Name Look Up Issues and Proposed Resolutions
============================================

452a - How does name look up proceeds for names after . or -> ?

Question 1:
-----------

5.2.4 [expr.ref] paragraph 3 says:
"If the nested-name-specifier of the qualified-id specifies a
namespace name, the name is looked in the context in which the
entire postfix-expression occurs. ... If the nested name
specifier of the qualified-id specifies a class name, the class
name is looked up as a type both in the class of the object
expression (or the class pointed to by the pointer expression) and
the context in which the entire postfix-expression occurs."

This is backward. One doesn’t know if the name is a namespace name
or class name until the name has been looked up. In which scope must
the name following the . or -> operator be first looked up?

namespace N { }
struct S {
  class N { }; 
};
S s;
... s.N::b ...

The scope of the object-expression ‘s’ or the scope in which the
entire expression takes place?

Proposal:
=========

Replace 3.4.4 [basic.lookup.classref] with the following:
"1 Given the postfix-expression:
  object-expression.id-expression
or
  object-expression->id-expression
the object-expression (in the first case) has type class T
and (in the second case) has type pointer to class T. The
id-expression can be a single name or a qualified-id.

2 If the id-expression is a single name, the name is looked up as a
member of class T. The program is ill-formed if class T doesn’t
have a member of that name. If the id-expression is a
conversion-function-id, its conversion-type-id is looked up in the
scope of class T and in the context in which the entire
postfix-expression occurs. The conversion-type-id might be found
in either or both contexts. If the name is found in both contexts.
  it shall denote the same type.

3 If the id-expression is a qualified-id:
-- if its nested-name-specifier begins with
class-name-or-namespace-name::...
  the class-name-or-namespace-name is looked up as follows:
-- the class-name-or-namespace-name is looked up as a class name
in the scope of class T. If such a class name is found, the
context in which the entire postfix-expression occurs is also
searched for a class name. If no class name is found in that
context, the result is the class name found in the scope of class T. If a class name is also found in the context of the entire postfix-expression, it shall refer to the same class type as the name found in the scope of class T. [Note: because the name of a class is inserted in its class scope (_class_), the name of a class is also considered a nested member of that class.];
-- otherwise, if name look up does not find a class name in the scope of class T, the context in which the entire postfix-expression occurs is searched for a namespace name or class name. If such a name is found, the result is the class name or namespace name;
-- otherwise, the program is ill-formed.

-- if its nested-name-specifier begins ::class-name-or-namespace-name::...
the class-name-or-namespace-name is looked up in global scope as a namespace name or class name. If such a name is found, the result is the class name or namespace name. Otherwise the program is ill-formed.

If the qualified-id refers to a conversion-function-id, its conversion-type-id shall denote the same type in both the context in which the entire postfix-expression occurs and in the context of the class name or namespace name of the nested-name-specifier.

4 If the nested-name-specifier of the qualified-id contains a class template-id (_temp.names_), its template-arguments are evaluated in the context in which the entire postfix-expression occurs."

Delete 5.2.4 [expr.ref] paragraph 3 and 4, and change paragraph 2 2nd sentence to say:
"The id-expression shall name (3.4.4 _basic.lookup.classref_) a member of that class, except that an imputed destructor can be explicitly invoked for a scalar type (_class.dtor_)."

Question 2:
-----------
Neal Gafter also asks:
> "In the syntax
> p->template T<args>::x
> in which scope(s) is T looked up?"

template <class X> class T { static X x; };
class C {
    template <class X> class T { static X x; };
};
C* p;
...p->template T<args>::x ...

Proposal: =========
The rules above already cover this.
The template name found is the one in the class scope, i.e. C::T.

433- What is the syntax for explicit destructor calls?

John Spicer asked the following questions:

Question 1:
-----------
> Can a typedef name be used following the ~, and if so, what are the
> lookup rules?
> 
> struct A {
> ~A(){}  
> }
> 
> typedef class A B;
> 
> int main()
> {  
> A* ap;
> ap->A::~A();    // OK
> ap->B::~B();    // cfront/Borland OK, IBM/Microsoft/EDG error
> ap->A::~B();    // cfront OK, Borland/IBM/Microsoft/EDG error
> ap->~B();     // OK?
> }
>
> 5.1[expr.prim] paragraph 8 last sentence says:
"Where class-name::~class-name is used, the two class-names shall
refer to the same class; this notation names the destructor."

12.4[class.dtor] paragraph 11 says:
"In an explicit destructor call, the destructor appears as a ~
followed by a type-name that names the destructor’s class type."
The interaction between these two rules is not really clear.

Proposal:
==========
Replace the first two sentences of 3.4.4 (proposed above) with:
"2 If the id-expression is a single name, the name is looked up
as follows:
-- the name is looked up as a member of class T. If T has a
member of that name, the id-expression refers to that class
member.
-- Otherwise, if the id-expression is an explicit destructor
call (12.4, _class.dtor_), the name is looked up as a type
name in the context in which the entire postfix-expression
occurs. If such a name is found, the result is the type
name;
-- otherwise, the program is ill-formed."

Add to 12.4, paragraph 11, as a second sentence:
"The destructor name in an explicit destructor call can also
be a qualified-id (5.1)."

For the example above:
ap->A::~A();    // OK
ap->B::~B();    // error: the class-names in a qualified-id cannot r
               // be typedef-names, see 5.1.
ap->A::~B();    // error: same as above
ap->~B();     // OK

Question 2:
-----------
12.4 [class.dtor], paragraph 10 says:
The notation for explicit call of a destructor may be used for
any simple type name. ... [Example:
int* p;
p->int::~int();
-- end example]

Must the destructor name be a qualified-id or can it be written
as:
p->~int();
?

The issue in question 1) also applies to the lookup of explicit
typedef int I;
typedef int I2;
int* i;
i->int::~int();
i->I::~I();
i->I::~int();
i->I::~I2();

Which ones of these are well-formed?

Proposal:
======
I am trying to mimic the behavior for class types.

Add to 12.4, paragraph 14:
"The destructor name in an explicit destructor call for a simple
type name shall have one of the following forms:
~simple-type-specifier
~typedef-name
simple-type-specifier::~simple-type-specifier
A typedef-name is looked up in the context in which the entire
postfix-expression occurs. The simple-type-specifier or the
type of the typedef-name shall represent a scalar type. Where
simple-type-specifier::~simple-type-specifier is used, the
simple-type-specifiers shall refer to the same scalar type."

570 - Name look up for anonymous union member names need to be better &
described

Question 1:
---------
9.5[class.union] paragraph 2 says:
"The names of the members of an anonymous union shall be distinct
from other names in the scope in which the union is declared; ..."
Is this true?

How about:
int I;
static union {
  class I { }; // error?
};
void f() {
  class I i; // is this OK?
}

How about:
class C;
static union {
  class C { }; // does this complete the type of global
               // class C?
};

Proposal:
======
Add a note at the end of paragraph 2 to make it clear that the
examples above are ill-formed.
"[Note: a class name cannot coexits with an object, function or
enumerator with the same name if one of these entities is an
anonymous union member and the other entity is declared in the
scope containing the anonymous union definition. Also, a class
defined in the member list of an anonymous union cannot complete a
class that is forward declared in the scope containing the
anonymous union definition.

Question 2: How can static members which are anonymous unions be initialized?

Mike Miller asked the following:
> class C {
>   static union {
>     int i;
>     char * s;
>   }
>   int C::i = 3; // ? Is this syntax valid?
>   int C::a = 5; // ? Is this syntax valid?
>
> 9.5 [class.union], paragraph 2 says:
"The names of the members of an anonymous union shall be distinct from other names in the scope in which the union is declared; they are used directly in that scope without the usual member access syntax (_expr.ref_)."

What does: "they are used directly in that scope without the usual member access syntax" allow? Can they be refer to using qualifiers?

Proposal:

I believe the names of anonymous union members should always be visible in the scope in which the anonymous union is declared whether the names are named with unqualified-ids, with qualified-ids, or using the class member access syntax. This seems consistent with the rules for naming members of unnamed namespaces and will probably be less confusing for users.

Change the sentence in 9.5 to:
"During name look up, the name of anonymous union member is found when the scope in which the anonymous union is declared is searched for the declaration of that name. [Note: the names of anonymous union members are found whether unqualified name look up (3.4.1 _basic.lookup.unqual_), qualified name look up (3.4.2 _basic.lookup.qual_) or the class member access look up (3.4.4 _basic.lookup.classref_) is applied.]"