A compile-time check mechanism for scope-bounded object lifetime

Author:

Ryabikov Aleksandr<rsashka@@mail.ru>

Status

Discussion draft for std-proposals

---

1. Summary

This note proposes a small language mechanism for expressing that certain types must exist only under a short-lived, deterministic lifetime root owner.

The proposed syntax as C++ attributes:

```
[[scope_lifetime]]
[[scope_owner]]
```

A type marked [[scope_lifetime]] may exist only when its lifetime is rooted in either:

1. an object with automatic storage duration, or
2. a temporary complete object materialized from a prvalue and destroyed at the end of the same full-expression.

A type marked [[scope_owner]] is treated as a valid owning type for such objects. For objects owned by such a type, the relevant property is the lifetime of the owner object, not the physical storage used to hold the owned objects.

This is intended to allow normal RAII containers and wrappers to remain usable even when they internally use dynamic allocation.

---

2. Motivation

Some types are only meaningful or safe when their lifetime is tightly bound to lexical scope or, more generally, to a short-lived deterministic owner or root.

Typical examples include:

- capability/token objects;
- objects used to ensure transaction idempotency;
- any RAII-managed resources that must not outlive the local context;
- types for which it is important to impose a compile-time lifetime restriction with predictable destruction at the end of the current scope, such as `std::scoped_lock`.

Today, standard C++ provides no direct way to express the rule:

> instances of this type must always remain under a short-lived deterministic ownership root

while still allowing those instances to appear naturally inside ordinary RAII abstractions such as:

- `std::optional<T>`
- `std::vector<T>`
- `std::unique_ptr<T>`

A restriction based on physical storage location is not sufficient, because many valid owners store their elements or managed objects in dynamically allocated memory. What matters is not whether the bytes are stored on the heap, but whether the object's lifetime remains under a valid, short-lived owner root.

This proposal is intended to capture exactly that distinction.

---

3. Proposed attributes

3.1 `[[scope_lifetime]]`

A class type marked [[scope_lifetime]] is constrained such that its instances may exist only under a permitted root lifetime context.

A permitted root lifetime context is one of:

- an object with automatic storage duration,
- a temporary complete object materialized from a prvalue and destroyed at the end of the same full-expression.

3.2 `[[scope_owner]]`

A class type marked [[scope_owner]] is treated as a valid strict owner for the objects it owns.

If it owns objects of a [[scope_lifetime]] type, those objects are considered lifetime-bound to the owner object. The relevant condition is the lifetime of the owner object itself, not the storage mechanism used internally by the owner.

---

4. Core rule

A program is ill-formed if an object of a [[scope_lifetime]] type can exist with a root owner that is neither:

- an object with automatic storage duration, nor
- a temporary complete object whose destruction is guaranteed at the end of the same full-expression.

This rejects:

- namespace-scope objects,
- `static` objects,
- `thread_local` objects,
- standalone dynamically allocated objects,
- containers or owner objects whose own lifetime is not short-lived and deterministic in the required sense.

---

5. Examples

5.1 Direct objects

```
struct [[scope_lifetime]] token {};

void f() {
    token t;   // OK
}

token g;               // ill-formed
static token s;        // ill-formed
thread_local token x;  // ill-formed
auto* p = new token;   // ill-formed
```

5.2 Standard owners

```
struct [[scope_lifetime]] token {};

void f() {
    std::optional<token> o;          // OK
    std::vector<token> v;            // OK
    std::unique_ptr<token> p;        // OK
}

static std::vector<token> gv;        // ill-formed
auto* pv = new std::vector<token>;   // ill-formed
```

5.3 Temporary owners

```
struct [[scope_lifetime]] token {};

void f() {
    consume(std::vector<token>{});        // OK
    consume(std::make_unique<token>());   // OK
}
```

These are intended to be valid because the temporary owner is destroyed at the end of the full-expression.

5.4 Propagation through composition

```
struct [[scope_lifetime]] token {};

struct holder {
    token value;
};

void f() {
    holder h;  // OK
}

static holder h;  // ill-formed
```

The intent is that a type containing a [[scope_lifetime]] subobject is itself subject to the same effective restriction.

---

6. Why containers should be allowed

The proposal is intentionally **not** about banning dynamic allocation as an implementation technique.

For a valid owner type, the important property is:

    whether the object remains lifetime-bound to the owner

not:

    where its bytes happen to be stored

For example, a local std::vector<token> should be valid even if it allocates dynamically, because the vector object itself remains the owner root of its elements and the elements are destroyed under that owner's control.

The same reasoning applies to std::unique_ptr<token> and similar wrappers.

---

7. Owner contract

A [[scope_owner]] type is expected to satisfy the following semantic contract for the objects it owns.

7.1 Deterministic destruction

Owned objects are destroyed no later than destruction of the owner object.

7.2 Early destruction is allowed

An owner may destroy an owned object earlier, through operations such as:

- `reset()`
- `clear()`
- `erase()`
- equivalent owner-controlled operations

This is intended to be valid.

7.3 No live escape from the owner graph

A [[scope_lifetime]] object must not remain alive after leaving a valid owner graph unless it is immediately destroyed or continuously transferred under another valid [[scope_owner]].

This is the key rule that distinguishes acceptable early destruction from unacceptable detachment.

7.4 No shared ownership

The same live object should not be simultaneously owned by multiple independent owners.

7.5 Copy and move

- Copying should either perform deep copy or be disallowed.
- Moving should transfer ownership.

---

8. Important distinction: destruction vs. release

This proposal should allow early destruction:

```
p.reset();   // OK
v.clear();   // OK
```

However, it should reject release-style escape where the object remains alive but no longer belongs to a valid owner:

```
auto* raw = p.release();  // should be ill-formed for [[scope_lifetime]] T
```

The goal is **not** to require that every such object live until the end of the owner's lifetime.

The goal is to require that, while alive, it always remains inside a permitted owner-rooted lifetime graph.

---

9. Natural standard-library owner candidates

The following standard-library types appear to fit the intended [[scope_owner]] model:

- `std::optional`
- `std::array`
- `std::pair`
- `std::tuple`
- `std::variant`
- `std::basic_string`
- `std::vector`
- `std::deque`
- `std::list`
- `std::forward_list`
- associative containers
- unordered associative containers
- `std::unique_ptr`

The following do not appear to fit:

- raw pointers
- `std::shared_ptr`
- `std::weak_ptr`
- `std::span`
- `std::string_view`
- other non-owning view types

std::shared_ptr in particular does not provide a single deterministic lifetime root in the intended sense.

---

10. User-defined owner types

```
template<class T>
class [[scope_owner]] box {
public:
    box() = default;
    explicit box(T value) : ptr_(new T(static_cast<T&&>(value))) {}

    box(box&& other) noexcept : ptr_(other.ptr_) {
        other.ptr_ = nullptr;
    }

    box& operator=(box&& other) noexcept {
        if (this != &other) {
            delete ptr_;
            ptr_ = other.ptr_;
            other.ptr_ = nullptr;
        }
        return *this;
    }

    box(box const&) = delete;
    box& operator=(box const&) = delete;

    ~box() { delete ptr_; }

    void reset() noexcept {
        delete ptr_;
        ptr_ = nullptr;
    }

private:
    T* ptr_ = nullptr;
};
struct [[scope_lifetime]] token {};

void f() {
    box<token> b;   // intended to be OK
}
```

---

11. Intended value of the feature

This proposal is not intended to be a full lifetime-safety system.

It is a narrow mechanism for expressing one specific policy:

> this type must remain under a short-lived deterministic ownership root

If adopted, it would let programmers express that policy directly in the type system and obtain compile-time rejection of clearly invalid roots such as:

- globals,
- `static`,
- `thread_local`,
- standalone `new`,
- long-lived owners.

At the same time, it would preserve the use of ordinary RAII containers and owner wrappers in local code.

---

12. Questions for feedback

I would particularly appreciate feedback on the following questions:

1. Is this problem worth solving at the language level?
2. Are `[[scope_lifetime]]` and `[[scope_owner]]` reasonable names?
3. Is allowing prvalue temporary owners destroyed at the end of a full-expression the right rule?
4. Should user-defined `[[scope_owner]]` types be supported, or should the model initially be limited to a closed set of standard-library owner types?
5. Is there prior work or an earlier paper already close to this design?
