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   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. Operator-dot overloading is under consideration in the committee [P0352R1, P0416R1]. This leaves `operator?:` as the last missing piece.

2.2   BLEND OPERATIONS

The conditional operator is a perfect match for expressing blend operations generically, i.e. such that a function template also works with fundamental types. 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

```cpp
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":

```cpp
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.

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.

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. Most sensible uses of the conditional operator will likely be similar to the "blend operations" case discussed for `simd` types, though. The motivation is not as strong as in the above case, since in most cases substitutability of the code to fundamental types is not a goal.

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`[1]. 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[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.

---

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

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

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.

```
// 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

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, `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.[3] Note that such pitfalls are not novel. All operator (and function) overloads can already be used to construct such inconsistencies (e.g. Figure 2).

```cpp
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`)

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 a 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".

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

---

3 `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())>);`

5

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?:`.

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[4], the compiler will know how to

```cpp
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.[5]

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.

---

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

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

4 WORDING

TBD.

5 CHANGELOG

5.1 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.

5.2 CHANGES FROM REVISION 1

Previous revision: [P0917R1]

- Discuss `common_type`.

- Discuss overloading `operator?:(bool, ...)`.

- Mention `chrono::duration` and other numeric types as motivation.

6 STRAW POLLS

6.1 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	?	?

6.2 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

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.

[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).

[P0416R1]   Bjarne Stroustrup and Gabriel Dos Reis. *P0416R1: Operator Dot (R3)*. ISO/IEC C++ Standards Committee Paper. 2016. URL: https://wg21.link/p0416r1.

[P0745R0]   Herb Sutter. *P0745R0: Concepts in-place syntax syntax*. ISO/IEC C++ Standards Committee Paper. 2018. URL: https://wg21.link/p0745r0.

[P0352R1]   Hubert Tong and Faisal Vali. *P0352R1: Smart References through Delegation*. ISO/IEC C++ Standards Committee Paper. 2017. URL: https://wg21.link/p0352r1.