

Document Number: D0917R4
Date: 2026-10-23
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: EWG
Target: C++29

MAKING OPERATOR?: OVERLOADABLE

ABSTRACT

This paper proposes user-defined overloads of `operator?:`. In addition to normal function calls, `operator?:` can also be defaulted, in which case it simply defines a common type to use for the built-in conditional operator. To avoid a breaking change, user-defined overloads of `operator?:` are only considered when no built-in candidate is viable.

CONTENTS

1	CHANGELOG	1
2	STRAW POLLS	2
3	SUGGESTED POLLS	3
4	INTRODUCTION	4
5	MOTIVATION	4
6	PROPOSAL	9
7	IMPLEMENTATION EXPERIENCE	12
8	DISCUSSION	12
9	WORDING	13
A	TEST CASES	13
B	BIBLIOGRAPHY	33

1

CHANGELOG

1.1

CHANGES FROM REVISION 0

Previous revision: P0917R0

- Added `bounded::integer` motivation and example.
- Added a reference to P0927R0; making a stronger case for the simple choice.

1.2

CHANGES FROM REVISION 1

Previous revision: P0917R1

- Discuss `common_type`.
- Discuss overloading `operator?:(bool, ...)`.
- Mention `chrono::duration` and other numeric types as motivation.

1.3

CHANGES FROM REVISION 2

Previous revision: P0917R2

- Add Tony tables.
- Explore defaulting `operator?:`.
- Discuss synthesizing `operator?:` from `common_type`.
- Define a `contextual_boolean` concept, that most overloads should use instead of a naïve `bool` parameter.
- Try to be clearer about generality, consistency, and orthogonality of this proposal.
- Add `boost::yap` as another existing library that is missing the ability to overload `?:`.

1.4

CHANGES FROM REVISION 3

Previous revision: P0917R3

- Basically a rewrite.
- Implemented and tested the feature while writing down the parts of the proposal.
- Add an example from quantity and units.
- Add motivation example from P2655R3 "common_reference_t of reference_wrapper should be a reference type".

2

STRAW POLLS

2.1

LEWG AT RAPPERSWIL 2018

Poll: Temperature of the room: LEWG supports overload of ?:

SF	F	N	A	SA
2	5	1	1	2

Poll: LEWG supports overload, assuming lazy eval is available

SF	F	N	A	SA
4	5	2	?	?

2.2

EWGI AT SAN DIEGO 2018

Poll: Should we commit additional committee time to overloading operator?: knowing it will leave less time for other work?

SF	F	N	A	SA
1	3	6	2	0

2.3

EWGI IN COLOGNE 2019

Poll: This proposal should explore defaulted operator?: only, instead of fully-customizable?

SF	F	N	A	SA
0	0	5	5	3

Poll: Lazy operators should be standardized before overloading operator?: can be standardized.

SF	F	N	A	SA
2	3	5	2	1

Poll: Continue spending committee time on this versus other proposals, given that time is limited?

SF	F	N	A	SA
1	9	5	0	0

2.4

EWGI IN BELFAST 2019

Poll: Pursue defaulted operator?:

SF	F	N	A	SA
0	3	6	3	1

Poll: Pursue 2-argument defaulted operator?:

SF	F	N	A	SA
2	3	2	5	2

Poll: Make operator?: overloadable with a requirement of a non-boolean condition (because of missing lazy eval)

SF	F	N	A	SA
0	5	4	3	0

Poll: Unrestricted operator?: overloads, trusting our users to use it responsibly

SF	F	N	A	SA
5	6	3	0	0

Poll: Forward to EWG

SF	F	N	A	SA
3	8	1	0	0

Specific requests:

- Discuss whether we still want `common_type` (also with multiple types) to derive its answer from user-defined conditional operators. For context, state the intent of `common_type`.
- Explore `= delete;` to go with defaulting.

3

SUGGESTED POLLS

Poll: Allow operator?: as class member (explore and report back if there are issues)

SF	F	N	A	SA

Poll: Pursue defaulted operator?:

SF	F	N	A	SA

Poll: Pursue 2-argument defaulted operator?:

SF	F	N	A	SA

Poll: Pursue 3-argument defaulted operator?: turning `bool` into contextually convertible to `bool`

SF	F	N	A	SA

Poll: `operator?:` should have special rules to avoid overriding the built-in operator

SF	F	N	A	SA

Poll: Make `operator?:` overloadable but require non-boolean condition until lazy eval lands

SF	F	N	A	SA

Poll: Unrestricted `operator?:` overloads, trusting our users to use it responsibly

SF	F	N	A	SA

4

INTRODUCTION

Most operators in C++ can be overloaded. The few exceptions are: `?:`, `::`, `..`, and `.*`. For the conditional operator, Stroustrup [3] writes: “There is no fundamental reason to disallow overloading of `?:`. I just didn’t see the need to introduce the special case of overloading a ternary operator. Note that a function overloading `expr1?expr2:expr3` would not be able to guarantee that only one of `expr2` and `expr3` was executed.”

In this paper I want to show important use-cases for overloading the conditional operator, how it makes the language more consistent, and how lazy evaluation is possible even without a generic language extension for lazy evaluation.

5

MOTIVATION

5.1

DESIGN PRINCIPLES

BE GENERAL “Don’t restrict what is inherent. Don’t arbitrarily restrict a complete set of uses. Avoid special cases and partial features.” [P0745R0]

C++ allows operator overloading for almost all operators. That `operator?:` cannot be overloaded is an arbitrary restriction (esp. in the face of `operator&&` and `operator||`). More importantly, the conditional operator expresses a numerical operation, implementing piecewise functions (and the simple case of step functions). We allow user-defined numeric types to define all their operators, except for piecewise functions. That works out fine as long as no mixed types are involved, because `operator?:` is essentially auto-generated for the simple cases. However, there are very important cases for numeric types (which are not interconvertible), which have a common type that is neither of the two types and thus fail to work with this one operator. That’s inconsistent. Also, the conditional operator naturally generalizes to a blend operation when applied element-wise (i.e. multiple booleans as condition). This paper proposes a complete set of uses:

- by not restricting the type of the condition (allow any user-defined type);
- by not restricting to only allow deferred evaluation (this is *necessary* for blend operations).

BE CONSISTENT “Don’t make similar things different, including in spelling, behavior, or capability. Don’t make different things appear similar when they have different behavior or capability.”

Currently, user-defined types that are interconvertible cannot be used with the conditional operator and require a function instead. Interconvertible types are often a bad idea, but are required, for example, when the goal is to model built-in integer types. I.e. without an overloadable `operator?:` it is impossible to write user-defined types that are a drop-in replacement for built-in types.

A consistency argument for defaulting `operator?:` can be made: We allow defaulting default constructors and comparison operators because in the majority cases, the behavior of these functions can be inferred. In addition defaulted default constructors enable trivial construction, which a default constructor that implements the same behavior explicitly does not enable. Implementing `operator?:`, as proposed in this paper, aligns with this approach.

Section 5.3 is another consistency argument.

BE ORTHOGONAL “Avoid arbitrary coupling. Let features be used freely in combination.”

The built-in conditional operator only evaluates the expression chosen by the predicate. User-defined functions/operators evaluate all expressions in their arguments before calling the function. There are situations where such functions/operators would benefit from lazy evaluation. Lazy evaluation for user-defined functions is an orthogonal problem to solve. It should not be a show-stopper for overloadable `operator?:`: if it—by itself—does not implement function calls with deferred evaluation. Such a facility is needed just as much for `operator&&` and `operator||` as it would be needed for `operator?:`. There has been concern about adding the ability to overload `operator?:` before/without a general mechanism for lazy evaluation. However, since the two features are orthogonal, this is a concern about trusting/hand-holding our users¹.

5.2

PIECEWISE FUNCTIONS / USER-DEFINED NUMERIC TYPES

Piecewise functions, such as $f(x)$ in (1), benefit from the conditional operator.

$$f(x) = \begin{cases} \text{formula}_1 & \text{if condition}_1 \\ \text{formula}_2 & \text{if condition}_2 \\ \vdots & \vdots \\ \text{formula}_n & \text{if condition}_n \end{cases} \quad (1)$$

However, if the type of the expressions in each branch are different, the conditional operator requires that one of the participating types is the common type (both are implicitly convertible to it, but not the other). An example where this can occur is Quantity and Units:

¹ Which is what coding guidelines are used for. With great power comes great responsibility.

```

1 quantity<isq::speed[km / h]> speed_1 = 42 * isq::speed[m / s];
2 quantity<isq::speed[m / s]> speed_2 = 42 * isq::speed[km / h];
3 quantity_of<isq::speed> auto speed_3 = speed_1 + speed_2; // OK
4 quantity_of<isq::speed> auto speed_4 = cond ? speed_1 : speed_2; // ill-formed
5
6 // however, a common type exists:
7 using ct = std::common_type_t<decltype(speed_1), decltype(speed_2)>;
8 static_assert(std::is_same_v<ct, decltype(speed_3)>);

```

A similar problem arises for integer types that only make value-preserving conversions implicit. In this case a common type between an unsigned and signed integer type can be a signed integer type that is neither of the two input types. It is therefore awkward to introduce a type where conversions are “safe”, because the conditional operator fails to support it.

Any library-based numeric type may have a need for overloading `operator?:` if the type carries information about the value or even modifies the value (e.g. for `std::chrono::duration`). Most of those types specialize `std::common_type`². Examples:

- `std::chrono::duration<Rep, Period>`
- `std::chrono::time_point<Clock, Duration>`
- `rational` from Seymour [P1438R1] (does not specialize `common_type`)
- `fractional<Numerator, Denominator>` from McFarlane [P1050R0]
- `fixed_point<Rep, Exponent, Radix>` from McFarlane [P0037R5]
- `bounded::integer<minimum, maximum>` from Stone [2]

Consider the `bounded::integer` example (cf. [2]):

```

1 bounded::integer<1, 100> const a = f();
2 bounded::integer<-3, 7> const b = g();
3 bounded::integer<-2, 107> c = a + b;
4 bounded::integer<-3, 100> d = some_condition ? a : b; // ill-formed

```

Line 3 is what the `bounded::integer` library can currently do for you. However, line 4 is currently not possible since it would require more control by the library over the types involved (arguments and result) with the conditional operator.

Any design that wants to allow different types on the second and third argument (without implicit conversions), and determine a return type from them, requires an overloadable conditional operator. Note that user-defined numeric types want a signature such as `operator?:(contextual_boolean auto, UDT1, UDT2)` in most cases. I.e. the idea to only allow non-boolean conditions (because of missing implicit deferred evaluation) on `operator?:` overloads breaks this use case. (I mentioned the idea in the previous revisions and it was also suggested in EWG-I discussion).

5.3

BLEND OPERATIONS

Besides its use for numeric types, the conditional operator is also a perfect match for simd-generic programming. The `std::simd::select` function was introduced as a poor-mans `operator?:` overload. The conditional operator is a perfect match for expressing the `select` operation, which is an

² cf. https://codesearch.isocpp.org/cgi-bin/cgi_ppsearch?q=struct+common_type%3C&search=Search

before	with D0917R4
<pre> 1 bounded::integer<1, 100> const a = f(); 2 bounded::integer<-3, 7> const b = g(); 3 auto c = BOUNDED_CONDITIONAL(4 some_condition, a, b); </pre>	<pre> 1 bounded::integer<1, 100> const a = f(); 2 bounded::integer<-3, 7> const b = g(); 3 auto c = some_condition ? a : b; </pre>

~~Tony~~Before/After Table 1: bounded::integer now and with overloadable operator?:

before	with D0917R4
<pre> 1 template<class T, class U> 2 void f(bool cond, T a, U b) 3 { 4 if constexpr (5 is_bounded_integer<T>::value 6 is_bounded_integer<U>::value) 7 g(BOUNDED_CONDITIONAL(8 some_condition, a, b)); 9 else 10 g(cond ? a : b); 11 } </pre>	<pre> 1 template<class T, class U> 2 void f(bool cond, T a, U b) 3 { 4 5 6 7 8 9 10 g(cond ? a : b); 11 } </pre>

~~Tony~~Before/After Table 2: supporting bounded::integer in a generic function

element-wise implementation of piecewise functions. To illustrate what I mean, consider the simple case of a generic absolute value function:

```

1 template <class T> T abs(T x) {
2     return x < 0 ? -x : x;
3 }

```

For arithmetic types, the conditional operator chooses to either evaluate $-x$ or x . For a `simd::vec<T>`, which stores multiple values of type T and implements all its operators to act element-wise, $x < 0$ yields multiple (different) booleans, which choose the corresponding element from $-x$ or x . Consequently, both right-hand side expressions must be evaluated for the blend operation (i.e. there is no need for implicit lazy evaluation in SIMD code).

The alternative to overloading operator?: for `std::simd` is to keep using `simd::select`:

```

1 template <class T> T abs(T x) {
2     return select(x < 0, -x, x);
3 }

```

This function is ...(cf. ~~Tony~~Table 3)

- less intuitive, since the function introduces an additional spelling to what `?:` provides.
- even less intuitive, since the arguments appear to be arbitrarily ordered (comma doesn't convey semantics such as `?` and `:` do)³.
- harder to use in generic code: If T is a built-in type, the `select` function will not be found via ADL; consequently, user code must spell out `return std::simd::select(x < 0, -x, x)` and include the `<simd>` header. This is annoying and the namespace qualification is easily

³ To make things worse x86 intrinsics/instructions for blend operations use all possible argument permutations.

forgotten since ADL works fine for `simd` arguments. Also, it makes refactoring code from built-in arithmetic types to `std::simd` unnecessarily harder.

- a bad choice for boolean conditions, because it cannot support lazy evaluation.

It is not possible (and not a good idea to extend the language in such a way, in my opinion) to overload `if` statements and iteration statements for non-boolean conditions. Thus, to support any “collection of `bool`”-like type in conditional expressions using built-in syntax, the conditional operator is the only reasonable candidate.

before	with D0917R4
<pre>1 #include <simd> 2 3 template <class T> void f(T x) 4 { 5 if constexpr (my_is_simd_vec_concept<T>) 6 return select(x < 0, g(x), x); 7 else 8 return x < 0 ? g(x) : x; 9 }</pre>	<pre>1 2 3 template <class T> void f(T x) 4 { 5 6 7 return x < 0 ? g(x) : x; 8 9 }</pre>

Before/After Table 3: Supporting `simd` and deferred evaluation for boolean `x < 0`

5.4

EXPRESSION TEMPLATES AND EMBEDDED DOMAIN SPECIFIC LANGUAGES

Embedded domain specific languages in C++ often redefine operators for user-defined types to create a new language embedded into C++. Having the conditional operator available makes C++ more versatile for such uses.

As existing practice consider Boost.YAP: “The main objective of Boost.YAP is to be an easy-to-use and easy-to-understand library for using the expression template programming technique.”⁴ YAP “defines a 3-parameter function `if_else()` that acts as an analogue to the ternary operator `(?:)`, since the ternary operator is not user-overloadable.”⁵

5.5

EXISTING PRACTICE

GCC supports the conditional operator with vector built-ins⁶, implementing a blend operation as discussed in Section 5.3. OpenCL uses the conditional operator for blending operations [1]. Allowing overloads of `operator?:` in C++ would enable users and `std::simd` to implement blend semantics with the same syntax and semantics as provided by GCC and OpenCL.

4 https://boostorg.github.io/yap/doc/html/boost_yap/rationale.html
5 https://boostorg.github.io/yap/doc/html/BOOST_YAP_USER_EX_idm15635.html
6 <https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html>

5.6

WE ARE ADDING WORKAROUNDS (P2655R3)

Xie et al. [P2655R3] specializes `common_reference_t` for `reference_wrapper` and reference types. This is clearly a workaround for `operator?:` not working where obviously a common reference type can be defined. In my prototype implementation I can simply define:

```
1 namespace std
2 {
3     template <typename T>
4     T& operator?: (bool, reference_wrapper<T>, T&) = default;
5 }
```

and that makes

```
1 static_assert(std::is_same_v<decltype(true ? std::ref(a) : a), int&>);
2 static_assert(std::is_same_v<decltype(true ? a : std::ref(a)), int&>);
```

pass.

6

PROPOSAL

In this part I'll introduce and explain my proposal.

6.1

DECLARATION AND DEFINITION OF OPERATOR?:

1. `operator?:` must have exactly three arguments, none of them defaulted, and no variadic arguments.
2. `operator?:` must be a non-member function (hidden friends are okay).
3. At least one of the three argument types must be a user-defined type.
4. Given a `operator?: (C c, T x, U y)` declaration, the expression `c ? x : y` must not match a built-in conditional operator.
5. Given a template `operator?: (auto, auto, auto)` declaration, a specialization that matches an existing built-in conditional operator is a substitution failure.
6. If the type of the first argument is `bool`, `operator?:` can be defaulted (`= default`), otherwise defaulting the operator is ill-formed.
7. For a defaulted conditional operator, `static_cast` to the declared return type of both arguments (as declared) must be well-formed. For example, `int& operator?: (bool, A&, int&) = default` requires that `static_cast<int&>(declval<A&>())` is well-formed.
8. Explicitly deleting an `operator?:` overload via `= delete` is possible (as for all other function overload sets), but it is ill-formed to delete an existing built-in conditional operator.

6.2

CALLING THE CONDITIONAL OPERATOR — OVERLOAD RESOLUTION

1. If a built-in conditional operator is valid, that operator is used and *no name lookup* for user-defined `operator?:` is performed.
2. Otherwise, perform the usual name lookup (including ADL).
3. Add reversed candidates if the types of `arg2` and `arg3` differ (non-`bool arg1`) and if applying unary `!` to `arg1` is well-formed. The reverse candidates use the type of `!arg1` rather than `arg1`.
4. Perform overload resolution. An additional tie-break is needed if the remaining two candidates are the normal and reversed forms of the same `operator?:` and are tied under partial ordering. In that case the normal (not-reversed) form wins.
5. If the winner is reversed, the `c ? a : b` expression is rewritten to `!c ? b : a`.

The first rule makes this proposal a non-breaking change. If the C++26 conditional operator applies, existing code continues to work with the exact same semantics. This is a reasonable approach because the existing conditional operator has always been providing auto-generated defaulted `operator?:` overloads, which we don't want to compete with: There is a single reasonable semantic for `?:` and that is already correctly implemented by the existing machinery.

If the built-in conditional operator does not apply, existing code would be ill-formed and so there is nothing to break. Importantly, avoiding name lookup avoids ADL, which in turn avoids template instantiation, which can hit the corner case that could otherwise lead to well-formed code becoming ill-formed (see [P2600R0])⁷.

6.3

SYNTHESIZING A DEFAULTED CONDITIONAL OPERATOR

If the expression `c ? <arg2-expression> : <arg3-expression>` is not a valid built-in conditional operator expression, and lookup with subsequent overload resolution finds a defaulted `operator?:` declaration with the declared return type `R` and the declared argument types `bool`, `A`, and `B`, then the expression is equivalent to:

```
1 c ? [] (A a) -> R { return static_cast<R>(a); } (<arg2-expression>)
2 : [] (B b) -> R { return static_cast<R>(b); } (<arg3-expression>)
```

which rewrites the expression to use the existing built-in conditional operator. Consequently, a defaulted conditional operator still implements lazy evaluation. This is analogous to defaulted default constructors allowing trivial construction.

6.4

SEMANTIC REQUIREMENT ON USER-DEFINED OPERATOR?:

For a not-defaulted `operator?: (c, a, b)` function, the implementation can assume that the result of `c ? a : b` and `!c ? b : a` are equivalent (if the expression `!c` is well-formed). A definition like

```
1 R operator?: (C c, A a, B b) { return a; }
```

breaks the semantic contract of a conditional operator. Without this assumption it would be incorrect to implement automatic rewriting from `c ? a : b` to `!c ? b : a`. This assumption is analogous to the assumption that `a == b` and `b == a` are equivalent.

⁷ The ADL issue was initially thought to be blocking this paper. Not anymore.

6.5

CALLING A USER-DEFINED CONDITIONAL OPERATOR

If the selected user-defined conditional operator is not defaulted, then a normal function call is emitted. (This implies not-lazy evaluation.)

6.6

TAKING THE ADDRESS OF A CONDITIONAL OPERATOR

A user-defined conditional operator is still an entity like all other user-defined operators, which can be spelled out as `operator?::`. It is thus possible to take the address of a conditional operator. Example:

```

1 R operator?:(bool, T, U) = default;
2
3 void f(T t, U u) {
4     R (*p1)(bool, T, U) = &operator?::; // OK
5     R x = p1(true, t, u); // x = static_cast<R>(t); no lazy eval!
6     R (*p2)(bool, U, T) = &operator?::; // ERROR
7 }
```

This is consistent with `operator==`:

```

1 bool operator==(T, U);
2
3 void f() {
4     bool (*p1)(T, U) = &operator==; // OK
5     bool (*p2)(U, T) = &operator==; // ERROR
6 }
```

Note that it is allowed to take the address of a *defaulted* `operator?::`. This implies that without `inline`, a defaulted `operator?::` emits a symbol.

6.7

DEFAULTED CONDITIONAL OPERATOR TEMPLATES

A defaulted `R operator?:(bool, T, U)` template always has an implicit

```

1 requires { static_cast<R>(std::declval<T>()); static_cast<R>(std::declval<U>()); }
```

constraint⁸. This constraint makes the template match the invariant enforced by refusing impossible non-template defaulted condition operator declarations:

- A defaulted `operator?::` non-template enforces the invariant ("R must be a genuine common type of the two branches") as a hard error at the declaration. Such an `operator?::` simply cannot exist.
- A defaulted `operator?::` template enforces the same invariant, but at the instantiation, as a substitution failure.

A defaulted `operator?::` *must never exist* for impossible conversions (template or not). This is important for `std::common_type`, which is defined via `?::`, so it inherits whatever failure mode `?::` has for a non-common pair.

- With the constraint: non-common pair $\Rightarrow$ SFINAE $\Rightarrow$ `common_type` correctly reports "no common type" by not having a `type` member.

⁸ This is equivalent to `explicitly-convertible-to<T, R> && explicitly-convertible-to<U, R>`, using the concept defined in [simd.expos].

- Without the constraint: non-common pair $\Rightarrow$ hard error at instantiation $\Rightarrow$ `common_type` becomes a diagnostic in the middle of a trait.

Adding the implicit constraint is the correct choice because the operator is defaulted and thus the body is synthesized. The constraint matches the body, implementing a hypothetical `requires(auto)`. For not defaulted conditional operator overloads the implementation makes no assumptions about the body of the function and therefore does not add any implicit constraints.

6.8

= DELETE

A conditional operator overload can be declared as deleted. This is not only a consistency choice. A deleted conditional operator is useful to subtract from an overload set. The user-provided diagnostic string on `delete` also allows better feedback from library implementers.

6.9

OBSERVATIONS

- A single

```
1 R operator?:(bool, T, U) = default;
```

declaration is sufficient to produce the same common type `R` for both argument orders `T, U` and `U, T`.

- `chrono`'s inconsistent “the conditional operator works *most* of the time” can now be resolved:

```
1 namespace std {
2     template <typename Rep1, typename Period1, typename Rep2, typename Period2>
3         common_type_t<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>>
4         operator?:(bool, chrono::duration<Rep1, Period1>,
5                   chrono::duration<Rep2, Period2>) = default;
6 }
7
8 std::chrono::duration<int, std::ratio<1, 3>> t1 {1};
9 std::chrono::duration<int, std::ratio<2, 5>> t2 {1};
10 auto t3 = true ? t1 : t2; // valid now, impossible with C++26
```

7

IMPLEMENTATION EXPERIENCE

This proposal is fully implemented in a branch of GCC trunk. The testcases I use for verification are in appendix A of this paper. The implementation can be found at

https://github.com/mattkretz/gcc/tree/P0917_overloadable_conditional_operator

and can be tested on Compiler Explorer as soon as the branch build lands. Please try to poke holes into it and let me know.

8

DISCUSSION

This section collects contentious or open design issues.

8.1

MEMBER AND/OR NON-MEMBER?

For all member operators, the first operand is omitted in the signature and implicitly uses `*this`. Consequently, a conditional operator as member of a class must be a member of the condition type. Since there does not appear to be a good use case and hidden friends can be used for restricting ADL, this paper proposes to *allow only non-member conditional operator overloads*. However, for consistency with all other operators, we might want to allow it nevertheless.

9

WORDING

TBD (after the design is confirmed).

A

TEST CASES

The following test cases all pass on my fork of GCC trunk.

```

1  struct mask {
2      bool v[4];
3      friend constexpr mask
4      operator!(mask m) {
5          for (int i = 0; i < 4; ++i)
6              m.v[i] = !m.v[i];
7          return m;
8      }
9  };
10
11 namespace foo
12 {
13     template <typename T = int>
14         struct vec { T v[4]; };
15
16     // found via ADL
17     template <typename T>
18         vec<T>
19         operator?: (const mask& m, vec<T> a, const vec<T>& b)
20         {
21             for (int i = 0; i < 4; ++i)
22                 a.v[i] = m.v[i] ? a.v[i] : b.v[i];
23             return a;
24         }
25
26     // found via ADL
27     vec<double>
28     operator?: (const mask& m, const vec<int>& a, vec<double> b)
29     {
30         for (int i = 0; i < 4; ++i)
31             b.v[i] = m.v[i] ? a.v[i] : b.v[i];
32         return b;
33     }
34 }
35
36 using foo::vec;
37

```

```

38 template <typename T>
39     bool
40     all_eq(const vec<T>& a, const vec<T>& b)
41     {
42         for (int i = 0; i < 4; ++i)
43             if (a.v[i] != b.v[i])
44                 return false;
45         return true;
46     }
47
48 // call the vec<T> template
49 void
50 test0()
51 {
52     const vec x = {1, 2, 3, 4};
53     const vec y = {5, 6, 7, 8};
54     {
55         const mask m = {true, false, false, true};
56         const vec expect = {1, 6, 7, 4};
57         const vec z = m ? x : y;
58         if (!all_eq(z, expect))
59             __builtin_abort ();
60     }
61     {
62         const mask m = {false, false, true, true};
63         const vec expect = {5, 6, 3, 4};
64         const vec z = m ? x : y;
65         if (!all_eq(z, expect))
66             __builtin_abort ();
67     }
68 }
69
70 // call the vec<int>, vec<double> overload
71 void
72 test1()
73 {
74     const vec x = {1, 2, 3, 4};
75     const vec y = {5., 6., 7., 8.};
76     {
77         const mask m = {true, false, false, true};
78         const vec expect0 = {1., 6., 7., 4.};
79         const vec z0 = m ? x : y;
80         if (!all_eq(z0, expect0))
81             __builtin_abort ();
82
83         // call with implicit arg2/arg3 reversal and !m
84         const vec expect1 = {5., 2., 3., 8.};
85         const vec z1 = m ? y : x;
86         if (!all_eq(z1, expect1))
87             __builtin_abort ();
88     }
89     {
90         const mask m = {false, false, true, true};
91         const vec expect0 = {5., 6., 3., 4.};

```

```

92     const vec z0 = m ? x : y;
93     if (!all_eq(z0, expect0))
94         __builtin_abort ();
95
96     // call with implicit arg2/arg3 reversal and !m
97     const vec expect1 = {1., 2., 7., 8.};
98     const vec z1 = m ? y : x;
99     if (!all_eq(z1, expect1))
100         __builtin_abort ();
101 }
102 }
103
104 // this has higher precedence over the reversed candidate from the overload
105 // above
106 // Note the different namespace.
107 vec<double>
108 operator?:(const mask& m, vec<double> b, const vec<int>& a)
109 {
110     // gives the "wrong" result to ensure we call the right overload
111     for (int i = 0; i < 4; ++i)
112         b.v[i] = m.v[i] ? a.v[i] : b.v[i];
113     return b;
114 }
115
116 void
117 test2()
118 {
119     const vec x = {1, 2, 3, 4};
120     const vec y = {5., 6., 7., 8.};
121     {
122         const mask m = {true, false, false, true};
123         const vec expect0 = {1., 6., 7., 4.};
124         const vec z0 = m ? x : y;
125         if (!all_eq(z0, expect0))
126             __builtin_abort ();
127
128         // call with implicit arg2/arg3 reversal and !m
129         const vec z1 = m ? y : x;
130         if (!all_eq(z1, expect0))
131             __builtin_abort ();
132     }
133     {
134         const mask m = {false, false, true, true};
135         const vec expect0 = {5., 6., 3., 4.};
136         const vec z0 = m ? x : y;
137         if (!all_eq(z0, expect0))
138             __builtin_abort ();
139
140         // call with implicit arg2/arg3 reversal and !m
141         const vec z1 = m ? y : x;
142         if (!all_eq(z1, expect0))
143             __builtin_abort ();
144     }
145 }

```

```

146
147 int
148 main ()
149 {
150     test0();
151     test1();
152     test2();
153     return 0;
154 }

```

Listing 1: Test 1

```

1 struct mask {
2     bool v[4];
3 };
4
5 template <typename T = int>
6     struct vec { T v[4]; };
7
8 // can't reverse this one because !mask is not defined
9 vec<double>
10 operator?: (const mask& m, const vec<int>& a, vec<double> b);
11
12 void
13 test1()
14 {
15     const mask m = {true, false, false, true};
16     const vec x = {1, 2, 3, 4};
17     const vec y = {5., 6., 7., 8.};
18     const vec z1 = m ? y : x; // { dg-error "could not convert" }
19 }
20
21 // explicit reversed overload
22 vec<double>
23 operator?: (const mask& m, vec<double> b, const vec<int>& a);
24
25 void
26 test2()
27 {
28     const mask m = {true, false, false, true};
29     const vec x = {1, 2, 3, 4};
30     const vec y = {5., 6., 7., 8.};
31     const vec z1 = m ? y : x; // well-formed now
32 }

```

Listing 2: Test 1b

```

1 struct A {};
2
3 struct S {
4     int operator?: (A, A); // { dg-error "must not be a member function" }
5 };

```

Listing 3: Test 2

```

1 struct mask { bool v; };
2 struct vec { int v; };
3
4 vec operator?: (mask m, vec a); // { dg-error "must have exactly three arguments" }

```

Listing 4: Test 3

```

1 struct mask { bool v; };
2 struct vec { int v; };
3
4 vec operator?: (mask m, vec a, vec b = vec{0}); // { dg-error "cannot have default arguments" }

```

Listing 5: Test 4

```

1 struct i32 { int v; };
2 struct u32 { unsigned v; };
3 struct i64 { long long v; };
4
5 constexpr i64
6 operator?: (bool c, i32 a, u32 b)
7 { return c ? i64{a.v} : i64{b.v}; }
8
9 int
10 main ()
11 {
12     {
13         i32 i = {42};
14         u32 f = {99};
15         i64 r = true ? i : f;
16         if (r.v != 42)
17             __builtin_abort ();
18     }
19     {
20         i32 i = {42};
21         u32 f = {99};
22         i64 r = false ? i : f;
23         if (r.v != 99)
24             __builtin_abort ();
25     }
26
27     // Reversed order: matched by reversed-argument lookup
28     {
29         i32 i = {42};
30         u32 f = {99};
31         i64 r = true ? f : i;
32         if (r.v != 99)
33             __builtin_abort ();
34     }
35     {
36         i32 i = {42};
37         u32 f = {99};
38         i64 r = false ? f : i;
39         if (r.v != 42)
40             __builtin_abort ();

```

```

41     }
42
43     return 0;
44 }
45
46 static_assert([] {
47     constexpr i32 i = {42};
48     constexpr u32 f = {99};
49     constexpr i64 r = false ? f : i;
50     return r.v == 42;
51 });

```

Listing 6: Test 6

```

1  struct mask { bool v[4]; };
2
3  template <typename T = int>
4      struct vec {
5          T v[4];
6
7          friend constexpr vec<T>
8          operator?: (const mask& m, vec a, const vec& b)
9          {
10             for (int i = 0; i < 4; ++i)
11                 a.v[i] = m.v[i] ? a.v[i] : b.v[i];
12             return a;
13         }
14     };
15
16 template <typename T>
17     bool
18     all_eq(const vec<T>& a, const vec<T>& b)
19     {
20         for (int i = 0; i < 4; ++i)
21             if (a.v[i] != b.v[i])
22                 return false;
23         return true;
24     }
25
26 int
27 main ()
28 {
29     const vec x = {1, 2, 3, 4};
30     const vec y = {5, 6, 7, 8};
31     {
32         const mask m = {true, false, false, true};
33         const vec expect = {1, 6, 7, 4};
34         const vec z = m ? x : y;
35         if (!all_eq(z, expect))
36             __builtin_abort ();
37     }
38     {
39         const mask m = {false, false, true, true};
40         const vec expect = {5, 6, 3, 4};
41         const vec z = m ? x : y;

```

```

42     if (!all_eq(z, expect))
43         __builtin_abort ();
44 }
45
46 return 0;
47 }

```

Listing 7: Test 7

```

1  template <typename T, typename U> T operator?:(bool, T, U) = default;
2
3  auto test(int a, double* b)
4  {
5      // does not consider user-defined operator?:
6      return true ? a : b; // { dg-error "operands to .* have different types" }
7  }
8
9  struct A { int x; };
10 struct B : A {};
11
12 auto test2(A a, B b)
13 {
14     // does not consider user-defined operator?: (built-in)
15     return true ? a : b;
16 }
17
18 struct C
19 {
20     int x;
21     C() = default;
22     C(A);
23 };
24
25 void test3(A a, C c)
26 {
27     // uses built-in, not user-defined operator?:
28     auto r = true ? a : c;
29     static_assert(__is_same(decltype(r), C), "");
30 }
31
32 struct D
33 {
34     int x;
35     D() = default;
36     explicit D(A);
37 };
38
39 void test4(A a, D d)
40 {
41     // instantiates template operator?: above (no built-in exists)
42     auto r = true ? d : a;
43     static_assert(__is_same(decltype(r), D), "");
44
45     // use the reversed operator?: template
46     auto r2 = true ? a : d;

```

```

47     static_assert(__is_same(decltype(r2), D), "");
48 }

```

Listing 8: Test 8

```

1  struct A
2  {
3      constexpr explicit
4      operator int() { return 1; }
5  };
6
7  struct B
8  {
9      constexpr explicit
10     operator int() { return 2; }
11 };
12
13 template <typename T, typename U> int operator?:(bool, T, U) = default;
14
15 int test(A a, B b)
16 {
17     auto r = true ? a : b;
18     static_assert(__is_same(decltype(r), int), "");
19     return r;
20 }

```

Listing 9: Test 8b

```

1  struct i32 { int v; };
2  struct u32 { unsigned v; };
3  struct i64 {
4      unsigned long long v;
5      i64(auto x) : v(x.v) {}
6  };
7
8  i64 operator?: (bool, i32, u32) = default;
9
10 int
11 main ()
12 {
13     {
14         i32 i = {42};
15         u32 f = {99};
16         i64 r = true ? i : f;
17         if (r.v != 42)
18             __builtin_abort ();
19     }
20     {
21         i32 i = {42};
22         u32 f = {99};
23         i64 r = false ? i : f;
24         if (r.v != 99)
25             __builtin_abort ();
26     }
27 }

```

```

28 // Reversed order: matched by reversed-argument lookup
29 {
30     i32 i = {42};
31     u32 f = {99};
32     i64 r = true ? f : i;
33     if (r.v != 99)
34         __builtin_abort ();
35 }
36 {
37     i32 i = {42};
38     u32 f = {99};
39     i64 r = false ? f : i;
40     if (r.v != 42)
41         __builtin_abort ();
42 }
43
44 return 0;
45 }

```

Listing 10: Test 9

```

1 struct B;
2
3 struct A
4 {
5     int x;
6     constexpr A (B);
7     constexpr A (int v) : x(v) {}
8 };
9
10 struct B
11 {
12     int x;
13     constexpr B (A);
14     constexpr B (int v) : x(v) {}
15 };
16
17 constexpr A::A (B f) : x(f.x) {}
18 constexpr B::B (A f) : x(f.x) {}
19
20 A operator?: (bool, A, B) = default;
21
22 int
23 main ()
24 {
25     // true branch: A -> C
26     {
27         constexpr A a = 42;
28         constexpr B b = 99;
29         if ((true ? a : b).x != 42)
30             __builtin_abort ();
31
32         static_assert((true ? a : b).x == 42);
33     }
34 }

```

```

35 // false branch: B -> C
36 {
37     constexpr A a = {42};
38     constexpr B b (99);
39     if ((false ? a : b).x != 99)
40         __builtin_abort ();
41
42     static_assert((false ? a : b).x == 99);
43 }
44
45 return 0;
46 }

```

Listing 11: Test 10

```

1 struct Base { int x; };
2 struct Derived : Base { int y; };
3
4 Base operator?: (bool, Base, Derived) = default; // { dg-error "built-in operator\?: already exists" }
5
6 Base operator?: (bool, Base, Derived) // { dg-error "built-in operator\?: already exists" }
7 { return Base{0}; };
8
9 namespace foo
10 {
11     struct A { int x; };
12     struct B : A {};
13
14     template <typename T, typename U>
15         T operator?: (bool, T, U) = default; // no error
16
17     A test(A a, B b)
18     { return true ? a : b; } // uses built-in, ignores the template
19 }

```

Listing 12: Test 11

```

1 struct mask { int v; explicit operator bool() const { return v != 0; } };
2
3 struct Vec2i { int x, y; };
4 struct Vec2f { float x, y; };
5 struct Vec2d
6 {
7     double x, y;
8     Vec2d (Vec2i v) : x(v.x), y(v.y) {}
9     explicit Vec2d (Vec2f v) : x(v.x), y(v.y) {}
10 };
11
12 Vec2d operator?: (bool, Vec2i, Vec2f) = default;
13
14 int
15 main ()
16 {
17     // Non-boolean condition via implicit conversion + defaulted operator
18     {

```

```

19     mask m = {1};
20     Vec2i a = {1, 2};
21     Vec2f b = {3.0f, 4.0f};
22     Vec2d z = m ? a : b;
23     if (z.x != 1 || z.y != 2)
24         __builtin_abort ();
25 }
26
27 {
28     mask m = {0};
29     Vec2i a = {1, 2};
30     Vec2f b = {3.0f, 4.0f};
31     Vec2d z = m ? a : b;
32     if (z.x != 3 || z.y != 4)
33         __builtin_abort ();
34 }
35
36 return 0;
37 }

```

Listing 13: Test 12

```

1 struct A
2 {
3     int x;
4     explicit operator bool() const { return x != 0; }
5 };
6
7 struct B { float x; };
8
9 struct Res
10 {
11     double x;
12     Res (A v) : x(v.x) {}
13     Res (B v) : x(v.x) {}
14 };
15
16 Res operator?: (bool, B, A) = default;
17
18 int
19 main ()
20 {
21     {
22         A a = {3};
23         B b = {99.0f};
24         Res z = a ?: b;
25         if (z.x != 3)
26             __builtin_abort ();
27     }
28     {
29         A a = {0};
30         B b = {5.0f};
31         Res z = a ?: b;
32         if (z.x != 5)
33             __builtin_abort ();

```

```

34     }
35
36     return 0;
37 }

```

Listing 14: Test 13

```

1  static int true_evaluated = 0;
2  static int false_evaluated = 0;
3
4  struct S1 { int v; };
5  struct S2 { int v; };
6  struct S12
7  {
8      int v;
9      constexpr S12 (S1 s) : v(s.v) {}
10     constexpr S12 (S2 s) : v(s.v) {}
11 };
12
13 // Defaulted operator?: -- should use built-in COND_EXPR internally,
14 // preserving lazy evaluation.
15 S12 operator?: (bool, S1, S2) = default;
16
17 // Track side effects in branch expressions
18 S1
19 get_true_branch ()
20 {
21     true_evaluated++;
22     return S1{42};
23 }
24
25 S2
26 get_false_branch ()
27 {
28     false_evaluated++;
29     return S2{99};
30 }
31
32 int
33 main ()
34 {
35     {
36         S12 result = true ? get_true_branch () : get_false_branch ();
37         if (true_evaluated != 1 || false_evaluated != 0)
38             __builtin_abort ();
39         if (result.v != 42)
40             __builtin_abort ();
41     }
42     {
43         S12 result = false ? get_true_branch () : get_false_branch ();
44         if (true_evaluated != 1 || false_evaluated != 1)
45             __builtin_abort ();
46         if (result.v != 99)
47             __builtin_abort ();
48     }

```

```

49
50     return 0;
51 }

```

Listing 15: Test 14

```

1  struct S1 { int v; };
2  struct S2 { int v; };
3
4  S1 operator?: (int, S1, S2) = default; // { dg-error "first parameter of defaulted.*must be .bool." }

```

Listing 16: Test 15

```

1  struct B;
2
3  struct A
4  {
5      int x;
6      A (int v) : x(v) {}
7      explicit A (const B&);
8  };
9
10 struct B
11 {
12     int x;
13     B (int v) : x(v) {}
14 };
15
16 struct C
17 {
18     int x;
19     C (int v) : x(v) {}
20     explicit C (const B& b) : x(b.x) {}
21 };
22
23 A::A (const B& b) : x(b.x) {}
24
25 A operator?: (bool, A, B) = default;
26 C operator?: (bool, C, B) = default;
27
28 int
29 main ()
30 {
31     {
32         using fn_t = A (*)(bool, A, B);
33         fn_t f = &operator?:;
34
35         A a = 42;
36         B b = 99;
37
38         A r1 = f (true, a, b);
39         if (r1.x != 42)
40             __builtin_abort ();
41
42         A r2 = f (false, a, b);

```

```

43     if (r2.x != 99)
44         __builtin_abort ();
45     }
46     {
47         using fn_t = C (*)(bool, C, B);
48         fn_t f = &operator?::;
49
50         C c = 42;
51         B b = 99;
52
53         C r1 = f (true, c, b);
54         if (r1.x != 42)
55             __builtin_abort ();
56
57         C r2 = f (false, c, b);
58         if (r2.x != 99)
59             __builtin_abort ();
60     }
61
62     return 0;
63 }

```

Listing 17: Test 18

```

1  struct A
2  {
3      int x;
4      explicit operator int&() & { return x; }
5      explicit operator const int&() const & { return x; }
6  };
7
8  int& operator?:(bool, A&, int&) = default;
9
10 int
11 main ()
12 {
13     bool failed = false;
14     {
15         A a = {1};
16         int b = 2;
17         int& c = false ? a : b;
18         if (c != 2)
19             {
20                 __builtin_printf("1");
21                 failed = true;
22             }
23         if (&c != &b)
24             {
25                 __builtin_printf("2");
26                 failed = true;
27             }
28     }
29     {
30         A a = {1};
31         int b = 2;

```

```

32     int& c = true ? a : b;
33     if (c != 1)
34     {
35         __builtin_printf("3");
36         failed = true;
37     }
38     if (&c != &a.x)
39     {
40         __builtin_printf("4");
41         failed = true;
42     }
43 }
44
45 return failed ? -1 : 0;
46 }

```

Listing 18: Test 19

```

1  -
2  struct C
3  {
4      int v;
5  };
6
7  struct A
8  {
9      int x;
10 };
11
12 struct A2
13 {
14     int x;
15 };
16
17 struct B
18 {
19     C c;
20     explicit operator C&() & { return c; }
21 };
22
23 // A has no conversion to C& - rejected at declaration.
24 C& operator?:(bool, A&, B&) = default; // { dg-error "cannot convert.*A.*to.*C" }

```

Listing 19: Test 20

```

1  struct C
2  {
3      int v;
4  };
5
6  struct A
7  {
8      C c;
9      explicit operator C&() & { return c; }
10 };

```

```

11
12 struct B
13 {
14     C c;
15     explicit operator C&() & { return c; }
16 };
17
18 C& operator?:(bool, A&, B&) = default;
19
20 int
21 main ()
22 {
23     bool failed = false;
24     A a; a.c.v = 42;
25     B b; b.c.v = 99;
26
27     C& r1 = true ? a : b;
28     if (r1.v != 42)
29     {
30         __builtin_printf("1");
31         failed = true;
32     }
33     if (&r1 != &a.c)
34     {
35         __builtin_printf("2");
36         failed = true;
37     }
38
39     C& r2 = false ? a : b;
40     if (r2.v != 99)
41     {
42         __builtin_printf("3");
43         failed = true;
44     }
45     if (&r2 != &b.c)
46     {
47         __builtin_printf("4");
48         failed = true;
49     }
50
51     return failed ? -1 : 0;
52 }

```

Listing 20: Test 21

```

1 struct C
2 {
3     int v;
4 };
5
6 struct A
7 {
8     C c;
9     explicit operator C&() & { return c; }
10 };

```

```

11
12 struct B
13 {
14     C c;
15     explicit operator C&() & { return c; }
16 };
17
18 C& operator?:(bool, A&, B&) = default;
19
20 int
21 main ()
22 {
23     bool failed = false;
24     A a; a.c.v = 42;
25     B b; b.c.v = 99;
26
27     using fn_t = C&(*)(bool, A&, B&);
28     fn_t f = operator?::;
29
30     C& r1 = f(true, a, b);
31     if (r1.v != 42)
32     {
33         __builtin_printf("1");
34         failed = true;
35     }
36     if (&r1 != &a.c)
37     {
38         __builtin_printf("2");
39         failed = true;
40     }
41
42     C& r2 = f(false, a, b);
43     if (r2.v != 99)
44     {
45         __builtin_printf("3");
46         failed = true;
47     }
48     if (&r2 != &b.c)
49     {
50         __builtin_printf("4");
51         failed = true;
52     }
53
54     return failed ? -1 : 0;
55 }

```

Listing 21: Test 22

```

1 struct C
2 {
3     int v;
4 };
5
6 struct B
7 {

```

```

8   C c;
9   explicit operator C&() & { return c; }
10 };
11
12 template <typename T>
13   C& operator?:(bool, T&, B&) = default;

```

Listing 22: Test 23

```

1  struct C
2  {
3      int v = 0;
4  };
5
6  struct A
7  {
8      C c;
9
10     constexpr explicit
11     operator C&() & { return c; }
12 };
13
14 struct B
15 {
16     C c;
17
18     constexpr explicit
19     operator C&() & { return c; }
20 };
21
22 template <typename T>
23   C& operator?:(bool, T&, B&) = default;
24
25 auto test(int &a, B& b)
26 {
27     return true ? a : b; // { dg-error "have different types" }
28 }

```

Listing 23: Test 23b

```

1  struct C
2  {
3      int v = 0;
4  };
5
6  struct A
7  {
8      C c;
9
10     constexpr explicit
11     operator C&() & { return c; }
12 };
13
14 struct B
15 {

```

```

16  C c;
17
18  constexpr explicit
19  operator C&() & { return c; }
20 };
21
22  template <typename T>
23  C& operator?:(bool, T&, B&) = default;
24
25  int main()
26  {
27      A a;
28      B b;
29      (true ? a : b).v = 1;
30      (false ? a : b).v = 2;
31      (true ? b : a).v += 1;
32      (false ? b : a).v += 3;
33      return (a.c.v == 4 && b.c.v == 3) ? 0 : 1;
34  }

```

Listing 24: Test 23c

```

1  struct A
2  {
3      int x;
4      explicit operator int&() & { return x; }
5  };
6
7  int& operator?:(bool, auto, int&) = default;
8
9  void
10 test0()
11 {
12     A a{0};
13     int b = 0;
14     int& r = (true ? a : b); // { dg-error "operands to .* have different types" }
15     (void)r;
16 }
17
18 // By-value A is an rvalue; operator int&() & is lvalue-only, so the branch
19 // is not directly-initializable to int& -- rejected at declaration.
20 int& operator?:(bool, A, int&) = default; // { dg-error "cannot convert.*A.*to.*int&" }
21
22 // By-reference A& is an lvalue; operator int&() & applies, so this one is
23 // still accepted -- the fix must not over-reject.
24 int& operator?:(bool, A&, int&) = default;

```

Listing 25: Test 24

```

1  struct A
2  {
3      int x;
4      operator int&() { return x; } // non-explicit: built-in ?: is viable for (A, int)
5  };
6

```

```
7 int& operator?:(bool, A&, int&) = default; // { dg-error "already exists" }
```

Listing 26: Test 25

```
1 struct A
2 {
3     int x;
4     explicit operator int&() { return x; }
5 };
6
7 int& operator?:(bool, A&, int&) = default;
8
9 int
10 main ()
11 {
12     A a {111};
13     int b = 222;
14
15     // The selected A branch yields a reference to a.x; assign through it,
16     // leaving the int branch (b) untouched.
17     (true ? a : b) = 1;
18     if (a.x != 1 || b != 222)
19         __builtin_abort ();
20
21     // The selected int branch yields a reference to b; assign through it,
22     // leaving the A branch (a.x) untouched.
23     (false ? a : b) = 2;
24     if (a.x != 1 || b != 2)
25         __builtin_abort ();
26
27     return 0;
28 }
```

Listing 27: Test 26

```
1 #include <stdfloat>
2
3 // is_convertible_v<float16_t, float> is false while
4 // static_cast<float16_t>(float) is valid
5
6 struct X
7 {
8     explicit operator std::float16_t()
9     { return 2.f16; }
10 };
11
12 struct Y
13 {
14     explicit operator std::float16_t()
15     { return 2.f16; }
16 };
17
18 std::float16_t operator?:(bool, float, X) = default;
19
20 template <typename T>
```

```

21     std::float16_t operator?:(bool, float, T) = default;
22
23     float a = 1.f;
24
25     int
26     main()
27     {
28         auto b = true ? a : X();
29         auto c = false ? a : X();
30         if (b != 1)
31             __builtin_abort();
32         if (c != 2)
33             __builtin_abort();
34
35         b = true ? a : Y();
36         c = false ? a : Y();
37         if (b != 1)
38             __builtin_abort();
39         if (c != 2)
40             __builtin_abort();
41
42         return 0;
43     }

```

Listing 28: Test 27

B

BIBLIOGRAPHY

- [1] Khronos OpenCL Working Group. *The OpenCL Specification*. Jan. 2011. URL: <http://www.khronos.org/registry/cl/specs/openc1-1.1.pdf>.
- [P2600R0] Matthias Kretz. *P2600R0: A minimal ADL restriction to avoid ill-formed template instantiation*. ISO/IEC C++ Standards Committee Paper. 2022. URL: <https://wg21.link/p2600r0>.
- [P0037R5] John McFarlane. *P0037R5: Fixed-Point Real Numbers*. ISO/IEC C++ Standards Committee Paper. 2018. URL: <https://wg21.link/p0037R5>.
- [P1050R0] John McFarlane. *P1050R0: Fractional Numeric Type*. ISO/IEC C++ Standards Committee Paper. 2018. URL: <https://wg21.link/p1050r0>.
- [P1438R1] Bill Seymour. *P1438R1: A Rational Number Library from C++*. ISO/IEC C++ Standards Committee Paper. 2019. URL: <https://wg21.link/p1438r1>.
- [2] David Stone. *davidstone / bounded_integer — Bitbucket*. URL: https://bitbucket.org/davidstone/bounded_integer (visited on 02/26/2018).
- [3] Bjarne Stroustrup. *Stroustrup: C++ Style and Technique FAQ*. URL: http://www.stroustrup.com/bs_faq2.html#overload-dot (visited on 01/31/2018).
- [P0745R0] Herb Sutter. *P0745R0: Concepts in-place syntax syntax*. ISO/IEC C++ Standards Committee Paper. 2018. URL: <https://wg21.link/p0745r0>.

- [P2655R3] Hui Xie, S. Levent Yilmaz, and Tim Song. *common_reference_t of reference_wrapper Should Be a Reference Type*. ISO/IEC C++ Standards Committee Paper. 2023. URL: <https://wg21.link/p2655r3>.

DRAFT