Class Template Argument Deduction
Assorted NB resolution and issues

Document #:	P0512R0
Date :	2016-11-10
Revises :	None
Working Group:	Core
Reply to:	Faisal Vali (faisalv@yahoo.com)

Mike Spertus Richard Smith Faisal Vali

Abstract

This paper provides wording for resolutions to NB comments US 19 and US 20 and also clarifies how overload resolution works with the initializer (especially when it is a braced-init-list) and how the explicit specifier is handled during class template argument deduction.

1 Background

Refer to the latest revision of P0091for background, motivation and rationale regarding the **class template argument deduction** feature and P00433 for background on using it within the standard library.

This paper contains core language wording for national body comments and additional fixes related to class template argument deduction:

1. US19 – give deduction guides preference over constructors as a tie breaker during overload resolution
2. US20 – do not allow accidental forwarding references during class template argument deduction
3. Allow the explicit decl-specifier on deduction guides, and only consider functions generated from deduction-guides or constructors specified as explicit, during direct-initialization.
4. Fix the wording so that the initialization expression can be used as arguments to the generated functions and function templates, and for overload resolution to do the right thing when the initializer is a braced-init-list (which only works for constructor calls but not function calls i.e. A{1} can only be valid if 'A' is a class but not if 'A' is a function) we need to say something more to help connect the dots during overload resolution.

2 Core Wording

[Drafting note: The following handles US19 - giving deductions guides precedence over constructors for class template argument deduction if the conversions are as good, and even if the function template generated from the constructor is more specialized than the one generated from the deduction guide - this was discussed in EWG - but it is not clear to the author (who was in the room at the time of the vote) if EWG has had a chance to weigh in on the details of when a deduction guide would override a constructor generated function template (i.e before partial ordering or after)]

Add bullet between 13.3.3/1.5 & 13.3.3/1.6

…

– F1 is generated from a deduction-guide (13.3.1.8) and F2 is not [*Example:*

```
template<class T> struct A {
  A(T, int*);                  // #1
```

```
   A(A<T>&, int*);               // #2
   enum { value };
};


template<class T, int N = T::value> A(T&&, int*) -> A<T>;  //#3

A a{1,0};  // uses #1 to deduce A<int> and initializes with #1
A b{a,0};  // uses #3 (not #2) to deduce A<A<int>&> and initializes
           // with #1
```

– *end example*] or, if not that,

— F1 is not a function template specialization and F2 is a function template specialization, or, if not that,

---

[Drafting note: The following wording handles US20 which prevents accidental forwarding references during class template argument deduction]

14.8.2.1/3

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. A forwarding reference is an rvalue reference to a cv-unqualified template parameter that does not represent a template parameter of a class template (during class template argument deduction (13.3.1.8)). If P is a forwarding reference and the argument is an lvalue, the type "lvalue reference to A" is used in place of A for type deduction.

[Example:

```
template <class T> int f(T&& heisenreference);
template <class T> int g(const T&&);
int i;
int n1 = f(i);   // calls f<int&>(int&)
int n2 = f(0);    // calls f<int>(int&&)
int n3 = g(i);    // error: would call g<int>(const int&&), which
                  // would bind an rvalue reference to an lvalue

template<class T> struct A {
   template<class U>
   A(T&&, U&&, int*);  // #1: T&& is not a forwarding reference
                       //     U&& is a forwarding reference
   A(T&&, int*);       // #2
};

  template<class T>
  A(T&&, int*) -> A<T>;  // #3: T&& is a forwarding reference
```

```
int *ip;
A a{i, 0, ip};  // error, cannot deduce from #1
A a0{0, 0, ip}; // uses #1 to deduce A<int> and #1 to initialize
A a2{i, ip};    // uses #3 to deduce A<int&> and #2 to initialize
```

—end example]

[Drafting Note: To get **class template argument deduction** to work so that the initialization expression can be used as arguments to the generated functions and function templates, and for overload resolution to do the right thing when the initializer is a braced-init-list (which only works for constructor calls but not function calls i.e. A{1} can only be valid if 'A' is a class but not if 'A' is a function) we need to say something more to help connect the dots during overload resolution.
Additionally do not use deduction guides or constructors specified as explicit during deduction when the initializer represents copy-initialization]

Modify the end of §14.9p1 [temp.deduct.guide] as follows:
deduction-guide:
explicit$_{opt}$ *template-name* (*parameter-declaration-clause*) ->
simple-template-id ;

Modify 13.3.1.8/1 as follows:
13.3.1.8 Class template argument deduction [over.match.class.deduct]

1 The overload set consists of:
A set of functions and function templates is formed comprising:
(1.1) — For each constructor of the class template designated by the template-name, a function template with the following properties is a candidate:
(1.1.1) — The template parameters are the template parameters of the class template followed by the template parameters (including default template arguments) of the constructor, if any.
(1.1.2) — The types of the function parameters are those of the constructor.
(1.1.3) — The return type is the class template specialization designated by the template-name and template arguments corresponding to the template parameters obtained from the class template.

(1.2) — For each deduction-guide, a function or function template with the following properties is a candidate:
(1.2.1) — The template parameters, if any, and function parameters are those of the deduction-guide.

(1.2.2) — The return type is the simple-template-id of the deduction-guide.

Initialization and overload resolution are performed as described in 8.6 and 13.3.1.3, 13.3.1.4, or 13.3.1.7 (as appropriate for the type of initialization performed) for an object of a hypothetical class type, where the selected functions and function templates are considered to be the constructors of that class type for the purpose of forming an overload set, and the initializer is provided by the context in which class template argument deduction was performed. Each such notional constructor is considered to be explicit if the function or function template was generated from a constructor or deduction-guide that was declared explicit.
[*Example:*

```
template<class T> struct A {
  explicit A(const T&, ...) noexcept;  // #1
  A(T&&, ...);            // #2
};

int i;
A a1 = { i, i }; // error: cannot deduce from rvalue reference in #2,
                 // and #1 is not a candidate during  deduction
                 // from copy-initialization.
A a2{i, i};    // OK, #1 deduces to A<int> and also initializes
A a3{0, i};    // OK, #2 deduces to A<int> and also initializes
A a4 = {0, i}; // OK, #2 deduces to A<int> and also initializes


template<class T> A(const T&, const T&) -> A<T&>; // #3
template<class T> explicit A(T&&, T&&)  -> A<T>;   // #4

A a5 = {0, 1}; // error: #3 deduces to A<int&> and #1 & #2 result
               // in same parameter constructors.
A a6{0,1};     // OK, #4 deduces to A<int> and #2 initializes
A a7 = {0, i}; // error: #3 deduces to A<int&>
A a8{0,i};     // error: #3 deduces to A<int&>

template<class T> struct B {
  template<class U> using TA = T;
  template<class U> B(U, TA<U>);
};
B b{(int*)0, (char*)0}; // OK, deduces B<char*>
```

--end example]

3 References/Acknowledgment

Jason Merrill for helping with the wording having to do with forwarding references.

John Spicer for helping me better understand certain aspects of template redeclaration

collisions vs those similar collisions being avoided during overload resolution.