Doc No: X3J16/96-0173 WG21/N0991
Date: September 24th, 1996
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

Core WG List of Issues

The issues listed as editorial or as closed in the version of the core list of issues that appeared in the Post-Stockholm mailing (96-0167/N0985) were resolved in the pre-Stockholm version of the working paper (WP) and are therefore not listed in this version of the core list of issues.

Notice that most of the issues listed below are editorial and will be handled during the editorial sessions at the Hawaii meeting.

Syntax

9.2 [class.mem]:
  692: ";opt" after member "function-definition" should be omitted

C Compatibility

1.1 [intro.scope]:
  604: Should the C++ standard talk about features in C++ prior to 1985?
clause 16:
  679: "Shall" is used incorrectly in clause 16
Annex C:
  680: Annex C subclause C.1 is out of date
  681: The type of string literals is array of const char - this has implicitions for C compatibility and should be in Annex C

Core1

Conformance model

1.7 [intro.compliance]:
  602: Are ill-formed programs with non-required diagnostics really necessary?
  619: Is the definition of "resource limits" needed?

Name Look Up

3.3.6 [basic.scope.class]:
  664: When does the reevaluation rule for class scope name lookup require a diagnostic?
3.4.X [basic.lookup.koenig]:
  686: Where is a function name looked up if an argument type is introduced with a typedef or a using-declaration?
3.4.2 [basic.lookup.qual]:
  665: In X::~Y is Y looked up in the context of the current expression?
3.4.3 [basic.lookup.elab]:
  666: Are class names used in an elaborated-type-specifier hidden by namespace names?
3.4.4 [basic.lookup.classref]:
  688: Rules for name lookup after :: . -> need to be clarified for conversion-function-id, template argument names and destructor names

7.3.3 [namespace.udecl]:
  673: Does a using-declaration for an enum type declare aliases for the
       enumerator names as well?
7.3.4 [namespace.udir]:
  612: name look up and unnamed namespaces
10.2 [class.member.lookup]:
  674: How do using-declarations affect class member lookup?
10.3 [class.virtual]:
  675: How do using-declarations influence the selection of a final virtual
       function overrider?

Linkage / ODR

3.5 [basic.link]:
  526: What is the linkage of names declared in unnamed namespaces?
9.5 [class.union]:
  505: Must anonymous unions declared in unnamed namespaces also be static?

Object/Memory Model

3.6.3 [basic.start.term]:
  663: Should the meaning of a coexisting C/C++ implementation be defined?
3.7.3 [basic.stc.dynamic]:
  667: What does "predeclared" operator new mean?
5.3.4 [expr.new]:
  638: When is access/ambiguity on operator delete checked?
  669: semantics for new and delete expressions should be separated from the
       requirements for operator new and delete
  690: Clarify the lookup of operator new in a new expression
6.7 [stmt.dcl]:
  635: local static variable initialization and recursive function calls
7.3.3 [namespace.udecl]:
  672: using-declarations and base class assignment operators
10.1 [class.mi]:
  624: class with direct and indirect class of the same type: how can the
       base class members be referred to?
12.6 [class.init]:
  138: When are default ctor default args evaluated for array elements?
12.8 [class.copy]:
  687: The WP prohobits the copy assignment of virtual base classes to behave
        like the copy constructor

+-------+
| Core2 |
+-------+

Sequence Points

1.8 [intro.execution]:
  603: Do the WP constraints prevent multi-threading implementations?
  605: The execution model wrt to sequence points and side-effects needs work

lvalue/rvalue

6.2 [stmt.expr]:
  645b: When is the result of an expression statement converted to an rvalue?

Types / Classes / Unions

3.9 [basic.life]:
  621: The terms "same type" need to be defined

Default Arguments

8.3.6 [dcl.fct.default]:

689: What if two using-declarations refer to the same function but the declarations introduce different default-arguments?

Types Conversions / Function Overload Resolution

4.2 [conv.array]:
668: Should the conversion from string-literal to pointer to char be an "array-to-pointer" conversion which has exact match rank in function overload resolution?
670: Is the comparison between void* and cv T* well-formed?
5.17 [expr.ass]:
691: is bool += 1 valid?
7.2 [dcl.enum]
683: What is the underlying type of an enumeration type if the value of an enumerator uses the value of a previous enumerator?
13.6 [over.built]:
682: operator ?: and operands of enumeration types
13.3.1.1.2 [over.call.object]:
662: Do cv-qualifiers on the class object influence the operator() called?
13.3.3.2 [over.ics.rank]:
684: The ranking for implicit conversion sequences for pointer types should take into account qualification conversions in 4.4
685: What is the ranking of a user-defined conversion that combines a pointer conversion with casting away cv-qualifiers?

+--------+
| Core 3 |
+--------+

Pointer to members

5.5 [expr.mptr.oper]:
644: Must the operand of .* and ->* have a complete class type?

RTTI

5.2.6 [expr.dynamic.cast]:
549: Is a dynamic_cast from a private base allowed?

Templates

6.8 [stmt.ambig]:
671: Does template instantiation happen during parser ambiguity resolution?
14.7.1 [templ.inst]:
676: When is a template instantiated?
14.8.2 [temp.deduct]:
677: Should the text on argument deduction be moved to a subclause discussing both function templates and class template partial specializations?

Exception Handling

15.1 [except.throw]:
678: Can the exception object created by a throw expression have array type?

==========
Chapter 1 - Introduction

Work Group: Core
Issue Number: 604
Title: Should the C++ standard talk about features in C++ prior to 1985?
Section: 1.1 [intro.scope]
Status: active

Description:
UK issue 229:
"Delete the last sentence of 1.1 and Annex C.1.2. This is the first standard for C++, what happened prior to 1985 is not relevant to this document."
Resolution:
Requestor: UK issue 229
Owner: Tom Plum (C Compatibility)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 602
Title: Are ill-formed programs with non-required diagnostics really necessary?
Section: 1.7 [intro.compliance]
Status: active
Description:
UK issue 9:
"We believe that current technology now allows many of the non-required diagnostics to be diagnosed without excessive overhead. For example, the use of & on an object of incomplete type, when the complete type has a user-defined operator&(). We would like to see diagnostics for such cases."

[note JL:]
At the Tokyo meeting, we discussed this a bit and decided that this issue required more dicussions.

Question: Do deprecated features render a program ill-formed but no diagnostic is required?

See also UK issue 93.
Resolution:
Requestor: UK issue 9
Owner: Josee Lajoie (Conformance Model)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 619
Title: Is the definition of "resource limits" needed?
Section: 1.7 [intro.compliance]
Status: active
Description:
1.7 para 1 says:
"Every conforming C++ implementation shall, within its resource limits, accept and correctly execute well-formed C++ programs..."
The term resource limits is not defined anywhere.
Is this definition really needed?
Resolution:
Requestor: ANSI Public comment 7.12
Owner: Josee Lajoie (Conformance Model)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 603
Title: Do the WP constraints prevent multi-threading implementations?
Section: 1.8 [intro.execution]
Status: active
Description:
UK issue 11:
"No constraints should be put into the WP that preclude an implementation using multi-threading, where available and

appropriate."

Bill Gibbons notes:
For example, do the requirements on order of destruction between sequence points preclude C++ implementations on multi-threading architectures?

Resolution:
Requestor: UK issue 11
Owner: Steve Adamczyk (sequence points)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 605
Title: The execution model wrt to sequence points and side-effects needs work
Section: 1.8 [intro.execution]
Status: active
Description:

See UK issues 263, 264, 265, 266:
1.8 para 9:
"What is a "needed side-effect"? This paragraph, along with footnote 3 appears to be a definition of the C standard "as-if" rule. This rule should be defined as such. [Proposed definition of "needed": if the output of the program depends on it.]"
1.8 para 10:
"It is not true to say that values of objects at the previous sequence point may be relied on. If an object has a new value assigned to it and is not of type sig_atomic_t the bytes making up that object may be individually assigned values at any point prior to the next sequence point. So the value of any object that is modified between two sequence points is indeterminate between those two points. This paragraph needs to be modified to reflect this state of affairs."

Also, para 11:
"Such an object [of automatic storage duration] exits and retains its last-stored value during the execution of the block and while the block is suspended ..."
This is not quite correct, the object may not retain its last-stored value.

Para 9, 10, 11 and 12 also contain some undefined terms.

Resolution:
Requestor: UK issues 263, 264, 265, 266
Owner: Steve Adamczyk (sequence points)
Emails:
Papers:

. . . . . .

==========

Chapter 2 - Lexical Conventions

==========

Chapter 3 - Basic Concepts

Work Group: Core
Issue Number: 664
Title: When does the reevaluation rule for class scope name lookup require a diagnostic?
Section: 3.3.6 [basic.scope.class]
Status: active
Description:

3.3.6 para 1 says:
1) The potential scope of a name declared in a class consists not only of the declarative region following the name’s declarator, but also of all function bodies, default arguments, and

constructor ctor-initializers in that class (including such things in nested classes).
2) The name N used in a class S shall refer to the same declaration when re-evaluated in its context and in the completed scope of S.
3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program's behavior is ill-formed, no diagnostic is required.

According to the wording above, a diagnostic is required to be issued for the following program. Should it?

```
typedef int I; //1

class D {
    typedef I I; //2
};
```

This is ill-formed according to rule 2) but not according to rule 3) (i.e. this not a reordering problem). Rule 3) is the rule for which "no diagnostic is required."

Should Rule 2) also say: "no diagnostic is required."? Otherwise, this will require that an implementation processes class member declarations twice in order to determine if names used by the declaration change meaning.

Resolution:
Requestor: Steve Adamczyk
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 686
Title: Where is a function name looked up if an argument type is introduced with a typedef or a using-declaration?
Section: 3.4.X [basic.lookup.koenig]
Status: active
Description:

basic.lookup.koenig says:

When an unqualified name is used as the postfix-expression in a function call (_expr.call_), other namespaces not considered during the usual unqualified look up (_basic.lookup.unqual_) may be searched; this search depends on the types of the arguments.

For each argument type T in the function call, there may be a set of zero or more associated namespaces to be considered; such namespaces are determined in the following way:
[...]
- If T is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.

This text is not very clear as to what happens if the type was introduced with a typedef or a using-declaration:

```
namespace N1 {
        struct T { };
        void f(T);
        void g(T);
};

namespace N2 {
        using N1::T;
        typedef N1::T U;
```

```
                void f(T);
                void g(U);
        };

        void foo() {
                N2::T t;
                N2::U u;

                f(t);           // which f?
                g(u);           // which g?
        }
```

Proposed Resolutions:

Bill Gibbons in core-7041:

> f(t) calls N1::f.
> A using-declaration guides the lookup, but if the result of the
> lookup is the using-declaration then the semantics are the same as
> if the result of the lookup had been the original declaration to
> which the using-declaration refers.

Tom Wilcox in core-7042:

> I would have said N1::f and N2::g. Since T is only used and not
> declared in N2, f should come from N1. And since U is declared in
> N2, we get g from N2.

Resolution:

Requestor: Andrew Koenig

Owner: Josee Lajoie (Name Lookup)

Emails: core-7041

Papers:

. . . . . .

Work Group: Core

Issue Number: 665

Title: In X::~Y is Y looked up in the context of the current expression?

Section: 3.4.2 [basic.lookup.qual]

Status: active

Description:

In an expression like

```
                p->X::~X();
```

where is the "X" that follows the "~" looked up?

3.4.4 [basic.lookup.classref] says that in an unqualified name, the name after the ~ is looked up in the current context and in the class of p. But it doesn’t say anything special about the qualified case. This implies that it is looked up in the scope of X only. If this is true, it seems to me that is a problem because it doesn’t work when X is a typedef, as in:

```
        struct A {
                ~A();
        };

        typedef A AB;

        int main()
        {
                AB *p;
                p->AB::~AB();
        }
```

This suggests that the name after ~ should always be looked up in the current context, even for the qualified name case. Presumably, for the qualified name case it would also be looked up in the class of the qualifier.

Resolution:
Requestor: John Spicer
Owner: Josee Lajoie (Name Look Up)
Emails:
Papers
. . . . . .
Work Group: Core
Issue Number: 666
Title: Are class names used in an elaborated-type-specifier hidden by namespace names?
Section: 3.4.3 [basic.lookup.elab]
Status: active
Description:

3.4.3 para 1:
"An elaborated-type-specifier may be used to refer to a previously declared class-name or enum-name even though the name has been hidden by an object, function, or enumerator declaration."

Shouldn't this list also include namespace names?

```
struct S { };
namespace A {
    namespace S {
        struct S sb; // ill-formed?
    }
}
```

Resolution:
Requestor:
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 688
Title: Rules for name lookup after :: . -> need to be clarified for conversion-function-id, template argument names and destructor names
Section: 3.4.4 [basic.lookup.classref]
Status: active
Description:

How is
o a destructor name
o an id-expression of a conversion-function-id
o a template-id
o the name of a template-argument
looked up when used following a nested-name-specifier or a class member access operator . or -> .

Bill Gibbons provided the following table, which I [Josee] filled up:

expression	name to look up	look in surrounding context	must be visible there ?	look in what class	must be visible there
A::b	b	no	---	A	yes
A::~T	T	no	---	A	yes
A::Z::~T	Z	no	---	A	yes
A::Z::~T	T	no	---	A::Z	yes
A::operator T	T	no	---	A	yes
A::operator Z::T	Z	no	---	A	yes
A::operator Z::T	T	no	---	A::Z	yes
A::C<D>	C	no	---	A	yes
A::C<D>	D	yes	yes	no	---

A::X::b	b	no	---	A::X	yes
A::X::~T	T	no	---	A::X	yes
A::X::Z::~T	Z	no	---	A::X	yes
A::X::Z::~T	T	no	---	A::X::Z	yes
A::X::operator T	T	no	---	A::X	yes
A::X::operator Z::T	Z	no	---	A::X	yes
A::X::operator Z::T	T	no	---	A::X::Z	yes
A::X::C<D>	C	no	---	A::X	yes
A::X::C<D>	D	yes	yes	no	---

a.b	b	no	---	A	yes
a.~T	T	yes	yes	A	yes
s.~T	T	yes	yes	---	---
a.operator T	T	yes	yes	A	yes
a.operator Z::T	Z	yes	yes	A	yes
a.operator Z::T	T	no	---	Z	yes
a.C<D>	C	no	---	A	yes
a.C<D>	D	yes	yes	no	---

a.X::b	X	yes	no	A	no
a.X::b	b	no	---	X	yes
a.X::~T	T	no	---	A::X	yes
s.X::~T	T	yes	yes	---	---
a.X::operator T	T	no	---	A::X	yes
a.X::operator Z::T	Z	no	---	A::X	yes
a.X::operator Z::T	T	no	---	A::X::Z	yes
a.X::C<D>	C	no	---	A::X	yes
a.X::C<D>	D	yes	yes	---	---

where a is an object of class type A
where s is an object of scalar type

We have to clarify the WP to ensure that the above resolutions are clear.

Bill also raises the following issues:

* The current rules for lookup of "T" in "a.operator T" break template because "T" must be visible in the class, which is impractical if "T" is a template type parameter. I propose changing the rule so the lookup is in the surrounding context only, as with template-id arguments.

* The current rules for lookup of "X" in "a.X::b" break templates because when "T" is a template type argument, the instantiation will fail if some base class of "A" (which might itself be a template type argument) happens to have a typedef or class member "T". This might be fixed as a special case in template name lookup, but I propose the simpler fix of changing the rule so the lookup is in the surrounding context only.

Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (Name Lookup)
Emails: core-6969
Papers
. . . . . .
Work Group: Core
Issue Number: 526
Title: What is the linkage of names declared in unnamed namespaces?
Section: 3.5 [basic.link] Program and linkage
Status: active
Description:
What is the linkage of names declared in an unnamed namespace?
Internal linkage?
Internal linkage applies to variables and functions.
What would the status of a type definition be in an unnamed namespace? No linkage?

Can it be used to declare a function with external linkage?
Can it be used to instantiate a template?

```
namespace {
  class A { /* ... */ };
}
extern void f(A&);                              // error?
template <class T> class X { /* ... */ };
X<A> x;                                         // error?
```

If A does not have external linkage, then the two declarations are probably errors. If it does have external linkage, then the two declarations are legal (and the implementation probably has to worry about name mangling).

Resolution:
Requestor: Mike Anderson
Owner: Josee Lajoie (Linkage)
Emails: core-5905 and following messages.
Papers:

. . . . . .

Work Group: Core
Issue Number: 663
Title: Should the meaning of a coexisting C/C++ implementation be defined?
Section: 3.6.3 [basic.start.term]
Status: active
Description:

3.6.3 Termination [basic.start.term], paragraph 4 states:

"Where a C++ implementation coexists with a C implementation, any actions specified by the C implementation to take place after the atexit functions have been called take place after all destructors have been called."

What exactly does it mean for a C++ implementation to "coexist" with a C implementation?

Is this quoted paragraph a constraint on conforming C++ implementations? That would raise the spectre where a C++ implementation could be rendered non-conforming by the mere *existence* of a certain (perhaps maliciously designed) C implementation!

Is the quoted paragraph a constraint on C implementations? (But how could this be? How could the C++ standard constrain C implementations, which don't claim to conform to the C++ standard?)

Or is the quoted paragraph simply a non-normative "hint" to compiler writers, the sort of thing that John Skaller would probably call meaningless waffle? (In which case, what is it doing in the main text of the standard?)

As the draft currently stands, I believe the third alternative is the most reasonable interpretation, although frankly the draft is not clear.

Proposed Resolution:

Delete the paragraph in question.

Resolution:
Requestor: Fergus Henderson
Owner: Josee Lajoie (Memory Model)
Emails: core-6823
Papers:

. . . . . .

Work Group: Core
Issue Number: 667
Title: What does "predeclared" operator new mean?

Section: 3.7.3 [basic.stc.dynamic]
Status: active
Description:
        3.7.3 para 2 says:
        "The following allocation and deallocation functions are implicitly
        declared in a program
                ::operator new(size_t)
                ::operator new[](size_t)
                ::operator delete(void*)
                ::operator delete[](void*)
        "

        One implication of having predeclared operators is that the
       declarations would have to be explicitly repeated if there were other
       overloads of operator new declared in global scope, otherwise the
       overload declarations would hide the implicit declaration. For
       instance,

        void* operator new(size_t, long); // hides predeclared op new

        int* i = new int;   // ill-formed: no operator new(size_t)
                                // visible at this point

        It seems that it depends on how we define "implicitly declared" to
       work -- are "implicit declarations" considered to be in an imaginary
       scope containing the global scope, or are implicit declarations in
       the global scope itself and act just the way an explicit declaration
        would in the global scope? Is it well-defined somewhere what
        "implicitly declared" means? We need to pin it down.
Resolution:
Requestor: Mike Miller
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 621
Title: The terms "same type" need to be defined
Section: 3.9 [basic.types]
Status: active
Description:
       The WP needs to define what it means for two objects/expressions
       to have the same type. The phrase is used a lot throughout the WP.
Requestor:
Owner: Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
==========
 Chapter 4 - Standard Conversions

Work Group: Core
Issue Number: 668
Title: Should the conversion from string-literal to pointer to char
               be an "array-to-pointer" conversion which has exact match
               rank in function overload resolution?
Section: 4.2 [conv.array]
Status: active
Description:
        4.2 para 2:
        "A string literal ... can be converted to an rvalue of type
         "pointer to char"... the result is a pointer to the first element of
        the array."

        The conversion of a string literal from the type "const char *" to
       the type "char *" is in the array-to-pointer conversion section.

This means that this conversion is ranked as an exact match during function overload resolution. i.e.

```
void f(char*);
void f(const char*);
f("abc"); // ambiguous
```

When the conversion is eventually removed (it is currently deprecated), then the call above will be well-formed, and void f(const char*) will be chosen. This is different from Kevlin Henney's proposal, which suggested that the function void f(const char*) be selected.

In private email, Steve Adamczyk noted that core 2 didn't notice the impact of the proposed wording on the overload resolution weighting.

Resolution:
Requestor:
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

Chapter 5 - Expressions

Work Group: Core
Issue Number: 549
Title: Is a dynamic_cast from a private base allowed?
Section: 5.2.6 [expr.dynamic.cast]
Status: active
Description:

paragraph 8 says:
"...if the type of the complete object has an unambiguous public base class of type T, the result is a pointer (reference) to the T sub-object of the complete object. Otherwise, the runtime check fails."

This contradicts the example that follows:

```
class A { };
class B { };
class D : public virtual A, private B { };
...
D d;
B* bp = (B*) &d;
D& dr = dynamic_cast<D&>(*bp); // succeeds
```

According to the wording in paragraph 8, the cast above should fail.

Bill Gibbons noted the following:

First, the access restrictions on dynamic_casts appear to come from the access restrictions on static_cast, where neither upcasting nor downcasting across private derivation is allowed.

Yet dynamic_cast does not apply these restrictions consistently, even for simple downcasts:

```
struct A { virtual void f() { } };
struct B : private A { };
struct C : public  B { };
void f() {
    A *a = (A*) new C;
    B *b = static_cast<B*>(a);  // ill-formed
    B *b = dynamic_cast<B*>(a); // OK under 1st "otherwise"
}
```

I see several ways to clean this up:

(1) Change the first "otherwise" clause to also require that "v points (refers) to a public base class sub-object of the most derived object". This seems closest to the intent of the current wording. It would make the above example ill-formed.

This is equivalent to saying that a dynamic cast is OK if it can be done with a static cast to the most derived type followed by a static cast to the final type, ignoring the uniqueness and virtual inheritance restrictions on static downcasts.

(2) Say something like:

A dynamic cast is well-formed if there exists a class X within the most derived object hierarchy (including the most derived class) such that:

-- "v" refers to X or a public base class of X; and

-- T is X or a public base class of X.

That is, a dynamic cast is OK if it can be done with any combination of two static casts, ignoring the uniqueness and virtual inheritance restrictions on static downcasts. This would also make the above example ill-formed.

(3) Change both dynamic_cast and static_cast; see below.

I had also forgotten (and was somewhat dismayed to rediscover) that static_cast cannot be used to break protection. For example:

```
struct A { };
struct B : private A { };
void f() {
    B *b = new B;
    A *a1 = (A*) b;               // OK
    A *a2 = static_cast<A*>(b);   // ill-formed
    A *a3 = dynamic_cast<A*>(b); // well-formed,
                                  // but "a3" not usable
}
```

Did we really intend to do this, or was it an accidental side effect of defining static_cast in terms of the inverse of an implicit cast?

Also, I see no reason to restrict downcasting across private inheritance. If static_cast were changed to allow it, I would consider the "across private inheritance" part to be implicit, and the "downcasting" part to be the one that required an explicit cast.

In that light, I would propose one of these changes to dynamic_cast:

(1) Remove the first "public" from paragraph 8 and also allow downcasting to the most derived class, regardless of access.

(2) The equivalent of (2) above:

A dynamic cast is well-formed if there exists a class X within the most derived object hierarchy (including the most derived class) such that:

-- "v" refers to X or a base class of X; and

```
                -- T is X or a public base class of X.
```

That is, a dynamic cast is OK if it can be done with a combination of two static casts, ignoring the uniqueness and virtual inheritance restrictions on static downcasts. This would also make the above example ill-formed.

Similarly, should upcasting of pointers to members across private inheritance be restricted more than upcasting of pointers to members across public inheritance?

Resolution:
Requestor:
Owner: Bill Gibbons (RTTI)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 638
Title: When is access/ambiguity on operator delete checked?
Section: 5.3.4 [expr.new] New
Status: active
Description:

5.3.4 para 15 indicates that access and ambiguity on operator delete are checked when a new expression is encountered.

This does not seem quite right for objects of class type with a virtual destructor.

Some tricky examples were provided on the reflector during the discussion on this topic:

Example 1:

Roly Perera [core-6993]:

```
> struct B {
>     virtual ~B ();
>     void operator delete (void*);
> };
>
> struct D : B {
>     void operator delete (void*);
> };
>
> int main () {
>     B* pb = ::new D; // 1.  requires ::delete
>     delete pb;       // 2.  should find D::operator delete
> }
```

The deallocation function used by the delete expression could be the class operator delete even if the new expression uses global operator new. So the ambiguity/access of the class operator delete should always be checked.

Example 2.

Erwin Unruh [core-6997]:

```
> struct B {
>     virtual ~B ();
>     void operator delete (void*);
> };
>
> struct D : B {
>     void operator delete (void*) { /* does nothing !! */ }
```

```
> };
>
> int main () {
>     D d;
>     pb = &d;
>     delete pb;
>     exit(1);
> }
```

Erwin's example (though somewhat sick ;-) shows that a delete expression can be used without any new operator ever being called to create the object. The example deletes a local variable and since the operator delete does nothing, only the destructor is run. The destructor at the end of the block is bypassed by the call to exit. (yuck!).

Erwin says:

> I am perfectly happy to make this program ill-formed. But I as
> an implementor would like to have a rule which makes sure that I
> never try to call an operator delete [at runtime] which is
> ambiguous or inaccessible. Having undefined behaviour is a bad
> solution.

Proposed Resolution:

To handle these examples, these resolutions were suggested:

- para 15 has to be clarified to say that even if the storage for the class object or the array of classes is allocated using the global operator new, ambiguity and access is done on the class operator delete.

To following two resolutions were proposed to handle the case when the class has a virtual destructor:

- Ambiguity/access is checked for operator delete at the time the class is defined if the class has a virtual destructor.

- Bill Gibbons proposed the following resolution in core-7002:

  > [When a non-abstract class with a virtual destructor is
  > defined,] for each virtual destructor in the class, consider
  > those base classes in which:
  >
  > - the base class destructor is virtual
  >
  > - it is possible to use the delete keyword (without global
  >   qualifier) on a pointer which refers (by static type) to
  >   that base class
  >
  > For each such base class, find the operator delete which would
  > have been the "final overrider" if operator delete had been a
  > virtual function (for this purpose, treat global scope as a
  > base class). Each such "final overrider" must exist and all
  > of them must be the same.

Resolution:
Requestor: John Skaller
Owner: Josee Lajoie (Memory Model)
Emails: core-6988
Papers:

. . . . . .

Work Group: Core
Issue Number: 669
Title: semantics for new and delete expressions should be separated from the requirements for operator new and delete
Section: 5.3.4 [expr.new], 5.3.5 [expr.delete]
Status: editorial

Description:

Erwin Unruh wrote a paper (96-0011/N0829) that suggested that the semantics for the new expression and the delete expression be reworked so that they would only describe which operator new (or operator delete) they call. The restrictions on the behavior of the allocation and deallocation functions called should be moved to the library section.

Subclause 5.3.4[expr.new] and 5.3.5[expr.delete] still has some troublesome passages.

5.3.4 New

o Paragraph 8, last sentence says:
"The pointer returned by the new-expression is non-null and distinct from the pointer to any other object."

The part of this sentence that says "and distinct from the pointer to any other object" should be deleted. This is really a requirement on the library operator new. Maybe a note should be added to say: "If the library allocation function is called, the pointer returned is distinct from the pointer to any other object."

o Paragraph 13, first sentence says:
"The allocation function shall either return null or a pointer to a block of storage in which space for the object shall have been reserved."

This sentence should be moved to the note that follows. Again, this is a requirement that applies to the semantics of the library operator new and should not be in the normative text for 5.3.4.

Also paragraph 13 should be moved after paragraph 10, which discusses allocation functions.

o Paragraph 16 says:
"The allocation function can indicate failure by throwing a bad_alloc exception (_except_, _lib.bad.alloc_). In this case no initialization is done."

This should be changed to:
"If the allocation function exits by throwing an exception, no initialization is done."

5.3.5 Delete

o Paragraph 2, the last few sentences say:
"In the first alternative (delete object), the value of the operand of delete shall be a pointer to a non-array object created by a new-expression, or a pointer to a sub-object (_intro.object_) representing a base class of such an object (_class.derived_). If not, the behavior is undefined. In the second alternative (delete array), the value of the operand of delete shall be a pointer to the first element of an array created by a new-expression. If not, the behavior is undefined. [Note: this means that the syntax of the delete-expression must match the type of the object allocated by new, not the syntax of the new-expression.]"

The requirements that the object (or array) must be created by a new-expression should be removed. If a user operator delete is called, and this operator does nothing, then all is fine.

o Paragraph 7 says:
"To free the storage pointed to, the delete-expression will call a

deallocation function (_basic.stc.dynamic.deallocation_)."

"To free the storage pointed to," should be removed. Again, whether the storage is freed depends on which operator delete is called. A user operator delete may not free the storage.

Resolution:
Requestor: Erwin Unruh
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 690
Title: Clarify the lookup of operator new in a new expression
Section: 5.3.4 [expr.new]
Status: active
Description:

5.3.4 should describe the lookup of operator new in a new expression.

Here is an interesting example:

```
struct C {
        operator void* new(size_t);
        operator void* new[](size_t);
};

... new C[N1][N2]; // which operator new is called?
```

Resolution:
Requestor:
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 644
Title: Must the operand of .* and ->* have a complete class type?
Section: 5.5 [expr.mptr.oper]
Status: active
Description:

Para 2:
"The binary operator .* binds its second operand, which shall be of type ‘‘pointer to member of T ’’ to its first operand, which shall be of class T or of a class of which T is an unambiguous and accessible base class."

And something similar in para 3 for the ->* operator.

Since pointer to members of an incomplete class type are allowed, i.e.

8.3.3 para 2 says:
"
```
class T;
char T::* pmc;
```
[...]
the declaration of pmc is well-formed even though T is an incomplete type."

Must T be a complete class type when a pointer to member operator .* or ->* is applied to the pointer to member?

Resolution:
Requestor: Jerry Schwarz
Owner: Bill Gibbons (Pointer to members)
Emails:
Papers:

. . . . . .

Work Group: Core

Issue Number: 670
Title: Is the comparison between void* and cv T* well-formed?
Section: 5.9 [expr.rel]
Status: active
Description:

5.9 para 2
"Pointer conversions and qualification conversions are performed on pointer operands ... to bring them to the same type, which shall be a cv-qualified or cv-unqualified version of the type of one of the operands."

Should the following be well-formed?

```
const int * pci;
void * pv;

pv == pci; // well-formed?
```

The current wording indicates that it is ill-formed since the common type of the operands, after pointer conversions and qualification conversions are applied, is 'const void *'. The wording says that the type to which both operands are converted "shall be a cv-qualified or cv-unqualified version of the type of one of the operands."

According to 3.9.3 paragraph 1, the cv-qualified versions of 'void *' is 'void * const', 'void * volatile' or 'void * const volatile'. Because 'const void *' is not a cv-qualified version of 'void *', the comparison above is ill-formed.

However, the code above is valid C code.

Either the comparison above should be well-formed (in which case the wording that says: "which shall be a cv-qualified or cv-unqualified version of the type of one of the operands" needs to be fixed) or, it is ill-formed (in which case annex C needs to indicate this incompatibility between C and C++).

5.16[expr.cond] has similar problems.

Resolution:
Requestor:
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 691
Title: is bool += 1 valid?
Section: 5.17 [expr.ass]
Status: active
Description:

5.17 para 7:
"The behavior of an expression of the form E1 op= E2 is equivalent to E1 = E1 op E2 except that E1 is evaluated only once. In += and -=, E1 shall either have arithmetic or enumeration type or be a pointer to a possibly cv-qualified completely defined object type. In all other cases, E1 shall have arithmetic type."

Can E1 have type bool? If yes, what are the semantics?

Resolution:
Requestor:
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

==========

Chapter 6 - Statements

Work Group: Core
Issue Number: 645b
Title: When is the result of an expression statement converted to an rvalue?
Section: 6.2 [stmt.expr]
Status: active
Description:

```
class C;
extern C& f();
void foo() {
        f(); //1
}
```

Is line //1 ill-formed because the return value of f() is converted to an rvalue and C is an incomplete class type?

Resolution:
Requestor:
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 635
Title: local static variable initialization and recursive function calls
Section: 6.7 [stmt.dcl]
Status: active
Description:

```
int foo(int i) {
        if (i == 0) return i;
        static int x ( foo (i-1) );
        return x;
}
... foo (10) ...
```

What is the value of x after it has been initialized?

The WP indicates that the variable "x" will be initialized with the value 0.

- There can only be one "first time control passes completely through a declaration."
- It is not possible to get to the statement following the declaration without control passing completely through the declaration, so there is no possibility that the variable will be uninitialized in the following statement.
- When entering the declaration, we won't know if this will be the first time control passes completely through, so we must compute the initializing expression each time we enter when the variable has not yet been initialized.
- If the processor completes computing the initializing expression, and the variable has already been intialized, it must discard the computed value because only the first time through should do the initialization.

The return value from the function f the first time "control passes completely through the declaration" is 0.

This contradicts the example from the ARM (page 92)

```
int foo(int i) {
        static int s = foo(2*i);
```

```
            return i+1; // <<==
        }

        should result in an infinite loop or other undefined behavior (due to
        integer overflow), because there is no way to reach the marked line
        without s initialized, and there is no way to initialize s without
        reaching the marked line.
Proposed Resolution:
        At the Stockholm meeting, the members of the core 1 WG favoured
        giving such programs undefined behavior. This will have to be
        formally voted on at the Hawaii meeting.
Resolution:
Requestor:      Neal M Gafter
Owner:          Josee Lajoie (Initialization)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:  671
Title:          Does template instantiation happen during parser ambiguity
                resolution?
Section:        6.8 [stmt.ambig]
Status:         active
Description:
        6.8 [stmt.ambig] para 3:
        "[Note: because the disambiguation is purely syntatic, template
         instantiation does not take place during the diambiguation
         step.]

        Is the compiler allowed or required to instantiate during
        parser ambiguity resolution?  The WP would imply "no" but how
        is one otherwise to deal with "x<y>::z" during ambiguity
        resolution?
Resolution:
Requestor:      Neal Gafter
Owner:          Bill Gibbons / John Spicer (Templates)
Emails:
Papers:
. . . . . .
==========================================================================
Chapter 7 - Declarations
--------------------------
Work Group:     Core
Issue Number:   683
Title:         What is the underlying type of an enumeration type if the
               value of an enumerator uses the value of a previous
               enumerator?
Section:        7.2 [dcl.enum]
Status:         active
Description:
        There is a small omission in the description of the
        constant-expression which is used to set an enumerator's value, e.g.

            enum A { a, b = a + 2 );  // expression "a + 2"

        The type of "a" in "a+2" presumably follows the usual expression
        rules.  But these rules say, in 4.5/2:

           An rvalue of type wchar_t (3.9.1) or an enumeration type (7.2) can
           be converted to an rvalue of the first of the following types that
           can represent all the values of its underlying type: int,
           unsigned int, long, or unsigned long.

        So the evaluation of "a+2" depends on the underlying type of "A",
        which in turn depends on the value of "b", which depends on the value
        of "a+2".
```

Although this is unlikely to affect real programs in practice, we should fix the definition. There are cases where it matters, e.g.:

```
// Assume an environment where "int" is 16 bits, just for
// convenience (The same problem occurs when "int" is larger.
// Think of systems where "int" is 32 bits and "long" is 64
// bits.)

enum A { a = 1, b = a-2, c = 32768U };
```

If we assume the underlying type will be "int", then b is -1 and the actual underlying type is "long".

If we assume the underlying type will be "unsigned int", then b is 65535 and the actual underlying type is "unsigned int".

The answer may seem obvious, but consider:

```
enum A { a = 1U, b = a-2, c = -1 };
```

The underlying type will clearly be signed. Does "b" have the value "-1" or is the code ill-formed?

There seem to be several possible solutions to this problem:

1) When an enumerator is used in the defining expression of a subsequent enumerator in the same enumeration, its type is the type of its defining expression (where the default defining expression is "previous-enumerator + 1" except the first one, where it is "0").

2) Give enumerations an "interim" underlying type which is recomputed after each enumerator, and use that underlying type in subsequent defining expressions.

3) Require that enumerator computation be done with an infinite number of bits - assuming that the "as if" rule makes this practical.

4) Say that if the value of a defininging expression depends on the underlying type of the enumeration, the program is ill-formed.

Bill Gibbons' preference is (1).
Bill doesn't think it matters much what the answer is, but the should be described by the working paper.

A related problem occurs with the implicit "next value" rule:

```
enum B { a = 32767, b };
```

Is the code well-formed? If so, what is the underlying type? Why?
This example would be fixed if solution (3) was adopted.

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)
Emails: core-6989
Papers:

. . . . . .

Work Group: Core
Issue Number: 672
Title: using-declarations and base class assignment operators
Section: 7.3.3 [namespace.udecl]
Status: active
Description:

7.3.3 should indicate what happens if a using-declaration refers to a base class assignment operator and the type of this assignment operator corresponds to the type of the derived class copy assignment operator.

```
struct B;
struct A {
        B& operator=(const B&);
};
struct B : A {
        // introduces B's copy-assignment operator
        using A::operator=;
};
```

Proposed Resolution:

Add to 7.3.3 para 4:

"If the using-declaration refers to an assignment operator from a base class and this assignment operator has a parameter-clause such that it could be the copy-assignment operator for the class containing the using-declaration, the assignment operator will be used as the copy-assignment operator for the class containing the using-declaration."

Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 673
Title: Does a using-declaration for an enum type declare aliases for the enumerator names as well?
Section: 7.3.3 [namespace.udecl]
Status: editorial
Description:

```
namespace N {
        enum E { a, b };
}
using N::E;
int i = a; //ok? Is the enumerator 'a' visible here?
```

Proposed Resolution:

No. A using-declaration is an alias declaration for a name. Only the enumeration type name is declared in global scope with the using-declaration. This should be clarified in the WP.

Resolution:
Requestor:
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 612
Title: name look up and unnamed namespace members
Section: 7.3.4 [namespace.udir]
Status: active
Description:

Should static not be deprecated?

paragraph 5 says:

"If name look up 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."

Consider the program:

```
   struct S { };
```

```
            static int S;
            int foo() { return sizeof(S); }

         The sizeof will resolve to the static int S, because nontypes are
         favored.

         The standard says that unnamed namespaces will deprecate the use of
         static so we should be able to rewrite the program as:

            struct S { };
            namespace {
               int S;
            }
            int foo() { return sizeof(S); }

         However, the sizeof becomes ambiguous according to 7.3.4 para 5
         because the two S are from different namespaces. Is this right?
         Doesn't this mean that static should not be deprecated?
Resolution:
Requestor:
Owner:          Josee Lajoie (Name Look up)
Emails:
Papers:
. . . . . .
================================================================================
 Chapter 8 - Declarators
--------------------------
Work Group:     Core
Issue Number:  689
Title:          What if two using-declarations refer to the same function but
                the declarations introduce different default-arguments?
Section:        8.3.6 [dcl.fct.default]
Status:         active
Description:
         7.3.3 para 10 says:
         "If the set of declarations and using-declarations for a single
          name are given in a declarative region,
          -- they shall all refer to the same entity, or all refer to
             functions; or ..."

        8.3.6 para 9 says:
        "When a declaration of a function is introduced by way of a using
         declaration, any default argument information associated with the
         declaration is imported as well."

         This is not really clear regarding what happens in the following
         case:
                 namespace A {
                         extern "C" void f(int = 5);
                 }
                 namespace B {
                         extern "C" void f(int = 7);
                 }

                 using A::f;
                 using B::f;

                 f(); // ???
Resolution:
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (Default Arguments)
Emails:
Papers:
. . . . . .
================================================================================
 Chapter 9 - Classes
```

Work Group: Core
Issue Number: 692
Title: ";opt" after member "function-definition" should be omitted
Section: 9.2 [class.mem]
Status: active
Description:
The syntax says:
member-declaration:
...
function-definition ;opt

";opt" should be omitted. Otherwise, the syntax is ambiguous.
Resolution:
Requestor:
Owner: (Syntax)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 505
Title: Must anonymous unions declared in unnamed namespaces also be declared static?
Section: 9.5 [class.union] Unions
Status: active
Description:
9.5p3 says:
"Anonymous unions declared at namespace scope shall be declared static."
Must anonymous unions declared in unnamed namespaces also be declared static?
If the use of static is deprecated, this doesn't make much sense.

Proposal:
Replace the sentence above with the following:
"Anonymous unions declared in a named namespace or in the global namespace shall be declared static."

This is related to issue 526.
Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (linkage)
Emails:
Papers:
. . . . . .
==========
Chapter 10 - Derived classes

Work Group: Core
Issue Number: 624
Title: class with direct and indirect class of the same type: how can the base class members be referred to?
Sections: 10.1 [class.mi] Multiple base classes
Status: editorial
Description:
para 3 says:
"[Note: a class can be an indirect base class more than once and can be a direct and indirect base class.]"
The WP should describe how base class members can be referred to, how conversion to the base class type is performed, how initialization of these base class subobjects takes place.
Resolution:
At the Stockholm meeting, the core 1 WG decided to handle this as an editorial issue.
A note will be added to the WP to clarify the restrictions on accessing members of the direct base class.

Requestor:
Owner: Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 674
Title: How do using-declarations affect class member lookup?
Section: 10.2 [class.member.lookup]
Status: active
Description:

10.2 para 2:
"First, every declaration for the name in the class and in each of its base class sub-objects is considered. A member name f in one sub-object B hides a member name f in a sub-object A if A is a base class sub-object of B. Any declarations that are so hidden are eliminated from consideration. If the resulting set of declarations are not all from sub-objects of the same type, or the set has a nonstatic member and includes members from distinct sub-objects, there is an ambiguity and the program is ill-formed."

```
struct A { static int i; };  // NOTE: static member
struct B : A { };
struct C : A { using A::i; };
struct D : B, C { void foo(); };
void D::foo()
{
    i;   // ambiguous?
}
```

Is this ambiguous?
The declarations found are from sub-objects of different types; however, the declarations found refer to the same static member from a sub-object of type A.

Resolution:
Requestor:
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 675
Title: How do using-declarations influence the selection of a final virtual function overrider?
Section: 10.3 [class.virtual]
Status: editorial
Description:

If a virtual function final overrider can be introduced by a using-declaration, the WP should provide an example of what happens for hierarchies with multiple inheritance. The result in some situations will be somewhat surprising for the users.

```
class A {
        void f();
};

class B {
        virtual void f() = 0;
};

class C: public A, public B {
        using A::f; // override B::f from A::f
} c;

main()
```

```
       {
               c.f(); // call A::f
       }
Resolution:
Requestor:     Neal Gafter
Owner:          Josee Lajoie (Name Lookup)
Emails:
       core-7060
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
================================================================================
 Chapter 11 - Member Access Control
----------------------------------
================================================================================
 Chapter 12 - Special Member functions
---------------------------------------
Work Group:     Core
Issue Number:   138 (WMM.89)
Title:          When are default ctor default args evaluated for array
                elements?
Section:        12.6 [class.init] Initialization
Status:         editorial
Description:
        From Mike Miller's list of issues.
        WMM.89. Are default constructor arguments evaluated for each element
        of an array or just once for the entire array?
                int count = 0;
                class T {
                        int i;
                public:
                        T ( int j = count++ ) : i ( j ) {}
                        ~T () { printf ( "%d,%d\n", i, count ); }
                };
                T arrayOfTs[ 4 ];
        Should this produce the output :-
                0,4
                1,4
                2,4
                3,4
        or should it produce :-
                0,1
                0,1
                0,1
                0,1
Proposed Resolution:
       8.3.6[dcl.fct.default] para 9 says:
       "Default arguments are evaluated at each point of call before the
        entry into a function."
       This should also be true if the function call is implicit.
       That is, the test case above should produce the first output
       suggested above.

       Para 9 should be clarified to say that it also applies to functions
        that are implicitly called.
Resolution:
       At the Stockholm meeting, the core 2 WG decided to handle this issue
       as an editorial issue.
Requestor:      Mike Miller / Martin O'Riordan
Owner:          Josee Lajoie (Object Model)
Emails:
        core-668
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:  687
Title:         The WP prohobits the copy assignment of virtual base classes
```

to behave like the copy constructor
Section: 12.8 [class.copy]
Status: active
Description:
The ARM specified:
"Objects representing virtual base classes will be assigned only once by a generated assignment operator."

This restriction has been removed.
The current WP says in 12.8 para 13:
"The direct base classes of X are assigned first, in the order of their declaration in the base-specifier-list, and then the immediate nonstatic data members of X are assigned, in the order in which they were declared in the class definition.
[...]
It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitlys-defined copy assignment operator."

The new specification does not allow the copy constructor ordering.
Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (Object Model)
Emails:
Papers: 96-0107/N0925

. . . . . .

==========

Chapter 13 - Overloading

Work Group: Core
Issue Number: 662
Title: Do cv-qualifiers on the class object influence the operator() called?
Section: 13.3.1.1.2 [over.call.object]
Status: active
Description:
Should this be unambiguous?

```
typedef int (*pf)(char);
int foo(char);

struct S {
   operator pf() const { return c1; }
   operator pf() volatile { return c2; }
};
void f() {
    volatile S vs;
    vs('a');
}
```

If so, paragraph 2 needs to be changed to only allow conversion functions whose cv-qualifiers are at least as qualified as the expression's qualifiers.
Resolution:
Requestor:
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 684
Title: The ranking for implicit conversion sequences for pointer types should take into account qualification conversions in 4.4.
Section: 13.3.3.2 [over.ics.rank]
Status: active

Description:

Section 13.3.3.2 [over.ics.rank] says:

Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules apply:

- Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if

[...]

- S1 and S2 differ only in their qualification conversion and they yield types identical except for cv-qualifiers and S2 adds all the cv-qualifiers that S1 adds (and in the same places) and S2 adds yet more cv-qualifiers than S1, or if not that,

[...]

This may predate the Koenig & Smith papers on safe cv-qualification conversions in multi-level pointer and reference types. Shouldn't the ranking be based on whether one type can safely be converted into the other? Of course that involves more than just "more qualifiers".

Proposed Resolution:

I suggest the following fix for the last paragraph:

- S1 and S2 differ only in their qualification conversion and they yield types identical except for cv-qualifiers and there is an implicit qualification [conv.qual] from S1 to S2, or if not that,

Resolution:

Requestor: Bill Gibbons

Owner: Steve Adamczyk (Type Conversions)

Emails: core-6996

Papers:

. . . . . .

Work Group: Core

Issue Number: 685

Title: What is the ranking of a user-defined conversion that combines a pointer conversion with casting away cv-qualifiers?

Section: 13.3.3.2 [over.ics.rank]

Status: active

Description:

5.4 para 5 says:

The conversions performed by
-- a const_cast (_expr.const.cast_),
-- a static_cast (_expr.static.cast_),
-- a static_cast followed by a const_cast,
-- a reinterpret_cast (_expr.reinterpret.cast_), or
-- a reinterpret_cast followed by a const_cast,
can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply.

This means that this code is well-formed:

```
struct A {
        operator const char *();
} a;

main () {
                // const_cast<char *>(static_cast<const char*>(a))
        char *p = (char *) a;
```

```
        }

        In which case the overloading rules in chapter 13 need to describe
        what happens in this case:

        struct A {
                operator const char *();
                operator const volatile char *();
        } a;

        main () {
                char *p = (char *) a;
        }
Resolution:
Requestor:      Jason Merrill
Owner:          Steve Adamczyk (Type Conversions)
Emails:         core-7023
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   682
Title:         operator ?: and operands of enumeration types
Section:        13.6 [over.built]
Status:         active
Description:
       The type of a conditional expression choosing between two enums of
       the same type was changed in the May WP from that enum type to the
       integral type it promotes to, breaking code.  I propose changing
       paragraph 27 of 13.6 [over.built] from

       27 For every type T, where T is a pointer or pointer-to-member type,
          there exist candidate operator functions of the form
                 T       operator?(bool, T, T);

       to

       27 For every type T, where T is an enumeration, pointer or
          pointer-to-member type, there exist candidate operator functions
          of the form
                 T       operator?(bool, T, T);

       ----------
       Should the following testcase be ambiguous?

         const char c;
         enum E { a } e;
         bool b;

         main ()
         {
           return b ? c : e;
         }

       The builtin candidates are:
           operator ?(bool, const char &, const char &)
            operator ?(bool, int, int)
Resolution:
Requestor:     Jason Merrill
Owner:          Steve Adamczyk (Type Conversions)
Emails:                core-6983, core-6987
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
================================================================================
 Chapter 14 - Templates
------------------------
Work Group:     Core
```

Issue Number: 676
Title: When is a template instantiated?
Section: 14.7.1 [templ.inst]
Status: active
Description:

14.7.1 para 3 says:
"If a class template for which a definition is in scope is used in a way that involves overload resolution, conversion to a base class, or pointer to member conversion, the template specialization is implicitly instantiated."

'In a way that involves overload resolution' is not very precise.

Consider the following case:

```
template <class T> class foo {
public:
   operator int();
};

void bar(int);
void bar(float);
void bar(foo<int>&);

void foo_bar(foo<int>& fi)
{
        bar(fi);
}
```

Is the template instantiated during overload resolution for the call to bar?

Suppose that bar(foo<int>&) isn't there, is the instantiation still required?

What about calls to friend functions:

```
    extern void foo(int&);
    template <class T> class X {
        friend void foo(X&);
    };
    void bar(X<int>& t) {
        foo(t); // is X<int> instantiated?
                // If not, does this call fail?
    }
```

The description in 14.7.1 should be improved to clarified these cases.

Resolution:
Requestor: Neal Gafter
Owner: Bill Gibbons/John Spicer (Templates)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 677
Title: Should the text on argument deduction be moved to a subclause discussing both function templates and class template partial specializations?
Section: 14.8.2 [temp.deduct]
Status: active
Description:

Template argument deduction is now used both for function templates and for class template partial specializations. The

text for temp.deduct should be moved out of the function template specializations subclause.

Here is the reorganization Bill Gibbons suggested in private email:

> 14.2 Names of template specializations (including functions)
> 14.3 Template arguments (including functions; cross-ref arg
>   deduction)
> ...
> 14.8 Template argument deduction
> 14.8.1 Deducing a template argument from an expression
> 14.8.2 Argument deduction for function calls
> 14.8.3 Argument deduction for partial specialization ordering
>
> 14.9 Function calls
> 14.9.1 Mixing explicit and deduced template arguments
> 14.9.2 Overload resolution
> 14.9.3 Overloading and template specializations

Resolution:
Requestor: Sean Corfield
Owner: Bill Gibbons/John Spicer (Templates)
Emails:
Papers:

. . . . . .

Chapter 15 - Exception Handling

Work Group: Core
Issue Number: 678
Title: Can the exception object created by a throw expression have array type?
Section: 15.1 [except.throw]
Status: active
Description:

```
try {
    int a[5];
    throw a;
}
catch (int (&array)[5]) { }
```

Does the handler catch the exception? Or is an array-to-pointer conversion applied to the operand of the throw expression, meaning that the exception thrown has type pointer to int and that the handler does not catch the exception?

15.1 para 3 refers to the subclause on function calls (5.2.2) and to its description of conversions on function call arguments to describe the conversions that apply to a throw expression.
5.2.2 says that whether the array-to-pointer conversion is applied to an argument in a function call depends on the type of the function parameter.
In the case of the throw expression, either the conversion is always performed or it is never performed, but I don't believe saying that it depends on the type of the handler makes any sense. I think this should be clearer in 15.1.

Resolution:
Requestor:
Owner: Bill Gibbons (Exceptions)
Emails:
Papers:

. . . . . .

Chapter 16 - Preprocessing Directives

Work Group: Core

Issue Number: 679
Title: "Shall" is used incorrectly in clause 16
Section: clause 16
Status: editorial
Description:
        John Spicer pointed out the following:

        > There are numerous uses of "shall" in clause 16 (much of which
        > came directly from the C standard). The problem is that
        > "shall" does not always mean the same thing in the two
        > documents (in only means the same thing when it appears in a
        > "constraint" in the C standard).
        >
        > It seems that someone should go though clause 16 and change
        > "shall" to the appropriate wording about undefined behavior.
        > If > this is not done, certain programs that are undefined in
        > C will become ill-formed in C++.
Resolution:
Requestor: John Spicer
Owner: Tom Plum (C Compatibility)
Emails:
        compat-324
Papers:
. . . . . .
==========

 Annex C - Compatibility

Work Group: Core
Issue Number: 680
Title: Annex C subclause C.1 is out of date
Section: C.1 [diff.c]
Status: editorial
Description:
        Jonathan Schilling wrote the following:

        The introduction to Annex C (Compatibility) and subclause C.1
        (Extensions) both look like they were quickly edited from the
        base document for use in the standard, but the edit missed some
        spots and left others making no sense ("... from the dialects of
        Classic C used up till now", "... since the 1985 version of this
        manual"). More attention is given to Classic C than is now
        necessary, and the new features list is very incomplete.

        The proposed rewrite of the introduction and subclause C.1 is
        below.

        An alternative course of action would be to drop C.1 altogether,
        but I think that once made accurate it serves a useful purpose.
Proposed Resolution:
        Replaced C.1 and C.1.1 with:

        Annex C (informative)
        Compatibility   [diff]

        This Annex summarizes the evolution of C++ and explains in
        detail the differences between C++ and ISO C, both in the
        language and in the standard library.

        With the exceptions listed in this Annex, programs that are both
        C++ and C have the same meaning in both languages. All
        differences between C++ and C can be diagnosed by an
        implementation, although converting programs between C++ and C
        may be subject to the vicissitudes of unspecified and undefined
        behavior.

C.1 Extensions [diff.c]

This subclause summarizes the major extensions to C provided by C++. Because C++ was originally based upon the C of the first edition of _The C Programming Language_, before C became an ISO standard, there was some parallel evolution between the two languages. This is noted here by the phrase "also in ISO C".

C.1.1 C++ features available in 1985 [diff.early]

This subclause summarizes the extensions to C provided by C++ by 1985, as described in the first edition of _The C++ Programming Language_:

< same feature list that's in current [diff.early] >

C.1.2 C++ features added 1985 - 1991 [diff.mid]

This subclause summarizes the major extensions to C++ between 1985 and 1991, as described in the second edition of _The C++ Programming Language_:

< same feature list that's in current [diff.c++], except:
take out "The bool type" (20)
take out the references to things being "moved to the anachronism subclause" (5, 8) >

C.1.3 C++ features added since 1991 [diff.late]

This subclause summarizes the major extensions to C++ since 1991, as described in this International Standard:

Universal character names ([lex.charset]), trigraphs ([lex.trigraph]), and operator keywords ([lex.key]).

The bool type; [basic.fundamental].

The wchar_t type; [basic.fundamental].

User-defined new and delete operators for arrays; [expr.new], [expr.delete].

Placement delete; [expr.new].

Run-time type identification, including dynamic_cast and typeid; [expr.dynamic.cast], [expr.typeid].

A new form for casts: static_cast ([expr.static.cast]), reinterpret_cast ([expr.reinterpret.cast]), and const_cast ([expr.const.cast]).

Declarations in tested conditions in if, switch, for, and while statements; [stmt.select], [stmt.iter].

Namespaces; [basic.namespace].

Class members can be declared mutable; [decl.stc].

The explicit keyword for providing non-converting constructors; [dcl.fct.spec].

Forward declaration of nested classes; [class.nest].

Static data member constants; [class.static.data].

Relaxation of the rule for return types of overriding functions; [class.virtual].

Overloading based on enumerations; [over.load].

Refinement of the template compilation model and addition of the export keyword; [temp].

The typename keyword in template parameters; [temp.param].

Default arguments for template type parameters; [temp.param].

Default arguments for template type parameters; [temp.param].

Explicit template argument specification in template function calls; [temp.arg.explicit].

Explicit template instantiation; [temp.explicit].

New syntax for template specialization; [temp.expl.spec].

Partial specialization of class templates; [temp.class.spec].

Member templates; [temp.mem].

Function try blocks; [except].

The uncaught_exception() function; [except.uncaught].

The C++ Standard library; [lib.library].

Resolution:
Requestor: Jonathan Schilling
Owner: Tom Plum (C compatibility)
Emails:
compat-352
Papers:
. . . . . .
Work Group: Core
Issue Number: 681
Title: The type of string literals is array of const char - this has implicitions for C compatibility and should be in Annex C
Section: C.2.1 [diff.lex]
Status: editorial
Description:
Jonathan Schilling wrote the following:
The WP changes for the motion at Stockholm to change the type of string literals didn't include anything for Annex C.2. Something is needed, since this represents a new incompatibility with C.
If no one has written up the new entry, I propose the attached.

Proposed Resolution:

C.2.1 Clause 2: lexical conventions [diff.lex]

(insert as paragraph 4)

Subclause 2.13.4

Change: Type of string literal is changed from array of char to array of const char, and type of wide string literal from array of wchar_t to array of const wchar_t.

Rationale: This improves the consistency of the C++ type system.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Syntactic transformation. The most common cases are handled by a new but deprecated standard conversion:

```
char* p = "abc";                // valid in C, deprecated in C++
char* q = expr ? "abc" : "de";  // valid in C, invalid in C++
```

How widely used: Common.

Resolution:

Requestor: Jonathan Schilling

Owner: Tom Plum (C Compatiblity)

Emails:

compat-350

Papers:

. . . . . .