Document Number: P0917R1
Date: 2018-05-06
Reply-to: Matthias Kretz <m.kretz@gsi.de>
Audience: EWG

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 [2] 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 as well as present a possiblity of deferred evaluation of `expr2` and `expr3`.

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[1].

2.2 BLEND OPERATIONS

The conditional operator is a perfect match for expressing blend operations generically, i.e. so that fundamental types still work with the same syntax. Consider [N4744], where a certain number (determined at compile time) of values of arithmetic type `T` are combined to a single object of type `simd<T, Abi>`. 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);
}
```

1 Maybe `operator::` as well. I didn’t take much time to consider what it would mean, though.

This is less intuitive, since the name is either long or cryptic, and the arguments appear to be arbitrarily ordered (comma doesn't convey semantics such as `?` and `:` do). More importantly, if `x` is a builtin type, the 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, 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 INCREASED FLEXIBILITY ABOUT PARTICIPATING TYPES

Consider the `bounded::integer` example (cf. [1]):

```
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.

2.5 EXISTING PRACTICE

GCC implements support for the conditional operator to allowing blending its vector builtins. OpenCL uses the conditional operator for blending operations. Allowing overloads of `operator?:` in C++ enables users to implement blend semantics with the same syntax as provided by GCC and OpenCL.

3 CHOICES

The main issue to decide when considering overloading the conditional operator, is deferred evaluation of the second and third expressions. [expr.cond]/1 specifies "Only one of the second and third expressions is evaluated". If the signature of overloadable `operator?:` were `T operator?:(U, T, T)` then all three expressions must be evaluated before calling the user-defined operator. To resolve this, the signature could be defined as `T operator?:(U, F0, F1)`, where `F0` and `F1` are callables with return type `T`. Calling code such as `auto x = cond ? a + b : g(a, b)` could then be transformed to `auto x = operator?:(cond, [&]() return a + b; , [&]() return g(a, b); )`.

This could be taken one step further: Instead of passing a callable, pass an object that is implicitly convertible to `T`. Its conversion operator invokes the expression. This may be easier to use, but it's also easier to use badly (as in invoking the conversion operator multiple times). I believe such an approach is too magical, so I will not pursue it further in this paper.

Dennett et al. [P0927R0] propose an extension that would easily solve the issue. Thus, if P0927R0 is adopted, the issue of deferred evaluation is easily resolved and under full control of the developer introducing the operator overload.

If P0927R0 is not adopted, we have two choices for the signature of the conditional operator overload (cf. Figure 1):

1. The simple approach, which follows the rules of the other operator overloads:

```
template <class T, class U>
MyReturnType<T, U> operator?:(MyCondition<T, U> c, T a, U b) {
  if (c) {
    return a;
  } else {
    return b;
  }
}
```

The expression `x = c ? f(a, b) : g(a, b)` means `x = operator?:(c, f(a, b), g(a, b))`.

2. An approach to support deferred evaluation:

```
template <class F0, class F1,
          class T = std::invoke_result_t<F0>,
          class U = std::invoke_result_t<F1>>
MyReturnType<T, U> operator?:(MyCondition<T, U> c, F0 a, F1 b) {
  if (c) {
    return a();
  } else {
    return b();
  }
}
```

The expression `x = c ? f(a, b) : g(a, b)` means `x = operator?:(c, [&]() { return f(a, b); }, [&]() { return g(a, b); })`.

3.1 Overload resolution

Choice 1 allows overload resolution on the types for the second and third argument. If we want to support overloads using `bool` for the first argument, then choosing the overload via the types of the remaining arguments is important.

Choice 2 requires considerably more complex overload resolution rules. Consider `template <class F0, class F1> UDT operator?:(bool, F0, F1)`, where `F0` and `F1` return a prvalue of type `UDT`. The built in operator would be viable for `some_bool ? udt_object_a : udt_object_b` and the overload, even without transformation of the expressions into lambda. I.e. the user-defined operator, requires a constraint that `F0` and `F1` are callables. Assuming we had this, we'd now have to require all uses of the conditional operator to consider both the implicit-lambda wrapper and the direct use of the built in operator for overload resolution. Should a viable user-defined conditional operator with an implicit-lambda wrapper then always be prefered over the built in operator? It seems this would be required. Consequently, it is very important that user-defined conditional operators use exact constraints on the second and third parameter types.

3.2 Life-time extension (non-)issue

Using choice 2, there is a case where life-time extension does not work, where it would otherwise work in the built-in case. Consider:

```cpp
template <class T0, class T1>
const auto ternary1(bool c, T0 &&yes, T1 &&no) {
    if (c) return std::forward<T0>(yes);
    return std::forward<T1>(no);
}

template <class F0, class F1>
const auto ternary2_impl(bool c, F0 &&yes, F1 &&no) {
    if (c) return yes();
    return no();
}
#define ternary2(c_, yes_, no_)                \
ternary2_impl((c_), [&]() { return yes_; }, \
                    [&]() { return no_; })

int a();
int b();

int f1(bool c) {
    return ternary1(c, a(), b());
}
int f2(bool c) {
    return ternary2(c, a(), b());
}
int f3(bool c) {
    return c ? a() : b();
}
```

```asm
f1(bool):
pushq %rbp
pushq %rbx
pushq %rax
movl %edi, %ebx
callq a()
movl %eax, %ebp
callq b()
testb %bl, %bl
cmovnel %ebp, %eax
addq $8, %rsp
popq %rbx
popq %rbp
retq

f2(bool):
testb %dil, %dil
je .LBB1_2
jmp a()
.LBB1_2:
jmp b()

f3(bool):
testb %dil, %dil
je .LBB2_2
jmp a()
.LBB2_2:
jmp b()
```

Figure 1: Demonstration of the two choices compared to a builtin conditional operator (f3), cf. https://godbolt.org/g/tpPrVR

```
template <class F0, class F1>
const auto &operator?:(MyCond c, F0 &&yes, F1 &&no) {
  if (c) {
    return yes();
  }
  return no();
}

const auto &x = c ? a + b : a - b;
```

In this case the temporary is produced inside the `operator?:` function, and thus returning a const-ref returns a reference to a local temporary object. Using choice 1, the temporary is produced before calling the overloaded operator, and thus lifetime extension would make the last line well-formed. The simple solution here is to return `auto` instead of `const auto &`, so this issue appears to be rather academic.

3.3 Allow different types?

The operator signature could enforce the types of the second and third expression to be equal, implicitly/explicitly convertible, or arbitrary. I believe the most flexible tool for users will be to allow arbitrarily different types. Users can put a restriction in place by themselves.

3.4 Is there a need for deferred evaluation?

Consider a conceivable implementation of the conditional operator using choice 1 for `simd<T, Abi>`:

```
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;
}
```

If this code is inlined, the compiler will know how to 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.

Consider a possible implementation of the conditional operator for `bounded::integer`:

```
template <BoundedInteger T0, BoundedInteger T1>
common_type_and_value_category_t<T0, T1> operator?:(bool cond, T0 a, T1 b) {
  return cond ? static_cast<common_type_and_value_category_t<T0, T1>>(a)
              : static_cast<common_type_and_value_category_t<T0, T1>>(b);
}
```

Again, inlining can cover all the important cases (i.e. all but side effects).

In the presence of inlining (and link-time optimizations), I would prefer to go with choice 1.

- I believe we do not have to complicate the language to support conditional side effects in overloaded conditional operators.
- Choice 1 is less of an implementation burden.
- Choice 1 is simpler to use and understand, even if slightly less powerful.
- Choice 2 would break with the overload syntax of all other overloadable operators.

If a mechanism as suggested in Dennett et al. [P0927R0] is adopted, I'd be strongly in favor of choice 1 / against choice 2.

4 WORDING

TBD.

5 CHANGELOG

5.1 CHANGES FROM R0

Previous revision: [P0917R0].

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

6 STRAW POLLS

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`.

[N4744] Jared Hoberock, ed. *Technical Specification for C++ Extensions for Parallelism Version 2*. ISO/IEC JTC1/SC22/WG21, 2018. URL: `https://wg21.link/n4744`.

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

[1] David Stone. *davidstone / bounded_integer — Bitbucket*. URL: `https://bitbucket.org/davidstone/bounded_integer` (visited on 02/26/2018).

[2] 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`.