Document Number: P0917R0
Date: 2018-02-12
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: `?:`, `::`, `.`[1], `.*`. For the conditional operator, Stroustrup [1] 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 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 Kretz [P0214R8], 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);
}
```

This is less intuitive, since the name is either long or cryptic and the arguments seem to be arbitrarily ordered (comma doesn't convey semantics such as `?` and `:` do). More importantly, a function will typically not be found via ADL, so that the code actually requires `return std::experimental::iif(x < 0, -x, x)` to work for fundamental types. 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.

1 cf. P0416R0

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.

2.2 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.

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 a `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.

Thus, we have two choices for supporting the operator overload (cf. Figure 1):

1. The simple approach, which follows the rules of the other operator overloads:
   ```
   T operator?:(Cond c, T a, T 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))`.

```
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();
}
```

```
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 with a builtin conditional operator (`f3`), cf. `https://godbolt.org/g/tpPrVR`

2. An approach to support deferred evaluation:

   ```
   T operator?:(Cond 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 (I do not know of a use case), then choosing the overload via the types of the remaining arguments is important. Choice 2 makes this harder, since the requirement must be expressed as a return type of the function call expression of the arguments.

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:

```
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 `const 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 the 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.

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.

4 WORDING

TBD.

A BIBLIOGRAPHY

[P0214R8] Matthias Kretz. *P0214R8: Data-Parallel Vector Types & Operations*. ISO/IEC C++ Standards Committee Paper. 2018. URL: https://wg21.link/p0214r8.

[1] Bjarne Stroustrup. *Stroustrup: C++ Style and Technique FAQ*. 2018. URL: http://www.stroustrup.com/bs_faq2.html#overload-dot (visited on 01/31/2018).