Doc No: SC22/WG21/N1585 = 04-0025

Project: JTC1.22.32

Date: Thursday, February 05, 2004

Author: Francis Glassborow

email: francis@robinton.demon.co.uk

# Uniform Calling Syntax (Re-opening public interfaces)

## 1 The Problems

Member functions and free functions have different call syntax. This makes it necessary to know if a function is a member function or a free function even if its semantics is substantially independent of the scope in which it is declared. Several examples can be found in the Standard C++ Library where the programmer is required to know what is effectively an implementation detail. For example `swap()` exists both as a member function of some collection classes and as a free function. `getline()` is both a member function of `std::istream` and a free-function (to read an `std::string` from an `istream`).

Template technology is hampered by different syntax being used to achieve the same objective (note that we fixed an issue with forwarding functions so that a function returning `void` could use the same syntax as other functions).

Unfortunately the different calling syntax is not the only difference between member functions and free functions. They necessarily have different scopes (which affects name look-up and overloading) and they have different access rights. There is also an issue of dynamic binding, member functions can be virtual, free functions cannot be.

There are good reasons to limit the in-class public interface

Any solution must take all these things into account. Strictly speaking it is already possible to export the public interface into the enclosing namespace (or any other namespace) by using inline forwarding functions. For example:

```
class ex {
public:
  void foo();
};
inline void foo(ex& e){return e.foo();}
```

While it is possible to specify a syntax to automate that export process (actually two variations, one at the point of declaration of the member function, and one in the namespace where the inline declarations would appear) there is sufficient complexity to make me hesitant to pursue that here.

The following proposal is intended to deal with the reverse process: a way to inject free functions into a class's public interface.

## 2 The Proposal

Add a special variant syntax for declaring a free-function that will place it in the class scope for the purposes of overloading, name look-up when called with a member function call syntax but not provide access to non-public members of the class. This latter is important because it ensures that the mechanism does not break the data-hiding aspect of encapsulation.

The proposed variant syntax is to allow any reference (possibly cv qualified) parameter to be declared with the name `*this`. A function declared with this variant syntax can be treated exactly as if it were a public member of the parameter type except that it has no access to non-public members of the class. If it is a cv qualified class type then it will be as if the equivalent member function were a cv qualified member function.

The selected parameter need not be the first parameter, nor need it be unique. I.e. a free function declaration using this variant syntax may be injected into the public interface of more than one class. E.g.

```
template<class chart, class traits, class Allocator>
  basic_istream<chart, traits>&
    getline(basic_istream<chart, traits>& *this,
      basic_string<chart, traits, Allocator>& *this,
      chart delim);
```

Declares that the results of instantiating this function template is a free function that has been injected into the relevant class interfaces `getline()`. [note that the definition context does not require the use of the variant syntax, and in the above case will not be able to use it because it needs to distinguish between the two class parameters] In the context of the above template declaration:

```
int main(){
  std::string s;
  std::cin.getline(s);
  s.getline(std::cin);
  getline(std::cin, s);
}
```

the three calls of `getline()` call the same body [as they are exact matches the overload resolution should be the same in all cases]. However the overload resolution will depend on the exact call. The first one overloads in the context of `istream`, the second in the context of `string` and the third in namespace `std`.

If such a free function cannot be distinguished from a member function then the member function is preferred (i.e. that is the final determiner in overload resolution).

Example:

```
class ex {
  public:
    ex(int j_val = 0): j(j_val){}
    int value() const{return j;}
    void foo();
  private:
```

```
    int j;
};
ostream & print_on(ostream & out, ex const & *this){
  out << (*this).value();
  return out;
}

void foo(ex & *this);
```

Given the above:

```
ex e;
e.print_on(cout);
```

and

```
ex e;
print_on(cout, e);
```

Are equivalent. However:

```
e.foo(); // calls the member function
foo(e);  // calls the free function
```

## Discussion

In effect this proposal provides for a strictly limited facility to 'reopen' a class's public interface. This could have advantages in making some features of the Standard C++ Library more uniform. For example a simple change of the declaration of `getline` in 21.3.7.9 by inserting `*th` before the '`is`' would result in removing any requirement to remember that `getline` for `std::string` is a free function that does not overload with the member functions of an `istream`.

I need a template specialist to consider if there are consequential implications on when a template is 'used' or instantiated but as far as I can see this proposal will not add any additional burden.

The special variant declarative syntax is only needed for the primary declaration, the rule that allows definitions to use different parameter names is not touched. However we might require that the definition of such a function must see a primary declaration and the class definition (just as the definitions of member functions require the class definition to be visible.)

Note that this proposal cannot have any direct implications on existing code because it is currently ill-formed to use `*this` as the name of a parameter in a declaration.

There are various issues that need to be considered. These injected members cannot, in my opinion, be treated as full members. For example not only do they lack access to non-public members but they also cannot be used through the pointer to member mechanism.

However, though this mechanism impacts on name lookup it does so in a clearly defined manner. For example in the context of inheritance name lookup issues are resolved exactly as they would be for an actual member function. The programmer must be careful that an injected name into a derived class does not hide overloads in a base class.

We have that problem today with normal class members and resolve it with a using declaration. In the case of injected members we could decide that such hiding (i.e. the derived class does not have the name brought forward with a using declaration) is to be allowed or to make it ill-formed. My personal preference is the latter.

## Changes to the Working Paper

These are not provided at this point. It seems more important to agree to pursue this idea or simply abandon efforts in this direction.