Type erasure – mal anders

(Type erasure, take two)

Tobias Hoffmann

C++ User Gruppe Aachen, 2026-06-10

Agenda

Type Erasure 101

- What? Why? Alternatives?
- `void *`, `dynamic_cast`
- `std::any` (“type safe void *”)

Another Idea

- Motivation: Tree Visitors
- `std::any` revisited
- Lock-and-Key Principle
- Example, Implementation idea
- `Opaque`, `unique_anyptr`, `shared_anyptr`

What is Type Erasure?

- “Forget” the original *compile-time* type
- Only for pointers!
- Why not (e.g.) `int *`?
  - Pointer arithmetic: `pInt32++` → `addr += 4`
  - Alignment
  - Just use `char *`!
  - Or even better: `void *`
- How: `static_cast<void *>(p)` / `<T *>`
- Possibly: `reinterpret_cast<intptr_t>(p)`

Features of void *

- No dereferencing (`*pVoid` → compiler error)
- Consider `pVoid += N`:
- `pVoid += N * sizeof(*pVoid)`
- Does not work (as per Standard):
  - MSVC: `error C2036: 'void *': unknown size`
  - CLANG: `error: arithmetic on a pointer to void`
- Extensions... (as if `sizeof(...) == 1`)
  - GCC: `warning: pointer of type 'void *' used in arithmetic [-Wpointer-arith]`

Why even use Type Erasure?

Examples:

- `someCallback(…, void *userData)`
- `std::map<int, …different types…>`
- `printf("%s: %d\n", "foo", 5)`
- `std::vformat(std::string_view fmt, std::format_args args)`
- `std::function<int()> fn = []() { … }`

Common alternatives

- `std::variant<std::monostate,Foo,int>` (a.k.a. “Tagged Union”)
- `template <class T> …`
- Base + Derived (runtime polymorphism)
  - most extreme: everything derives from `Object`, `IUnknown` (outside of C++ type system), …
  - even when original class hierarchies cannot be altered, composition is usually possible

Problems with void *

- *Zero* type safety
- Destructor?
- Strict aliasing violations possible
  - Different pointer types expected to point to different memory locations
  - **Only** `char *`, `signed/unsigned char *` (and `std::byte` in C++17)) are safe
- Code intent unclear to human readers
- Idea: Use `void *` as “Base” …

When to (not) use dynamic_cast?

```
//void *p = ...; // not a complete class type
                 // (and not polymorphic, either)
Base *p = ...; // = nullptr;

if (Foo *pf = dynamic_cast<Foo *>(p)) {
  // ...
}

// or: Foo &f = dynamic_cast<Foo &>(*p);  // → std::bad_cast
```

- Pointer value is *valid* (or `nullptr`)
- RTTI is not disabled (`-fno-rtti`)
  (except for cases already handled by `static_cast`)
- Object is of *polymorphic type*: `std::is_polymorphic`
  → has at least one virtual function (possibly via base)

STD::ANY (“TYPE SAFE VOID *”)

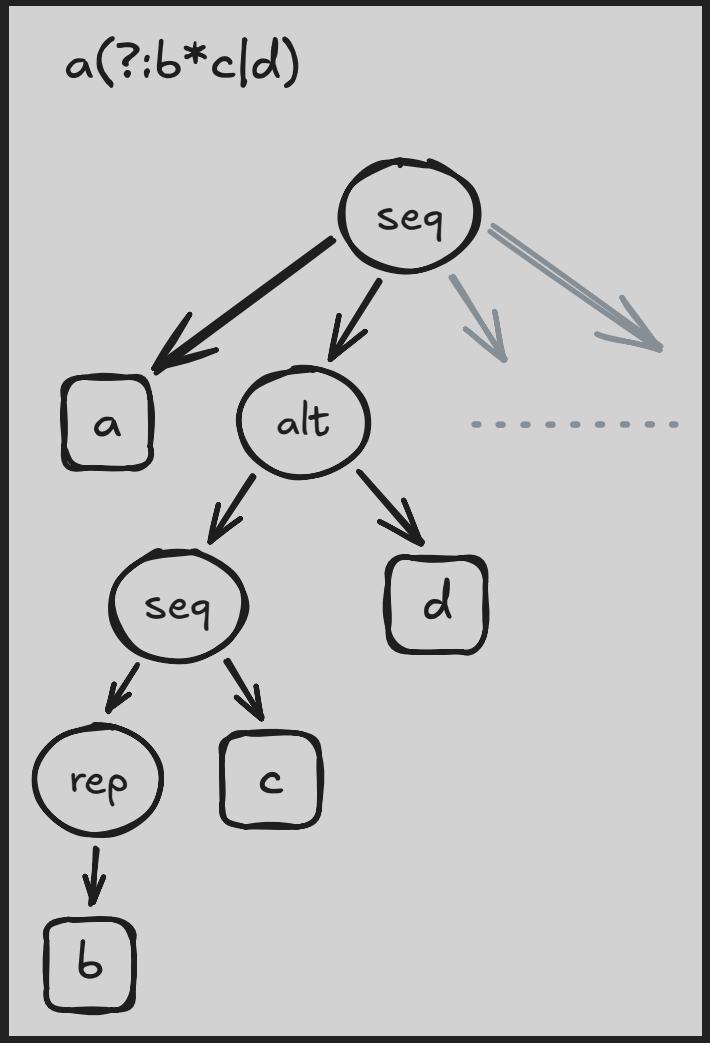
```
// std::any foo = User{"Alice", 30};  // temporary object
std::any foo = std::make_any<User>("Alice", 30);  // RVO (manc

if (std::string *foo = std::any_cast<std::string>(&foo)) {
  // ...
}

// or: std::string &bar = std::any_cast<std::string &>(foo);
                          // → std::bad_any_cast
```

- C++17 [with RTTI] (alternative: `boost::any`)
- Value semantics. Copyable. Memory-safe (dtor).
- `std::any ap = std::unique_ptr<…>(…)` (`unique_ptr` is move-only … → C++23)
- “Interface”: Clone/Copy, Destroy

Another way?

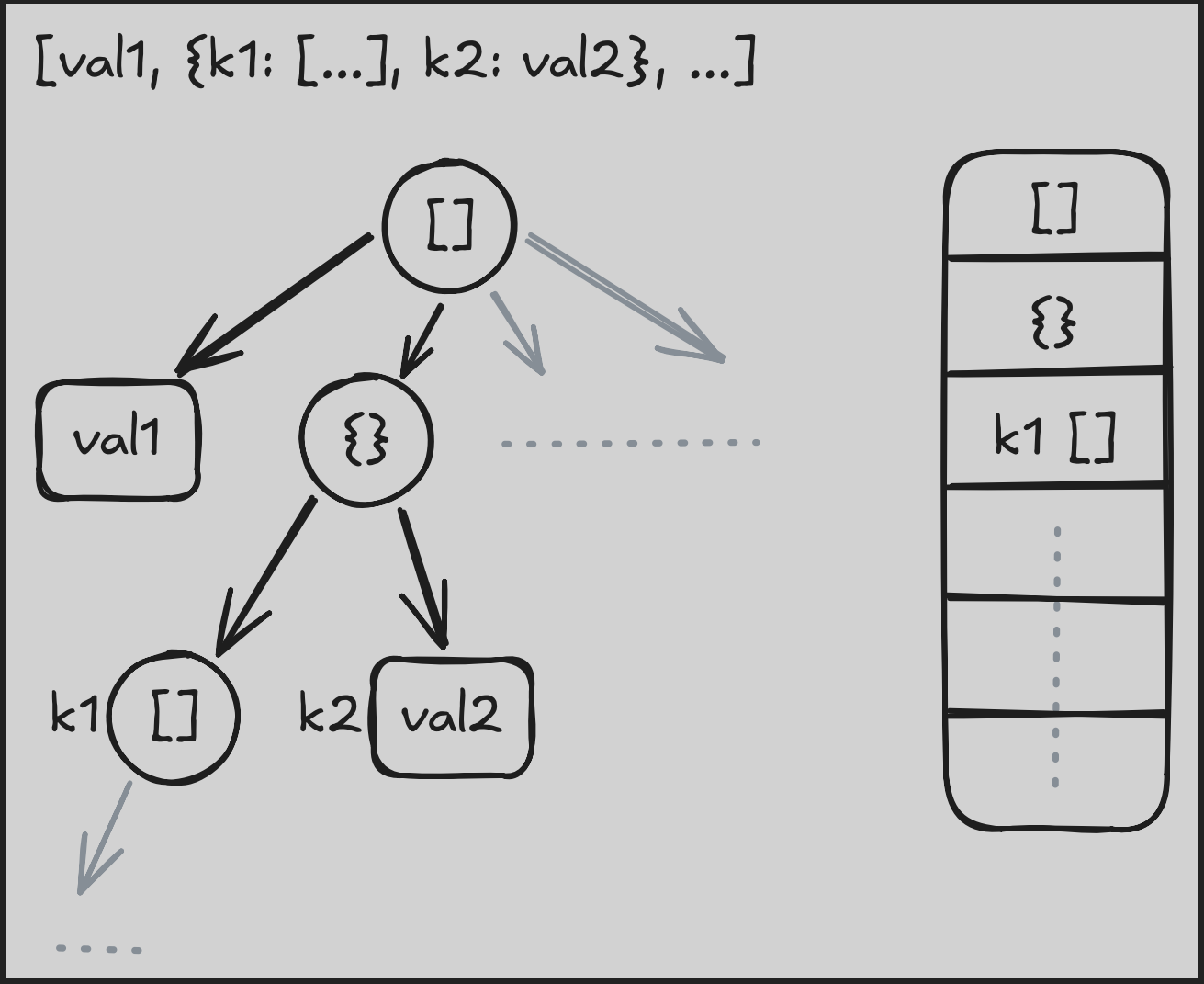
- Questions so far?
- Does `std::any` allow extracting Base for Derived?
- Interfaces, (Runtime) Duck typing, …
- `IUnknown::QueryInterface`, possibly dynamically (not in COM!)
- Data-oriented design (one pointer per type, separate arrays, …)
- Entity-Component-System (ECS)

Motivation: Tree Visitors



- Scenario: Traversing a complex Tree Structure, e.g. Abstract Syntax Tree (AST)
- Polymorphic node types
  → Visitor pattern
- Possibly implicit: during parse, only stack "exists"
- Idea: Build `map`, `vector`, …

JSON: The Stack

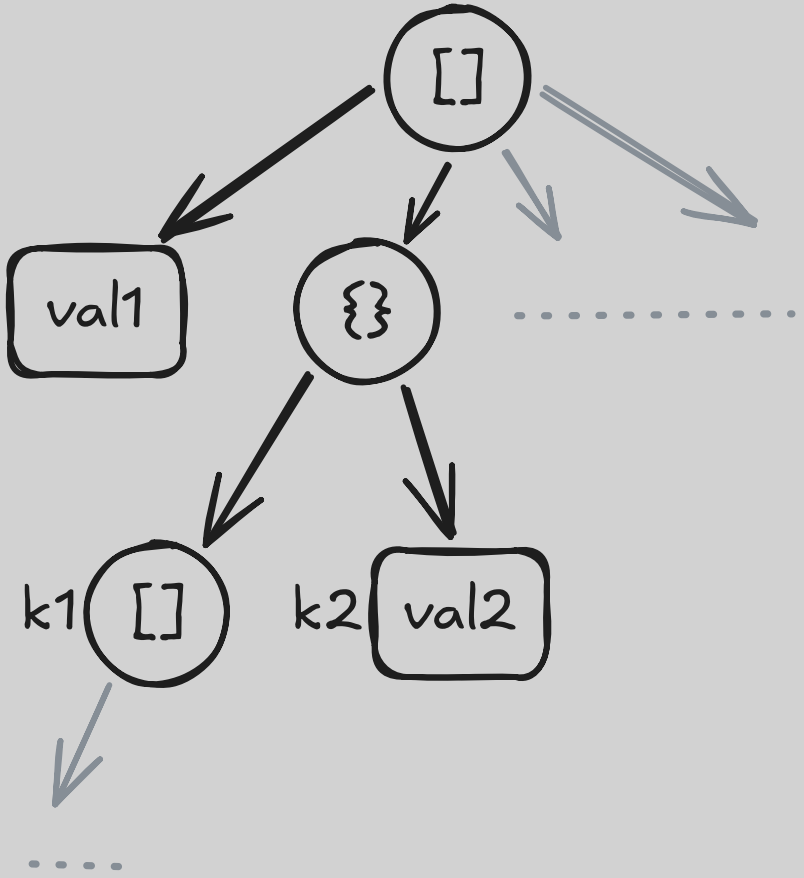


JSON: The Dilemma

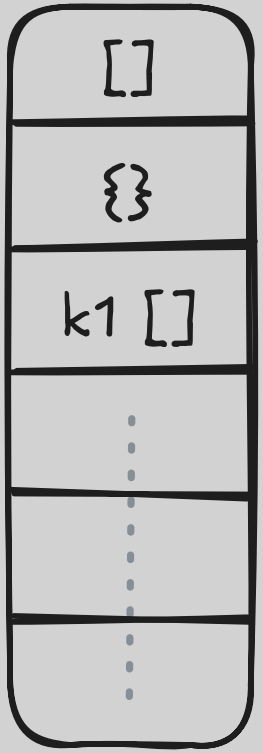
- Problem: Inner nodes are visited multiple times. Information on the way down also needed on the way back up.
- Solution A: Visitor maintains a parallel, manual stack “synced” with the traversal depth.
- What is the `value_type` of the (e.g.) `std::stack<T>`?

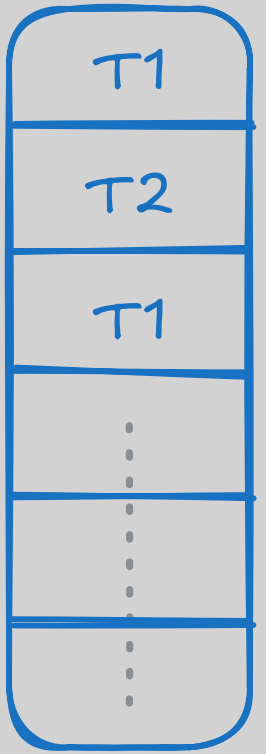
JSON: The second Stack

[val1, {k1: [...], k2: val2}, ...]



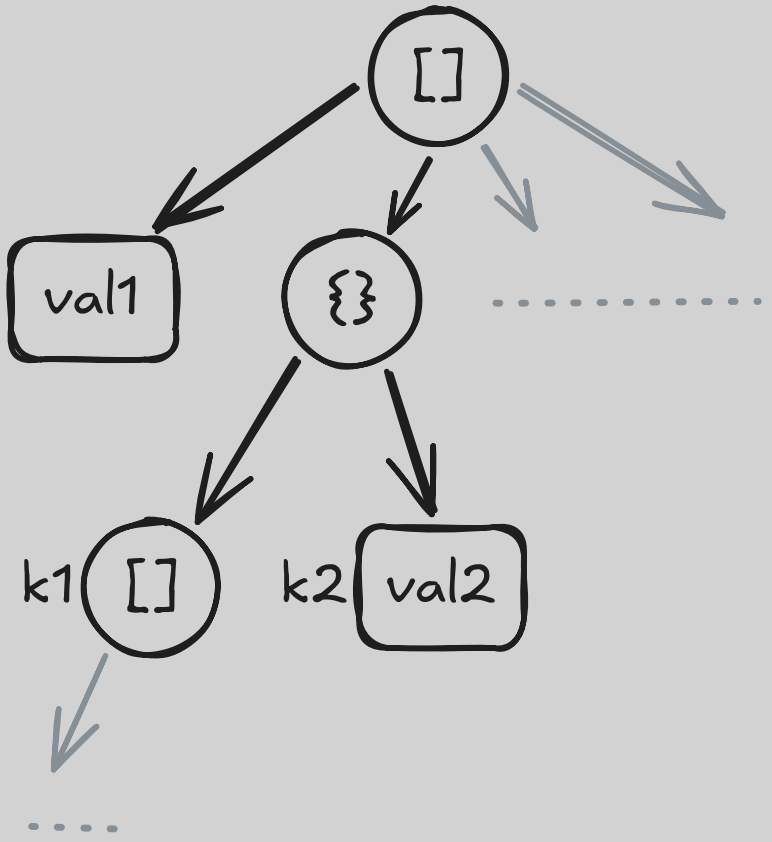
std::variant<T1,T2>

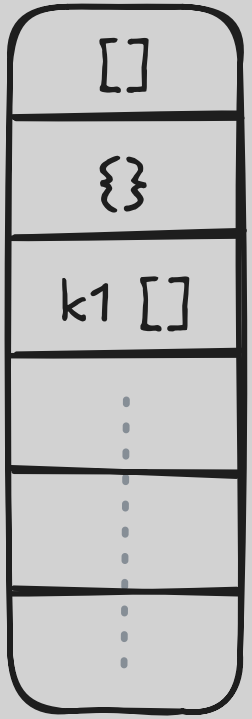




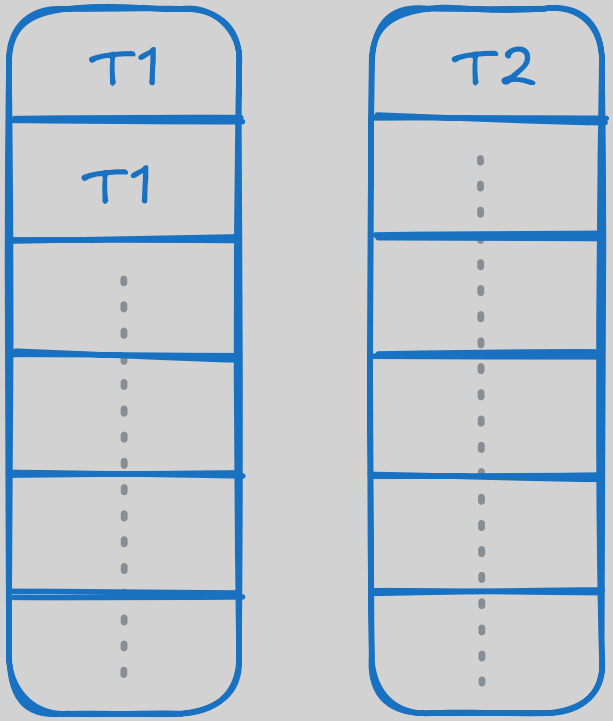
JSON: The third Stack...?

[val1, {k1: [...], k2: val2}, ...]



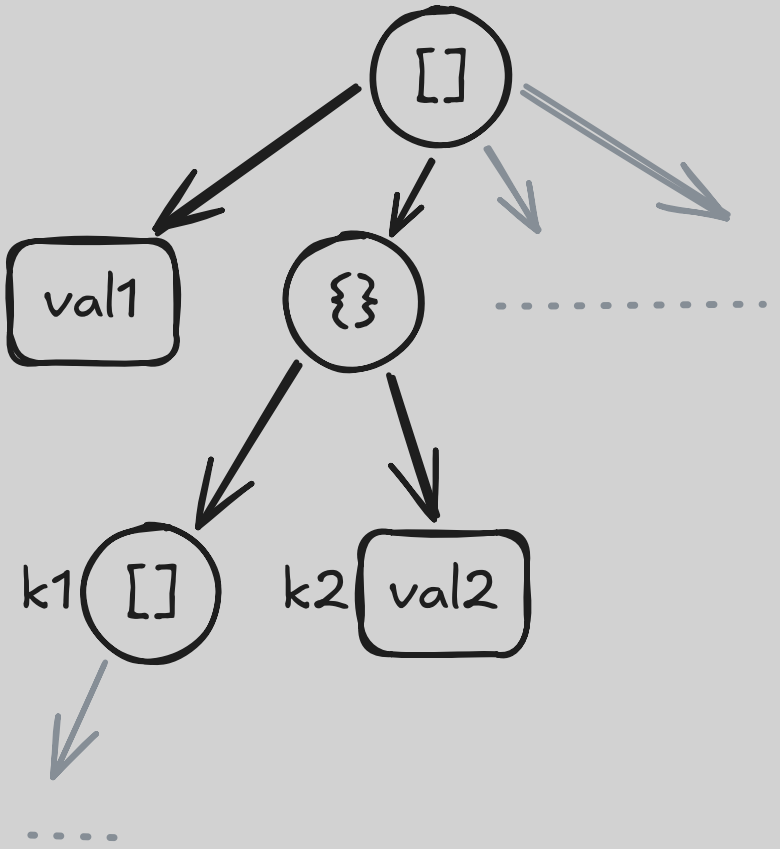


vector<T1> vector<T2>

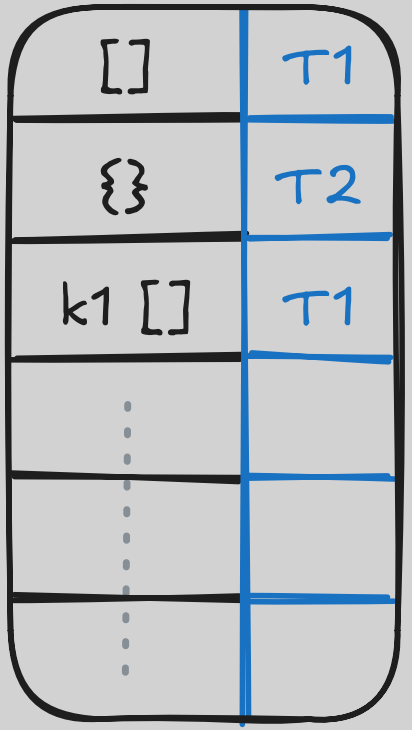


JSON: Unified Stack!

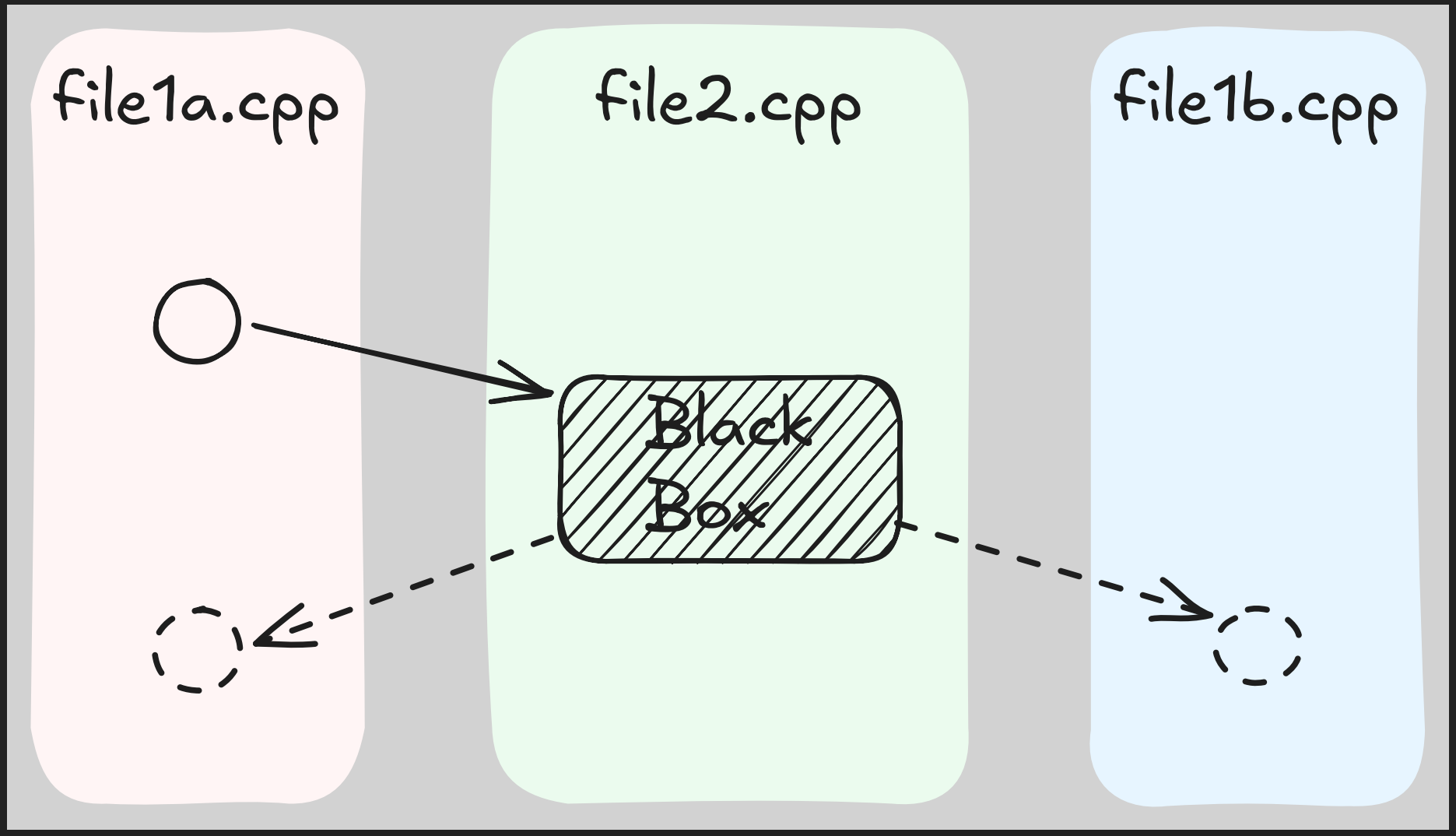
[val1, {k1: [...], k2: val2}, ...]



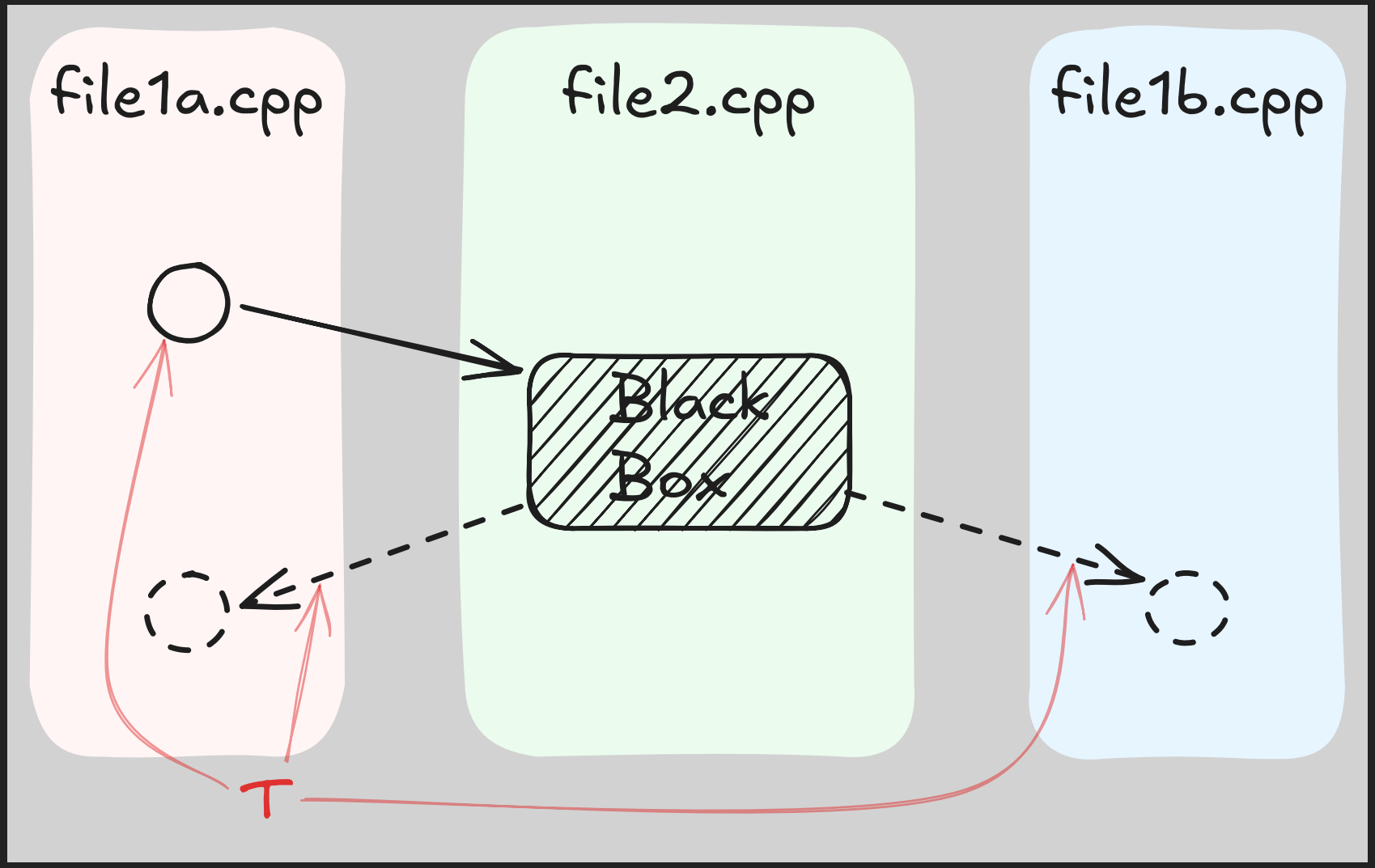
std::variant<T1,T2> ?
void * ? / std::any ?



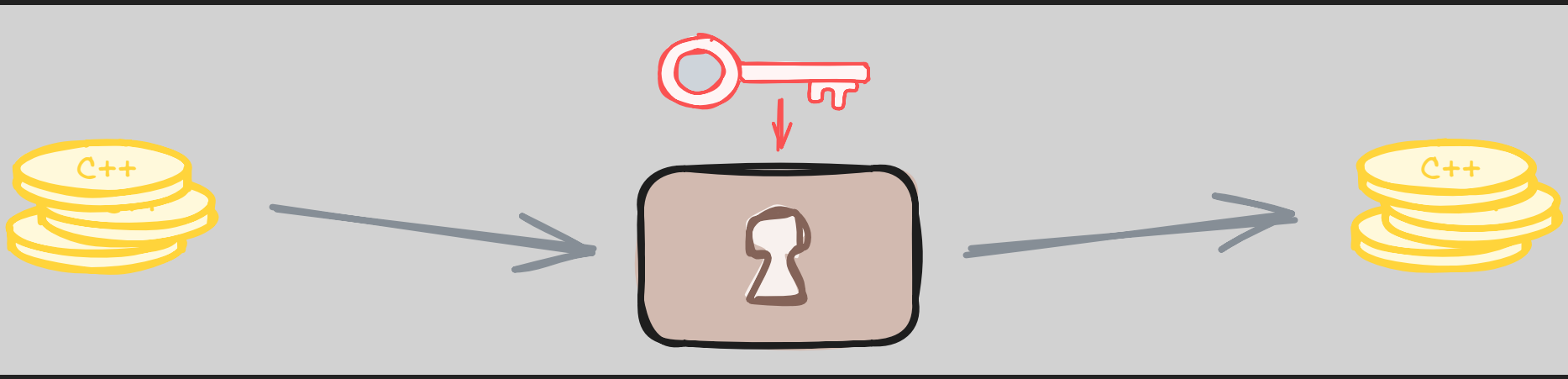
std::any revisited: The scene



std::any revisited: The key

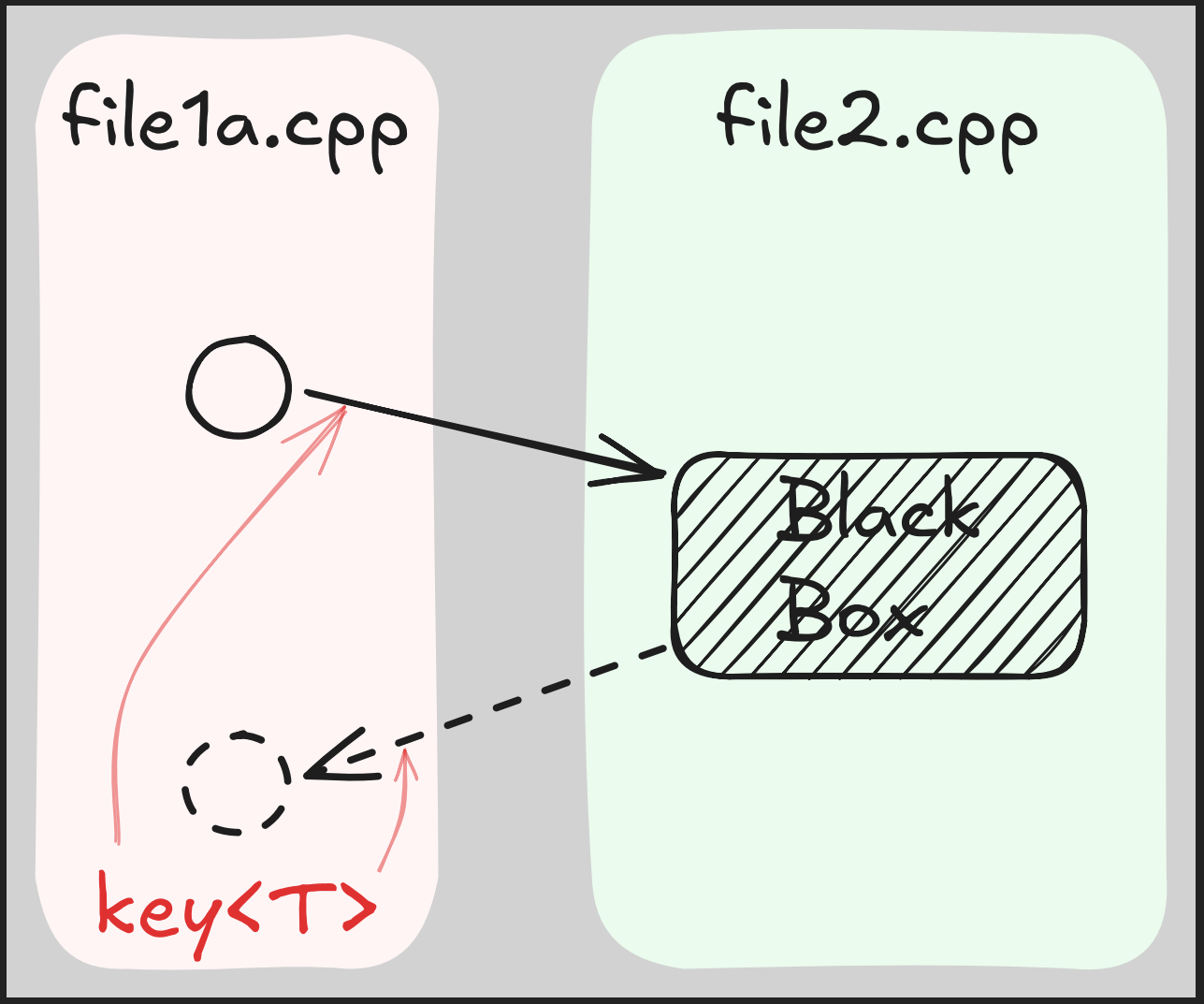


The Lock-and-Key Principle

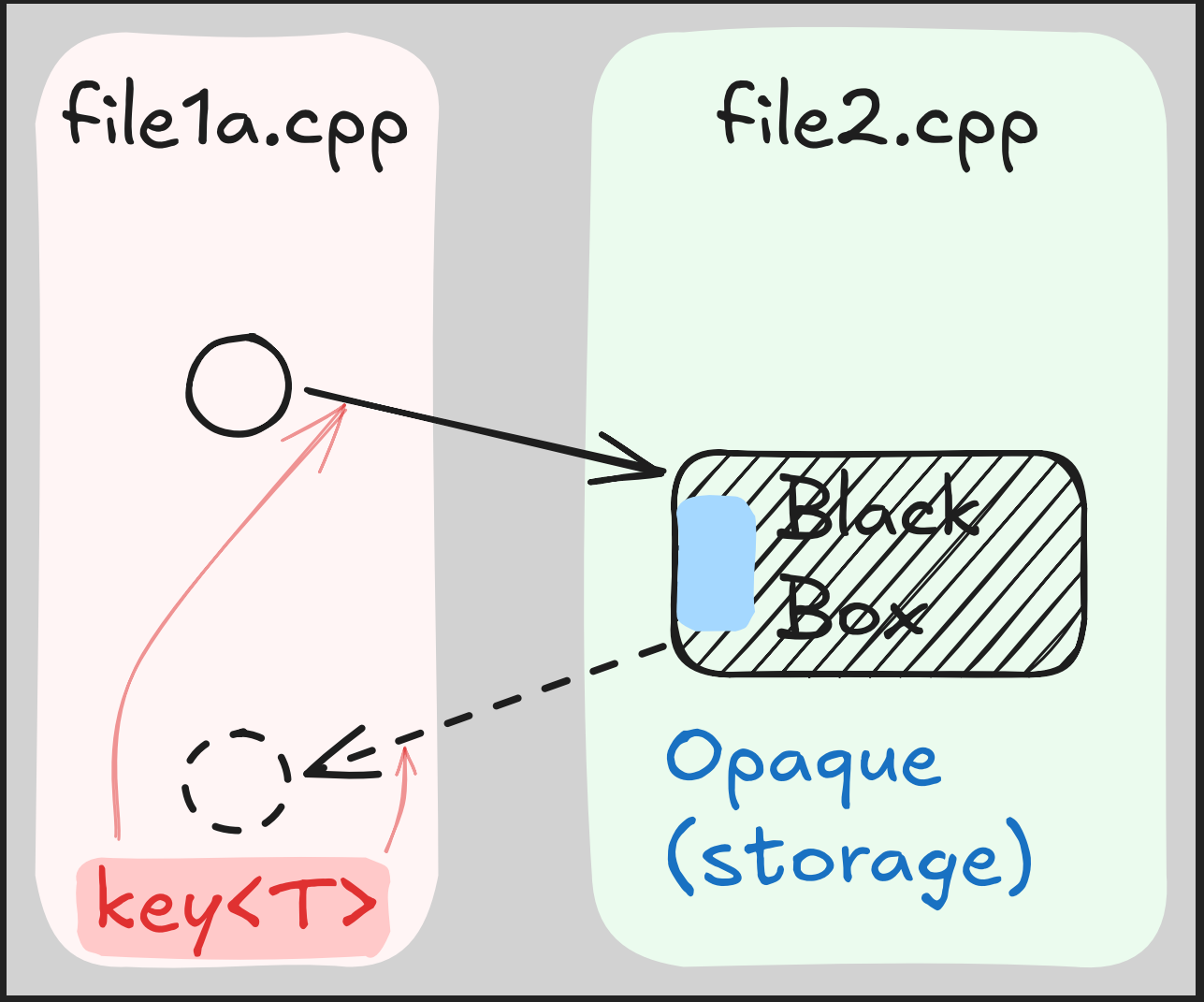


- Key is required to store and retrieve the “contents” to/from the “storage”.
- Lock allows only exact match (here).
- But: Key does not have to be the “type”!

Reification



Object instance as key



Opaque: Example (I)

```
// type-distinct alias, for overloading
struct sequence_t : Opaque {
  sequence_t() = default;
  sequence_t(Opaque &&o) : Opaque(std::move(o)) { }
};
// ... struct alternative_t : Opaque

void AST::Sequence::visit(Visitor &v) {
  sequence_t s;

  v.begin(s);
  for (auto &child : children) {
    v.seq(s);  // (e.g.)
    child.visit(v);
  }
  v.end(s);
}
```

Opaque: Example (II)

```
sequence_t::Accessor<MySequence *> access_seq; // = Opaque::…
alternative_t::Accessor<MyAlternative *> access_alt;

void MyVisitor::begin(sequence_t &s) /*override*/ {
  s.set(access_seq, new MySequence);
}

void MyVisitor::seq(sequence_t &s) /*override*/ {
  // assert(s.is(access_seq));
  auto node = s.get(access_seq);
  …
}

void MyVisitor::end(sequence_t &s) /*override*/ {
  std::unique_ptr<MySequence> node(s.release(access_seq));
  …
}
```

Architecture

- Principle: Separating Data and Access
- Storage (`Opaque`): Holds only an `intptr_t` and a pointer to a vtable (`Accessor_base`).
- Key (`Accessor<T>`): Only the correct accessor instance can get, set, release the data.
- Lock: Invalid pointer casting prevented via strict instance-equality checks (`access == &_access`).
- Destructor of `Opaque` uses stored Accessor-pointer for cleanup.

Implementation idea (I)

```
class Opaque {
public:
  template <typename T, typename Deleter = …std::default_delete<T>…>
  class Accessor;

  // ctor, dtor, move, swap, get, release, set, is, reset …

private:
  class Accessor_base {
    virtual void destroy(intptr_t data) const = 0;
    friend class Opaque;
  };

  intptr_t data = {};
  const Accessor_base *access = {};
};
```

Implementation idea (II)

```
template <typename T, typename Deleter>
class Opaque::Accessor<T *,Deleter> : public Opaque::Accessor_base {
public:
  typedef T *value_type;

  value_type operator()(intptr_t data) const {
    return reinterpret_cast<value_type>(data);
  }

private:
  void destroy(intptr_t data) const override {
    del(operator()(data));
  }
  Deleter del;
};

// Accessor<…, nullptr_t> (no deleter); Accessor<int, …>
```

Implementation idea (III)

```
template <typename Access>
typename Access::value_type Opaque::get(Access &_access) const {
  if (access != &_access) {
    return {};
  }
  return _access(data);
}

template <typename Access>
void Opaque::set(Access &_access, typename Access::value_type _data)
  if (access == &_access && _access(data) == _data) {
    return; // unchanged
  } else if (access) {
    access->destroy(data);
  }
  data = reinterpret_cast<intptr_t>(_data);
  access = &_access;
}
```

Summary/Recap

- Address of the Accessor is the key, must stay alive (singleton / static) until last `Opaque` dtor ran.
- `intptr_t` (specializations: w/ vs. w/o `Deleter`).
- Key usually directly available (same file, …).
- Bad key: `std::bad_cast` / `nullptr` / `.is(…)`
- No `typeid(…)`, `dynamic_cast`, RTTI, … used/needed.
- Minimal memory requirements (`intptr_t` + `const Accessor_base *`) at `Opaque`.

UNIQUE_ANYPTR

- PImpl Idiom (Pointer to implementation)
- Often `std::unique_ptr<Impl>` with only private / forward declared `struct Impl;`
- Works, as long as destructor (`= default;`) not in header, but in implementation.
- Why not hide it fully?
- `unique_anyptr impl;` (in header / public class)
- `unique_anyptr::Accessor<Impl> access_impl`
- (not on public github yet)

SHARED_ANYPTR

- But what if I need a shared pointer?
- Don't reinvent the wheel, also wrt. interop!
- `std::shared_ptr<void>` to the rescue:
- `template <class Y, class Deleter>`
  `void reset(Y *p, Deleter d)`
  I.e.: Deleter is type erased (but can also be just a function pointer)
- `std::dynamic_pointer_cast(…)` will not work (`dynamic_cast` under the hood!)
- (github code does not have interop yet)

Links, Questions?

- Opaque: https://github.com/smilingthax/rexgen/blob/master/opaque.h

  https://github.com/smilingthax/rexgen/blob/master/re-builder_base.h#L11

  https://github.com/smilingthax/rexgen/blob/master/re-builder.cpp#L11
- shared_anyptr:

  https://github.com/smilingthax/unpdf/blob/master/lsyscpp/shared_anyptr.h
- Interfaces: https://github.com/wickedmic/any, https://github.com/smilingthax/poly
- PImpl: https://en.cppreference.com/cpp/language/pimpl
- Slides:

  https://smilingthax.github.io/slides/type_erasure_take_two/