Document Number:	D0917R4
Date:	2024-06-18
Reply-to:	Matthias Kretz <m.kretz@gsi.de>
Audience:	EWG
Target:	C++26

Making operator?: overloadable

ABSTRACT

This paper explores user-defined overloads of `operator?:`.

CONTENTS

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

✎ Explore: `operator?:` for contextual-bool automatically uses `common_type` and cannot be overloaded otherwise. It does deferred evaluation. Non-boolean conditions (i.e. the type of the first argument is not contextually convertible to `bool`) can be overloaded and are therefore a normal function call without deferred evaluation.

✎ `auto operator?:(T, U) = default` could mean `common_type_t<T, U> operator?:(...)`, i.e. synthesize a builtin operator using `common_tpye` for the return type.

✎ Explore: Definition of `operator?:(Cond, A, B)` automatically synthesizes `operator?:(Cond, B, A)` calling the former. (Note, there are no compatibility issues like for defaulted compares.)

✎ More motivation: P2655R1 "`common_reference_t` of `reference_wrapper` should be a reference type". P2655R1 resolves the problem via another template specialization. Instead making `operator?:` do the right thing would be more generic and an obvious fix for the type's semantics.

✎ Units example:

```
quantity<isq::speed[km / h]> speed_1 = 42 * isq::speed[m / s];
quantity<isq::speed[m / s]> speed_2 = 42 * isq::speed[km / h];
quantity_of<isq::speed> auto speed_3 = speed_1 + speed_2;

using ct = std::common_type_t<decltype(speed_1), decltype(speed_2)>;

static_assert(std::is_same_v<ct, decltype(speed_3)>);
static_assert(std::is_same_v<ct, quantity<reference<
  isq::speed, scaled_unit<mag<ratio{1, 18}>, derived_unit<si::metre,
                                                per<si::second>>>{}>{}>>);

// this cannot but should work:
quantity_of<isq::speed> auto = cond ? speed_1 : speed_2;
```

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 leess 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 INTRODUCTION

Most operators in C++ can be overloaded. The few exceptions are: `?:`, `::`, `.`, `.*`. 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.

3.1 COMMON_TYPE

It is important to consider `std::common_type` when discussing changes to the conditional operator. `common_type` was introduced to C++11 by Hinnant et al. [N2661]. Its original intent was stated as:

> In a nutshell, `common_type` is a trait that takes 1 or more types, and returns a type which all of the types will convert to. The default definition demands this conversion be implicit. However the trait can be specialized for user-defined types which want to limit their inter-type conversions to explicit, and yet still want to interoperate with the `common_type` facility.

The member type `type` for the two-type case was defined as `decltype(true ? declval<T>() : declval<U>())`[1]. More tweaks to `common_type` were applied over time so that with [N4842] the default `common_type_t<T, U>` is the first valid type of:

1. if both `decay_t<T>` and `decay_t<U>` denote comparison category type, the common comparison type;

1 `declval` came later, but it already had the same behavior

2. `decay_t<decltype(false ? declval<decay_t<T>>() : declval<decay_t<U>>())>;`
3. `decay_t<decltype(false ? declval<decay_t<T>(&)()>()() : declval<decay_t<U>(&)()>()())>`.

Consequently, if the conditional operator becomes overloadable on boolean conditions, `common_type` is affected.

3.2 Deferred evaluation

The first revision of this paper discussed how to enable deferred evaluation for non-synthesized conditional operators. But after Dennett et al. [P0927R2] tried to solve deferred evaluation in general, I realized it is an orthogonal problem to solve. This paper will therefore focus on the conditional operator and leave a deferred evaluation facility to progress independently.

4 Motivation

4.1 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 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 condition to be exclusively non-boolean or exclusively boolean;
- by not restricting to only allow deferred evaluation (this is necessary for blending).

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.

Section 4.2 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

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

Listing 1: Generic absolute value function

```
template <class T> simd<T> abs(simd<T> x) {
  simd<T> r = x;
  for (size_t i = 0; i < simd<T>::size(); ++i) {
    if ((x < 0)[i])
      r[i] = -x[i];
  }
  return r;
}
```

Listing 2: Equivalent `abs` for element-wise application of the conditional operator

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 and should not be tied to a solution for overloading the conditional operator. 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 users[2].

4.2 Blend operations

The conditional operator is a perfect match for expressing blend operations generically. To illustrate what I mean, consider Listing 1: For arithmetic types, the conditional operator chooses to either evaluate `-x` or `x`. For a type `T` that stores multiple values of arithmetic type and implements 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.

Consider `simd<T, Abi>` [N4808, §9], where a certain number (determined at compile time) of values of arithmetic type `T` are combined to a single object and operators act element-wise and concurrently. The conditional operator intuitively translates from fundamental types to `simd` types: Element-wise application of the conditional operator blends the elements of `-x` and `x` into a single `simd` object according to the `simd_mask` object (`x < 0`), equivalent to Listing 2.

The alternative to overloading `operator?:` is to use a function such as "simd blend":

```
template <class T> T abs(T x) {
  return simd_blend(x < 0, -x, x);
}
```

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

A “simd blend” function is

- less intuitive, since the name is either long or it is cryptic.
- 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 `simd_blend` function will not be found via ADL; consequently, user code must spell out `return std::experimental::simd_blend(x < 0, -x, x)` and include a relevant header to be generic. This is annoying and the namespace qualification is easily forgotten since ADL works fine for `simd` arguments.
- 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 possible candidate.

Considering cases where generality of the syntax, i.e. extension from the built-in case to user-defined types, is important, we see that all such use cases will have a type for the condition that is not contextually convertible to `bool` because the user-defined condition object stores multiple boolean states. Overloading the conditional operator is thus most interesting for stating conditional evaluation of multiple data sets without imposing an order and thus enabling parallelization.

before

```
#include <simd>

template <class T>
  void f(T x)
{
  if constexpr (std::is_simd_v<T>)
  {
    where(x < 0, x) = g(x);
    return x;
  }
  else
    return x < 0 ? g(x) : x;
}
```

with D0917R4

```
template <class T>
  void f(T x)
{
  return x < 0 ? g(x) : x;
}
```

Tony Table 1: Supporting `simd` and deferred evaluation for boolean `x < 0`

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

4.3 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]

4.4 User-defined numeric types

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`[6]. 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;
```

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

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

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).

before

```
bounded::integer<1, 100> const a = f();
bounded::integer<-3, 7> const b = g();
auto c = BOUNDED_CONDITIONAL(
           some_condition, a, b);
```

with D0917R4

```
bounded::integer<1, 100> const a = f();
bounded::integer<-3, 7> const b = g();
auto c = some_condition ? a : b;
```

Tony Table 2: bounded::integer now and with overloadable `operator?:`

before

```
template<class T, class U>
  void f(bool cond, T a, U b)
{
  if constexpr (
      is_bounded_integer<T>::value ||
      is_bounded_integer<U>::value)
    g(BOUNDED_CONDITIONAL(
      some_condition, a, b));
  else
    g(cond ? a : b);
}
```

with D0917R4

```
template<class T, class U>
  void f(bool cond, T a, U b)
{









  g(cond ? a : b);
}
```

Tony Table 3: supporting bounded::integer in a generic function

4.5 Existing practice

GCC supports the conditional operator with vector built-ins[7], implementing a blend operation as discussed in Section 4.2. 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.

5 Exploration

5.1 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 would need to be a member of the condition type. Since

7 https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

```
// most common usage of ?: with UDTs:
struct Point { float x, y, z; };
static_assert(is_same_v<Point, decltype(bool() ? Point() : Point())>);

// less common:
struct A { explicit operator bool(); };
struct B { operator float(); };
struct C { operator float(); };
using X = decltype(A() ? B() : C());  // X = float (GCC, Clang), double (ICC),
                                      //     ill-formed (MSVC)
struct D {
  operator B();
  operator float();
};
using Y = decltype(A() ? B() : D());  // Y = B
struct E;
struct F { operator E(); };
struct E { operator F(); };
using Z = decltype(A() ? F() : E());  // Z = F (MSVC), ill-formed (GCC, Clang, ICC)
```

Listing 3: Examples of the conditional operator with UDTs

```
1 struct A { operator int() const; };
2 struct B { operator float() const; };
3 template <class A, class B> struct CT { using type = decltype(A() + B()); };
4 static_assert(std::is_same_v<CT<A, B>::type, decltype(A() + B())>);
5 short operator+(A, B);
6 static_assert(std::is_same_v<CT<A, B>::type, decltype(A() + B())>);  // fails
```

Listing 4: A pitfall of overloading: `CT<A, B>` is specialized only on line 4

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*.

5.2 Can a user-defined conditional operator change existing code?

The conditional operator already works in many situations where user-defined types are used. A few examples are shown in Listing 3.

Should the user be able to define a conditional operator that takes precedence over the built-in operator? Of course, to be consistent with all other operator overloads, `operator?:` overloads will require at least one user-defined type in their signature. The examples in Listing 3 seem to motivate maximal freedom in overloading `operator?:`; but let's not use implementation divergence for motivation.

If we allow user-defined `operator?:` to be a better match than built-in `operator?:`, we open the door to situations where the return type (and value) of the same conditional operator is different at different places in the TU (such as in `https://godbolt.org/z/xMMbaE`), as is the case for all other

```cpp
struct A {};
struct B { operator int(); };
struct C { operator int(); };

void f() {
  A() + A(); // ill-formed
  B() + B(); // built-in operator+(int, int)
  B() + C(); // built-in operator+(int, int)
}

A operator+(A, A);
B operator+(B, B);
B operator+(B, int);

void g() {
  A() + A(); // operator+(A, A)
  B() + B(); // operator+(B, B)
  B() + C(); // operator+(B, int)
}
```

Listing 5: Some examples of operators overriding built-in candidates

operators already. However, note that of course `common_type` can only be specialized once (on first use). Consequently, if a user-defined conditional operator were allowed to overload combinations that the built-in operator can handle, one could construct examples where `common_type<A, B>` and `decltype(false ? A() : B())` agree in one part of the TU and disagree in the other part.[8] Note that such pitfalls are not novel. All operator (and function) overloads can already be used to construct such inconsistencies (e.g. Listing 4).

Without strong motivation, overloads of the conditional operator should not behave differently with regard to overriding built-in candidates than all other overloadable operators. Consequently if we consider the examples in Listing 5, a user-defined conditional operator would be able to override any expression that involves a user-defined type in the second and/or third operand. For the type of the first operand there can be a simple distinction early in the overload resolution process:

- Either the first operand is contextually convertible to `bool` and built-in candidates are relevant,
- or otherwise only user-defined overloads need to be considered.

In `f()` the built-in conditional operator works in all three combinations (`(A, A)`, `(B, B)`, and `(B, C)`[9]). The return type is `A`, `B`, and `int`, respectively.

8 `using X = common_type_t<A, B>; /*overload operator?:(bool, A, B)*/ static_assert(is_same_v<common_type_t<A, B>, decltype(false ? A() : B())>);`

9 except MSVC

The current specification of the conditional operator [N4842, [expr.cond]] mentions the use of overload resolution in paragraph 6:

[expr.cond]

6 Otherwise, the result is a prvalue. If the second and third operands do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is used to determine the conversions (if any) to be applied to the operands (12.4.1.2, 12.7). If the overload resolution fails, the program is ill-formed. Otherwise, the conversions thus determined are applied, and the converted operands are used in place of the original operands for the remainder of this subclause.

This paragraph is responsible for the more obscure situations, such as

```
struct A { operator int(); };
struct B { operator int(); };
int a(bool cond, A a, B b) {
  return cond ? a : b;
}
```

<++>

I believe a requirement on `operator?:` overloads to not cover cases that could be handled with the built-in operator is complicating the language for little gain and might even inhibit valid use cases. For example should Listing **??** be ill-formed, because line 8 can be handled either by the built-in or by the overloaded operator? Do we want to require users to add a constraint `!same_as<T, U>`? Can we diagnose the issue when the overload is declared, or only when the "bad" specialization is instantiated? The latter could mean both operators are equally viable and thus the expression is ambiguous. If it should not be ill-formed, should line 8 use the built-in operator or the overload?

I would prefer to make `operator?:` just as useful and dangerous as all other overloads. I.e. stay consistent with the overload resolution rules of the other operators. Note however, that there are built-in candidates for all user-defined types Consequently, if the operator overload involves a user-defined type, it is preferred over the built-in operator.

5.3 Defaulted conditional operator overload

In most scalar cases (boolean condition), the implementation of the conditional operator is trivial (i.e. return either `b` or `c`, depending on `bool(a)` for `a ? b : c`). Consequently the only important information supplied by the developer writing the overload is the return type (common type for `b` and `c`). Thus defaulting the operator appears like a logical solution. Listing 6 shows a few use cases that should be supported.

In addition consider value categories of the built-in conditional operator:

```
struct A { operator int&(); };
struct B { operator int&(); };
int& operator?:(bool, A&, B&) = default;
void a(bool cond, A& a, B& b) {
  int &x = cond ? a : b;
}
```

Listing 6: Use cases a defaulted conditional operator should support.

```
struct A {};
struct B {
  B(const B&);
  B(B&&);
  B(const A&);
  B(A&&);
};

B f1(bool c, A a, B b) {
  return c ? a : b; // B(const A&) or B(const B&)
}
B f2(bool c, A a, B b) {
  return c ? std::move(a) : std::move(b); // B(A&&) or B(B&&)
}
B f3(bool c, A a, B b) {
  return c ? a : std::move(b); // B(const A&) or B(B&&)
}
B f4(bool c, A a, B b) {
  return c ? std::move(a) : b; // B(A&&) or B(const B&)
}
```

Matching this behavior would require four overloads or forwarding references for every conditional operator one wants to overload (e.g. if the conversion constructors were `explicit`):

```
template <class T>
  concept contextual_boolean = std::is_constructible_v<bool, T>;

inline B operator?:(contextual_boolean auto c, const A& a, const B& b) {
  return c ? B(a) : b;
}
inline B operator?:(contextual_boolean auto c, const A& a, B&& b) {
  return c ? B(a) : std::move(b);
}
inline B operator?:(contextual_boolean auto c, A&& a, const B& b) {
  return c ? B(std::move(a)) : b;
}
inline B operator?:(contextual_boolean auto c, A&& a, B&& b) {
  return c ? B(std::move(a)) : std::move(b);
```

```
}
```

This is verbose and hard to get right. Forgetting to use a concept for allowing explicitly convertible to `bool` arguments for the condition is also highly likely.

Oh, I forgot: we need to handle `c ? b : a` as well. We need to add another four overloads.

And all this only to say: "when you have to choose one of `A` or `B`, use `B` as the type". This is a strong indication that users should not implement the conditional operator themselves. The missing operators (i.e. those cases not covered by the built-in operator) should be synthesized by the compiler. Note that if the compiler synthesizes the operator, *it is possible to also support deferred evaluation*.

When exploring the best way to express the intent to synthesize conditional operator overloads keep in mind:

- There are use cases for non-boolean conditions, which cannot be synthesized.
- There already exist many explicit specializations of `std::common_type`.

Ideas:

1. synthesize from `common_type`
2. `R operator?:(bool, T, U) = default`
3. `R operator?:(T, U) = default`
4. `operator?:(T, U) = default` with implicit return type `common_type_t<T, U>`

I will use the following criteria to judge each idea:

- opt-in vs. opt-out: Does a user have enough control to decide whether his type supports or doesn't support expressions involving the conditional operator?
- discoverability: How easy can a C++ user understand where the operator comes from?
- contextual conversion to `bool` works and is impossible/hard to get wrong
- deferred evaluation works
- DRY[10]: Considering the connection of `common_type` and the conditional operator, how much code needs to be written and is it easy/fool-proof to get consistent results?

10 don't repeat yourself

5.3.1 synthesize from common_type

Whenever a boolean conditional expression choosing between expressions of type `T` and `U` does not match the built-in conditional operator, use `std::common_type_t<T, U>` as the resulting type of the expression. If `std::common_type<T, U>` has no member type `type`, then the program is ill-formed. Otherwise, the expression `c ? a : b` is turned into

```
c ? std::common_type_t<decltype(a), decltype(b)>(a)
  : std::common_type_t<decltype(a), decltype(b)>(b)
```

. This transformation leads to the correct constructor calls (i.e. move vs. copy) and implicitly enables deferred evaluation[11]. Also, as long as `common_type_t<T, U>` and `common_type_t<U, T>` are the same type, the expressions `c ? a : b` and `!bool(c) ? b : a` are equivalent.

Obviously, this choice would affect existing code without requiring an opt-in. But since its strictly an addition, i.e. expressions that were ill-formed are now well-formed, the only relevant impact is SFINAE related, which we happily ignore. However, opting out from the operator overload when an explicit specialization of `common_type` exists would require a deleted operator overload like:

```
common_type_t<T, U> operator?:(bool, T, U) = delete;
```

This comes with questions like:

- Does it delete non-`bool` expressions?
- Does it delete `U`, `T` as well, or does it require an extra deletion?
- Why can I delete an operator that I cannot define/default?

Synthesizing from `common_type` thus gets the following rating:

criteria	result
opt-in vs. opt-out	? opt-in via explicit specialization of `common_type`; no opt-out or explicit deletion if `common_type` is specialized
discoverability	✘ no declaration/definition of the operator overload; connection from `common_type` to synthesizing `operator?:` must be learned
contextual `bool`	✔
deferred eval	✔
DRY	✔ `common_type` and `operator?:` are forced to be consistent with a single definition of the common type.

5.3.2 R operator?:(bool, T, U) = default

A discoverable approach that is consistent with other defaulted operator overloads spells out the operator signature and explicitly defaults it. Thus, the conditional operator is only synthesized on

11 because the transformed code uses the built-in conditional operator

explicit opt-in. `R operator?:(bool, T, U) = default;` would create a defaulted (trivial?) conditional operator overload that is used whenever an expression

```
cond ? a : b
```

matches the following:

- `is_constructible_v<bool, decltype(cond)>` is `true`, and either
- `remove_cvref_t<decltype(a)>` is `T` and `remove_cvref_t<decltype(b)>` is `U`
- or `remove_cvref_t<decltype(a)>` is `U` and `remove_cvref_t<decltype(b)>` is `T`.

If it matches, the conditional expression is rewritten as `cond ? R(a) : R(b)`. This calls the correct constructors of `R` (copy vs. move) and enables deferred evaluation.

If the operator overload is defined as a template such as

```
template <class T, std::integral U>
R operator?:(bool, X<T>, X<U>) = default;
```

The defaulted 3-argument operator overload thus gets the following rating:

criteria	result
opt-in vs. opt-out	✔
discoverability	✔
contextual `bool`	✔ ? Unless the proposal to interpret `bool` in the signature as contextually convertible to `bool` is rejected.
deferred eval	✔
DRY	? Consistent if only the conditional operator is overloaded. Opt-out from consistency via explicit `common_type` specialization.

5.3.3 R operator?:(T, U) = default

Since defaulting the conditional operator is only allowed for boolean conditions, spelling out the type of the first parameter is just boilerplate. Omitting the type makes the definition of the overload more concise and resolves the mismatch between implicitly and explicitly convertible to `bool`. Thus

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

transforms the expression

```
cond ? a : b
```

into

```
cond ? R(a) : R(b)
```

if

- `is_constructible_v<bool, decltype(cond)>` is `true`, and either
- `remove_cvref_t<decltype(a)>` is `T` and `remove_cvref_t<decltype(b)>` is `U`
- or `remove_cvref_t<decltype(a)>` is `U` and `remove_cvref_t<decltype(b)>` is `T`.

The defaulted 2-argument operator overload thus gets the following rating:

criteria	result
opt-in vs. opt-out	✔
discoverability	✔ It is less confusing than spelling out `bool`, which is interpreted as contextually convertible to `bool`— inconsistent with the rest of C++. ? The conditional operator has three arguments, using only two parameters in the definition may seem confusing.
contextual `bool`	✔
deferred eval	✔
DRY	? Consistent if only the conditional operator is overloaded. Opt-out from consistency via explicit `common_type` specialization.

5.3.4 `operator?:(T, U) = default`

criteria	result
opt-in vs. opt-out	✔
discoverability	✘ The connection to `common_type` is implicit and must be learned, i.e. cannot easily be discovered. The conditional operator has three arguments, using only two parameters in the definition of the overload is confusing.
contextual `bool`	✔
deferred eval	✔
DRY	✔ `common_type` and `operator?:` are forced to be consistent with a single definition of the common type.

5.3.5 summary

criteria	5.3.1	5.3.2	5.3.3	5.3.4
opt-in vs. opt-out	?	✔	✔	✔
discoverability	✘	✔	✘	✘
contextual `bool`	✔	✔	✔	✔
deferred eval	✔	✔	✔	✔
DRY	✔	?	?	✔

Noting that the built-in conditional operator accepts arguments that are "contextually convertible to `bool` ", we see that using `bool` in the `operator?:` defintion above is not the perfect choice. We would need to use a concept such as instead of `bool`:

```
template<class B>
```

Alternatively, a defaulted `operator?:` could omit the first argument if it should accept anything contextually convertible to `bool`:

```
R operator?:(B b, C c) = default;
...
R x = a ? b : c;
```

A non-defaulted `operator?:` would behave like any other operator overload and need an orthogonal mechanism for lazy evaluation.

5.4 synthesizing the conditional operator from common_type specializations

An obvious idea from the above discussion is to simply synthesize a conditional operator when `common_type` is defined, but `?:` is not usable. Basically `a ? b : c` gets turned into `a ? static_cast<std::common_type_t<decltype(b), decltype(c)>>(b) : static_cast<std::common_type_t<decltype(b), decltype(c)>>(c)`.

Note that this would be an incomplete solution as it would not generalize to non-boolean cases / blend operations. Also, implementing expression templates via this solution should be possible but be awkward: The common type of two expressions would have to be defined as a "conditional expression" on two operands.

5.5 parameter order

5.6 exploring interaction with common_type

As discussed in Section 3.1, a user-defined conditional operator would automatically extend `common_type`. I will explore the interaction using the `rational` class from Seymour [P1438R1]. `rational` is not a template and has the following constructors:

```
class rational {
public:
```

```
  // ...
  explicit rational(float num);
  explicit rational(double num);
  explicit rational(long double num);
  explicit rational(integer num);
  // ...
```

Using Concepts this should be:

```
class rational {
public:
  // ...
  explicit rational(floating_point auto num);
  explicit rational(integral auto num);
  // ...
```

This means `rational` is explicitly convertible from all arithmetic types (including `bool`). `rational` models the built-in arithmetic types, aiming to be a drop-in replacement where a built-in type does not suffice.

`rational` also defines operator overloads which allow mixed operations of `rational` and integral types, effectively allowing implicit conversion from integers to `rational` within binary operations.[12] Consequently, on initial inspection it is unclear why

```
void g(const rational&);
void f(rational x) {
  g(x < 0 ? 0 : x);
}
```

should not work[13]. The built-in conditional operator does not work because `int` is not implicitly convertible to `rational`. So let us define the necessary overloads:

```
rational operator?:(contextual_boolean auto, const rational&, std::integral auto)
  = default;
rational operator?:(contextual_boolean auto, const rational&, std::floating_point auto)
  = default;
```

The overloads work as expected, however we implicitly made a change to `common_type` without touching it. The following `common_type` specializations are now valid:

```
// all of these are OK:
static_assert(std::is_same_v<std::common_type_t<rational, int>, rational>);
static_assert(std::is_same_v<std::common_type_t<rational, float>, rational>);
static_assert(std::is_same_v<std::common_type_t<short, rational>, rational>);
static_assert(std::is_same_v<std::common_type_t<double, rational>, rational>);
// ...
```

12 Let's not discuss design choices of `rational` and assume this is exactly what the class designer needs.
13 Wait for it.

These are all sensible types when asking for a common type that can store either a `rational` or an `int`, `float`, `short`, or `double`. After all, that's exactly what the return type of the conditional operator needs to be: the common type that can store either of the inputs.

For what it's worth, I have not come up with an example where the return type of `operator?:` would be a bad default for `common_type`. And should such a case come up, it is still possible to explicitly specialize `common_type`.

TODO: Note that maybe the conditional operator for `rational` is not that good of an idea. But we still want to define a `common_type`. That implies that the presence of a `common_type` should not imply the conditional operator works. I.e. the idea to synthesize an operator from `common_type` has issues.

Considering the semantics of the conditional operator, the return type of the operator must be a type that can represent either of the two last operands. Consequently it is a common type in the sense that both objects can be converted to it. If the user-defined conditional operator is used in the intended semantics of the language, the resulting `common_type` specialization is highly likely to be what the user intended.

Currently, `std::common_type` is specified in terms of the `decltype` of the conditional operator. Consequently, if the `common_type` specification is not changed, the declaration of user-defined conditional operators affects the result of `common_type`. I strongly believe this is the preferred behavior. Either `common_type` specializations should extend `operator?:` or `operator?:` overloads should extend `common_type`. The inconsistency we currently have from user-defined specializations of `common_type` is suboptimal (i.e. a common type is defined, but the conditional operator still is not usable). The DRY ("don't repeat yourself") principle implies we should enable a way for users to extend `operator?:` and `common_type` with a single definition. The more flexible and natural customization point is `operator?:`.

5.7 deferred evaluation

One of the expected features of the conditional operator is deferred evaluation of the expressions after the question mark. However, deferred evaluation is an orthogonal problem, and best handled via an independent proposal such as [P0927R2]. A desire to first solve deferred evaluation before deciding on overloading the conditional operator was voiced a few times. I strongly believe `operator?:` overloading is worthwhile even if [P0927R2] (or a different facility solving that same problem) does not move forward. This is because a major part of the motivation for `operator?:` overloading is for blend operations. Blend operations cannot make use of deferred evaluation and thus can benefit from the simplest way of `operator?:` overloading.

Consider a conceivable implementation of the conditional operator for `simd<T, Abi>` as shown in Listing **??**. If this code is inlined[14], the compiler will know how to improve the calling code without

14 A reasonable `simd` implementation forces inlining for most functions.

```
template <class T, class Abi>
simd<T, Abi> operator?:(simd_mask<T, Abi> mask, simd<T, Abi> a, simd<T, Abi> b) {
  if (all_of(mask)) [[unlikely]] {
    return a;
  } else if (none_of(mask)) [[unlikely]] {
    return b;
  }
  where(mask, b) = a;
  return b;
}
```

Listing 7: Simple `operator?:` for `simd<T, Abi>`

the need for explicit deferred evaluation of `a` and `b`. Only if the expressions in the second and third argument to the conditional operator have side effects, is the difference important.[15]

Pure numerical code (thus without side effects) can also optimize a simple conditional operator that does not make use of deferred evaluation. For expression templates, `operator?:` overloads can and have to implement deferred evaluation themselves anyway.

5.8 Partial feature until lazy evaluation lands

There has been concern that we should not add another feature to the language that would get an immediate entry into coding guidelines forbidding its use in most situations. The concern is that, similar to `operator&&` and `operator||`, the conditional operator should not be used because it does not implement the same lazy evaluation semantics as the builtt-in operators do. Those guidelines are correct for the great majority of cases, except for the few cases where lazy evaluation is irrelevant and it is okay to overload `&&` and `||` even without lazy evaluation (examples are `valarray` and `simd`). So the language should rather be restricted to avoid errors from users that do not follow guidelines.

As a committee we could follow that reasoning and still provide an overloadable conditional operator. It would have to be restricted to non-boolean conditions, i.e. `!std::is_constructible_v<bool, T>`.

This would enable the blend use cases but leave many valid use cases (expression templates, `bounded::integer`) on the floor. It would be possible to extend `operator?:` to boolean conditions once lazy evaluation is added to the language.

6 Suggested Polls

Poll: Pursue defaulted `operator?:`

SF	F	N	A	SA

15 Side effects in those expressions are likely bugs anyway (printf debugging maybe being an exception)

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

7 WORDING

TBD.

A BIBLIOGRAPHY

[P0927R2] James Dennett and Geoff Romer. *P0927R2: Towards A (Lazy) Forwarding Mechanism for C++*. ISO/IEC C++ Standards Committee Paper. 2018. URL: `https://wg21.link/p0927r2`.

[N2661] Howard E. Hinnant et al. *N2661: A Foundation to Sleep On*. ISO/IEC C++ Standards Committee Paper. 2008. URL: `https://wg21.link/n2661`.

[N4808] Jared Hoberock, ed. *Working Draft, C++ Extensions for Parallelism Version 2*. ISO/IEC JTC1/SC22/WG21, 2019. URL: `https://wg21.link/n4808`.

[1] Khronos OpenCL Working Group. *The OpenCL Specification*. Jan. 2011. URL: `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[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.

[N4842] Richard Smith, ed. *Working Draft, Standard for Programming Language C++*. ISO/IEC JTC1/SC22/WG21, 2019. URL: https://wg21.link/n4842.

[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.