

You must type it three times

Bloomberg

Vittorio Romeo

<https://vittorioromeo.info>
vittorio.romeo@outlook.com

C++ Now 2017
lightning talk

```
inline constexpr auto foo = [](auto&& x)
    noexcept(noexcept(bar(x)))
    → decltype(bloop(x))
{
    // ...
}
```

- The user has a `foo` *function object* with complicated `noexcept` and *return type*.
- You are a **library developer**, and want to provide a generic `log_and_call` utility that must work properly with `foo`.

attempt #0

```
template <typename F, typename ... Ts>
decltype(auto) log_and_call(F&& f, Ts&& ... xs)
{
    log << "calling `f`\n";
    return std::forward<F>(f)(std::forward<Ts>(xs) ... );
}
```

Any issues?

noexcept is missing.

```
noexcept(log_and_call(foo, /* ... */))
```

Will always evaluate to **false** .

attempt #1

```
template <typename F, typename ... Ts>
decltype(auto) log_and_call(F&& f, Ts&& ... xs)
    noexcept(noexcept(
        std::forward<F>(f)(std::forward<Ts>(xs) ... )
    ))
{
    log << "calling `f`\n";
    return std::forward<F>(f)(std::forward<Ts>(xs) ... );
}
```

Any issues?

SFINAE unfriendliness.

```
std::is_invocable<log_and_call, foo, /* ... */>
```

May produce a **compiler error** instead of getting *SFINAE-d away*!

It will also behave differently from:

```
std::is_invocable<foo, /* ... */>
```

attempt #2

```
template <typename F, typename ... Ts>
auto log_and_call(F&& f, Ts&& ... xs)
    noexcept(noexcept(
        std::forward<F>(f)(std::forward<Ts>(xs) ... )
    ))
→ decltype(
    std::forward<F>(f)(std::forward<Ts>(xs) ... )
)
{
    log << "calling `f`\n";
    return std::forward<F>(f)(std::forward<Ts>(xs) ... );
}
```

Any issues?

constexpr unfriendliness.

attempt #3

```
template <typename F, typename ... Ts>
constexpr auto log_and_call(F&& f, Ts&& ... xs)
    noexcept(noexcept(
        std::forward<F>(f)(std::forward<Ts>(xs) ... )
    ))
→ decltype(
    std::forward<F>(f)(std::forward<Ts>(xs) ... )
)
{
    log << "calling `f`\n";
    return std::forward<F>(f)(std::forward<Ts>(xs) ... );
}
```

...ah, beautiful.

```
std::forward<F>(f)(std::forward<Ts>(xs) ... )
```

...had to be **manually repeated three times** in order to achieve:

- noexcept correctness.
- SFINAE-friendliness.

Why can't the compiler do this for us?

```

#define RETURNS( ... ) \
    noexcept(noexcept(__VA_ARGS__)) \
    → decltype(__VA_ARGS__) \
{ \
    return __VA_ARGS__; \
}

```

```

template <typename F, typename ... Ts>
constexpr auto log_and_call(F&& f, Ts&& ... xs)
    RETURNS(
        log << "calling `f`\n",
        std::forward<F>(f)(std::forward<Ts>(xs) ... )
    )

```

```

template <typename F, typename ... Ts>
constexpr auto log_and_call(F&& f, Ts&& ... xs)
    RETURNS(
        log << "calling `f`\n",
        std::forward<F>(f)(std::forward<Ts>(xs) ... )
    )

```

...compare to...

```

template <typename F, typename ... Ts>
constexpr auto log_and_call(F&& f, Ts&& ... xs)
    noexcept(noexcept(
        std::forward<F>(f)(std::forward<Ts>(xs) ... )
    ))
→ decltype(
    std::forward<F>(f)(std::forward<Ts>(xs) ... )
)
{
    log << "calling `f`\n";
    return std::forward<F>(f)(std::forward<Ts>(xs) ... );
}

```

Advantages of RETURNS :

- Avoids code triplication.

Disadvantages of RETURNS :

- Macro, will be exposed to clients of the library.
- Assumes something will be returned.
 - Only works with *expressions*.
 - Doesn't play nicely with `void` return type.
 - The *comma operator* must be (ab)used to simulate *compound statements*.

Why can't the compiler do this for us?

Barry Rezvin proposed

"Abbreviated Lambdas for Fun and Profit" (P0573)

Part of the proposal was about a new `⇒ expr` syntax.

That is, the lambda:

```
[ ](auto&& x) ⇒ test(x)
```

shall be exactly equivalent to the lambda:

```
[ ](auto&& x) noexcept(noexcept(test(x)))  
    → decltype(test(x)) { return test(x); }
```

Unfortunately most of the proposal was rejected.

Inspired by it, I think that \Rightarrow could be used as a new **generic function body definition syntax**, supporting *lambdas*, *functions*, and any kind of *expression* or *compound-statement*.

dream pseudocode

```
auto triple = [](auto x)  $\Rightarrow$  x * 3;
```

```
template <typename F>  
void call_twice(F&& f)  $\Rightarrow$  {  
    f();  
    f();  
}
```

Can we make this happen?

The biggest issue is that both:

- `noexcept(/* compound-statement */)`
- `decltype(/* compound-statement */)`

are currently **ill-formed**.

In the case of `noexcept(/* compound-statement */)`, it would be necessary to create some formal "splitting" rules for the compiler.

E.g.

```
noexcept({ a(); b(); })
```

...expands to...

```
noexcept(a()) && noexcept(b())
```

```
noexcept( { if(a()) { b(); } } )
```

...expands to...

```
noexcept(a()) && noexcept(b())
```

...*et cetera*.

Similar ideas would apply to

```
decltype( /* compound-statement */ ) .
```

Some additional new features might be required. E.g.

```
decltype({  
    if constexpr(a()){ return b(); }  
    else { return c(); }  
})
```

...might expand to...

```
//                                true case  
//                                vvvv  
conditional_decltype(a())(b())(c())  
//                                ^^^^      ^^^^  
//    constant-expression          //  
//                                //  
//                                false case
```

**Maybe all of this is impossible due to
implementation difficulties...**

...but *one man can dream*.

```
template <typename F, typename ... Ts>
constexpr auto log_and_call(F&& f, Ts&& ... xs) ⇒
{
    log << "calling `f`\n";
    return std::forward<F>(f)(std::forward<Ts>(xs) ... );
}
```


Until then...

...you must type it three times.

```
template <typename F, typename ... Ts>
constexpr auto log_and_call(F&& f, Ts&& ... xs)
    noexcept(noexcept(
        std::forward<F>(f)(std::forward<Ts>(xs) ... )
    ))
→ decltype(
    std::forward<F>(f)(std::forward<Ts>(xs) ... )
)
{
    log << "calling `f`\n";
    return std::forward<F>(f)(std::forward<Ts>(xs) ... );
}
```

Thanks!

- vromeo5@bloomberg.net
- <https://vittorioromeo.info>
- vittorio.romeo@outlook.com
- [@supahvee1234](#)