X3J16/97-0054R1
W21/N1092
July 17th 1997
John H. Spicer, Edison Design Group

Core-3 Working Paper Changes - Part 1

1. L6915 - Partial ordering rules for conversion functions

In 14.5.5.2 [temp.func.order] p3, change each of the bullets from “each occurrence of that parameter in the function parameter list” with “each occurrence of that parameter in the function parameter list, or for a template conversion operator, in the return type”.

2. L7168 & L3401 - Explicit instantiation is point of instantiation

Add to 14.6.4.1 [temp.point] after paragraph 3: “An explicit instantiation directive is an instantiation point for the specialization or specializations specified by the explicit instantiation directive.”

3. L7116 - Equivalence of nontype template arguments in function template declarations

Add the following to the end of 14.5.5.1 [temp.over.link]:

When an expression that references template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the nontype template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. For example:

```
template <int I, int J> A<I+J> f(A<I>, A<J>); // #1
template <int K, int L> A<K+L> f(A<K>, A<L>); // same as #1
template <int I, int J> a<I-J> f(A<I>, A<J>); // different from #1
```

[Note: Most expressions that use template parameters use nontype template parameters, but it is possible for an expression to reference a type parameter. For example, a template type parameter can be used in the sizeof operator.]

For two expressions involving template parameters to be considered equivalent the expressions shall satisfy the conditions that must be met for two definitions to be equivalent as specified in the “One Definition Rule” (3.3), except that tokens used to name a template parameter in one expression shall name the corresponding template parameter in the other expression. For example:

```
template <int I, int J> void f(A<I+J>); // #1
template <int K, int L> void f(A<K+L>); // same as #1
```

Expressions that do not satisfy this requirement are not equivalent but may be “functionally equivalent”. Two expressions are functionally equivalent if, for any given set of template arguments, the evaluation of the expression results in the same value. Two function templates are considered equivalent if they are declared in the same scope, have the same name, have identical template parameter lists, and have return types and function parameter lists that are identical using the equivalence rules described above to compare expressions involving nontype template parameters. Two function templates are considered functionally equivalent if they are equivalent except that one or more of the nontype expressions that involve template parameters is functionally equivalent but not equivalent. If a program contains declarations of function templates that are functionally equivalent, the program is ill-formed. No diagnostic is required.

[Note: This rule guarantees that equivalent declarations will be linked with one another, while not requiring implementations to use heroic efforts to guarantee that functionally equivalent declarations will be treated as distinct. For example, the following declarations are functionally equivalent and would cause a program to be ill-formed:]

```
// Guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+10>);

// Guaranteed to be different
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+11>);

// Ill-formed, no diagnostic required
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
```

4. L7069 - Use of explicit function template argument lists

4.1 Revisions to 14.8.1 [temp.arg.explicit]

The following should replace 14.8.1, with the examples from 14.8.1 merged in.

Template arguments can be specified when referring to a function template specialization by qualifying the function template specialization name with the list of template-arguments in the same way as template-arguments are specified in uses of a class template specialization.

A template argument list may be specified when referring to a specialization of a function template
- when a function is called,
- when the address of a function is taken, or when a pointer to member address of a function is taken,
- in an explicit specialization,
- in an explicit instantiation, or
- in a friend declaration.

Trailing template arguments that can be deduced (14.8.2) may be omitted from the list of explicit template arguments, including the omission of all of the arguments. [Note: An empty template-argument list can be used to indicate that a given reference refers to a specialization of a function template even when a normal (i.e., nontemplate) function is visible that would otherwise be used. For example:

```
template <class T> int f(T);  // #1
int f(int);  // #2
int k = f(1); // uses #2
int l = f<>(1);  // uses #1
```

Template arguments that are present are specified in the declaration order of their corresponding template-parameters. The template-argument list shall not specify more template arguments than there are corresponding template-parameters.

Implicit conversions (_conv_) will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter type contains no template parameters that participate in template argument deduction. [Note: Template parameters do not participate in template argument deduction if they are either explicitly specified, or are used only in nondeduced contexts (see 14.8.2 [temp.deduct]).]

When a friend declaration refers to a specialization of a function template, the function parameter declarations shall not include default arguments, nor shall the inline specifier be used in such a declaration.

4.2 Other Changes

In 14.8.2 [temp.deduct] remove paragraph 17 and the normative portion of paragraph 20.