Unified function call syntax (UFCS)

Document Number: **P3021 R0**
Date: 2023-10-13
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: EWG

Contents

1 Overview 2
2 Motivation 4
3 Design choices and alternatives considered 6
4 Proposal 14
5 Draft wording 15
6 References 16

Abstract

Since the last UFCS proposals in 2014-2016, which nearly reached plenary consensus, we have new information.

The lack of a single call syntax that can call both member and nonmember functions causes real problems in C++:

It makes writing generic code harder, and makes libraries harder to use and compose.

It makes tools such as IDE autocomplete less useful for C++ code, because they work well for `x.f()` syntax, but not for `f(x)` syntax.

(new) It continues to lead to workarounds, including in the standard library, such as overloading `operator|` (e.g., ranges) or providing nonmember versions of a limited set of common functions that cannot be made members on built-in types such as arrays (e.g., `std::begin`).

(new) It continues to lead to repeated language evolution proposals for narrow features, such as [N1742] proposing [extension methods], and [P2011R0] and [P2672R0] currently proposing `operator|>`.

(new) These workarounds and language evolution proposals would not be needed if `x.f()` could call `f(x)`. By being inherently left-to-right workarounds and proposals, the new experience information provides further evidence of which current call syntax is important to make UFCS-capable.

An experimental implementation of this proposal is available in the [cppfront] compiler.

1 Overview

C++ needs a way to uniformly call member or non-member functions, as we already support for operators.

Like several past papers, starting with [N4165] and [N4174], this paper proposes generalizing the member call syntax `x.f(a,b)` or `x->f(a,b)` to fall back to calling a nonmember function `f(x,a,b)` or `f(*x,a,b)`.

This single proposal aims to address three major issues:

- **Enable generic code.** Today, generic code cannot invoke a function on a `T` object without knowing whether the function is a member or nonmember, and must commit to one. This is a long-standing known issue in C++. With this paper, generic code can call any function using a single notation.
- **Enable call chaining without library workarounds or a separate one-off language feature.** Today libraries such as `std::ranges` overload `operator|` to get chaining of sequences of operations. However, because the `operator|` workaround has limitations, we see current proposals like [P2672R0] for a new `operator|>` to improve that piping with language support. With this paper, neither would be needed; normal `obj.next(thing).to(call)` syntax would "just work" including for many cases that already overload `operator|` today.
- **Enable "[extension methods]" without a separate one-off language feature.** The proposed generalization enables calling nonmember functions (and function pointers, function objects, etc.) symmetrically with member functions, but without a separate and more limited "extension methods" language feature such as was proposed in [N1742]. Further, unlike "extension methods" in other languages which are a special-purpose feature that adds only the ability to add member functions to an existing class, this proposal would immediately work with calling existing library code without any change.

It also achieves three other major benefits:

- **Consistency, simplicity, and teachability.** Having a single call syntax makes the language simpler and improves teachability. The nonmember function call syntax would continue to be legal, but optional (except for nonmember calls with no arguments).
- **Improve the discoverability and usability of existing code.** The feature immediately works with calling existing C++ library code. Further, it also naturally works with calling existing C-style libraries, including but not limited to the C standard library, as-is without change. This makes it a powerful way to learn and use existing libraries.
- **Improve tool support.** This feature directly assists the creation of new tool features that make working with C++ more powerful and convenient. Even more importantly, it directly leverages existing tool features, notably code editor autocomplete features, which will become significantly more powerful.

Unlike [N4174], this paper does not propose any changes to the nonmember `f(x,y)` call syntax; see Q&A below.

1.1 Revision history

R0 (2023-10): Initial revision, largely following [N4165].

1.2 Design principles

Note These principles apply to all design efforts and aren't specific to this paper. Please steal and reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability. – For example, enable generic code to call a named function without requiring it to be provided only as a member function or only as a non-member function. Replace the need current workarounds such as invoking non-member `std::begin` and providing range/view `operator|`. Reduce the incentive for future special-purpose language evolution features like `operator|>`.
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination. – For example, allow all functions that work on a given type, including non-member non-friends (whether written by the class author themselves for better encapsulation, or by a library user), be used uniformly with objects of that type, without the need for special features like [extension methods].
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features. – For example, today we already have UFCS, but only for overloaded operators; it should be provided for all functions.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

Additional design principles include: Make important things and differences visible. Make unimportant things and differences less visible. — This proposal enables a class author, and the class' users, to use the features provided by a class without having to know which call syntax to use, as C++ has already long supported for operators.

1.3 Acknowledgments

Thanks to Gabriel Dos Reis and Bjarne Stroustrup for their comments and feedback on this topic and/or on drafts of this paper's predecessor, [N4165].

Thanks to Bjarne Stroustrup, Francis Glassborow, and Barry Revzin for their previous papers in this area, and to all those who gave feedback on those papers and this topic in WG 21.

2 Motivation

"Having two call syntaxes makes it hard to write generic code" — B. Stroustrup (N4474)

2.1 Enable generic code

It is a well-known and long-standing problem that C++ has two incompatible calling syntaxes:

- `x.f()` and `x->f()` can only be used to invoke members, such as member functions and callable data members; and
- `f(x)` can only be used to invoke nonmembers, such as free functions and callable nonmember objects.

Unfortunately, this means that calling code must know whether a function is a member function or not. In particular, this syntactic difference defeats writing generic code which must select a single syntax and therefore has no reasonable and direct way to invoke a function `f` on an object `x` without first knowing whether f is a member function or not for that type. Because there is no single syntax that can invoke both, it is difficult or impossible to write generic code that can adapt.

This problem and proposed solutions have been raised in the past. However, the proposed solutions generally have favored extending the nonmember function call syntax to be able to invoke members.

This proposal goes the other direction: Extend the *member* function call syntax to be able to invoke nonmembers. There are several reasons for this, including that member function call syntax is technically easier to extend in existing C++ without breaking language changes.

2.1.1 Note: We already have UFCS… but only for natural operator notation

Today we already have UFCS for operators:

```
auto add( auto a, auto b ) {
    return a + b;                // generic call: UFCS calls member or non-member!
}
```

Except when we don't, if we call the operator like a function:

```
auto add( auto a, auto b ) {
    return operator+(a,b);       // non-generic: only calls non-member operator+
}

auto add( auto a, auto b ) {
    return a.operator+(b);       // non-generic: only calls member operator+
}
```

For completeness, yes we can write the following in C++20, but it's generally impractical for even a single function, and infeasible for multiple functions because it's combinatorial:

```
auto add( auto a, auto b ) {
    if constexpr( requires{ operator+(a,b); } ) {          // beware: here be
        return operator+(a,b);                             //  combinatorial dragons
    }
    else if constexpr( requires{ a.operator+(b); } ) {
        return a.operator+(b);
    }
    else {
        //  ?
    }
}
```

2.2 Enable encapsulated code: Nonmember nonfriends increase encapsulation, lower coupling

"Functions want to be free." Scott Meyers and others have observed and taught that it is good to prefer nonmember nonfriend functions as these naturally increase encapsulation. However, the current rules disadvantage nonmember functions because they are visibly different to callers (have a different call syntax), and are less discoverable. This proposal would remove the major reasons to avoid following this good design guidance by making nonmember functions as easy to use as member functions, particularly for discoverability and tool support (see next section).

3 Design choices and alternatives considered

3.1 Why the member call syntax should be UFCS

A class author controls its primary interface; therefore, regardless which UFCS syntax(es) we might adopt, it is desirable to have one where member functions are preferred, or at least equal, to nonmember functions for name lookup. It is much simpler to get that result by extending the member call syntax, which already looks up members first (and then stops today), than by extending the nonmember call syntax which creates a tension between preferring members and backward source compatibility with existing code.

Making the member function syntax be UFCS also follows the existing practice of the workarounds, as shown later in this section.

3.1.1 It's friendlier to editors and tools

In any language, the member function call syntax is preferable because it puts the argument first rather than the function first. When you start with the function name, the list of "objects/expressions you can pass to that function" can be undecidable and possibly infinite. When you start with an object or expression to be manipulated up front, it's easy to narrow down a useful list of "things you can do to that object." This aids programmer productivity, discoverability of APIs (what can I do with this object), and tool support.

This is why "start with the argument" is friendlier to editors than "start with the function." Consider the following examples, where ▒ is the current cursor position: Which is easier to provide autocomplete for, A or B? Let's start with A:

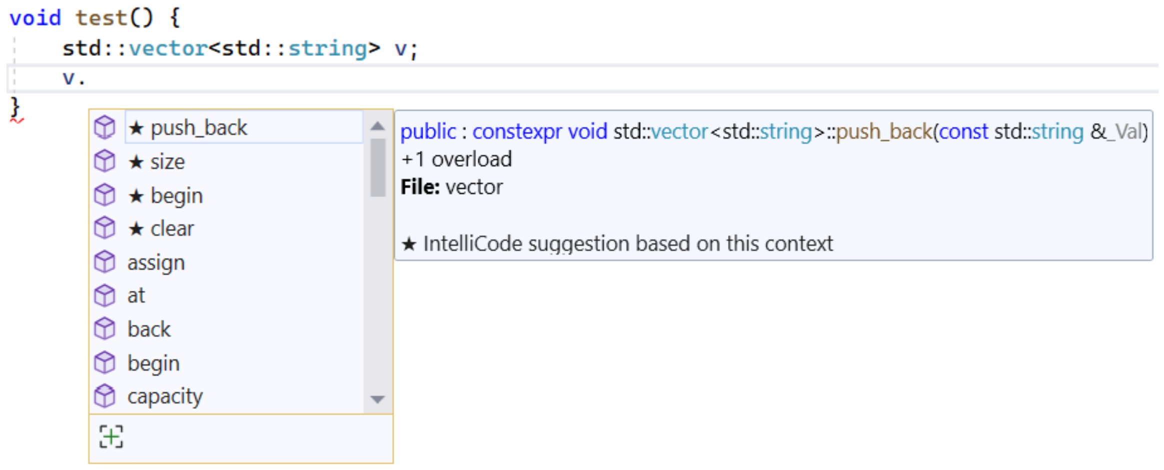
```
A:          f(▒
```

Case A is fundamentally hostile to providing useful autocomplete suggestions. I speculate without proof that it might be halting-hard, because the possible valid set of expressions that can be passed as an argument is often infinite. For example, to decide a candidate list of "what could the programmer intend to pass to `f`?", perhaps we might try to enumerate all possible variables in scope to see which could possibly be validly passed as a first parameter to some possibly-overloaded `f` in some scope—including via ADL on each such candidate object, and including after conversions, etc. That heuristic alone would be very difficult in itself, and it would be completely insufficient in real code where often the programmer actually intends to pass not just a simple variable `x` to call `f(x)`, but rather to pass an expression such as to `f(x + y * z)`… how would autocomplete reasonably find such a valid suggestion, and come up with a usefully short list of possible intended parameters?

Now consider case B:

```
B:          x.▒
```

Case B is friendly to providing useful autocomplete suggestions, and our editors do it all the time… they not only know what the valid callable functions are, but they commonly even use AI-based heuristics to recommend the most common ones in a particular context (see the first four starred examples here):

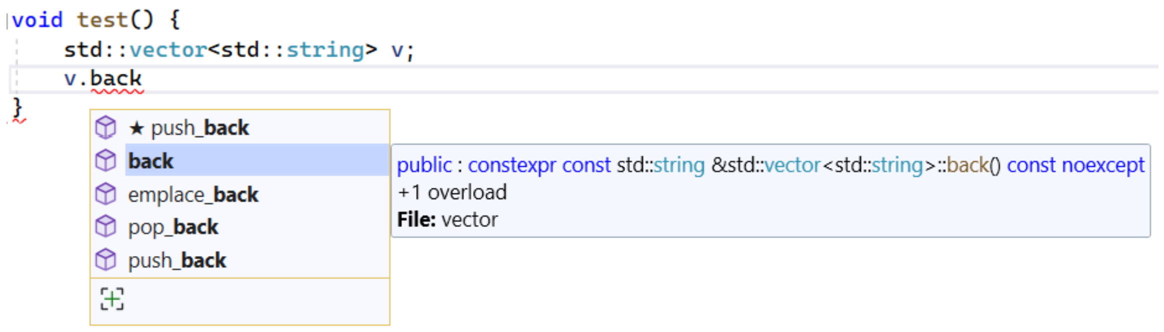


```
void test() {
    std::vector<std::string> v;
    v.
}
```

public : constexpr void std::vector<std::string>::push_back(const std::string &_Val)
+1 overload
File: vector

★ IntelliCode suggestion based on this context

including to let you automatically filter names by substrings, such as typing “`.back`” to start seeing a list of all the things you can do with “back”:



```
void test() {
    std::vector<std::string> v;
    v.back
}
```

public : constexpr const std::string &std::vector<std::string>::back() const noexcept
+1 overload
File: vector

Note AI-based recommended functions may vary for the same type in different contexts. Here, immediately after declaring a `vector` object, an AI copilot would know that the most common next function to call that contains the substring “back” would be `push_back` — not `back` itself, which would be an error on a empty container. So even though the programmer wrote “back” the IDE should rightly display `push_back` as the top suggestion.

This means we would immediately get another carrot to migrate code from C to C++, namely by making many C libraries (even large parts of the C standard library) easier to use, and with much better autocomplete and other tool support by doing nothing more than adopting the new C++ member call syntax. For example:

```
// C code
void f( FILE* file ) {
    fseek(file, 9, SEEK_SET);    // arg is buried; no autocomplete help

// proposed new C++ code
void f( FILE* file ) {
    file.fseek(9, SEEK_SET);     // nice autocomplete after “.”
```

Note See this 1-minute video clip for a demonstration of how natural UFCS is for this example.

Notice how well the C standard library works “out of the box” with UFCS. Larger C libraries would benefit even more by making their code more navigable and discoverable (and put them in a position where it’s easier to start adopting even more C++).

Here is another C-and-C++ example: It feels strange to some of us that we have taught a generation of C++ developers to invoke `c.begin()` and `c.end()` to get iterators, yet in C++11 to instead prefer `std::begin(c)` and `std::end(c)` because the latter is more extensible to arrays and adapted non-STL containers. Under this proposal, we re-enable existing long-standing guidance to use the member syntax `c.begin()` and `c.end()`, which among other things not only would now work correctly for C arrays and adapted non-STL containers, but would enable better editor and tool support for C arrays such as offering autocomplete suggestions that include "`.begin()`" on a *C array* name.

3.1.2 It's friendlier to programmers: Discoverability

The fact that the member call syntax is friendlier to code editors is only a special case of a broader benefit: it's friendlier to tools in general, and to programmers to discover what they can do next in their code.

Autocomplete support like the above makes libraries and APIs immensely more discoverable for humans.

This illustrates why the member call syntax is generally friendlier to programmers. I believe the "start with the object" point of view is a much more usable programming model because it directly enables answering the constant question of how to discover "what can I do next?". What you can do next (functions) depends on what you have now to do it with (your objects and variables).

To use the example of C arrays again, offering "`.begin()`" as an autocomplete suggestion when the programmer writes the name *of a C array variable* would make the C++11 `std::begin` feature immediately more discoverable.

3.1.3 It more directly corresponds to what programs do: State transition

Programs exist to express ways to transition the system through successive valid states. And that's the difference: Functions and methods are merely names for the *transitions*. Objects are names for parts of the current *state*. And the state is far more important than the transitions: the result of a program is its final *state*; the thing you serialize and deserialize (to save to disk and restore, to send on a wire and receive) is its *state* (or part thereof). If we recognize that programs are all about expressing how to get from this state to the next state, it becomes natural to realize that code wants to be written the same way, and that the fundamental question on every new line of code is: "From my state I have now (my objects and variables), what can I do to get to the next state (functions and methods)?" I start with the state I have now (my objects and variables), and *then* I do something to that (function or method).[1]

3.1.4 The existing-practice workarounds already simulate extending the member call syntax, not the nonmember syntax

I argue that the member syntax is the "visually correct" left-to-right order because it follows the order in which expressions are evaluated and actually executed:

```
first().second().third().fourth();      // in evaluation and execution order

fourth(third(second(first())));         // exactly backwards / inside-out
```

But don't take my word for it, look at existing practice… the most common workaround is to overload `operator|` to get left-to-right chaining syntax. Consider this code from [cppreference.com transform_view]:

[1] This is true even in pure functional languages, where the new state is expressed in new objects or values instead of mutated objects or values. The program state as a whole has indeed been changed, and it is represented in its values.

```
//  Today’s existing practice is left-to-right, like x.f(y):
//    Overload operator| to simulate 'x.f(y)' UFCS as 'x | f(y)'

std::ranges::for_each( in | std::views::transform(rot13), show );
std::ranges::for_each( in | transform(rot13), show );      // with ‘using std::views’
```

Note two things:

- The | operator is left-to-right, so we read the steps in the same order in which they occur.
- With UFCS, the same code works by simply replacing | with ., eliminating the need for the additional helper operator| functions which could be deprecated and removed to simplify the standard library:

```
std::ranges::for_each( in.std::views::transform(rot13), show );
std::ranges::for_each( in.transform(rot13), show );        // with ‘using std::views’
```

3.1.5 The currently-proposed special-purpose language extensions also simulate extending the member call syntax, not the nonmember syntax

This paper was motivated by [cppfront #741] which includes this extended example from [Hoekstra] (see last 6 minutes of the talk) that is discussed in [P2672R0]:

```
//  (again) Today’s existing practice is left-to-right, like x.f(y)

auto filter_out_html_tags(std::string_view sv) {
    auto angle_bracket_mask =
        sv | rv::transform([](auto e) { return e == '<' or e == '>'; });
    return rv::zip(rv::zip_with(std::logical_or{},
            angle_bracket_mask,
            angle_bracket_mask | rv::partial_sum(std::not_equal_to{})), sv)
        | rv::filter([](auto t) { return not std::get<0>(t); })
        | rv::transform([](auto t) { return std::get<1>(t); })
        | ranges::to<std::string>;
}
```

Notice above that the operator| workaround helps, but not all the uses in this code fit it naturally. This has led to proposal [P2672R0] to add operator|>, so this code could be written more simply like this when the left-hand object is not in the first position and/or is mentioned multiple times:

```
//  Today’s evolution proposals are still left-to-right, like x.f(y):
//    Proposing to overload operator|> to simulate 'x.f(y)' UFCS as 'x |> f(y)'

auto filter_out_html_tags(std::string_view sv) {
    return sv
        |> transform($, [](auto e) { return e == '<' or e == '>'; })
        |> zip_transform(std::logical_or{}, $,
                      scan_left($, true, std::not_equal_to{}))
        |> zip($, sv)
        |> filter($, [](auto t) { return not std::get<0>(t); })
        |> values($)
        |> ranges::to<std::string>($);
}
```

I recently encountered this code example when it was reported in [cppfront #741] as a suggestion that we add operator|> to my experimental alternate "syntax 2" for C++ (Cpp2 for short). Because Cpp2 already had UFCS, my initial reaction was to ask whether UFCS would just work. After a little discussion and fixing a compiler bug we found the following compiled and ran just fine… translating it to today's syntax with this proposal:

```
//  General-purpose helper to make it easier to pass a non-first
//  parameter and/or pass the same parameter multiple times
auto&& call(auto&& o, auto&& f)
    { return std::forward<decltype(f)>(f) ( std::forward<decltype(o)>(o) ); }

//  The following is working code using a UFCS implementation in cppfront
//  (changed by hand to today's syntax + this proposal)
//      - no need for an operator|> new language extension
//      - no need for the operator| workaround (we just ignore it)

auto filter_out_html_tags(std::string_view sv) {
    return sv
        .transform( [](auto e){ return e == '<' || e == '>'; } )
        .call( [](auto const& x){ return zip_transform(std::logical_or(), x,
                                     scan_left(x, true, std::not_equal_to())); } )
        .zip(sv)
        .filter( [](auto t){ return !t.get<0>(); } )
        .values()
        .to<std::string>();

}
```

To me, this code even better than with the proposed special-purpose language feature — Stroustrup's classic 'simplification through generalization.' The code is natural and readable with just plain old familiar x.f() function syntax, it naturally obsoletes the operator| library workarounds which can be deprecated, and avoids inventing a new operator|> special-purpose language feature to implement and learn to use. Much existing ranges code just works naturally out of the box with . now in cppfront.

Note This example would be even more readable if we also had a terse lambda syntax that defaulted away unused parts of the syntax, whereby `[](auto const& x){ return expression; }` could be spelled `[](x) expression;` (as I do in Cpp2, where it can be spelled `:(x) expression;`):

```
//  Working code using a UFCS implementation in cppfront Just Works with '.'
//  (changed by hand to today's syntax + this proposal + a terse lambda syntax)

auto filter_out_html_tags(std::string_view sv) {
    return sv
        .transform( [](e) e == '<' || e == '>'; )
        .call( [](x) zip_transform(std::logical_or(), x,
                                   scan_left(x, true, std::not_equal_to())); )
        .zip(sv)
        .filter( [](t) !t.get<0>(); )
        .values()
        .to<std::string>();
}
```

3.2 Q&A

3.2.1 Q: Is this fully backward-compatible without breaking existing code? A: Yes.

There is no breaking change.

3.2.2 Q: Wait, isn't `std::begin`/`end`/`size` a counterexample, evidence favoring *non-member* function call syntax being UFCS? A: No.

No, it's evidence favoring member function call syntax being UFCS, in order to call the `std::` non-members as "extension methods" for C arrays without needing a special-purpose extension methods language feature.

First, the only reason those are nonmember functions are that we can't write member functions on C arrays. In fact these are a great motivating example of UFCS subsuming extension-methods, because the nonmember `begin`/`end`/`size` are totally fine *as extension methods to use for C arrays*, rather than as also becoming the primary call syntax.

Second, even after their presence in the standard for a decade, the nonmembers are not widely used, which is evidence that they are less adoptable and will never be widely used. It is still common for programmers to call *member* `.begin`/`.end`/`.size` instead, likely not realizing they are making their code a little less generic because it can no longer work with C arrays. Let's ask codesearch.isocpp.org:

```
std::begin     6,257 matches found

.begin       607,443 matches found
```

With this proposal, all the above 600,000 uses of `.begin` that are general dependent calls that work on any STL container would automatically become fully generic and naturally start working just fine for C arrays too. It's not often that we have the opportunity to silently improve and fix existing code, and this UFCS proposal would achieve that.

3.2.3 Q: Why not the reverse, let nonmember call syntax find members? A: It's possible to do that too, but would find members as a fallback.

This paper **does not** propose to also extend the nonmember function call syntax to find members. Any such future proposal would have to decide what should happen when both a member and a nonmember would be viable. There are three main options: (a) members are preferred, (b) members and nonmembers are equal (e.g., they overload), and (c) nonmembers are preferred. Only (c) is feasible, because (a) and (b) would break large amounts of existing code. For example:

```
struct X { void f(); }; // 1

void f( X ); // 2

f(x); // ok, calls 2 today
      // under choice (a): ok, calls 1 (breaks code)
      // under choice (b): error, ambiguous (breaks code)
      // under choice (c): ok, still calls 2
```

3.2.4 Q: What about proposals to make `x.f()` and `f(x)` be identical? A: It's neither possible nor desirable.

Some have suggested *extending* `x.f()` similarly to this proposal, and also *changing* `f(x)` to mean the same thing.

First, doing the latter is not possible for the reasons given in the previous section. Whereas *extending* the meaning of `x.f()` as in this proposal is not a breaking change because it preserves the meaning of code that uses that syntax today, and by adding a fallback gives meaning to cases that would be an error today, *changing* the meaning of `f(x)` would change the meaning of existing code. For example, changing the meaning of `f(x)` to prefer nonmembers would break every use of a nonmember call `draw(my_cowboy, graphics_device)` that would now be hidden by `cowboy::draw(from_holster_number)`.

Second, having `x.f()` and `f(x)` as two syntaxes for the same thing is not desirable, even if it were not a breaking change. Having two redundant syntaxes in the language for the same thing has two drawbacks: 1. It confuses programmers because they will want to know when to spell it one way vs the other. 2. It "burns a syntax" for no benefit by occupying a syntactic place in the language that then can never be used to express a distinct meaning in the future (and in this case is actually already used for a different meaning). Furthermore, in this particular case, it seems undesirable to have one syntax that prefers the member, and preclude ever having another natural syntax that prefers the nonmember, which would be consistent with C++'s flexibility of being able to express what you want.

3.2.5 Q: What about the pointer-to-member call syntax, `x.*f` or `x->*f`? A: No change.

No change is proposed to that syntax. It is specific to members and rarely used, so it need not be generalized.

3.2.6 Q: What about allowing "`this`" in non-first positions? A: Sure, that could be a future further-generalizing proposal.

This paper **does not** propose to also allow `this` in non-first positions. However, a future proposal could propose further allowing the expression that appears to the left of the member selection operator to invoke member functions where that expression is not the first parameter.

This would require design work. One possible meaning would be to allow `x.f(a,b)` to invoke functions that could be invoked by moving `x` into any position in the parameter list, i.e., functions that could be invoked by `y.f(x,a,b)`, `y.f(a,x,b)`, or `y.f(a,b,x)`.

This would have the side effect of making C libraries (including but not limited to the C standard library) slightly even more usable, because the C standard library's "explicit `this`" style doesn't always put the "`this`" in the first parameter location. For example:

```
// C code
FILE* file = fopen( “a.txt”, “wb” );
if (file) {
    fputs(“Hello world”, file);
    fseek(file, 9, SEEK_SET);
    fclose(file);
}
```

```
// proposed new C++ code
FILE* file = fopen( "a.txt", "wb" );
if (file) {
    file.fputs("Hello world");
    file.fseek(9, SEEK_SET);
    file.fclose();
}
```

Merely writing `file.` immediately allows code editors to present a dropdown of exactly the C standard library functions that take a `FILE*` in some position. This makes even 30-year-old C libraries "better in C++" and offers a strong reason to switch to C++ for just this one productivity feature, and once a project has adopted a C++ compiler it can easily start to use more C++ features.

This also has important future-proofing advantages, where potential future C++ language features such as multimethods would, if adopted, further encourage writing the `this` in parameter positions other than the first parameter, and even on multiple parameters.

3.2.7 Q: But isn't the nonmember syntax `f(x,y)` more general than the member syntax `x.f(y)`, for example if C++ ever gets multimethods? A: No.

Some argue that the dot notation is inherently tied to *single* dynamic dispatch, and that if C++ were to eventually allow *multiple* dynamic dispatch then the symmetrical `f(x,y)` notation is more appealing. For example, `intersect(s1,s2)` may appear to be more general than `s1.intersect(s2)`.

However, neither syntax is more general. Even if we support multimethods. In this proposal, if the previous section were also adopted to allow `this` in the non-first position, it would just mean that if `Shape` has no member function `intersect`, then `s1.intersect(s2)` and `s2.intersect(s1)` mean the same thing – whether or not the nonmember `intersect` is allowed to be a multimethod in a future version of C++! Furthermore, not only is the member syntax not inferior in generality, but it continues to enjoy its advantage in discoverability – only with the member syntax can the IDE offer a list of available multimethods, which means that the member syntax is superior for discovering also potential future features C++ does not yet have.

4 Proposal

For simplicity this section discusses only member selection using `x.y` (dot), but everything applies also equally to member selection using `x->y`.

4.1 Extend the member call syntax to fall back to find nonmembers

I propose allowing `x.f(a,b,c)` to find nonmembers as follows:

- First perform lookup for member functions, as today. This preserves backward compatibility with existing code, and it also makes design sense to prefer member functions.
- If no accessible member function is found, the perform name lookup as if `f(x,a,b,c)` had been invoked, including being able to find functions via ADL and invoke function objects. Apart from SFINAE tricks, this should be a pure extension that gives meaning to previously ill-formed programs.

Now we can write generic code that works for both members and nonmembers:

```
struct X { };
void f( X );

struct Y {
    f();
};

template<class T>
void generic( T t ) {
    t.f(); // if T is X, calls f(X); if T is Y, calls Y::f()
}
```

5 Draft wording

In [expr.ref], add a new paragraph [9]:

> If `E2` is not found as a postfix function call expression using the dot operator, and `E1.E2` is followed by a function argument list `(args)`, treat the postfix expression `E1.E2(args)` as a postfix function call expression `E2(E1,args)`.
>
> [[Example:

```
namespace N {
    struct Y { void k(); };
    void h(Y);
};

struct X { void k(); };

void h(X);

struct S {
    void f(X x) {
        x.g();   // invoke S::g(X): (*this).g(x)
        x.h();   // invoke ::h(X)
        x.k();   // invoke X::k(): x.k()
        x.::k(); // error: there is no global k
    }

    void g(X);

    void f2(N::Y y) {
        y.g(); // invoke S::g(N::Y): (*this).g(y)
        y.h(); // invoke N::h(N::Y); N.h() found by ADL
        y.k(); // invoke N::Y::k(): y.k()
        y.N::k(); // error: there is no k in namespace N
    }

    void g(N::y);

};
```

> --end example]]

6 References

[N1742] F. Glassborow. “Auxiliary class interfaces” (WG 21 paper, November 2004)

[N4165] H. Sutter “Unified call syntax” (WG 21 paper, October 2014)

[N4174] B. Stroustrup. “Call syntax: x.f(y) vs. f(x,y)” (WG 21 paper, October 2014)

[N4474] B. Stroustrup and H. Sutter. “Unified call syntax: x.f(y) and f(x,y)” (WG 21 paper, April 2015)

[P0131R0] B. Stroustrup. “Unified call syntax concerns” (WG 21 paper, September 2015)

[P0251R0] B. Stroustrup and H. Sutter. “Unified call syntax wording” (WG 21 paper, February 2016)

[P2011R0] B. Revzin. “A pipeline-rewrite operator” (WG 21 paper, January 2020)

[P2672R0] B. Revzin. “Exploring the design space for a pipeline operator” (WG 21 paper, October 2022)

[cppfront] Cppfront compiler (GitHub, 2022-present)

[cppfront #741] Neil Henderson et al. “[SUGGESTION] Implement the pipeline operator from P2011 & P2672” (October 2023)

[cppreference.com transform_view] `transform_view` documentation (cppreference.com, retrieved October 2023)

[extension methods] “Extension methods” (Wikipedia, retried October 2023)

[Hoekstra] C. Hoekstra. “New algorithms in C++23” (CppNorth 2023)