Proposed Resolutions for Core Language Issues 6, 14, 20, 40, and 89

J16/99-0005 = WG21 N1182
William M. Miller (wmm1@flash.net)
February 23, 1999

ISSUE 6

This issue deals with describing conditions that allow copy operations to be eliminated by a conforming implementation. 12.8¶15 currently describes two such situations. The Committee agreed that a third such optimization would also be desirable: suppression of the copy from an argument to a non-reference parameter in a call to an inline function. However, wording to describe when such an elision would be allowed has not been produced.

Analysis

The "as-if" rule (1.9¶1) allows an implementation to eliminate operations if it can be determined that the observable behavior of the program will not be affected by the optimization. However, because copy constructors and destructors can have side effects that do alter the program's observable behavior, copy operations cannot in general be elided under the as-if rule alone. 12.8¶15 relaxes the requirements of the as-if rule by allowing an implementation to optimize away certain copy operations, even if side effects in the no-longer-called copy constructor and destructor would have produced different observable behavior. Nevertheless, apart from the absence of these side effects, the remaining semantic constraints must be honored.

For the situations currently described in 12.8¶15, these constraints boil down to the requirement that the copy constructor be accessible and unambiguous. In the function parameter optimization, however, additional constraints apply. In particular, none of the operations of the function can be allowed to change the value of the object passed as an argument. Furthermore, changes to a volatile object passed as an argument must not affect the value of the parameter during the execution of the function.

A couple of considerations regarding modification of the argument (pointed out by Erwin Unruh in reflector email) are worthy of special mention. First, it is not only assignment operations in the function itself that must be considered as possibly changing the argument value; if the address is taken or a reference is bound to the argument or any of its non-static data members, the argument might be modified by another function. The wording below forbids such modification by any of the function's dynamic descendants. If the pointer or reference persists after the return of the function, using it to modify the argument at that point would be undefined behavior anyway because of accessing (what would have been) an object of automatic storage duration after its lifetime has ended.

Consequently, the implementation is allowed to permit the program to modify the actual argument under the "anything-goes" character of undefined behavior.

The second issue has to do with making the decision at run-time whether to perform the optimization or not. Consider code like

```
struct S {
     int i;
};
int f(S s, bool reset) {
     if (reset)
          s.i = 0;
     return s.i;
}
```

This code is safe for the optimization if called with `reset == false` and not otherwise. The proposed wording allows an implementation to avoid the copy in some calls but not in others by applying the "no modification" requirement only to the operations actually performed by the function.

Although the issue is written specifically to address inline functions, there seems to be no reason to preclude applying the optimization to non-inline functions as well.

Proposed Resolution

Add the following bullet to the list of permissible optimizations in 12.8¶15. (The form of this proposed change assumes the restructuring proposed below for the resolution of issue 20.)

- when an argument in a call to a function is a non-volatile lvalue of the same cv-unqualified class type as its corresponding (non-reference) parameter, and none of the operations performed by the function (including calls to other functions) can change the value of any of the parameter's non-static data members, the copy operation can be omitted by treating the parameter as if it were a reference to the argument instead of a copy

In addition, the example should be changed to reflect all three situations in which the optimization can be applied:

[Example:

```
struct Thing {
     Thing();
     ~Thing();
     Thing(const Thing&);
     int i;
};
```

```
Thing f() {
     Thing t;
     return t;
}

int g(Thing tp) {
     return tp.i;
}

Thing t2 = f();
int j = g(t2);
```

Here the criteria for elision can be applied to eliminate three calls to the copy constructor of class `Thing`. In the initialization of `t2`, the copying of the local automatic object `t` into the temporary object for the return value of function `f()` and the copying of that temporary object into object `t2` can both be elided. Effectively, the construction of the local object `t` can be viewed as directly initializing the global object `t2`, and that object's destruction will occur at program exit. Then the copy of `t2` into `g()`'s parameter `tp` can be eliminated, so that the reference to `tp.i` in the function body is effectively a reference to `t2.i`. *--end example]*

ISSUE 14

Issue 14 poses two questions, summarized in the following example:

```
extern "C" int f();
typedef int T;

namespace N {
     extern "C" int f();
     typedef int T;
}
using namespace N;
int i = f();        // "f" ambiguous?
T j;                // "T" ambiguous?
```

Even though the declarations of `f()` and `T` declare the same function and type, respectively, the name lookups find declarations in two distinct namespaces; is this fact sufficient to cause ambiguity, or must the implementation "look through" the declarations to decide the question of ambiguity?

Analysis

The Standard specifies the criteria for ambiguity in the presence of *using-declaration*s in 7.3.4¶4:

> If name lookup finds a declaration for a name in two different namespaces, and the declarations do not declare the same entity and do not declare functions, the use of the name is ill-formed.

The issue regarding the function declarations is easily settled by reference to 7.5¶6:

> Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function.

The two declarations of `f()` in the example thus satisfy the criterion of "[declaring] the same entity," so there is no ambiguity in its use.

The question of whether the two `typedef` declarations cause an ambiguity is more involved. According to 3¶3,

> An *entity* is a value, object, subobject, base class subobject, array element, variable, function, instance of a function, enumerator, type, class member, template, or namespace.

Conspicuously absent from this list is a typedef-name. Instead, a typedef-name is simply a synonym for an entity (7.1.3¶1). Since both the `typedef` declarations declare the same entity (type), once again there is no ambiguity in the use of the name.

Proposed Resolution

In order to make the treatment of typedef names clearer, a typedef should be added to the example in 7.3.4¶4:

```
namespace A {
     class X { };
     typedef int T;
     extern "C"     int g();
     extern "C++"   int h();
}
namespace B {
     void X(int);
     typedef int T;
     extern "C"     int g();
     extern "C++"   int h();
}
using namespace A;
using namespace B;

void f() {
     X(1); // error: name X found in two namespaces
```

```
        T I;    // okay: name T refers to the same entity
        g();    // okay: name g refers to the same entity
        h();    // error: name h found in two namespaces
}
```

ISSUE 20

There are three related sub-issues in issue 20, all dealing with the elision of copy constructors as described in 12.8¶15:

1) The text should make clear that the requirement that the copy constructor be accessible and unambiguous is not relaxed in cases where a call to a copy constructor is elided.

2) It is not clear from the text that the two optimizations described can be applied transitively, and, if so, the implications for the order of destruction are not spelled out.

3) The text should exclude applying the function-return optimization if the expression names a static or volatile local object. (This comment was made verbally in Santa Cruz, although it does not appear in the current text of the issue list.)

Analysis

After discussion in Santa Cruz, the core group decided that sub-issue #1 required no change; the necessity of an accessible and unambiguous copy constructor is made clear in 12.2¶1 and need not be repeated in this text. The remaining two sub-issues appear to be valid criticisms and should be addressed.

Proposed Resolution

The paragraph in question should be rewritten as follows. (Note: this restructuring is also intended to facilitate the inclusion of the proposal to resolve issue 6.) In addition, references to this section should be added to the index under "temporary, elimination of," "elimination of temporary," and "copy, constructor elision."

> When certain criteria are met, an implementation is allowed to omit copying a class object, even if the copy constructor and/or destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization *[footnote:* Because only one object is destroyed instead of two, and one copy constructor is not executed, there is still one object destroyed for each one constructed. -- *end footnote]*. This elision of copy operations is permitted in the following circumstances (which may be combined to eliminate multiple copies):

- in a `return` statement in a function with a class return type, where the expression is the name of a non-volatile automatic object with the same cv-unqualified type as the function return type, the copy operation can be omitted by constructing the automatic object directly into the function's return value

- when a temporary class object (12.2) would be copied to a class object with the same cv-unqualified type, the copy operation can be omitted by constructing the temporary object directly into the target of the omitted copy

[Example:

```
class Thing {
public:
     Thing();
     ~Thing();
     Thing(const Thing&);
};

Thing f() {
     Thing t;
     return t;
}

Thing t2 = f();
```

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class `Thing`: the copying of the local automatic object `t` into the temporary object for the return value of function `f()` and the copying of that temporary object into object `t2`. Effectively, the construction of the local object `t` can be viewed as directly initializing the global object `t2`, and that object's destruction will occur at program exit. -- *end example]*

Issue 40

Issue 40 has two sub-issues. The first concerns the statement in 8.3¶1,

> The *id-expression* of a *declarator-id* shall be a simple *identifier* except for the declaration of some special functions (12.3, 12.4, 13.5) and for the declaration of template specializations or partial specializations (14.7).

The second sub-issue is regarding another statement in the same paragraph:

> A *declarator-id* shall not be qualified except for the definition of a member function (9.3) or static data member (9.4) or nested class (9.7) outside of its class, the definition or explicit instantiation of a function, variable or class member of a namespace outside of its namespace, or...

Analysis

The problem in the first sub-issue is that the wrong syntactic non-terminal is mentioned. The relevant portions of the grammar are:

declarator-id:
id-expression
:: $_{opt}$ *nested-name-specifier* $_{opt}$ *type-name*

id-expression:
unqualified-id
qualified-id

unqualified-id:
identifier
operator-function-id
conversion-function-id
~ class-name
template-id

The exceptions in the citation from 8.3¶1 are all the non-*identifier* cases of *unqualified-id*: 12.3 is for *conversion-function-id*s, 12.4 is for destructors, 13.5 is for overloaded operators, and 14.7 is for *template-id*s. If taken literally, this sentence would exclude all *qualified-id*s, which it obviously is not intended to do. Instead, the apparent intent is something along the lines of

> If an *unqualified-id* is used as the *id-expression* of a *declarator-id*, it shall be a simple *identifier* except...

However, it does not appear that this restriction has any meaning; all of the possible cases of *unqualified-id*s are represented in the list of exceptions! Rather than recasting the sentence into a correct but useless form, it would be better to remove it altogether.

The second sub-issue deals with the conditions under which a *qualified-id* can be used in a declarator, including "the definition of a...nested class" and "the definition or explicit instantiation of a...class member of a namespace." However, the name in a class definition is not part of a declarator; these constructs do not belong in a list of declarator contexts.

Proposed Resolution

Delete the third sentence ("The *id-expression* of a *declarator-id* shall be a simple *identifier*...") from 8.3¶1. Delete the words "or nested class (9.7)" and change "function, variable or class member" to "function or variable member" in the fourth sentence of the same paragraph.

ISSUE 89

Issue 89 deals with construction of a new object in the same location as an existing object. 3.8¶9 makes this action undefined behavior if the object was `const` and either static or automatic, presumably to allow optimizers to rely on the known value of such an object. However, nothing is said about an object with a member of reference type; the issue suggests that a similar restriction be applied for the same reason.

Analysis

When this issue was discussed in Santa Cruz, there was general agreement in the core group that something should be done to make it undefined behavior to "rebind" a reference member through reconstruction of its containing object. The issue suggests amending 3.8¶9 to cover the reference case as well as the `const` case.

The proposal below, however, takes a different approach. The current 3.8¶9 seems more intended to address the "ROM-ability" of an object than optimizer considerations. For one thing, its applicability is limited only to static and automatic objects, which are more susceptible to the kind of analysis required to determine if an object can be placed into read-only memory than are objects that are dynamically allocated in free store and whose lifetime is less easily determined. Furthermore, the restriction is not upon use of the value of such an object after reallocation, which is where an optimizer would be expected to encounter trouble; instead, it is the mere act of reallocation itself that produces undefined behavior (just like any other attempt to modify a `const` object, per 7.1.5.1¶4).

3.8¶7 lists restrictions on using a pointer, reference, or name that previously referred to an object to refer to a new object created in the same storage. Since the concern here is optimization, in particular assumptions about the binding of a reference member whose previous binding was known, it seems more natural to augment the restrictions of 3.8¶7 to address the issue of reference members (i.e., placing a limitation on the use of the new object in confusing ways rather than on its creation). In addition, this paragraph is not limited to static and automatic objects, and there seems to be no reason to allow "rebinding" of references in free-store objects, either.

Proposed Resolution

Add a new bullet to the list of restrictions in 3.8¶7, following the second bullet ("the new object is of the same type..."):

- the type of the objects, if a class type, does not contain any non-static data member whose type is `const`-qualified or a reference type, and