Symantec™

Evaluating structured binding as a condition

Zhihao Yuan

2024/5/29

Previous example with this proposal

```
if (auto [first, last] = parse(begin(), end()))
{
    // interpret [first, last) into a value
}
```

R1 Semantics

- If we model it after a syntax sugar, then

```
if (auto [a, b, c] = fn())
{
    statements;
}
```

condition

is equivalent to

```
if (auto [a, b, c] = fn(); underlying-object)
{
    statements;
}
```

init-statement

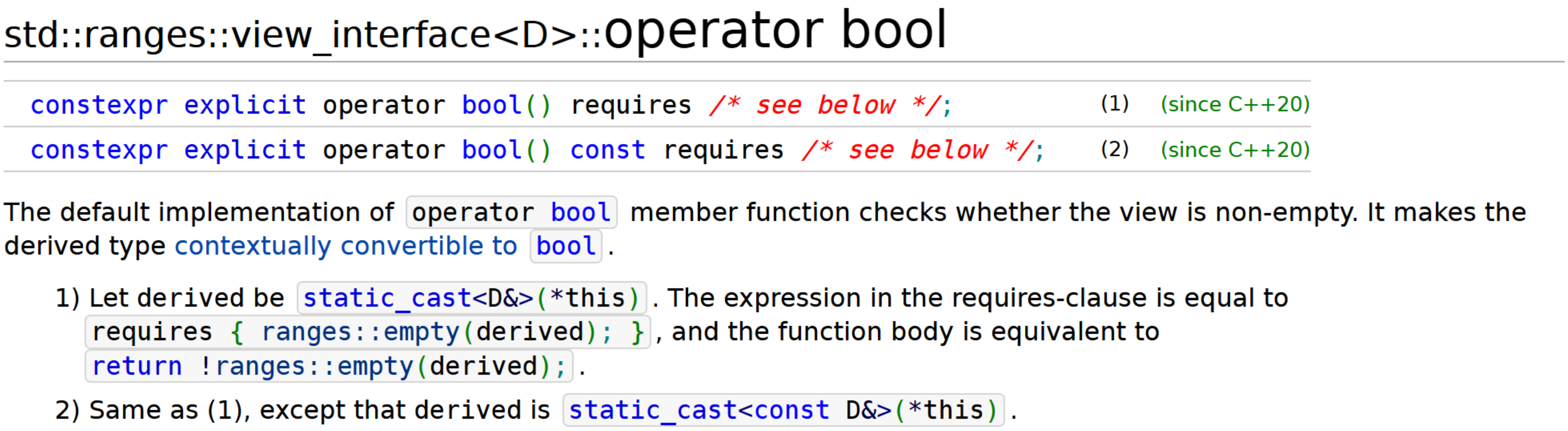
Operator bool in the example

```
struct parse_window
{
    char const *first, *last;

    explicit operator bool() const noexcept
    {
        return first != last;
    }
};

parse_window parse(char const*, char const*);
```

Structured binding protocol

condition protocol

Operator bool in reality



std::ranges::view_interface<D>::operator bool

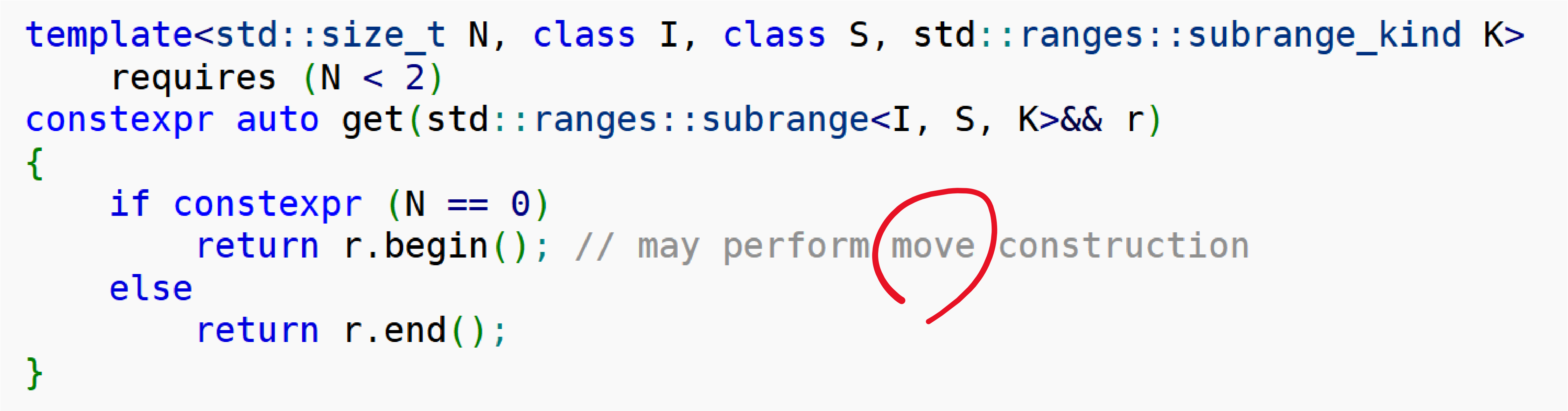
`constexpr explicit operator bool() requires /* see below */;`	(1)	(since C++20)
`constexpr explicit operator bool() const requires /* see below */;`	(2)	(since C++20)

The default implementation of `operator bool` member function checks whether the view is non-empty. It makes the derived type contextually convertible to `bool`.

1) Let derived be `static_cast<D&>(*this)`. The expression in the requires-clause is equal to `requires { ranges::empty(derived); }`, and the function body is equivalent to `return !ranges::empty(derived);`.

2) Same as (1), except that derived is `static_cast<const D&>(*this)`.

Move-only ranges



```
template<std::size_t N, class I, class S, std::ranges::subrange_kind K>
    requires (N < 2)
constexpr auto get(std::ranges::subrange<I, S, K>&& r)
{
    if constexpr (N == 0)
        return r.begin(); // may perform move construction
    else
        return r.end();
}
```

Moving get() + operator bool

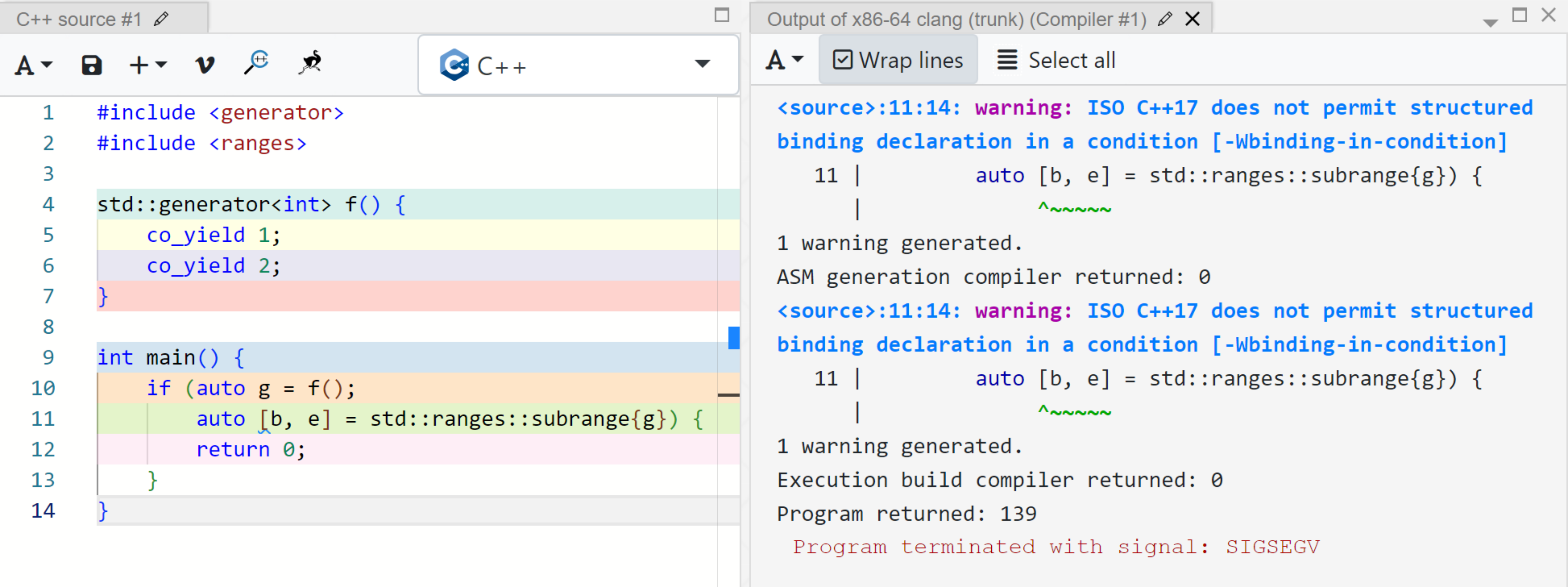
```
if (auto [first, last] = compute_some_subrange())
{
    // ...
}
```

If we reuse the desugaring result

```
auto e = compute_some_subrange();
if (auto [first, last] = std::move(e); e) // approximately
{
    // ...
}
```

Testing a moved-out object

UB in action



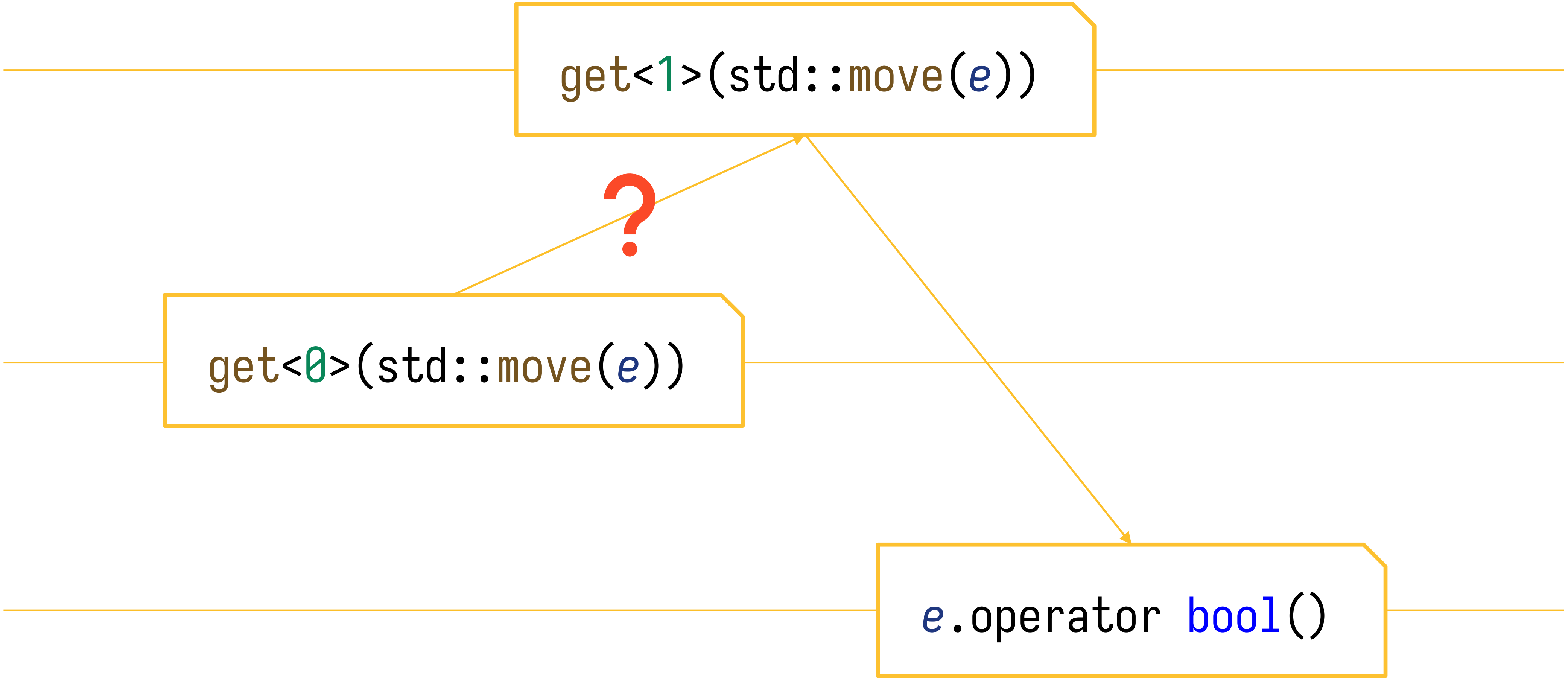
```cpp
#include <generator>
#include <ranges>

std::generator<int> f() {
    co_yield 1;
    co_yield 2;
}

int main() {
    if (auto g = f();
        auto [b, e] = std::ranges::subrange{g}) {
        return 0;
    }
}
```

```
<source>:11:14: warning: ISO C++17 does not permit structured binding declaration in a condition [-Wbinding-in-condition]
   11 |         auto [b, e] = std::ranges::subrange{g}) {
      |              ^~~~~~
1 warning generated.
ASM generation compiler returned: 0
<source>:11:14: warning: ISO C++17 does not permit structured binding declaration in a condition [-Wbinding-in-condition]
   11 |         auto [b, e] = std::ranges::subrange{g}) {
      |              ^~~~~~
1 warning generated.
Execution build compiler returned: 0
Program returned: 139
 Program terminated with signal: SIGSEGV
```

Reimagine

Evaluation order

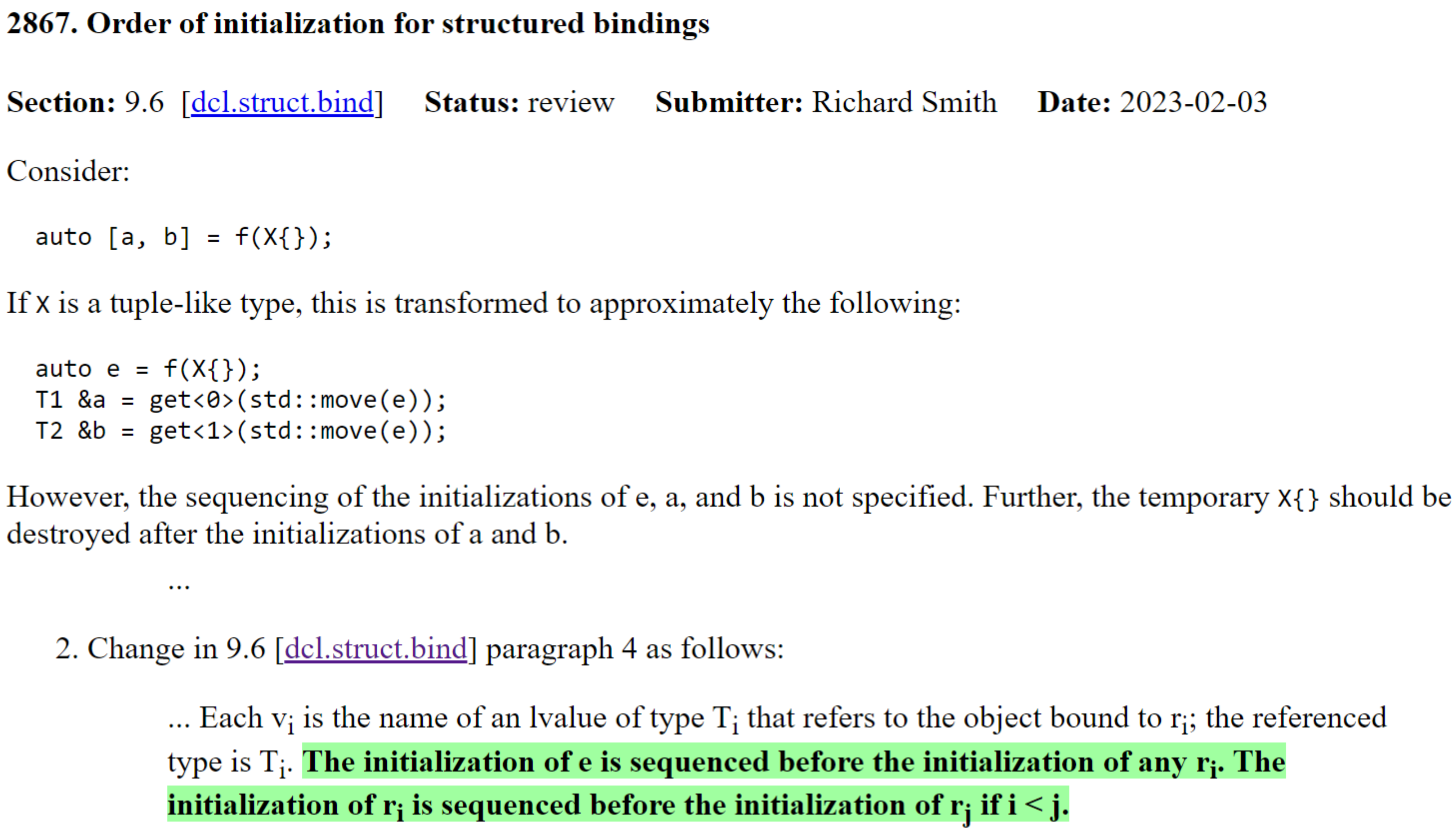
```
auto e = compute_some_subrange();
using E = decltype(e);
using T_1 = std::tuple_element<0, E>::type;
using T_2 = std::tuple_element<1, E>::type;
T_1&& first = get<0>(std::move(e));
T_2&& last = get<1>(std::move(e));
bool t(e.operator bool());
if (t)
{
    // ...
```



```
get<1>(std::move(e))
```

```
get<0>(std::move(e))
```

```
e.operator bool()
```



2867. Order of initialization for structured bindings

Section: 9.6 [dcl.struct.bind] **Status:** review **Submitter:** Richard Smith **Date:** 2023-02-03

Consider:

```
auto [a, b] = f(X{});
```

If `X` is a tuple-like type, this is transformed to approximately the following:

```
auto e = f(X{});
T1 &a = get<0>(std::move(e));
T2 &b = get<1>(std::move(e));
```

However, the sequencing of the initializations of e, a, and b is not specified. Further, the temporary `X{}` should be destroyed after the initializations of a and b.

...

2. Change in 9.6 [dcl.struct.bind] paragraph 4 as follows:

... Each v_i is the name of an lvalue of type T_i that refers to the object bound to r_i; the referenced type is T_i. **The initialization of e is sequenced before the initialization of any r_i. The initialization of r_i is sequenced before the initialization of r_j if $i < j$.**

R2 Semantics

- Evaluating the condition before initializing bindings

```
if (auto [a, b, c] = fn())
{
    statements;
```

can be understood as a hypothetical `if` statement

condition

init-statement

```
if (auto underlying-obj = fn(); auto [a, b, c] = underlying-obj)
{
    statements;
```

Imagined evaluation order as of R1

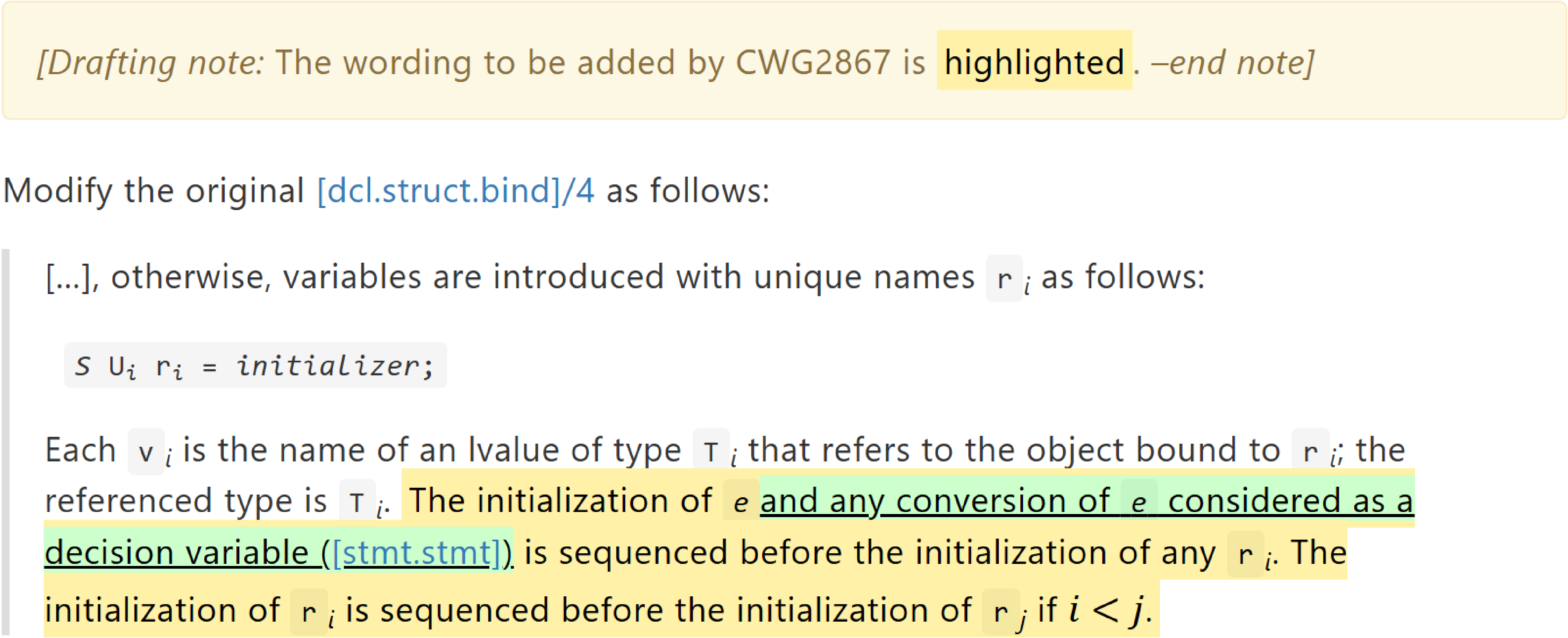
```
auto e = compute_some_subrange();
using E = decltype(e);
using T1 = std::tuple_element<0, E>::type;
using T2 = std::tuple_element<1, E>::type;
T1&& first = get<0>(std::move(e));
T2&& last = get<1>(std::move(e));
bool t(e.operator bool());
if (t)
{
    // ...
```

Proposed evaluation order

decision variable

```
auto e = compute_some_subrange();
using E = decltype(e);
using T1 = std::tuple_element<0, E>::type;
using T2 = std::tuple_element<1, E>::type;
bool t(e.operator bool());
T1&& first = get<0>(std::move(e));
T2&& last = get<1>(std::move(e));
if (t)
{
    // ...
```

R2 Wording



[Drafting note: The wording to be added by CWG2867 is highlighted. *–end note]*

Modify the original [dcl.struct.bind]/4 as follows:

> [...], otherwise, variables are introduced with unique names `r`$_i$ as follows:
>
> $\mathit{S}\ \mathtt{U}_i\ \mathtt{r}_i = \mathit{initializer};$
>
> Each `v`$_i$ is the name of an lvalue of type `T`$_i$ that refers to the object bound to `r`$_i$; the referenced type is `T`$_i$. The initialization of *e* and any conversion of *e* considered as a decision variable ([stmt.stmt]) is sequenced before the initialization of any `r`$_i$. The initialization of `r`$_i$ is sequenced before the initialization of `r`$_j$ if $i < j$.

Thank you