Unified Call Syntax: x.f(y) and f(x,y)

Bjarne Stroustrup [bs@ms.com]

Herb Sutter [hsutter@microsoft.com]

Abstract

In Urbana, EWG discussed two proposals for generalizing/unifying the two function call notations **x.f(y)** and **f(x,y)**. The papers are

- N4165: Herb Sutter: *Unified Call Syntax*
- N4174: Bjarne Stroustrup: *Call syntax: x.f(y) vs. f(x,y)*

Our reading of the discussion is that the papers were seen as promising, but that there were concerns about compatibility in the case of Bjarne's paper. There were also a few questions about exactly where this call unification would be useful.

This note discusses the motivation, considers a few alternatives, and presents a revised proposal. Briefly, **x.f(y)** will first look for members of **x**'s class (as ever) and if no valid call is found will look for functions as if **f(x,y)** had been written. Conversely, f**(x,y)** fist looks for functions (as ever) and if no valid call is found looks for functions as if **x.f(y)** had been written. We handle **p->f(y)** and **f(p,y)** similarly. This proposal is backwards compatible (modulo SFINAE trickery, of course).

Motivation

Having two call syntaxes makes it hard to write generic code: which syntax can we assume from a template argument type? When we start writing concepts, we will either have to support both syntaxes (verbose, potentially doubling the size of concepts) or make assumptions about how objects of certain binds of types are to be invoked (and we may be wrong close to 50% of the time).

Today, we already have the problem that many standard-library types are supported by two functions, such as, **begin(x)** and **x.begin()**, and **swap(x,y)** and **x.swap(y)**. This is an increasing problem.

The problem was solved for operators. An expression **a+b** may be resolved by a free-standing function **operator(X,X)** or a member function **X::operator(X)**.

The problem was solved for range-**for** by allowing both **begin(X)** and **X::begin()** to be found.

The existence of two special-case solutions and a lot of duplicated functions indicate a general need.

Each of the two notations has advantages (e.g., open overload sets for non-members and member access for members) but to a user the need to know which syntax is provided by a library is a bother.

Thus implementation concerns can determine the user interface. We consider that needlessly constraining.

Concepts make this (well-known and long-standing) problem more acute by formalizing the requirements of generic components. Consider writing a concept for a container:

```
template<typename C> concept bool Container =
        requires(C a, C b, int i)  {
                { a[i] } -> typename C::value_type&;      // [] must be member
                                                          // (says the language
                { a==b} -> bool;            // operator ==() can be a member or a
                                            // nonmember (says the language)
                { begin(a) } -> typename C::Iterator;     // begin() can be non-member
                { a.begin() } -> typename C::Iterator;    // begin()  can be member
                { a.size() } -> int;        // size() must be member
                // …
        };
```

Consider **c.size()**. It must be a member function. This **Container** concept does not support **size(c)**. Someone might define a **size(X)** for a type **X**, but callers of a template using a **Container** cannot rely on **size(x)** and neither can the implementers of a template function using **Container**s. This forces the use of **x.size()** even if **size(x)** is defined for **X**. If **X** is a type we cannot change, this is a serious problem because adding a member is an intrusive operation. This is the observation that has pushed STL-inspired libraries towards the use of free-standing functions.

Consider the two requirements for **begin()**. We know that if we include **<iterator>** we can use **begin(c)** even for types that has only **c.begin()**, but the writer of **Container** cannot assume **<iterator>** for all uses, so to allow **begin(c)** for any type, it must require it for all types.

What I wrote for **size()** constrains all users to use **c.size()** and all **Container** classes to implement a member **size()**. That is constraining on both users and implementers, and potentially intrusive on older container implementations. What I wrote for **begin()** gives users the freedom of choice between **begin(c)** and **c.begin()** at the cost of forcing every container writer the ensure that both are defined (somehow). As usual, a user of **begin()** has to make sure that the appropriate declarations are in scope. This could be difficult for pre-STL-style containers.

For every function mentioned in a concept, we must either define it twice (as a member and a non-member) or make a (semi-arbitrary) decision on which syntax is to be used.

Note that this is ***not*** a problem with concepts. It has been a problem with templates since day #1. It is a problem of what users of a template can assume about the template definition and what a template implementer can require from users. We have always had concepts, but moving them from comments to a proper interface specification brings the problem out in the open.

Ideals

We consider it ideal if both notations would have the same meaning. Ideally, that meaning would be the same as what we have for operators and for range-**for**. That way, there would be only one rule to learn and only one possible meaning of a concept.

Unfortunately, this is not possible:

- Range-**for** uses the prefer **x.begin()** over **begin(x)** strategy (6.5.4 The range-based for statement [stmt.ranged])
- Operator+ uses overload resolution to decide between **operator+(x,y)** and **x.operator+(y)** (13.3.1.2 Operators in expressions [over.match.oper])

Similarly, having only one meaning for both call notations would make it impossible to achieve 100% backwards compatibility: Currently

- **f(x,y)** will not find a **X::f(Y)**
- **x.f(Y)** will not find a **f(X,Y)**, including lambdas

If we grant precedence to one resolution (e.g., prefer member over free-standing function, as done for range-**for**) or if we select based on overload resolution, we can easily construct examples that change meaning compared to status quo. For example:

```
struct S {
        void f(int);
};

void f(const S&, double);

void f(S& s)
{
        s.f(2.0);       // overload resolution would call free-standing f()
                        // status quo will call the member
        f(s,2);         // member-preference would call member
                        // status quo will call the freestanding f()
}
```

Herb's proposal addresses this problem by generalizing only **x.f(y)** to call **f(x,y)** if there is no match for **x.f(y)**, but not generalizing **f(x,y)** to find a member function callable as **x.f(y)**. This is "too object-oriented" for some people (including Bjarne). Many don't want to have to write **x.sqrt()** rather than **sqrt(x)**, e.g., **2.sqrt()**, and that's what we'd all have to do in all generic code and all concepts to get the full generality that is the primary aim of these proposals. Please don't underestimate this problem:

- Bjarne received a fair amount of outraged mail from people who thought he had proposed that (and only that).
- Giving preference to **x.f(y)** encourages making functions members, thus increasing coupling.

- A lambda passed as a template argument must be called using the **f(x,y)** notation.

Not all opposition to **x.f(y)** is emotional and aesthetic.

We should "generalize" both **f(x,y)** and **x.f(y)** and that the (possibly unachievable) ideal is that they should have the same meaning.

Alternatives

We see six alternatives:

1. Do nothing. That implies the complication of concepts and the continuing duplication of library functions.
2. Generalize only **x.f(y)** (to call **f(x,y)** if there is no valid resolution for **x.f(y)** as a member function). Herb's original proposal. That would leave us writing **x.sqrt()** for generality.
3. Generalize only **f(x,y)** (to call **x.f(y)** if there is no valid resolution for **f(x,y)** as a free-standing function). Nobody is proposing that; we mention it only for completeness. This would leave us writing **draw(obj)** for generality.
4. Prefer member functions over free-standing functions for **x.f(y)** and prefer free-standing functions over member functions for **f(x,y)**. This is 100% compatible with current rules, but leaves **operator+** as a special case. Also, from the point of view of a concept, **x.f(y)** and **f(x,y)** are not completely equivalent. Gaby mentioned this alternative in EWG in Urbana.
5. Use overload resolution to choose the best match from the set of free-standing and member functions. This is not 100% compatible with current rule, and leaves range-**for** as a special case (but we could possibly fix that).
6. Prefer a member function for both **x.f(y)** and **f(x,y)** notations. Bjarne's original proposal. This is the least compatible variant.

[6] (two notations and one semantics) is the simplest and solves all problems with concept writing. We consider it reasonable to prefer member functions because they are the ones supplied by the designer of a type. It solves the concept writer's problem. It is not 100% compatible.

[5] (overload resolution) is almost 100% compatible, and arguably the resolution is always better than what the current rules give. Some people have expressed concern about the potential size of the overload sets involved, but that's no different than for **operator+**, so this many not be a problem.

[4] (two notations with precedence) is backwards compatible, but will make some concept writers list both alternatives to cope with rare cases.

[3] (only generalize **f(x,y)**). would de-emphasize the OO notation and make it harder to provide online/IDE support a la "intellisence."

[2] (only generalize **x.f(y)**) is "too OO", encouraging increased coupling and discouraging lambdas.

[1] (status quo) causes problems for programmers, notably for library writers.

Discussion points

If we want 100% compatibility, we must choose [1], [2], [3], or [4]:

- [1] doesn't solve the problem
- [2] would de-emphasize the functional notation and encourage increased coupling. Concepts would be written **x.f(y)**.
- [3] would de-emphasize the OO notation. We don't think anyone likes it as a C++ extension (even though it may be seen as ideal if we had no compatibility constraints). Concepts would be written **f(x,y)**.
- [4] would give people a choice of notations. Since the notations would have slightly different meanings (to preserve compatibility) concepts will have to be written with a (slight) bias towards one notation or the other.

[5] is a technically better solution than [4] in that it will give identical resolutions for both notations and arguably the technically best solution in all cases. It is what we currently do for operators. If we choose [5], we should consider changing the rules for range-**for** to get a uniform rule.

[6] is (we think) a simpler solution for humans to fathom than [4]. It would lead to surprises (incompatibilities) only for people using the **f(x,y)** notation. And only for **f(x,y)** used where **x.f(y)** would give a different and inferior solution to **f(x,y)**. Bjarne conjectures that this is very rare, but can't prove it.

In cases [5] and [6] a compiler could warn if a function that would be invoked under current rules was not chosen.

A major motivator for Herb (though not for Bjarnee) is to encourage the use of **x.f(y)** to make "intellisence" more effective. Herb points out the effectiveness of an IDE making suggestions based on **x.** compared to making suggestions based on **f(** . After **x.**, a compiler can easily made a list of all possible member names that can appear after the dot. For **f(** a compiler must look through all active scopes to make a list of possible **f**s. To find the full set of alternatives, we need to know the full set of arguments to see if there are any candidate functions found by ADL. If no **f** is found after **x.f**(the compiler must look into the active scopes for an **f** (just like after a "plain" **f(**. In **x.f(**, **x** specifies an exact type, whereas in **f(x,**, **x** can be subject to conversions.

Bjarne doesn't doubt that there is an advantage to "intellisence" nor that the advantage is most significant when using member functions. However, that also gives a bias towards an asymmetric notation specialized on the first argument ("the object" is given a very special role). The implied encouragement to the use of member functions is a concern because it increases coupling. The prefix functional notation (e.g., **f(x,y,z)**) and suffix functional notation (e.g., **(x,y,z).f**) are general, but the infix object-oriented notation (e.g., **x.f(y,z)**) as specialized (and special purpose). We could generalize to allow **f(x,y,z)** and **x.f(y,z)** and **(x,y).f(z)** and **(x,y,z).f**) to address this (there is an example of that in D&E), but we don't propose to go that way; for starters, it would be confusing unless those 4 calls had identical meaning, but that can be achieved only by choosing an incompatible solution.

Should we get multimethods in the future, the **x.f(y)** notation will start to look even more "special purpose." Thus deciding on **x.f(y)** only would close a promising evolution path.

It has been conjectured that looking into a second scope would slow down compilation. It is not obvious that this slowdown would be significant (we already do it for operators). For [5] (overload resolution), the extra lookup would always happen. For [4] and [6] (relying on looking into one scope first), the cost would only be incurred when no match were found in the preferred scope.

Questions

1. Do anyone have a tool that let us scan a large body of code to see whether the concern over non-compatible resolution is real? To get a real problem, we would need an **x.f(y)** and an **f(x,y)** and their meaning must be different. Most cases where I see such two functions, then have the same meaning. To do a perfect job, such a tool would need to be type sensitive and know the C++ lookup and conversion rules, but maybe a non-perfect approximation could give useful data.
2. If we consider the compatibility problem significant, is [4] (Prefer member functions over free-standing functions for **x.f(y)** and prefer free-standing functions over member functions for **f(x,y)**.) sufficient for us to write concepts in terms on one call syntax only?
3. If we don't consider the compatibility problem significant, is [5] (overload resolution) or [6] (member preference) best?
4. Would it be enough to generalize only **x.f(y)**?

Our best answers are "we don't know", yes, "we are not sure", and no.

Suggested resolutions

We consider solutions [4] (two notations with precedence), [5] (overload resolution), and [6] (two notations and one semantics) acceptable. Based on the discussions in Urbana, let us consider [4] as our initial solution:

> Prefer member functions over free-standing functions for **x.f(y)** and prefer free-standing functions over member functions for **f(x,y)**. This is 100% compatible with current rules, but leaves **operator+** as a special case. Also, from the point of view of a concept, **x.f(y)** and **f(x,y)** are not completely equivalent. Gaby mentioned this alternative in EWG in Urbana.

This takes the compatibility issues off the table. Let us consider [5] (overload resolution), and [6] (two notations and one semantics) later if and only if we get evidence that the compatibility problems are negligible.

Choice of function

With [4] we have to look seriously at whether it solves the problem of supporting template interfaces (specified in terms of concepts). The meaning of **f(x,y)** and **x.f(y)** can differ, but will they – in realistic examples – differ in ways that will encourage template writers to define two functions in a concept?

Consider a general form of the problem:

```
struct S1 {
        int f(int);
};

struct S2 {

        int f(double);
};

int f(S1,double);
int f(S2,int);

void use(S1 s1, S2 s2)
{
        S1.f(2.8);      // prefer member: truncate
        f(s1,2.8);      // prefer nonmember
        s1.f(2);        // prefer member
        f(s1,2);        // prefer nonmember: promote to double

        s2.f(2.8);      // prefer member
        f(s2,2.8);      // prefer nonmember: truncate
        s2.f(2);        // prefer member: promote to double
        f(s2,2);        // prefer nonmember
}
```

This is obviously the exact behavior we have today. Hopefully, the truncations will yield warnings. The difference given uniform call syntax is simply that if we removed a member or a nonmember **f()**, the example would still compile and the meaning would be that of the now one and only **f()**.

```
struct S1 {
        int f(int);
};

struct S2 {
         // int f(double);
};


// int f(S1,double);
int f(S2,int);

void use(S1 s1, S2 s2)
{
        S1.f(2.8);      // truncate
        f(s1,2.8);      // truncate
        s1.f(2);
        f(s1,2);

        s2.f(2.8);      // truncate
        f(s2,2.8);      // truncate
        s2.f(2);        // promote to double
        f(s2,2);        // promote to double
}
```

Again, we can hope for warnings against the truncations. So far, so good. We gained a bit of flexibility at no real cost. But how about templates? Consider:

```
struct S1 {
        int f(int);
};

struct S2 {
        int f(double);
};

int f(S1,double);
int f(S2,int);

template<typename T>
void use(T t)
{
        t.f(2.8);       // prefer member
        f(t,2.8);       // prefer nonmember
        t.f(2);         // prefer member
        f(t,2);         // prefer nonmember
}

use(S1{});      // exactly as S1 before
use(S2{});      // exactly as S2 before
```

There are three ways the constrain **T** in **use(T)**:

- Favor members
- Favor nonmembers
- Explicitly cater for both

By commenting out a line or not, we can choose an alternative:

```
template<typename F> concept bool Fmem =
        requires(F a, int i, double d)  {
                { f(a,d); } -> int;             // begin() can be non-member
                { a.f(i); } -> int;             // begin()  can be member
        };
```

Picking one line (constraint) or the other will be sufficient for the realistic examples we can think of. If you feel otherwise, push for one of the less compatible alternatives: [5] (overload resolution) or [6] (two notations and one semantics).

Inaccessible and uncallable members

What if a call **x.f(y)** finds a members that cannot be called? Consider:

```
class X {
private:
        void f(int);
public:
        int g;

};

void f(X&,double);
void g(X&,double);

void use(X& x)
{
        x.f(2.9);       // calls ::f(double) or error?
        x.g(2.9);       // calls ::g(double) or error?
}
```

Equivalent examples can be constructed for uncallable global names hiding perfectly valid member functions (or callable objects). The solution is the same in both cases: Consider failure to find a function to call and finding something that cannot be called equivalent. In that case, try the other notation. In particular, the two calls in the example are valid.

Pointers

Calls using pointers can be handled in a way similar to the way object are handled

- For **p->f(y)**, see if **p->f(y)** is valid and if so do that call; otherwise if **f(p,y)** is valid do that call.
- For **f(p,y)**, see if **f(p,y)** is valid and if so do that call; otherwise if **p->f(y)** is valid do that call.

The main design choice here is whether to accept a "smart pointer" (an object of a class with **->** defined) as a pointer. We propose to accept "smart pointers" – they are common and useful, so they should not become second class citizens.

If a **p->f(y**) call fails, we look at **p** to see whether it is a pointer or of a class with an **operator->()**. If so we try **f(p,y)**. Similarly, if a **f(p,y)** fails, we look at **p** to see whether it is a pointer or of a class with an **operator->()**. If so we try **p->f(y)**.

This implies a bit of extra work for the compiler, but only in the case where a call currently fails.

Wording

The intent of the wording is:

- For **x.f(y)**, see if **x.f(y)** is valid and if so do that call; otherwise, if **f(x,y)** is valid, do that call.
- For **p->f(y)**, see if **p->f(y)** is valid and if so do that call; otherwise, if **f(p,y)** is valid, do that call.
- For **f(x,y)**, see if **f(x,y)** is valid and if so do that call; otherwise if **->** is defined for **x**, try **x->f(y)**, otherwise try **x.f(y)**.