

Unified Call Syntax

Herb Sutter

Document #: N4165
Date: 2014-10-04
Reply to: Herb Sutter
(hsutter@microsoft.com)

Contents

Overview	2
Motivation.....	2
1. Generic code	2
2. Nonmember nonfriends increase encapsulation.....	3
3. Discoverability and tool support: “Start with the argument” puts intent up front.....	3
It’s friendlier to editors and tools	3
It’s friendlier to programmers: Discoverability.....	4
It more directly corresponds to what programs do: State transition.....	5
Proposal	5
Extend the member call syntax to fall back to find nonmembers	5
Additionally allow “this” in not only the first parameter location.....	6
Q&A.....	6
Q: Is this fully backward-compatible without breaking existing code? A: Yes.	6
Q: Why not do the reverse, extend the nonmember call syntax to find members? A: It’s possible to do that too, but if so should find members as a fallback.....	7
Q: What about proposals to make $x.f()$ and $f(x)$ be identical? A: It’s neither possible nor desirable.	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.....	8
Q: What about the pointer-to-member call syntax, $x.*f$ or $x->*f$? A: No change.....	8
Acknowledgments.....	8
References	8

Overview

Like [1], this paper proposes generalizing the member call syntax `x.f(a,b)` or `x->f(a,b)` to allow calling nonmembers. This single proposal aims to address two major issues:

- *Enable more-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 long-standing known issue in C++.
- *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. 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. (See also following points.)

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 redundant (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 C++ 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 [1], this paper does not propose any changes to the nonmember `f(x,y)` call syntax; see Q&A below.

Motivation

1. 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. Nonmember nonfriends increase encapsulation


“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. Discoverability and tool support: “Start with the argument” puts intent up front

This section may sound philosophical at first, but it isn’t about philosophy. It’s all about practicality.

More importantly, I am becoming convinced that member function call syntax is actually preferable in any language, 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.

It’s friendlier to editors and tools

First, “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:

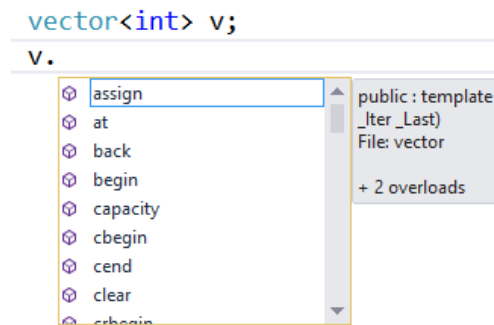
A: `f(`

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 `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:



Note: 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 "->"
```

This example uses the C standard library just to show how well it works “out of the box” with C code, but 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 `begin(c)` and `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.

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 this C++11 feature immediately more discoverable.

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

Proposal

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

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 reperform name look 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()
}
```

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.

Note that regardless which unified call syntax we might adopt, member functions should be 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), than by extending the nonmember call syntax which creates a tension between preferring members and backward source compatibility with existing code.

Additionally allow “this” in not only the first parameter location

We can further allow the expression that appears to the left of the member selection operator to invoke member functions where that expression is not the first parameter.

The meaning is that we 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 has the side effect of making C libraries (including but not limited to the C standard library) even more usable, because its “explicit this” style doesn’t always put the “this” in the first parameter location:

```
// 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 “best 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.

Q&A

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

There is no breaking change.

Q: Why not do the reverse, extend the nonmember call syntax to find members? A: It's possible to do that too, but if so should find members as a fallback.

If we tried to extend the nonmember function call syntax to find members, we 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.

The first two choices are infeasible because they 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
```

Choice (c) is possible:

```
void f( X ); // 2

...
f(x); // ok, calls 2 today

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

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
```

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 as shown in 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.

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 my proposal, 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.

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.

Acknowledgments

Thanks to Gabriel Dos Reis and Bjarne Stroustrup for their comments and feedback on this topic and/or on drafts of this paper.

References

- [1] B. Stroustrup. N4174: Call syntax: `x.f(y)` vs. `f(x,y)` (October 2014).
- [2] F. Glassborow. [N1742: Auxiliary class interfaces](#) (November 2004).