WG-N4174
2014-10-11

Call syntax: x.f(y) vs. f(x,y)

Bjarne Stroustrup (bs@ms.com)

Abstract

This note explores the possibility of providing a uniform call syntax by giving member functions preference over non-member functions. Offering the choice between the **x.f(y)** and **f(x,y)** notations with different meanings means that different people will chose differently for their function definitions, so that users have to know the choice and write calls appropriately. This gives users more opportunities for making mistakes, makes it harder to write generic code, and has led to replication when people define both a member and a non-member function to express the same thing. I suggest that providing different meanings to the two syntaxes offers no significant advantage.

Several suggestions/proposals for dealing with this have appeared over the years (for example [Glassborow,2004], but most were not documented in formal documents). Operators, such as **==**, can be used with a uniform notation (e.g., **x==y**) independently of whether they are defined as members or non-members. Also, the rules for range-**for** are based on the idea to first look for a member function **x.begin()** and if one isn't found then look for a nonmember function **begin(X)**. Doing so generally would give a bit more control to class writers and simplify most common uses. This note explores this idea further.

This is a discussion of alternatives with a suggestion at the end. It is obviously not a complete and finished proposal.

The basic suggestion is to define **x.f(y)** and **f(x,y)** to be equivalent. In addition, to increase compatibility and modularity, I suggest we explore the possibility of ignoring uncallable and inaccessible members when looking for a member function (or function object) to call.

1 Introduction

Consider first a vague idea:

x.f(y) means

1. First try **x.f(y)** –does x's class have a member **f**? If so try to use it
2. Then try **f(x,y)** – is there a function **f**? If so, try to use it
3. otherwise error

f(x,y) means

1. First try **x.f(y)** – does **x**'s class have a member **f**? If so try to use it
2. First try **f(x,y)** – is there a function **f**? If so, try to use it
3. otherwise error

Does **f(x,y)** mean exactly the same as **x.f(y)**? That depends on the exact formulation of the idea above.

Ideally, we would have only one function call syntax, and ideally, that would be the conventional functional syntax, **f(x,y)**. The **x.f(y)** notation "favors" the first argument and suggests an asymmetry that exists some but not all cases: consider **x.sqrt()** and **x.operator==(y)**. The functional (mathematical) notation is far older and more general than the object-oriented dot notation. The dot notation is a result of single-dynamic dispatch in Simula and successor languages (such as C++). However, once we consider multimethods, the symmetrical functional notation become appealing even when relying on dynamic dispatch ([Stroustrup,1994], [Pirkelbauer,2007]). For example, I prefer **intersect(s1,s2)** over **s1.intersect(s2)**.

For compatibility, we must support both notations, but unless one syntax provides something fundamental that the other does not, ideally the meaning of **x.f(y)** would be identical to that of **f(x,y)**.

1.1 What does x.f(y) mean?

The first case, **x.f(y)**, appears to be the simplest. First, just apply the current rules and if that would lead to a compile-time error, try **f(x,y)**. For example:

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

int f(X&,int);
X x;
x.f(2);         // OK: call X::f(int)

int g(X&,int);
x.g(2);         // OK: call g(x,2);
```

A member function can hide a nonmember function that apparently is better match. For example:

```
struct X {
        int f(double);
};

int f(X&,int);
x.f(2);         // OK: x.f(double(2)); not f(x,2)
```

One way to avoid such "hijacking" by a member function is to generate an overload set from member and nonmember functions, and pick the best match. However, if that is done, the nonmember function could be seen as doing the hijacking – that would be most surprising to someone used to the current rules and there is an opportunity for silent change of meaning compared to all earlier versions of C++.

Furthermore, member functions are often used specifically to limit scope (limit the overload set), so ***not*** giving priority to member functions would take away a significant and widely used feature.

Giving priority to members respects people's naïve/historical expectations from the syntax used and also gives the writer of a class the ability to control what needs to be controlled. The only new aspect is that a call **x.f(y)** will work whenever the writer of **x**'s class has not expressed an interest in the name **f**, but the writer of the surrounding scope has.

1.2 Member type and visibility

So far, I have considered only public function members. What about private members? Data members? Member function objects? Member types? **static** member functions?

The most compatible solution would be to consider them all, exactly as today. That would make the proposal a pure extension. However

- Should a data member hide a perfectly good function?
- Should a member type hide a perfectly good function?
- Should a private member hide a perfectly good function?
- Should a member that cannot be called with the given arguments hide a perfectly good function?

Considering only, the **x.f(y)** notation, "yes: members hide nonmember functions" would be an acceptable (though not ideal) answer.

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

int f(X,int);
X x;
f(x,2);   // error: X::f() Is private

int g(X,int);
g(x,2);   // error: X::g is not a function
```

This mirrors other lookup rules (lookup before access check and type check) and makes the answer independent of whether that call is done from within the class or outside it. For example:

```
int g(X,int);

class X {
        int f(int) { x.g(2); }        // error: X::g is not a function
public:
        int g;
```

```
};
```

However, having a call to an accessible function masked by a non-function or a private member is at least surprising. Fundamentally, it makes calls that would otherwise work be vulnerable to changes in the implementation of a class. This becomes more serious when we start considering the **f(x,y)** notation (§1.3).

Public member function objects are perfectly callable under the current rule. For example:

```
struct Op {
        double operator()(double d);
};

struct X {
        Op f;
};

int f(X,int);
X x;
x.f(1);         // OK: call X::f
x.f(1.1);       // OK: call X::f
```

However, we still cannot overload a function with a function object

```
struct X {
        int f(int);
        Op f;    // error: can't overload
};
```

That may be a separate problem. The suggestion at the end of this paper does not try to address this.

Finally, consider the possibility of a member with a "wrong" argument type:

```
struct X {
        int f(string);
};

int f(X,int);
X x;
f(x,2);   // error: X::f() requires a string argument
```

I think this is the correct resolution (independently of how private and noncallable members are handled). Bypassing/ignoring functions that are not callable with a given argument could lead to quite brittle code.

1.3 What does f(x,y) mean?

Now consider **f(x,y)**. We could

1. give priority to a non-member function
2. give priority to a member function
3. do overload resolution across the member and non-member scope

Alternative [1], nonmember priority, would be backward compatible and respect people's naïve/historical expectations from the syntax. In addition, it would allow member functions to be accessed without the special dot notation. For example:

```
struct X {
        int f(double);
        int g;
};

int f(X&,int);
X x;
f(x,2);         // OK: call x.f(2)

int g(X&,int);
g(x,2);         // OK: call g(x,2)

int h(X&,int);
h(x,2);         // OK: calls nonmember h()
```

Alternative [2], member priority, gives the designer of a class novel control of the meaning of functions. Basically, **f(x,y)** becomes equivalent to **x.f(y)** for every member **f** of **x**'s class. For example:

```
struct X {
        int f(double);
        int g;
};

int f(X&,int);
X x;
f(x,2);         // OK: call x.f(double(2))

int g(X&,int);
g(x,2);         // error: tries to call x.g(2)

int h(X&,int);
h(x,2);         // OK: calls nonmember h()
```

Note the conversion to **double**. Member priority allows the class to "hijack" using a function that would be an inferior match under overload resolution. With member priority, **x.f(y)** would have the same effect as a nonmember **f(x,y)**. As with **x.f(y)**, we must consider if we can modify the lookup rules to exclude noncallable members. For **f(x,y)**, that would be more significant.

Alternative [3], overload resolution, simply adds a member function to the overload set considered. For example:

```
struct X {
        int f(double);
        int g;
};

int f(X&,int);
f(x,2);         // OK: call f(x,2) not x.f(double(2))

int g(X&,int);
g(x,2);         // error: cannot overload function and nonfunction
```

Now the class can "hijack" the call only if it is the best match. However, we might modify the overload resolution rules to exclude noncallable names and get

```
g(x,2);         // OK: call nonmember g()
```

Further, we would have to decide how to handle a function object. For example:

```
 struct Op {
        double operator()(double d);
};

struct X {
        Op f;
};

int f(X,int);

X x;
x.f(1);         // overload? Call nonmember f?
```

Given lambdas, it may not be easy to exclude function objects from consideration.

1.4 What about p->f()?

If **x.f()** can call **f(x)**, can **p->f()** call **f(p)**? Yes, and **f(p)** can call **p->f()**.

1.5 Tool support

Herb Sutter ([Sutter,2014]) and others point out that an important aspect of **x.f(y)** is that it limits the scope of **f** to **x**'s class (incl. base classes). The reason that **x.f(x)** is easier to deal with than **f(x,y)** is that the scope of **f()** is designated (by **x**) and limited (to the members of **x**'s class). Unfortunately, that scope is also closed. That problem is handled by looking for **f()** in the current scope of **x.f()** after looking into **x**. That's two lookups. For **f(x,y)**, we also need to look at two scopes, the same two scopes as for **x.f()**. With the suggestion of member priority, we even have to look into those scopes in the same order.

As Herb points out in [Sutter,2014], a tool can look into **x**'s scope and find all possible members after just **x.** and after seeing **x.f** it can find all fs. Conversely, after **f(**, a tool can find the full set of nonmember **f**s and after **f(x** it can see both scopes and determine the full set of possible **f**s. Herb and others consider this difference in order significant. I agree that the members are more important that the nonmembers in that the members take priority. However, my guess is that the problems with the two notations are complementary and manageable. Also, I consider the **f(x,y)** notation fundamental and unavoidable.

2 Evaluation

Here, I'll look at the call syntax and the possible impact of a change in lookup rules to lessen the impact of class implementation details on calls.

2.1 x.f(y)

I think we have to give members priority when the **x.f(y)** notation is used. Overloading isn't sufficiently compatible and we need a way to say "give me the member function (if it exists)." For **x.f(y)**, member priority is a compatible extension: it turns previously illegal examples into valid code without changing the meaning of existing programs.

For **x.f(y)**, we can choose between considering all names in a class (as currently done) and ignoring nonaccessible and/or noncallable members. Either choice would make some previously illegal programs valid. That would be compatible, and convenient for programmers. The latter choice would increase modularity and make calls less vulnerable to changes in a class' implementation. It would also require work on lookup rules (§2.3).

Unfortunately, choosing the (current) fully compatible solution with no changes to lookup rules of **f(x,y)**, gives only an illusion of stability. If **x.f(y)** handles more real world cases than current or "modified" **f(x,y)** (is "more generic"), the pressure to rewrite generic code in terms of **x.f(y)** becomes irresistible. However, when that is done, generic code becomes vulnerable to noncallable and inaccessible members. The programmer will then be faced with a most unpleasant choice between generality (relying on **x.f(y)**'s novel ability to invoke both member and nonmember functions) and protection against implementation details of classes (relying on the current meaning of **f(x,y)**).

2.2 f(x,y)

Choosing a meaning for **f(x,y)** is less obvious:

- Alternative [1], nonmember priority, is a pure extension, but allows a nonmember function to hide even a perfect match in the class. The designer of the class has no control over the meaning of a call using the conventional function call notation (like today). This is the simplest and most compatible solution. It does, however, differ from the (current) resolution of operators and for range-**for**. It also makes **f(x,y)** differ from **x.f(y)**.
- Alternative [2], member priority, makes **f(x,y)** and **x.f(y)** equivalent. The designer of a class has strong control of the meaning of **f(x,y)**. This is the rule used for operators, such as **==**, today, and for **begin()** and **end()** in a range-**for** loop.

- Alternative [3], overload resolution, gives the most fine-grained resolution, but can give a class writer a false sense of being in control. It is also a bit more work to specify than alternative [2]. We would have to (somehow) exclude non-function members and deal with the possibility of overloading a function object and a function.

Consider the problem from the point of view of a programmer who would prefer not to have to know whether a class designer uses a member or a nonmember function to implement an idea. In particular, consider the plight of the writer of a generic algorithm. Unless we modify the lookup rules, **x.f(y)** and alternative [2] of **f(x,y)** allow private or data members in a class to block the use of a nonmember function **f**. I do not know how serious a problem this is.

By using the (for generic algorithms) conventional **f(x,y)** notation, we have to either abandon hope of using member functions, use **x.f(y)** systematically, or choose among the three alternatives.

- Alternative [1], nonmember priority, leaves us open to having a best match ignored. That best match would often be the function or function object most explicitly associated with the class (as a member). We would, like currently, have to duplicate functions (one member plus one nonmember) to ensure consideration of a member. Note that **virtual** functions must be members and that the most fundamental functions are often members.
- Alternative [2], member priority, leaves us open to current programs changing meaning when a member is chosen over a nonmember. One would hope that (as is common) **f(x,y)** means the same as **x.f(y)** when **f()** is a public member, but that cannot be guaranteed. Also a currently valid **f(x,y)** could become an error because of a private or noncallable members of **x**'s class. Note that these problems can occur only for the first argument of a nonmember function, rather than all arguments. To minimize the second problem, I think that we would have to choose to ignore private and noncallable members when looking for a nonmember function.
- Choosing alternative [3] (overload resolution) would force us to solve the overloading of functions and function object problem.

For many uses, I prefer the (most traditional) functional notation **f(x,y)**. For example, I would hate to have to write **x.sqrt()** rather than **sqrt(x)** to guard against someone defining **sqrt()** as a member of some class.

2.3 Vulnerability

By using a lookup/selection rule that do not consider inaccessible and non-callable members, we would minimize the "vulnerability" of the call mechanism to "irrelevant implementation details." For example:

```
struct X {
        int f(double);
        int g;
private:
        int h(int);
};
```

```
int f(X&,int);
X x;
f(x,2);          // OK: call x.f(double(2)) – novel resolution

int g(X&,int);
g(x,2);          // OK: call g(x,2)  -  novel resolution: ignore X::g

int h(X&,int);
h(x,2);          // OK: call h(x,2)
```

Unfortunately, this would be novel and may cause implementation problems.

The resolution of **x.f(2)** points to the most serious potential problem with considering for **f(x,y)** equivalent to **x.f(y)**: silent selection of a different function. I cannot quantify how real or how serious this problem is. To become a problem requires an x such that

- **x.f(arguments)** and **f(x,arguments)** are both valid today and
- the meaning of the member and nonmember functions are not identical and
- the member function call isn't an equally good or better resolution of the call.

By the latter, I primarily refer to the (many) cases where people today have added a forwarding function to get the effect I am proposing. For example:

```
struct Container {
        Iterator begin1();
        Iterator begin2();
        // …
};

Iterator begin1(Container& c) { return c.begin1(); }

Container xxx;
auto p = begin1(xxx);    // novel resolution
auto q = xxx.begin2();   // as ever
```

We need to look at some significant amount of code to see if this is viable.

We should seriously consider changing the lookup rules to ignore inaccessible and uncallable members. Something like this must be seriously being considered for modularity anyway.

If **x.f(y)** handles more real world cases than **f(x,y)** (is "more generic"), the pressure to rewrite generic code in terms of **x.f(y)** becomes irresistible. However, when that is done, generic code becomes vulnerable to noncallable and/or inaccessible members. Modifying the meaning of **x.f(y)**, but not **f(x,y)** will not allow us to escape this dilemma.

I instituted the current rule, "first select the best member, then check if it is usable", for what seemed like good reasons at the time (1983 or so) and it has served us reasonably well. I knew that there was no

perfect solution, but a decision had to be made. The language and the pressure on the rules are different today. We should at least consider a change.

The main effect of a change would still be to make previously invalid examples work. That's compatible in that it turns previous errors into valid code. For example, a private function in a derived class would/might no longer hide a viable alternative in a base class:

```
struct Base {
        void f(double);
};

class B : public Base {
public:
        void g(double);
        void h();
private:
        void g(int);
        void f(int);
};

void current(D& d)      // current rules
{
        d.f(1);   // error: tries to call D::f(int)
        d.g(1);  // error: tries to call D::g(int)
}

void alternative(D& d)  // an alternative to consider
{
        d.f(1);   // OK: ignores D::f(int) and calls Base::f(double)
        d.g(1);  // OK: ignores D::g(int) and calls D::g(double)
}
```

This example is meant to illustrate the difficulties of a change. However, if we changed the rules in general, rather than just for **x.f(y)**, we should be able construct examples that would give different answers. So far, I have not found any. However, it is easy to construct examples where a call from a member (or friend) resolves to something different from a call from a nonmember. For example:

```
void Derived::h(D& d)   // compare to alternative()
{
        d.f(1);   // OK: calls D::f(int)
        d.g(1);  // OK: calls D::g(int)
}
```

Avoiding such differences was one of the reasons for the current rule. Note that such differences may be "odd" but they do not make the change I'm considering – with the hope of feedback – incompatible. In this case it changes an error into a resolution that might be considered surprising.

If the change in lookup rules is viable it would help us achieve **x.f(y)==f(x,y)**, which would be a major simplification – especially for the writers of generic code. It may open the door for multiple dispatch ([Pirkelbauer,2007)] and for a uniform notation for dynamic and static dispatch.

3 Conclusion

I think a proposal like this is viable (though obviously it needs more work). Of the alternative, we must choose member-priority for **x.f(y)**. For **f(x,y)**, I suggest that member priority is by far the best answer yielding **x.f(y)==f(x,y)**. We should seriously consider if we can ignore uncallable and/or inaccessible members when looking for functions in a class.

4 References

[Glassborow,2004] F. Glassborow: *Uniform Calling Syntax (Re-opening public interfaces)*. N1585.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.

[Pirkelbauer,2007] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup: *Open Multi-Methods for C++*. Proc. ACM 6th International Conference on Generative Programming and

[Sutter,2014] H. Sutter: *Unified Call Syntax*. N40xx.

5 Acknowledgements

Thanks to the many who have considered and discussed this problem with me over the years, notably Gabriel Dos Reis, Francis Glassborow, Herb Sutter, and Andrew Sutton.