Doc No: SC22/WG21/ N1717 04-0157

Project: Programming Language C++

Date: Friday, November 5, 2004

Author: Francis Glassborow & Lois Goldthwaite

email: francis@robinton.demon.co.uk

explicit class and default definitions
revision of SC22/WG21/N1582 = 04-0022 and SC22/WG21/ N1702 04-0142

1 The Problems being addressed

Certain member functions of a C++ class will be automatically generated by the compiler if the programmer fails to mention them in code. These "special member functions" are discussed in Clause 12 of the Standard:

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

Clause 12p1 says that code for these implicitly declared 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 it. The generated special member functions are `inline public` members of their class.

Some other functions are also usable in programs even if not declared by the programmer:

- operator & (address-of)
- operator , (comma)

and also – since declaration of a user-defined type T also implies existence of a T* –

- operator -> (class member access)
- operator * (indirection)
- operator ->* (pointer-to-member)

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

- operator .
- operator .*
- operator ::

There are additional operators (arithmetic, logical, function call, subscripting, and new/delete) that can be defined by the programmer but are neither declared nor defined 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. This is 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 we propose that we enhance the grammar (9p1) to allow a *class-specifier* to be qualified as `explicit`.

   class-specifier:
   class-head explicit[1] *opt* *{ member-specification*opt *}*

   This would permit usage like this:

   ```
   class  foo explicit { … };
   struct bar explicit { … };
   union baz explicit { … };
   class fee: public fum explicit { ... };
   ```

[1] In the Changes to the Working Paper section, we propose an addition to the grammar grandly called *postfix-decl-specifier*, whose only production at the present time is `explicit`. This is a far-sighted provision to allow for possible future modifiers to be tagged on the end of declarations; hypothetical examples include such words as `pure` and `interface`.

`explicit` does not become part of the class type and therefore shall not be used with forward declarations and elaborated type specifiers. The `explicit`-ness of a class definition is an implementation detail.

`explicit` unions are allowed as a matter of grammatical consistency with the other class-keys; declaring a union `explicit` is not expected to be as commonly useful as with class and struct, if only because unions themselves are less widely used. Although a union can have member functions, including constructors and destructors, it cannot contain as a member any object that has a non-trivial constructor, copy constructor, destructor, or copy assignment operator, nor can it be a base or derived class (9.5p1). It follows logically that the implicit special member functions would themselves be trivial – even basic default value initialization is tricky if the subobject's type cannot be determined. Nevertheless it might be useful to inhibit pass-by-value or stack allocation for an object of such type, by declaring it `explicit`.

Furthermore, as `explicit` qualification absolves the compiler from any responsibility for generating implicit functions for a union and requires the programmer to define them if needed, it might in future allow the restrictions to be loosened on which object types can be made members of a union. If member objects do not comply with the current rules on union membership, it would not be possible explicitly to declare and default-define the special member functions, however. This extension is not part of the current proposal.

In such an `explicit` qualified class none of the four special member functions that can otherwise have compiler-generated definitions will be implicitly declared. Note that it is the 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 clause 12 so as to exclude implicit declaration of the special member functions for classes that are declared as `explicit`. See the section "Changes to the Working Paper" below.

2) Provide a mechanism for explicit invocation of compiler generation of the definition of a special member function. Our proposed syntax is:

```
declarator {default}
```

So that, for example, the destructor for a class `mytype` would be defined in class by:

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

[Note that no semicolon is necessary, but one is allowed to follow `default` without error since most programmers will add one automatically out of habit.]

Two alternate syntaxes were also considered:

```
declarator  default;

declarator = default;
```

The preferred syntax was chosen as the clearest indication that this is a definition, not just a declaration. On the other hand, an argument can be made in favor of "= default" that it parallels the "= 0" syntax for declaring pure virtual functions.

Such definitions shall be available exactly as if implicitly defined according to 12.1p6 (default constructor), 12.8p8 (copy constructor), 12.8p13 (copy assignment operator), and 12.4p6 (destructor). However, placing the explicit declaration in the protected or private section of the class would affect accessibility. An explicit declaration of a special member function must match the signature of some form of one that would be implicitly declared in order to qualify as a special member function that can be default-defined if desired.

Though they would likely be provided inside the class definition (in which case they are `inline`) 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, or wanted to avoid all `inline` code to ensure future binary compatibility.

<u>Non-inline default definitions of the special member functions shall only be available if the relevant special functions are explicitly declared in the class definition</u>. This is to avoid ODR violations caused by implicit declarations combined with non-`inline` default definitions in some translation unit.

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, diagnostic required.

The generation of an explicitly declared and default-defined default constructor is not suppressed by the declaration of other constructors. Even a class which is not qualified as `explicit` may use this mechanism to generate implementations of special member functions.

Example:
```
class example explicit{
      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 ~example which first redirects
                          // to an implicitly generated ~derived
                          // 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.

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

Discussion

1. 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.

2. 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. A call to some constructor for the base class can be included in the member initializer list of the derived class, or the programmer could simply allow the default constructor of the base class to execute before the body of the derived class constructor is entered (which is what happens currently if a user-written copy constructor or copy assignment for a derived class omits to call a specific base class constructor).

That the base class subobject is default-initialized if the programmer omits to mention it is obvious to all experienced C++ programmers. But to belabor the obviousness of it, here is an example program demonstrating what happens, and therefore why the non-existence of a base class copy constructor need not prevent a derived class from being copied:

```
#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;
}
```

[There is an apocryphal tale about the Cambridge mathematician G H Hardy, who was giving a lecture and said something was obvious... then paused... left the hall, returned fifteen minutes later, reassured the audience that it was indeed obvious and continued the lecture.]

3. An example use case for a base class that could never be copy constructed, but only default initialized, might be a base class that maintains a counter to keep track of how many times the object executes a certain function. A concrete example 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.

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, so logically all counters should be initialized to 0 for a new instance, no matter how it is constructed.

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 {
      // private & unimplemented
      do_not_copy_me( do_not_copy_me const& );

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.

4. 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 inaccessible 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 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`).

5. Special member functions which have been explicitly declared and default-defined are never trivial. Therefore an `explicit` class can never be a POD, even if its special member functions are default-defined. (The justification for this restriction is that the semantics of a class should not change if its `inline` default-defined functions are moved out of line.)

6. A question still under discussion is whether an `explicit` class can be an aggregate as defined in 8.5.1p1: "An aggregate is an array or a class (clause 9) with no user-declared constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3)." If it is deemed to comply with these requirements, then an object could be initialized with a brace-enclosed list of values instead of a constructor.

7. An explicitly declared and default-defined constructor can be declared an explicit constructor:

```
class foo {

      // stuff
public:
		foo( int j, int k );
		foo() { default } // would otherwise
		                          // be suppressed
	};
```

8. If a class template is qualified as `explicit`, then implicit special member function declarations will be suppressed in all instantiations of that template:

```
	template< class T >
	class furble explicit{
		// no special member functions for any furble<T>
	};
```

9. A specialization of a non-`explicit` class template can be declared as `explicit`. If an `explicit` class template is specialized, then that specialization will be `explicit` only if it is also declared as such.

10. An object of an `explicit` class can be used as a member subobject of another class or union. However, if it lacks a default constructor or copy constructor, that will affect compiler generation of the special member functions for the containing class.

11. If move semantics are adopted, an explicit class might be made movable even if not copyable.

12. An explicit class without a user-declared copy constructor is deemed to have no accessible copy constructor, even if it has a standard conversion to a base class which does have a copy constructor defined. This may require some reinterpretation of 12.8p10: "Because a copy assignment operator is implicitly declared for a class if not declared by the user, a base class copy assignment operator is always hidden by the copy assignment operator of a derived class." Standardese for a mechanism to hide the base class copy assignment operator even in the absence of a derived class copy assignment operator needs to be found.

```
	base b { };
	class derived : public base explicit{ };
	base b;
	derived d;
	b = d;  // OK, but slices
	d = b;  // error
```

13. The issue of *exception-specifications* for the special member functions of `explicit` classes requires special scrutiny. Implicitly declared member functions have an exception specification. 15.4p13 details what that exception specification must be: "If `f` is an implicitly declared default constructor, copy constructor, destructor, or copy assignment operator, its implicit exception-specification specifies the type-id T if and only if T is allowed by the exception-specification of a function directly invoked by `f'`s implicit definition; `f` shall allow all exceptions if any function it directly invokes allows all exceptions, and `f` shall allow no exceptions if every function it directly invokes allows no exceptions."

Furthermore 15.4p3 says, "If any declaration of a function has an *exception-specification*, all declarations, including the definition and an explicit specialization, of that function shall have an *exception-specification* with the same set of *type-ids*… If an *exception-specification* is specified in an explicit instantiation directive, it shall have the same set of *type-ids* as other declarations of that function. A diagnostic is required only if the sets of *type-ids* are different within a single translation unit."

Should a user declaring special member functions be required to add an *exception-specification* to all declarations? This seems to impose an unreasonable burden on the programmer. Can a default definition be stretched to generate an *exception-specification* for a user-declared function automagically? This seems to impose an unreasonable burden on the compiler. It seems more reasonable to say that a user-declared function, even if default-defined, has the exception-specification declared by the programmer, if any.

14. There is also an issue when overriding virtual functions with *exception-specifications*. For purposes of this paper, the issue mainly affects destructors, since that is the special member function most likely to be virtual. (The assignment operator can be made virtual but rarely is.) An overriding function in a derived class must have an *exception-specification* at least as restrictive as the overridden function in its base class (15.4p4). When multiple inheritance is involved, this can get tricky (15.4p13). However, this does not appear to be a problem unique to explicit classes.

15. To return briefly to the topic of operators that can be overloaded by the programmer, this paper proposes that they not be suppressed by an explicit qualification. Although operators '&', ',', and '->' can be used without programmer-supplied semantics, and thus might be considered to be implicitly declared in some sense, 13.3.1.2, 5.3.1, and 5.18 state clearly that such usage invokes the built-in operators rather than implicitly generated member functions.

The address-of operator has such obviously useful semantics that suppressing it would make C++ harder, not easier, to learn. Besides, many definitions of copy constructors rely on it to check for identity assignment, so it would nearly always have to be reinstated anyway. Operator comma is not so obviously useful in ordinary programming, but the only putative benefit of suppressing it routinely might be to check for typos that should have been semi-colons instead of commas. Any benefit would be insignificant compared with the trouble of changing the world's compilers. In the same vein, when novices are learning how to access a class member through a pointer, they don't need the additional complication of defining operator ->.

Summary

This paper proposes a change to C++ grammar to allow the keyword `explicit` to be included in a *class-specifier*. The effect of qualifying a class type definition in this way would be to suppress the implicit declaration and definition of the four special member functions discussed in clause 12 of the Standard. A second change in the grammar would allow the programmer to un-suppress the implicit definitions on a selective basis. This could also be invoked for classes that are not `explicit`-qualified, but where the special member functions have been suppressed under circumstances specified in the present

Standard, or even where the desire is simply to document that the compiler-generated default definitions are suitable.

No new keywords are added to the language, since `explicit` and `default` are already reserved words which cannot appear in the context of class and member function definitions.

The benefits of the suggested change include:

- Code would better document the programmer's intent.
- Because the new rules would be more straightforward, they would make C++ easier to learn and teach. Current means of suppressing or enabling member functions are complicated and difficult to understand.
- Some coding standards forbid the use of implicit member functions, even if their semantics are correct, and require the programmer explicitly to define all functions. And some functions must be explicitly defined in any case because they have been suppressed by other declarations. This raises the opportunity to create errors in the additional code (such as by forgetting to invoke a base class copy constructor in the derived class). Being able to invoke explicit generation of these common functions will result in programs that are shorter and more readable.
- Another motivation for defining all member functions out of line is to preserve binary compatibility. This mechanism allows them to be generated out of line, achieving the same benefit of shorter, more readable, code with fewer errors.
- Apart from the out-of-line generation of implicit functions – which might be achievable now by taking the address of such functions – this paper does not propose any detectable change to executable C++ programs. The grammar changes should not break any existing code because the two keywords are already reserved and are merely being used in new contexts.

Changes to the Working Paper

The actual changes to the WP will require four things:

1) Changed grammar in clause 9 to allow class definitions to be qualified as `explicit`. [We have added a new category of *postfix-decl-specifier*, to indicate a modifier that follows the normal declaration. Currently `explicit` is the only member of this group, but it could later be extended to cover `pure` and `nothrow` and …
2) Added material to chapter 12 to describe the explicit class syntax and the default definition syntax. [We have placed all the new stuff into a separate section, rather than tinkering with a sentence here and there.]
3) Added grammar to 8.4 to allow `default` as a definition of a function body.
4) Possible changes to deal with 12 above.

9 paragraph 2 and A.8 (page 790 of WP):

From:

class-specifier:

class-head { member-specification $_{opt}$ }

To:

postfix-decl-specifier:

`explicit`

class-specifier:

class-head postfix-decl-specifier $_{opt}$ { member-specification $_{opt}$ }

8.4 paragraph 1 and A.7 (page 789 of WP)

From:

function-body:

compound-statement

To:

function-body:

compound-statement
`{ default ;`$_{opt}$ `}`

[The authors are aware that an additional constraint that `default` can only be used with special member functions will be required somewhere but are uncertain as to where that should go.]

12 paragraph 1:

From:
The implementation will implicitly declare these member functions for a class type when the program does not explicitly declare them, except as noted in 12.1. The implementation will implicitly define them if they are used, as specified in 12.1, 12.4 and 12.8.

To:
The implementation will implicitly declare these member functions for a class type when the program does not explicitly declare them, except as noted in 12.1 and 12.9. The implementation will implicitly define them if they are used, as specified in 12.1, 12.4, 12.8 and 12.9.

Add:

12.9 Explicit classes

Implicit declaration and definition of special member functions can be suppressed by qualifying the class definition with the *postfix-decl-specifier* `explicit`. *[Example:*

```
class E explicit {
                          // has no constructor, copy constructor,
                                        // assignment operator or destructor
        };
```

-- *end example.]*

However, these functions can be explicitly declared. Functions so declared obey the usual access rules (clause 11). A program is ill-formed if one of these special member functions of an explicit class is used without being declared.

An explicitly-declared special member function is never trivial.

An explicitly-declared special member function of any class (not restricted to those qualified as `explicit`) may be implicitly defined by the compiler, if the body of the definition consists of `{default}`. Such a definition is called a *default-definition*.

A default-defined special member function is implicitly defined at the point of the definition, according to the rules in 12.1[p6], 12.4[p6], and 12.8[p8 and p13], except that such functions are not automatically `inline` or `public`. If it is not possible to generate a default-definition of an explicitly declared special member function at the point of definition the code is ill-formed, diagnostic required.

A default-defined special member function is inline if it is either declared as `inline` or the definition is provided in the class definition. *[Example:*

```
struct A {

        A();

        A(const A&){default}  // inline explicit default definition

        virtual ~A();

};

inline A::A(){default} // inline explicit default definition

A::~A() {default} // out of line explicit default definition
```

-- *end example.]*

An explicitly-declared special member function has an *exception-specification* according with its declaration, even if an implicitly-declared function might have a different *exception-specification*.

An object of an `explicit` class can be used as a member subobject of another class or union. However, if its special member functions are missing, implicit or default definition of the special member functions for the containing class may be ill-formed.

A class derived from an explicit base class is not explicit unless it is so qualified, but if special member functions are missing from the base class definition, implicit or default definition of the derived class's member functions may be ill-formed.

If a class template is qualified as `explicit`, then implicit special member function declarations will be suppressed in all instantiations of that template:

```
        template< class T >
        class furble explicit {
                // no special member functions for any furble<T>
        };
```

A specialization or partial specialization of an explicit class template is not explicit unless it is so qualified. A specialization or partial specialization of a non-explicit class template may be qualified as explicit.

[The authors are uncertain as how to phrase (in stnadardese) the requirements of point 12 of the Discussion, above.]