Overload resolution hook: declcall(unevaluated-call-expression)

Document #: D2825R3
Date: 2024-12-16
Project: Programming Language C++
Audience: EWG
CWG
Reply-to: Gašper Ažman
<gasper.azman@gmail.com>

Contents

1 Introduction

This paper introduces a new expression into the language `declcall(`*`expression`*`)`.

The `declcall` expression is a constant expression of type pointer-to-function (PF) or pointer-to-member-function (PMF). Its value is the pointer to the function that would have been invoked if the *expression* were evaluated. The *expression* itself is an unevaluated operand.

In effect, `declcall` is a hook into the overload resolution machinery.

2 Motivation and Prior Art

The language already has a number of sort-of overload resolution facilities:

- static_cast
- assignment to a variable of a given function pointer type
- function calls (implicit) - the only one that actually works

All of these are unsuitable for ad-hoc type-erasure that library authors (such as [P2300R6]) need.

We can sometimes indirect through a lambda to "remember" the result of an overload resolution to be invoked later, if the function pointer type is not a perfect match:

```
template <typename R, typename Args...>
struct my_erased_wrapper {
  using fptr_t = R(*)(Args_...);
  fptr_t erased;
};
// for some types R, T1, T2, T3
my_erased_wrapper<R, T1, T2, T3> vtable = {
    +[](T1 a, T2 b, T3 c) -> R { return some_f(FWD(a), FWD(b), FWD(c)); }
};
```

… however, this does not work in all cases, and has suboptimal code generation.

- it introduces a whole new lambda scope
  - expensive for optimizers because extra inlining
  - annoying for debugging because of an extra stack frame
  - decays arguments prematurely or reifies prvalues both of which inhibit copy-elision
- it places additional unwelcome requirements on the programmer, who must:
  - divine the correct `noexcept(which?)`
  - explicitly (and correctly) spell the return type of the erased function
  - explicitly (and correctly) spell out the exact parameter types of the forwarded function
- if one fails to do the above, one must at least ensure that
  - arguments are convertible
  - return type is convertible
  - … both of which result in suboptimal codegen
- Nested erasures do not flatten: we cannot subset type-erased wrappers (we can't divine an exact match in the presence of overloads) (think "subsetting vtables")

Oh, if only we had a facility to ask the compiler what function we'd be calling and then *just have the pointer to it.*

This is what this paper is trying to provide.

2.1 Related Work

2.1.1 Reflection

The reflection proposal does not include anything like this. It knows how to reflect on constants, but a general-purpose feature like this is beyond its reach. Source: hallway discussion with Daveed Vandevoorde.

We probably need to do the specification work of this paper to understand the corner cases of even trying to do this with reflection.

Reflection ([P2320R0],[P1240R1],[P2237R0],[P2087R0],[N4856]) might miss C++26, and is far wider in scope as another `decltype`-ish proposal that's easily implementable today, and `std::execution` could use immediately.

Regardless of how we chose to provide this facility, it is dearly needed, and should be provided by the standard library or a built-in.

See the Alternatives to Syntax chapter for details.

2.1.2 Library fundamentals TS v3

The Library Fundamentals TS version 3 defines `invocation_type<F(Args...)>` and `raw_invocation_type<F(Args...)>` with the hope of getting the function pointer type of a given call expression.

However, this is not good enough to actually be able to resolve that call in all cases.

Observe:

```
struct S {
  static void f(S) {} // #1
  void f(this S) {}   // #2
};
void h() {
  static_cast<void(*)(S)>(S::f) // error, ambiguous
  S{}.f(S{}); // calls #1
  S{}.f(); // calls #2
  // no ambiguity for declcall
  declcall(S{}.f(S{})); // &#1
  declcall(S{}.f());    // &#2
}
```

A library solution can't give us this, no matter how much we try, unless we can reflect on unevaluated operands (which Reflection does).

3 Proposal

We propose a new (technically) non-overloadable operator (because `sizeof` is one, and this behaves similarly):

```
declcall(expression);
```

Example:

```
int f(int);  // 1
int f(long); // 2
constexpr auto fptr_to_1 = declcall(f(2));
constexpr auto fptr_to_2 = declcall(f(2l));
```

The program is ill-formed if the named *`postfix-expression`* is not a call to an addressable function (such as a constructor, destructor, built-in, etc.).

```
struct S {};
declcall(S()); // Error, constructors are not addressable
declcall(__builtin_unreachable()); // Error, not addressable
```

The expression is not a constant expression if the *`expression`* does not resolve for unevaluated operands, such as with function pointer values and surrogate functions.

```
int f(int);
using fptr_t = int (*)(int);
constexpr fptr_t fptr = declcall(f(2)); // OK
declcall(fptr(2)); // Error, fptr_to_1 is a pointer
struct T {
    constexpr operator fptr_t() const { return fptr; }
};
declcall(T{}(2)); // Error, T{} would need to be evaluated
```

If the `declcall(`*`expression`*`)` is evaluated and not a constant expression, the program is ill-formed (but SFINAE-friendly).

However, if it is unevaluated, it's not an error, because the type of the expression is useful as the type argument to `static_cast`!

Example:

```
int f(int);
using fptr_t = int (*)(int);
constexpr fptr_t fptr = declcall(f(2));
static_cast<declcall(fptr(2))>(fptr); // OK, fptr, though redundant
struct T {
    constexpr operator fptr_t() const { return fptr; }
};
static_cast<declcall(T{}(2))>(T{}); // OK, fptr
```

This pattern covers all cases that need evaluated operands, while making it explicit that the operand is evaluated due to the `static_cast`.

This division of labor is important - we do not want a language facility where the operand is conditionally evaluated or unevaluated.

Examples:

```
void g(long x) { return x+1; }
void f() {}                                                    // #1
void f(int) {}                                                 // #2
struct S {
  friend auto operator+(S, S) noexcept -> S { return {}; } // #3
  auto operator-(S) -> S { return {}; }                       // #4
  auto operator-(S, S) -> S { return {}; }                    // #5
  void f() {}                                                  // #6
  void f(int) {}                                               // #7
  S() noexcept {}                                              // #8
  ~S() noexcept {}                                             // #9
  auto operator->(this auto&& self) const -> S*;              // #10
  auto operator[](this auto&& self, int i) -> int;            // #11
  static auto f(S) -> int;                                     // #12
  using fptr = void(*)(long);
  auto operator fptr const { return &g; }                     // #13
  auto operator<=>(S const&) = default;                       // #14
};
S f(int, long) { return S{}; }                                 // #15
struct U : S {}

void h() {
  S s;
  U u;
  declcall(f());                     // ok, &#1            (A)
  declcall(f(1));                    // ok, &#2            (B)
  declcall(f(std::declval<int>()));  // ok, &#2            (C)
  declcall(f(1s));                   // ok, &#2 (!)        (D)
  declcall(s + s);                   // ok, &#3            (E)
  declcall(-s);                      // ok, &#4            (F)
  declcall(-u);                      // ok, &#4 (!)        (G)
  declcall(s - s);                   // ok, &#5            (H)
  declcall(s.f());                   // ok, &#6            (I)
  declcall(u.f());                   // ok, &#6 (!)        (J)
  declcall(s.f(2));                  // ok, &#7            (K)
  declcall(s);                       // error, constructor  (L)
  declcall(s.S::~S());               // error, destructor   (M)
  declcall(s->f());                  // ok, &#6 (not &#10)  (N)
  declcall(s.S::operator->());       // ok, &#10            (O)
  declcall(s[1]);                    // ok, &#11            (P)
  declcall(S::f(S{}));               // ok, &#12            (Q)
  declcall(s.f(S{}));                // ok, &#12            (R)
  declcall(s(1l));                   // error, #13          (S)
  static_cast<declcall(s(1l)>(s));   // ok, &13             (S)
  declcall(f(1, 2));                 // ok, &#15            (T)
  declcall(new (nullptr) S());       // error, not function (U)
  declcall(delete &s);               // error, not function (V)
  declcall(1 + 1);                   // error, built-in     (W)
  declcall([]{
      return declcall(f());
    }()());                          // error (unevaluated) (X)
  declcall(S{} < S{});               // error, synthesized  (Y)
}
```

TODO: call out difference between `declcall(obj.f())` and `declcall(obj.Base::f())` for virtual f.

3.1 Interesting cases in the above example

- resolving different members of a free-function overload set (A, B, C, D, T)
  - the (D) case is important - the `short` argument still resolves to the `int` overload!
- constructors and destructors (L, M, U, V) - see the **possible extensions** chapter.
- resolving different member of a member-function overload set (I, J, K, N, Q, R)
  - the (J) example is important - the call on `u` still resolves to a member function of `S`.
- resolving built-in non-functions (W): we could make this work in a future extension (see that chapter).
- resolving `operator->` (N and O). [expr.post.general] specifies that *`postfix-expression`*s group left-to-right, which means the top-most postfix-expression is the call to `f()`, and not the `->`. To get to `S::operator->`, we have to ask for it explicitly.
- surrogate function call (S) - again, the top-most call-expression is the function call to `g`, so the type of `g` is returned, but it's not a constant expression. We can get it by evaluating the operand with `static_cast`.
- nested calls: (X) the top-level call is a call to a function-pointer to #2, so that is what is returned, but since obtaining the value of the function pointer requires evaluation, this is ill-formed. Getting the type is fine.
- Synthesized operators (Y) - these are not functions that we can take pointers to, so unless we "force-manufacture" one, we can't make this work.

3.2 Design question about pointers to virtual member functions

This paper is effectively a counterpart to `std::invoke` - give me a pointer to the thing that would be invoked by this expression, so I can do it later.

This poses a problem with pointers to virtual member functions obtained via explicit access. Observe:

```
struct B {
    virtual B* f() { return this; }
};
struct D : B {
    D* f() override { return this; }
};
void g() {
    D d;
    B& rb = d; // d, but type is ref-to-B

    d.f();    // calls D::f
    rb.f();   // calls D::f
    d.B::f(); // calls B::f

    auto pf = &B::f;
    (d.*pf)(); // calls D::f (!)
}
```

This begs the question: should there be a difference between these three expressions?

```
auto b_f = declcall(d.B::f()); // (1)
auto rb_f = declcall(rb.f());  // (2)
auto d_f = declcall(d.f());    // (3)
```

Their types are not in question. (1) and (2) certainly should have the same type (`B* (B::*) ()`), while (3) has type (`D* (D::*) ()`).

However, what about when we use them?

```
// (d, rb, b_f, rb_f, d_f as above)
(d.*rb_f)(); // definitely calls D::f, same as rb.f()
(d.*d_f)();  // definitely calld D::f, same as d.f()
(d.*b_f)(); // does it call B::f or D::f?
```

It is the position of the author that `(x.*declcall(x.Base::f()))()` should call `Base::f`, because INVOKE should be a perfect inverse.

However, this kind of pointer to member function currently does not exist, although it's trivially implementable. Its type would not be distinguishable from the current kind.

EWG should vote on this.

3.3 Alternatives to syntax

We could wait for reflection in which case `declcall` is implementable when we have expression reflections.

```
namespace std::meta {
  template<info r> constexpr auto declcall = []{
    if constexpr (is_nonstatic_member(r)) {
      return pointer_to_member<[:pm_type_of(r):]>(r);
    } else {
      return entity_ref<[:type_of:]>(r);
    } /* insert additional cases as we define them. */
  }();
}

int f(int); //1
int f(long); //2
constexpr auto fptr_1 = [: declcall<^f(1)> :]; // 1
```

It's unlikely to be quite as efficient as just hooking directly into the resolver, but it does have the nice property that it doesn't take up a whole keyword.

It *also* currently only works for constant expressions, so it's not general-purpose. For general arguments, one would need to pass reflections of arguments, and if those aren't constant expressions, this gets really complicated. `declcall` is far simpler.

Many thanks to Daveed Vandevoorde for helping out with this example.

3.4 Naming

I think `declcall` is a reasonable name - it hints that it's an unevaluated operand, and it's how I implemented it in clang.

codesearch for declcall comes up with zero hits.

For all intents and purposes, this facility grammatically behaves in the same way as `sizeof`, except that we should require the parentheses around the operand.

We could call it something other unlikely to conflict, but I like `declcall`

- `declcall`
- `declinvoke`
- `calltarget`
- `expression_targetof`
- `calltargetof`
- `decltargetof`
- `resolvetarget`

4 Usecases

Broadly, anywhere where we want to type-erase a call-expression. Broad uses in any type-erasure library, smart pointers, ABI-stable interfaces, compilation barriers, task-queues, runtime lifts for double-dispatch, and the list goes on and on and on and …

4.1 What does this give us that we don't have yet

4.1.1 Resolving overload sets for callbacks without lambdas

```cpp
// generic context
std::sort(v.begin(), v.end(), [](auto const& x, auto const& y) {
    return my_comparator(x, y); // some overload set
});
```

becomes

```cpp
// look ma, no lambda, no inlining, and less code generation!
std::sort(v.begin(), v.end(), declcall(my_comparator(v.front(), v.front()));
```

Note also, that in the case of a `vector<int>`, the ABI for the comparator is likely to take those by value, which means we get a better calling convention.

`static_cast<bool(*)(int, int)>(my_comparator)` is not good enough here - the resolved comparator could take `long`s, for instance.

4.1.2 Copy-elision in callbacks

We cannot correctly forward immovable type construction through forwarding function.

Example:

```cpp
int f(nonmovable) { /* ... */ }
struct {
    #if __cpp_expression_aliases < 202506
    // doesn't work
    static auto operator()(auto&& obj) {
        return f(std::forward<decltype(obj)>(obj)); // 1
    }
    #else
    // would work if we also had expression aliases
    static auto operator()(auto&& obj)
       = declcall(f(std::forward<obj>(obj)));      // 2
    #endif

} some_customization_point_object;

void continue_with_result(auto callback) {
    callback(nonmovable{read_something()});
}

void handler() {
    continue_with_result(declcall(f(nonmovable{}))); // works
    // (1) doesn't work, (2) works
    continue_with_result(some_customization_point_object);
}
```

4.2 That's not good enough to do all that work. What else?

Together with [P2826R0], the two papers constitute the ability to implement *expression-equivalent* in many important cases (not all, that's probably impossible).

[P2826R0] proposes a way for a function signature to participate in overload resolution and, if it wins, be replaced by some other function.

This facility is the key to *finding* that other function. The ability to preserve prvalue-ness is crucial to implementing quite a lot of the standard library customization points as mandated by the standard, without compiler help.

5 Guidance Given:

5.1 Reflection?

Do we want to punt the syntax to reflection, or is this basic enough to warrant this feature? (Knowing that reflection will need more work from the user to get the pointer value).

SG7 said no, this is good. So has EWG.

5.2 Unevaluated operands only?

Do we care that it only works on unevaluated operands? (With the `static_cast` fallback in runtime cases)

SG7 confirmed author's position that this is the correct design, and so has EWG.

6 Proposed Wording

We base the approach on 7.6.1.3 [expr.call], which distinguishes calls to lvalues that refer to functions, and prvalues of function pointer type. We must then also handle operators which resolve to a call to a function.

In the table of keywords, in [lex.key], add

`declcall`

In 7.6.2.1 [expr.unary.general]

unary-expression:

  …

  `alignof ( type-id )`

  `declcall ( expression )`

Add new section under 7.6.2.6 [expr.alignof], with a stable tag of [expr.declcall].

1 The `declcall` operator yields a pointer to the function or member function which would be invoked by its *expression*. The operand of `declcall` is an unevaluated operand.

2 If *expression* is not a function call (7.6.1.3 [expr.call]), but is an expression that is transformed into an equivalent function call (12.2.2.3 [over.match.oper]/2), replace *expression* by the transformed expression for the remainder of this subclause. Otherwise, the program is ill-formed.

Such a (possibly transformed) *expression* is of the form *postfix-expression* (*expression-list*$_{opt}$).

3 If *postfix-expression* is a prvalue of pointer type (7.6.1.3 [expr.call]/1), the `declcall` expression yields an unspecified value of the same type as *postfix-expression*, and the `declcall` expression shall not be potentially-evaluated.

4 Otherwise, let the *F* be the function selected by overload resolution (12.2.2.2 [over.match.call]).

(4.1) — If *F* is a surrogate call function (12.2.2.2.3 [over.call.object]/2), the `declcall` expression yields an unspecified value of type pointer to *F*, and the `declcall` expression shall not be potentially-evaluated.

(4.2) — Otherwise, if *F* is a constructor, destructor, synthesized candidate (7.6.9 [expr.rel], 7.6.10 [expr.eq]), or a built-in operator, the program is ill-formed.

(4.3) — Otherwise, if *F* is an implicit object member function, the result is a pointer to member function denoting *F*.

If the *id-expression* in the class member access expression of this call is a *qualified-id*, the resulting pointer to member function points to *F*, bypassing virtual dispatch (see compare with 7.6.1.3 [expr.call]/2).

[Example:

```
struct B { virtual B* f() { return this; } };
struct D : B { D* f() override { return this; } };
void g() {
  D d;
  B& rb = d;
  auto b_f = declcall(d.B::f()); // type: B* (B::*)()
  auto rb_f = declcall(rb.f());  // type: B* (B::*)()
  auto d_f = declcall(d.f());    // type: D* (D::*)()
  (d.*b_f)();  // B::f
  (d.*rb_f)(); // D::f, via virtual dispatch
  (d.*d_f)();  // D::f, via virtual dispatch
}
```

– end example]

(4.4) — Otherwise, when *F* is an explicit object member function, static member function, or function, the result is a pointer to *F*.

(4.5) — Otherwise, the program is ill-formed.

Into 13.8.3.2 [temp.dep.type], add to the end of p10:

(10.13) — denoted by `decltype(`*`expression`*`)`, where *expression* is type-dependent.

(10.14) — denoted by `declcall(`*`expression`*`)`, where *expression* is type-dependent.

Into 13.8.3.4 [temp.dep.constexpr], add to list in 2.6:

```
…
noexcept ( type-id )
declcall ( expression )
```

Add feature-test macro into 17.3.2 [version.syn] in section 2

```
#define __cpp_declcall 2025XXL
```

Modify the index of grammar productions as appropriate to reflect the new grammar production.

7 Acknowledgements

— Daveed Vandevoorde for many design discussions
— Joshua Berne, Davis Herring, Barry Revzin, Christoph Meerwald, and Brian Bi for core language review, changes, and suggestions

8 References

[N4856] David Sankel. 2020-03-02. C++ Extensions for Reflection.
https://wg21.link/n4856

[P1240R1] Daveed Vandevoorde, Wyatt Childers, Andrew Sutton, Faisal Vali, Daveed Vandevoorde. 2019-10-08. Scalable Reflection in C++.
https://wg21.link/p1240r1

[P2087R0] Mihail Naydenov. 2020-01-12. Reflection Naming: fix reflexpr.
https://wg21.link/p2087r0

[P2237R0] Andrew Sutton. 2020-10-15. Metaprogramming.
https://wg21.link/p2237r0

[P2300R6] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. 2023-01-19. ‘std::execution‘.
https://wg21.link/p2300r6

[P2320R0] Andrew Sutton, Wyatt Childers, Daveed Vandevoorde. 2021-02-15. The Syntax of Static Reflection.

https://wg21.link/p2320r0

[P2826R0] Gašper Ažman. Replacement functions.
https://wg21.link/P2826R0