From: Randy Meyers
ZKO 2-3/N30
Digital Equipment Corp.
110 Spitbrook Road
Nashua, NH 03062
Phone: 603-881-2743
Email: rmeyers@decc.enet.dec.com

To: X3 Secretariat
Attn: Deborah J. Donovan
1250 EyeStreet, NW
Suite 200
Washington, DC 20005

CC: ANSI
Attn: BSR Center
11 West 42nd Street
New York, NY 10036

Subj: Public Review Comment on ISO/IEC CD 14882

Enclosed are my comments on ISO/IEC CD 14882. Please forward them to X3J16.

Thank You.

1 GRAMMAR ISSUE

1.1 Default Arguments In Template Definitions

Subclause 14.2, paragraph 3, contains the following rule to resolve parsing problems in the names of template specializations:

> When parsing a template-id, the first non-nested > is taken as the end of the template-argument-list rather than a greater-than operator.

No similar rule exists for declarations, yet similar parsing problems exist since non-type template parameters may have default arguments:

```
const int x = 6;
const int y = 5;
template<int i = x > y> class A {float a[i];};
```

The parsing is particularly challenging for compilers who wish to give intelligent error messages for (or support in a compatibility mode) the obsolete practice of "default int":

```
class C {
  template<int i = x > y() {return i;}  //C::y(), not ::y
}
```


I tested 5 compilers: four of them assumed the first non-nested > ended the template parameters. The fifth compiler did not support any form of default arguments.

Recommendation: Add the following at the end of paragraph 3 of Clause 14:

> When parsing a template-declaration, the first non-nested > is taken as the end of the template-parameter-list rather than a greater-than operator.

The following examples could also be added:

```
const int x = 6;
const int y = 5;
template<int i = x > y> class A {float a[i];};   // syntax
                                                 //   error
template<int i = (x > y)> class B {float b[i];}; // ok
```

2 CORE LANGUAGE ISSUE

2.1 Member Access Control

Clause 11, paragraph 1 says:

> A member of a class can be
>
> --private; that is, its name can be used only by member functions, static data members, and friends of the class in which it is declared.
>
> --protected; that is, its name can be used only by member functions, static data members, and friends of the class in which it is declared and by member functions, static data members, and friends of classes derived from this class (see _class.protected_).

This seems overly restrictive in contrast to saying:

> A member of a class can be
>
> --private; that is, its name can be used only by members and friends of the class in which it is declared.
>
> --protected; that is, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class (see _class.protected_).

For example, the current wording prevents reasonable uses such as:


```
class C {
    class INNER { ... };
    INNER private_data;         // Bad use of INNER?
    class IN2 : INNER { ... };  // Bad use of INNER?
};
```

Recommendation: Use the alternative wording above.

3 C COMPATIBILITY ISSUES

3.1 Universal Character Names

WG14/X3J11 has voted to adopt the universal character set name proposal from the C++ Working Paper. However, WG14/X3J11 discovered that a piece of the original proposal was accidentally dropped from the proposal voted into C++. During original discussions of the UCN proposal, the intent was that a UCN could not be used to write a character from the Basic Source Character Set.

For example, you could not end a quoted string by specifying the UCN for quote.

WG14/X3J11 directed the editorial board drafting the final words for the C9x Standard to add such a restriction. C++ should also add this restriction.

In a discussion with Tom Plum, we came to the conclusion that writing a UCN for a character in the Basic Source Character Set should be a constraint violation for C, and should make the program ill-formed for C++.

Ill-formed was chosen over undefined behavior since it is undesirable to permit extensions:

1. Some implementations might treat identifiers spelled using UCNs as distinct from identifiers spelled directly.

2. Some implementations might treat the UCN for a quote as ending a string; some might consider the UCN for a quote as a quote character in the string's value.

3. Allowing UCNs to represent characters in the Basic Source Character Set is likely to slow down the lexical processing of C++ source.

However, that brings up another issue: Although the "printing" characters have an unambiguous UCN, the UCN for some of the non-printing characters from the Basic Source Character Set have an implementation defined UCN. For example, some implementations use linefeed as the newline source character; other implementations use return as the newline source character. In my discussion with Plum, we decided the best way to handle this is to

prohibit UCNs with hex codes less than 0x20, which disallows all of the traditional ASCII control characters.

Recommendation: Add the following words at the end of paragraph 2 of Subclause 2.2:

> If the hexadecimal value for a universal character name is less than 0x20 or if the universal character name designates a character in the basic source character set, then the program is ill-formed.

3.2 Type Rules For Integer Constants

WG14/X3J11 has adopted a change to the rules for determining the type of an integer literal: If an integer literal is of the form

that normally is always signed (for example, unsuffixed decimal constants) then do not permit it to have unsigned type if it can not be represented as long (currently, such a constant may have unsigned long type).

The reason for the change is that C9x has adopted "long long", and some C compilers are contemplating integer types even longer than long long. The current rules introduce an anomaly every time a new larger integer type is added: normally signed constants pass through ranges where they are unsigned. See Footnote 21 of Subclause 2.13.2 for problems that this can cause.

Programs that depend upon unsuffixed decimal literals being unsigned (as opposed to signed) are not portable anyway: implementations with a larger word size might be able to represent the constant as a signed type.

Implementations are free to continue to use an unsigned type as a last resort to represent a large decimal constant: After issuing a diagnostic, the implementation could then use unsigned long at the constant's type.

Recommendation: Change paragraph 2 of Subclause 2.13.1 to read:

> The type of an integer literal depends on its form, value, and suffix. If it is decimal and has no suffix, it has the first of these types in which its value can be represented: int, long int. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: int, unsigned int, long int, unsigned long int. If it is suffixed by u or U, its type is the first of these types in which its value can be represented: unsigned int, unsigned long int. If it is suffixed by l or L, its type is long int. If it is suffixed by ul, lu, uL, Lu, Ul, lU, UL, or LU, its type is unsigned long int.

Delete footnote 21 formerly referenced by this paragraph, or rewrite the footnote to discuss why normally decimal constants

without a U suffix are never unsigned.