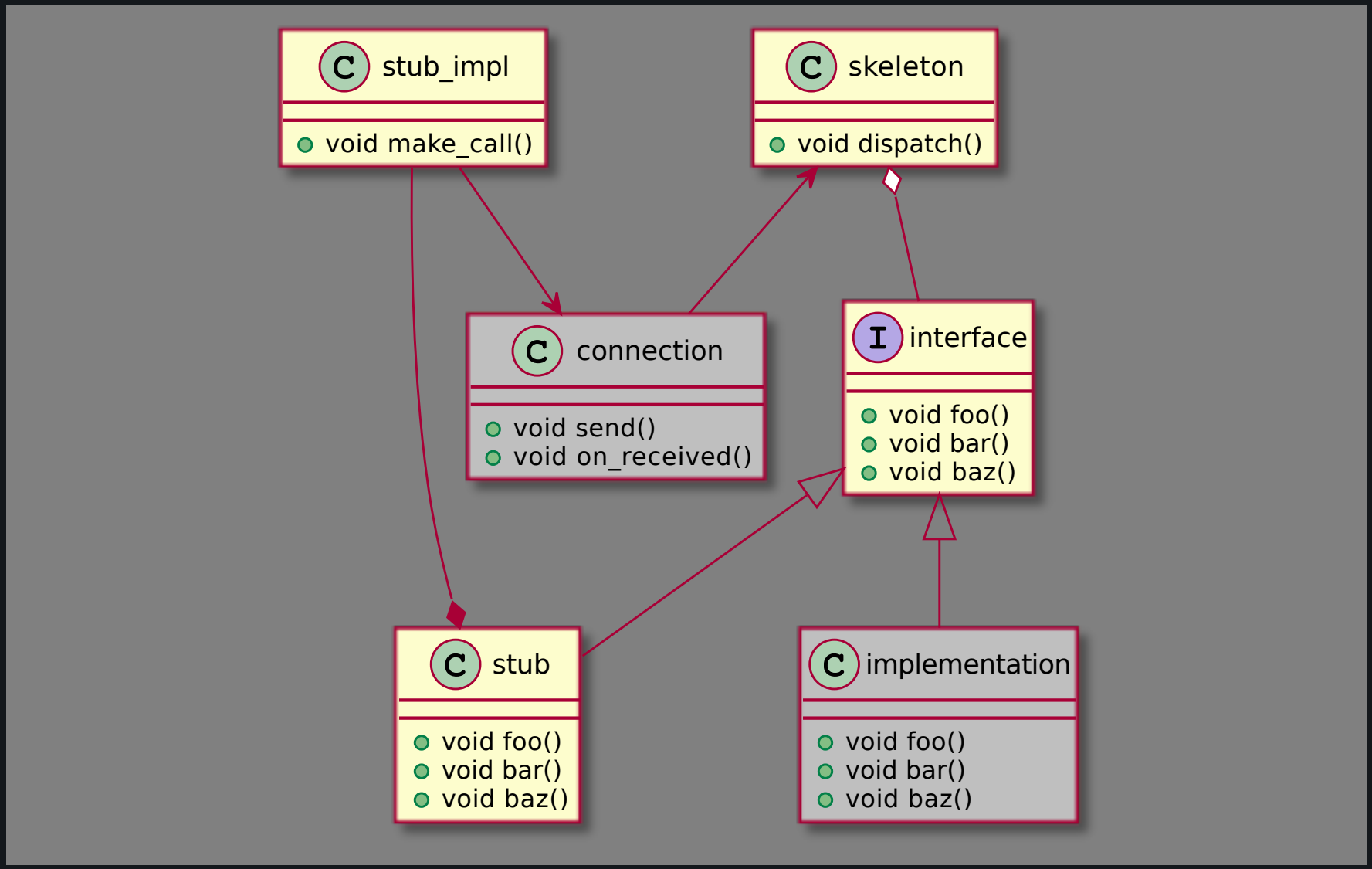
```
options opts;
ifstream cfg_in{"config.json"};
if(parse(opts, cfg_in)) {
    for(int i : ranges::views::iota(1, opts.count + 1)) {
        cout << i << ": " << opts.message << endl;
        this_thread::sleep_for(opts.interval);
    }
}
```

Remote procedure calls

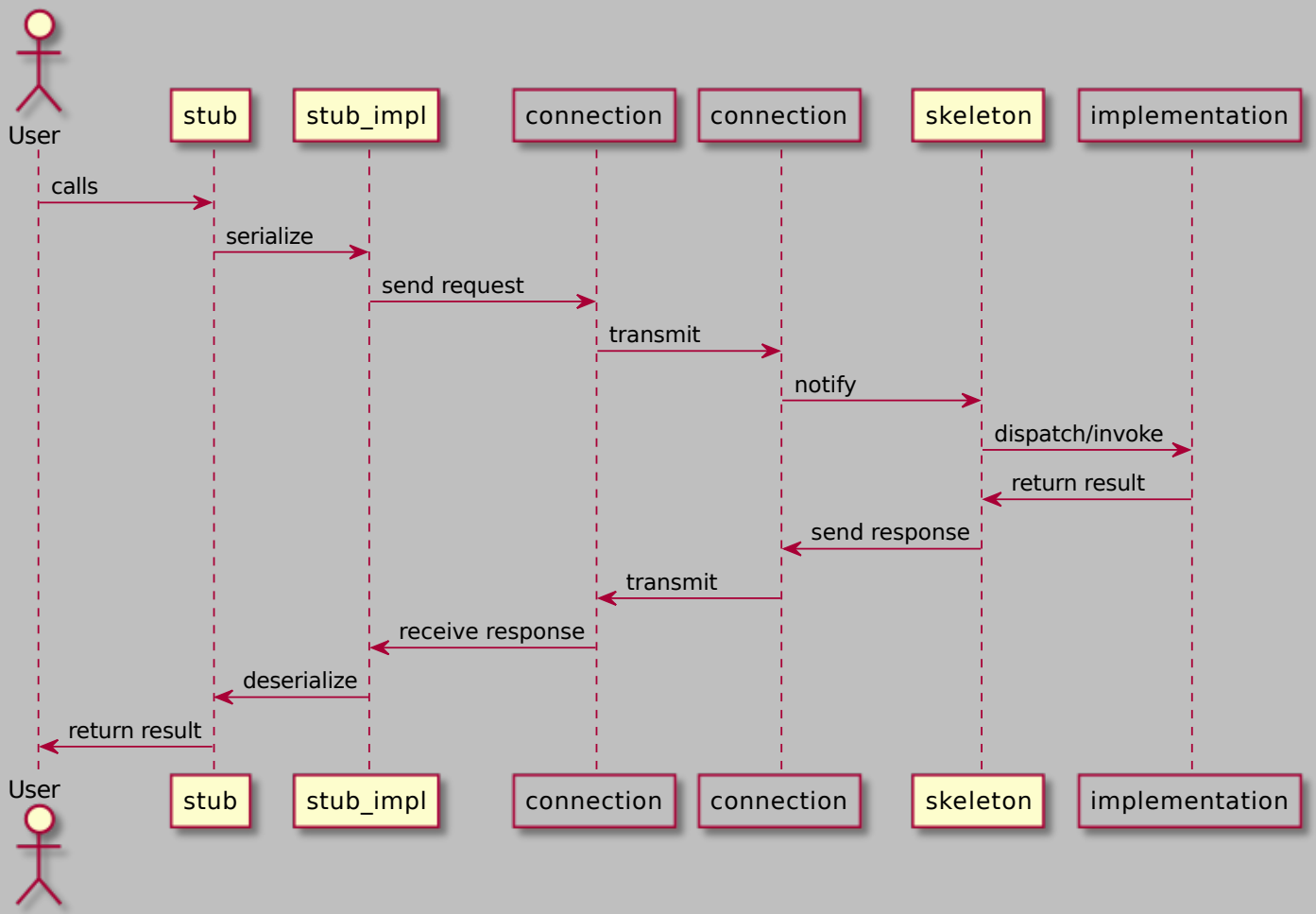
skipping the source code generator[14]

[14] *almost* completely

Remote procedure calls – class overview



Remote procedure calls – synchronous call sequence



RPC stubs/skeletons – the interface

```
struct calculator {
  virtual float add(float, float) = 0;
  virtual float subtract(float, float) = 0;
  virtual float multiply(float, float) = 0;
  virtual float divide(float, float) = 0;
  virtual float negate(float) = 0;
  virtual float invert(float) = 0;
};
```

RPC stubs/skeletons – the stub

```
class calculator_stub : public calculator {
private:
    rpc_stub_impl _impl;
public:
  float add(float l, float r) final {
    return _impl.make_call(
      mirror((calculator::add(l, r)))[15], l, r);
  }

  float subtract(float l, float r) final {
    return _impl.make_call(
      mirror((calculator::subtract(l, r))), l, r);
  }
  float multiply(float l, float r) final;
  float divide(float l, float r) final;
  float negate(float x) final;
  float invert(float x) final;
};
```

[15] expression reflection

RPC stubs/skeletons – the generic stub implementation[16]

```
class rpc_stub_impl {
  template <typename T>
  auto _deserialize(packet&, type_identity<T>) -> T;
public:
  auto make_call(metaobject auto mo, auto&... args) {

    packet request;
    _serialize(request, mo, args...);
    packed response{_send_and_receive(request)};

    return _deserialize(
      response,
      get_reflected_type(
        get_type(
          get_callable(
            get_subexpression(mo)))));
  }
};
```

[16] pseudocode

RPC stubs/skeletons – the skeleton interface

```
struct rpc_skeleton {
  virtual void dispatch(
    packet& request,
    packet& response) = 0;
};
```

- Could be plugged-into a network connection
- Handle incoming data
- After call is finished the connection can send the response

```
class connection {
  unique_ptr<skeleton> _skel;
  // ...
  void send(address host, packet& data);
  void on_received(address host, packet& request) {
    packet response;
    _skel->dispatch(request, response);
    send(host, response);
  }
};
```

RPC stubs/skeletons – the skeleton implementation[17]

```
template <typename Intf>
class rpc_skeleton_impl : public rpc_skeleton {
private:
  unique_ptr<Intf> _impl;
public:
  void dispatch(packet& request, packet& response) final {
    const auto method_id{_get_method_id(request)};

    for_each(get_member_functions(mirror(Intf)),
      [&](auto mf) {
        if(_get_method_id(mf) == method_id) {
          auto params = make_value_tuple(
            transform(get_parameters(mf), get_type(_1)));

          deserialize(params, request);
          auto result = apply(mf, *_impl, params);
          serialize(method_id, result, response);
        }
    });
  }
}
```

[17] pseudocode

Representational state transfer[18] API

automating the boilerplate

[18] where I finally learned what ReST means...

ReST – the API operation result

```
class rest_api_response {
private:
  /* ... */[19]
public:
  void set_result(const auto& result);
  void set_error_code(auto code);
  void set_error_message(
    string_view format,
    const auto&... args);

  auto json_str() const -> string;
};
```

[19] rapidjson inside

ReST – the adapter

```
template <typename Backend>
class rest_api_adapter {
  Backend _backend[20]{};
public:
  auto handle(
    const url& request,
    rest_api_response& response) -> bool {
    if(request) {
      return handle_scheme(request, response);
    } else {
      response.set_error_message(
        "invalid URL `{1}`",
        request.str());
    }
    return false;
  }
};
```

[20] implements the actual application functionality

ReST – some unimportant URL checking...

↓

```
bool handle_scheme(const url& request, rest_api_response& response) {
  if(request.has_scheme(_scheme)) {
    return handle_domain(request, response);
  } else {
    if(const auto scheme{request.scheme()}) {
      response.set_error_code(error_code::invalid_scheme);
    } else {
      response.set_error_code(error_code::missing_scheme);
    }
  }
  return false;
}
```

↓

```
bool handle_domain(const url& request, rest_api_response& response) {
  if(request.has_host(_domain)) {
      return handle_dispatch(request, response);
  } else {
    if(const auto domain{request.host()}) {
      response.set_error_code(error_code::invalid_domain);
    } else {
      response.set_error_code(error_code::missing_domain);
    }
  }
  return false;
}
```

↓

ReST – little more suspense...

```
bool handle_dispatch(
  const url& request,
  rest_api_response& response) {

  bool success{false};
  if(const auto opt_path{request.path()}) {
    bool found{false};

    /* this is where it happens */

    if(!found) {
        response.set_error_code(error_code::invalid_path);
    }
  } else {
    response.set_error_code(error_code::missing_path);
  }
  return success;
}
```

ReST – find the correct function from URL path

↓

```
const auto& path{extract(opt_path)};
for_each(
  filter(
    get_member_functions(mirror(Backend)),
    is_public(_1)),
  [&](auto mf) {
    const auto func_name{get_name(mf)};

    if(path.starts_with("/") &&
       path.ends_with(func_name)) {
        found = true;
        success =
          handle_call(mf, request, response);
    }
  });
```

↓

ReST – . . . scan the function parameters, prepare space for arguments. . .

↓

```
bool handle_call(
  metaobject auto mf,
  const url& request,
  rest_api_response& response) {
    const auto mp{get_parameters(mf)};

    const auto arg_names{
      make_array_of<string_view>(
      mp, get_name(_1))};
    }
    auto arg_values{
      make_value_tuple(
        transform(mp, get_type(_1)))};

    if(handle_args(
      arg_names, arg_values, request, response)) {
      return apply(mf, /* ... */);
    }
    return false;
}
```

↓

ReST – . . . extract the argument values, and make the call

```
template <typename Value>
auto handle_arg(
  string_view name,
  Value& dst,
  const url& request,
  rest_api_response& response) -> bool {
    if(const auto arg{request.argument(name)}) {
      if(const auto value{from_string<Value>(extract(arg))};
         has_value(value)) {
        dst = extract(value);
      }
    }
}
```

```
if(handle_args(
  arg_names, arg_values, request, response)) {
  return apply(
    mf, get_reflected_type(get_type(mf)),
    arg_values, response);
}
```

ReST – the backend

```
class too_smart_home {
public:
  enum class error_code { no_such_room, /* ... */ };

  template <typename T>
  using result = variant<T, error_code>;
  using session_id_t = uintmax_t;

  auto add_user(
    session_id_t sid, string username, string password)
    -> result<string>;
  auto login(string username, string password)
    -> result<session_id_t>;
  auto logout(session_id_t sid) -> result<string>;

  auto open_windows(session_id_t sid, string room_name)
    -> result<string>;
  auto close_windows(session_id_t sid, string room_name)
    -> result<string>;
  auto windows_status(session_id_t sid, string room_name)
    -> result<string>;
  // ...
};
```

ReST – putting it together...

```
rest_api_adaptor<too_smart_home>
  server("https", "home", {});

const auto get =
  [&](const url& request) -> optional<string> {
    rest_api_response response;
    if(server.handle(request, response)) {
      return {response.json_str()};
    } else {
      cerr << server.domain() << ": "
           << response.json_str() << endl;
    }
    return {};
};

const auto show = [&](const auto& result) {
  if(has_value(result)) {
      cout << server.domain() << ": "
           << extract(result) << endl;
  }
};
```

ReST – usage[23]

```
rest_api_adaptor<too_smart_home> server("https", "home");
auto sid[21] =
  get({"https://admin:supersecret[22]@home/login"});
```

```
if(sid) {
  show(get(
    {"https://home/add_user?username=johnny+password=qwerty+sid=" + *sid}));
  show(get({"https://home/logout?sid=" + *sid}));

  sid = get({"https://johnny:qwerty@home/login"});
  show(get({"http://home/shutdown?sid=" + *sid}));
  show(get({"https://home/shutdown?sid=" + *sid}));
  show(get({"https://home/open_windows?room_name=kitchen+sid=" + *sid}));
  show(get({"https://home/window_status?room_name=bathroom+sid=" + *sid}));
  show(get({"https://home/windows_status?room_name=bathroom+sid=" + *sid}));
  show(get({"https://home/close_windows?room_name=study+sid=" + *sid}));
  show(get({"https://home/logout?sid=" + *sid}));
  show(get({"https://home/open_windows?room_name=bedroom+sid=" + *sid}));

  sid = get({"https://admin:supersecret@home/login"});
  show(get({"https://home/shutdown?sid=" + *sid}));
  show(get({"https://home/logout?sid=" + *sid}));
}
```

[21] login session id

[22] sending passwords in URLs; don't try this @home!

[23] *sans* the boring networking part

ReST – the output

```
home: "user added"
home: "Bye, admin!"
home: {"error_code": "invalid_scheme",
       "message": {
         "format": "invalid scheme ‘{1}‘ in request",
         "args": ["http"]}}
home: {"error_code": "permission_denied"}
home: "windows opened"
home: {"error_code": "invalid_path",
       "message": {
         "format": "invalid path ‘{1}‘ in request",
         "args": ["/window_status"]}}
home: "closed"
home: "already closed"
home: "Bye, johnny!"
home: {"error_code": "invalid_session"}
home: "shutdown"
home: {"error_code": "is_offline"}
```

Scripting

language bindings[24]

[24] reflection on a quest. . .

Scripting – is fun, hand-coding bindings, not so much

```
void add_to(chaiscript::ChaiScript& chai, auto mos)
  requires(is_object_sequence(mos));
```

```
void do_add_to(
chaiscript::ChaiScript& chai,
metaobject auto mo[25],
metaobject auto ms[26]) {
  if constexpr(reflects_object_sequence(mo)) {
    /* ... */
  } else if constexpr(reflects_base(mo)) {
    /* ... */
  } else {
    const string name{get_name(mo)};
    if constexpr(reflects_variable(mo)) {
    } else if constexpr(reflects_constructor(mo)) {
    } else if constexpr(reflects_function(mo)) {
    } else if constexpr(reflects_record(mo)) {
    }
  }
}
```

[25] what is being registered

[26] the scope

Scripting – handling sequences[27]

```
void do_add_to(
  chaiscript::ChaiScript& chai,
  metaobject auto mo,
  metaobject auto ms) {
    if constexpr(reflects_object_sequence(mo)) {
      for_each(mo, [&](auto me) {
        do_add_to(chai, me, ms);
      });
    } else if constexpr(reflects_base(mo)) {
      /* ... */
    } else {
      /* ... */
    }
}
```

[27] "That's easy!" – sir Robin of Camelot

Scripting – base classes

```
template <typename Base, typename Derived>
void add_base_class(
  chaiscript::ChaiScript& chai,
  type_identity<Base>,
  type_identity<Derived>) {
    chai.add(chaiscript::base_class<Base, Derived>());
}
```

```
void do_add_to(
  chaiscript::ChaiScript& chai,
  metaobject auto mo,
  metaobject auto ms) {
    if constexpr(reflects_object_sequence(mo)) {
    } else if constexpr(reflects_base(mo)) {
      add_base_class(
        chai,
        get_reflected_type(get_class(mo)),
        get_reflected_type(ms));
    } else {
      /* ... */
    }
}
```

Scripting – variables and data members

```
const string name{get_name(mo)};

if constexpr(reflects_variable(mo)) {
  if constexpr(reflects_record_member(mo)) {
    if constexpr(is_public(mo)) {
        chai.add(chaiscript::fun(
          get_pointer(mo)), name);
    }
  } else {
    chai.add(chaiscript::var(
        get_reference(mo)), name);
  }
}
```

Scripting – constructors

```
template <typename T, typename... P>
void add_constructor(
  chaiscript::ChaiScript& chai,
  type_identity<T>,
  type_list<P...>,
  const string& name) {
    chai.add(chaiscript::constructor<T(P...)>(), name);
}
```

```
const string name{get_name(mo)};

if constexpr(reflects_constructor(mo)) {
  if constexpr(is_public(mo)) {
      add_constructor(
        chai,
        get_reflected_type(get_scope(mo)),
        extract_types(transform(
          get_parameters(mo),
          get_type(_1))),
        name);
  }
}
```

Scripting – functions and operators

```
template <typename From, typename To>
void add_conversion(
  chaiscript::ChaiScript& chai,
  type_identity<From>,
  type_identity<To>) {
    chai.add(chaiscript::type_conversion<From, To>());
}
```

```
if constexpr(reflects_function(mo)) {
  if constexpr(reflects_record_member(mo) && is_public(mo)) {
      if constexpr(reflects_conversion_operator(mo)) {
        if constexpr(!is_deleted(mo)) {
            add_conversion(
              chai,
              get_reflected_type(get_scope(mo)),
              get_reflected_type(get_type(mo)));
        }
      } else {
        if constexpr(!is_deleted(mo)) {
          chai.add(chaiscript::fun(get_pointer(mo)), name);
        }
      }
  } else {
    chai.add(chaiscript::fun(get_pointer(mo)), name);
  }
}
```

Scripting – types and classes

```
template <typename T>
void add_type(
  chaiscript::ChaiScript& chai,
  type_identity<T>,
  const string& name) {
    chai.add(chaiscript::user_type<T>(), name);
}
```

```
const string name{get_name(mo)};

if constexpr(reflects_record(mo)) {
  add_type(chai,  get_reflected_type(mo), name);
  do_add_to(chai, get_base_classes(mo), mo);
  do_add_to(chai, get_member_types(mo), mo);
  do_add_to(chai, get_data_members(mo), mo);
  do_add_to(chai, get_constructors(mo), mo);
  do_add_to(chai, get_member_functions(mo), mo);
  do_add_to(chai, get_operators(mo), mo);
}
```

Scripting – Monty C++'s Flying Circus

```
class scene {
public:
  void person_says(
    const person&, string_view line);
  void person_relocates(
    person& p, string_view how);
  void event_happens(
    string_view what);
  void pause();
};
```

```
class person {
public:
  person(string name);

  auto name() const -> string_view;
  auto current_location() -> auto&;
  auto enter(location&);
  auto is_thrown_to(location&);
  void say(const string&);
};
```

```
class location {
public:
  location(string name, scene&);
  auto name() const -> string_view;
};
```

```
class king : public person {
public:
  king(string name);
};
```

```
class mysterious_force {
public:
  mysterious_force(location&);
  void throw_into_chasm(person&);
};
```

Scripting – set the scene. . .

```
scene at_the_bridge;
chaiscript::ChaiScript chai;

add_to(
  chai, make_sequence(
  mirror(mysterious_force),
  mirror(scene), mirror(location),
  mirror(person), mirror(king), mirror(at_the_bridge)));
```

```
chai[28](R"(
  var road = location("road leading to the bridge", at_the_bridge);
  var bridge = location("bridge of death", at_the_bridge);
  var chasm = location("chasm", at_the_bridge);

  var the_force = mysterious_force(chasm);

  var bridgekeeper = person("the Bridgekeeper");
  var king_arthur = king("king Arthur");
  var sir_lancelot = person("sir Lancelot");
  var sir_robin = person("sir Robin");
  var sir_galahad = person("sir Galahad");
  var sir_bedevere = person("sir Bedevere");
)");
```

[28] should have done this in Python!

Scripting – . . . and *action!*

```
chai(R"(
  bridgekeeper.enter(bridge);
  king_arthur.enter(road);
  sir_lancelot.enter(road);
  sir_robin.enter(road);
  sir_galahad.enter(road);
  sir_bedevere.enter(road);

  bridgekeeper.say(
    "Stop."
    "Who would cross the Bridge of Death "
    "must answer me these questions three, "
    "ere the other side he see.");
  sir_lancelot.say(
    "Ask me the questions, bridgekeeper. "
    "I am not afraid.");
  // ...
  the_force.throw_into_chasm(sir_robin);
  sir_robin.say("Auuuuuuuugh.");

  // ...
)");
```

UML diagrams

without external tools

UML – the entry point

```
template <typename... T>
ostream& print_puml(ostream& out) {
  out << "@startuml\n";
  (..., print_type_puml(
    out, get_aliased(mirror(T))));

  out << "\n";

  (..., print_type_rel_puml(
    out, get_aliased(mirror(T))));
  out << "@enduml\n";

  return out;
}
```

UML – the helpers

```
void print_type_puml[29](auto mt, auto ms);
```

```
auto get_related_type_name(auto mt, auto ms)
  -> string_view ;
```

```
void print_type_rel_puml(ostream& out, auto mt) {
  if constexpr(reflects_record(mt)) {
    for_each(
      get_member_functions(mt),
      [&](auto mf) {
        auto rel_name = get_related_type_name(
          get_type(mf), get_scope(mt));
        if(!rel_name.empty()) {
            out << get_name(mt)
                << " --> " << rel_name << "\n";
        });
      // ...
    }
  }
}
```

[29] is_enum, is_union, get_name, for-each data member, yadi yadi yada...

UML – source and output

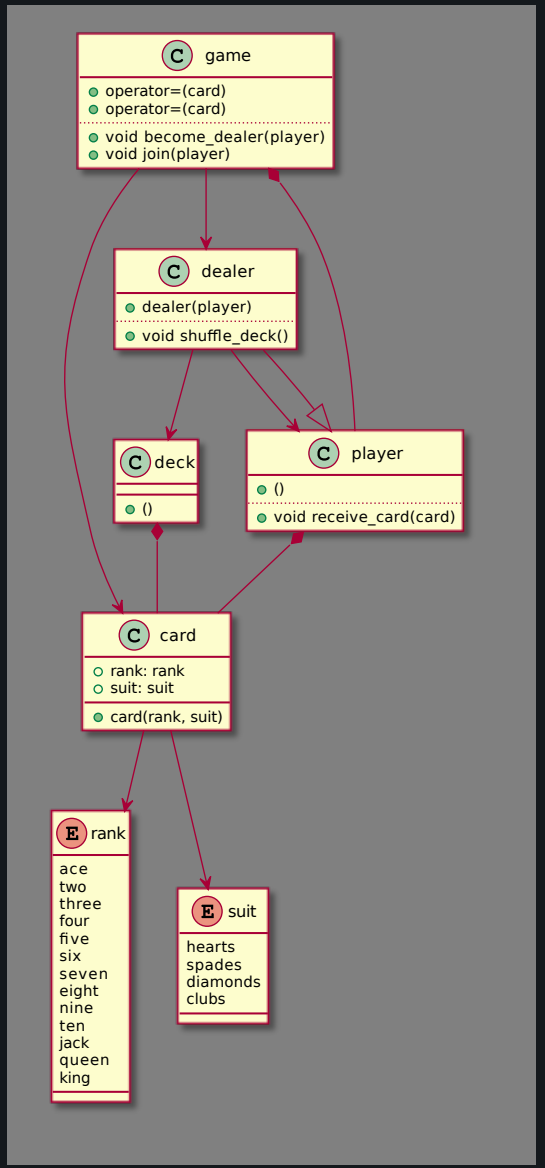
```
enum class rank {
  ace = 1, two, /* ... */ king };
```

```
enum class suit {
  hearts, spades, diamonds, clubs };
```

```
struct card {
  enum rank rank;
  enum suit suit;
};
```

```
class deck {
public:
  auto shuffle(auto& gen) -> deck&;
};
```

```
class player {
public:
  void receive_card(card c);
};
```



Relational DBs

fetching structured data

Fetching data from SQLite3 – table row wrapper

```
class sqlite3_row {
public:
  auto names() const -> span<string_view>;
  auto values() const -> span<string_view>;

  auto size() const -> size_t;
  auto index_of(string_view column_name) const
    -> optional<size_t>;

  auto value(size_t idx) const
    -> optional<string_view>;

  auto value_of(string_view column_name) const
    -> optional<string_view>;

  template <typename T>
  auto fetch(T& instance) const -> bool
    requires(is_class_v<T>);
```

Fetching data from SQLite3 – into a class instance

```
template <typename T>
auto fetch(T& instance) const -> bool
  requires(is_class_v<T>) {
    bool result = true;

    for_each(get_data_members(mirror(T)),
      [&](auto mdm) {
        if(const auto opt_val{from_extractable_string(
              value_of(get_name(mdm)),
              get_reflected_type(get_type(mdm)))};
            has_value(opt_val)) {

          get_reference(mdm, instance) =
            extract(opt_val);
        } else {
          result = false;
        }
      });
    return result;
}
```

Fetching data from SQLite3 – database wrapper

```
class sqlite3_db {
public:
  void execute(
    string_view sql,
    function<void(const sqlite3_row&> callback);

  template <typename T>
  auto fetch(
    string_view sql, vector<T>& dest) -> auto&
    requires(is_class_v<T>) {
      execute(sql, [&](const auto& row) {
        T instance{};
        if(row.fetch(instance)) {
            dest.emplace_back(move(instance));
        }
      });
      return dest;
  }
};
```

Fetching data from SQLite3 – usage

```
struct person {
  uintmax_t person_id;
  string given_name;
  string family_name;
  string email_address;
};
```

```
sqlite3_db db{"people.db"};
vector<person> ps;
```

```
db.ensure_table[30]<person>();

string query{"SELECT * FROM person"};

for(const auto& p : db.fetch(query, ps)) {
  cout << p.given_name << " "
       << p.family_name << endl;
}
```

[30] scans DB schema and ensures that there is a matching person table

RDB queries

generating SQL from C++

Generating SQL queries – the schema

```
struct person {
    string given_name;
    string family_name;
    string email;
};
```

```
template <typename Impl>
struct operations {
  Impl _impl;
public:
  template <typename T>
  using result = typename Impl::result<T>;

  auto get_by_given_name(string_view name)
    -> result<person> {
    return _impl(mirror((get_by_given_name(name))), name);
  }
  auto get_by_email(string_view email)
    -> result<person> {
    return _impl(mirror((get_by_email(email))), email);
  }
};
```

Generating SQL queries – the implementation

```
class query_generator_impl {
public:
  template <typename T>
  struct result : string {
      result(string s) : string{move(s)} {}
  };

  auto operator()(metaobject auto me, const auto& arg)
    -> string {
    const auto mf = get_callable(get_subexpression(me));
    const auto mt = get_type(mf);

    stringstream query;
    query << "SELECT * FROM ";
    query << _table_name(get_reflected_type(mt));
    query << " WHERE ";
    query << get_name(mf).substr("get_by_"sv.size());
    query << " = " << quoted(arg);
    query << ";";
    return query.str();
  }
};
```

Generating SQL queries – usage

```
using query_generator =
  operations<query_generator_impl>;

query_generator gen;

cout << gen.get_by_first_name("Joe")
     << endl;
cout << gen.get_by_email("joe@example.com")
     << endl;
```

Output:

```
SELECT * FROM person WHERE first_name = 'Joe';
SELECT * FROM person WHERE email = 'joe@example.com';
```

Almost done

Conclusions

- Reflection is not complicated
- Reflection has many use-cases
  - as we have seen, but also:
  - defining concepts based on metadata,
  - implementing the *factory* pattern,
  - generating GUIs for visualization and data input,
  - generating `boost::spirit` parsers and formatters,
  - ...
- We are just beginning

What's next for reflection?

- More things to reflect
  - templates
  - specifiers
  - ...
- More "un-reflection"
  - code fragment splicing
  - splicing *identifiers* depending on metaobjects[31]
- Support for more use-cases

[31] maybe even with custom formatting

Links

- *The Mirror library repository* – `https://github.com/matus-chochlik/mirror`
- *The Mirror library reference (W.I.P.)* – `https://matus-chochlik.github.io/mirror/doxygen/`
- *Reflection TS in `clang`* – `https://github.com/matus-chochlik/llvm-project`

That's all...

Thanks for your attention!

Happy to answer any additional questions.