Doc No: SC22/WG21/N1582 = 04-0022

Project: JTC1.22.32

Date: Thursday, February 12, 2004

Author: Francis Glassborow

email: francis@robinton.demon.co.uk

Compiler Generated Defaults

1 The Problems

This paper supersedes my earlier paper on class defaults. It narrows the target to precisely the four members of a class that can under some circumstances be compiler generated.

- Declaring a copy constructor suppresses compiler generation of both the copy constructor and the default constructor whilst declaring any non-copy constructor permits compiler generation of a copy constructor.
- The declaration of a non-copy assignment has no impact on the generation of a copy assignment.
- The need to declare a virtual destructor inhibits compiler generation of the destructor.
- Currently the ways to suppress compiler generated members is simultaneously too specialized and too extensive.
- There are good reasons for allowing compiler generation of some member functions but we need a better way to control that behavior.

2 The Proposals

1) Provide a specific mechanism to turn off all implicit member function generation. For this I propose that we enhance the grammar of class definition to allow a class to be qualified as explicit. In such a class none of the four special functions will be implicitly declared.
2) Provide a mechanism to explicitly declare a default constructor, copy constructor, copy assignment and destructor. For this purpose I propose that 'default' preceding the declaration explicitly requires the member function be compiler generated. The generation of an explicitly declared default constructor is not suppressed by the declaration of other constructors. The rules for compiler generation of the four members are as they are for the generation of the implicit versions. [12.1 para. 7 for implicit default constructor, 12.8 para. 4-8 for the copy constructor, 12.8 para. 10-14 for copy assignment, 12.4 para. 3-5,]

Example:
```
explicit class example{
    public:
        default example();
        example(int * i_ptr):val(*i_ptr){}
```

```
          default virtual ~example();
     private:
          int val;
};
// note the class is silly as such but is sufficient to
demonstrate the combined
// use of explicit and default.

class derived: public example{
     public:
// public interface members
     private:
          std::string s;
};
int main(){
     int i(12);
     example e1; // OK, uses compiler generated default
     example e2(&i); // OK uses second constructor
     example e3(e1); // ERROR no implicit copy ctor
     derived d1;    // OK
     derived d2(d1);     //ERROR, cannot generate copy ctor
     example* d_ptr = new derived;
     delete d_ptr;  // calls an implicitly generated
                    // ~derived() first which calls ~string
}
```

Discussion

Explicit declaration of any of the four member functions that would otherwise be implicitly declared is OK even in a class that is not qualified as explicit. Two common uses of this would be where you want the compiler to generate a virtual destructor, and where you want the compiler to generate a default constructor in the presence of other user declared constructors.

An explicit base class has some influence on derived classes but the explicit requirement is not in itself inherited. For example if the base class does not provide explicit copy construction and assignment then the derived class cannot generate a copy constructor or copy assignment. However the programmer can provide complete definitions of either of those though the assignment case will be problematical if the base class contains any data members. [How should the owner of a class derived from an explicit class provide for copying of private, base class data members unless the base class provide read and write access to those members through a non-private interface?]

One advantage of explicit qualification of a class is that it allows earlier diagnosis of some errors. For example the conventional hack to suppress copy semantics results in delayed diagnosis of attempts to copy instances in class scope, explicit class qualification allows immediate diagnosis.

Another advantage of this ‘explicit’ qualification is that it places information about the class right out front and rather than buried in the private interface. That a class does not support copy semantics is a public quality and should not be implied by a private declaration. This better documents the programmer's intent and provides better diagnostics than those currently available through such library mechanisms as Boost's non-copyable. On the other hand it does allow the designer of the derived class to explicitly override the non-copyable property by writing the special member copy functions for the derived class though constrained by possibly not being able to copy the base class members.

Note that an explicit class that does not have a declared destructor cannot reside on the stack or as a static object because it is not destructable. However it could be constructed dynamically. The burden for clean-up then remains with the class user (possibly through explicit destruction of sub-objects followed by a call to `operator delete`).

Alan Stokes has raised an issue of what:

```
default virtual ~mytype = 0;
```

should mean. I think it is consistent with the proposal to require this to be a compiler generated default constructor which is also a pure virtual (i.e. makes the class an abstract base class). An attempt by the class implementor to define that destructor is ill-formed (redefinition).

Changes to the Working Paper

These are not provided at this point. It seems more important to agree the mechanism.

The actual changes will require four things:

1) Added material to chapter 12 to describe the explicit class syntax and the default declaration syntax.

2) Modification to all the above-specified paragraphs where the nature of implicitly generated functions are specified to allow both for not implicitly declaring the special member functions.

3) Modification to those paragraphs to convert 'implicitly-declared' to 'implicitly-declared or explicitly default declared'.

4) Adding relevant material re using implicitly declared members in derived classes (requirement that the base class has either implicitly or explicitly provided them and that the implicit provision has not been disabled by the base class being an explicit class.