Working Paper Changes for the Template Compilation Model

John Wilkinson, Silicon Graphics
Bjarne Stroustrup, AT&T

Note: N0973 (no revision) was an interim copy distributed only at the Stockholm meeting and not included in any mailing.

The following Working Paper changes are required for the new template compilation model proposal.

Part 1: The following changes are required only to support separation.

2.11: Add keyword "export" to Table 3.

14 paragraph 3: Replace the grammar for template-declaration by:

template-declaration:
export (opt) template<template-parameter-list> declaration

14 paragraph 6: Delete

to have external linkage

14: Add paragraph 7:

A non-inline template function or a 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. Declaring a class template exported is equivalent to declaring all of its function members, static data members, and member templates which are defined in that translation unit exported.

Templates defined in an unnamed namespace shall not be exported. A template shall be exported only once in a program. An implementation is not required to diagnose a violation of this rule. A non-exported template that is neither explicitly specialized nor explicitly instantiated must be defined in every translation unit in which it is implicitly instantiated (14.7.1) or explicitly instantiated (14.7.2); an exported template need only be declared (and not necessarily defined) in a translation unit in which it is instantiated. A template function declared both exported and inline is just inline and not exported.

14: Add paragraph 8: (Note that this probably belongs with phases of translation (2.1 paragraph) instead of here.)

An implementation may require that a translation unit containing the definition of an exported template be compiled before any other translation unit instantiating that template.

14.7.2 paragraph 3: Replace

A definition of the static data member template

by

A declaration of the static data member template

Replace box 29 by

The definition of a non-exported function template or non-exported data member template shall be present in every translation unit in which it is explicitly instantiated.

Part 2: The following changes are independent of whether separation is permitted or not:

Remove boxes 16 and 27.

14.6 paragraph 6: replace by

Three kinds of names can be used within a template definition:

-- The name of the template itself, the names of the template parameters (14.1), and names declared within the template itself.

-- Names dependent on a template parameter (14.6.2).

-- Names from scopes with are visible within the template definition.

14.6.4: replace entire section by the following:

14.6.4 Dependent Name Resolution

In resolving dependent names, we consider names from the following sources:

-- Declarations that are visible at the point of definition of the template.

-- Declarations from namespaces associated with the types of the function arguments, both from the instantiation context (14.6.4.1) and from the definition context.

14.6.4.1 Point of Instantiation

If the instantiating reference of an implicit template function specialization is a dependent function call, then the point of instantiation of the specialization is the point of instantiation of the template function specialization containing the instantiating reference.

Otherwise, the point of instantiation of the specialization is the point immediately preceding the definition containing the instantiating reference. (For the purposes of this definition we consider the definition of a member function defined within its class to follow the outermost class definition containing the member function definition.)

By the instantiating context of a dependent call, we mean the set of declarations with external linkage visible at the point of instantiation of the template function specialization containing the call.

14.6.4.2 Associated Namespaces

With each type T we associate a set of namespaces.

If T is a fundamental type, its associated set of namespaces is empty.

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.

If T is a union or enumeration type, its associated namespace is the namespace in which it is defined.

If T is a pointer to U, a reference to U, or an array of U, its associated namespaces are the associated namespaces of U.

If T is a pointer to function type, its associated namespaces are the namespaces of the function parameter types and of the return type.

If T is a pointer to member function of a class X, its associated namespaces are the namespaces of the function parameter and return types, together with the namespaces associated with X.

If T is a pointer to a data member of a class X, its associated namespaces are the namespaces associated with X and the namespaces associated with the member type.

If T is a template-id, its associated namespaces are the namespace of the template and the namespaces of the type template arguments.

14.6.4.3 Candidate Functions

One set of candidate functions comes from the definition context.

The others come from the associated namespaces of the types of the arguments of the function call or of the operands of the operator (14.6.4.2). Only names with external linkage are considered.

Only functions actually declared in a given namespace are considered, not functions imported into the namespace by using directives, or functions declared in enclosing namespaces.

From these namespaces only declarations from the definition context or the instantiation context are considered. If a function with external linkage declared in one of these namespaces is a better match for a given dependent call than any of the functions declared in that namespace in either the definition or the instantiation context, then the program has undefined behavior.

14.6.4.4 Conversions

All standard conversions are permitted in matching candidate functions. A user-defined conversion must be either a member conversion from its argument class, or a member constructor from its result class. It must come from either the definition context or the instantiation context. (Note that the set of candidate functions is formed first, before conversions are considered, so the possible conversions do not affect the set of candidate functions.)