Extension methods for C++

ISO/IEC JTC1 SC22 WG21 - P0079R0

(Uniform-function-calling syntax lite)

Jonathan Coe <jbcoe@me.com>

Roger Orr <rogero@howzatt.demon.co.uk>

Motivation

When writing generic code, we are forced to differentiate between calling member functions and free functions: `x.foo()` and `foo(x)` are not the same thing. When using concepts to constrain generic code we will run into the same issue and will have to (arbitrarily) pick member functions or free functions to describe the concept. This is undesirable and has led to the standard library introducing free function equivalents for member functions like `begin`, `end` and `data`. In all cases the free function invokes the member function where it is defined.

There have been proposals [N1585, N4165, N4174, N4474] suggesting changes to C++'s function resolution rules to allow free function invocation syntax to invoke member functions and vice-versa. We are concerned both about the impact such changes may have on existing code (although this concern has been addressed in some of the more recent proposals) and also about the potential for code maintenance problems, were unconstrained uniform function calling to be adopted, when class interfaces are changed since any free function that could potentially be called using member function syntax might have been so used.

We propose an explicit opt-in alternative by introducing extension methods and using concept-constrained free functions.

Constraining free-functions with concepts

Without language changes, we can introduce free functions that will be applied only to objects which support member function(s) with a given name and signature (This may be a weak abuse of the notion of concepts.). Any user-defined type that implements the member functions required by the concept can have the concept-constrained generic free-function invoked upon it.

```
struct A {
  void foo(int i);
};
```

```
A a;
foo(a, 0); // will not compile

template <typename T>
concept bool MemberFooable() {
  return requires(T& t, int i) {
    {t.foo(i)} -> void;
  };
}

void foo(MemberFooable& t, int i) {
  t.foo(i);
}

foo(a,0); // calls a.foo(0)
```

The above is a cute trick but gets us half way to uniform function calling syntax. By using concepts and writing a little bit of boilerplate code, we can allow member functions to be invoked with free-function syntax even for yet-unseen usef-defined types.

The second part, allowing member function syntax to invoke a free function requires language changes: extension methods.

Adding extension methods to C++

Extension methods are free functions that can be invoked with member-function syntax. Adding extension methods to c++ requires a language change. Addition of extension methods was previously proposed [N1585] but not pursued further. We believe that the impending introduction of concepts into C++ makes them a vastly more appealing proposition.

For the struct `B` we introduce an extension method `bar` below:

```
struct B;

// 'this' makes 'bar' an extension method
void bar(B* this, int x);

B b;
b.bar(1); // invokes bar(&b, 1)
```

The first argument of an extension method is named `this`. When invoked with member function syntax, arguments to the member function become second and later arguments to the extension method. Extension methods do not need to be invoked as member functions but can be invoked as free functions.

The behaviour of extension methods is designed to minimise the impact on existing well-formed code (Ignoring SFINAE tricks).

Overload resolution

The decision to invoke an extension method is made at compile time as part of overload resolution. If a member function could be invoked with the supplied arguments then the member function will be chosen in preference. Extension methods should only be invoked when there is no possible member function invocation.

```
struct D {
  void foo(unsigned u);
};

void foo(D* this, int i) {
  // some implementation
}

D d;

d.foo(3); // invokes D::foo(3);
```

Adding the extension method does not break existing code, but the behaviour may be undesirable. Compiler warnings may be desirable if an extension method has entered an overload set but is not selected.

Non-public or deleted member functions can prevent an extension method from being invoked as they will be preferentially selected during overload resolution but rendered non-invokeable.

Access to class members

Extension methods are like normal free functions apart from their invocation syntax and have no access to non-public members of a class.

```
struct E {
  private:
  void foo(int i);
};
```

```
void bar(E* this, int i) {
  this->foo(i); // will not compile as E::foo(int i) is private
}
```

Interaction with virtual functions

Extension methods do not interact with virtual functions. If a derived class has a virtual method with the same signature as an extension method that is not present in the base class then the extension method will be invoked if accessed through a base-class reference.

```
struct F {
  virtual void foo(int i);
};

void bar(F* this, int i) {
  this->foo(i+1);
}

struct G : F {
  virtual void bar(int i) {
    foo(i-1);
  }
}

G g;
g.bar(0); // invokes member G::bar(0)

F& rf = g;
rf.bar(0); // invokes extension method bar(&g, 0)
```

The non-invocation of derived class member functions may be non-desirable but ensures that adding an extension method cannot change the behaviour of compiling code. Adding a member function to a class where extension methods are defined with the same signature may result in non-obvious behaviour. Compiler warnings may be desirable. [Note that the an issue very similar to that above is already present when derived class methods shadow virtual base class methods.]

Combining extension methods with concepts

We can write generic extension methods and constrain them with concepts. Concept-constrained extension methods can be used to enable member-function-style invocation only when a given free function invocation is valid.

```
class C;

void foo(C& c, int x);

C c;
c.foo(0); // will not compile

template <typename T>
concept bool FreeFooable() {
  return requires(T& t, int i) {
    {foo(t, i)} -> void;
  };
}

// Naming the first parameter 'this' makes 'foo' an extension method
void foo(FreeFooable* this, int i) {
  foo(*this, i);
}

c.foo(0); // calls foo(&c,0)
```

The concept-constrained code above allows member function invocation to work for any type for which an equivalent free function can be found.

Comparison with existing uniform call proposals

The extension methods proposed by N1585 do not require `this` to be unique nor for it to be the first function parameter. This proposal follows the extension method approach used in the C# programming language [C#].

Unlike N4165, N4174 and N4474, the uniform calling syntax this paper enables is opt-in and requires explicit addition of boilerplate code. The changes proposed to overload resolution rules by this paper are slight and will have no effect on existing code.

Conclusion

We have proposed no changes to existing overload resolution rules that could cause existing valid code to fail to compile or to change its meaning. The addition of concept-constrained free functions or extension methods into existing valid code will not cause it to fail to compile or change its meaning (ignoring SFINAE-based tricks). Selective use of concept-constrained free functions and extension methods will allow opt-in uniform-calling syntax.

Open questions and bikeshedding

Should `this` in extension methods be a reference or a pointer? It really should not be null, so a reference is appealing. In all current contexts where `this` appears in C++, it is a pointer. The decision is whether to value consistency or correctness. The author(s) have selected consistency but this has no deep implications on the proposed changes.

References

- [N1585] *'Uniform Calling Syntax (Re-opening public interfaces)'*, Francis Glassborow, N1585.
- [N4165] *'Unified Call Syntax'*, Herb Sutter, N4165.
- [N4174] *'Call syntax: x.f(y) vs. f(x,y)'*, Bjarne Stoustrup, N4174.
- [N4474] *'Unified Call Syntax: x.f(y) and f(x,y)'*, Bjarne Stoustrup and Herb Sutter, N4474.
- [C#] https://msdn.microsoft.com/en-us/library/bb383977.aspx

Appendix

Setting aside uniform calling syntax, we can employ extension methods very usefully to add functionality to existing classes. Adding an extension method to `std::string` to see if a substring appears at the beginning or end is as easy as:

```
using std::string;
using std::begin;

namespace my_extension_methods {
  bool begins_with(const std::string* this,
                   const std::string& substring) {
    return this->compare(0, substring.length(), substring) == 0;
  }
}

std::string s("Extension methods are useful in their own right.");

using namespace my_extension_methods;

assert(s.begins_with("Extension methods"));
```