X3J16/97-0052
WG21/N1090
July 16th, 1997
John H. Spicer, Edison Design Group

Revised Template Argument Deduction Section (14.8.2 [temp.deduct])

1. Introduction

The purpose of this document is to provide revised wording for section 14.8.2 [temp.deduct] to address the following problems that have been identified:

- It does not describe the handling of nondeduced contexts, such as T in A<T>::B.A
- It does not fully describe deduction when taking the address of an overloaded function
- It does not address deduction of template conversion operators.

The sections below are intended to replace 14.8.2 paragraphs 1 through 4, although the examples from that section must be merged with the text below.

The existing section describes argument deduction for function calls only. The revised section begins with a general description of the template argument deduction process and then describes how deduction is applied in the various contexts in which it can be used.

2. Template Argument Deduction

When a template function specialization is referenced, all of the template arguments must have values. The values can be either explicitly specified or, in some cases, deduced from the reference.

2.1 Deducing Template Arguments from a Type

Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it P) is compared with an actual type (call it A), and an attempt is made to find template argument values (a type for a type parameter, a value for a nontype parameter, or a template for a template parameter) that will make P, after substitution of the deduced values (call it the deduced A), compatible with A.

In some cases, the deduction is done using a single set of types P and A, in other cases, there will be a set of corresponding types P and A. Type deduction is done independently for each P/A pair, and the deduced template argument values are then combined. If type deduction cannot be done for any P/A pair, or if for any pair the deduction leads to more then one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails.

A given type P can be composed of a number of references to other types, templates, and nontype values:

- A function type includes the types of each of the function parameters and the return type.
- A pointer to member type includes the type of the class object pointed to, and the type of the data member pointed to (for a data member) or the function type pointed to (for a member function).
- A type that is a specialization of a class template (e.g., A<int>) includes the types, templates, and nontype values referenced by the template argument list of the specialization.
- An array type includes the array element type and the value of the array bound.

In most cases, the types, templates, and nontype values that are used to compose P participate in template argument deduction. That is, they may be used to determine the value of a template argument, and the value so determined must be consistent with the values determined elsewhere. In certain contexts, however, the value does not participate in type deduction, but instead uses the values of template arguments that were either deduced elsewhere or explicitly specified. If a template parameter is used only in nondeduced contexts and is not explicitly specified, template argument deduction fails.

The nondeduced contexts are:

- The nested-name-specifier of a type that was specified using a qualified-name.
- A type that is a template-id in which one or more of the template-arguments is an expression that references a nontype template parameter.

When a type name is specified in a way that includes a nondeduced context, all of the types that comprise that type name are also nondeduced. For example, if a type is specified as A<T>::B<T2>, both T and T2 are nondeduced. Likewise, if a type is specified as A<I+J>::X<T>, I, J, and T are nondeduced. However, a compound type can include both deduced and nondeduced types. For example, if a type is specified as void f(A<T>::B, A<T>), the T in A<T>::B is nondeduced but the T in A<T> is deduced.

2.2 Deducing template arguments from a function call

Template argument deduction is done by comparing each function template parameter type (call it P) with the type of the corresponding argument of the call (call it A) as described above.

If P is not a reference type:

- If A is an array type, the pointer type produced by the array-to-pointer standard conversion is used in place of A for type deduction; otherwise,
- If A is a function type, the pointer type produced by the function-to-pointer standard conversion is used in place of A for type deduction; otherwise,
- If A is a cv-qualified type, the top level cv-qualifiers of A's type are ignored for type deduction.

If P is a cv-qualified type, the top level cv-qualifiers of P's type are ignored for type deduction. If P is a reference type, the type referred to by P is used for type deduction.

In general, the deduction process attempts to find template argument values that will make the deduced A identical to A (after the type A is transformed as described above). However, there are three cases that allow a difference:

- If the original P is a reference type, the deduced A (i.e., the type referred to by the reference) can be more cv-qualified than A.
- A can be another pointer or pointer to member type that can be converted to the deduced A via a qualification conversion.
- If P is a class, and P has the form class-template-name<arguments>, A can be a derived class of the deduced A. Likewise, if P is a pointer to a class of the form class-template-name<arguments>, A can be a pointer to a derived class pointed to by the deduced A.

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails.

2.3 Deducing template arguments when taking the address of a function template

For purposes of taking the address of a function, a function template is considered an overloaded function name. When taking the address of an overloaded function in which the overload set contains one or more function templates, a matching function or specialization is selected as follows:

- Any nontemplate functions are considered first. If a single function that matches the required type is found, that function is used. If more than one function matches the type, the program is ill-formed.
- If no matches are found among the nontemplate functions in the overload set, any function templates in the set are examined to determine if a specialization of the template matches the specified type. The function template's function type and the specified type are used as the types of P and A, and the deduction is done as described in section 2.1. If more than one function template can generate the required specialization, the partial ordering rules are used to choose among them. If the partial ordering rules do not result in a single more specialized function, type deduction fails (or should it be ill-formed?)

If the specified function type includes deducible template parameters, only nontemplate functions will be considered when looking for a matching function.

2.4 Deducing template arguments when calling a template conversion operator

Template argument deduction is done by comparing the return type of the template conversion operator (call it P) with the type that is required as the result of the conversion (call it A) as described in section 2.1 above.
If A is not a reference type:

- If P is an array type, the pointer type produced by the array-to-pointer standard conversion is used in place of P for type deduction; otherwise,
- If P is a function type, the pointer type produced by the function-to-pointer standard conversion is used in place of P for type deduction; otherwise,
- If P is a cv-qualified type, the top level cv-qualifiers of P's type are ignored for type deduction.

If A is a cv-qualified type, the top level cv-qualifiers of A's type are ignored for type deduction. If A is a reference type, the type referred to by A is used for type deduction.

In general, the deduction process attempts to find template argument values that will make the deduced A identical to A. However, there are two cases that allow a difference:

- If the original A is a reference type, A can be more cv-qualified than the deduced A (i.e., the type referred to by the reference)
- The deduced A can be another pointer or pointer to member type that can be converted to A via a qualification conversion.

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails.

3. Other WP changes

- Remove 14.8.2p14.
- In 13.4, add to paragraph 1: Indicate that the name of a function template is treated as an overloaded function name, including a function template name followed by an explicit template argument list.