This list contains the issues I received since the Nashua meeting (either through private email or because they were posted on the core reflector) and a few issues that where on the Core list of issues before the Nashua meeting but that have not been addressed by a working group.

The committee may decide to address some of these issues before the first C++ standard is published. The remaining issues will be passed along to the committee working on the 5-year revision of the standard.

---

Lexical Conventions
-------------------

2.10 [lex.name]:
849: Which names are reserved to implementations?

Annex E[extendid]:
891: The list of hexadecimal code for CJK Unified Ideographs seems incorrect

Name Look Up
-------------

3.4.1 [basic.lookup.unqual]:
850: How does name look up proceed in the parameter list of a friend function?

5.1 [expr.prim]:
855: ::name is not a qualified-id

5.3.1 [expr.unary.op]:
860: Is ptr->~T() a call to a destructor?

7.3.3 [namespace.udecl]:
863: Can the name introduced by a using-declaration be the same as the name of an entity already declared in that scope?

8.3 [dcl.meaning]:
887: Can an extern declaration refer to a qualified name?

8.4 [dcl.fct.def]:
865: What is the potential scope of a function parameter?

9 [class]:
869: Is a class name inserted in its own class scope considered a member name for the purpose of name look up?

Linkage
-------

7.1.5.1 [dcl.type.cv]:
862: A local name declared const does not have internal linkage

7.5 [dcl.link]:
864: Does extern "C" affect the linkage of function names with internal linkage?

Object / Memory Model
---------------------

3.6.1 [basic.start.main]:
851: Must a diagnostic be issued if main is called in a program?
3.6.3 [basic.start.term]
852: Should the destruction of array objects be inter-leaved with calls
to the functions registered with atexit?

5.3.4 [expr.new]
866: What arguments are passed to placement operator delete?

5.10 [expr.eq]
861: Should the WP say that \&x == \&y is false if x not same object as y?

8.5.1 [dcl.init.aggr]
868: description of aggregate initialization should refer to default
initialization

9.4.2 [class.static.data]
870: Is an error required if a static data member is used and not
defined?

9.5 [class.union]
871: Can a class with a constructor but with no default constructor
be a member of a union?

12.2 [class.temporary]
874: Clarify lifetime of temporary example
12.6.2 [class.base.init]
875: If a constructor has no ctor-initializer, but the class has a
const member, is the constructor definition ill-formed?

12.8 [class.copy]
876: The optimization that allows a copy of a class object to alias
another object is too permissive

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

Sequence Points/Execution Model
-------------------------------
1.8 [intro.execution]
848: What can be done in a signal handler?
694: List of full-expressions needed

Access
-----
11 [class.access]
872: How do access control apply to constructors/destructors implicitly
called for static data members?
873: How/when is access checked in default arguments of function
templates?
11.2 [class.access.base]
888: Can a class with a private virtual base class be derived from?
11.5 [class.protected]
752: When accessing a base class member, the qualification is not
ignored

Types
-----
3.9.1 [basic.fundamental]
853: Should typeid(void-expression) be allowed?

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?
776: Name look up in default argument expressions

Types Conversions / Function Overload Resolution
------------------------------------------------
4.2 [conv.array]
885: Can array rvalues decay to pointers?
4.8 [conv.double]:
    712: Should the result value of a floating-point conversion be implementation-defined?
4.10 [conv.ptr]:
    854: Must a null pointer constant be an rvalue of integer type of value 0?
5.2.9 [expr.static.cast]:
    857: When can temporaries created by cast expressions be eliminated?
5.2.10 [expr.reinterpret.cast]:
    858: Can an expression of any type be cast to its own type using a reinterpret_cast?
    859: When can a pointer to member function be used to call a virtual function with a covariant return type?
8.5 [dcl.init]:
    866: cv-qualifiers and type conversions
    867: copy constructors do not have parameters of derived class type
13.3 [over.match]:
    877: 13.3.1.6 isn't about binding to a temporary
13.3.3.1 [over.best.ics]:
    733: Implicit conversion sequences and scalar types
13.3.3.2 [over.ics.rank]:
    779: identity conversion is preferred over lvalue-to-rvalue conversion
13.6 [over.built]:
    889: pseudo prototypes for built-in operators and operands of enumeration types need fine tuning

Expressions
-----------
5 [expr]:
    748: Should we say that operator precedence is derived from the syntax?
5.6 [expr.mul]:
    719: Is unsigned arithmetic modulo 2^N for multiplication as well?

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

RTTI
----
5.2.8 [expr.typeid]:
    856: Should the WP mention the type extended_type_info?

Templates
---------
3.5 [basic.link]:
    771: When are two different function template declarations with the same template name ill-formed or valid overloads?
14 [temp]:
    878: Can a template declaration not followed by a definition specify export?
14.2 [temp.names]:
    883: Can "template" be used to specify that an unqualified function name refers to a template specialization?
14.3 [temp.arg]:
    879: What conversions can apply to a template argument to bring it to the type of the corresponding nontype template parameter?
14.5.3 [temp.friend]:
    880: When does a friend declaration refer to a global function or to a template instantiation?
    890: Clarification of the interaction of friend declarations and use of explicit template arguments
14.5.4 [temp.class.spec] and 14.7.3 [temp.expl.spec]:
881: What class-key can be used in declarations of specializations and partial specializations?
14.6 [temp.res]:
882: typename is not permitted in functional cast notation
14.6.3 [temp.nondep]:
884: no diagnostics required for semantics errors in template definitions

==========================================================================
===
Chapter 1 - General
---------------------
Work Group: Core
Issue Number: 848
Title: What can be done in a signal handler?
Section: 1.8 [intro.execution]
Status: active
Description:
[Erwin Unruh:] Throwing an exception from within a signal handler should be undefined. All you can portably do within a handler is to set a global flag of type "volatile sig_atomic_t".

[Greg Colvin:] The C standard allows a signal handler to call signal(), and in some cases abort(), exit(), and longjmp().

The C++ draft does say the following in 1.8 para 10: "When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects modified after the preceding sequence point are indeterminate during the execution of the signal handler, and the value of any object not of volatile sig_atomic_t that is modified by the handler becomes undefined."

This seems less restrictive than the C standard, which allows undefined behavior if a signal handler "refers to any object of static storage duration other than by assigning a value to a static storage duration variable of type volatile sig_atomic_t".

[Erwin Unruh:] 1.8 para 10 should be deleted. It severely restricts optimizers. We all think that in the following code

```c
int a,b;
a = 7;
b = 5;
a = 9;
```

the first assignment is optimized away. 1.8 para says a compiler must put the assignment down because a signal handler might refer to a. I think this is an unacceptable situation with regard to C.

[Erwin Unruh's proposed resolution:] A function registered as a signal handler may only do what it is entitled to do in the C standard. A function which uses (even potentially) a language or library feature not in C will cause undefined behaviour.
[Note: This also covers very minor additions!]

[Example:
inline void f();          // inline is no C
void g(int) { if (0) f(); } // g uses a non-C feature

signal( SIGINT, &g );     // undefined behaviour

Although f is never called, activating a SIGINT causes
undefined behaviour. Note that using exception handling or
RTTI would most probably cause problems on some machines.

The result of this discussion should go into another
paragraph in section [lib.support.runtime] 18.7.

Resolution:
Requestor:      Greg Colvin/Erwin Unruh
Owner:          Steve Adamczyk (Sequence Points/Execution Model)
Emails:           
Papers:           

Work Group:     Core
Issue Number:   694
Title:          List of full-expressions needed
Section:        1.8 [intro.execution]
Status:         editorial
Description:
1.8p14: "certain contexts in C++ cause the evaluation of a
full-expression that results from a syntactic construct other
than expression"

Is it enumerated anywhere exactly what these contexts are?
Do the contexts themselves at least identify themselves as
surrogate full-expressions?

I didn't read the cited example (8.3.6) as thoroughly as I
might, but I didn't see anything there that explicitly said
"this is treated like a full-expression." Probably you could
make the case based on combining several passages together, but
if the other ones are like this, it would take some real
detective work to figure it out. If someone knows what contexts
were intended here, even if something might be omitted, it would
be an improvement to make it explicit, either here or in the
various contexts.

Steve Adamczyk:
> I looked at the wording and I agree it could be clearer. At
> the least we should make normative the idea that when a
> construct is implemented by an implicit function call, the
> entire function call is considered a full expression. 3.2p2
> may be useful as a list of implicit references.

Resolution:
Requestor:      Mike Miller
Owner:          Steve Adamczyk (Sequence Points)
Emails:           
Papers:           

Chapter 2 - Lexical Conventions

Work Group:     Core
Issue Number:   849
Title:          Which names are reserved to implementations?
Section:        2.10 [lex.name]
Regarding names that are reserved for C++ implementations, Sections 2.10 and 17.3.3.1.2 both say that identifiers containing a double underscore (__) or beginning with an underscore and an upper-case letter are reserved for use by C++ implementations and standard libraries.

Section 17.3.3.1.2 also says the following:
--Each name that begins with an underscore is reserved to the implementation for use as a name with file scope or within the namespace std in the ordinary name space.

This is missing from 2.10. I assume the wording in 17.3.3.1.2 takes precedence?

2.10 should be changed to just reference 17.3.3.1.2.

struct A {
    typedef int AT;
    void foo(AT);
};
struct B {
    typedef int BT;
    friend void A::foo(AT); // does name lookup find AT?
    friend void A::foo(BT); // does name lookup find BT?
};

3.4.1 is not clear describing how the scopes are searched for the parameter list of a friend function declaration when the friend function is a member function of another class. i.e. Is the scope of B ever considered?

3.6.1 para 3 says:
"The function main shall not be called from within a program."

Does 'call' mean function call in the program source or does it refer to call during the execution of the program? The "shall not" phrase can mean either that a diagnostic is required or that violation results in undefined behaviour depending on which one of these options the term 'call' refers to.

1.3 [intro.compliance] para 5 says:
"Whenever this International Standard places a requirement on the execution of a program (that is, the values of data that are used as part of program execution) and the data encountered during execution do not meet that requirement, the behavior of the program is undefined and this International Standard places no requirements at all on the behavior of the program."

Proposed Resolution:
A diagnostic is required. "call" refers to a source code construct.
Maybe the sentence should be rewritten as follows to make the requirement explicit:
"A program shall not contain a call to the function main."

Resolution:
Requestor: Steve Clamage/Fergus Henderson
Owner: Josee Lajoie (Object Model)
Emails: 
Papers: 

Work Group: Core
Issue Number: 852
Title: Should the destruction of array objects be inter-leaved with calls to the functions registered with atexit?
Section: 3.6.3 [basic.start.term]
Status: active
Description:
What is the defined order of atexit-registered function calls in the following program:

C f() { atexit(&func1); }
C g() { atexit(&func2); }
C x[] = { f(), g() };

3.6.3 para 3 says:
"If a function is registered with atexit (see <cstdlib>, __lib.support.start.term_) then following the call to exit, any objects with static storage duration initialized prior to the registration of that function will not be destroyed until the registered function is called from the termination process and has completed. For an object with static storage duration constructed after a function is registered with atexit, then following the call to exit, the registered function is not called until the execution of the object's destructor has completed."

The current draft (3.6.3) indicates that, upon termination, atexit will call registered functions in the example above in the following order:

Destructor for x[1]
func2
Destructor for x[0]
func1
This result is inconsistent with the behaviour of the following slightly different program:

```c
C f() { static C local1; }
C g() { static C local2; }
C x[] = { f(), g() };
```

The last sentence in 3.6.3 paragraph 1 says:

"For an object of array or class type, all subobjects of that object are destroyed before any local object with static storage duration initialized during the construction of the subobjects is destroyed."

This mandates that destructor be called in the following order:

- Destructor for x[1]
- Destructor for x[0]
- Destructor for local2
- Destructor for local1

Should the ordering for these two programs be consistent? Shouldn't the first program call functions registered with atexit in the following order?

- Destructor for x[1]
- Destructor for x[0]
- func2
- func1

Proposed Resolution:

[Josee: I remember a discussion where members of the committee didn't want to require that an implementation remember the order of destruction of every array element separately. I think this should be the case both when destructors for local static variables are called and when functions registered with atexit are called.]

Resolution:

Requestor: Josee Lajoie (Object Model)
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

Work Group: Core
Issue Number: 853
Title: Should typeid(void-expression) be allowed?
Section: 3.9.1 [basic.fundamental]
Status: active

Description:

[Bill Gibbons, core-7398:]
The restriction on expressions of void type in 3.9.1/9:
"An expression of type void shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), or as a second or third operand of ?: (5.16)."

makes this code ill-formed:

```c
#include <typeinfo>
void f() { }
void g() {
    typeid(f()); // ill-formed
    typeid(void); // OK
}
```
Should expressions of type void be allowed as operands of typeid? (Note that they are already allowed as operands of ?:, so there is a precedent for allowing them.)

[Sean Corfield, core-7404:]
Should we consider this as part of the issue to relax uses of void? This just seems to be 'yet another bug' in the handling of void (that's how I view the 'unnecessary' restrictions since they get in the way of writing templates).

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)

Chapter 4 - Standard Conversions

Work Group: Core
Issue Number: 885
Title: Can array rvalues decay to pointers?
Section: 4.2 [conv.array]
Status: active

Description:

Somewhere in the Kona edits, 4.2 [conv.array] paragraph 1 changed from saying "An lvalue or rvalue of array type ... can be converted to an rvalue of type pointer ..." to saying "An lvalue of array type ..." etc.

96-0178/N0996 has "lvalue or rvalue", and 96-0219R1/N1037 (the CD2 version with change bars) has "lvalue".

The consequence of this change is that examples like the following are currently ill-formed:

```cpp
struct A {
    int arr[5];
};
A f();
void g() {
    f().arr[3] = 1;
}
```

This sort of example was discussed by the committee and we agreed it should be valid.

The status quo makes C++ more like C. However, it is different from what the ARM says and what the committee decided.

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

Work Group: Core
Issue Number: 712
Title: Should the result value of a floating-point conversion be implementation-defined?
Section: 4.8 [conv.double]
Status: active
Description:
4.8 says for floating-point conversions:
   If the [floating-point] source value is between two adjacent
   [floating-point] destination values, the result of the
   conversion is an unspecified choice of either of those values.

yet 2.13.3 says for floating-point literals:
   the result is either the nearest representable value, or the
   larger or smaller representable value immediately adjacent to
   the nearest representable value, chosen in an
   implementation-defined manner.

Why not say "implementation-defined" for conversions too?

This also applies to the integral to floating conversions
described in 4.9 [conv.fpint].

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

Work Group:     Core
Issue Number:   854
Title:          Must a null pointer constant be an rvalue of integer
type of value 0?
Section:        4.10 [conv.ptr]
Status:         editorial
Description:
4.10 para 1:
"An integral constant expression (5.19) rvalue of integer type
that evaluates to zero (called a null pointer constant) can be
converted to a pointer type."

Is this supposed to be a definition for the "null pointer
constant"? It doesn't really say that. If an A is a D, it
does not mean that other things can't also be Ds. I don't find
any other definition of "null pointer constant".

Could an implementation define NULL to be a zero value of a
magic internal compiler type that was compatible with all
pointer types but not with integral types? In that case, given

    void f(int);
    void f(char*);

the expression f(NULL) would call f(char*), but with a usual
implementation would call f(int). In addition, usual
implementations would allow

    int i = 2 + NULL;

but the hypothetical implementation would flag it as an error.

Proposed Resolution:
The sentence in 4.10 is intended to define the term "null
pointer constant".

The first two phrases of 4.10 para 1 could be reversed to show
the intent better:
"A null pointer constant, which is an integral constant
expression (5.19) rvalue of integer type that evaluates to zero,
can be converted to a pointer type."
According to the definition of NULL in chapter 18, NULL must be a null pointer constant.

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

Chapter 5 - Expressions
-------------------------
Work Group: Core
Issue Number: 748
Title: Should we say that operator precedence is derived from the syntax?
Section: 5[expr]
Status: editorial
Description:
  para 4:
  "Except where noted, the order of evaluation of operands of individual operators and subexpressions of individual expressions, and the order in which side effects take place, is unspecified."

"Except where noted"
Should we say that operator precedence is derived from the syntax? The C syntax says this in a footnote. (Footnote 35).

Resolution:
Requestor: 
Owner: Steve Adamczyk (Expressions)
Emails: 
Papers: 

Work Group: Core
Issue Number: 855
Title: ::name is not a qualified-id
Section: 5.1 [expr.prim]
Status: editorial
Description:
The term "qualified-id" is sometimes used in the WP to designate a name solely prefixed by the :: operator. However, the grammar does not allow a qualified-id to be preceded by a leading ::. This should be clarified.

For example, 3.4.4 para 1 says:
"The class-name or enum-name in the elaborated-type-specifier may either be a simple identifier or be a qualified-id."
The above does not allow:
class ::B ....
to refer to a global class name.

Resolution:
Requestor: 
Owner: Josee Lajoie (Name Look Up)
Emails: 
Papers: 

Work Group: Core
Issue Number: 856
Title: Should the WP mention the type extended_type_info?
Section: 5.2.8 [expr.typeid]

"Except where noted"
Status: active

Description:
Someone asked on the reflector:
> The extended_type_info is no longer mentioned in the draft.
> Is there a conforming way to provide extended type information now?

Bill Gibbons answered the following:
> The working paper should say that typeid yields an lvalue
> referring to a type_info object or an object of type derived from type_info.
> The name "extended_type_info" should probably still appear in a note, but of course it is totally non-normative.

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

Work Group: Core
Issue Number: 857
Title: When can temporaries created by cast expressions be eliminated?
Section: 5.2.9 [expr.static.cast]
Status: active

Description:
S s;
(S)s; // Must this cast expression create a temporary of type S? // Even though s has type S already?

A more interesting example:
class S {
    int i;
    public:
        S foo() { i = 1; return *this; }
};

S s;
(S(s)).foo(); // Does this change the value of s.i?

5.2.9 para 2 says that a temporary is created for S(s). Is the implementation allowed to eliminate this temporary?

Resolution:
Requestor: Josee Lajoie
Owner: Steve Adamczyk (Type Conversions)

Work Group: Core
Issue Number: 858
Title: Can an expression of any type be cast to its own type using a reinterpret_cast?
Section: 5.2.10 [expr.reinterpret.cast]
Status: active

Description:
This complements issue 796.

5.2.10 para 2 says:
"Any expression may be cast to its own type using a reinterpret_cast operator."
There are two problematic cases with this scenario:
(1) Array types. It's a little weird to be able to cast an lvalue array to its own (array) type.
(2) Class types. Maybe it's okay to cast a class expression to its own type, but what are the semantics? Is a copy made? If so, presumably the copy constructor is not called. (?)

Both could be resolved by saying that the reinterpret_cast does nothing in that case, i.e., it's like a set of parentheses, but even there, one would have to be careful to indicate whether the expression is forced to an rvalue.

Proposed Resolution:
All things considered, it seems it would be better to make a change here like the one recommended for const_cast (Issue 796).

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

Work Group: Core
Issue Number: 859
Title: When can a pointer to member function be used to call a virtual function with a covariant return type?
Section: 5.2.10 [expr.reinterpret.cast]
Status: active
Description:
5.2.10 para 10 says: "Calling a member function through a pointer to member that represents a function type that differs from the function type specified on the member function definition results in undefined behavior, except when calling a virtual function whose function type differs from the function type of the pointer to member only as permitted by the rules for overriding virtual functions."

Does the above intend to allow the following:
struct X { }
struct Y: X { }

struct A {
    virtual X* f();
};
struct B: A {
    virtual Y* f();
};

X* (A::*pm)() = &A::*f;
Y* (B::*pm2)();
pm2 = reinterpret_cast<Y*(B::*)> (pm);

B b;
b.*pm2(); // is this supposed to be well formed?

If so, then the example should be added to the WP.
Resolution:
Requestor: Steve Adamczyk (Type Conversions)
Emails: 
Papers: 

...
Work Group: Core
Issue Number: 860
Title: Is ptr->~T() a call to a destructor?
Section: 5.3.1 [expr.unary.op]
Status: active
Description:
5.3.1 para 9:
"There is an ambiguity in the unary-expression ~X(), where X is a class-name. The ambiguity is resolved in favor of treating the ~ as a unary complement rather than treating ~T as referring to a destructor."

This seems to contradict 12.4 [class.dtor] para 12:
struct B {
    virtual ~B() { }
};
struct D : B {
    ~D() { }
};

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;
...
B_ptr->~B(); // calls D's destructor ??complement op??
B_ptr->~B_alias(); // calls D's destructor ??complement op??
...

Should 5.3.1 para 9 say that it only applies if the unary operator is not part of a postfix expression?

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

Work Group: Core
Issue Number: 886
Title: What arguments are passed to placement operator delete?
Section: 5.3.4 [expr.new]
Status: editorial
Description:
In [expr.new] (5.3.4), para 19 says:
"A declaration of placement operator delete matches the declaration of a placement operator new when it has the same number of parameters and ... all parameter type except the first are identical."

but para 20 says:
"If placement operator delete is called, it is passed the same arguments as were passed to placement operator new."

Same arguments, even for the first parameter?
Proposed Resolution:
Para 20 should say same _additional arguments_.
Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

...
According to 3.9.1/3, unsigned arithmetic is always modulo \(2^N\). For addition and subtraction this is easy to remember, but for multiplication the rule should probably be repeated here since it is less obvious.

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Expressions)

---

The relational operators (5.9) produce unspecified results when comparing addresses of unrelated objects. I'm using "unrelated" to mean that neither is a subobject of the other, neither is a subobject of the same object, and they are not both part of the same array. I also am referring to the built-in address-of operator, not an user-defined operator. Prototypical example:

```cpp
void f() {
    int x, y;
    bool b = (&x <= &y); // unspecified result
    ...
}
```

So far, so good.

Section 5.10 says the equality operators have the same rules as the relationals, but goes on to provide some cases when addresses must compare equal. Conspicuously absent is any statement about equality of addresses of unrelated objects.

```cpp
bool b = (&x == &y); // also unspecified!
```

I thought I remembered a guarantee that \((&x!=&y)\) in early drafts of the C standard, but the current C standard does not make the guarantee. (So far as I can tell. Fergus Henderson thinks the C standard is open to interpretation on that point but I don't see why. It is irrelevant in any case, since the C++ standard could tighten the requirement without causing any problems. Surely there is no program that depends on x and y having addresses that compare equal!)

It seems like a peculiar omission, since we generally expect the address of an object to determine its identity. In particular, I think much of STL relies on the proposition:

\((&x==&y)\) if and only if x and y are the same object

Was the "only if" part of the proposition deliberately left out, and if so, can someone explain why?
Resolution:
Requestor:      Steve Clamage
Owner:          Josee Lajoie (Object Model)
Emails:         
Papers:         

==========================================================================
===
Chapter 7 - Declarations
--------------------------
Work Group:     Core
Issue Number:   862
Title:          A local name declared const does not have internal linkage
Section:        7.1.5.1 [dcl.type.cv]
Status:         editorial
Description:
7.1.1 para 6 says:
"A name declared in a namespace scope without a storage-class-specifier has external linkage unless it has internal linkage because of a previous declaration and provided it is not declared const. Objects declared const and not explicitly declared extern have internal linkage."

but 7.1.5.1 para 2 misses the `namespace scope' part (i.e., it forgets about objects with no linkage, I think):

"An object declared with a const-qualified type has internal linkage unless it is explicitly declared extern or unless it was previously declared to have external linkage.[...]

Proposed Resolution:
7.1.5.1 should say:
"An object declared in a namespace scope ...".

Resolution:
Requestor:      David Vandevoorde
Owner:          Josee Lajoie (Linkage)
Emails:         
Papers:         

Work Group:     Core
Issue Number:   863
Title:          Can the name introduced by a using-declaration be the same as the name of an entity already declared in that scope?
Section:        7.3.3 [namespace.udecl]
Status:         active
Description:
7.3.3/1 says:
"A name specified in a using-declaration in a class or namespace scope shall not already be a member of that scope."

7.3.3/10 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
-- exactly one declaration shall declare a class name or enumeration name and other declarations shall all refer to the same entity or all refer to functions; in this case the class name or enumeration name is hidden."

7.3.3 para 1 should probably be changed to reflect what
7.3.3 para 10 says.

[Bill Gibbons also mentions]

There is a note at the end of 7.3.3/13:

[Note: two using-declarations may introduce functions with the same name and the same parameter types. A call to such a function is ill-formed unless name look up can unambiguously select the function to be called (because the function name is qualified by its class name, for example).]

The note in 7.3.3/12 says the same thing about namespace and block scope.

Why must the ambiguity be resolved by name lookup, and not by overload resolution? For example:

```c
namespace A {
  void f(int);
  void f(long);
}
namespace B {
  void f(long);
  void f(double);
}
namespace C {
  using A::f;
  using B::f;
  void g() {
    f(123);  // ill-formed ?
  }
}
```

As written, the WP makes this ill-formed because there are two different functions "f(long)" at the point of the call. Of course overload resolution would not be ambiguous.

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

Work Group:   Core
Issue Number: 864
Title:        Does extern "C" affect the linkage of function names with internal linkage?
Section:      7.5 [dcl.link]
Status:       active
Description:
7.5 para 6 says the following:
"At most one of a set of overloaded functions with a particular name can have C linkage."

Does this apply to static functions as well?
For example, is the following well-formed?

```c
extern "C" {
  static void f(int) {}
  static void f(float) {}
};
```

Can a function with internal linkage "have C linkage" at all
(assuming that phrase means "has extern "C" linkage"), for how can a function be extern "C" if it's not extern?

The function *type* can have extern "C" linkage -- but I think that's independent of the linkage of the function *name*. It should be perfectly reasonable to say, in the example above, that extern "C" applies only to the types of f(int) and f(float), not to the function names, and that the rule in 7.5 para 6 doesn't apply.

Mike's proposed resolution:
The extern "C" linkage specification applies only to the type of functions with internal linkage, and therefore some of the rules that have to do with name overloading don't apply.

Resolution:
Requestor: Mike Anderson
Owner: Josee Lajoie (Linkage)
Emails:
Papers: 

Chapter 8 - Declarators
------------------------
Work Group: Core
Issue Number: 887
Title: Can an extern declaration refer to a qualified name?
Section: 8.3 [dcl.meaning]
Status: active
Description:
8.3 para 1 says:
"A declarator-id shall not be qualified except for the definition of a member function (_class.mfct_) or static data member (_class.static_) or nested class (_class.nest_) outside of its class, the definition or explicit instantiation of a function, variable or class member of a namespace outside of its namespace, the definition of a previously declared explicit specialization outside of its namespace, or the declaration of a friend function that is a member of another class or namespace (_class.friend_)."

This does not allow the following. Should id be allowed?

```cpp
namespace X {
    void f();
    void g() {
        extern void X::f(); // should this be allowed?
    }
}
```

Resolution:
Requestor: Josee Lajoie (Name Look Up)
Owner: 
Emails:
Papers: 

Work Group: Core
Issue Number: 776
Title: Name look up in default argument expressions
Section: 8.3.6 [dcl.fct.default]
Status: active
Description:
para 5 says:
"The names in the expression are bound, and the semantic
At the point of declaration of what? the function or the parameter?

In this example, to which 'f' does the default argument refers? ::f or N::f?

typedef int (*PF)();
int f(PF);
namespace N {
int f(PF p = &f);
}

Resolution:
Requestor:          Steve Adamczyk (Default Arguments)

Owner:          Steve Adamczyk (Default Arguments)

Emails:
Papers:

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:
At the Hawaii meeting, the core WG agreed that the example above was an error and suggested that this be clarified in the WP as an editorial matter.

Requestor:      Bill Gibbons
Owner:          Steve Adamczyk (Default Arguments)

Emails:
Papers:

Work Group:     Core
Subclause 3.3.2 paragraph 2 reads:
"The potential scope of a function parameter name in a function
definition (_dcl.fct.def_) begins at its point of declaration.
If the function has a function try-block the potential scope
of a parameter ends at the end of the last associated handler,
else it ends at the end of the outermost block of the function
definition. A parameter name shall not be redeclared in the
outermost block of the function definition nor in the
outermost block of any handler associated with a function
try-block."

But subclause 8.4 paragraph 2 reads:
"The parameters are in the scope of the outermost block of the
function-body."

I presume the latter sentence should simply be removed. The
following shows why it makes a difference.

    const int n = 1;
    void f(int n,
           int m = n); // which n?

Resolution:
Requestor: Neal Gafter
Owner: Josee Lajoie (Name Lookup)

---

Work Group: Core
Issue Number: 866
Title: cv-qualifiers and type conversions
Section: 8.5 [dcl.init]
Status: active

Description:
1. The description of copy-initialization in 8.5 para 14 says:
   "The user-defined conversion so selected is called to
   convert the initializer expression into a temporary, whose
type is the type returned by the call of the user-defined
conversion function, with the cv-qualifiers of the
destination type."

Why must the temporary have the cv-qualifiers of the
destination type? Shouldn't the cv-qualifiers of the
conversion function dictate the cv-qualifiers of the
temporary? For example,

    struct A {
        A(A&);  
    };
    struct B : A { };

    struct C {
        operator B&();
    };

    C c;
    const A a = c; // allowed?

The temporary created with the conversion function is an
lvalue of type B.

If the temporary must have the cv-qualifiers of the destination type (i.e. const) then the copy-constructor for A cannot be called to create the object of type A from the lvalue of type const B.

If the temporary has the cv-qualifiers of the result type of the conversion function, then the copy-constructor for A can be called to create the object of type A from the lvalue of type const B.

This last outcome seems more appropriate.

2. the treatment of cv-qualifiers in 13.3.1.4 is also puzzling:

"Assuming that cv1 T is the type of the object being initialized...

--When the type of the initializer expression is a class type "cv S", the conversion functions of S and its base classes are considered. Those that are not hidden within S and yield type "cv2 T2", where T2 is the same type as T or is a derived class thereof, and where cv2 is the same cv-qualification as, or lesser cv-qualification than, cv1, are candidate functions."

Why must the result of the conversion function be equally or less cv-qualified than the object initialized? Shouldn't the cv-qualification of the copy-constructor parameter determine whether the cv-qualification on the result of the conversion function is appropriate or not? For example:

```cpp
struct A {
    A(const A&);
};
struct B : A {
};
struct C {
    operator const B&();
};
C c;
A a = c;
```

The conversion function returns an lvalue of type const B. Shouldn't this be allowed since the copy constructor for class A accepts arguments that are const lvalues?

3. Is sub-clause 13.3.1.5 only for the initialization of non-class objects?

The wording in this clause makes this somewhat confusing. The bullet in paragraph 1 says:

"Conversion functions that return a nonclass type "cv2 T" are considered to yield cv-unqualified T for this process of selecting candidate functions."

All the conversion functions considered in this section return "nonclass type". In which case, all the bits about cv-qualifiers are not necessary (and are somewhat confusing).
Work Group: Core
Issue Number: 867
Title: copy constructors do not have parameters of derived class type
Section: 8.5 [dcl.init]
Status: editorial
Description:
Editorial Issue:
In the definition of copy-initialization in section 8.5 para 14, footnote 87 says:
"Because the type of the temporary is the same as the type of the object being initialized, or is a derived class thereof, this direct-initialization, if well-formed, will use a copy constructor (_class.copy_) to copy the temporary."

The term "copy constructor" is not used correctly here. Direct initialization considers not only the copy constructor but all constructors such that a constructor that accepts a derived class type would be preferred in this situation:

```cpp
struct D;
struct B {
  B(const B&);
  B(const D&);
};
struct D {};

struct X {
  operator D();
};

B b = x;
```

Isn't the temporary created by this copy-initialization of type D (i.e. "the type returned by the call of the user-defined conversion function")? Shouldn't B(const D&) be selected? B(const D&) is not a copy constructor.

Resolution:
Requestor: Josee Lajoie
Owner: Steve Adamczyk (Type Conversions)

---

Work Group: Core
Issue Number: 868
Title: description of aggregate initialization should refer to default initialization
Section: 8.5.1 [dcl.init.aggr]
Status: editorial
Description:
8.5.1 para 7 says that "each member not explicitly initialized shall be initialized with a value of the form T()".

This should instead say that the member should be default-initialized. This matters when the type T is an array, because you can't write T() for an array type T.

If this change is made, paragraph 8 (about leaving a reference uninitialized) can be made a note, because default-initialization of a reference is ill-formed ([dcl.init] para 5).
Resolution:
Requestor: Steve Adamczyk
Owner: Josee Lajoie (Object Model)

---

Chapter 9 - Classes

---

Work Group: Core
Issue Number: 869
Title: Is a class name inserted in its own class scope considered a member name for the purpose of name look up?
Section: 9 [class]
Status: editorial

Description:

class A {
    class X {
        class A {
            class Y : ::A {
                A a; // base class A or X::A?
            }
        }
    }
}

The answer to this is almost clear. Members of base class members are found before names declared in containing classes (3.6.1p7), and the class name is inserted into the class (9p2), so I would say that the reference to A must be the base class.

What is not clear is whether the insertion of the class name is considered to be a "member" for the purpose of 3.6.1p7. I think it's intended to be, but the terminology is not consistent, probably because the concept of "membership" as applying to other than data members and member functions evolved over time.

Resolution:
Requestor: Mike Miller
Owner: Josee Lajoie (Name Lookup)

---

Work Group: Core
Issue Number: 870
Title: Is an error required if a static data member is used and not defined?
Section: 9.4.2 [class.static.data]
Status: editorial

Description:

9.4.2 para 2 says:
"A definition shall be provided for the static data member if it is used (3.2) in a program."

9.4.2 para 5 says:
"There shall be exactly one definition of a static data member that is used in a program; no diagnostic is required;

Para 2 does not say: "no diagnostic required".
The duplication and difference between these two sentences is a bad thing. The sentence in 9.4.2 para 2 should be removed.

Resolution:
Requestor: Josee Lajoie (Object Model)

---

Chapter 11 - Member Access Control
------------------------------------

Work Group: Core
Issue Number: 872
Title: How do access control apply to constructors/destructors implicitly called for static data members?
Section: 11 [class.access]
Status: active
Description:
Here's a question that is being discussed in comp.std.c++ for which I don't find a clear answer in the draft.

```cpp
class C {   // has private constructor and destructor
friend class D;
    C();
    ~C();
};

class D {
public:
    static C c; // static member
};

C D::c; // can this be constructed, and if so, can it be // destroyed?
```
Members of D can create and destroy objects of type C because
the ctor and dtor are accessible. What about the static C
member of D? Is its construction and destruction in the scope
of D (accessible) or in global scope (inaccessible)? Where is
the answer defined in the draft?

Resolution:
Requestor:      Steve Clamage
Owner:          Steve Adamczyk (Access)
Emails:          
Papers:          

Work Group:     Core
Issue Number:   873
Title:          How/when is access checked in default arguments of
                function templates?
Section:        11 [class.access]
Status:         active
Description:
The proposed relaxation of default argument checking for
function templates (and presumably member functions of class
templates) informally given by Stroustrup in N1070/97-0032 is:

A default argument to a template function is checked only if
used.

In N1062R1/97-0024R1, Unruh proposes the following wording:

A default argument expression specialization is implicitly
instantiated only when the function specialization is
referenced in a context that requires the default argument
expression to exist.

The point of instantiation of a default argument expression
specialization is the same as that for the function. [Note:
Even if only some of the calls use a default argument, all
points of instantiation of the function can be used to
instantiate the default argument expression specialization.]

For access checking, the current working paper says, in 11/7:

The names in a default argument expression (8.3.6) are bound at
the point of declaration, and access is checked at that point
rather than at any points of use of the default argument
expression.

Obviously this would have to change. But some details are
missing.

In particular, there are two points about access checking which
should be made more clear. Here is my understanding of what is
intended by the proposal:

* Access checking is done relative to the original scope of the
default argument. Default arguments are treated as part of
the body of the function for access checking; the only
difference from the current rule is the deferred instantiation
(which implies that some currently ill-formed default
arguments are no longer ill-formed if they are never used.)

and:

* If no valid specialization could ever be generated for a
default argument, the program is ill-formed (no diagnostic required).

Examples:

```cpp
class A {
protected:
    typedef int Z;
    static int x;
};
template<class T> class B : T {
    void f(A::Z); // ill-formed, even if only instantiation is
                   // diagnostic is required
    void g()       // well-formed, because there is
                   // at least one instantiation B<A>::g
                   // in which the access is valid
        { int y = A::x; }
    void h(int y = A::x); // well-formed, same reason
};

class C {
protected:
    typedef int Z;
    static int x;
};
template<class T> class D { // ONLY DIFFERENCE IS NO BASE CLASS
    void f(C::Z); // ill-formed
                   // diagnostic is required
    void g()       // ill-formed, because there is
                   // no possible specialization
                   // in which the access is valid
        { int y = C::x; }
    void h(int y = C::x); // ill-formed, same reason
};
```

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Access)

```
Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Access)
```
testcase, EDG and HP accept it. Perhaps this case should be clarified by a note in the draft.

In short, it looks like a class with a virtual private base can't be derived from.

Resolution:
Requestor:    Jason Merrill
Owner:        Steve Adamczyk (Access)
Emails:
Papers:
.................................................................
Work Group:   Core
Issue Number: 752
Title:        When accessing a base class member, the qualification is not ignored
Section:      11.5[class.protected]
Status:       editorial
Description:
11.2 para 4 says:
"The access to a member is affected by the class in which the member is named. This naming class is the class in which the member name was looked up and found. [Note: this class can be explicit, e.g., when a qualified-id is used, or implicit, e.g., when a class member access operator (_expr.ref_) is used (including cases where an implicit this-" is added. If both a class member access operator and a qualified-id are used to name the member (as in p->T::m), the class naming the member is the class named by the nested-name-specifier of the qualified-id (that is, T). ]"

This is contradictory to the example in 11.5 para 1:

class B {
    protected:
        int i;
        static int j;
    }

class D1 : public B {
    
}:

class D2 : public B {
    friend void fr(B*,D1*,D2*);
    void mem(B*,D1*);
};
void fr(B* pb, D1* p1, D2* p2)
{
    p2->B::i = 4; // ok (access through a D2,
    // *** qualification ignored ***
}

According to 11.2 para 4, the qualification is not ignored.

Resolution:
Requestor:    Jason Merrill
Owner:        Steve Adamczyk (Access)
Emails:
Papers:
.................................................................
Description:

12.2 paragraph 5 example:
" class C {
   // ...
   public:
   C();
   C(int);
   friend const C& operator+(const C&, const C&); // problem
      ~C();
   
};
C obj1;
const C& cr = C(16)+C(23);
C obj2;

the expression C(16)+C(23) creates three temporaries. A first
temporary T1 to hold the result of the expression C(16), a
second temporary T2 to hold the result of the expression C(23),
and a third temporary T3 to hold the result of the addition of
these two expressions. The temporary T3 is then bound to the
reference cr."

Binding the result of the expression to "C const& cr" is a nice
example of a const reference to a temporary; however, the
function does not return a temporary, it returns a "C const&".
With the snapshot of the example given, it is very difficult
(impossible?) to determine where T3 came from. If the function
returns a "C" rather than a "C const&", everything makes sense.

Resolution:

Requestor: John Potter via Steve Clamage
Owner: Josee Lajoie (Object Model)

The CD is clear that the following:

struct A {
   ~A();
};

struct Y {
   Y() : d(0.0) {}
   A const a;
   double d;
};

is ill-formed because the mem-initializer-list for Y does not
include an initializer for `a' (which is a const non-POD class
without a user-declared default-ctor). [class.base.init]/4

However, if Y were defined as:
struct Y {
    Y() {}
    A const a;
};

then the answer is not clear: the rules for mem-initializer-lists do not apply since there is no mem-initializer-list.

Proposed Resolution:
The intention was that it be ill-formed. In the opening sentence of [class.base.init]/4, we should add "(including the case where there is no mem-initializer-list because the constructor definition has no ctor-initializer)".

Resolution:
Requestor:      David Vandevoorde
Owner:          Josee Lajoie (Object Model)

Work Group:     Core
Issue Number:   876
Title:          The optimization that allows a copy of a class object to alias another object is too permissive
Section:        12.8 [class.copy]
Status:         active
Description:
12.8 [class.copy] Paragraph 15.
A comment on comp.std.c++ said the following:
"This paragraph is fundamentally flawed and should either be removed or substantially reworked (preferably removed). The "optimisation" it describes allows the compiler to arbitrarily violate the basic semantic axiom that arguments passed by value are not modified."

Andrew Koenig replies:
> In c++std-core-7448, John Skaller discusses a potential problem with the rule that says, in effect, that for optimization purposes a compiler is allowed to assume that copy constructors copy their objects and that the original and the copy can be aliased if one of them is never used again.
>
> I think the problem can be summarized by saying that objects can bind resources, and even if an object is not used, the resource it binds might be. The kind of thing that might happen is
>
> Thing x = /* some value */;
> SubThing y = x.extract_portion();
> Thing z = x;
> z.clobber_portion();
> // now try to fetch the value of y
>
> If x is never used again, the compiler is entitled to alias z and x. However, if y actually refers to part of the storage that x used, clobbering z (which is an alias to x) might also clobber y.
>
> I can think of a few ways of dealing with this problem:
>
> 1. Acknowledge that the problem exists, but don't solve it.
>
> 2. Outlaw the optimization except in very restricted
circumstances.

3. Offer a way for class authors to say `Don't optimize'

I haven't decided whether or not I think (1) is a good idea,
but I don't think (2) is a good idea, unless we put a whole
lot of work into defining the cases. The reason is that the
optimization makes a tremendous difference in fairly common
cases like these:

class Point {
   // ...
   int x, y;
   // ...
   
   Point& operator+=(Point p) {
      x += p.x; y += p.y; return *this;
   }
   // ...
};

inline Point operator+(Point p, point q) {
   Point r = p;
   r += q;
   return r;
}

Perhaps the author should have used const Point& instead of
just Point, but not every does. Anyway, the optimization
allows the compiler to rewrite the parameters of operator+ to
avoid copying them, even if Point has an explicit copy
constructor. I'd hate to lose that.

On the other hand, I have a simple way of allowing for (3):
just say that if a class has an `explicit' copy constructor,
that means that the compiler is not allowed to optimize it
away (with the possible exception of the return value
optimization). I suspect that anyone who knows enough to
define classes that play aliasing games will know enough to
say `explicit'.

[Fergus Henderson, core-7469]
I suspect that at the time it [allowing the aliasing] was
considered, the committee may not have considered the
implications of the word "unused".

Just as we have "bitwise const" and "logical const", so we
can talk about "bitwise use" and "logical use". Which sort
of "use" does 12.8 para 15 refer to?

The optimization in question is reasonable if and only if
the original is subsequently logically unused. This has lead
some people (e.g. Pete Becker) to interpret the current text
as referring to logical use, and I suspect that many people
voting for the resolution may have been implicitly assuming
that use meant logical use.

However, if your machine does not have a "read the
programmer's mind" instruction, then logical use is not
computable. If the text is interpreted to mean logical use,
then the paragraph becomes non-normative waffle, because no
earthy compiler can take advantage of it.

So as I see it, the status quo is that the working paper is
ambiguous. If "use" was intended to mean "logical use", as I
suspect it may have been, then (due to problems that were not
noticed at the time) the text that was voted in turns out to
be useless, and so it should be deleted. If "use" was
intended to mean "bitwise use", as it generally does
elsewhere in the WP, then the text that was voted in is
useful, but breaks some programs that really ought to be
legal (and again, I suspect that these problems were not
really understood at the time it was voted in).

Given that this distinction between bitwise use and logical
use was not made clear at the time (please correct me if I'm
wrong), and given that the problems that the bitwise use
version causes were not made clear at the time (again, please
correct me if I'm wrong), I think that the committee ought to
reconsider this issue.

Resolution:
Requestor: John Skaller
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

Chapter 13 - Overloading
--------------------
Work Group: Core
Issue Number: 877
Title: 13.3.1.6 isn't about binding to a temporary
Section: 13.3 [over.match]
Status: editorial
Description:
13.3 para 2 says:
"Overload resolution selects the function to call in seven
distinct contexts within the language:
...
 invocation of a conversion function for initialization of a
temporary to which a reference (_dcl.init.ref_) will be
directly bound (_over.match.ref_)."

But 13.3.1.6 [over.match.ref] isn't about binding to a
temporary, it's about binding to an lvalue.

13.3.1.6 [over.match.ref] para 1 says:
"Under the conditions specified in _dcl.init.ref_, a reference
can be bound directly to an lvalue that is the result of
applying a conversion function to an initializer expression."

Resolution:
Requestor: Jason Merrill
Owner: Steve Adamczyk (Overload Resolution)
Emails:
Papers:

Work Group: Core
Issue Number: 733
Title: Implicit conversion sequences and scalar types
Section: 13.3.3.1 [over.best.ics]
Status: editorial
Description:
13.3.3.1 para 6:
"The implicit conversion sequence is the one required to convert
the argument expression to an rvalue of the type of the
parameter. ... When the parameter has a class type and the argument expression is an rvalue of the same type, the implicit conversion sequence is identity conversion. When a parameter has class type and the argument expression is an lvalue of the same type, the implicit conversion sequence is an lvalue-to-rvalue conversion."

Shouldn't the last two sentences also apply to non-class types?

Jason Merrill also notes in core-7309:

> In this test case, I assert that under the current overloading rules the second and third functions are equally good matches for the argument, even though the third is "obviously" the right choice. The ics for the third a reference binding to the lvalue, while the ics for the second is a reference binding to a temporary, but that also has identity rank because there are no lvalue-to-rvalue conversions for built-in types. Perhaps there should be?

```c
int f(char &);
int f(const char &);
int f(volatile char &);
int f(const volatile char &);

int main()
{
  volatile char c = 'a';
  f(c);
}
```

To which Stephen Adamczyk replies:

> I believe there are lvalue-to-rvalue conversions for built-in types.
> Perhaps you're interpreting 13.3.3.1 para 6 (over.best.ics) as saying there aren't, because it mentions them explicitly for class types but not for built-in types.
> But the class wording is needed because it is a special case.

For built-in types, the lvalue-to-rvalue conversion is a normal part of the implicit conversion sequence, and as 13.3.3.1.1 (over.ics.scs) says, that includes an lvalue-to-rvalue conversion when appropriate.

[Josee:] I think a note or footnote should be added to make this clear. I have seen many trip over this.

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

Work Group: Core
Issue Number: 779
Title: identity conversion is preferred over lvalue-to-rvalue
Subclause 13.3.2 paragraph 3, third sub-bullet has the following example:

```c
int g(const int&);
int g(int);
int i;
int k = g(i);      // ambiguous
```

The call to g is not ambiguous.
The match to g(const int&) is identity.
The match to g(int) requires an lvalue-to-rvalue conversion.

The first sub-bullet of paragraph 3 says that:
"the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence" because of this rule, g(const int &) is be preferred.

Issue 1:
Here's a program that was formerly valid, and now gets an ambiguity error:

```c
enum E {E1};

struct A {
    A();
    A(E);
    friend int operator==(A, E);
};

int main()
{
    E e;
    A a;
    e == E1; // Now ambiguous
}
```

The problem is that the 13.6 pseudo-prototypes for the "==" operator (and many others) do not explicitly deal with enums. As a consequence, any time an enum expression participates in an operation, it has to undergo at least a promotion to get to an arithmetic type. In the above example, that means the built-in operator "==" is worse than the friend function on the second operand. Since the built-in operator is better on the first operand, the case is ambiguous.

Issue 2:
This is a case that wasn't valid previously (because it declares an operator function with an enum parameter and no class parameter), but which gets a surprising answer:

```cpp
enum E {E1};

E operator+(E, int);

int main()
{
    E e;
    e + E1;  // Uses ::operator+
}
```

Case 2 seems less serious to me than case 1, partly because addition is not an operation on enums. I think adding two enums can reasonably be interpreted as going through the integral promotions before the addition is done.

Proposed Resolution:

The solution is probably to add more pseudo-prototypes in 13.6 to deal with the case where the operands of a builtin operation have the same enum type. This is particularly important for comparison operators, and for the "?" operator (but there is already an open core issue for that one).

Resolution:

Requestor: Steve Adamczyk
Owner: Steve Adamczyk (Overload Resolution)
Emails: 
Papers: 

---
Chapter 14 - Templates
---

Work Group: Core
Issue Number: 878
Title: Can a template declaration not followed by a definition specify export?
Section: 14 [temp]
Status: active
Description:

[John Spicer, core 7399:]
Can a template that is only declared (and not defined) in a translation unit be declared "export"?

The WP says:

"A non-inline template function or static data member template is called an exported template if its definition is preceded by the keyword export or if it has been previously declared using the keyword export in the same translation unit."

This does not make it clear whether an exported declaration is ill-formed or whether the "export" is simply ignored.

[Erwin Unruh, core-7407:]
We have five possible solutions:
1: Allow export only on definitions.
2: Allow export only on entities defined in that translation unit.
3: Allow export on declarations but without semantics if not followed by a definition.
4: Allow export on declarations with the semantic that this will
be an exported template.

5: Require export on all declarations of an exported template.

Requestor:      John Spicer
Owner:          Bill Gibbons (Templates)

In the following example:

```cpp
namespace A {
    struct B {};  
    template<class T> void f(T t);
}
void g(A::B b) {
    f<3>(b);
}
```

does type-dependent (Koenig) lookup apply to the lookup of "f"? Without the explicit "<>" template arguments, the answer is currently yes, because the lookup of "f" (other than to determine whether it is a type) can be deferred until after the arguments have been parsed. But with explicit template arguments, there is no way to parse the expression without knowing that "f" is a template.

Proposed Resolution:
We will propose that the "template" keyword be allowed in this context so that type-dependent lookup can be used even when there are explicit template arguments.

Resolution:      Mike Ball
Owner:          Bill Gibbons (Templates)

In the following example:

```cpp
template <int i> class S { }; S<3.3> s;
Can the template argument for the nontype template parameter i be a floating point constant?

14.3 para 3 says:
"A template-argument for a non-type non-reference template-parameter shall be an constant expression of integral type, ...
"

14.3 para 6 says:
"Standard conversions (4) are applied to an expression used as a template-argument for a non-type template-parameter to bring it to the type of its corresponding template parameter."
Proposed Resolution:

For parameters of integral or enumeration type, only the integral promotions and integral conversions are allowed (and not, for example, floating/integral conversions). For pointer and reference parameters, only derived-to-base conversions and conversion to "void*" are allowed. (If "void&" is added and conversion to "void&" is a standard conversion, then this would be allowed also.)

(Note that array-to-pointer and function-to-pointer conversions would always be done under the proposed resolution to 2.1; see 2.2 also.)

Resolution:
Requestor: Bill Gibbons (Templates)
Owner:          Bill Gibbons (Templates)
Emails:        . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Papers:        . .

Work Group:     Core
Issue Number:   880
Title:          When does a friend declaration refer to a global function or to a template instantiation?
Section:        14.5.3 [temp.friend]
Status:         editorial
Description:
Now that a global function can overload a template function, when does a friend declaration in a template class refers to the global function or when it refers to a template instantiation. For example:

```cpp
int foo(int);
template<class T> int foo(T);

template<class T> class C1 {
    friend int foo(int);
};
template<class T> class C2 {
    friend int foo(T);
};
template<class T> class C3 {
    friend int foo<int>(int);
};
```

[John Spicer's answer:]
> A friend declaration in which the declarator is not qualified, and that does not specify an explicit template argument list always declares a normal (i.e., nontemplate) function. So, C1 makes the previously declared foo(int) a friend. C2 does too, when T is int, otherwise it declares a new normal (nontemplate) function. C3 always refers to instances of the template foo. C3 does not make the global foo(int) a friend, it makes an instance of template foo a friend.

> One fuzzy area is what happens if you say

```cpp
template <class T> void f(T);
template <class T> struct A {
    friend void ::f(T);
};
```

> Does the global qualifier permit this to map onto the template? Without a WP change to permit this, my answer would be no.
How and when must the template specialization syntax be used with friend declarations?

```cpp
int foo(int);
template<class T> int foo(T);
template<> int foo<double>(double);

template<class T> class S2 {
    template<> friend int foo<T>(T);
};
```

[John Spicer's answer:]
> The "template <>" is not permitted in friend declarations.
> Core 3 did discuss and agree upon these issues (except that the issue I raised about the global qualifier was not discussed).

Proposed Resolution:
When explicit template arguments are provided, the friend declaration refers to the specialization. Other than that, if the enclosing scope has a function template and no non-template function, the friend declares a (hidden) new nontemplate function, exactly as if the function template declaration did not exist.

Resolution:
Requestor: Bill Gibbons (Templates)
Owner: Bill Gibbons (Templates)

For example, does the friend declaration in `A` make an instance of `N::f` a friend?

```cpp
namespace N {
    template <class T> void f(T);
}

template <class T> struct A {
    friend void N::f(T);
};
```

John Spicer's answer:
> It should be a valid means of making an instance of `N::f` a friend. Only unqualified friend declarations should be prohibited from referring to a previously declared template unless explicit template arguments are used. Our rationale for this is:
> 1. It is consistent with the way in which functions are called. An explicit template argument list is only needed in a call when the user wants to force the compiler to use
a template. In the absence of an explicit template argument list, overload resolution (for a call) or type matching (for the address of a function) is used to select the best match.

2. The real need is to guarantee that an unqualified declaration introduces a new function, and does not refer to the template. Permitting qualified references to previously declared templates in no way compromises this.

3. It eliminates a gratuitous incompatibility with existing code.

| Requestor:      | John Spicer          |
| Owner:          | Bill Gibbons (Templates) |
| Emails:        |                         |
| Papers:        |                         |

Work Group:     Core
Issue Number:   881
Title:          What class-key can be used in declarations of specializations and partial specializations?
Section:        14.5.4 [temp.class.spec] and 14.7.3 [temp.expl.spec]
Status:         active
Description:
Is it legal to have a specialization of a class template with a different class-key than that with which it was declared.
For example, the template is declared a class and the specialization is declared a union.

How about partial specializations?

I can't find any mention of template unions at all, but I presume that they are allowed since there is nothing disallowing them.

| Resolution:     |                         |
| Requestor:      | Mike Ball               |
| Owner:          | Bill Gibbons (Templates) |
| Emails:        |                         |
| Papers:        |                         |

Work Group:     Core
Issue Number:   882
Title:          typename is not permitted in functional cast notation
Section:        14.6 [temp.res]
Status:         editorial
Description:
The syntax does not permit "typename" to be used as part of a functional notation cast.

```cpp
template <class T> int f(T)
{
  return typename T::inner(); // typename not allowed here
}
```

If "typename" is not present, then T::inner is assumed to be a function name and the program is ill-formed if, during an instantiation, it turns out to be a type.

[Matt Austern:] There are a few places in the library description, such as in 20.4.4 (specialized algorithms) where it is assumed that this syntax is valid.
Resolution:  
Requestor:  John Spicer  
Owner:  Bill Gibbons (Templates)  
Emails:  
Papers:  

Work Group:  Core  
Issue Number:  884  
Title:  no diagnostics required for semantics errors in template definitions  
Section:  14.6.3 [temp.nondep]  
Status:  editorial  
Description:  
14.6.3 para 1 has the following example:  
    template<class T> class Z {  
public:  
    void f() {  
        h++;  // error cannot increment function  
    }  
};  

Maybe the comment should also indicate that an implementation doesn't have to diagnose this if the template is not instantiated.  

Something similar to the example in 14.6 paragraph 5 would be helpful:  
  // may be diagnosed even if ... is not instantiated.  

Proposed Resolution:  
The example in 14.6.3 should make it clear that although an implementation is allowed to diagnose this kind of error when processing the template definition, it is not required to diagnose such errors until the point of instantiation.  

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

Annex E - Universal-character-names  
------------------------------------  
Work Group:  Core  
Issue Number:  891  
Title:  The list of hexadecimal code for CJK Unified Ideographs seems incorrect  
Section:  Annex E[extendid]  
Status:  editorial  
Description:  
The list has the following:  
    fe74, 5e76-fefc, ...  
    5e76 should be replace by fe76.  
Resolution:  
Requestor:  
Owner:  Tom Plum (Annex E)  
Emails:  
Papers:  
