X3J16/96-0085
WG21/N0903

Exception-Specifications of Implicitly Declared Functions

R. Michael Anderson
Edison Design Group
rma@edg.com

March 29, 1996

The Issue

When a constructor, destructor, or copy assignment operator is implicitly declared, what is its exception-specification? For example,

```
struct A { A() throw(X); }
struct B { B() throw(X,Y); };
struct D : public A, public B { };
```

Here the default constructor `D::D()` will be implicitly declared — but the Working Paper does not specify what its exception-specification should be. Several possibilities have been mentioned on the reflector:

1. There should be no exception-specification — the generated function will throw anything.

   ⇒ implicit declaration is: `D::D();`

2. The exception-specification is the *union* of the exception-specifications of corresponding functions of base classes and members.

   ⇒ implicit declaration is: `D::D() throw(X,Y);`

3. The exception-specification is the *intersection* of the exception-specifications of corresponding functions of base classes and members.

   ⇒ implicit declaration is: `D::D() throw(X);`

4. The exception-specification will throw nothing.

   ⇒ implicit declaration is: `D::D() throw();`

A Solution

The suggestion that makes most sense to me is option 2. In other words, an implicitly declared function `f` will be implicitly specified to throw a given exception `T` if and only if `T` is among the exceptions that will be thrown by the functions directly invoked when `f` is implicitly defined. The rationale is common sense: if a function `f` calls several other functions $\mathtt{f}_1$, $\mathtt{f}_2$, $\ldots\mathtt{f}_n$ and if `f` does not itself catch any exceptions the latter might throw, `f` may be assumed to (re)throw anything $\mathtt{f}_1$ or $\mathtt{f}_2$ or $\mathtt{f}_3$, etc., might throw.

An objection is that, if the implicitly declared function is a virtual destructor, the common-sense approach can result in a violation of a constraint in WP 15.4 [except.spec] para 2:

> If a virtual function has an exception-specification, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall have an exception-specification at least as restrictive as that in the base class.

In other words, this would seem to require that the exception-specification on an implicitly declared virtual destructor be (at most) the *intersection* (and certainly not the *union*) of the exception-specifications of the destructors of the base classes. For example:

```
struct A { virtual ~A() throw(X); };
struct B { virtual ~B() throw(X,Y); };
class D : public A, public B { };
```

By option 2 the implicit declaration would be `D::~D() throw(X,Y)`, but by the constraint quoted above it must be either `D::~D() throw(X)` or `D::~D() throw()`.

My proposal is that this example be treated as an error. If a virtual destructor is implicitly declared, then (1) its implicit exception-specification shall be the union of the exception-specifications of destructors from base classes and members (in accord with option 2); and (2) if the resulting exception-specification violates the constraint in 15.4 para 2, the program is ill-formed.

Wording for the Working Paper

The Working Paper should be changed by adding the following paragraph to 15.4 [except.spec]:

> An implicitly declared function shall have an exception-specification. If `f` is an implicitly declared default constructor, copy constructor, destructor, or copy assignment operator, it is implicitly specified to throw exceptions of type `T` if and only if `T` belongs to the exception-specification of a function directly invoked when `f` is implicitly defined; `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. [Example:
>
> ```
> struct A {
>   A();
>   A(const A&) throw();
>   ~A() throw(X);
> }
> struct B {
>   B() throw();
>   B(const B&) throw();
>   ~B() throw(X,Y);
> };
> struct D : public A, public B {
>   // Implicit declaration of D::D();
>   // Implicit declaration of D::D(const D&) throw();
>   // Implicit declaration of D::~D() throw (X,Y);
> };
> ```
>
> Furthermore, if `A::~A()` or `B::~B()` were virtual, `D::~D()` would be virtual as well, but since its exception-specification would not be as restrictive as that of `A::~A()`, the program would be ill-formed. —end example]