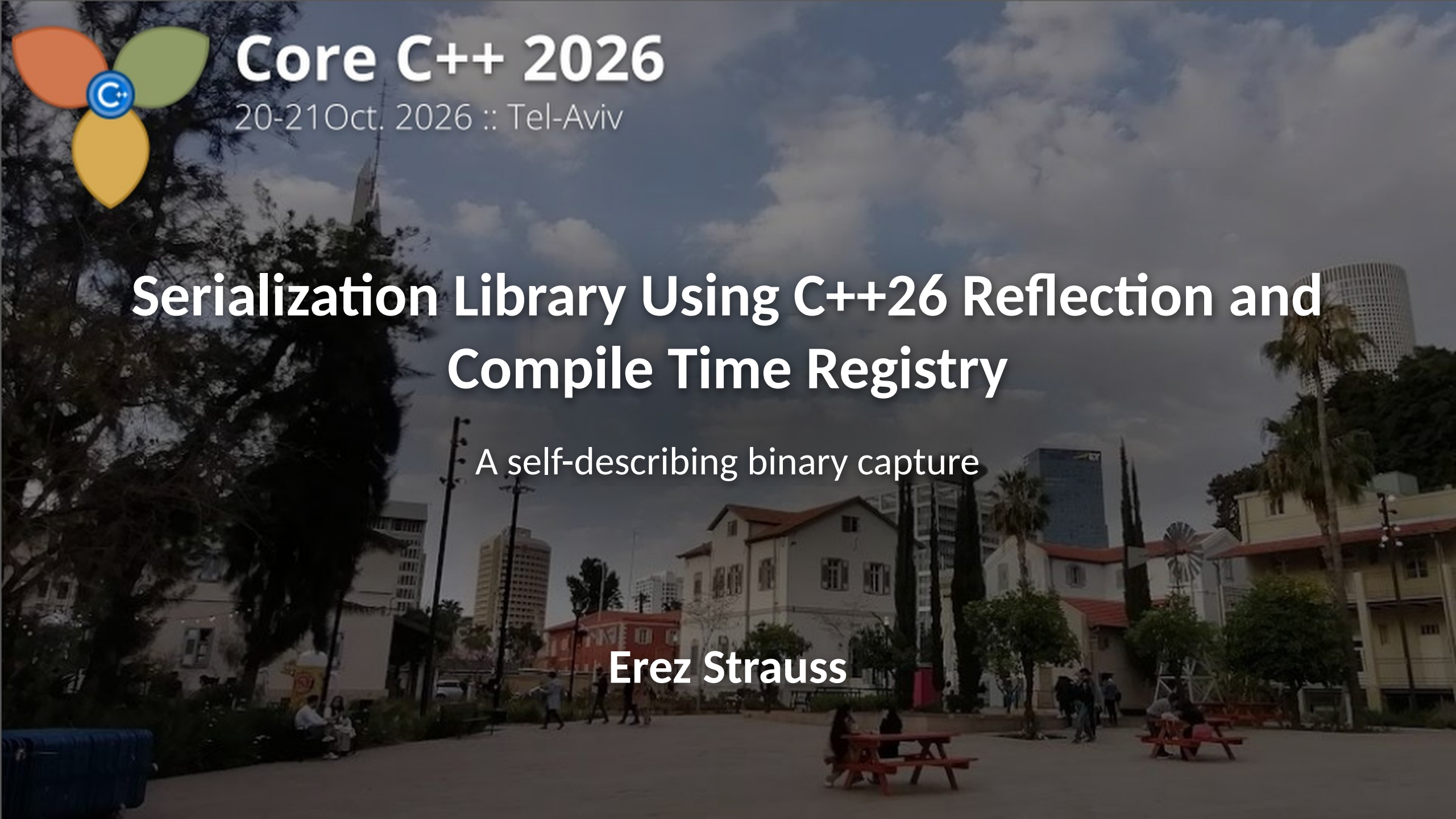


Serialization Library Using C++26 Reflection and Compile Time Registry

A self-describing binary capture

Erez Strauss

This talk in one slide

- **Two C++ mechanisms: static reflection** (P2996) that generates *data* — a type's schema, names and hashes — at compile time, and **odr-use registration** that collects it before `main()`
- **One worked example:** a binary serialization library — `serialize(out, "new order"_, order)`, nothing to declare, register or generate; the file carries its own dictionary
- **The two questions** the mechanisms answer: how do *other programs* read the raw bytes? how do we hold a **dictionary of all message types**?
- Plan: the example in 5 minutes, then reflection (12), then registration (12), then the bytes and the decisions



The goals of the serialization library

- **Nothing to write but the struct** — no IDL, no macros, no generator, no registration call: the type is the schema
- **The writer knows everything before `main()`** — every type, literal and call is registered pre-main, so the dictionary is written **in-band**: ahead of the frames, into every binary, out of a running process
- **The readers get everything from the data** — the dictionary travels with the bytes; a reader parses the raw records with it: **no headers, no build, nothing compiled in**
- **Stable 64-bit identities** — schema, layout, call and program hashes, independent of compiler and library spelling



The usual answers

Raw data

	Who writes the schema	A flat struct costs	A reader without your headers has
IDL + generator (Protobuf, FlatBuffers, Cap'n Proto, SBE)	an IDL file, a build step	encode, or a vtable walk	the schema file — or nothing
Macros / intrusive (cereal, Boost.Serialization, bitsery)	a macro or a `serialize()` member	a call per field	nothing
Text (JSON, CSV)	nobody	parse and print	the text itself — slow, large
`memcpy` **of a POD**	nobody	one copy	a wiki page, and hope

- The hole: plain C++ types, no IDL, no macros, `memcpy` speed — **and** a schema that travels with the data

Two questions

Raw data

- **How do we access the raw binary data from another program?**
  → the data carries ids; a **dictionary** maps id → schema
- **How do we hold a dictionary of all message types?**
  - → reflection builds each entry at compile time — reflection produces **data** (a schema), not only code
  - → **static initialisation** collects the entries before `main()`, or at `dlopen`
- The data outlives the program: another process, another build, another machine, next year — a reader that may have **no headers** and may not be C++



Thirty seconds: write

Ease of use

```cpp
#include <es/serreg/serreg.hpp>
using namespace es::serreg;
using namespace es::serreg::literals;
namespace corpus {
enum class Side : std::uint8_t { buy = 1, sell = 2 };
struct Order { std::uint64_t id; std::int64_t px; std::uint32_t qty; Side side; char sym[8]; };
}

int main() {
  capture_file file("orders.ser");                    // header + dictionary written now
  if (!file.is_open()) return 1;
  corpus::Order o{42, 1050000, 100, corpus::Side::buy, "AAPL"};
  if (!serialize(file, "new order"_, o)) return 1;    // one frame: 16-byte header + 32 bytes
  return file.close() ? 0 : 1;
}
```

- No registration call, no macro, no generated code
- `"new order"_` costs 0 bytes on the wire
- Live, GCC 16 `-freflection`: godbolt.org/z/PzGTv95xv — the program, then its file taken apart byte by byte

Thirty seconds: read, with nothing compiled in

Ease of use

```
$ srdump orders.ser
dictionary  program_hash 0xd6bdb39e65bcecb7  (3 types, 8 literals, 9 calls)
enum corpus::Side : u8  schema 0x074e267cca5188e1  size 1  align 1
  buy = 1
  sell = 2
struct corpus::Order  schema 0x367f8cd8c6441dd2  size 32  align 8  flat
  id: u64 @0 [8,8]
  px: i64 @8 [8,8]
  qty: u32 @16 [4,4]
  side: enum<corpus::Side> @20 [1,1]
  sym: array<char,8> @21 [8,1]
  /* size 32: payload 29 (90.6%), padding 3 (9.4%) = holes 0/0 B, tail 3 B */
...
frame 1  call 0xbcf8e25bde66fdb2  (2 records)
  [0] "new order"
  [1] corpus::Order { id: 42  px: 1050000  qty: 100  side: corpus::Side::buy (1)  sym: "AAPL\0\0\0\0" }
```

- `srdump` was compiled without `corpus::Order`; everything comes from the file — including the `pahole`-style layout accounting
- `--json`, `--verify`, `--elf ./app`, `--diff old.rri ./new-app`, `--wireshark serreg.lua`

Literals: identity in the type

Ease of use

```
serialize(out, "order rejected"_lit, o);                    // UDL
serialize(out, "order rejected"_, o);                       // short UDL
serialize<"order rejected">(out, o);                        // template argument
constexpr lit<"order rejected"> k_rejected;  serialize(out, k_rejected, o);
```

- `operator""_` is a literal-operator **template** taking a `fixed_string` NTTP: the text becomes part of the type
- Only an 8-byte id (XXH3-64 of the text) exists, and in a compact frame not even that: the call hash names it
- A runtime `std::string_view` cannot be a literal: registration happens before any run-time value exists

Where does the dictionary come from?

Ease of use

- Three places, one blob format (the **ENTRY**):
  1. **in the capture**: opening dictionary + delta frames
  2. **in the binary**: ELF section `es_serreg_dict` → `srdump --elf ./app`, even if the app never ran
  3. **from a running process**: `ES_SERREG_DUMP_REGISTRY_FD=3`, a signal, or `registry::instance().export_image()`
- All three are generated by reflection at compile time and collected before `main()`



C++26 reflection in one slide (P2996)

Reflection: from a type to a schema

```
#include <meta>
template<class T> void show(const T& v) {
  template for (constexpr std::meta::info m : std::define_static_array(
                  std::meta::nonstatic_data_members_of(^^T, std::meta::access_context::unchecked())))
    std::println("{} @{} = {}", std::meta::identifier_of(m), std::meta::offset_of(m).bytes, v.[:m:]);
}
```

- `^^T`: a reflection, a value of type `std::meta::info`
- `[: m :]`: a splice, back from the value to code
- `template for`: an expansion statement, unrolled per member
- GCC 16: `-std=c++26 -freflection`; clang-p2996: `-freflection-latest -stdlib=libc++`

The metafunctions this library needs

Need	Metafunction
members, bases	`nonstatic_data_members_of`, `bases_of`
names	`identifier_of`, `has_identifier`, `parent_of`, `template_arguments_of`
types	`type_of`, `dealias`, `remove_cv`
layout	`offset_of(m).bytes`, `size_of`, `alignment_of`, `is_bit_field`
enums	`enumerators_of` (name + value)
metadata	`annotations_of`, `extract<A>`
to run time	`define_static_string`, `define_static_array`

- One header uses reflection (`reflect.hpp`); everything else is plain C++

The canonical name

```
template<class T> consteval std::string canonical_name() {
  constexpr std::meta::info r = std::meta::dealias(^^T);
  if constexpr (std::meta::has_template_arguments(r)) {
    std::string s = scope_of<std::meta::parent_of(r)>()
                  + std::string(std::meta::identifier_of(std::meta::template_of(r))) + '<';
    /* template for over template_arguments_of(r): a type → canonical_name<[:a:]>(), a value → "3" */
    return s + '>';
  }
  return scope_of<std::meta::parent_of(r)>() + std::string(std::meta::identifier_of(r));
}
```

- **1.** Walk `parent_of` to the global namespace: `corpus::Order` — never `display_string_of` / `__PRETTY_FUNCTION__`, their spelling is the compiler's
- **2.** Built-ins from a table, by width: `i64`, not `long`
- **3.** No name without linkage: an unnamed-namespace or local type needs `[[=name("…")]]`
- **4.** Allocators, default comparators and hashers of vocabulary containers are not in the name; a non-default comparator is

The canonical string: a schema as text

```
schema:  struct<corpus::Order>[32,8]{id:u64@0[8,8];px:i64@8[8,8];qty:u32@16[4,4];
                                     side:enum<corpus::Side,u8>@20[1,1];sym:array<char,8>@21[8,1];}
layout:  struct[32,8]{u64@0[8,8];i64@8[8,8];u32@16[4,4];enum<u8>@20[1,1];array<char,8>@21[8,1];}
enum:    enum<corpus::Side,u8>[1,1]{buy=1;sell=2;}
dynamic: dyn<corpus::DynMsg>[20,4]{id:u32@0[4,4];name:string@4[8,4];xs:vector<i16>@12[8,4];}
```

- Built by a `consteval` function: `std::string` concatenation at compile time
- `[size,align]` per type and per member; `@offset` per member; nested types by their hash
- `since` / `deprecated` annotations appear as `;sN;dN`

Building it: index sequences over `members<T>()`

```
template<class T, std::size_t I> consteval void append_member(std::string& s, bool schema) {
  if constexpr (!member_skipped<T, I>()) {                // [[=skip]]
    using M = part_t<T, I>;                               // [: type_of(members<T>()[I]) :]
    if (schema) { s += part_name<T, I>(); s += ':'; }     // identifier_of(members<T>()[I])
    append_kindspec<M>(s, schema);                        // u64, enum<…>, array<…>, struct<0x…>
    s += '@'; append_uint(s, member_offset<T, I>());
    append_dims(s, image_size<M>(), image_align<M>(), /* since, deprecated, */ schema);  s += ';';
  }
}
template<class T, std::size_t... B, std::size_t... M>  // make_index_sequence<base_count<T>()>,
consteval std::string struct_string(bool schema, std::index_sequence<B...>, std::index_sequence<M...>) {
  std::string s = /* "struct<name>" + dims + '{' */;    // make_index_sequence<part_count<T>()>
  (append_base<T, B>(s, schema), ...);  (append_member<T, M>(s, schema), ...);
  return s += '}';
}
```

- `members<T>()[I]` in a constant expression; `make_index_sequence` + a fold unroll the loop — no `template for` here
- The same index walk builds the fields, the padding table, the image layout, encode/decode and registration; `schema == true` keeps the names, `false` drops them

Schema hash and layout hash

```
template<class T> consteval std::uint64_t schema_hash_of()
{ return fingerprint64(canonical_string<T>(true)); }      // names, field types, offsets, since/deprecated
template<class T> consteval std::uint64_t layout_hash_of()
{ return fingerprint64(canonical_string<T>(false)); }     // the same walk without the names
```

- The **call signature** is the same walk over an argument list — `"new order"_, Order` — and its hash is the frame's id (16-byte header, the literal costs 0 bytes)
- `fingerprint64` = XXH3-64 as `consteval` C++; **schema hash** identifies the type on the wire, **layout hash** says two types are bit-compatible whatever their names; same schema hash + different layout = a **conflict**, reported at load

Edit to `Order`	Schema	Layout	Typed reader	Bytes
rename `px` → `price`	moves	same	refuses, unless projecting	same
add a member	moves	moves	refuses	different
`std::pmr::string` for `std::string`	same	same	accepts	same

Padding and `bool`: deterministic bytes

```
struct corpus::Order (size 32):  id@0[8] px@8[8] qty@16[4] side@20[1] sym@21[8]  → bytes 29..31 are padding
```

```
template<class T> ES_SERREG_HIDDEN inline constexpr auto gaps_v  = make_gaps<T>();   // {29, 3}
template<class T> ES_SERREG_HIDDEN inline constexpr auto bools_v = make_bools<T>();  // bool offsets
```

- Reflection collects the covered intervals of every leaf scalar; the complement is padding
- `serialize`: `memcpy`, then zero the gaps and normalise `bool` bytes to 0/1, all from constant tables
- Why: equal values give equal bytes, and no stack garbage leaks into a file

Annotations: metadata without macros (P3394)

```
struct Order : Hdr {
  std::uint64_t id; std::int64_t px; Side side; char sym[8];
  [[=since(2)]] std::uint32_t venue;          // in the schema string and the dictionary
  [[=skip]]     std::uint32_t cache;          // never serialised
};
namespace { struct [[=es::serreg::name("app::Quote")]] Quote { /* ... */ }; }
```

```
template<class A> consteval std::uint16_t annotation_value(std::meta::info m) {
  for (std::meta::info a : std::meta::annotations_of(m))
    if (std::meta::remove_cv(std::meta::type_of(a)) == ^^A) return std::meta::extract<A>(a).v;
  return 0;
}
```

- An annotation is a **value** of a structural type, attached to a declaration
- `since` / `deprecated` / `skip` / `name` today; validation hints (`range`, `max_size`) proposed

ENTRY blobs: `consteval` bytes

```
template<fixed_string S> consteval std::vector<std::byte> literal_blob_bytes() {
  cbuf b;  b.begin_entry(fmt::tag_literal);
  b.put(fingerprint64(S.view()));  b.put(std::uint32_t(S.size()));  b.put(std::uint32_t{0});
  for (char c : S.view()) b.v.push_back(std::byte(c));
  b.end_entry();  return b.v;
}
template<class P> consteval auto to_blob_array() {                  // vector → array:
  std::array<std::byte, P::bytes().size()> out{};                   //   call once for the size,
  const auto v = P::bytes();                                        //   once for the bytes
  for (std::size_t i = 0; i < out.size(); ++i) out[i] = v[i];
  return out;
}
```

- Every type, literal and call becomes one **ENTRY**: `u32 len | u8 tag | u8 ver | u16 rsv | body`
- A compile-time `std::vector` cannot outlive constant evaluation → copied into a sized `std::array`
- The result is `constexpr` data in `.rodata`: the dictionary is built by the compiler

Reflection: from a type to a schema, in three lines

- `members<T>()`, `identifier_of`, `offset_of`, `size_of`, `enumerators_of`, annotations: the compiler always knew; P2996 lets the program ask
- The canonical string gives every type one name, one schema hash and one layout hash — deterministic bytes, padding and `bool` included — at compile time, into `consteval` ENTRY blobs
- Each type now has a blob with its complete description; missing: every blob in one place before anyone writes a frame



The registration

Registration before main()

- The blobs exist, scattered over every translation unit and every shared object
- The dictionary must list **every type the program can write**, and must be ready:
  - before the first frame (the file header contains it)
  - in a process that never calls `serialize` for some type yet
- Nobody should have to write `register<Order>()`
- C++ has no "list all instantiations" query, and reflection cannot enumerate a program



The trick: a static member of a class template

Registration before main()

```
template<class T> struct registrar {
  static reg_node    node;     // constant-initialised: descriptor, blob, hash, name
  static const guard g;        // dynamic initialisation: guard() pushes &node
};
template<class T> reg_node    registrar<T>::node{&descriptor_v<T>, descriptor_v<T>.blob,
                                                 descriptor_v<T>.schema_hash, descriptor_v<T>.name, /*…*/};
template<class T> const guard registrar<T>::g{&registrar<T>::node};

struct guard {
  explicit guard(reg_node* n) noexcept : n_(n) { registry::instance().push(n); }
  ~guard()                                     { registry::instance().on_guard_destroyed(n_); }
  reg_node* n_;
};
```

- `registrar<T>::g` exists only if something **odr-uses** it ([temp.inst])
- Its constructor runs during the **static initialisation of the image** that instantiated it
- Stand-alone, 90 lines, no library: `examples/odr-use-premain/main.cpp` — two lambdas registered before `main()` by odr-use, a third not, because its template was never instantiated — live: godbolt.org/z/f4jx7j6Pa

`serialize()` odr-uses the guards

Registration before main()

```
template<class... Sig, sink S, class... Ts>
result<std::size_t> serialize_frame(serialize_options o, S& out, const Ts&... vs) {
  (detail::touch_arg<Sig>(), ...);                          // every argument type, recursively
  static_cast<void>(&detail::registrar_call<Sig...>::g);    // the call signature itself
  /* … claim, header, memcpy, scrub, commit … */
}
```

- Taking the address is an odr-use → the guard is instantiated → it runs before `main()`
- The cost at run time: **zero**; the expressions fold away at `-O1`
- Writing `serialize(out, x)` anywhere, even in code that never runs, registers `x`'s type

`touch<T>`: every type inside, too

Registration before main()

```
template<class T> void touch() {
  using U = mapped_t<T>;
  if constexpr (is_scalar_v<U>) {}
  else if constexpr (array_traits<U>::is_array) touch<typename array_traits<U>::elem>();
  else if constexpr (std::is_enum_v<U>)         static_cast<void>(&registrar<U>::g);
  else if constexpr (is_dyn_member_v<U>)        /* elements, alternatives, key + value */;
  else {
    static_cast<void>(&registrar<U>::g);
    touch_components<U>(std::make_index_sequence<bases<U>().size()>{},
                        std::make_index_sequence<members<U>().size()>{});   // bases + members
  }
}
```

- Recursion over bases, members, array elements, vector/optional elements and variant alternatives
- The dictionary is **complete by construction**, and only as large as what the program writes
- `register_types<Ts...>()` exists for types the program only reads

The registry: one per process

Registration before main()

```
// libserreg.so
constinit registry registry_state::inst{};                    // constexpr constructor: no SIOF
state& G() noexcept {                                         // created on first push,
  static state* const s = new state{};  return *s;            // never destroyed
}
```

- Guards run in **unordered** dynamic initialisation (template statics) → the order is not defined
- The registry must work whatever runs first:
  - the object is `constinit`; its state is created on first use and **never** destroyed
  - entries are keyed by hash; duplicates from several TUs collapse into one
  - `program_hash` is order-independent
- Storage lives in the **shared** `libserreg`: a static copy inside a plugin would be a second registry

Shared objects: `dlopen` and `dlclose`

Registration before main()

```
dlopen(path)  → the plugin's guards run inside dlopen (loader lock held)
                push(): new entry | alias | revive a retained entry | conflict
                no index built, no callback → the next read publishes once
dlclose(h)    → guard destructors: drop the reference; refs == 0 → the entry is RETAINED
                no abort, no allocation, no callback
reload        → same hash, same layout → the retained entry revives
```

- `dladdr` gives each node its **image**; references are counted per image
- Retained entries keep old frames decodable after their plugin is gone
- A rebuilt plugin with a changed type adds a new entry next to the old one

Visibility and STB_GNU_UNIQUE

Registration before main()

```
#define ES_SERREG_HIDDEN [[gnu::visibility("hidden")]]
template<class T> struct ES_SERREG_HIDDEN registrar { /* … */ };
```

- Without care, GCC emits template statics as STB_GNU_UNIQUE
  - glibc then marks the DSO DF_1_NODELETE: dlclose silently does nothing
  - a reloaded, rebuilt plugin never runs its guards again
- So: -fno-gnu-unique is a PUBLIC flag of the CMake target, and the registrars are hidden
- Each image has its own copy of every guard → the registry counts them per image

The dictionary inside the binary

Registration before main()

```
#define ES_SERREG_DICT_SECTION [[gnu::section("es_serreg_dict"), gnu::used, gnu::retain]]
template<class T>
ES_SERREG_DICT_SECTION alignas(8) static constexpr auto type_blob_copy_v = type_blob_bytes_v<T>;

template<class T> consteval void embed_type_blob() { static_cast<void>(&type_blob_copy_v<T>); }
```

```
$ srdump --elf ./trading-app            # never executed
$ srdump --diff old.rri ./trading-app   # identical | names-only | metadata-only | breaking | …
```

- `gnu::retain` keeps the copies under `--gc-sections` (binutils ≥ 2.36); `gnu::used` keeps them unreferenced
- **TU-local** (`static`) on purpose: GCC puts every COMDAT variable naming a section into **one** section group, which breaks multi-TU links
- Duplicates (about 250 B per type per TU) are harmless: readers key by hash

Registration before main()

Lock-free reads: the published index

- Mutations (`push` at load, guard destruction at unload) take a mutex and mark the registry **stale**
- The next reader publishes one **immutable, sorted index**; readers acquire-load a pointer and binary-search
  - one `std::flat_map<hash, position>` per category (type, literal, call, scope)
- Indexes are never freed before exit → every pointer handed out stays valid
- One publish per burst: 10 000 types registered before `main()` → one index, built when `main()` first asks

```
std::println("{} entries", registry::instance().size());   // slide 5's main(): "4 entries"
const type_desc* d = registry::instance().find_type(0x367f8cd8c6441dd2);  // "corpus::Order"
```

- The 4 entries of slide 5's program: `corpus::Order`, `corpus::Side`, the literal `"new order"`, the call

Frame 1, byte by byte

The reader's side

```
30 00 00 00 | 00 00 | 02 00 | b2 fd 66 de 5b e2 f8 bc     frame_len 48 · flags 0 · 2 records · call hash
                                                          [0] "new order": 0 bytes (in the call hash)
2a 00 00 00 00 00 00 00                                   id   = 42
90 05 10 00 00 00 00 00                                   px   = 1050000
64 00 00 00                                               qty  = 100
01                                                        side = buy
41 41 50 4c 00 00 00 00                                   sym  = "AAPL"
00 00 00                                                  padding, zeroed (gaps_v)
```

- Header: `u32 frame_len | u16 flags | u16 n_records | u64 call_hash`
- Compact records are the object's own bytes; sizes come from the dictionary
- Opt-in: full records (self-describing per record), a CRC32C trailer per frame

The file: dictionary first, then frames

The reader's side

```
FILE   "ESSR" | u16 1, u16 7 | flags | hash_algo | program_hash | dict_offset | dict_len | abi_tag   (48 B)
RRI    "ESRG" | … | n_entries | program_hash | abi_tag (40 B) | ENTRY × n      ← export_image()
CHAIN  u64 next_dictionary | u64 first_checkpoint
FRAME  writer-info · data · data · [dictionary delta] · data · [checkpoint] · …
```

- The opening dictionary is `registry::export_image()`: everything registered before `main()`
- A plugin loaded later → a **delta** dictionary frame, forward-linked; nothing is rewritten at close
- The ABI tag records the writer's ABI (`ptr8 long8 wchar4 char-signed …`)

Names: one type, one name, everywhere

- Spelled by the compiler, libc++ names `std::monostate` as `std::__1::monostate` → a different hash for the same type
- Fix: a **std vocabulary** with frozen names (`std::string`, `std::vector<E>`, `std::map<K, V>` …) through `type_properties<T>`, independent of the standard library
- A class in an unnamed namespace or a local class has no unique name → `static_assert`, or name it:

```
namespace { struct [[=es::serreg::name("app::Quote")]] Quote { /* ... */ }; }
template<> struct es::serreg::type_properties<third_party::Px> {
  static consteval std::string name() { return "tp::Px"; }
};
```

Shared objects, lifetimes and scale

- The registry keeps what it hands out: entries, descriptors, names, blobs and indexes stay valid after `dlclose`
- It cannot keep plugin **code** alive: callbacks, handlers and visitors from a plugin must go before `dlclose`
- A **static** `libserreg` inside a plugin = a second, private registry (warned at load)
- Scale: 10 000 types in ≤ 10 ms of start-up; main-program entries are views (no copy)
  - section size: about 250 B per type per TU; nothing on the hot path



Toolchain status, and schema evolution in one line

- Exact match: `deserialize(frame, …)` refuses another version of `Order` by hash; a dictionary reader reads anything
- **Level 1** for flat types: `deserialize(projection_context{&dict}, frame, out)` projects by member name — widening, defaults, explicit renames; dynamic members next
- Toolchain status:
  - GCC 16 `-freflection`: the reference
  - clang-p2996 + libc++: same golden bytes
  - libstdc++ `<meta>` with clang: not possible today (GCC intrinsics)
  - EDG: rejects `template for`



Close

Summary: the data carries its own description

- **Reflection** turns each type into data: canonical name, schema string, hashes, padding table, ENTRY blob
- **Static initialisation** collects the data before `main()` (or at `dlopen`): no macros, nothing to forget
- **One blob format, three carriers**: the capture, the binary's ELF section, the running process
- **The reader needs nothing compiled in**: it reads the dictionary from the capture, the ELF section or the process, then the frames
- **Any reader**: typed C++ at `memcpy` speed, a generic C++ tool, or Python
- Code, slides and the book: `github.com/erez-strauss/serreg`



Try it: the Compiler Explorer examples

Close

	One translation unit, GCC 16 `-std=c++26 -freflection` (clang-p2996 + libc++ too)
odr-use	slide 20's stand-alone program: two lambdas registered before `main()` by odr-use, a third not — godbolt.org/z/f4jx7j6Pa
0	slide 9's program: `^^T`, `[: m :]`, `template for` — no library — godbolt.org/z/raaYjnzne
1, 2	canonical names and hashes of the corpus against the golden values; ENTRY blobs and images byte for byte
3, 4	the frames of `orders.ser`, typed; dynamic members and `dispatch`
5	frames through an `es::ipc` SPSC ring between two threads
6	slide 5's `capture_file`: the file written, then taken apart from its own dictionary — godbolt.org/z/PzGTv95xv
7	the logger: lines as frames, 0 bytes for level/file/line/format, arguments below the floor not evaluated — godbolt.org/z/KrWPcj5hK

References and links — and your questions

Close

Reference	Link
This library, these slides, the book (MIT)	`github.com/erez-strauss/serreg`
Try it live: slide 5's program on Compiler Explorer	`godbolt.org/z/PzGTv95xv`
P2996 *Reflection for C++26*	`wg21.link/p2996`
P3394 *Annotations for reflection*	`wg21.link/p3394`
P1306 *Expansion statements* (`template for`)	`wg21.link/p1306`
P3491 `define_static_string` / `_array` / `_object`	`wg21.link/p3491`
GCC 16 reflection status	`gcc.gnu.org/projects/cxx-status.html`
clang-p2996 (Bloomberg)	`github.com/bloomberg/clang-p2996`
XXH3-64, the hash of every id	`github.com/Cyan4973/xxHash`