[WG21 Proposal Starting Point]

Document number	PXXXXR0
Date	2026-03-24
Audience	EWG, SG17
Reply-to	Lorand Szollosi <lorro@lorro.eu>
Sources	http://lorro.hu/cplusplus/statement_expressions.html
	https://github.com/lorros/wg21_lorros_clang_uneval_proposal_implementation

Table of Contents

1. Abstract

This paper proposes three interlocking language features that together substantially close the gap between what C++ can express in library code and what currently requires either special-purpose language constructs, macros, or verbose lambda workarounds.

Feature I — Statement Expressions (SE). A block of statements used as an expression, of the form `({ stmt... expr; })`, yielding the type and value of the final expression. Extended with a `produce` keyword for multiple exit points, and with *Parametric* (PSE) and *Named* (NSE) forms that allow a statement-expression body to use the control-flow constructs of its calling context. This feature already exists in GCC, Clang, IBM, Intel, and Open64 in a basic form; this paper proposes standardisation and extension.

Feature II — Unevaluated Parameters (`unevaluated<>`). A function parameter declared `unevaluated<T> p` receives the caller's argument expression without evaluating it; the function body decides if, when, and how many times to evaluate it. This formalises and generalises the call-by-name semantics of the built-in short-circuit operators (`&&`, `||`, `?:`) and of `std::assume` / `__builtin_assume`, making them user-replicable in ordinary functions and operator overloads.

Feature III — Continuation Passing (`cpass<>`). A value declared as `cpass<T>` signals that it is a *cpassable monad*: when used as an argument, the call is transformed into a continuation-passing chain where the monadic value decides whether and how to invoke the rest of the computation. This enables monadic binding, safe short-circuit propagation through nested calls, and assertions that are provably satisfiable while still being checkable.

These three features are mutually reinforcing. Feature I provides first-class expression-level control flow. Feature II provides call-by-name argument evaluation. Feature III builds on both to provide composable, library-level control structures without syntactic overhead. A working implementation exists for Features II and III as a Clang-tidy rewriting pass that translates the proposed spelling into current valid C++. Feature I has prior implementation experience in multiple major compilers.

---

2. Motivation and Scope

2.1 The macro gap

C++ programmers write macros for two deep reasons: the language does not permit a callee to use the caller's control-flow constructs, and it does not permit a callee to defer or repeat argument evaluation. Both restrictions are fundamental, and both drive programmers towards macros or verbose lambda patterns that obscure intent.

Consider complex variable initialisation:

```
// Today — the lambda workaround
std::vector<Handler> handlers = [&] {
    std::map<HandlerEnum, Handler::CPtr> e2handler{
        { EBlueHandler,  new BlueHandler(...)  },
        { EGreenHandler, new GreenHandler(...) },
        { ERedHandler,   new RedHandler(...)   },
        { EPinkHandler,  new PinkHandler(...)  }
    };
    assert(e2handler.size() == NHandlers);
    return e2handler | map_values;
}();    // <- easy to silently omit ()
```

The closing `()` is invisible and easy to drop. The reader cannot tell whether `handlers` is a lambda or the evaluated result until reaching the very end. Compare with a statement expression:

```
// With Feature I
std::vector<Handler> handlers = ({
    std::map<HandlerEnum, Handler::CPtr> e2handler{
        { EBlueHandler,  new BlueHandler(...)  },
        { EGreenHandler, new GreenHandler(...) },
        { ERedHandler,   new RedHandler(...)   },
        { EPinkHandler,  new PinkHandler(...)  }
    };
    assert(e2handler.size() == NHandlers);
    produce e2handler | map_values;
});
```

The `({...})` form is unambiguously an expression. There is no silent non-evaluation if the closing delimiter is omitted.

2.2 Short-circuit in library code

Built-in `||` never evaluates its right operand when its left operand is true. No user-defined function can replicate this today:

```
bool result = a() || expensive_b();      // built-in: b() skipped when a() is tru
bool result = my_or(a(), expensive_b()); // today: always evaluates both
```

The `unevaluated<>` proposal (Feature II) resolves this directly. The motivating example from `clangtv/orig.cpp`:

```
bool f(bool b) { std::cout << "f:" << b << std::endl; return b; }

bool either(unevaluated<bool> a, unevaluated<bool> b)
{
    if (a) return true;
    return b;
}

int main()
{
    std::cout << either(f(true), f(false)) << std::endl;
    // With unevaluated<>: prints "f:1" then "1"  (f(false) never called)
    // Without it (today): prints "f:0" then "f:1" then "1"
}
```

The lambda-based workaround (`clangtv/new.cpp`) requires rewriting every call site:

```
// Current workaround — caller must manually cooperate
std::cout << either([&]{ return f(true); }, [&]{ return f(false); }) << std::endl
```

With `unevaluated<>` as a native feature, the call site is unchanged. The caller writes `either(f(true), f(false))` regardless of whether `either` declares its parameters `unevaluated<>` or not.

2.3 Assertion semantics and the assume problem

`std::assume` (C++23) and `__builtin_assume` tell the compiler that an expression is true without evaluating it. This is useful for optimisation, but the condition cannot be checked in debug builds through the same interface. A library `assert` built on `unevaluated<bool>` can choose whether to evaluate:

```
void assert_true([[nodiscard]] unevaluated<bool> cond)
{
#ifdef NDEBUG
    std::assume(cond);   // never evaluates; tells the optimiser cond holds
#else
    if (!cond)           // evaluates cond exactly once
        std::terminate();
#endif
}
```

The condition is provably always true from the caller's perspective (it is documented by the assertion), but can still be checked in debug builds — without requiring a separate macro.

2.4 Monadic composition

`std::optional::and_then` is useful but threading an error condition through nested calls remains awkward. With Feature III, a `cpass<T>` monad propagates the error automatically. The direct call:

```
std::cout << f(1, f(2, my_cpassable_monad<int>::ret(0))) << " * " << std::endl;
```

— when `0` is the argument — fires a warning and short-circuits the entire chain without reaching the division or the output. The programmer writes natural nested call syntax; the compiler performs the transformation. This is the pattern from `uneval/example/test_cpass.cpp`.

---

3. Feature I — Statement Expressions

3.1 Basic form

A *statement expression* (SE) is an expression of the form `({ stmt-seq expr; })`. Its type is the type of the final expression; its value is that expression's value after executing the preceding statement sequence.

```
int product = ({
    int j = 1;
    for (auto i : v)
        j *= i;
    produce j;
});
```

3.2 The `produce` keyword

`produce` marks a value-producing exit point inside a SE, analogously to `return` in a function. When no `produce` is present, the last expression provides the value (backward compatibility with existing GCC/Clang code). `produce` is necessary whenever multiple exit points in a single SE must each carry a distinct value:

```
#define return_if_false(a) ({ auto tmp = (a); if (!tmp) return std::move(tmp); pr
```

3.3 Grammar

```
expression:
```

```
    ...
    ({ statement-seq_opt expression ; })

jump-statement:
    ...
    produce expression ;
```

3.4 Destructor ordering

Variables created inside a statement expression are destroyed in reverse order of construction when control leaves the SE, regardless of exit path — exactly as automatic-duration variables in a function body. This matches GCC and Clang behaviour.

3.5 Control-flow capture

A SE appearing inside a function may use `return` to return from that enclosing function. A SE appearing inside a loop or `switch` may use `break` and `continue`. Each keyword applies to the innermost enclosing context of the appropriate kind:

```
// Safe accumulation: returns from product_of_vector if any element is zero
#define accumulate_range(range, init, op) \
({                                        \
    auto result = (init);                 \
    for (auto elem : (range))             \
        result = op(result, elem);        \
    produce result;                       \
})

#define product_op(lhs, rhs) \
({                            \
    if (rhs == 0) return 0;  \
    produce lhs * rhs;        \
})

int product_of_vector(const std::vector<int>& v)
{
    return accumulate_range(v, 1, product_op);
}
```

The table below summarises available control-flow constructs inside SEs:

Construct	Available when SE is inside...
`produce`	Always — yields SE's value

`return`	A function or lambda
`break`	A `for`, `while`, or `switch`
`continue`	A `for` or `while`
`co_yield`, `co_return`	A coroutine
`throw`	Always

3.6 Feature detection macros

Feature	Proposed macro
Basic SE	(common subset — no macro)
`return` / `break` / `continue` in SE	`__cplusplus_se_ret`
Parametric SE	`__cplusplus_se_param`
Named SE	`__cplusplus_se_named`
Copy elision in SE	`__cplusplus_se_move`
Non-trivial destructors in SE	`__cplusplus_se_destructor`
Dynamic local statics in SE	`__cplusplus_se_static`
Non-POD class definitions in SE	`__cplusplus_se_class`
Exception handling in SE	`__cplusplus_se_except`

Each macro, if defined, expands to the standard version number in which it was accepted (analogous to `__cplusplus`).

3.7 POD and non-POD class definitions

Class definitions are permitted inside statement expressions, enabling scope-limited helper types:

```
#define try_find(haystack, needle) ({                              \
    using key_t   = decltype(haystack.find(needle)->first);        \
    using value_t = decltype(haystack.find(needle)->second);       \
    auto it = haystack.find(needle);                               \
    struct result_t { key_t key; value_t value; };                 \
```

```
    std::optional<result_t> result;                                  \
    if (it != haystack.end())                                        \
        result = result_t{ it->first, it->second };                  \
    produce result;                                                  \
})
```

3.8 Parametric Statement Expressions (PSE)

A parametric statement expression is similar to a lambda in its capture and arguments, but its body is a statement expression that may use the control-flow constructs of its *calling* context:

```
// PSE used as loop body — may break and continue in the caller's loop
auto loop_body = [&](auto&& x) ({
    if (check1(x)) on_break();
    if (!x) return 0;           // returns from product_of_range
    if (check2(x)) on_break();
    if (x > ignoreLimit) continue;
    i *= x;
});
```

When calling a PSE, the caller explicitly offers its available control-flow constructs (in a template parameter list). It is a compile-time error if the PSE body uses a construct the caller has not offered. It is permitted to offer constructs the PSE does not use.

3.9 Named Statement Expressions (NSE)

A named statement expression is like a function with a statement expression body, which may use the caller's control-flow constructs as declared:

```
// NSE declaration: advertises that it may use return and break from its caller
template<return, break, typename T>
T return_if_zero(T t);

template<typename T>
T return_if_zero(T t)
({
    auto tmp = std::move(t);
    if (!tmp) return 0;    // returns from the CALLER of return_if_zero
    produce tmp;
})

template<typename R>
int product_of_range(const R& range)
{
```

```
    int result = 1;
    for (int elem : range)
        result *= return_if_zero(elem);  // may return 0 from product_of_range
    return result;
}
```

NSEs must be defined in the same compilation unit as their call sites and are inline candidates.

3.10 Opt-in symmetry

The requirement that an NSE/PSE must explicitly declare which control-flow constructs it uses from its caller is analogous to the principle underlying C++ friend declarations. Just as a class grants friendship rather than having it imposed upon it, a function declares which control-flow constructs it borrows from its caller, rather than silently capturing them. The caller cannot involuntarily surrender control-flow to a function that has not declared its intent. This preserves local reasoning: reading a function, one can always determine which constructs may transfer control outside it.

3.11 Inline exceptions (related, separately proposed)

An `inline` qualifier on `throw` and `catch` indicates that the throw/catch pair is an inline candidate — the entire call path must lie within the same compilation unit with no virtual or function-pointer calls. When satisfied, an optimising compiler may implement the exceptional path as cheaply as a `goto` with deterministic destructor calls.

```
struct Break { elem_t elem_; };
try {
    for (auto&& elem : range) {
        if (fn(elem)) throw inline Break{ elem };
        process(elem);
    }
} catch inline (const Break& brk) {
    return brk.elem_;
}
postprocess();
return std::nullopt;
```

NSE/PSE semantics can be reduced to inline exceptions, providing an implementability proof for compilers that wish to build on existing exception infrastructure.

---

4. Feature II — Unevaluated Parameters

4.1 The problem

Every user-defined function in C++ evaluates all its arguments at the call site, unconditionally and exactly once. The built-in operators `&&`, `||`, and `?:` do not. `std::assume` and `__builtin_assume` do not. There is no way for a user-defined function or operator overload to replicate this.

4.2 Proposed spelling

```
template<typename T>
using unevaluated = T;   // current no-op alias — recognised by the proposal
```

In the proposed native implementation, `unevaluated<T>` becomes a parameter qualifier understood by the compiler. The existing no-op-alias spelling is used in the current implementation (§9) for forward compatibility.

4.3 Semantics

A parameter declared `unevaluated<T> p` is passed the caller's argument expression without evaluating it. The body evaluates it by naming it in an evaluated context — each such use is a separate evaluation:

```
// Short-circuit either: b is only evaluated if a is false
bool either(unevaluated<bool> a, unevaluated<bool> b)
{
    if (a) return true;
    return b;
}
```

Call site: `either(f(true), f(false))` — identical to calling a function with regular parameters. The compiler handles the difference; the caller writes nothing special.

4.4 Opt-in requirement

A function must explicitly declare `unevaluated<T>` on a parameter to receive an unevaluated argument. A caller cannot impose unevaluated passing on a function that has not opted in. This mirrors the C++ friend model: the feature is granted by the callee, not demanded by callers. Changing a parameter from `T` to `unevaluated<T>` is a source-compatible change; no alterations at call sites are required.

4.5 Multiple evaluation

Each syntactic use of an `unevaluated<T>` parameter is a separate evaluation of the original

argument expression — the same semantics as a macro argument, but with type safety, proper scoping, and RAII:

```
void assert_not_negative(unevaluated<int> val)
{
    if (val < 0)                          // first evaluation
        std::cerr << "got " << val        // second evaluation
                  << ", expected >= 0\n";
}
```

Implementations should warn when an `unevaluated<T>` parameter is used more than once with a side-effectful or expensive argument.

4.6 Non-evaluation use: `std::assume`

An `unevaluated<bool>` parameter may be passed to `std::assume` without evaluating it:

```
void assert_true(unevaluated<bool> cond)
{
#ifdef NDEBUG
    std::assume(cond);   // never evaluates; informs the optimiser
#else
    if (!cond)           // evaluates once
        std::terminate();
#endif
}
```

The condition is provably always true as documented, yet checkable in debug builds.

4.7 Unevaluated operator overloads

`unevaluated<>` extends to operator overloads, enabling user-defined operators with the same short-circuit properties as built-in `&&` and `||`:

```
// User-defined short-circuit OR with the same semantics as built-in ||
bool operator||(bool lhs, unevaluated<bool> rhs)
{
    if (lhs) return true;
    return rhs;
}
```

This is previously impossible to express for any user-defined operator.

---

5. Feature III — Continuation Passing

5.1 The problem

Monadic error propagation in C++ requires explicit checking at every call site, or a chain of `.and_then(...)` calls that obscures the original computation. Neither form scales well for multi-step nested calls. `cpass<>` recovers natural nested call syntax while enabling the monad to intercept and short-circuit the continuation chain.

5.2 Proposed spelling

```
template<typename T>
using cpass = T;    // current no-op alias — recognised by the proposal
```

A value of type `cpass<T>` is a *cpassable monad*. When used as an argument to a function call, the call is transformed: the caller's continuation (the remaining computation) is passed as a callable to the monad's `operator()`. The monad decides whether to invoke the continuation, potentially short-circuiting the entire chain.

5.3 The cpassable monad interface

A type is a cpassable monad if it provides `operator()(F&& f)` where `f` is the continuation. The monad's `operator()` returns `cpass<...>` to enable further chaining:

```
template<typename T>
struct my_cpassable_monad
{
    T i;

    my_cpassable_monad(T i) : i(i) {}

    template<typename F>
    cpass<my_cpassable_monad<std::invoke_result_t<F, T>>> operator()(F&& f) const
    {
        if constexpr(std::is_same_v<T, int>) {
            if (!i) {
                // Short-circuit: warn and produce a default sentinel.
                // The continuation f is never called.
                std::cout << "Warning: i == 0" << std::endl;
                if constexpr(std::is_same_v<std::invoke_result_t<F, T>, std::ostr
                    return my_cpassable_monad<std::invoke_result_t<F, T>>(
                                std::forward<F>(f)("0 or INF"));
                else
                    return my_cpassable_monad<std::invoke_result_t<F, T>>({});
```

```
            }
        }
        return my_cpassable_monad<std::invoke_result_t<F, T>>(
                    std::forward<F>(f)(i));
    }

    operator T() const { return i; }   // not called in normal cpass flow

    static cpass<my_cpassable_monad<T>> ret(T i)
    { return my_cpassable_monad<T>(i); }
};
```

5.4 Basic example — division with zero guard

From `uneval/example/test_cpass.cpp`. The programmer writes:

```
int f(int x, int y) { return x / y; }

// Natural spelling: nested calls, no explicit chaining
std::cout << f(1, f(2, my_cpassable_monad<int>::ret(0))) << " * " << std::endl;
```

The cpass rewriting pass (§9) transforms this to the continuation-passing form:

```
// Generated form (current valid C++):
my_cpassable_monad<int>::ret(0)
    ([&](auto&& cval) -> decltype(f(2, std::forward<decltype(cval)>(cval)))
        { return f(2, std::forward<decltype(cval)>(cval)); })
    ([&](auto&& cval) -> decltype(f(1, std::forward<decltype(cval)>(cval)))
        { return f(1, std::forward<decltype(cval)>(cval)); })
    ([&](auto&& cval) -> decltype(std::cout << std::forward<decltype(cval)>(cval)
        { return std::cout << std::forward<decltype(cval)>(cval); })
    ([&](auto&& cval) -> decltype(std::forward<decltype(cval)>(cval) << " * ")
        { return std::forward<decltype(cval)>(cval) << " * "; })
    ([&](auto&& cval) -> decltype(std::forward<decltype(cval)>(cval) << std::endl
        { return std::forward<decltype(cval)>(cval) << std::endl; });
```

When `ret(0)` is called, its `operator()` detects `i == 0`, prints the warning, and returns a default sentinel — none of the subsequent lambdas are invoked. Division by zero is avoided automatically.

5.5 Design evolution in the repository

The repository contains four earlier revisions documenting the design progression:

orig1 — Monomorphic. `operator()` returns `cpass<int>` directly. Limited: cannot chain

through `std::ostream&` . Demonstrates the basic interception pattern.

orig2 — Returns `cpass<my_cpassable_monad>` wrapping the result. Enables chaining `f(1, f(2, ret(0)))` but cannot chain through heterogeneous return types.

orig3 — Two-overload design. When `F(int)` returns `int` , wraps in monad; when it returns something else (e.g. `std::ostream&`), returns raw. Allows the chain to terminate via `operator<<` . Still monomorphic in the held type.

orig4 / final — Fully generic `my_cpassable_monad<T>` . The held type `T` is a template parameter, propagated through `std::invoke_result_t<F, T>` at each step. Uses `if constexpr` to specialise behaviour for `T == int` . This is the production form.

5.6 Comparison with `std::optional::and_then`

`std::optional::and_then(f)` requires `f` to return `std::optional<U>` . The `cpass<>` approach is more general: the monad type is user-defined, the conversion logic is user-defined, and the call site is unchanged natural nested-call syntax. Unlike `and_then` , the programmer does not rewrite calls into a chain of method calls; the compiler performs the transformation from the natural form.

---

6. How the Three Features Compose

6.1 Safe accumulation with early exit, assertion, and monadic propagation

```
// NSE (Feature I): may return from its caller
template<return, typename T>
T return_if_zero(T t)
({
    auto tmp = std::move(t);
    if (!tmp) return 0;
    produce tmp;
})

template<typename R>
std::optional<int> range_product(const R& range)
{
    // Feature I (SE): complex initialisation with assertions
    auto validated = ({
        auto v = range;
        assert_true(v.size() <= MAX_RANGE);   // Feature II: provably always true
        produce v;
    });
```

```
    int result = 1;
    bool empty = true;
    for (auto&& x : validated) {
        empty = false;
        // Feature III: cpass<int> propagates zero through the monad
        result = return_if_zero(cpass<int>(x));
    }
    if (empty) return std::nullopt;
    return result;
}
```

6.2 Library-level `for_else` via PSE syntax sugar

Many proposals for new loop forms (`for...else` , `for...break` , `if (auto x : opt)`) require language changes. With PSE and the proposed in-place PSE syntax, they become library features — the language change is in the generic mechanism, not in each individual construct:

```
// No special language construct — for_if is a library NSE
for_if(auto&& x : v) {
    if (check1(x)) break;
    if (!x) return 0;
    if (check2(x)) break;
    if (x > ignoreLimit) continue;
    i *= x;
}
.on_break([&](auto x) ({ /* handle break case */ }))
.on_empty([&]         ({ return std::nullopt; }))
.on_done ([&]         ({ return i; }));
```

6.3 `unevaluated<>` and `cpass<>` together

When a cpassable monad's `operator()` takes `unevaluated<>` parameters, the continuation is only evaluated if the monad chooses to proceed — and within the continuation, further arguments may themselves be lazy. This is the composition that enables monadic computation with full call-by-need semantics in user-defined library types.

---

7. Impact on the Standard

7.1 Grammar additions

```
expression:
    ...
    ({ statement-seq_opt expression ; })

jump-statement:
    ...
    produce expression ;

template-parameter:
    ...
    return
    break
    continue
```

7.2 New keyword

`produce` is proposed as a new keyword. Code using `produce` as an identifier requires updating. A `#define produce` (empty define) enables backward compatibility for SE code without multiple exit points.

7.3 Parameter qualifier

`unevaluated` is proposed as a parameter qualifier (contextual keyword or standard-library type alias with compiler backing). `cpass` is proposed as a qualifier understood by the compiler on parameter and return types of `operator()`.

7.4 Related proposals

These proposals are not competing; they are aspects of an integrated feature set:

- Abbreviated lambda syntax — PSE provides comparable power in a different register
- Continuations proposal — SE provides a library-level mechanism for some continuation patterns
- `for...else`, `for...break`, `if (auto x : opt)` — become library-implementable via PSE/NSE
- `std::assume` (C++23) — `unevaluated<>` provides a typed, optionally-checkable complement
- Monadic operations for `std::optional` (P0798) — `cpass<>` generalises these to arbitrary user monads with natural call syntax
- Contracts (ongoing) — `unevaluated<>` enables library-level contract assertions that degrade gracefully across build modes

8. Design Decisions and Alternatives

8.1 `produce` vs. last-expression rule

Existing GCC/Clang SE implementations use the last expression as the value, without a keyword. This proposal retains that rule as optional — when no `produce` is present, the last expression provides the value. `produce` is additionally proposed so that multiple exit points within a single SE can each specify a value explicitly. Without `produce`, an early exit from a SE (e.g. `if (!x) return 0;`) cannot carry a distinct SE-level value separately from the function return value.

8.2 Why a native feature rather than a library alias

The no-op-alias form is sufficient for the clang-tidy rewriting tool and demonstrates well-defined semantics. Standardising as a language feature is motivated by:

1. The alias form requires a separate tool pass; a native feature requires no tooling.
2. The alias form changes call-site code; a native feature is call-site transparent.
3. The native feature participates in overload resolution and template deduction in ways the alias cannot.
4. The alias form cannot natively interact with `std::assume`, which is a compiler intrinsic.

8.3 Staged acceptance

A possible outcome is staged acceptance: the common subset of existing SE implementations (basic SE, `produce`, control-flow capture) is accepted first; PSE and NSE form a second stage. Feature detection macros (§3.6) are proposed precisely to support this: implementations may provide subsets, and user code can detect which features are available.

8.4 Opt-in symmetry

The decision that `unevaluated<>` must be declared by the callee (not imposed by the caller) mirrors the C++ `friend` system. Just as a class grants friendship rather than having it forced upon it, a function grants unevaluated-argument reception. This preserves local reasoning: reading a call site, the programmer knows arguments are evaluated normally unless the function's declaration says otherwise.

8.5 Multiple evaluation and side effects

The multiple-evaluation semantics of `unevaluated<T>` are the same as macro argument semantics, but with type safety, proper scoping, and RAII. For expensive or side-effectful arguments, single-evaluation is obtained by binding: `auto tmp = p;` . Implementations should warn when an `unevaluated<T>` parameter is used more than once with a non-trivially-copyable or visibly side-effectful argument.

---

9. Implementation Experience

9.1 Statement expressions

The basic SE form (`({ stmt... expr; })`) is implemented in GCC (since version 3), Clang, IBM XL C++, Intel C++, and Open64. Destructor support and `return` / `break` / `continue` capture are supported in GCC and Clang. PSE and NSE are new in this proposal.

9.2 `unevaluated<>` and `cpass<>` — clang-tidy rewriting pass

A complete implementation exists as a Clang-tidy check (`UnevaluatedAliasToLambdaCheck`) in the repository `https://github.com/lorros/wg21_lorros_clang_uneval_proposal_implementation` , under `uneval/src/` . The check provides four mechanical transformations that together faithfully encode the proposed semantics in current valid C++:

Transformation 1 — Parameter type rewrite. `unevaluated<T> p` → `auto p` (accepts a zero-argument callable).

Transformation 2 — Call-site argument wrapping. The argument `expr` passed to an `unevaluated<T>` parameter → `[&]{ return (expr); }` .

Transformation 3 — Body use rewrite. Each use of `p` (an `unevaluated<T>` parameter) in the function body → `p()` .

Transformation 4 — `cpass<T>` call rewriting. A call where one argument is a `cpass<T>` value is rewritten to a continuation-passing chain. Given:

```
f(a1, cpass_val, a3)
```

the rewriter produces:

```
cpass_val([&](auto&& cval)
    -> decltype(f(a1, std::forward<decltype(cval)>(cval), a3))
    { return f(a1, std::forward<decltype(cval)>(cval), a3); })
```

The check then unconditionally walks up the AST parent chain, appending a lambda for each parent call that immediately consumes the result — until no further consuming parent is found. It handles `CXXOperatorCallExpr` (including `operator<<`), materialized temporaries, implicit object arguments, and avoids recursion limit issues.

The existence of this mechanical encoding demonstrates that:

- The semantics of both features are well-defined and expressible in current C++
- The features are implementable via AST rewriting; a native implementation can follow the same semantics
- The encoding is faithful: every program using the proposed features has a well-formed equivalent in current C++

9.3 Example files

`uneval/example/test.cpp` — Simplest `unevaluated<T>` example: a function `f` taking an `auto` callable and an `int`, called via lambda today, corresponding to the proposed natural spelling.

`uneval/example/test_either.cpp` — Both-sides `unevaluated<bool>` short-circuit `either`:

```
template<typename T>
using unevaluated = T;

bool either(unevaluated<bool> a, unevaluated<bool> b)
{
    if (a) return true;
    return b;
}
```

`uneval/example/test_cpass.cpp` — Full monadic chain through `my_cpassable_monad<T>` demonstrating zero-guard propagation, continuation through `f(2, cval)` and `f(1, cval)`, and final chain termination through `std::ostream&`.

---

10. Acknowledgements and Related Work

The author thanks the ISO C++ Standard – Future Proposals Group for feedback on statement expressions.

Related proposals and prior work:

- GCC statement expressions extension (GCC 3 onwards)

- `std::assume` — P1774R8 (C++23); `unevaluated<>` provides a typed, optionally-checkable complement
- Monadic operations for `std::optional` — P0798R6 (C++23); `cpass<>` generalises to arbitrary user monads
- Contracts — ongoing; `unevaluated<>` enables library-level assertions across build modes
- P2893 — Variadic friends; the opt-in bilateral consent model of §4.4 draws on the same design principles as the friend system
- Abbreviated lambdas — PSE provides comparable expressiveness in a statement-expression register
- Coroutines (C++20) — `co_yield` and `co_return` are expected to compose naturally with SEs

---

11. References

- GCC Statement Expressions: https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html
- Statement Expressions proposal: http://lorro.hu/cplusplus/statement_expressions.html (Lorand Szollosi, 2017)
- Implementation repository: https://github.com/lorros/wg21_lorros_clang_uneval_proposal_implementation
- P0798R6 — Monadic operations for `std::optional` (Sy Brand, 2021)
- P1774R8 — `std::assume` (Timur Doumler, Antony Polukhin)
- P2893R2 — Variadic friends
- P3859R0 — Assertions are not necessarily for changing program behavior (Andrzej Krzemieński)
- P1105R1 — Leaving no room for a lower-level language: A C++ Subset (Ben Craig)

Document number PXXXXR0 — to be assigned on submission to the WG21 pre-meeting mailing.