Doc No: SC22/WG21/ N1702 04-0142

Project: JTC1.22.32

Date: Wednesday, September 08, 2004

Author: Francis Glassborow & Lois Goldthwaite

email: francis@robinton.demon.co.uk

explicit class and default definitions
revision of SC22/WG21/N1582 = 04-0022

1 The Problems being addressed

Some common member functions of a C++ class will be automatically generated by the compiler if the programmer fails to mention them in code. The ones most commonly discussed in the literature and in training material are:

- Default constructor
- Copy constructor
- Copy assignment operator =
- Destructor

Clause 12p1 says that code for these member functions and operators is generated only if actually used in the program but adds that programs may explicitly refer to an implicitly-declared function, such as by taking its address or forming a pointer to such a member function. The generated special member functions are `inline public` members of their class (12.1p5, 12.8p5, et al.)

Some other functions which are seldom discussed but are also generated are:

- operator & (address-of)
- operator , (comma)
- operator -> (when a pointer is used)
- operator * (to dereference a pointer)
- operator ->* (to dereference pointer-to-member)

There are also these built-in operators that take a class or enumeration type operand but which may not be overridden by the programmer:

- operator .
- operator .*
- operator ::

Also there are additional operators (arithmetic, logical, function call, subscripting, and new/delete) which can be defined by the programmer but are not automatically generated by the compiler for user-defined types. These operators are not further discussed here.

The implicitly generated code for the four principal special member functions sometimes has the wrong semantics – the most common example is shallow copy of pointers for allocated memory – and therefore the programmer must be careful to define what semantics are needed. Implicit generation also gives rise to the following syntax-related problems:

- Declaring a copy constructor suppresses compiler generation of both the copy constructor and the default constructor whereas declaring any non-copy constructor permits compiler generation of a copy constructor. Note that because a template constructor is explicitly not a copy constructor, the existence of such a template does not prevent the generation of a copy constructor.
- The declaration of a non-copy assignment has no impact on the generation of a copy assignment. Curiously different from the rule for constructors.
- In the absence of a declared destructor the one generated by the compiler will be non-virtual. If a destructor needs to be virtual it must be declared and defined by the programmer.
- Currently the recommended ways to suppress compiler generated members are 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 implicit special member function generation. For this I propose that we enhance the grammar (9 para 1) of class-key to allow a class/struct to be qualified as `explicit`.

```
explicit[opt] class
explicit[opt] struct
union.
```

   In such a class none of the four special member functions that can otherwise have compiler-generated definitions will be implicitly declared. Note that it is my intention that these members are not declared as well as not defined by the compiler so that an attempt to use them will result in a compile time diagnostic. This will require redrafting of 12 para 1 so as to exclude implicit declaration of the special member functions for classes that are declared as explicit.

2) Provide a mechanism to explicitly require compiler generation of the definition of a special member function. If it is not possible to generate a definition of a special member function in response to an explicit generation statement the code is ill-formed. My proposed syntax is:

```
declarator {default} // note, no semicolons
```

   So that, for example, the destructor for a class `mytype` would be defined in class by one of (we should choose just one for the Standard):

```
~mytype(){default}
```

The reason for this choice is to make it clear that this is a definition, not just a declaration. Such definitions shall be available exactly as any other definition. Though they would likely be provided in class (in which case they are inline definitions) they can also be provided in a normal implementation file, in which case they are not inline. Note that this allows programmers to provide non-inline default definitions of the four special member functions. This might be useful where a programmer wanted to use instrumented versions for debugging and profiling but wanted to use the compiler generated default in production code.

The generation of an explicitly defined default constructor is not suppressed by the declaration of other constructors. The rules for compiler generation of the special 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:
          example(){default}
          example(int * i_ptr):val(*i_ptr){}
          virtual ~example(){default};
     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
                    // ~example which calls ~string
}
```

3) The issue of how a pure virtual default destructor should be provided clarifies that explicit defaults are definitions and can be used as implementations. For example:

```
class abc {
public:
     abc();
     abc(int);
     virtual ~abc() = 0;
private:
     // details
};
inline abc::~abc(){default}
```

The definition could be moved to a separate implementation file. If the required function definition cannot be compiler generated at the definition point the code is ill-formed. This demonstrates that the availability of explicit compiler generation of the relevant member is useful even in classes that have not been declared as explicit.

Note that out-of-class default definitions are not inline unless defined with the `inline` keyword.

Discussion

Explicit declaration of any of the member functions that would otherwise be implicitly declared is supported 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. [Lois: 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? Francis: programmers ingenuity knows no bounds.] The base class could be default-initialized, or initialized with one of its regular constructors, by including it in the member initializer list. Actually this is what happens anyway, if you forget to invoke the base class copy ctor yourself. Look at this program:

```
#include <iostream>
using namespace std;

class example
{
   public:
      int i;
      example() : i(99) { }
      example( example const & other ) : i( other.i ) { }
```

```
      virtual ~example()  { }
};

class derived : public example
{
      // implicit copy ctor calls base class copy ctor
};

class descendant : public example
{
   public:
      int j;
        descendant() : j(21)  { }
       // explicit copy ctor calls base class default ctor
       //  -- if you want to copy base class, include it in
       // mem initializer list
      descendant( descendant const & other ) : j( other.j ) { }

};

int main()
{
  derived d1;
  d1.i = 55;
  derived d2( d1 );
  cout << d2.i << endl; // 55

  descendant d3;
  d3.i = 77;
  descendant d4( d3);
  cout << d4.i << endl;  // 99 -- d3.i wasn't copied!!

   return 0;
}
```

Actually I think I can see a use case for a base class that could never be copy constructed, but only default initialized. Suppose the base class maintains some sort of counter to keep track of how many times the object is invoked, or copied, or whatever. In such a case, you wouldn't want to duplicate the secret data for the original's base class when creating another instance from it. A concrete example case might be something following the Factory pattern, which is used to create instances of various types. Keeping track of how many times each individual Factory has been used could help with load balancing.

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:

```
class do_not_copy_me {
     do_not_copy_me( do_not_copy_me const& );
          // private & unimplemented
public:
     //
```

```
};
```

results in delaying diagnosis of attempts to copy instances in class scope until link time, whereas `explicit` class qualification allows immediate compile-time diagnosis at the point of error.

Another advantage of this ‘explicit’ qualification is that it places information about the copy semantics of a class right out front 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. explicit qualification will better document the programmer's intent and provide 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. [Lois: though as I said above, the class designer may have a reason for not allowing the base class members to be copied]

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 destructible. 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`).

I remain uncertain about the operator & and operator, . I can imagine that requiring explicit declaration of the former could be useful. However operator, seems to be exceptional because it is currently implicitly overloaded for all conceivable combinations of left and right operands. At the same time we allow explicit overloading of the operator, a feature that is widely used in some problem domains. I think we need to carefully consider whether there is any utility in suppressing these two operators and the potential impact on future code if we include these operators in those required to be made explicit in an explicit class. In the case of operator& there is a single implied member function but in the case of operator, there is, effectively an implied two parameter function template generated by the compiler.

Changes to the Working Paper

The actual changes to the WP will require four things:

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

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

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

4) Adding relevant material re using implicitly declared members in derived classes one of whose base classes is an explicit qualified class.

We intend to present preliminary wording at the Redmond meeting.