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

Document #:	P2825R5
Date:	2025-06-20
Project:	Programming Language C++
Audience:	EWG CWG

Reply-to:	Gašper Ažman <gasper.azman@gmail.com> Bronek Kozicki <brok@spamcop.net>

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 Revision History

0. new paper!
1. seen as `calltarget` in EWGi
2. `calltarget` was a bad name, and design refined
3. Seen in EWG as `declcall`, well received, must come back
4. Added core wording and got it reviewed, more revisions. Added devirtualized pointers to member functions.
5. — Expanded motivation section
   — resolved design question re. devirtualized pointers
   — corrections from wording review from CWG, specifically Hubert Tong's
   — New section for EWG: resolve questions around comparison between devirtualized pointer values and `final` functions
   — added section about implementation experience

3 Motivation and Prior Art

The language facilities for overload resolution that aren't the actual call syntax have severe limitations.

The facilities available as of C++23 are:

— assignment to a variable or parameter of a given function pointer type
— `static_cast<function_pointer_type>(function_identifier)`
— `&function_name` (if not overload set)

All of these are unsuitable for ad-hoc type-erasure that library authors (such as [P2300R6]) need whenever implicit conversions may be involved in any form.

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
  — cause unnecessary template instantiations through value-category combinations of deduced parameters
— 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 a pointer to it.*

This is what this paper is trying to provide.

3.1 Related Work

3.1.1 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.

3.1.2 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).

4 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 a function (such as when it is a constructor, destructor, built-in, etc.). Inventing functions should be a facility built in a library *on top* of `declcall`.

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

`declcall(expr)` should always be a compile-time constant, and `expr` always unevaluated. If the expression is valid, but figuring out the function pointer would have required evaluating parts of the expression, such as when calling through a pointer to function or a surrogate function, we specify that `declcall(expr)` is only legal to use inside other unevaluated contexts that only care about the type, such as `sizeof()` and `decltype()`. Specifically, we say that in such cases, it is not a constant expression, and its value is unspecified.

```
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<decltype(declcall(fptr(2)))>(fptr); // OK, fptr, though redundant
struct T {
    constexpr operator fptr_t() const { return fptr; }
};
static_cast<decltype(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(short{1}));              // 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<decltype(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)
}
```

4.1 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!

— (G) — we are resolving to #4 even through a call to a derived class (`u`)

— resolving inline friend a.k.a. “hidden friend” (E)

— 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). (7.6.1.1 [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.

4.2 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 calls 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 complement to `declcall`.

However, this kind of pointer to member function currently does not exist, although it's trivially implementable (GCC has an extension: https://gcc.gnu.org/onlinedocs/gcc-14.2.0/gcc/Bound-member-functions.html). The author proposes to add the "devirtualized member function pointer"

Unlike the GCC extension, the type of the devirtualized member function pointer is not distinguishable from the regular member function pointer, to enable generic programming with the regular pointer-to-member-function call syntax.

4.2.1 Pointers to pure virtual functions

Conciser the following:

```
struct ABC {
    virtual int f() = 0;
};

void h() {
    auto x = declcall(declval<ABC&>().f()); // 1
    auto y = declcall(declval<ABC&>().ABC::f()); // 2
}
```

What is (1)? What is (2)? Should (2) be ill-formed? Return a regular non-devirtualized pointer-to-member-function?

No. Unfortunately, pure virtual functions can actually be defined;

```
int ABC::f() { return 1; } // OK!

struct B : ABC {
    int f() { return 2; }
};

int g() {
    B b;
    b.f(); // OK, 2
    b.ABC::f(); // OK, 1
}
```

We propose that the following happens:

```
int h() {
    B b;
    auto x = declcall(declval<ABC&>().f());
    auto y = declcall(declval<ABC&>().ABC::f());
    (b.*x)(); // OK, 2, does virtual dispatch
    (b.*y)(); // OK, 1, does not do virtual dispatch
}
```

4.2.2 Comparison of devirtualized values of pointer-to-member-function type

Comparison of values of pointer-to-member-function type where either operand points to a `virtual` function is unspecified in c++23 (7.6.10 [expr.eq]).

We have the ability to define it further.

Specifically, if both operands are devirtualized pointers to members, the comparison should be performed under the current rules for pointers to nonvirtual member functions; virtual and nonvirtual pointers to member functions should compare unequal.

We could also define taking the address of member functions marked `final` to yield devirtualized pointers-to-member-functions, but that would be a separate paper.

EWG question: Is EWG interested in such a paper?

4.3 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.

4.4 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. EWG voted for it as well.

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`

`declcall` can be thought of as "resolve this call expression to the declaration of the called entity".

5 Implementation experience

Hana Dusíková implemented the proposal in a clang fork along with some other extensions, an example of which can be found here: https://compiler-explorer.com/z/71T6GoeMo.

The implementation took her approximately 2 afternoons. The proposal was implemented for the constexpr evaluator, and has an additional capability where base expressions that are constant expressions are not ill-formed, but instead return that constant, getting around the requirement for `static_cast`.

There were no issues with the implementation of devirtualized function pointers, at least on the Itanium ABI (they are already part of the ABI). Microsoft's ABI readily admits an implementation as well, and we have confirmation of this from Microsoft.

6 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 …

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

6.1.1 Resolving overload sets for callbacks without lambdas

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

becomes

```
// 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.

6.1.2 Copy-elision in callbacks

We cannot correctly forward immovable type construction through forwarding function.

Example:

```
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);
}
```

6.1.3 Taking the address of a hidden friend function

The language has currently no mechanism to take a pointer to a "hidden friend" function.

```
struct S {
  inline friend int foo(S, int a) { return a; }
};

constexpr int(*foo_ptr)(S, int) = &S::foo; // error, no S::foo
declcall(foo(S{}, 1)); // ok, S::foo.
```

6.1.4 No way to skip virtual dispatch when calling through pointer-to-member-function

There is also no tool skip the virtual dispatch of a pointer to virtual member function (later in the paper called "devirtualized pointer").

```
struct B {
    virtual char f() { return 'B'; }
};
struct D : B {
    char f() override { return 'D'; }
};
```

If we have an object of type `D`, we can still explicity invoke `B::f`, but there is no way to do that via a pointer-to-member-function in C++23.

```
D x;
x.B::f(); // 'B'
x.f();    // 'D'
x.*(&B::f)();  // 'D' (!)
```

6.2 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.

7 Guidance Given:

7.1 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.

7.2 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.

8 Proposed Wording

In the table of keywords, in 5.12 [lex.key], add

```
declcall
```

In 6.3 [basic.def.odr], clarify that `declcall` also "names" functions in the same way that taking their address does.

(4.1) A function is named by an expression or conversion if

- it is the selected member of an overload set (6.5 [basic.lookup], 12.2 [over.match], 12.3 [over.over]) in an overload resolution performed as part of forming that expression or conversion, unless it is a pure virtual function and either the expression is not an *id-expression* naming the function with an explicitly qualified name or the expression forms a pointer to member (7.6.2.2 [expr.unary.op])~~.~~; or
- it is the result of a `declcall` expression, unless the result is a pointer-to-member-function that is not devirtualized.

[Note 2: This covers taking the address of functions (7.3.4 [conv.func], 7.6.2.2 [expr.unary.op], [expr.declcall]), calls to named functions (7.6.1.3 [expr.call]), operator overloading (12 [over]), user-defined conversions (11.4.8.3 [class.conv.fct]), allocation functions for new-expressions (7.6.2.8 [expr.new]), as well as non-default initialization (9.4 [dcl.init]). A constructor selected to copy or move an object of class type is considered to be named by an expression or conversion even if the call is actually elided by the implementation (11.9.6 [class.copy.elision]). — end note]

In 7.6.2.1 [expr.unary.general]

unary-expression:

…

`alignof ( type-id )`

`declcall ( expression )`

In 7.6.10 [expr.eq], edit paragraph 4.3

(4.3) — If either is a pointer to a virtual member function and not devirtualized, the result is unspecified. [Note: If both are devirtualized, they are treated as pointers to non-virtual member functions for the remainder of this paragraph. – end note]

In 9.3.4.4 [dcl.mptr], insert a new paragraph 5 after p4, and renumber section:

5 The value of a pointer to member function can be *devirtualized*. [Note: Devirtualized pointers to member functions may be formed by `declcall` ([expr.declcall]), and call expressions involving them call the function they point to, and not the final overrider (7.6.1.3 [expr.call], 11.7.3 [class.virtual]). – end note]

In 7.6.1.3 [expr.call], change p2:

If the selected function is non-virtual, or if the *id-expression* in the class member access expression is a *qualified-id* , or the bound function prvalue (7.6.4 [expr.mptr.oper]) is obtained from a devirtualized pointer (9.3.4.4 [dcl.mptr]), that function is called. Otherwise, its final overrider in the dynamic type of the object expression is called; such a call is referred to as a virtual function call.

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

1 When evaluated, the `declcall` operator yields a pointer or pointer to member to the function or member function which would be invoked by its expression. The result is a prvalue. The operand of declcall is an unevaluated operand. [Note: It can be said that, when evaluated, `declcall` resolves the call to a specific declaration. –end note]

2 If the *expression* is transformed into an equivalent function call (12.2.2.3 [over.match.oper]), replace the *expression* by the transformed expression for the remainder of this subclause. If the *expression* is not a (possibly transformed) function call expression (7.6.1.3 [expr.call]), the program is ill-formed. Such a (possibly transformed) *expression* is of the form *postfix-expression* (*expression-list*$_{opt}$).

3 If the *postfix-expression* is of the (possibly transformed) form `E1 .* E2` (7.6.4 [expr.mptr.oper]) where `E2` is a *cast-expression* of type `T`, then the result has type `T` and the `declcall` expression shall not be potentially-evaluated.

4 Otherwise, if the *postfix-expression* is neither a qualified nor an unqualified function call (12.2.2.2.2 [over.call.func]), then: where `T` is the type of the *postfix-expression*, if `T` is a pointer-to-function type, then the result has type `T`, and "pointer to `T`" otherwise.

5 Otherwise, let F be the function selected by overload resolution (12.2.2.2 [over.match.call]), and `T` its type.

(5.1) — If F is a constructor, destructor, or synthesized candidate (7.6.9 [expr.rel], 7.6.10 [expr.eq]), the program is ill-formed.

(5.2) — Otherwise, if F is an implicit object member function declared as a member of class `C`, the *postfix-expression* is a class member access expression (7.6.1.5 [expr.ref]). The result has type "pointer to member of class `C` of type `T`" and points to F. If the *id-expression* in the class member access expression of this call is a *qualified-id*, the result pointer is a *devirtualized pointer*. (6.8.4 [basic.compound]).

[Example:

```
struct B { virtual B* f() { return this; } };
struct D : B { D* f() override { return this; } };
struct D2 : B {};
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::*)()
   declcall(D2().f());            // type: B* (B::*)()
   (d.*b_f)();  // B::f, devirtualized
   (d.*rb_f)(); // D::f, via virtual dispatch
   (d.*d_f)();  // D::f, via virtual dispatch
}
```

– end example]

(5.4) — Otherwise, the result has type "pointer to `T`" and points to F.

In 11.7.4 [class.abstract], turn the text about when a pure virtual function need be defined into a note, and fill out the note:

2 A virtual function is specified as a *pure virtual function* by using a *pure-specifier* (11.4 [class.mem]) in the function declaration in the class definition.

[Note 2: Such a function might be inherited: see below. — end note]

A class is an abstract class if it has at least one pure virtual function.

[Note 3: An abstract class can be used only as a base class of some other class; no objects of an abstract class can be created except as subobjects of a class derived from it (6.2 [basic.def], 11.4 [class.mem]). -— end note]

[Note 4: A pure virtual function need be defined only if called with, or as if with (11.4.7 [class.dtor]), the qualified-id syntax (7.5.5.3 [expr.prim.id.qual]). A devirtualized pointer to member function is always obtained from such an expression within a `declcall` expression (9.3.4.4 [dcl.mptr]) –end note].

In 13.6 [temp.type] add:

2 Two values are template-argument-equivalent if they are of the same type and

(2.1) — they are of integral type and their values are the same, or
(2.2) — they are of floating-point type and their values are identical, or
(2.3) — they are of type `std::nullptr_t`, or
(2.4) — they are of enumeration type and their values are the same,113 or
(2.5) — they are of pointer type and they have the same pointer value, or
(2.6) — they are of pointer-to-member type and they refer to the same class member and they are either both devirtualized or both not devirtualized, or are both the null member pointer value, or
(2.7) — they are of reference type and they refer to the same object or function, or
(2.8) — they are of array type and their corresponding elements are template-argument-equivalent,114 or
(2.9) — they are of union type and either they both have no active member or they have the same active member and their active members are template-argument-equivalent, or
(2.10) — they are of a closure type (7.5.6.2 [expr.prim.lambda.closure]), or
(2.11) — they are of class type and their corresponding direct subobjects and reference members are template-argument-equivalent.

Into the list in (13.8.3.4 [temp.dep.constexpr]) paragraph 2.6, add `declcall` alongside `alignof`, to ensure that the `declcall` expression is value-dependent only if it is type-dependent (and thus not value-dependent if all the type information in the expression is known).

2 An *id-expression* is value-dependent if

— …
(2.6) — it names a potentially-constant variable (7.7 [expr.const]) that is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the *unary-expression* or *expression* is type-dependent or the *type-id* is dependent:

`sizeof` *unary-expression*
`sizeof` (*type-id*)
`typeid` (*expression*)
`typeid` (*type-id*)
`alignof` (*type-id*)
`declcall ( expression )`

[Note 1: For the standard library macro offsetof, see [support.types]. -— end note]

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

```
#define __cpp_declcall 2025XXL
```

Add to Annex C:

> **Affected subclause:** 5.12 [lex.key]
> **Change:** New keyword.
>
> **Rationale:** Required for new features.
>
> **Effect on original feature:** Added to Table 5, the following identifier is now a new keyword: `declcall`. Valid C++ 2023 code using these identifiers is invalid in this revision of C++.

9 Acknowledgements

— Alex Kremer for reviews,
— Daveed Vandevoorde for many design discussions and implementation guidance
— For core language review, changes, and suggestions (alphabetized): Barry Revzin, Brian Bi, Christof Meerwald, Davis Herring, Hubert Tong, Jens Maurer, Joshua Berne, and Roger Orr.

10 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