X3J16/97-0101
WG21/N1139
November 12th, 1997
John H. Spicer, Edison Design Group
jhs@edg.com

Working Paper Changes for Template Issues 97-0087/N1125

1. Changes for issue 6.60

Replace 14.5.3 [temp.friend] paragraph 1 with (but retain the example at the end of paragraph 1):

> A friend of a class or class template can be a function template or class template, a specialization of a function template or class template, or an ordinary (nontemplate) function or class. For a friend function declaration that is not a template declaration:
>
> - if the name of the friend is a qualified or unqualified *template-id*, the friend declaration refers to a specialization of a function template, otherwise
> - if the name of the friend is a *qualified-id* and a matching nontemplate function is found in the specified class or namespace, the friend declaration refers to that function, otherwise,
> - if the name of the friend is a *qualified-id* and a matching specialization of a template function is found in the specified class or namespace, the friend declaration refers to that function specialization, otherwise,
> - the name shall be an *unqualified-id* that declares (or redeclares) an ordinary (nontemplate) function.

2. Changes for issue 6.62

Add to 14.5.3 [temp.friend] following paragraph 1:

A friend function declaration that is not a template declaration and in which the name of the friend is an unqualified template-id shall refer to a specialization of a function template declared in the nearest enclosing namespace scope.

```
namespace N {
      template <class T> void f(T);
      void g(int);
      namespace M {
            template <class T> void h(T);
            template <class T> void i(T);
            struct A {
                  friend void f<>(int);  // ill-formed - N::f
                  friend void h<>(int);  // okay - M::h
                  friend void g(int); // okay - new decl of M::g
                  template <class T> void i(T);
                  friend void i<>(int); // ill-formed - A::i
            };
      }
}
```

3. Changes for issue 6.63

Add to the beginning of section 3.4.5 [basic.lookup.classref]:

If the sequence of tokens following the `.` or `->` is of the form `identifier<` the identifier must be looked up to determine whether the `<` is the beginning of a template argument list (14.2 [temp.names]) or a less-than operator. The identifier is first looked up in the class of the object expression. If the identifier is not found, it is then looked up in the context of the entire *postfix-expression* and shall name a class or function template. If the lookup in the class of the object expression finds a template, the name is also looked up in the context of the entire *postfix-expression* and

- if the name is not found, the name found in the class of the object expression is used, otherwise
- if the name is found in the context of the entire *postfix-expression* and does not name a class template, the name found in the class of the object expression is used, otherwise
- if the name found is a class template, it must refer to the same entity as the one found in the class of the object expression, otherwise the program is ill-formed.

4. Changes for issue 6.64

Add to the end of 14.8.2 [temp.deduct] (before 14.8.2.1):

When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

1. The specified template arguments must match the template parameters in kind (i.e., type, nontype, template), and there must not be more arguments than there are parameters; otherwise type deduction fails.
2. Nontype arguments must match the types of the corresponding nontype template parameters, or must be convertible to the types of the corresponding nontype parameters as specified in 14.3.2 [temp.arg.nontype], otherwise type deduction fails.
3. All references in the function type of the function template to the corresponding template parameters are replaced by the specified template argument values. If a substitution in a template parameter or in the function type of the function template results in an invalid type, type deduction fails. [Note: The equivalent substitution in exception specifications is done only when the function is instantiated, at which point a program is ill-formed if the substitution results in an invalid type.] Type deduction may fail for the following reasons:

i) Attempting to create an array with an element type that is void, a function type, or a reference type, or attempting to create an array with a size that is zero or negative.

```
template <class T> int f(T[5]);
int I = f<int>(0);
int j = f<void>(0);  // invalid array
```

ii) Attempting to use a type that is not a class type in a qualified name.

```
template <class T> int f(typename T::B*);
int i = f<int>(0);
```

iii) Attempting to use a type in the qualifier portion of a qualified name that names a type when that type does not contain the specified member, or if the specified member is not a type where a type is required.

```
template <class T> int f(typename T::B*);
struct A {};
struct C { int B; };
int i = f<A>(0);
int j = f<C>(0);
```

iv) Attempting to create a pointer to reference type.

v) Attempting to create a reference to a reference type or a reference to `void`.

vi) Attempting to create "pointer to member of T" when T is not a class type.

```
template <class T> int f(int T::*);
int i = f<int>(0);
```

vii) Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration.

```
template <class T, T*> int f(int);
int i2 = f<int,1>(0); // can't conv 1 to int*
```

viii) Attempting to create a function type in which a parameter has a type of `void`.

ix) Attempting to create a *cv-qualified* function type.

After this substitution is performed, the function parameter type adjustments described in 8.3.5 [dcl.fct] are performed. [Example: a parameter type of `void ()(const int, int[5])` becomes `void(*)(int,int*)`]. [Note: A top-level qualifier in a function parameter declaration does not affect the function type but still affects the type of the function parameter variable within the function.

```
template <class T> void f(T t);
template <class X> void g(const X x);
template <class Z> void h(Z, Z*);
int main()
{
      // #1: function type is f(int), t is nonconst
      f<int>(1);
      // #2: function type is f(int), t is const
      f<const int>(1);
      // #3: function type is g(int), x is const
      g<int>(1);
      // #4: function type is g(int), x is const
      g<const int>(1);
      // #5: function type is h(int, const int*)
      h<const int>(1,0);
}
```

Note that `f<int>(1)` and `f<const int>(1)`call distinct functions even though both of the functions called have the same function type. - end note]

The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. When all template arguments have been deduced, all uses of template parameters in nondeduced contexts are replaced with the corresponding deduced argument values. If the substitution results in an invalid type, as described above, type deduction fails.

Except as described above, the use of an invalid value shall not cause type deduction to fail. [Example: In the following example 1000 is converted to char and results in an implementation-defined value as specified in 4.7. In other words, both templates are considered even though 1000, when converted to signed char, results in an implementation-defined value.

```
template <int> int f(int);
template <signed char> int f(int);
int i1 = f<1>(0); // ambiguous
int i2 = f<1000>(0);  // ambiguous
```

■ end example]