Implicitly-Deleted Special Member Functions

Author: Douglas Gregor, Apple
Document number: N2924=09-0114
Date: 2009-07-17
Project: Programming Language C++, Core Working Group
Reply-to: Douglas Gregor <doug.gregor@gmail.com>

Introduction

Concepts [2] introduced the notion of implicitly-deleted special member functions into the C++0x working paper. With the removal of concepts, we would like to retain this non-concepts-specific change. The essence of this change is that an implicitly-declared special member function (default constructor, copy constructor, destructor, copy assignment operator) will be implicitly defined as deleted (with `= delete`) if its definition would be ill-formed. For example:

```
struct HasReference {
  int &ref;
  // implicitly declares:
  // HasReference() = delete;
  // HasReference(const HasReference&);
  // HasReference& operator=(const HasReference&);
  //  HasReference();
};
```

The intent of this change is to ensure that template argument deduction fails when it finds an implicitly-declared special member function whose definition would fail to compile. The problem [1] came up in the context of concepts, where the type `std::pair<const int, int>` was getting an implicitly-declared copy-assignment operator, such that it met the requirements of the `CopyAssignable` concept. However, the definition of the copy-assignment operator is ill-formed (due to the `const` member), causing unexpected failures. By making this copy-assignment operator deleted, it can no longer be used to satisfy the requirements of the `CopyAssignable` concept. Even without concepts, this issue still comes up with the extended SFINAE rules.

This proposal is identical to the non-concepts changes made by N2773 and the most recent working paper containing concepts (N2857).

Editorial note: Rather than reverting to pre-concepts wording and then applying the proposed wording in this document, one could instead remove the concepts changes in [class.conv.fct] and [class.inhctor], which will have the same effect for clause [special].

Chapter 12 Special member functions [special]

12.1 Constructors [class.ctor]

5 A *default* constructor for a class `X` is a constructor of class `X` that can be called without an argument. If there is no user-declared constructor for class `X`, a constructor having no parameters is implicitly declared. An implicitly-declared default constructor is an `inline public` member of its class. ~~For a union-like class that has a variant member with a non-trivial default constructor, an implicitly-declared default constructor is defined as deleted ([dcl.fct.def]).~~ A default constructor is *trivial* if it is implicitly-declared and if:

- its class has no virtual functions ([class.virtual]) and no virtual base classes ([class.mi]), and
- all the direct base classes of its class have trivial default constructors, and
- for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

An implicitly-declared default constructor for class `X` is deleted if:

- `X` is a union-like class that has a variant member with a non-trivial default constructor,
- any non-static data member is of reference type,
- any non-static data member of const-qualified type (or array thereof) does not have a user-provided default constructor, or
- any non-static data member or direct or virtual base class has class type `M` (or array thereof) and `M` has no default constructor, or if overload resolution ([over.match]) as applied to `M`'s default constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared default constructor.

7 A non-user-provided default constructor for a class is *implicitly defined* when it is used ([basic.def.odr]) to create an object of its class type ([intro.object]). The implicitly-defined or explicitly-defaulted default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with an empty *mem-initializer-list* ([class.base.init]) and an empty function body. If the implicitly-defined copy constructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor ([decl.constexpr]), the implicitly-defined default constructor is constexpr. Before the non-user-provided default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared default constructor has an *exception-specification* ([except.spec]). An explicitly-defaulted definition has no implicit *exception-specification*. — *end note*]

12.4 Destructors [class.dtor]

3 If a class has no user-declared destructor, a destructor is declared implicitly. An implicitly-declared destructor is an `inline public` member of its class. ~~If the class is a union-like class that has a variant member with a non-trivial destructor, an implicitly-declared destructor is defined as delected ([dcl.fct.def]).~~ A destructor is *trivial* if it is implicitly-declared and if:

— all of the direct base classes of its class have trivial destructors and

— for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

An implicitly-declared destructor for a class `X` is deleted if:

— `X` is a union-like class that has a variant member with a non-trivial destructor,

— any of the non-static data members has class type `M` (or array thereof) and `M` has an deleted destructor or a destructor that is inaccessible from the implicitly-declared destructor, or

— any direct or virtual base class has a deleted destructor or a destructor that is inaccessible from the implicitly-declared destructor.

5 An implicitly-declared destructor is *implicitly defined* when it is used to destroy an object of its class type ([basic.stc]). A program is ill-formed ~~if the class for which a destructor is implicitly defined has:~~if the implicitly-defined destructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

— ~~a non-static data member of class type (or array thereof) with an inaccessible destructor, or~~

— ~~a base class with an inaccessible destructor.~~

Before the implicitly-declared destructor for a class is implicitly defined, all the implicitly-declared destructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared destructor has an *exception-specification* ([except.spec]). — *end note*]

12.8 Copying class objects [class.copy]

4 If the class definition does not explicitly declare a copy constructor, one is declared *implicitly*. ~~If the class is a union-like class that has a variant member with a non-trivial copy constructor, an implicitly-declared copy constructor is defined as deleted ([dcl.fct.def]).~~ Thus, for the class definition

```
struct X {
        X(const X&, int);
};
```

a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of `X`'s copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

5 The implicitly-declared copy constructor for a class `X` will have the form

```
X::X(const X&)
```

if

— each direct or virtual base class `B` of `X` has a copy constructor whose first parameter is of type `const B&` or `const volatile B&`, and

— for all the non-static data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy constructor whose first parameter is of type `const M&` or `const volatile M&`.[1)]

Otherwise, the implicitly declared copy constructor will have the form

```
X::X(X&)
```

An implicitly-declared copy constructor is an `inline public` member of its class. An implicitly-declared copy constructor for a class `X` is deleted if `X` has:

— a variant member with a non-trivial copy constructor and `X` is a union-like class,

— a non-static data member of class type `M` (or array thereof) that cannot be copied because overload resolution ([over.match]), as applied to `M`'s copy constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy constructor, or

— a direct or virtual base class `B` that cannot be copied because overload resolution ([over.match]), as applied to `B`'s copy constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy constructor.

7 A non-user-provided copy constructor is *implicitly defined* if it is used to initialize an object of its class type from a copy of an object of its class type or of a class type derived from its class type[2)]. [*Note:* the copy constructor is implicitly defined even if the implementation elided its use ([class.temporary]). — *end note*] A program is ill-formed ~~if the class for which a copy constructor is implicitly defined or explicitly defaulted has:~~if the implicitly-defined copy constructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

— ~~a non-static data member of class type (or array thereof) with an inaccessible or ambiguous copy constructor, or~~

— ~~a base class with an inaccessible or ambiguous copy constructor.~~

Before the non-user-provided copy constructor for a class is implicitly defined, all non-user-provided copy constructors for its direct and virtual base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared copy constructor has an *exception-specification* ([except.spec]). An explicitly-defaulted definitions has no implicit *exception-specifion*. — *end note*]

10 If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. ~~If the class is a union-like class that has a variant member with a non-trivial copy assignment operator, an implicitly-declared copy assignment operator is defined as deleted ([dcl.fct.def]).~~ The implicitly-declared copy assignment operator for a class `X` will have the form

```
X& X::operator=(const X&)
```

if

— each direct base class `B` of `X` has a copy assignment operator whose parameter is of type `const B&`, `const volatile B&` or `B`, and

[1)] This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a `volatile` lvalue; see [diff.special].

[2)] See [dcl.init] for more details on direct and copy initialization.

— for all the non-static data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy assignment operator whose parameter is of type `const M&`, `const volatile M&` or `M`.[3)]

Otherwise, the implicitly declared copy assignment operator will have the form

```
X& X::operator=(X&)
```

The implicitly-declared copy assignment operator for class `X` has the return type `X&`; it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy assignment operator is an `inline public` member of its class. An implicitly-declared copy assignment operator for class `X` is deleted if `X` has:

— a variant member with a non-trivial copy assignment operator and `X` is a union-like class,

— a non-static data member of `const` non-class type (or array thereof), or

— a non-static data member of reference type, or

— a non-static data member of class type `M` (or array thereof) that cannot be copied because overload resolution ([over.match]), as applied to `M`'s copy assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy assignment operator, or

— a direct or virtual base class `B` that cannot be copied because overload resolution ([over.match]), as applied to `B`'s copy assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy assignment operator.

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 ([over.ass]). A *using-declaration* ([namespace.udecl]) that brings in from a base class an assignment operator with a parameter type that could be that of a copy-assignment operator for the derived class is not considered an explicit declaration of a copy-assignment operator and does not suppress the implicit declaration of the derived class copy-assignment operator; the operator introduced by the *using-declaration* is hidden by the implicitly-declared copy-assignment operator in the derived class.

12 A non-user-provided copy assignment operator is *implicitly defined* when an object of its class type is assigned a value of its class type or a value of a class type derived from its class type. A program is ill-formed ~~if the class for which a copy assignment operator is implicitly defined has:~~if the implicitly-defined copy assignment operator is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

— ~~a non-static data member of `const` type, or~~

— ~~a non-static data member of reference type, or~~

— ~~a non-static data member of class type (or array thereof) with an inaccessible copy assignment operator, or~~

— ~~a base class with an inaccessible copy assignment operator.~~

Before the non-user-provided copy assignment operator for a class is implicitly defined, all non-user-provided copy assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared copy assignment operator has an *exception-specification* ([except.spec]). An explicitly-defaulted definition has no implicit *exception-specification*. — *end note*]

[3)] This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a `volatile` lvalue; see [diff.special].

Bibliography

[1] D. Gregor. Implicit assignments and construction. http://conceptgcc.wordpress.com/2006/05/04/implicit-assignments-and-construction/, May 2006.

[2] D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Proposed wording for concepts (revision 9). Technical Report N2773=08-0283, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2008.