

# Overload Set Types

Document #: DxxxxR0  
Date: 2024-04-20  
Project: Programming Language C++  
Audience: EWGI  
Reply-to: Bengt Gustafsson  
<[bengt.gustafsson@beamways.com](mailto:bengt.gustafsson@beamways.com)>

## 1 Abstract

This proposal defines a type for each overload set of more than one function. Unique such *overload-set-types* are created each time a placeholder type is *deduced* from an overloaded function name. These types have no runtime state, in contrast with function pointers. This proposal does not specify any new keywords or operators, it just expands what placeholder types can be deduced from.

An object of *overload-set-type* can be called like the function overloads it represents, and overload resolution works exactly the same as if the overloaded function is called directly at the *point of deduction*. This includes overloads found by ADL and defaulted function parameters. Additionally, an object of *overload-set-type* can be implicitly converted to the function pointer type of any of the overloaded functions it represents.

```
auto callWithFloat(auto f)
{
    double (*dfp)(double) = f;    // The appropriate overload of f is selected, error if none.
    return f(3.14f);              // If f is overloaded overload resolution occurs here.
}

// As std::sin is overloaded the type of f above is an overload set type for std::sin at this
// point of translation
float x = callWithFloat(std::sin);
```

As this feature only relies on compile time overload resolution it works also for constructors, destructors, operators and member functions. For member functions and destructors the member function pointer call syntax must be used, while for constructors and operators regular function call syntax is used.

## 2 Motivation

Today you can't use functions that are overloaded when the type is deduced, only functions that are not overloaded. This is annoying for instance with algorithms such as `std::transform` which often require wrapping a function in a lambda just because it is overloaded. With this proposal any function, overloaded or not, can be used in such scenarios.

### 2.1 Natural predicate syntax

Here are some examples involving `std::transform`.

```
std::vector<float> in = getInputValues();

std::vector<float> out;
std::transform(in.begin(), in.end(), std::back_inserter(out), @__std::sin__@);
```

```

// Or with ranges
auto out_r = std::views::all(in) |
             std::views::transform(@__std::sin__@) |
             std::ranges::to<std::vector>();

// Also works with operators
auto neg_r = std::views::all(out_r) |
             std::views::transform(@__operator-__@) |
             std::ranges::to<std::vector>();

```

## 2.2 Even more perfect forwarding

Another problem that this proposal solves is that function pointers don't work with *perfect forwarding* when passing an overloaded function. Here is an example involving `std::make_unique`.

```

class MyClass {
public:
    MyClass(float (*fp)(float));
};

auto ptr = std::make_unique<MyClass>(@__std::sin__@); // Works with this proposal!

```

Today `make_unique` can't be called with `std::sin` as argument although `MyClass` can be constructed with it. This is as the `auto&&` placeholder type of `make_unique` can't be deduced from a an overloaded function name.

## 2.3 Helping contract condition testing

This proposal also solves the issue encountered in [P3183R0] where `declcall` of [P2825R2] and macros was employed to allow overloaded functions to be tested by its `check_preconditions` and `check_postconditions` functions. This proposal also allows operator, constructor and destructor contracts to be tested without further compiler magic.

```

// P3183R0
#define CHECK_PRECONDITION_VIOLATION(F, ...) \
CHECK(!check_preconditions<__builtin_calltarget(F(__VA_ARGS__))>(__VA_ARGS__))

CHECK_PRECONDITION_VIOLATION(myOverloadedFunction, 1, "Hello");

// With this proposal:
CHECK(!check_preconditions<myOverloadedFunction>(1, "Hello"));

// And also
CHECK(!check_preconditions<&MyClass::func>(MyClass{}, 1, "Hello"));
CHECK(!check_preconditions<operator+>(MyClass{}, 1));
CHECK(!check_preconditions<MyClass::MyClass>(1, "Hello"));
CHECK(!check_preconditions<MyClass::~~MyClass>());

```

## 3 Proposal

This proposal allow placeholder types to be deduced from function names for overloaded functions. The proposal has no effect for functions that are not overloaded, these deduce the placeholder type to the function's function reference type as today, and sets the value to the function pointer.

The idea is that all aspects of the original function overload set is preserved when applying the function call operator to an object of *overload-set-type*.

Each time a placeholder type is deduced from the name of an overloaded function a unique conceptual *overload-set-type* is created. This type has a generic call operator calling the function and a conversion operator to each function pointer type. The *overload-set-type* as such is regular and all instances compare equal. All overload resolution is done at compile time, only depending on the *overload-set-type*. In contrast with a function pointer there is no runtime data to carry around, so the actual function calls compiled into the object code doesn't do runtime dispatch (except virtual dispatch).

Here is an example assuming that `std::sin` has two overloads:

```
// Exposition only
struct __std_sin_overload_set_type_1 {
    decltype(auto) operator()(auto&&... as) {
        return @__std::sin_@ (std::forward<decltype(as)>(as)...);
    }

    @implicit_@ operator float (&)(float) const { return @__std::sin_@; }
    @implicit_@ operator double (&)(double) const { return @__std::sin_@; }
};
```

Note the trailing `_1` on the struct's name. This is to indicate that each time a placeholder type is deduced from `std::sin` a new *overload-set-type* is created. The intent is for this mechanism to work exactly as for lambdas today, where each lambda has unique type even if its definition is the same. The rules are also the same as for lambdas when defined in header files consumed by multiple TUs.

The analogy with lambdas goes only thus far, a compiler is required to elide the generic call operator and function pointer cast operators. In reality the *overload-set-type* is just a way to express that the compiler remembers the contents of the overload set at the point of deduction. This is why the cast operators are marked *implicit* in the example, they don't actually count as user defined conversions in overload resolution.

Although the exact type of each unique *overload-set* is unnamed (just like a lambda) it can be retrieved using `decltype` (just like a lambda). As there are no data members an *overload-set* is always default constructible, copyable and assignable, but only within each *overload-set-type*, which means that you can't change the contents of the overload set by assignment.

Here are some examples of code valid with this proposal:

```
void compose(auto F, auto G, auto value) { return F(G(value)); }

double one = compose(std::tan, std::atan, 1);

auto s = std::sin;

using SinOverloads = decltype(s);

callWithT(SinOverloads(), 3.15);

void cc(float (*f)(float));

cc(s);

auto sptr = @declcall_@(s(2.0f));

cc(sptr);

auto sptr2 = static_cast<float(*)>(s);

cc(sptr2);
```

```
double x = s(3.14);

auto(std::sin);

auto{std::sin};

decltype(std::sin);

template<auto F> void myFun()
    requires requires { F(1.2, "Hello"); }
{
    F(2.3, "Bye");
}
```

### 3.1 Defaulted parameters

As the intent with this proposal is to be able to call an object of overload set type *exactly* as the function the type was deduced from defaulted parameters are considered in the overload resolution process. By the same convention, when converting an object of overload set type to a function pointer defaulted parameters are *not* considered.

### 3.2 Template functions

If the overload set contains template functions these are included in the *overload-set-type* and selected by overload resolution as usual. Specializations of template functions declared after the point of deduction are not considered but if a matching template instantiation is selected the program is ill-formed just like when any template is explicitly specialized after the compiler has already instantiated the base template. Just like with lambdas this can happen if an explicit specialization is declared between the point of deduction and the point of call, not only if the explicit specialization is declared after the point of call.

### 3.3 Specifiers such as noexcept and constexpr

In keeping with the general idea all specifiers on functions in the overload set are carried over to the *overload-set-type* and work the same as if the function name was used directly.

### 3.4 Free functions

Overload sets for free functions include overloads found by ADL which means that an *overload-set-type* is even more magic and requires the compiler to remember the entire state of the symbol tables at the point of deduction. This is the same rule as for generic lambdas containing function calls.

The function call operator can be applied to an object of *overload-set-type* just as if the original function was called. Overload resolution also happens when a *overload-set-type* object is used inside a `declcall` construct of [P2825R2] while `static_cast` and binding to a function pointer or reference just selects one overload with matching signature if one exists.

### 3.5 Member functions

The proposed feature works also when a placeholder type is deduced from an overloaded member function name, except that to call a member function the `(ref.*f)(args...)` or `(ptr->*f)(args...)` syntax must be used, just as if the member function was not overloaded. Overload resolution works exactly as for the overloaded member function the object of overload set type represents. Likewise, an object of overload set type can be implicitly converted to the member function pointer type of any of the member function overloads it represents.

When an *overload-set-type* is deduced from a *member* function name it can always be used with an object reference, even if it contains `static` overloads. This ensures that overload resolution works the same as if the member function was called directly.

When a virtual function is selected by overload resolution the call is dispatched virtually even though the member function pointer call syntax is used, which today just calls the member function provided in the member function pointer.

For overload sets that contain static member functions it is also possible to use regular function call syntax, and just as when using the `Class::function` syntax this fails if a non-static function is selected by overload resolution.

Further, converting the object of overload set type to a member function pointer type works the same as when converting a member function name directly, but if the function pointer type is for a free function there must be a matching *static* member function or member function with explicit object reference in the overload set.

### 3.5.1 Allowing `std::invoke` with all *overload-set-types*

`std::invoke` and similar functions should work as expected for *overload-set-types* created from member function overload sets as well as for free function overload sets if the invoke overload selection is made based on constraints rather than on matching function pointer or member function pointer signatures. As constraints are relatively new it is likely that many `std::invoke` implementations will have to be rewritten to allow calling with an object of *overload-set-type* as the first argument. There should be no ABI breakage caused by this type of change as the selected overload of any `std::invoke` call that can be made today has the same signature.

## 3.6 Constructors

An *overload-set-type* can be created by deducing a placeholder type from a constructor name in the form `&Class::Class`. Objects of type `Class` can subsequently be created by applying the call operator to the an object of this overload set type. As constructors don't have function pointer types there is no possibility to convert an *overload-set-type* deduced from a constructor to a "constructor pointer".

As non-overloaded constructors don't have function pointer types even non-overloaded constructors deduce to overload set types.

## 3.7 Destructors

Even though destructors are not overloadable *overload-set-types* can be deduced from destructor names in the form `&Class::~Class` as there is no corresponding function pointer type. Objects of the deduced *overload-set-type* contain one overload which can be called just like a parameterless member function pointer and which can't be converted to a "destructor pointer" as there is no such thing.

## 3.8 Operators

An *overload-set-type* can be created by deducing a placeholder type from an operator in the form `operator@` for each *overloadable-operator* `@`. When an *overload-set-type* is deduced from an overloaded operator it follows the rules of overload resolution for the operator, which means that both member function and free function operators are considered, except when explicitly scoping the operator to a class or namespace. This behavior is consistent with using the *overloadable-operator* inside a lambda. This proposal does not support `operator->`, `operator->*` which can't be overloaded or overloaded conversion operators which can't be spelled in the grammar. Whether `operator`, `operator new`, `operator delete`, `operator""` and `operator co_await` can and should be supported is still not properly investigated.

An object of an *overload-set-type* deduced from an operator can be called with one or two arguments as applicable for the operator. Overload sets for operators that have both unary and binary forms contain both unary and binary overloads found at the point of deduction. When a placeholder type is deduced from an unqualified operator name all available overloads, including those for fundamental types and those found via ADL are

included in the overload set. This ensures that calling an object of such an *overload-set-type* performs overload resolution in exactly the same way as if the operator had been called using the form such as `lhs @ rhs` or `@ arg`. Note that this is *not* the same as the call `operator@(lhs, rhs)` or `operator@(arg)` which only considers operators declared as free functions.

The prefix and postfix versions of `++` and `--` are treated as they usually are; the postfix version requires an extra argument convertible to `int`.

The inconsistency introduced above means that there is no way to deduce an overload set type that contains only the free function operator overloads as the syntax `operator@` is already used. This drawback seems acceptable as it is much more useful to be able to denote an entire overload set regardless of whether individual operators are declared as free functions (including friends) or member functions. A possible alternative is discussed in the next chapter.

If an *overload-set-type* is deduced from a qualified operator name such as `MyClass::operator@` only the operator overloads declared in the class (with bases) are included. Note that as `operator[]` and `operator()` can only be declared as member functions they can only be deduced from a qualified operator name, never from just the operator name. *This may be the wrong decision, it doesn't seem that these actually differ from other operators where the complete set of overloads declared as members must be conveyed to the overload-set-type. However, intuitively it seems strange to pass operator[] to a function to let it index anything.*

## 4 Possible extensions and alternatives

### 4.0.1 Allowing member function references

Today there is no such thing as a member function reference. Thus we for consistency require the qualified member function name to be explicitly converted to a member function pointer using a prefix `&` operator. This is what this proposal suggests.

Introducing member references could be a good idea, but it is another proposal. This has its own complications as the rules for automatic conversion from free functions to function references and function pointers are somewhat contrived due to historical reasons, and decisions have to be made as to if member pointers and references should work the same or deviate somehow.

### 4.0.2 Calling non-static member functions using regular function call syntax.

It would be possible to allow using regular function call syntax on objects with *overload-set-type* even if the type is deduced from an overloaded member function. Providing the implicit object reference as the first argument would be required at each call site, just as for `std::invoke`. If the overload set contains static member functions these would then be represented by two synthesized overloads in the *overload-set-type* which causes a risk for ambiguity not present when calling the static member function directly.

The advantage of this idea is that the user of the *overload-set-type* object does not have to know if the *overload-set-type* was deduced from a free function or a member function. This is a very limited form of *unified function call syntax* and does not solve the poster child use case for UFCS where for instance `begin` and `end` can either be free functions taking an object or a member of that object's type.

To make this functionality useful it would have to be made possible to call a *member function pointer* as a regular function, providing the implicit object reference explicitly. Otherwise we get functionality that *only* works if the member function is overloaded. On the flip side we today have a situation where static, explicit object reference and non-static member functions work differently in this respect, which such a feature would unify.

```
struct MyClass {
    void f();
    static void g();
    void h(this MyClass& o);
};
```

```

auto fp = &MyClass::f;
auto gp = &MyClass::g;
auto hp = &MyClass::h;

MyClass o;

(o.*fp)();      // Ok
(o.*gp)();      // Error
(o.*hp)();      // Error
fp(o);          // Error
gp(o);          // Error
hp(o);          // ok

gp();           // ok

```

For consistency with the overloaded case as defined above all possibilities for calling must be allowed, where the calls to `g` just ignore the object reference.

As extending rules for member function pointers in this way is a prerequisite for extending overload-set-type objects with free function callability this is not included in this proposal. Instead this would be an addition to a UFCS proposal to make sure it works consistently between named functions and *overload-set-type* objects.

#### 4.0.3 Deduction from operator tokens

To avoid above mentioned inconsistency for operators, where a direct call of `operator@` does not correspond to the function call operator applied to an *overload-set-type* deduced from `operator@` it would be nice to be able to deduce an placeholder type from the *operator-token* itself. Then we could write code like this:

```

std::vector<float> in = getInputValues();

std::vector<float> out;
std::transform(in.begin(), in.end(), std::back_inserter(out), @__-__@); // Invert values

auto plusses = @__+__@;          // All overloads of +
auto free_plusses = operator+;    // Only free function overloads of +

```

This functionality is the same for operators that can be defined both as free functions and member functions and for those that can only be defined as member functions.

To make this work a new production in the *assignment-expression* rule can be added. As this production only contains an *operator-token* there is no risk of parsing ambiguity. The only strain on the compiler is that if an expression starts with an *operator-token* the next token must be checked to see if is a comma, semicolon, right parenthesis, bracket or brace. If so select the *operator-token* only case whereas for any other token parsing proceeds as today, eventually consuming the *operator-token* in *unary-expression*.

```

assignment-expression:
    conditional-expression
    logical-or-expression assignment-operator initializer-clause
    throw-expression
    operator-token

```

It would be possible to forbid expressions consisting of only an *operator-token* when not used to deduce a placeholder type, but this seems complicated wording wise, and compilers usually have warnings like *statement has no effect*, in these cases which may be sufficient to filter out the case where a stray *operator-token* is mistakenly typed.

The reason for placing this production in the *assignment-expression* rule is to be able to use *overloaded-operators*



as function arguments. If it was placed in the *expression* rule any usage as a function argument would have to be enclosed in parentheses. A third option is to *mandate* surrounding parentheses at all use by instead introducing a new production in *primary-expression* adding a third type of parentheses along with fold expressions and nested expressions. This approach may be safer and the extra parenthesis highlights that something special is going on.

```
std::vector<float> in = getInputValues();

std::vector<float> out;
std::transform(in.begin(), in.end(), std::back_inserter(out), @__(-)__@); // Invert values

auto plusses = @__(+).__@;           // All overloads of +
auto free_plusses = operator+;       // Only free function overloads of +
```

An expression of the form `(@)` will evaluate to an instance of an *overload-set-type* regardless of whether a placeholder type is deduced from it. This seems necessary as there is no other type this expression could have. This means that by coincidence it is now possible to write `(@)(lhs, rhs)` or `(@)(arg)` due to how the grammar works.

An advantage of this formulation is that we can still never write two consecutive commas, whereas with the production in *assignment-expression* we can write `auto x = ,,1;` as the first comma is an *assignment-expression* but then the *expression* production including the comma operator is employed so the second comma causes the first comma to be ignored and `x` is initialized to the integer 1.

*NOTE: As this changes the meaning of deducing a placeholder type from `operator@` this addition must be included from the beginning and can not be added later. For this reason the author would prefer for this option to be added included in the proposal.*

#### 4.0.4 Supporting overloaded conversion functions

It may be possible to create an *overload-set-type* from all the conversion operators of a class, but this would require some specific new syntax such as `operator typename` to be able to denote this overload set. Overload set types of this variety would be very special just as overloaded conversion operators themselves, as the overload selection happens due to the *required return type* rather than the argument types (which is always the object itself). This type of backwards overload resolution is already implemented in compilers so it doesn't seem overly complicated to allow this too, except for the special keyword parsing.

```
class MyClass {
    operator int() { return 0; }
    operator double() { return 1; }
};

auto conv = &MyClass::operator @**typename**@;

MyClass c;

int a = (c.*conv)();
double a = (c.*conv)();
```

This feature would not be needed for [P3183R0] to test contracts on conversion functions as each overload has its own name which enables for instance:

```
CHECK(!check_preconditions<MyClass::operator int>(myObject));
```

In fact it seems rather unnecessary to support overload sets of conversion operators given that it requires a syntactical extension and seems more risky when it comes to implementation effort depending on compiler internals.



The use of an overload set of conversion functions could be to force a value to be converted to some type, for instance using `transform`. But in this use case you usually don't know if the conversion is done via a conversion function or a converting constructor in the resulting type. To let the compiler select how to convert the value you can already use the destination type as a predicate and the compiler will select to either construct the destination type from the source or employ a conversion operator in the source. (CHECK THIS), probably you have to provide a lambda doing `static_cast` for such selection to take place as in regular code:

```
std::vector<C1> in = getInputValues();

std::vector<C2> out;
std::transform(in.begin(), in.end(), std::back_inserter(out), C2); // does not work, obviously.

std::transform(in.begin(), in.end(), std::back_inserter(out), [](C1 x)->C2 { return x; }); // Works

std::transform(in.begin(), in.end(), std::back_inserter(out), &C2::C2); // Works only for C2 ctor.
```

This proposal does not offer a good solution to this and the old lambda based solution must still be used.

## 4.1 Supporting opt-in UFCS

With some additional building blocks this proposal could support an opt-in version of unified function call syntax. Such functionality could be accomplished by defining a way to merge two overload sets and some way to denote an empty overload set.

For instance if you want to call whichever is available of `begin(object)` and `object.begin()` you can do that today, but it requires name specific coding using SFINAE but this method is very verbose, which can be mitigated by using a macro. For the example name *begin* the code could look like this:

```
template<typename T> auto call_begin(T& object) {
    if constexpr (requires { object.begin(); }) // This impl. prefers members if both exist.
        return object.begin();
    else
        return begin(object); // ADL in effect.
}
```

What we would need help from this proposal for is to not have to code this for each name, but to be able to send the function name in to the function as an extra parameter:

```
template<typename T, typename F> auto call_named(F f, T& object);
```

However, as written, this only works for operators due to the inconsistent definition of the `operator@` name or via the extension mentioned above. What we want here is a way to do the same for any function name. If we can just figure that one out it seems that opt-in UFCS is within reach. Note that with a syntax to represent a function name's free function and member function overloads we would do overload resolution within the set of both free and member functions in the same way it is done for operators today.

This suggests changing the syntax suggested for *overloadable-operators* indicated above to something that can be used for any function name. This does not work for the `(function)` syntax that works for operators as this already has meaning (the same meaning as without the parentheses). Instead we could define a new prefix operator out of the remaining operator tokens that can't be used in a prefix position. Candidates include `|` and `%` and there is also the newly added characters where the backtick could be used to enclose a function name like so: `'function'`. Such quoting would work well for operator tokens too, and would stand out suitably in the source code.

Another and possibly better idea is to provide `ufcs_invoke` as a magic library function which can take an object of *overload-set-type* and any number of additional parameters and then check both for member and free functions named as the function or operator specified when the *overload-set-type* was deduced from said name.

Taken to the extreme this could always happen when deducing a placeholder type from a function name. This means that to call a member or free function begin you just write `auto{begin}(object)`. While weird this would be logical. This however breaks the rule that an object of the *overload-set-type* shall behave exactly as using the function name it was deduced from.

A really exotic idea would be to use `operator function_name` for this purpose. This would work syntactically as the same name can not refer to a type (as today) and a function at the same time in the same scope. The problem is that this relies on the *inconsistency* when deducing from operator tokens, which is something we want to get rid of.

## 5 Implementation experience

None so far.

## 6 Acknowledgements

Thanks to Jonas Persson for valuable insights regarding uniqueness of the overload-set-types.

Thanks to Joe Gottman for feedback on P3183R0 which spurred adding constructor and destructor support here.

Thanks to my employer, ContextVision AB for sponsoring my attendance at C++ standardization meetings.

## 7 References

- [P2825R2] Gašper Ažman. 2024-04-16. Overload Resolution hook: declcall(unevaluated-postfix-expression).  
<https://wg21.link/p2825r2>
- [P3183R0] Bengt Gustafsson. 2024-04-15. Contract testing support.  
<https://wg21.link/p3183r0>