Doc No: SC22/WG21/ N1742=04-0182

Project: Programming Language C++

Date: Sunday, November 07, 2004

Author: Francis Glassborow

email: francis@robinton.demon.co.uk

Auxiliary class interfaces

(This is a replacement for the paper N1716 distributed on the WIKI under the title *Re-opening Class Interfaces*)

1 The Problems

The proposal in this paper addresses three arguably serious problems:

1) Providing a mechanism for programmers to provide extra functions that will be called with the member function syntax.

2) Providing a mechanism to support removing private members and replacing them with declarations in the class implementation file.

3) ADL problems when using namespace scope functions.

Several other minor warts get addressed as side effects of the proposal:

4) It becomes possible to add functionality to a class without using the sometimes dubious use of inheritance

5) It will be possible to declare much of the functionality of a class alongside a forward declaration. This will reduce the number of places where a class definition is needed.

6) A variant of 4 is that the user will be able to select the functionality they want to use.

Solving problem 1 is particularly valuable in the context of templates because currently the template author has to deal with the problem of member and free functions having different call syntax. This proposal will enable a programmer to require any function with a natural parameter that is a reference to a class type to be called with a member function like syntax.

Problem 2 is one that is often commented on. `private` members are inherently part of the implementation of a class, yet they have to be exposed in the class definition. Such exposure has several undesirable effects. Among them is the forced recompilation of all dependant files if the implementer of a class adds or changes a declaration of a `private` member. It also sometimes results in dragging in other header files because a `private` member function has a parameter of some other user-defined type. In the presence of templates this can cause a considerable cascade of file inclusions.

Using member function call syntax removes ADL problems.

Problem 4 is small but a common one for novices and incidental programmers. Providing an alternative way to add a function to the public interface of a class will reduce the temptation to use inheritance for this purpose.

Problem 5 concerns the fact that the current rules make forward declaration of a class less useful than it might be. If we could expose the much of public functionality of a class without exposing the private declarations of the data, the ability to work with references becomes much more powerful. In the discussion section of this paper I will show how much more can be achieved by using auxiliary class interfaces.

Problem 6 occurs because programmers sometimes want to limit the functionality of a class that will be available in a given context. Currently they usually (ab)use inheritance to achieve this end.

2 The Proposal

Provide syntax to declare a new set of (extendable) interfaces for a class called 'auxiliary interfaces'. This syntax to closely follow the existing syntax for a class definition but is not a class definition, only a class declaration.

```
friend class example {
public:
// here are declarations and inline definitions of
// auxiliary members that can only access the public
// members of example
protected:
// here are declarations and inline definitions of
// auxiliary members that can only access the public and
// protected members of example these members can only be
// accessed by class members, derived class members, class
// friends, and by protected or private auxiliary members.
private:
// here are declarations and inline definitions of
// auxiliary members that have full access to the class
// members these members can only be accessed by class
// members, class friends, and by other private auxiliary
//  members.
};
```

An auxiliary class interface is one in which:

1) Functions can be declared with both restricted access from outside and restricted access to the internals of the class.

2) An auxiliary member function may **not** share a name with a member function (i.e. no overloading between member functions and auxiliary member functions.)

3) An auxiliary member function may not be declared as virtual

4) An auxiliary member function may not be a special member function, nor may it be a constructor. (If forwarding constructors are accepted, the latter might be relaxed)

5) An auxiliary interface behaves like a namespace in that it can be re-opened and it also follows the name-lookup rules (only names with visible declarations will be found).

6) An auxiliary interface may not include declarations of instance data. However it can include `typedef`s, nested type definitions and static data members and functions.

7) A single translation unit may include multiple auxiliary interface declarations that add declarations for different auxiliary members.

8) Functions that are defined in an auxiliary class declaration are treated as being declared as `inline`.

The first of the above needs some explanation. Just as a class has three interfaces (`public`, `protected` and `private`) it will, under this proposal, also have three auxiliary interfaces. The `public` auxiliary interface has no special access privileges to its class; it only has access to `public` members of the class. However it can be used from anywhere as long as the declaration of the name is visible. A `protected` auxiliary member can only be used by entities that have access to the `protected` interface of the class and such members only have access to the `public` and `protected` interfaces and `public` and `protected` auxiliary interfaces of the class. A `private` auxiliary member function has full access to its class but can only be used by entities that have `private` access to the class (i.e. is a member of the class, is a `private` auxiliary member or is a `friend` of the class.

A major difference between class interfaces and auxiliary class interfaces is that the former are fixed by the class definition but the latter are extensible (in a way that is similar to the way that a namespace is extensible.)

The reason for constraints 2 and 4 above is to ensure that the class behavior cannot be subverted by adding an overload as an auxiliary member function. However note that 2 is not intended to prohibit overloading between different auxiliary interface declaration blocks.

Constraint 3 is because I cannot currently see a way to allow re-opening an auxiliary interface (an essential requirement for auxiliary interfaces) with the possibility of virtual auxiliary member functions. The current mechanisms for supporting virtual functions would not work with such a possibility.

Item 6 above is to allow as much latitude as possible to move both implementation and usage details into the auxiliary interfaces.

Under this proposal a class definition can be restricted to public (and sometimes protected) members that need access to the private/protected interfaces or auxiliary interfaces of the class, virtual member functions and instance data.

Mechanism

Provide the following syntax:

```
friend class example {
public:
// here are declarations and inline definitions of
// auxiliary members that can only access the public
// members of example
protected:
// here are declarations and inline definitions of
// auxiliary members that can only access the public and
// protected members of example these members can only be
// accessed by class members, derived class members, class
// friends, and by protected or private auxiliary members.
private:
// here are declarations and inline definitions of
// auxiliary members that have full access to the class
// members these members can only be accessed by class
// members, class friends, and by other private auxiliary
//  members.
};
```

The `'friend class example'` (or whatever syntax is chosen for the declaration) acts as a forward declaration for the class. That means that references and pointers to `example` are possible. That means that any of the `public` auxiliary functions are also available for use. The above usage will require an allowance for 'forward' declaring a type name that will be provided by a `typedef`. I think that is a desirable minor extension as long as we require that in such usage the `typedef` provides a type name for a `class`. However if that extension proves to be unacceptable it should still be possible to use the above code but with suitable minimal added declarations such as:

```
template<typename charT,
          typename traits = char_traits<chart>,
          typename Allocator = allocator<chart> >
class basic_string;
typedef basic_string<char> string;
```

Auxiliary member functions, `static` member functions and `static` data members are defined using the class syntax. The restriction on no overloading between members and auxiliary members ensures that the intentions of the class designer are not accidentally subverted by overloading member functions with auxiliary member functions. (see *further discussion* below).

Discussion

Like namespaces, it is intended that multiple declarations of auxiliary interfaces be allowed and that such declarations need not (and in general will not) declare the same members. For example, a class implementer would declare an auxiliary interface in the class implementation file that would provide all the `private` members that support the implementation of the class. On the other hand an entirely different declaration of

`public` auxiliary members would likely be provided in the class header file or even in an entirely distinct header file. Where more than one auxiliary interface declaration is visible for a class the compiler will synthesize a union of the member declarations. Incompatible redeclarations (e.g. with different return types but identical parameter types) should be a diagnosable error. Irresolvable overloads should be ambiguous if selected but otherwise should not be erroneous.

Normally the `private` auxiliary interface will be declared in the class implementation file and so will be completely invisible to users of the class. The only cases where it will be necessary to place a `private` member function in the class definition is when it is a virtual member or when the class designer wishes to provide a `private` overload of a member function that also has a `public` version.

The `public` auxiliary interface provides the potential for a new idiom in which even the data of the class becomes invisible to many users:

```
friend class example {
public:
  static example * create();
// possibly other overloads of create();
  static bool destroy(example *);
// other auxiliary members of example
};
```

Given a class definition it is possible for a user to add this idiom even if it was not originally provided by the class designer. In many cases the `destroy()` function would not be needed as the return value from a `create()` function will be a raw pointer to a dynamic instance and so the underlying object can be removed by a `delete` expression.

There are quite a few details that need to be worked on including auxiliary interfaces for templates and considering how best to deal with auxiliary interfaces in the context of inheritance. For the latter I think that a treatment of them as if they are a form of namespace would be a viable way forward.

There is also the issue of function pointers. It seems to me that there should be little problem in supporting pointer to member function syntax. However I would favor also allowing plain pointer to function syntax by treating auxiliary member functions as having an extra implicit parameter of type reference to or `const` reference to the class type. This implicit parameter being treated as if it were the first parameter.

Example of use of an auxiliary interface

In these examples I have used `std::string` and `std::istream` as if they were class names rather than `typedefs` for specializations of templates. It is intended that should the proposal be accepted these examples would work. The examples have been chosen because they offer a solution to an existing problem that causes problems to some programmers.

Consider:

```
typedef char stringdata[100];
void foo(istream & in){
```

```
  stringdata data;
  in.getline(data, 100);
// rest of definition
};
```

Note that if that `typedef` is changed to:

```
typedef std::string stringdata;
```

We have to change the calling syntax used for `getline()` to a free function one. While ADL is unlikely to have adverse effects here, there are other situations with user provided functions where it becomes a consideration. However with the proposed language extension a programmer could write:

```
friend class std::istream{
public:
  std::istream & get_line(std::string & s, size_t = 0){
    return std::getline(*this, s);
  }
  std::istream & get_line(char * s, size_t = 0){
    return std::getline(s);

};
```

It is now possible to use member function syntax to call `get_line()` (note the changed spelling for the auxiliary versions) for a `std::string` and for a C-style array of `char` type string.

The defaulted anonymous parameter allows simple conversion from code that uses a C-style string to one that uses a C++ `std::string`. This is just one example of the potential of auxiliary member functions.

We could add a similar function to the `std::string` class:

```
friend class std::string{
public:
  std::istream & get_line(std::istream & in, size_t = 0){
    return std::getline(in, *this);
  }
};
```

This means that programmers are free to view `get_line()` as a mechanism provided by either `std::string` or `std::istream`. That increases the symmetry of implementation which is implicit in such functions.

One of the advantages of using an auxiliary `private` interface is that it makes refactoring implementation code easier. The class definition does not have to be touched and so there will be no requirement for recompilation of dependent source code.

My attention has been drawn to the fact that the type names (`std::string` and `std::istream`) in these examples are in fact `typedef`s for template specializations. I propose that we should allow the use of `typedef` names for the purpose of providing auxiliary interface declarations. The proposal is intended to provide for normal classes

and for class templates as well. The following example is intended to be supported by the proposal:

```
template <typename T>
class X {
// declarations of members
};
template <typename T>
friend class X<T> {
// declarations of auxiliary members
};
```

Further Discussion

Kevlin Henney

This now looks a lot like partial classes in C# or the way that classes can be added to in Ruby, so that similarity may be worth mentioning (as well as some AOP/SOP-like concepts). Talking about prior art would seem to be valuable given some of the consequences of untried novelties that crept into C++98. Overall the syntax and concepts seem cleaner than before, although I am still unsure about the complexity this would add to working with the language for the benefits it might provide. But I am more convinced than before :-)

FG

I am happy that there is some prior art to support this proposal unfortunately I cannot personally comment on it because I am not that familiar with partial classes or with Ruby. If the proposal also adds support for AOP/SOP (Aspect/Subject Oriented Programming) then I think that is a further reason for giving it serious consideration.

Kevlin Henney
The overloading limitation becomes a problem in trying to extend a class interface, as is made apparent by having to write "`get_line`" rather than "`getline`". This is unpleasant, confusing and, assuming that one cannot overload across different auxiliary interface definitions, means that you cannot extend an existing class in a regular fashion. If my reading of this situation is correct it severely limits the usefulness of this idea. The constraint on overloading should be removed because it doesn't really solve a problem.
FG
It is intended (and I think that the text above actually specifies) that auxiliary interfaces combine so that a function name can be overloaded by declarations provided by more than one auxiliary interface declaration. What this proposal currently prohibits is overloading between the functions declared in a class definition and those declared in an auxiliary class declaration.
The constraint has the benefit that we do not need to further specify what happens if the class owner later adds a function that has the same signature as an auxiliary one. However it implies that programmers would probably need to develop idioms for naming member functions that would allow providers of auxiliary members to ensure that there would not be a conflict.

The programmer who wants to add overloads to a set of member functions has the option to declare auxiliary forwarding functions (as the `getline`/`get_line` example demonstrates.) I may be wrong, but I anticipate that acceptance of this proposal (for auxiliary interfaces) would lead to a programming style where the auxiliary interfaces were the ones that were primarily used by many end users.
Operator overloading provides a different challenge because those do, currently, overload across global/class scope. Perhaps the same dispensation should be allowed to overloading them in an auxiliary scope.

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.