Overload Set Types

Document #: P3312R0
Date: 2024-05-22
Project: Programming Language C++
Audience: EWGI
Reply-to: Bengt Gustafsson
<bengt.gustafsson@beamways.com>

Contents

1 Abstract

This proposal defines a type for each overload set of more than one function. Such *overload-set-types* are created when a placeholder type is deduced from the name of an overloaded function. In contrast with function pointers these types have no runtime state. 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*. 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.

Each *point of instantiation* may create a new *overload-set-type* which means that as class and variable templates have only one point of instantiation only the overload set contents at that point is considered. For function templates each function invocation is a point of instantiation which may create a different instance if the contents of the overload set is different.

2 Motivating examples

Today you can't use functions that are overloaded when the function pointer type is deduced. This is troublesome 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 (but as proposed, only if they are defined as free functions)
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 had to combined 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.

```
// With 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 allows placeholder types to be deduced from names of overloaded functions. The proposal has no effect for functions that are not overloaded, these deduce the placeholder type to the function's function pointer type as today.

The *basic idea* is that all aspects of the original function overload set are preserved when applying the function call operator to an object of *overload-set-type*. The different design decisions described below mostly stem from this basic idea.

When a placeholder type is deduced from the name of an overloaded function an *overload-set-type* is synthesized. For exposition purposes this type can be viewed as having a generic call operator calling the function and a conversion operator to each function reference type. The *overload-set-type* itself is regular and all instances compare equal. All overload resolution is done at compile time, only depending on the point of deduction of 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 if applicable).

Here is an example of a conceptual *overload-set-type* 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)...);
    }

    operator float (&)(float) const { return std::sin; }
    operator double (&)(double) const { return std::sin; }
};
```

This type is so conceptual that compilers are required to elide the generic call operator and function pointer conversion operators. 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 means that the conversion operators don't actually count as user defined conversions in overload resolution.

Although each *overload-set-type* 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.

The function call operator can be applied to an object of *overload-set-type* just as if the original function was called. Overload resolutions 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.

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);

callWihtFloat(SinOverloads());

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

cc(s);

auto sptr = declcall(s(2.0f));

cc(sptr);

auto sptr2 = static_cast<float(*)(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); }
{
    F(2.3);
}

myFun(std::sin);
```

3.1 Defaulted parameters

According to the basic idea of this proposal defaulted parameters are considered in the overload resolution process. By the same basic idea defaulted parameters are *not* considered when converting an object of *overload-set-type* to a function pointer.

3.2 Function Templates

If the overload set contains function templates these are included in the *overload-set-type* and selected by overload resolution as usual. If an explicit specialization is encountered after the point of deduction of the *overload-set-type* and gets selected by the overload resolution of a subsequent application of the function call operator to an object of the *overload-set-type* the program is ill-formed.

3.3 Specifiers such as noexcept and constexpr

In keeping with the basic 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 ADL

*Overload-set-type*s for unqualified free function names 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. Deducing a placeholder type from a namespace-qualified function name results in an *overload-set-type* containing only the overloads visible in that namespace.

3.5 Member functions

The proposed feature works also when a placeholder type is deduced from an overloaded member function name prefixed with `&`. In this case the `(ref.*f)(args...)` or `(ptr->*f)(args...)` syntax must be used when calling the function for consistency with the case that the member function is not overloaded.

Overload resolution works exactly as if the overloaded member function was called directly. When an *overload-set-type* is deduced from a member function name it can always be used with an object reference or pointer, even if it contains `static` overloads and/or member functions with *explicit object reference*. This ensures that overload resolution works the same as if the member function was called directly. Note however that this differs from unoverloaded static functions and member functions with *explicit object reference* where the function pointer type is a non-member function pointer type and can only be called using function call syntax.

For overload sets that contain static member functions it is also possible to use regular function call syntax. This allows passing *overload-set-types* containing static member functions where unoverloaded static member function pointers can be passed today. Just as when using the `Class::function(args...)` syntax this fails if a non-static member function is selected by overload resolution. Note that calling using the regular function call syntax does not apply to member functions with *explicit object reference* as these can't be called like static member functions.

Converting the object of overload set type to a member function pointer type works the same as when converting a member function name directly. So 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.

```
struct B {
    virtual int f(int) { return 1; }
    static int f(float) { return 3; }
    int f(this B& self, double) { return 4; }
};


struct C : public B {
    int f(int) override { return 2; }
};

void callf(auto memp)
{
    B* bp = new C;

    (bp->*memp)(1);     // returns 2: Virtual dispatch happens.
    (bp->*memp)(1.2f);  // returns 3
    (bp->*memp)(2.3);   // returns 4

    int (B::*ip)(int) = memp;     // Set to overload for int
    int (*fp)(float) = memp;      // Set to static overload for float
    int (*dp)(B&, double) = memp; // Set to overload with explicit object reference
}

int main()
{
    callf(&B::f);
}
```

3.6 Constructors

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

Implicitly generated default, copy and move constructors are part of the *overload-set-type* and if one of these is selected the compiler has to make sure its code is generated as if the constructor was selected directly.

3.7 Destructors

Even though destructors are not overloadable *overload-set-types* can be deduced from destructor names of 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 member function pointer to a parameterless function, but which can't be converted to a "destructor pointer" as there is no such thing.

3.8 Conversion functions

In this proposal user-defined conversion functions are not included as there is no way to spell "all the conversion functions" and if a specific overload is named, for instance `&MyClass::operator int` this results in a member function pointer, so no *overload-set-type* can be deduced. A possible extension to handle this situation is described below.

3.9 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 calling operators using the `operator@(args...)` syntax. This means that in this proposal there is no way to create an overload set containing both free function and member function operators corresponding to when an operator is used via its *operator-token*. In keeping with the *basic idea* the built in operator overloads for fundamental and pointer types are included in operator overload sets when deduced from just `operator@`. For the same reason *overload-set-types* deduced from comparison operators include the synthesized operators created from opposite or argument swapped operators, as well as from the `<=>` operator.

Note that as we already have standard library types such as `std::less` for each operator the need to deduce an *overload-set-type* from an operator is less pronounced than for named functions. However, using `std::less` does not work for testing contract conditions of overloaded operators as the function provided to the `check_preconditions` call must be the function with the pre conditions, not a wrapper function or type.

An extension that enables creating *overload-set-types* containing both free function and member function operator overloads is described below.

There is a rather theoretical problem with namespace scope operators: If there is exactly one user defined operator @ on the namespace level visible when an *overload-set-type* is deduced it today deduces successfully to a member function pointer type. Thus when used it only has this one overload to select from, not the operators for built in types that were intended here. This proposal favors backwards compatibility and thus the operator name still deduces to a function pointer in this case. In reality there are usually a plethora of overloads visible in any namespace, so such deduction rarely works today.

3.9.1 Allowing std::invoke with all *overload-set-types*

`std::invoke` and similar functions in the standard library must 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 this should just work. As constraints are relatively new it is however 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 will retain the same signature.

3.10 Rules for template instantiations

There are two conflicting requirements on the identity of *overload-set-types*. For understandability class/variable templates should work differently from function templates.

3.10.1 Class and variable templates

For class and variable templates we only want one instance for each fully qualified function name. This is consistent with the fact that each class and variable template has just one point of instantiation, which is just before the first declaration that needs the instance. This ensures that static data members are instantiated only once and that virtual functions are unambiguously defined. As the fully qualified function name used to deduce the template parameter can be part of the mangled name of the template instance its static data members, vtables and virtual functions can be shared between TUs (for variable templates this ensures that there is only one exemplar of each instance).

For class and variable templates it is still the programmer that has to guarantee that the overload set contents is the same at the point of instantiation of each class template instance in all TUs linked together. Violating this is ill-formed no diagnostic required.

3.10.2 Function templates

For function templates it would however feel strange to only consider function overloads that were present at the first instantiation of the function template. Instead the user expectation is that the contents of the overload set is picked up each time a placeholder type is deduced from a function name. This is usually when the function is called, but it is also possible to supply an already deduced placeholder type or an object of such type when calling a function. This rule includes the case of member functions of class templates using template parameters of the class template that were originally deduced from a function name.

Here are some examples. Note that the contents of the overload set is always picked up when the function name is mentioned.

```
template<auto F> class Caller {
    void call() { F(1); }
    template<auto G> callG() { G(1); }
};

void f(const char*);
void f(float);

auto FF = f;

Caller<f> call_f;   // #1

callf.callG<f>();   // Calls f(float)

void f(int);

callf.call();       // #2: Calls f(float)

callf.callG<f>();   // #3: Calls f(int).
callf.callG<FF>();  // #4: Calls f(float) as f(int) was not present at the point of deduction of FF.
```

Here the `Caller<f>` class template instance has only one point of instantiation at #1 so the use of F inside `Caller<f>::call` sees only the two first overloads of f, despite the fact that `Caller<f>::call` itself is not instantiated until #2. However, the call to `Caller<f>::callG<f>` at #3 has its own point of deduction from f to a placeholder type so it sees the new f(int) overload. Although the call of `Caller<f>::callG<FF>` occurs after the declaration of f(int) the point of deduction is where FF was formed, so f(int) is not called.

The current rules for function templates is that each call has its own point of instantiation but that the compiler can rely on any of these producing the same code, so it only needs to generate one function implementation for all of these points of instantiation. This rule has to be modified to achieve the desired result: Each point of instantiation for a function template must result in code that corresponds to the contents of the overload set at the point of deduction of all *overload-set-types* involved. Thus as a minimum, compilers must generate separate code when overload resolution changes depending on the contents of the overload set.

There are different ways for compilers to achieve this, with different trade offs between compiler complexity, compile time and code size. To allow as many implementation strategies as possible the guarantee that the address of equal instantiations compare equal, similarly to the rules for inline function addresses.

One simple strategy is to never reuse function template instantiations if any template argument is an *overload-set-type*. This idea is similar to viewing each deduction of a placeholder type as creating a new lambda. This works but is taxing on compiler time and object code size as many equal function implementations are likely to be generated.

A more elaborate strategy is to keep track of the overload set contents for a function and somehow compare the current overload set to previously done instantiations and reuse the existing code if possible. Linkage is internal so there is no need for elaborate name mangling schemes, but code size and compile times within each TU are still reduced substantially.

To avoid code bloat caused by equal implemenations generated in several TUs it would be possible to involve the linker and allow it to do *comdat folding* of functions marked as being instantiations of the same function template for the same function name(s). The linker can then just compare the object code for the functions to check which ones that can be merged into one.

It is also possible to actually use name mangling or some other metadata scheme to keep track of different instantiations between TUs to be able to share them without special handling by the linker. This alternative may result in very long mangled names (for instance for `std::swap`) but for functions with just a few overloads this may be feasible, so it would be possible for an ABI to define that up to N functions are handled by mangling, while object code for functions templated for larger overload sets is never shared. As modules become more prevalent metadata in the module files will have an easier time keeping track of which instances are equal.

As it is only differences in which functions that are actually selected by overload resolution for each call site inside the function template instance that affects its implementation it would also be possible to record which function was selected at each call site involving the template parameter that was deduced to an *overload-set-type*. This brings the amount of information down from a potentially vast overload set to one function signature per call site in the function template instantiation. This is usually only one but it can be a few. With this strategy it is feasible to mangle all the information required to uniquely identify the template function instance so that its implementation can be shared between TUs without risking extremely long mangled names. However, this is novel in the respect that the mangled name can't be deduced from the function declaration, only from the definition, which can affect compile times as overload resolution for all call sites using an *overload-set-type* must be done before the compiler can know that the code has already been generated.

3.10.3 Virtual member functions

Virtual methods have their point of instantiation at the point of instantiation of the containing class template. This is obvious as the class has only one point of instantiation and to create objects of the class its vtable is needed, which causes all virtual functions to be instantiated. Thus, if virtual functions use *overload-set-types* provided to the class template's template parameters the contents is defined from the point of instantiation of the class template.

Template member functions can never be virtual and thus they can be instantiated according to the function template instantiation rules described above, leading to potentially multiple instantiations differing only by *overload-set-type* contents.

3.10.4 Using class and variable templates inside function template definitions

When the template parameter type of a function template is explicitly used as a class or variable template template argument the class or variable template only has its point of instantiation if the function name of the *overload-set-type* has not been used for instantiation before. This ensures that there is only one class and variable template instance for each function name even in this situation.

```
template<auto F> struct CountCalls {
    auto operator()(auto... args) { count++; return F(args...); }
    static inline size_t count = 0;
};

int f(int) { return 1; }

int countedCall(auto F, auto... args)
{
    CountCalls<F> counting;
    return counting(args...);   // #1
}

int a = countedCall(f, 1.2);             // a is 1.

int f(double) { return 2; }

int b = countedCall(f, 1.2);             // b is not 2.

assert(CountCalls<f>::count == 2);       // Only one instance of CountCalls<f> can exist.
```

While this example serves to show that there can be only one class template instance per fully qualified function name it won't actually work as intended for the same reaons: At #1 the user defined function call operator uses F which is the template parameter of the class template and thus only has one contents regardless of the number of times CountedCall is called for a function name with different overload sets for its function. A better implementation is instead:

```
template<auto F> struct CountCalls {
    static inline size_t count = 0;
};

int f(int) { return 1; }

int countedCall(auto F, auto... args)
{
    CountCalls<F>::count++;
    return F(args...);
}

int a = countedCall(f, 1.2);             // a is 1.

int f(double) { return 2; }

int b = countedCall(f, 1.2);             // b is 2.

assert(CountCalls<f>::count == 2);       // Only one instance of CountCalls<f> can exist.
```

This works as the actual calling of f is not delegated to the class template.

3.10.5 Overload sets of one function

The code above still does not work without additional tweaking of the rules as in the first instantiation of CountedCall f is not even overloaded, so F is bound to an instance of the type `int(*)(int)` while at the second instantiation f is overloaded and thus CountedCall should be instantiated for the *oerload-set-type* of all f functions, which certainly is not a function pointer type.

Also note that it is possible to instantiate CountCalls above for different function pointer types even if the function name is overloaded, using static_cast:

```
CountCalls<static_cast<int(int)>(f)> cc1;
CountCalls<static_cast<int(double)>(f)> cc2;
```

The cc1 and cc2 objects are definitely of different types and thus have separate `count` members. Thus it is impossible to change the rules so that a unoverloaded function gets an overload set type. The only way I see that this can be done is to introduce some kind of cast or other mechanism to force an unoverloaded function to still deduce as an *overload-set-type*.

This could be implemented as a magic class template in std, why not call it `std::overload_set_type`. Mentioning an overloaded function implicitly creates an object of `overload_set_type<F>` where F is a representation of the fully qualified function name. But for an unoverloaded function an object can be created using a CTAD-enabled constructor: `std::overload_set_type(myFunction)` guarantees that an overload set type for the function name is created even if it is not overloaded at this point in the program. We can now write:

```
int countedCall(auto F, auto... args)
{
    CountCalls<std::overload_set_type(F)> counting;
    return counting(args...);
}
```

Another possibility is to add a new placeholder type which can only be deduced from function names and always deduces to an *overload-set-type* even if the function is not overloaded. This could also be a magic *concept* in `std::` to avoid name clashes but with the general drawbacks of making library names magic.

With the `std::overload_set` concept (and using a variable template) we can now write:

```
template<std::overload_set auto F> size_t callCounter = 0;

int f(int) { return 1; }

int countedCall(auto F, auto... args)
{
    callCounter<F>++;                     // #1
    return F(args...);
}

int a = countedCall(f, 1.2);              // a is 1.

int f(double) { return 2; }

int b = countedCall(f, 1.2);              // b is 2.

assert(callCounter<f> == 2);              // Only one instance of callCounter<f> can exist.
```

Now there can only be one callCounter for f even if CountedCall is called both when f is overloaded and when it is not. However, this formulation that looks like a concept may be misleadning in the way that it seems that callCounter can only be instantiated when the F is overloaded. Maybe a better name is enough to avoid such confusion.

Regardless of this detail it seems that it is less error prone to specify that even unoverloaded functions are to be treated as overload sets in the class/variable template parameter list rather than at the point of use of the class/variable template.

As this type of use is just at the fringe this proposal does not have an opinion of whether this particular problem is worth solving. This section just serves to introduce the problem and one possible solution.

3.11 Explicit instantiation and specialization

A class or variable template can be explicitly instantiated as usual even with its template parameters deduced to *overload-set-types* and as usual the contents of the overload set at the first instantiation is used in the implementation. This also explicitly instantiates the non-template functions of class templates which is not problematic as there is only one contents of any *overload-set-types* their implementations can reach.

However, a function template can not be explicitly instantiated for template arguments or parameter types including function names as this would be misleading and does not allow the function definition to be hidden from any call sites, much like an inline function. An alternative is to allow but ignore explicit instantiations. The important part is that the definition must be visible when calls are made.

Another way of thinking is that when an explicit function template instantiation is done the *intent* is to lock the overload-set-type(s) affecting the instantiation so that further calls for the same function name are forced to use the explicit function instantiation even if the contents of those overload sets has changed. This is logical if the explicit instantiation is viewed as a way to tell the compiler which *point of instantiation* to use when generating code for the function.

Which way to resolve this is still an open question.

3.12 The problematic copying

When the type of a function is deduced using decltype and this turns out to be an overload-set-type it is possible to create objects of this type. But allowing assignment between such objects may cause problems as what looks like the same type may not actually be. There are some possibilities, but let's start with an example.

```
int f(int);

using F1 = decltype(f);     // A function pointer type.

int f(double);

using F2 = decltype(f);     // An overload set type
using F2b = decltype(f);        // An overload set type

int f(const char*);

using F3 = decltype(f);     // Another overload set type.

F1 f1;
F2 f2 = f1;
F3 f3 = f2;
F2 f4 = f;
F3 f5 = f;
F2b f2b = f2;           // #1

F1 f6 = f1;             // #2
F2 f7 = f2;             // #3
```

Here we create three different types for f when it has 1, 2 and 3 overloads. We then try to initialize variables of these fixed types from variables of the other types. This can be treated in different ways. First observation is that regardless of how the value is initialized it has its own type related to the contents of the overload set of f at the point of deduction, i.e. the decltype call, so the only issue is in which cases there should be an error.

This proposal makes all these initializations illegal except the last two. At #2 we are just assigning function pointers which must continue to work. At #3 we are assigning between objects of the type of the same decltype use so it should be easy to allow.

The problematic case is #1 where where have two decltype-generated types that *happen* to have the same functions in the overload set. To allow this and make those types the same would require keeping track of the entire overload set.

As these initializations and the corresponding assignments have no effect anyway it is proposed to make it illegal to mix objects of *overload-set-type* when the type has initially been deduced separately, regardless of if the contents of the overload set has changed between these dedcuctions or not.

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, operator or destructor. 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* as you can call a static function using the `obj.static_function()` syntax although `obj` is not used. This 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* (UFCS) and does not solve the main 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 (deducing this) functions work differently from non-static member functions work differently in this respect, which this extension would unify.

```cpp
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 in the example 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 shortcoming for operators, that there is no way to create an *overload-set-type* containing both free function and member function operator overloads 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 parenetheses there 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.

As an alternative another syntax which allows for function names can be used, this is described below, as an alternate extension.

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 `&MyClass::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 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 only reason seems to be completeness.

4.1 Supporting opt-in UFCS

It can be observed that the combination of free function and member function overloads used in lookup for operator-tokens is actually the same as needed for *universal function call syntax* (UFCS) except for the actual call syntax. In an extension described above it was suggested that an *overload-set-type* containing both the free function and member function overloads of an operator could be created by deducing from the operator-token itself. As doing the same thing for a named function is what constitutes UFCS it could be better to introduce another syntax which can be used not only for *operator-tokens* but also for function names.

Candidates for such syntax are mainly *operator-tokens* which are not prefix operators already and the new *back-tick* character (‘). Furthermore it would be possible to require the token to be repeated after the name to make this construct stand out better in the program code. Possible candidates among the operator-tokens are: `/`,`|` and `%` or possibly to enclose in a `< >` pair. The latter has the problem that to enclose comparison operators, especially the spaceship operator requires extra spaces: `< <=> >` so I would not recommed this.

Note that for function names even an undefined name must be allowed as it may turn out at the point where the *overload-set-type* is used that there are member functions of this name, although no free functions are declared at the point of deduction. This should not be a problem as the leading operator indicates that any identifier follows.

With reflection comes a lot of new operator combinations which have to be considered so using the back-tick may be the best choice.

Here is an example of how an opt-in UFCS call would look:

```
template<typename C> void do_something(C& container)
{
    auto iter = `begin`(container);
    auto end = `end`(container);
    auto count = `-`(end, iter);        // Operator-token also works.
}
```

Note that the back-ticks force immediate deduction, no placeholder type needed. Instead the back-tick pair itself creates a new *overload-set-type* and an object of this type (to which the function call operator is immediately applied).

While this is not as powerful as automatic UFCS at all call sites it may be less contentious as it is opt-in. It also bypasses the contentious issue of whether free functions "win" over member functions or vice versa as it piggy backs on the current rules for operators, which means that such overload sets are ambiguous. **OOPS: This is not what we want, then begin on a vector is ambiguous if `using std::begin` is done.**

The only possibility seems to be to select that free functions are preferred (as this is the syntax employed at the call site). By instead writing `(container.*begin)()` you could indicate that you want to prefer member functions. Brittle!

4.1.1 UFCS as a library funcion

If we can deduce an *overload-set-type* from name with back ticks this could be allowed even if there is nothing in scope by that name. It would then be possible to check if there is a free function and/or member function using constraints, and by this build a pair of functions templates `ufcs_f` and `ufcs_m` where the trailing letter indicates what to prefer in case both are allowed.

```
template<typename OST, typename Obj, typename... Args> decltype(auto) ufcs_f(OST ost, Obj&&, Args&&... ar
{
    if constexpr(requires { ost(obj, args...); })   // Call free function if possible.
        return ost(std::forward<Obj>(obj), std::forward<Args>(args)...);
    else
        (obj.*ost)(std::forward<Args>(args)...);    // Otherwise call member function, or error.
}

template<typename C> algo(C& container)
{
    auto i = ufcs_f(`begin`, container);
    // Use i somehow.
}
```

It would feel better if we could skip the back ticks but that seems too error prone, error messages come somewhere in the called code whenever a variable to be deduced to an auto function parameter is to be deduced.

4.2 Supporting fundamental, pointer and array types

When placeholder types are deduced from operators it is natural that "operator overloads" for fundamental types are included, as it improves the consistency to the direct use of the operator token.

For pointers there are a few operators defined as well as the dereference operators. With this extension those operators should also be included in the overload set for the relevant operators.

For arrays it would be most likely that the pointer operators are used after *array-to-pointer* decay.

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