Template argument deduction

In order to instantiate a function template, every template argument must be known, but not every template argument has to be specified. When possible, the compiler will deduce the missing template arguments from the function arguments. This occurs when a function call is attempted, when an address of a function template is taken, and in some other contexts:

```
template<typename To, typename From> To convert(From f);

void g(double d)
{
    int i = convert<int>(d);    // calls convert<int, double>(double)
    char c = convert<char>(d);  // calls convert<char, double>(double)
    int(*ptr)(float) = convert; // instantiates convert<int, float>(float)
}
```

This mechanism makes it possible to use template operators, since there is no syntax to specify template arguments for an operator other than by re-writing it as a function call expression:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world" << std::endl;
    // operator<< is looked up via ADL as std::operator<<,
    // then deduced to operator<<<char, std::char_traits<char>> both times
    // std::endl is deduced to &std::endl<char, std::char_traits<char>>
}
```

Template argument deduction takes place after the function template name lookup (which may involve argument-dependent lookup) and before template argument substitution (which may involve SFINAE) and overload resolution.

Deduction from a function call

Template argument deduction attempts to determine template arguments (types for type template parameters **T**i, templates for template template parameters **TT**i, and values for non-type template parameters **I**i), which can be substituted into each parameter **P** to produce the type *deduced* **A**, which is the same as the type of the argument **A**, after adjustments listed below.

If there are multiple parameters, each **P**/**A** pair is deduced separately and the deduced template arguments are then combined. If deduction fails or is ambiguous for any **P**/**A** pair or if different pairs yield different deduced template arguments, or if any template argument remains neither deduced nor explicitly specified, compilation fails.

If removing references and cv-qualifiers from **P** gives `std::initializer_list<P'>` and **A** is a braced-init-list, then deduction is performed for every element of the initializer list, taking **P**' as the parameter and the list element **A**' as the argument:

```
template<class T> void f(std::initializer_list<T>);
f({1, 2, 3});  // P = std::initializer_list<T>, A = {1, 2, 3}
               // P'1 = T, A'1 = 1: deduced T = int
               // P'2 = T, A'2 = 2: deduced T = int
               // P'3 = T, A'3 = 3: deduced T = int
               // OK: deduced T = int
f({1, "abc"}); // P = std::initializer_list<T>, A = {1, "abc"}
               // P'1 = T, A'1 = 1: deduced T = int
               // P'2 = T, A'2 = "abc": deduced T = const char*
               // error: deduction fails, T is ambiguous
```

If removing references and cv-qualifiers from **P** gives **P**'[**N**], and **A** is a non-empty braced-init-list, then deduction is performed as above, except if **N** is a non-type template parameter, it is deduced from the length of the initializer list:

```
template<class T, int N> void h(T const(&)[N]);
h({1, 2, 3}); // deduced T = int, deduced N = 3

template<class T> void j(T const(&)[3]);
j({42}); // deduced T = int, array bound is not a parameter, not considered

struct Aggr { int i; int j; };
template<int N> void k(Aggr const(&)[N]);
k({1, 2, 3});       // error: deduction fails, no conversion from int to Aggr
k({{1}, {2}, {3}}); // OK: deduced N = 3

template<int M, int N> void m(int const(&)[M][N]);
m({{1, 2}, {3, 4}}); // deduced M = 2, deduced N = 2

template<class T, int N> void n(T const(&)[N], T);
n({{1}, {2}, {3}}, Aggr()); // deduced T = Aggr, deduced N = 3
```

(since C++17)

If a parameter pack appears as the last **P**, then the type **P** is matched against the type **A** of each remaining argument of the call. Each match deduces the template arguments for the next position in the pack expansion:

```
template<class... Types> void f(Types&...);

void h(int x, float& y)
{
    const int z = x;
    f(x, y, z); // P = Types&..., A1 = x: deduced first member of Types... = int
                // P = Types&..., A2 = y: deduced second member of Types... = float
                // P = Types&..., A3 = z: deduced third member of Types... = const int
                // calls f<int, float, const int>
}
```

If **P** is a function type, pointer to function type, or pointer to member function type and if **A** is a set of overloaded functions not containing function templates, template argument deduction is attempted with each overload. If only one succeeds, that successful deduction is used. If none or more than one succeeds, the template parameter is non-deduced context (see below):

```
template<class T> int f(T(*p)(T));
int g(int);
int g(char);
f(g); // P = T(*)(T), A = overload set
      // P = T(*)(T), A1 = int(int): deduced T = int
      // P = T(*)(T), A2 = int(char): fails to deduce T
      // only one overload works, deduction succeeds
```

Before deduction begins, the following adjustments to **P** and **A** are made:

1) If **P** is not a reference type,
   a) if **A** is an array type, **A** is replaced by the pointer type obtained from array-to-pointer conversion;
   b) otherwise, if **A** is a function type, **A** is replaced by the pointer type obtained from function-to-pointer conversion;
   c) otherwise, if **A** is a cv-qualified type, the top-level cv-qualifiers are ignored for deduction:

```
template<class T> void f(T);
int a[3];
f(a); // P = T, A = int[3], adjusted to int*: deduced T = int*
const int b = 13;
f(b); // P = T, A = const int, adjusted to int: deduced T = int
void g(int);
f(g); // P = T, A = void(int), adjusted to void(*)(int): deduced T = void(*)(int)
```

2) If **P** is a cv-qualified type, the top-level cv-qualifiers are ignored for deduction.

3) If **P** is a reference type, the type referred to by **P** is used for deduction.

4) If **P** is an rvalue reference to a cv-unqualified template parameter (so-called "forwarding reference"), and the corresponding function call argument is an lvalue, the type lvalue reference to **A** is used in place of **A** for deduction (Note: this is the basis for the action of `std::forward`):

```
template<class T>
int f(T&&);       // P is an rvalue reference to cv-unqualified T (forwarding reference)
template<class T>
int g(const T&&); // P is an rvalue reference to cv-qualified T (not special)

int main()
{
    int i;
    int n1 = f(i); // argument is lvalue: calls f<int&>(int&) (special case)
    int n2 = f(0); // argument is not lvalue: calls f<int>(int&&)

//  int n3 = g(i); // error: deduces to g<int>(const int&&), which
                   // cannot bind an rvalue reference to an lvalue
}
```

After these transformations, the deduction processes as described below (cf. section "Deduction from type") and attempts to find such template arguments that would make the deduced **A** (that is, **P** after adjustments listed above and the substitution of the deduced template parameters) identical to the *transformed* **A**, that is **A** after the adjustments listed above.

If the usual deduction from **P** and **A** fails, the following alternatives are additionally considered:

1) If **P** is a reference type, the deduced **A** (i.e., the type referred to by the reference) can be more cv-qualified than the transformed **A**:

```
template<typename T> void f(const T& t);
bool a = false;
f(a); // P = const T&, adjusted to const T, A = bool:
      // deduced T = bool, deduced A = const bool
      // deduced A is more cv-qualified than A
```

2) The transformed **A** can be another pointer or pointer to member type that can be converted to the deduced **A** via a qualification conversions or a function pointer conversion (since C++17):

```
template<typename T> void f(const T*);
int* p;
f(p); // P = const T*, A = int*:
      // deduced T = int, deduced A = const int*
      // qualification conversion applies (from int* to const int*)
```

3) If **P** is a class and **P** has the form *simple-template-id*, then the transformed **A** can be a derived class of the deduced **A**. Likewise, if **P** is a pointer to a class of the form *simple-template-id*, the transformed **A** can be a pointer to a derived class pointed to by the deduced **A**:

```
template<class T> struct B { };
template<class T> struct D : public B<T> { };
template<class T> void f(B<T>&) { }

void f()
{
    D<int> d;
    f(d); // P = B<T>&, adjusted to P = B<T> (a simple-template-id), A = D<int>:
          // deduced T = int, deduced A = B<int>
          // A is derived from deduced A
}
```

Non-deduced contexts

In the following cases, the types, templates, and non-type values that are used to compose **P** do not participate in template argument deduction, but instead *use* the template arguments that were either deduced elsewhere or explicitly specified. If a template parameter is used only in non-deduced contexts and is not explicitly specified, template argument deduction fails.

1) The *nested-name-specifier* (everything to the left of the scope resolution operator `::`) of a type that was specified using a qualified-id:

```
// the identity template, often used to exclude specific arguments from deduction
template<typename T> struct identity { typedef T type; };
template<typename T> void bad(std::vector<T> x, T value = 1);
template<typename T> void good(std::vector<T> x, typename identity<T>::type value = 1);
std::vector<std::complex<double>> x;
bad(x, 1.2);  // P1 = std::vector<T>, A1 = std::vector<std::complex<double>>
              // P1/A1: deduced T = std::complex<double>
              // P2 = T, A2 = double
              // P2/A2: deduced T = double
              // error: deduction fails, T is ambiguous
good(x, 1.2); // P1 = std::vector<T>, A1 = std::vector<std::complex<double>>
              // P1/A1: deduced T = std::complex<double>
              // P2 = identity<T>::type, A2 = double
              // P2/A2: uses T deduced by P1/A1 because T is to the left of :: in P2
              // OK: T = std::complex<double>
```

2) The expression of a decltype-specifier:

```
template<typename T> void f(decltype(*std::declval<T>()) arg);
int n;
f<int*>(n); // P = decltype(*declval<T>()), A = int: T is in non-deduced context
```

(since C++14)

3) A non-type template argument or an array bound in which a subexpression references a template parameter:

```
template<std::size_t N> void f(std::array<int, 2 * N> a);
std::array<int, 10> a;
f(a); // P = std::array<int, 2 * N>, A = std::array<int, 10>:
```

```
      // 2 * N is non-deduced context, N cannot be deduced
      // note: f(std::array<int, N> a) would be able to deduce N
```

4) A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done:

```
template<typename T, typename F>
void f(const std::vector<T>& v, const F& comp = std::less<T>());
std::vector<std::string> v(3);
f(v); // P1 = const std::vector<T>&, A1 = std::vector<std::string> lvalue
      // P1/A1 deduced T = std::string
      // P2 = const F&, A2 = std::less<std::string> rvalue
      // P2 is non-deduced context for F (template parameter) used in the
      // parameter type (const F&) of the function parameter comp,
      // that has a default argument that is being used in the call f(v)
```

5) The parameter **P**, whose **A** is a function or a set of overloads such that more than one function matches **P** or no function matches **P** or the set of overloads includes one or more function templates:

```
template<typename T> void out(const T& value) { std::cout << value; }
out("123");     // P = const T&, A = const char[4] lvalue: deduced T = char[4]
out(std::endl); // P = const T&, A = function template: T is in non-deduced context
```

6) The parameter **P**, whose **A** is a braced-init-list, but **P** is not std::initializer_list or a reference to one:

```
template<class T> void g1(std::vector<T>);
template<class T> void g2(std::vector<T>, T x);
g1({1, 2, 3});     // P = std::vector<T>, A = {1, 2, 3}: T is in non-deduced context
                   // error: T is not explicitly specified or deduced from another P/A
g2({1, 2, 3}, 10); // P1 = std::vector<T,> A1 = {1, 2, 3}: T is in non-deduced context
                   // P2 = T, A2 = int: deduced T = int
```

7) The parameter **P** which is a parameter pack and does not occur at the end of the parameter list:

```
template<class... Ts, class T> void f1(T n, Ts... args);
template<class... Ts, class T> void f2(Ts... args, T n);
f1(1, 2, 3, 4); // P1 = T, A1 = 1: deduced T = int
                // P2 = Ts..., A2 = 2, A3 = 3, A4 = 4: deduced Ts = [int, int, int]
f2(1, 2, 3, 4); // P1 = Ts...: Ts is non-deduced context
```

8) The template parameter list that appears within the parameter **P**, and which includes a pack expansion that is not at the very end of the template parameter list:

```
template<int...> struct T { };

template<int... Ts1, int N, int... Ts2>
void good(const T<N, Ts1...>& arg1, const T<N, Ts2...>&);

template<int... Ts1, int N, int... Ts2>
void bad(const T<Ts1..., N>& arg1, const T<Ts2..., N>&);

T<1, 2> t1;
T<1, -1, 0> t2;
good(t1, t2); // P1 = const T<N, Ts1...>&, A1 = T<1, 2>:
              // deduced N = 1, deduced Ts1 = [2]
              // P2 = const T<N, Ts2...>&, A2 = T<1, -1, 0>:
              // deduced N = 1, deduced Ts2 = [-1, 0]
bad(t1, t2);  // P1 = const T<Ts1..., N>&, A1 = T<1, 2>:
              // <Ts1..., N> is non-deduced context
              // P2 = const T<Ts2..., N>&, A2 = T<1, -1, 0>:
              // <Ts2..., N> is non-deduced context
```

9) For **P** of array type (but not reference to array or pointer to array), the major array bound:

```
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);    // P = int[i][20], array type
template<int i> void f3(int (&a)[i][20]); // P = int(&)[i][20], reference to array

void g()
{
    int a[10][20];
    f1(a);     // OK: deduced i = 20
    f1<20>(a); // OK
    f2(a);     // error: i is non-deduced context
    f2<10>(a); // OK
    f3(a);     // OK: deduced i = 10
    f3<10>(a); // OK
}
```

In any case, if any part of a type name is non-deduced, the entire type name is non-deduced context. However, compound types can include both deduced and non-deduced type names. For example, in `A<T>::B<T2>`, T is non-deduced because of rule #1 (nested name specifier), and T2 is non-deduced because it is part of the same type name, but in `void(*f)(typename A<T>::B, A<T>)`, the T in `A<T>::B` is non-deduced (because of the same rule), while the T in `A<T>` is deduced.

Deduction from a type

Given a function parameter **P** that depends on one or more type template parameters **T**i, template template parameters **TT**i, or non-type template parameters **I**i, and the corresponding argument **A**, deduction takes place if **P** has one of the following forms:

This section is incomplete
Reason: possibly a table with micro-examples

- `T`;
- `cv-list T`;
- `T*`;
- `T&`;
- `T&&`;
- `T[integer-constant]`;
- `class-template-name<T>`;
- `type(T)`;
- `T()`;
- `T(T)`;
- `T type::*`;
- `type T::*`;
- `T T::*`;
- `T(type::*)()`;

- `type(T::*)();`
- `type(type::*)(T);`
- `type(T::*)(T);`
- `T (type::*)(T);`
- `T (T::*)();`
- `T (T::*)(T);`
- `type[i];`
- `class-template-name<I>;`
- `TT<T>;`
- `TT<I>;`
- `TT<>;`

where

- `(T)` is a function parameter type list where at least one parameter type contains T;
- `()` is a function parameter type list where no parameters contain T;
- `<T>` is a template argument list where at least one argument contains T;
- `<I>` is a template argument list where at least one argument contains I;
- `<>` is a template argument list where no arguments contain T or I.

If **P** has one of the forms that include a template parameter list **<T>** or **<I>**, then each element **P**i of that template argument list is matched against the corresponding template argument **A**i of its **A**. If the last **P**i is a pack expansion, then its pattern is compared against each remaining argument in the template argument list of **A**. A trailing parameter pack that is not otherwise deduced, is deduced to an empty parameter pack.

If **P** has one of the forms that include a function parameter list **(T)**, then each parameter **P**i from that list is compared with the corresponding argument **A**i from **A**'s function parameter list. If the last **P**i is a pack expansion, then its declarator is compared with each remaining **A**i in the parameter type list of **A**.

Forms can be nested and processed recursively: `X<int>(*)(char[6])` is an example of `type(*)(T)`, where *type* is `class-template-name<T>` and T is `type[i]`.

Template type argument cannot be deduced from the type of a non-type template argument:

```
template<typename T, T i> void f(double a[10][i]);
double v[10][20];
f(v); // P = double[10][i], A = double[10][20]:
      // i can be deduced to equal 20
      // but T cannot be deduced from the type of i
```

If a non-type template parameter is used in the parameter list, and the corresponding template argument is deduced, the type of the deduced template argument (as specified in its enclosing template parameter list, meaning references are preserved) (since C++14) must match the type of the non-type template parameter exactly, except that cv-qualifiers are dropped, and except where the template argument is deduced from an array bound—in that case any integral type is allowed, even bool though it would always become true:

```
template<int i> class A { };
template<short s> void f(A<s>); // the type of the non-type template param is short

void k1()
{
    A<1> a;  // the type of the non-type template param of a is int
    f(a);    // P = A<(short)s>, A = A<(int)1>
             // error: deduced non-type template argument does not have the same
             // type as its corresponding template argument
    f<1>(a); // OK: the template argument is not deduced,
             // this calls f<(short)1>(A<(short)1>)
}

template<int&> struct X;
template<int& R> void k2(X<R>&);
int n;
void g(X<n> &x) {
    k2(x); // P = X<R>, A = X<n>
           // parameter type is int&
           // argument type is int& in struct X's template declaration
           // OK since C++14 with CWG 2091: deduces R to refer to n
}
```

Type template parameter cannot be deduced from the type of a function default argument:

```
template<typename T> void f(T = 5, T = 7);

void g()
{
    f(1);     // OK: calls f<int>(1, 7)
    f();      // error: cannot deduce T
    f<int>(); // OK: calls f<int>(5, 7)
}
```

Deduction of template template parameter can use the type used in the template specialization used in the function call:

```
template<template<typename> class X> struct A { }; // A is a template with a TT param
template<template<typename> class TT> void f(A<TT>) { }
template<class T> struct B { };
A<B> ab;
f(ab); // P = A<TT>, A = A<B>: deduced TT = B, calls f(A<B>)
```

Other contexts

Besides function calls and operator expressions, template argument deduction is used in the following situations:

auto type deduction

Template argument deduction is used in declarations of variables, when deducing the meaning of the **auto specifier** from the variable's initializer.

The parameter **P** is obtained as follows: in T, the declared type of the variable that includes `auto`, every occurrence of `auto` is replaced with an imaginary type template parameter U or, if the initializer is a brace-init-list, with `std::initializer_list<U>`. The argument **A** is the initializer expression. After deduction of U from **P** and **A** following the rules described above, the deduced U is substituted into T to get the actual variable type:

```
const auto& x = 1 + 2; // P = const U&, A = 1 + 2:
                       // same rules as for calling f(1 + 2) where f is
                       // template<class U> void f(const U& u)
                       // deduced U = int, the type of x is const int&
auto l = {13}; // P = std::initializer_list<U>, A = {13}:
               // deduced U = int, the type of l is std::initializer_list<int>
```

In direct-list-initialization (but not in copy-list-initalization), when deducing the meaning of the auto from a braced-init-list, the braced-init-list must contain only one element, and the type of auto will be the type of that element: (since C++17)

```
auto x1 = {3}; // x1 is std::initializer_list<int>
auto x2{1, 2}; // error: not a single element
auto x3{3};    // x3 is int (before C++17 it was std::initializer_list<int>)
```

auto-returning functions

Template argument deduction is used in declarations of functions, when deducing the meaning of the auto specifier in the function's return type, from the return statement.

For auto-returning functions, the parameter **P** is obtained as follows: in T, the declared return type of the function that includes `auto`, every occurrence of `auto` is replaced with an imaginary type template parameter U. The argument **A** is the expression of the return statement, and if the return statement has no operand, **A** is `void()` . After deduction of U from **P** and **A** following the rules described above, the deduced U is substituted into T to get the actual return type:

(since C++14)

```
auto f() { return 42; } // P = auto, A = 42:
                        // deduced U = int, the return type of f is int
```

If such function has multiple return statements, the deduction is performed for each return statement. All the resulting types must be the same and become the actual return type.

If such function has no return statement, **A** is `void()` when deducing.

Note: the meaning of `decltype(auto)` placeholder in variable and function declarations does not use template argument deduction.

Overload resolution

Template argument deduction is used during overload resolution, when generating specializations from a candidate template function. **P** and **A** are the same as in a regular function call:

```
std::string s;
std::getline(std::cin, s); // "std::getline" names 4 function templates,
// 2 of which are candidate functions (correct number of parameters)
// 1st candidate template:
// P1 = std::basic_istream<CharT, Traits>&, A1 = std::cin
// P2 = std::basic_string<CharT, Traits, Allocator>&, A2 = s
// deduction determines the type template parameters CharT, Traits, and Allocator
// specialization std::getline<char, std::char_traits<char>, std::allocator<char>>
// 2nd candidate template:
// P1 = std::basic_istream<CharT, Traits>&&, A1 = std::cin
// P2 = std::basic_string<CharT, Traits, Allocator>&, A2 = s
// deduction determines the type template parameters CharT, Traits, and Allocator
// specialization std::getline<char, std::char_traits<char>, std::allocator<char>>
// overload resolution ranks reference binding from lvalue std::cin and picks
// the first of the two candidate specializations
```

If deduction fails, or if deduction succeeds, but the specialization it produces would be invalid (for example, an overloaded operator whose parameters are neither class nor enumeration types), (since C++14) the specialization is not included in the overload set, similar to SFINAE.

Address of an overload set

Template argument deduction is used when taking an address of a overload set, which includes function templates.

The function type of the function template is **P**. The target type is the type of **A**:

```
std::cout << std::endl; // std::endl names a function template
// type of endl P =
// std::basic_ostream<CharT, Traits>& (std::basic_ostream<CharT, Traits>&)
// operator<< parameter A =
// std::basic_ostream<char, std::char_traits<char>>& (*)(
//   std::basic_ostream<char, std::char_traits<char>>&
// )
// (other overloads of operator<< are not viable)
// deduction determines the type template parameters CharT and Traits
```

An additional rule is applied to the deduction in this case: when comparing function parameters **P**i and **A**i, if any **P**i is an rvalue reference to cv-unqualified template parameter (a "forwarding reference") and the corresponding **A**i is an lvalue reference, then **P**i is adjusted to the template parameter type (T&& becomes T).

If the return type of the function template is a placeholder (`auto` or `decltype(auto)`), that return type is a non-deduced context and is determined from the instantiation. (since C++14)

Partial ordering

Template argument deduction is used during partial ordering of overloaded function templates

This section is incomplete
Reason: mini-example

Conversion function template

Template argument deduction is used when selecting user-defined conversion function template arguments.

A is the type that is required as the result of the conversion. **P** is the return type of the conversion function template, except that

a) if the return type is a reference type then **P** is the referred type;
b) if the return type is an array type and **A** is not a reference type, then **P** is the pointer type obtained by array-to-pointer conversion;
c) if the return type is a function type and **A** is not a reference type, then **P** is the function pointer type obtained by function-to-pointer conversion;
d) if **P** is cv-qualified, the top-level cv-qualifiers are ignored.

If **A** is cv-qualified, the top-level cv-qualifiers are ignored. If **A** is a reference type, the referred type is used by deduction.

If the usual deduction from **P** and **A** (as described above) fails, the following alternatives are additionally considered:

a) if **A** is a reference type, **A** can be more cv-qualified than the deduced **A**;
b) if **A** is a pointer or pointer to member type, the deduced **A** is allowed to be any pointer that can be converted to **A** by qualification conversion:

```
struct A { template<class T> operator T***(); };
A a;
const int* const* const* p1 = a; // P = T***, A = const int* const* const*
// regular function-call deduction for
// template<class T> void f(T*** p) as if called with the argument
// of type const int* const* const* fails
// additional deduction for conversion functions determines T = int
// (deduced A is int***, convertible to const int* const* const*)
```

c) if **A** is a function pointer type, the deduced **A** is allowed to be pointer to noexcept function, convertible to **A** by function pointer conversion;
d) if **A** is a pointer to member function, the deduced **A** is allowed to be a pointer to noexcept member function, convertible to **A** by function pointer conversion. (since C++17)

See member template for other rules regarding conversion function templates.

Explicit instantiation

Template argument deduction is used in explicit instantiations, explicit specializations, and those friend declarations where the declarator-id happens to refer to a specialization of a function template (for example, `friend ostream& operator<< <> (...)`), if not all template arguments are explicitly specified or defaulted, template argument deduction is used to determine which template's specialization is referred to.

P is the type of the function template that is being considered as a potential match, and **A** is the function type from the declaration. If there are no matches or more than one match (after partial ordering), the function declaration is ill-formed:

```
template<class X> void f(X a);  // 1st template f
template<class X> void f(X* a); // 2nd template f
template<> void f<>(int* a) { } // explicit specialization of f
// P1 = void(X), A1 = void(int*): deduced X = int*, f<int*>(int*)
// P2 = void(X*), A2 = void(int*): deduced X = int, f<int>(int*)
// f<int*>(int*) and f<int>(int*) are then submitted to partial ordering
// which selects f<int>(int*) as the more specialized template
```

An additional rule is applied to the deduction in this case: when comparing function parameters **P**i and **A**i, if any **P**i is an rvalue reference to cv-unqualified template parameter (a "forwarding reference") and the corresponding **A**i is an lvalue reference, then **P**i is adjusted to the template parameter type (T&& becomes T).

Deallocation function template

Template argument deduction is used when determining if a deallocation function template specialization matches a given placement form of `operator new`.

P is the type of the function template that is being considered as a potential match, and **A** is the function type of the deallocation function that would be the match for the placement operator new under consideration. If there is no match or more than one match (after overload resolution), the placement deallocation function is not called (memory leak may occur):

```
struct X
{
    X() { throw std::runtime_error(""); }
    static void* operator new(std::size_t sz, bool b) { return ::operator new(sz); }
    static void* operator new(std::size_t sz, double f) { return ::operator new(sz); }
    template<typename T> static void operator delete(void* ptr, T arg)
    {
        ::operator delete(ptr);
    }
};

int main()
{
    try
    {
        X* p1 = new (true) X; // when X() throws, operator delete is looked up
                              // P1 = void(void*, T), A1 = void(void*, bool):
                              // deduced T = bool
                              // P2 = void(void*, T), A2 = void(void*, double):
                              // deduced T = double
                              // overload resolution picks operator delete<bool>
    } catch(const std::exception&) { }
    try
    {
        X* p1 = new (13.2) X; // same lookup, picks operator delete<double>
    } catch(const std::exception&) { }
}
```

Alias templates

Alias templates are never deduced:

```
template<class T> struct Alloc { };
template<class T> using Vec = vector<T, Alloc<T>>;
Vec<int> v;

template<template<class, class> class TT> void g(TT<int, Alloc<int>>);
g(v); // OK: deduced TT = vector

template<template<class> class TT> void f(TT<int>);
f(v); // error: TT cannot be deduced as "Vec" because Vec is an alias template
```

Implicit conversions

Type deduction does not consider implicit conversions (other than type adjustments listed above): that's the job for overload resolution, which happens later.

However, if deduction succeeds for all parameters that participate in template argument deduction, and all template arguments that aren't deduced are explicitly specified or defaulted, then the remaining function parameters are compared with the corresponding function arguments. For each remaining parameter **P** with a type that was non-dependent before substitution of any explicitly-specified template arguments, if the corresponding argument **A** cannot be implicitly converted to **P**, deduction fails.

Parameters with dependent types in which no template-parameters participate in template argument deduction, and parameters that became non-dependent due to substitution of explicitly-specified template arguments will be checked during overload resolution:

```
template<class T> struct Z { typedef typename T::x xx; };
template<class T> typename Z<T>::xx f(void*, T); // #1
template<class T> void f(int, T);                // #2
struct A { } a;

int main()
{
    f(1, a); // for #1, deduction determines T = struct A, but the remaining argument 1
             // cannot be implicitly converted to its parameter void*: deduction fails
             // instantiation of the return type is not requested
             // for #2, deduction determines T = struct A, and the remaining argument 1
             // can be implicitly converted to its parameter int: deduction succeeds
             // the function call compiles as a call to #2 (deduction failure is SFINAE)
}
```

(since C++14)

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 1391 (http://open-std.org/JTC1/SC22/WG21/docs/cwg_defects.html#1391)	C++14	effect of implicit conversions of the arguments that aren't involved in deduction were not specified	specified as described above
CWG 2052 (http://open-std.org/JTC1/SC22/WG21/docs/cwg_defects.html#2052)	C++14	deducing an operator with non-class arguments was a hard error (in some compilers)	soft error if there are other overloads
CWG 2091 (http://open-std.org/JTC1/SC22/WG21/docs/cwg_defects.html#2091)	C++14	deducing a reference non-type parameter did not work due to type mismatch against the argument	type mismatch avoided