Document Number: P0917R3
Date: 2019-10-07
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: EWG-I
Target: C++23

Making operator?: overloadable

ABSTRACT

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

CONTENTS

1 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 a need for overloading the conditional operator.

It is important to consider `std::common_type` when discussing changes to the conditional operator. `common_type_t<T, U>` basically is defined as `decltype(false ? T() : U())`. Consequently, if the conditional operator supports more types via user-defined overloads, `common_type` would automatically support them as well.

A previous revision of this paper discussed how to enable deferred evaluation. But since Dennett et al. [P0927R2] is trying to solve deferred evaluation in general, this paper will instead rely on the facilities of [P0927R2].

2 MOTIVATION

2.1 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).

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, except 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 of the built-in types.

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. Lazy evaluation is an orthogonal problem to solve and should not be tied to a solution for overloading the conditional operator. It is needed just as much for `operator&&` and `operator||` as it would be needed for `operator?:`. Whether adding the ability to overload `operator?:` should wait for lazy evaluation to become available is not about orthogonality but about hand-holding our users[1].

2.2 Blend operations

The conditional operator is a perfect match for expressing blend operations generically. I.e. a function template using the conditional operator uses blending of objects of user-defined types but can also use fundamental types, where blending means boolean selection of scalar values. 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. All operators act element-wise and concurrently. Thus, the meaning of

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

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`). The alternative solution for `simd` blend operations is to use a function, such as "inline-if":

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

An "inline-if" function is

- less intuitive, since the name is either long or it is cryptic, and the arguments appear to be arbitrarily ordered (comma doesn't convey semantics such as `?` and `:` do).
- harder to use in generic code: If `T` is a built-in type, the `iif` function will not be found via ADL; consequently, user code requires `return std::experimental::iif(x < 0, -x, x)` to be generic. This is annoying and easily forgotten since ADL works fine for `simd` arguments.

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

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 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	after

<pre>template <class T>
  void abs(T x)
{
  if constexpr (std::is_simd_v<T>)
    {
      where(x < 0, x) = -x;
      return x;
    }
  else
    return x < 0 ? -x : x;
}</pre>
	<pre>template <class T>
	  void abs(T x)
	{
	  return x < 0 ? -x : x;
	}</pre>

Tony Table 1: generic `abs` function supporting `simd`

2.3 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."[2] 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."[3]

2 https://boostorg.github.io/yap/doc/html/boost_yap/rationale.html
3 https://boostorg.github.io/yap/doc/html/BOOST_YAP_USER_EX_idm15635.html

2.4 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[4]. Examples:

- std::chrono::duration<Rep, Period>
- std::chrono::time_point<Clock, Duration>
- fractional<Numerator, Denominator> from [P1050R0]
- fixed_point<Rep, Exponent, Radix> from [P0037R5]
- bounded::integer<minimum, maximum> from [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?:(std::Boolean, T1, T2) in most cases. I.e. the idea to only allow non-bool conditions on operator?: overloads breaks this use case. (I mentioned the idea in the previous revisions and it was also suggested in EWGI discussion).

2.5 Existing practice

GCC implements support for the conditional operator to allowing blending its vector builtins[5]. 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 cf. https://codesearch.isocpp.org/cgi-bin/cgi_ppsearch?q=struct+common_type%3C&search=Search
5 https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

<table>
<tr><th>before</th><th>after</th></tr>
<tr><td>

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

</td><td>

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

</td></tr>
</table>

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

<table>
<tr><th>before</th><th>after</th></tr>
<tr><td>

```
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);
}
```

</td><td>

```
template<class T, class U>
  void f(bool cond, T a, U b)
{
  g(cond ? a : b);
}
```

</td></tr>
</table>

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

3 EXPLORATION

3.1 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 Figure 1.

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 Figure 1 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 operators already. However,

```
// 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)
```

Figure 1: Examples of the conditional operator with UDTs

`common_type` behaves differently, since it can only be specialized once. 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.[6] Note that such pitfalls are not novel. All operator (and function) overloads can already be used to construct such inconsistencies (e.g. Figure 2).

Nevertheless, because of the connection between `common_type` and the conditional operator, I believe we should consider the possibility of disregarding user-defined operators whenever the built-in operator is a candidate. It would be nicer to make the declaration of such operator overloads ill-formed. But I believe this is impossible since it appears to be a similar problem as definition checking for concepts. We could, however, consider to make such operator overload declarations ill-formed NDR.

That said, I believe such a constraint on `operator?:` is complicating the language for little gain and might even inhibit valid use cases. I would prefer to make `operator?:` just as useful and dangerous as all other overloads. Suggested poll: "`operator?:` should have special rules to avoid overloading the built-in operator".

6 `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())>);`

```
struct A { operator int() const; };
struct B { operator float() const; };

template <class A, class B> struct my_common_type {
  using type = decltype(A() + B());
};
template <class A, class B>
using my_common_type_t = typename my_common_type<A, B>::type;

using X = my_common_type_t<A, B>;
static_assert(std::is_same_v<X, my_common_type_t<A, B>>);
static_assert(std::is_same_v<X, decltype(A() + B())>);

short operator+(A, B);
static_assert(std::is_same_v<X, my_common_type_t<A, B>>);
static_assert(std::is_same_v<X, decltype(A() + B())>);  // fails
```

Figure 2: A pitfall of overloading (cf. `https://godbolt.org/z/iqbj1a`)

3.2 Should `common_type` ignore user-defined conditional operators?

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?:`.

3.3 Defaulted conditional operator overload

In most scalar cases, the implementation of the conditional operator is trivial (i.e. return either `b` or `c`, depending on `a` for `a ? b : c`). The interesting choice when overloading the conditional operator is the return type. Thus defaulting the operator appears like a logic step.

When the implementation is defaulted, it is simple to make these operators implement lazy evaluation. Consider:

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

The definition of this operator could mean the equivalent of

```
R x = a ? static_cast<R>(b) : static_cast<R>(c);
```

and thus implement lazy evaluation. 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>
  concept contextual_boolean = std::is_constructible_v<bool, 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.

3.4 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.

3.5 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 Figure 3. If this code is inlined[7], the compiler will know how to

```
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;
}
```

Figure 3: Simple `operator?:` for `simd<T, Abi>`

improve the calling code without 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.[8]

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.

3.6 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>`.

7 A reasonable `simd` implementation forces inlining for most functions.
8 Side effects in those expressions are likely bugs anyway (printf debugging maybe being an exception)

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.

4 SUGGESTED POLLS

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: 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

5 WORDING

TBD.

6 CHANGELOG

6.1 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.

6.2 Changes from revision 1

Previous revision: [P0917R1]

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

6.3 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 `?:`.

7 Straw Polls

7.1 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	?	?

7.2 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

7.3 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

A BIBLIOGRAPHY

[P0927R0] James Dennett and Geoff Romer. *P0927R0: Towards A (Lazy) Forwarding Mechanism for C++*. ISO/IEC C++ Standards Committee Paper. 2018. URL: `https://wg21.link/p0927r0`.

[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`.

[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*. 2011. URL: `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[P0917R0] Matthias Kretz. *P0917R0: Making operator?: overloadable*. ISO/IEC C++ Standards Committee Paper. 2018. URL: `https://wg21.link/p0917r0`.

[P0917R1] Matthias Kretz. *P0917R1: Making operator?: overloadable*. ISO/IEC C++ Standards Committee Paper. 2018. URL: https://wg21.link/p0917r1.

[P0917R2] Matthias Kretz. *P0917R2: Making operator?: overloadable*. ISO/IEC C++ Standards Committee Paper. 2019. URL: https://wg21.link/p0917r2.

[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.

[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.