

Document: P2173R0
Revises: (original)
Date: 15-May-2020
Audience: EWG
Authors: Inbal Levi (sinbal21@gmail.com)
Daveed Vandevoorde (daveed@edg.com)
Ville Voutilainen (ville.voutilainen@gmail.com)

Attributes on Lambda-Expressions

Introduction

This paper proposes a fix for Core Issue 2097 (http://open-std.org/JTC1/SC22/WG21/docs/cwg_toc.html#2097), to allow attributes for lambdas, those attributes appertaining to the function call operator of the lambda.

Lambdas are shorthands for function objects; it seems reasonable to allow attributes like `[[nodiscard]]`, `[[deprecated]]` and `[[noreturn]]` for the lambda call operator. For a lambda that wraps a `[[noreturn]]` function, it's half-evidently reasonable to allow the lambda to be `[[noreturn]]`. For lambdas that are in a wider scope than a block scope (and even for lambdas in a block scope, to catch misuses), it seems reasonable to allow `[[nodiscard]]` and `[[deprecated]]`.

This arguably makes the language more regular: Function objects allow marking their call operators with attributes and, with this change, shorthand function objects allow that too.

The C++20 grammar for a lambda expression is as follows (`[expr.prim.lambda]/1`):

lambda-expression :

lambda-introducer lambda-declarator_{opt} compound-statement

lambda-introducer < template-parameter-list > requires-clause_{opt}

lambda-declarator_{opt} compound-statement

lambda-introducer :

[lambda-capture_{opt}]

lambda-declarator :

(parameter-declaration-clause) decl-specifier-seq_{opt} noexcept-specifier_{opt}

attribute-specifier-seq_{opt} trailing-return-type_{opt} requires-clause_{opt}

Of note for this paper is that this grammar currently reserves exactly *one* spot for attributes — a spot that parallels the similar grammar location for function declarators — for attributes that

appertain to the corresponding function type. However, attributes appertaining to declarator types are less common than attributes appertaining to just about *any other kind of thing* in C++¹.

So this paper argues for introducing an additional syntactic location for attributes in *lambda-expressions*, so that, for example, the following would become valid:

```
auto lm = [][[nodiscard, vendor::attr]]()->int { return 42; };
```

The remainder of this paper discusses design options and provides wording to enable this extension. All references to the working paper are based on N4861.

Design and Options

When considering the grammar in the introduction, we observe:

1. The existing location for attributes and their appertenance is consistent with other contexts permitting a function-like declarator. Besides, changing this could break existing code (though likely very little). We therefore do not think it wise to change that aspect of the existing syntax.
2. Because many of the elements of the lambda syntax are optional, there are only *three* potential syntactic locations for additional attributes: (a) prior to the *lambda-introducer*, (b) after the *lambda-introducer* but before the optional *lambda-declarator*, and (c) after the *compound-statement*. However, option (a) is not really available because it would introduce an ambiguity for the grammar of expression statements ([stmt.pre]/1).

When we consider the *semantics* of attributes in the context of lambda expressions, we also observe that a *lambda-expression* is a shorthand notation for a closure object and its (class) type and that type includes various elements (sometimes optional) like:

- the closure (class) type itself: attributes could be meaningful for it (e.g., to control alignment)
- the call operator of the closure: allowing attributes for this operator is the *raison d'être* of this paper and there is ample anecdotal evidence that programmers want to be able to say that this operator is `[[noreturn]]` or `[[nodiscard]]`, for example
- a conversion operator: it too could conceivably use attributes in some situations (a `[[deprecated]]` attribute, for example)
- various constructors, each of which could also conceivably want meaningful attributes

Providing a mechanism to allow attributes for *all* these elements would severely burden the syntax of *lambda-expressions*. We therefore propose to introduce additional attributes *only* for the call operator (using the location just after the *lambda-capture*), with a nod to the possibility of later also adding them for the closure class using trailing attributes (i.e., right after the

¹ And, for example, none of the *standard-supplied* attributes appertain to function types.

compound-statement). Note that the use of trailing attributes for expressions is not novel, since they can occur in *new-expressions* ([*expr.new*]/1). For elements that are not covered with a lambda-specific syntax, the programmer can instead write out a function-object class type explicitly.

So we propose to allow in C++23:

```
auto rethrower = [][[noreturn]]() { throw; };
```

leaving a potential later option for:

```
auto cntr = [i=0]{ return ++i; } alignas(64);
```

Proposed Wording Changes

Change the grammar for *lambda-expression* in [*expr.prim.lambda*] as follows:

lambda-expression :

lambda-introducer **attribute-specifier-seq_{opt}**

*lambda-declarator*_{opt} *compound-statement*

lambda-introducer < *template-parameter-list* > *requires-clause*_{opt}

attribute-specifier-seq_{opt} *lambda-declarator*_{opt} *compound-statement*

lambda-introducer :

[*lambda-capture*_{opt}]

lambda-declarator :

(*parameter-declaration-clause*)

*decl-specifier-seq*_{opt} *noexcept-specifier*_{opt}

*attribute-specifier-seq*_{opt} *trailing-return-type*_{opt} *requires-clause*_{opt}

Modify [*expr.prim.lambda.closure*]/4 as follows:

- 4 [...] An *attribute-specifier-seq* in a *lambda-declarator* appertains to the type of the corresponding function call operator or operator template. **An *attribute-specifier-seq* in a *lambda-expression* preceding a *lambda-declarator* appertains to the corresponding function operator or operator template.** [...]

Note: Some standard attribute descriptions talk about an attribute being “applied to” a *declarator-id*, whereas others talk about them being “applied to” the entity being declared by that *declarator-id*. Since lambda expressions do not involve a *declarator-id*, the other formulation seems preferable. The following changes are to consistently use that other formulation.

Modify [dcl.attr.depend]/1 as follows:

- 1 [...] The attribute may be applied to ~~the *declarator-id* of~~ a *parameter-declaration* in a function declaration or lambda, in which case it specifies that the initialization of the parameter carries a dependency to (6.9.2) each lvalue-to-rvalue conversion (7.3.1) of that object. The attribute may also be applied to ~~the *declarator-id* of~~ a function declaration **(including a lambda call operator)**, in which case it specifies that the return value, if any, carries a dependency to the evaluation of the function call expression.

Modify [dcl.attr.nodiscard]/1 as follows:

- 1 The attribute-token `nodiscard` may be applied to ~~the *declarator-id* in~~ a function declaration **(including a lambda call operator)** or to the declaration of a class or enumeration. [...]

Modify [dcl.attr.norturn]/1 as follows:

- 1 [...] The attribute may be applied to ~~the *declarator-id* in~~ a function declaration **(including a lambda call operator)**. [...]